

using springboot 4 and jpa how can implement i18n at database level for lookup entities? Can I intercept joins and add extra locale parameter on join?

Short answer: **you cannot reliably “intercept joins” in JPA and inject a locale parameter.** JPQL/Criteria joins are compiled before execution, and Hibernate does not expose a safe hook to rewrite joins per request.

But you **can implement clean, production-grade DB-level i18n for lookup entities** in Spring Boot + JPA using a few well-known patterns. Below I'll walk you through **what works, what does not, and recommended architectures.**

✖ Why intercepting joins won't work

In JPA/Hibernate:

- JPQL / Criteria queries are parsed and compiled before execution
- Joins are static in the query plan
- Hibernate interceptors (`Interceptor`, `StatementInspector`) operate **after** SQL generation and are:
 - DB-dialect dependent
 - fragile
 - unsafe for parameter binding

You *can* technically rewrite SQL using `StatementInspector`, but:

- you cannot safely inject parameters
- query plan caching breaks
- it will fail on pagination, count queries, and joins

→ This is not recommended for i18n.

✓ Recommended DB-level i18n patterns

Pattern 1 – Translation table (BEST PRACTICE)

This is the **standard and most flexible solution**.

Schema

```
sql

lookup_type
-----
id
code

lookup_type_i18n
-----
id
lookup_type_id
locale
name
description
```

 Copy code

Entity mapping

```
java

@Entity
public class LookupType {
    @Id
    private Long id;

    private String code;

    @OneToMany(mappedBy = "lookupType", fetch = FetchType.LAZY)
    private Set<LookupTypeI18n> translations;
}
```

 Copy code

```
java

@Entity
@Table(
    uniqueConstraints = @UniqueConstraint(
        columnNames = {"lookup_type_id", "locale"})
```

```
)  
)  
public class LookupTypeI18n {  
  
    @ManyToOne  
    @JoinColumn(name = "lookup_type_id")  
    private LookupType lookupType;  
  
    private String locale;  
    private String name;  
}
```

 Copy code

Query with locale (explicit and safe)

```
java
```

```
@Query("""  
select lt, i.name  
from LookupType lt  
join LookupTypeI18n i  
on i.lookupType = lt  
and i.locale = :locale  
""")  
List<Object[]> findAllLocalized(@Param("locale") String locale);
```

 Copy code

- ✓ Safe
 - ✓ Cacheable
 - ✓ Works with pagination
 - ✓ Works with second-level cache
-

Pattern 2 – Hibernate @Filter (automatic locale injection)

If you want **transparent locale handling** without changing every query:

Entity

```
java
```

```
@FilterDef(  
    name = "localeFilter",  
    parameters = @ParamDef(name = "locale", type = String.class)  
)  
  
@Filter(  
    name = "localeFilter",  
    condition = "locale = :locale"  
)  
  
@Entity  
public class LookupTypeI18n {  
    ...  
}
```

 Copy code

Enable filter per request

java

```
Session session = entityManager.unwrap(Session.class);  
session.enableFilter("localeFilter")  
    .setParameter("locale", LocaleContextHolder.getLocale().toString());
```

 Copy code

Now all joins automatically respect locale.

- ✓ No SQL rewriting
- ✓ Clean separation
- ✓ Supported by Hibernate
- ⚠ Hibernate-specific (not pure JPA)

Pattern 3 – Map translations by locale

Useful if lookup tables are small.

java

```
@OneToMany(mappedBy = "lookupType")  
@MapKey(name = "locale")  
private Map<String, LookupTypeI18n> translations;
```

 Copy code

Usage:

```
java
```

```
lookup.getTranslations().get(locale).getName();
```

 Copy code

✓ Simple

⚠ Fetches all locales unless filtered

⚠ Not ideal for large datasets

✖ Patterns to avoid

Approach	Why
SQL rewriting via <code>StatementInspector</code>	Unsafe, brittle
Intercepting joins dynamically	Not supported
One column per language	Schema explosion
JSON translations in column	Poor indexing, no joins

🧠 Advanced option: View-based localization

For read-only lookups:

```
sql
```

```
CREATE VIEW lookup_type_v AS
SELECT l.id, l.code, i.name
FROM lookup_type l
JOIN lookup_type_i18n i
ON i.lookup_type_id = l.id
AND i.locale = current_setting('app.locale');
```

 Copy code

Set locale per connection:

```
sql
```

```
SET app.locale = 'en';
```

 Copy code

- ✓ Very fast
 - ✓ Clean entities
 - ⚠ DB-specific (Postgres best)
-

🏆 Recommended choice

If you want correctness + maintainability:

1. Translation table
2. Hibernate `@Filter` for locale
3. Explicit joins for critical queries

This is exactly how large Spring + Hibernate systems implement DB-level i18n.

If you want, I can:

- show a full Spring Boot example
- integrate this with second-level cache
- explain fallback locale logic (en → default)
- show query performance comparisons

Just tell me 



show a full Spring Boot example with Hibernate `@Filter`

Below is a complete, minimal, working Spring Boot + JPA + Hibernate `@Filter` i18n example.

This shows database-level localization for lookup entities with automatic locale injection per request.

I'll use Spring Boot 4, Hibernate 6, H2 (for demo), and REST.

