Beyond the Blob

Mastering JSON with Hibernate ORM

Disclaimer

The content of this presentation is my own and doesn’t necessarily represent IBM’s positions, strategies or opinions.

Who am I?

- Christian Beikov
- Long time Hibernate ORM community contributor
- Full time Hibernate dev at Red Hat/IBM since 2020
- Founder of Blazebit and creator of Blaze-Persistence
- Living in Düsseldorf/DE and Vienna/AT
- Like to play tennis and basketball

Hibernate ORM

- Open Source Java ORM since May 2001
- Major component in Spring, Quarkus and Jakarta EE
- Implementation of Jakarta Persistence spec
- Radical internal changes in version 6.0
- Apache License 2.0 since version 7.0
- Support for more advanced SQL
- SQL types