1 Database schema

sql

```
CREATE TABLE lookup_type (
    id BIGINT PRIMARY KEY,
    code VARCHAR(50) NOT NULL
);

CREATE TABLE lookup_type_i18n (
    id BIGINT PRIMARY KEY,
    lookup_type_id BIGINT NOT NULL,
    locale VARCHAR(10) NOT NULL,
    name VARCHAR(100) NOT NULL,
    CONSTRAINT fk_lookup FOREIGN KEY (lookup_type_id)
        REFERENCES lookup_type(id),
    CONSTRAINT uq_lookup_locale
        UNIQUE (lookup_type_id, locale)
);
```

 Copy code

2 Entities with Hibernate @Filter

LookupType

java

```
@Entity
@Table(name = "lookup_type")
public class LookupType {

    @Id
    private Long id;

    private String code;

    @OneToMany(mappedBy = "lookupType", fetch = FetchType.LAZY)
    private Set<LookupTypeI18n> translations = new HashSet<>();
```

```
// convenience method
public String getName() {
    return translations.stream()
        .findFirst()
        .map(LookupTypeI18n::getName)
        .orElse(null);
}

// getters/setters
}
```

Copy code

LookupTypeI18n

```
java

@Entity
@Table(name = "lookup_type_i18n")
@FilterDef(
    name = "localeFilter",
    parameters = @ParamDef(name = "locale", type = String.class)
)
@Filter(
    name = "localeFilter",
    condition = "locale = :locale"
)
public class LookupTypeI18n {

    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "lookup_type_id")
    private LookupType lookupType;

    private String locale;
    private String name;
```

```
// getters/setters  
}
```

 Copy code

- ✓ Hibernate automatically appends `locale = ?` to joins
- ✓ No query modification required

3 Repository

java

```
public interface LookupTypeRepository  
    extends JpaRepository<LookupType, Long> {  
}
```

 Copy code

No custom JPQL needed 

4 Enable the filter per request

Locale Filter Activator (IMPORTANT)

java

```
@Component  
@RequiredArgsConstructor  
public class HibernateLocaleFilterEnabler {  
  
    private final EntityManager entityManager;  
  
    public void enable(Locale locale) {  
        Session session = entityManager.unwrap(Session.class);  
        session.enableFilter("localeFilter")  
            .setParameter("locale", locale.toLanguageTag());  
    }  
}
```

 Copy code

5 REST Controller

```
java
```

```
@RestController  
@RequiredArgsConstructor  
@RequestMapping("/lookups")  
public class LookupController {  
  
    private final LookupTypeRepository repository;  
    private final HibernateLocaleFilterEnabler filterEnabler;  
  
    @GetMapping  
    @Transactional(readOnly = true)  
    public List<Map<String, String>> getLookups(Locale locale) {  
  
        filterEnabler.enable(locale);  
  
        return repository.findAll()  
            .stream()  
            .map(l -> Map.of(  
                "code", l.getCode(),  
                "name", l.getName())  
            )  
            .toList();  
    }  
}
```

 Copy code

- ✓ Locale comes from `Accept-Language` header
- ✓ Filter is applied inside transaction
- ✓ One locale row per lookup is loaded

6 Sample data

```
sql
```

```
INSERT INTO lookup_type VALUES (1, 'STATUS_ACTIVE');  
INSERT INTO lookup_type VALUES (2, 'STATUS_PASSIVE');  
  
INSERT INTO lookup_type_i18n VALUES (1, 1, 'en', 'Active');  
INSERT INTO lookup_type_i18n VALUES (2, 1, 'tr', 'Aktif');
```

```
INSERT INTO lookup_type_i18n VALUES (3, 2, 'en', 'Passive');  
INSERT INTO lookup_type_i18n VALUES (4, 2, 'tr', 'Pasif');
```

 Copy code

7 Test the API

English

bash

```
curl -H "Accept-Language: en" http://localhost:8080/lookups
```

 Copy code

json

```
[  
  { "code": "STATUS_ACTIVE", "name": "Active" },  
  { "code": "STATUS_PASSIVE", "name": "Passive" }  
]
```

 Copy code

Turkish

bash

```
curl -H "Accept-Language: tr" http://localhost:8080/lookups
```

 Copy code

json

```
[  
  { "code": "STATUS_ACTIVE", "name": "Aktif" },  
  { "code": "STATUS_PASSIVE", "name": "Pasif" }  
]
```

 Copy code

8 Important rules (avoid common bugs)

 Must enable filter:

Rule	Why
Inside transaction	Filter only applies to session
Before entity loading	Cached results won't re-filter
Once per request	Thread-safe

9 Optional: fallback locale support

Add a second filter or union query if translation is missing:

- Try requested locale
- Fall back to `en`

I can show this if needed.

🏆 Summary

- ✓ No join interception
- ✓ No SQL rewriting

— ChatGPT ▾ [+ Upgrade to Go](#) ✕

↑ ⚡ ...

▼ Clear API

- ✓ Works with pagination & cache

If you want, next I can:

- add 2nd-level cache support
- show fallback locale
- compare @Filter vs JPQL performance
- show multi-tenant + locale combined filters

Just tell me 🤗

🔗 👍 💬 ⌂ ⌃ ...

Enhance this project with adding more complex internationalized entities. Entities with child entities that has internationalized lookups

Great question — this is where `@Filter` really shines 

Below I'll extend the previous project to a realistic, hierarchical domain model:

Parent entity → child entities → child has i18n lookups (also filtered by locale)

We'll build this step-by-step and highlight important Hibernate behaviors you must understand to avoid subtle bugs.



Target Domain (Realistic Example)

Order → OrderItem → ProductCategory (lookup, i18n)

SCSS

 Copy code

Order

 └─ OrderItem

 └─ ProductCategory

 └─ ProductCategoryI18n (locale-filtered)

- ✓ Parent entity
- ✓ Child collection
- ✓ Child references internationalized lookup
- ✓ Single locale automatically applied everywhere

1 Database Schema

sql

 Copy code

```
CREATE TABLE orders (
    id BIGINT PRIMARY KEY,
```

```

        order_no VARCHAR(50)
    );

CREATE TABLE order_item (
    id BIGINT PRIMARY KEY,
    order_id BIGINT,
    category_id BIGINT,
    quantity INT,

    CONSTRAINT fk_order
        FOREIGN KEY (order_id) REFERENCES orders(id)
);

CREATE TABLE product_category (
    id BIGINT PRIMARY KEY,
    code VARCHAR(50)
);

CREATE TABLE product_category_i18n (
    id BIGINT PRIMARY KEY,
    category_id BIGINT,
    locale VARCHAR(10),
    name VARCHAR(100),

    CONSTRAINT fk_category
        FOREIGN KEY (category_id) REFERENCES product_category(id),
    CONSTRAINT uq_category_locale
        UNIQUE (category_id, locale)
);

```

2 Hibernate Filter (reused globally)

We reuse the **same locale filter name** everywhere.

java

 Copy code

```
@FilterDef(
    name = "localeFilter",
```

```
    parameters = @ParamDef(name = "locale", type = String.class)
)
```

3 ProductCategory (Lookup)

java

 Copy code

```
@Entity
@Table(name = "product_category")
public class ProductCategory {

    @Id
    private Long id;

    private String code;

    @OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
    private Set<ProductCategoryI18n> translations = new HashSet<>();

    public String getName() {
        return translations.stream()
            .findFirst()
            .map(ProductCategoryI18n::getName)
            .orElse(null);
    }
}
```

4 ProductCategoryI18n (Filtered)

java

 Copy code

```
@Entity
@Table(name = "product_category_i18n")
@Filter(name = "localeFilter", condition = "locale = :locale")
public class ProductCategoryI18n {

    @Id
```

```
private Long id;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "category_id")
private ProductCategory category;

private String locale;
private String name;
}
```

🔥 Important

The filter applies **only to this table**, but affects **all joins** to it.

5 OrderItem (Child entity)

java

 Copy code

```
@Entity
@Table(name = "order_item")
public class OrderItem {

    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "order_id")
    private Order order;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    private ProductCategory category;

    private int quantity;
}
```

6 Order (Root aggregate)

java

 Copy code

```
@Entity
@Table(name = "orders")
public class Order {

    @Id
    private Long id;

    private String orderNo;

    @OneToMany(mappedBy = "order", fetch = FetchType.LAZY)
    private List<OrderItem> items = new ArrayList<>();
}
```

7 Repository

java

 Copy code

```
public interface OrderRepository
    extends JpaRepository<Order, Long> {
}
```

8 Locale Filter Enabler (same as before)

java

 Copy code

```
@Component
@RequiredArgsConstructor
public class HibernateLocaleFilterEnabler {

    private final EntityManager entityManager;

    public void enable(Locale locale) {
        Session session = entityManager.unwrap(Session.class);
        session.enableFilter("localeFilter")
            .setParameter("locale", locale.toLanguageTag());
```

```
    }  
}
```

9 REST Controller (deep traversal)

java

 Copy code

```
@RestController  
@RequiredArgsConstructor  
@RequestMapping("/orders")  
public class OrderController {  
  
    private final OrderRepository orderRepository;  
    private final HibernateLocaleFilterEnabler filterEnabler;  
  
    @GetMapping("/{id}")  
    @Transactional(readOnly = true)  
    public Map<String, Object> getOrder(  
        @PathVariable Long id,  
        Locale locale) {  
  
        filterEnabler.enable(locale);  
  
        Order order = orderRepository.findById(id).orElseThrow();  
  
        return Map.of(  
            "orderNo", order.getOrderNo(),  
            "items", order.getItems().stream()  
                .map(item -> Map.of(  
                    "quantity", item.getQuantity(),  
                    "categoryCode", item.getCategory().getCode(),  
                    "categoryName", item.getCategory().getName()  
                ))  
                .toList()  
        );  
    }  
}
```

🔍 What Hibernate actually generates (simplified SQL)

sql

 Copy code

```
select pci.*  
from product_category_i18n pci  
where pci.category_id = ?  
and pci.locale = ?
```

- ✓ Applied automatically
- ✓ No JPQL changes
- ✓ No manual joins

⚠ CRITICAL Rules for Complex Graphs

1 Filter must be enabled before traversal

Lazy loading AFTER filter activation works ✓

Lazy loading BEFORE activation → wrong locale ✗

2 Second-level cache considerations

If you enable 2nd-level cache:

- Cache must be locale-aware
- Use:

java

 Copy code

```
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
```

only on I18n tables, not parent entities

3 Fetch joins + filters

java

 Copy code

```
@Query("""
select o from Order o
join fetch o.items i
join fetch i.category c
""")
```

✓ Works

✓ Filter still applies to `ProductCategoryI18n`

1 0 Sample Response

Accept-Language: tr

json

 Copy code

```
{
  "orderNo": "ORD-1001",
  "items": [
    {
      "quantity": 2,
      "categoryCode": "FOOD",
      "categoryName": "Gıda"
    },
    {
      "quantity": 1,
      "categoryCode": "ELEC",
      "categoryName": "Elektronik"
    }
  ]
}
```

🏆 Architecture Summary

Layer	Responsibility
DB	Holds translations

Layer	Responsibility
Hibernate Filter	Enforces locale
Entity graph	Clean, no locale logic
Controller	Enables filter

This scales cleanly even with:

- nested child entities
- multiple i18n lookups
- pagination
- caching



Want to go further?

I can show you:

- fallback locale (tr → en)
- multi-tenant + locale filters together
- DTO projection optimization
- cache-safe locale separation
- GraphQL-friendly i18n

Just tell me which one you want next 🤗



Enhance project adding more than one entities having
@Filter and enable that filter for all

Perfect — this is exactly the right question once you start scaling i18n beyond a single lookup 👍

Below I'll enhance the project so MULTIPLE entities use the SAME `@Filter`, and we'll enable it once and have it apply everywhere.

I'll also show the **correct, safe way** to enable it **globally per request**, not manually in every controller.

Goal

- ✓ Multiple i18n entities
 - ✓ Same `localeFilter` applied to all
 - ✓ Enabled **once per request**
 - ✓ Works for deep entity graphs
 - ✓ No controller duplication
-

1 Global Locale Filter Definition (ONE place)

Define the filter **once** on a base class or package-level entity.

Option A (recommended): Base I18n superclass

```
java Copy code  
  
@MappedSuperclass  
@FilterDef(  
    name = "localeFilter",  
    parameters = @ParamDef(name = "locale", type = String.class)  
)  
@Filter(name = "localeFilter", condition = "locale = :locale")  
public abstract class AbstractI18nEntity {  
  
    @Column(nullable = false)  
    protected String locale;  
}
```

- ✓ Defined once
 - ✓ Reused everywhere
 - ✓ No duplication
-

2 First i18n entity – ProductCategoryI18n

java

 Copy code

```
@Entity
@Table(name = "product_category_i18n")
public class ProductCategoryI18n extends AbstractI18nEntity {

    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    private ProductCategory category;

    private String name;
}
```

3 Second i18n entity – ProductStatusI18n

java

 Copy code

```
@Entity
@Table(name = "product_status_i18n")
public class ProductStatusI18n extends AbstractI18nEntity {

    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "status_id")
    private ProductStatus status;

    private String label;
}
```

4 Lookup entities (non-i18n)

java

 Copy code

```
@Entity
@Table(name = "product_status")
public class ProductStatus {

    @Id
    private Long id;

    private String code;

    @OneToMany(mappedBy = "status")
    private Set<ProductStatusI18n> translations;

    public String getLabel() {
        return translations.stream()
            .findFirst()
            .map(ProductStatusI18n::getLabel)
            .orElse(null);
    }
}
```

5 Third i18n entity – UIMessageI18n (not a lookup!)

java

 Copy code

```
@Entity
@Table(name = "ui_message_i18n")
public class UIMessageI18n extends AbstractI18nEntity {

    @Id
    private Long id;

    private String messageKey;
    private String text;
}
```

- ✓ Same filter
 - ✓ Different domain
 - ✓ Same locale handling
-

6 Enable filter ONCE per request (GLOBAL)

Hibernate Session filter interceptor

java

 Copy code

```
@Component
@RequiredArgsConstructor
public class HibernateLocaleFilterInterceptor
    implements HandlerInterceptor {

    private final EntityManager entityManager;

    @Override
    public boolean preHandle(
        HttpServletRequest request,
        HttpServletResponse response,
        Object handler) {

        Locale locale = LocaleContextHolder.getLocale();

        Session session = entityManager.unwrap(Session.class);
        session.enableFilter("localeFilter")
            .setParameter("locale", locale.toLanguageTag());

        return true;
    }
}
```

Register interceptor

java

 Copy code

```
@Configuration  
@RequiredArgsConstructor  
public class WebConfig implements WebMvcConfigurer {  
  
    private final HibernateLocaleFilterInterceptor interceptor;  
  
    @Override  
    public void addInterceptors(InterceptorRegistry registry) {  
        registry.addInterceptor(interceptor);  
    }  
}
```

🔥 Now the filter is active for ALL repositories, ALL entities, ALL queries.

7 Repository examples (no locale code)

java

 Copy code

```
public interface ProductRepository  
    extends JpaRepository<Product, Long> {  
}
```

java

 Copy code

```
public interface UiMessageRepository  
    extends JpaRepository<UiMessageI18n, Long> {  
}
```

- ✓ Zero locale logic
- ✓ Zero duplication

8 Controller example (deep graph)

java

 Copy code

```
@GetMapping("/products/{id}")  
@Transactional(readOnly = true)
```

```
public ProductDto getProduct(@PathVariable Long id) {  
    return mapper.toDto(productRepository.findById(id).orElseThrow());  
}
```

No locale handling here — **filter already active**.

9 What happens internally (Hibernate)

If ANY query touches:

- `product_category_i18n`
- `product_status_i18n`
- `ui_message_i18n`

Hibernate automatically appends:

sql

 Copy code

```
and locale = ?
```

using the **same parameter**.

⚠ Important gotchas (REAL production issues)

❗ Filter must be enabled before session use

The interceptor runs before controller & repository.

❗ Async / @TransactionalEventListener

Filters are **NOT inherited** by new sessions.

You must re-enable in async tasks.

❗ Second-level cache

Do **NOT** cache parent entities that join filtered collections.

Correct:

java

 Copy code

```
@Cache(READ_ONLY)  
public class ProductCategoryI18n { }
```

Wrong:

java

 Copy code

```
@Cache  
public class ProductCategory { } // ✗ Locale Leak
```

🏆 Final Architecture

sql

 Copy code

AbstractI18nEntity

```
↑  
├── ProductCategoryI18n  
├── ProductStatusI18n  
└── UiMessageI18n
```

Hibernate **@Filter**

```
↓  
Interceptor (once/request)  
↓  
ALL repositories
```

🚀 Want next?

I can show:

- fallback locale logic (tr → en)
- locale-aware 2nd-level cache
- multi-tenant + locale filters combined

- DTO projections with filtered joins
- GraphQL support

Just say the word 



Is there another way better than this approach?

Short answer: yes — in many real systems there *is* a better approach, and `@Filter` should be used **selectively**, not as the default.

Below is a clear, opinionated comparison based on real Spring + Hibernate production experience, so you can decide *when to keep `@Filter` and when to replace it*.

🔴 Why `@Filter` is often *not* the best long-term solution

Hibernate `@Filter` looks elegant, but it has **hidden costs**:

1 Invisible behavior

Queries return different data depending on thread context.

java

 Copy code

```
repository.findAll(); // Locale-dependent 😬
```

Hard to debug, easy to misuse.

2 Breaks query reasoning & caching

- Second-level cache becomes dangerous
- Query cache must be locale-aware
- Hard to reason about result consistency

3 Session-scoped, not request-scoped

- Async processing
 - Scheduled jobs
 - Batch jobs
- ➡ locale silently disappears unless re-enabled
-

4 Hard to express fallback logic

Example:

arduino

 Copy code

```
tr → en → default
```

With filters this becomes ugly or impossible.

● Better architectural alternatives (ranked)

🏅 BEST PRACTICE: Explicit Translation Joins (DDD-friendly)

Make locale explicit in queries and DTOs.

Why this wins

- ✓ Predictable
 - ✓ Cache-safe
 - ✓ Easy fallback
 - ✓ Database-optimizable
 - ✓ Works with projections
-

Example JPQL (clean & explicit)

java

 Copy code

```

@Query("""
    select new com.app.dto.CategoryDto(
        c.code,
        coalesce(tr.name, en.name)
    )
    from ProductCategory c
    left join ProductCategoryI18n tr
        on tr.category = c and tr.locale = :locale
    left join ProductCategoryI18n en
        on en.category = c and en.locale = 'en'
""")
List<CategoryDto> findAllLocalized(@Param("locale") String locale);

```

- ✓ No magic
 - ✓ Easy fallback
 - ✓ Index-friendly
-

When to choose this

- ✓ Large teams
 - ✓ Long-lived systems
 - ✓ Heavy caching
 - ✓ Performance-critical paths
-

Database Views (Read-Only Lookups)

Perfect for lookup/reference data.

Example (Postgres)

sql

 Copy code

```

CREATE VIEW product_category_v AS
SELECT c.id, c.code, i.name
FROM product_category c
JOIN product_category_i18n i

```

```
ON i.category_id = c.id  
AND i.locale = current_setting('app.locale');
```

Set locale per connection:

sql

 Copy code

```
SET app.locale = 'tr';
```

Pros

- ✓ Zero Hibernate magic
- ✓ Very fast
- ✓ Clean entities

Cons

- ✗ DB-specific
- ✗ Read-only



Hybrid Approach (MOST COMMON IN REAL SYSTEMS)

Use different strategies for different needs

Use case	Best solution
Lookup tables	Views or explicit joins
Aggregates	Explicit joins
Admin screens	@Filter acceptable
Background jobs	Explicit joins
API DTOs	Explicit joins



When @Filter IS acceptable

`@Filter` is still useful when:

- ✓ Small lookup tables
- ✓ Low write frequency
- ✓ Minimal caching
- ✓ Rapid prototyping
- ✓ Admin UI

But **not** as a global default.

🚫 What to avoid completely

Approach	Reason
SQL rewriting	Unsafe
Intercepting joins	Unsupported
Entity getters returning locale-aware values	Hidden logic
Caching parent entities	Locale leaks

🏆 My recommendation (clear guidance)

Do NOT build your core domain model around `@Filter`.

Instead:

- 1 Use explicit JPQL / DTO projections for business logic
- 2 Use **views** for lookup/reference data
- 3 Use `@Filter` only where invisibility is acceptable

This yields:

- predictable behavior
- safe caching
- simpler debugging

- easier onboarding
-

Final rule of thumb

If the language matters to business logic → make it explicit.
If it's only presentation → filters are OK.

If you want, I can:

- refactor the previous project into the **hybrid model**
- show **performance benchmarks**
- design a **fallback-safe query DSL**
- show **multi-locale caching strategies**

Just tell me how far you want to take this 



but I don't want to write JQL or SQL. What can I do then?

That constraint changes the answer in an important way — thanks for stating it clearly.

If you refuse to write JPQL/SQL, then your options narrow a lot, and we can be very precise about what is realistically possible with JPA/Hibernate *without* query strings.

I'll go from **best** → **worst**, and be very honest about trade-offs.

TL;DR

If you don't want to write JPQL/SQL, the best option is still `@Filter`, but you must constrain where and how you use it.

There is **no magic alternative** in JPA that:

- injects locale into joins

- works transparently
- stays safe
- and avoids JPQL/SQL

Anything claiming to do that is either:

- Hibernate-internal
 - brittle
 - or hiding JPQL somewhere else
-

What you can do (real options)

 Option 1 — `@Filter`, but DONE CORRECTLY (Best fit for your constraint)

You already saw this approach. To make it acceptable:

Rules you MUST follow

1. Use it only on i18n tables
2. Never cache parent entities
3. Enable filter once per request
4. Never depend on fallback logic
5. Never use it in batch/async jobs without re-enabling

If you accept these rules, `@Filter` is the cleanest no-JPQL solution.

Why it's the “least bad”

- ✓ No query strings
 - ✓ Works with Spring Data repositories
 - ✓ Traverses deep object graphs
 - ✓ Predictable if constrained
-

 Option 2 — Hibernate `@Where` (Static locale only)

java

 Copy code

```
@Where(clause = "locale = 'en'")
```

Why it's usually bad

- Locale is hardcoded
- No per-request variation
- Requires entity duplication per language

✗ Almost never correct

✓ Only for single-language deployments

🥇 Option 3 — Application-level localization (Entity stays language-neutral)

Instead of localizing in DB:

java

 Copy code

```
@Entity  
class ProductCategory {  
    String code;  
}
```

Use:

- message bundles
- Redis
- JSON dictionary

Pros

- ✓ No JPQL
- ✓ No Hibernate magic
- ✓ Cache-friendly

Cons

- ✗ Translations outside DB
- ✗ Harder admin UI

This is actually how **many large systems do it.**

✖ What you CANNOT do (without JPQL/SQL)

Idea	Why it fails
Intercept joins	Not supported
Criteria API	Still query building
Entity listeners	Too late
StatementInspector	Unsafe
Spring Data magic	Still generates JPQL

There is **no hook** in JPA to inject a parameter into joins.

A pragmatic architecture (NO JPQL)

Here is the **cleanest realistic design** under your constraint:



Plus these safeguards:

- `@MappedSuperclass` for i18n
- Request interceptor enables filter
- Strict code review rule: no cache on parents
- Integration tests per locale

When this becomes a dead end

You'll feel pain when:

- fallback language is required
- analytics jobs run
- you add query cache
- performance tuning starts

That's your **signal** to allow explicit queries.

Honest conclusion

If “no JPQL/SQL” is a hard rule, `@Filter` is the only sane option.

Everything else either:

- secretly uses queries
- or breaks correctness

There is no better *transparent* mechanism in JPA today.

If you want, next I can:

- harden this approach with **guard rails**
- show **testing strategies**
- show **how to detect missing translations**
- show **how to evolve away from filters later**

Just tell me 



How Option 3 — Application-level localization works?

Great question.

This is actually the **most common approach in large systems**, but it's rarely explained

clearly.

I'll explain how application-level localization works, why it scales, and how to implement it in Spring Boot without JPQL/SQL.

What "Application-level localization" really means

Your database stores language-neutral data only.

Localization happens *after* data is loaded, in the application layer.

So instead of:

```
scss
```

 Copy code

```
ProductCategory
```

```
  └─ ProductCategoryI18n (locale)
```

you have:

```
nginx
```

 Copy code

```
ProductCategory
```

```
  └─ code = "FOOD"
```

And localization is resolved like:

```
arduino
```

 Copy code

```
"FOOD" → "Gıda"
```

Core idea

1 DB stores only stable identifiers

```
java
```

 Copy code

```
@Entity  
public class ProductCategory {
```

```
@Id  
private Long id;  
  
private String code; // FOOD, ELEC, BOOK  
}
```

No locale. No joins. No filters.

2 Translations live outside entities

Typical storage options:

- messages.properties
 - Redis
 - JSON in S3
 - Config server
 - External translation service
-

Implementation in Spring Boot (clean & simple)

1 Message source configuration

java

 Copy code

```
@Configuration  
public class I18nConfig {  
  
    @Bean  
    public MessageSource messageSource() {  
        ReloadableResourceBundleMessageSource ms =  
            new ReloadableResourceBundleMessageSource();  
  
        ms.setBasename("classpath:i18n/messages");  
        ms.setDefaultEncoding("UTF-8");  
        ms.setFallbackToSystemLocale(false);  
        return ms;  
    }  
}
```

```
    }  
}
```

2 Message bundles

messages_en.properties

```
properties  
  
category.FOOD=Food  
category.ELEC=Electronics
```

 Copy code

messages_tr.properties

```
properties  
  
category.FOOD=Gıda  
category.ELEC=Elektronik
```

 Copy code

3 Localization service (centralized)

```
java  
  
@Service  
@RequiredArgsConstructor  
public class LocalizationService {  
  
    private final MessageSource messageSource;  
  
    public String localize(String key, Locale locale) {  
        return messageSource.getMessage(key, null, key, locale);  
    }  
}
```

 Copy code

4 DTO mapping (where localization happens)

java

 Copy code

```
@Component
@RequiredArgsConstructor
public class ProductMapper {

    private final LocalizationService localizationService;

    public ProductDto toDto(Product product, Locale locale) {
        return new ProductDto(
            product.getId(),
            localizationService.localize(
                "category." + product.getCategory().getCode(),
                locale
            )
        );
    }
}
```

5 Controller (no DB changes)

java

 Copy code

```
@GetMapping("/products/{id}")
@Transactional(readOnly = true)
public ProductDto getProduct(
    @PathVariable Long id,
    Locale locale) {

    return mapper.toDto(
        repository.findById(id).orElseThrow(),
        locale
    );
}
```

Where do translations come from in real systems?

Small apps

- `messages.properties`

Medium apps

- JSON files in Git
- Redis
- Feature flags

Large apps

- Translation service
 - CMS
 - Admin UI backed by Redis / NoSQL
 - CDN-cached dictionaries
-

Why this scales extremely well

Problem	DB-level i18n	App-level i18n
Query complexity	High	None
Performance	Joins	O(1) lookup
Caching	Hard	Easy
Fallback	Hard	Built-in
Testing	Complex	Simple
Async jobs	Painful	Safe

How fallback works (automatically)

Spring already does:

arduino

 Copy code

```
tr → en → default
```

No extra logic needed.

Handling dynamic content (plural, params)

properties

 Copy code

```
order.items=You have {0} items
```

java

 Copy code

```
messageSource.getMessage(  
    "order.items",  
    new Object[]{count},  
    locale  
);
```

What about user-editable translations?

Two common patterns:

1 DB-backed dictionary cache

- Admin UI writes to DB
- App loads into Redis
- MessageSource reads from Redis

2 External translation service

- Microservice
- Cached per locale
- Versioned

When this approach is NOT suitable

- ✖ Search/filtering by localized value
- ✖ Sorting by translated name
- ✖ Reporting in multiple languages
- ✖ Business rules depending on language

In those cases → DB-level i18n is needed.

Hybrid approach (most realistic)

- | Use DB-level i18n only where required.
- | Use app-level i18n everywhere else.

This avoids complexity explosion.

Final recommendation

If:

- You don't want JPQL/SQL
- You want predictable behavior
- You want easy caching
- You want clean entities

→ Application-level localization is the best choice.

If you want, I can:

- show a **Redis-backed MessageSource**
- build an **admin UI for translations**
- show **DTO-only localization**
- show **GraphQL integration**
- show **migration strategy from DB-level**

Just tell me what's next 

But I need queries which involves i18n data. For instance Turkish users can search child contains "Gıda" and English users can search childs contains "Food"

This is the **exact point** where the trade-offs become unavoidable — and you've identified the *one capability* that **pure application-level localization cannot provide**.

Searching by localized text ("Gıda" vs "Food") fundamentally requires language-aware data at query time.

So let's be very precise and honest about **what is possible**, **what is not**, and **what architectures actually work in production without writing JPQL/SQL manually**.

Key constraint (non-negotiable)

If users can search like:

- TR user: "Gıda"
- EN user: "Food"

Then **the database (or search engine)** must see **localized text** at query time.

There are **only 4 viable architectures** for this.

✖ What is impossible

These **cannot work**, no matter how clever the framework:

Idea	Why
App-level localization only	DB never sees "Gıda"
@Filter without queries	LIKE still needs join
Entity getters	Happens after query

Idea	Why
Spring Data magic methods	Still generate JPQL
→ Search requires a localized column or index.	

✓ The 4 architectures that actually work

I'll rank them from **best** → **worst**, given your constraints.

🥇 BEST: Search-optimized table (NO JPQL, NO runtime joins)

This is the **cleanest solution** when you don't want JPQL.

Idea

Create a **denormalized search table** that already contains localized text.

markdown

Copy code

```
child_search
-----
child_id
locale
category_name
```

Example data

child_id	locale	category_name
10	tr	Gida
10	en	Food

Entity (simple!)

java

Copy code

```
@Entity
@Table(name = "child_search")
public class ChildSearch {

    @Id
    private Long childId;

    private String locale;
    private String categoryName;
}
```

Repository (Spring Data ONLY)

java

 Copy code

```
List<ChildSearch> findByLocaleAndCategoryNameContainingIgnoreCase(
    String locale,
    String text
);
```

- ✓ No JPQL
- ✓ Uses Spring Data query derivation
- ✓ Indexed
- ✓ Fast
- ✓ Cacheable

How data stays in sync

- DB trigger
- Application event
- Batch refresh
- CDC (Debezium)

 This is how large systems do multilingual search.

Database VIEW + Session locale (NO JPQL)

If DB-specific features are acceptable.

View (Postgres example)

sql

 Copy code

```
CREATE VIEW child_v AS
SELECT
    c.id,
    ci.name AS category_name
FROM child c
JOIN category_i18n ci
    ON ci.category_id = c.category_id
    AND ci.locale = current_setting('app.locale');
```

Set locale once per request

java

 Copy code

```
jdbcTemplate.execute("SET app.locale = 'tr'");
```

Entity maps to view

java

 Copy code

```
@Entity
@Table(name = "child_v")
public class ChildView {
    @Id
    private Long id;
    private String categoryName;
}
```

Repository

java

 Copy code

```
List<ChildView> findByCategoryNameContainingIgnoreCase(String text);
```

- ✓ No JPQL
 - ✓ DB does the join
 - ✗ DB-specific
 - ✗ Read-only
-

🥇 Full-text search engine (Elasticsearch / OpenSearch)

Best UX, but heavier.

Index structure

```
json Copy code  
  
{  
  "childId": 10,  
  "locale": "tr",  
  "categoryName": "Gıda"  
}
```

Query

```
java Copy code  
  
search(locale = "tr", text = "Gıda")
```

- ✓ Language analyzers
 - ✓ Stemming
 - ✓ Ranking
 - ✗ Infrastructure cost
-

🟡 @Filter + Specification (borderline acceptable)

This is the **absolute limit** of “no JPQL”.

```
java Copy code  
  
Specification<Child> spec = (root, query, cb) ->  
  cb.like(
```

```
cb.lower()  
    .root.join("category")  
        .join("translations")  
            .get("name")  
    ),  
    "%" + text.toLowerCase() + "%"  
);
```

⚠ This IS query construction

⚠ Hard to maintain

⚠ Poor performance

⚠ Fragile

I do **not** recommend this for real search.

🏆 Clear recommendation (based on your needs)

Requirement	Best solution
No JPQL/SQL	Search table or View
Locale-aware search	Search table
Performance	Search table
Simplicity	Search table
DB-agnostic	Search table

If users search by localized text, you need localized data at query time.
The cleanest way without JPQL is a dedicated search projection.

Mental model (important)

OLTP tables are for correctness.

Search tables are for user experience.

Trying to do both in one model causes pain.

Final answer (very direct)

✖ There is NO way to support locale-aware search without:

- “localized columns”
- “OR denormalized projections”
- “OR explicit queries”

If you want:

- no JPQL
- clean Spring Data
- fast multilingual search

→ Use a locale-aware search table or view.

If you want next, I can:

- design the sync strategy
- show indexes
- show fallback behavior
- show how to combine with filters
- show migration from @Filter to search table

Just say which one you want 🎉



Show me all you mentioned

Got it 👍

I'll show ALL the approaches I mentioned, end-to-end, in one coherent architecture, so you can see how they fit together, when each is used, and how they coexist without JPQL/SQL.

This is how **real multilingual enterprise systems** are built.



Big Picture (One System, Multiple Strategies)

java

Copy code

```
OLTP entities (language-neutral)
  ↓
I18n source of truth
  ↓
Search projection (locale-aware)
  ↓
Spring Data repositories (NO JPQL)
```

You do NOT choose one — you **combine them**.

1

Source of Truth (Correctness layer)

Core entities (NO i18n)

java

Copy code

```
@Entity
public class ProductCategory {

    @Id
    private Long id;

    @Column(unique = true)
    private String code; // FOOD, ELEC
}
```

java

Copy code

```
@Entity
public class Child {

    @Id
```

```
private Long id;

@ManyToOne(fetch = FetchType.LAZY)
private ProductCategory category;

}
```

- ✓ Stable
 - ✓ Cache-safe
 - ✓ Business-logic-safe
-

2 I18n Storage (Editable translations)

This is NOT queried directly by users.

java

 Copy code

```
@Entity
@Table(
    uniqueConstraints = @UniqueConstraint(
        columnNames = {"category_id", "locale"})
)
public class ProductCategoryI18n {

    @Id
    private Long id;

    @ManyToOne
    private ProductCategory category;

    private String locale;
    private String name; // Food, Gıda
}
```

- ✓ Admin UI writes here
 - ✓ Validation & workflows live here
-

3 Search Projection Table (THE KEY PART)

This is what makes localized search possible without JPQL.

Table

sql

 Copy code

```
CREATE TABLE child_search (
    child_id BIGINT,
    locale VARCHAR(10),
    category_name VARCHAR(100),

    PRIMARY KEY (child_id, locale)
);

CREATE INDEX idx_child_search_locale_name
    ON child_search(locale, category_name);
```

Entity

java

 Copy code

```
@Entity
@Table(name = "child_search")
public class ChildSearch {

    @Id
    private Long childId;

    private String locale;
    private String categoryName;
}
```

Repository (Spring Data ONLY)

java

 Copy code

```
public interface ChildSearchRepository  
    extends JpaRepository<ChildSearch, Long> {  
  
    List<ChildSearch> findByLocaleAndCategoryNameContainingIgnoreCase(  
        String locale,  
        String text  
    );  
}
```

- ✓ No JPQL
- ✓ No Criteria
- ✓ Uses indexes
- ✓ Fast

4 How search data is synchronized

Option A — Application events (most common)

java

 Copy code

```
@Transactional  
public void updateCategoryTranslation(...) {  
    // update ProductCategoryI18n  
    publisher.publishEvent(  
        new CategoryTranslationChanged(categoryId)  
    );  
}
```

java

 Copy code

```
@EventListener  
@Transactional  
public void rebuildSearch(CategoryTranslationChanged e) {  
    // recompute child_search rows  
}
```

Option B — DB triggers (simple, DB-level)

sql

 Copy code

```
CREATE TRIGGER rebuild_child_search
AFTER INSERT OR UPDATE ON product_category_i18n
FOR EACH ROW
EXECUTE FUNCTION rebuild_child_search();
```

Option C — Batch job (eventual consistency)

java

 Copy code

```
@Scheduled(cron = "0 */10 * * *")
void rebuildSearchTable() { ... }
```

5 Locale-aware Search (USER QUERY)

Controller

java

 Copy code

```
@GetMapping("/search")
public List<Long> search(
    @RequestParam String q,
    Locale locale) {

    return childSearchRepository
        .findByLocaleAndCategoryNameContainingIgnoreCase(
            locale.toLanguageTag(),
            q
        )
        .stream()
        .map(ChildSearch::getChildId)
        .toList();
}
```

- ✓ TR user searches "Gıda"
 - ✓ EN user searches "Food"
 - ✓ DB sees localized text
-

6 Loading full entities after search

java

 Copy code

```
List<Child> children =  
    childRepository.findAllById(childIds);
```

- ✓ Search & load separated
 - ✓ Clean architecture
-

7 Optional: Hibernate `@Filter` (READ ONLY UI)

For **non-search pages** (admin, dropdowns):

java

 Copy code

```
@FilterDef(  
    name = "localeFilter",  
    parameters = @ParamDef(name = "locale", type = String.class)  
)  
@Filter(name = "localeFilter", condition = "locale = :locale")
```

Enabled **once per request**.

- ✓ Zero JPQL
 - ✓ Zero duplication
 - ✗ Not used for search
-

8 Optional: Database VIEW (Alternative to search table)

If DB-specific features are OK.

sql

[Copy code](#)

```
CREATE VIEW child_v AS
SELECT
    c.id,
    ci.name AS category_name
FROM child c
JOIN product_category_i18n ci
    ON ci.category_id = c.category_id
    AND ci.locale = current_setting('app.locale');
```

java

[Copy code](#)

```
@Entity
@Table(name = "child_v")
public class ChildView {
    @Id
    private Long id;
    private String categoryName;
}
```

java

[Copy code](#)

```
List<ChildView> findByCategoryNameContainingIgnoreCase(String q);
```

- ✓ No JPQL
- ✗ Read-only
- ✗ DB-specific

9 Optional: Elasticsearch (Advanced UX)

Index structure:

json

[Copy code](#)

```
{
    "childId": 10,
    "locale": "tr",
```

```
        "categoryName": "Gıda"  
    }
```

Query:

java

 Copy code

```
search(locale="tr", q="Gıda")
```

- ✓ Best UX
- ✓ Language analyzers
- ✗ Infrastructure

🔥 Why this architecture works

Concern	Solved by
Correctness	Core entities
Admin edits	I18n table
Search	Search projection
No JPQL	Spring Data
Performance	Indexed search table
Locale isolation	<code>locale</code> column

🏆 Final Guidance (VERY IMPORTANT)

Search and OLTP must be separated.

Trying to do multilingual search directly on domain entities leads to:

- “complex queries”
- “caching bugs”
- “poor performance”
- “unreadable code”

This design avoids all of that.

If you want next

I can:

- generate **DDL + Flyway scripts**
- add **full-text indexes**
- show **fallback locale search**
- show **multi-tenant + locale**
- show **testing strategy**
- show **migration from @Filter-only**

Just tell me where you want to go next 

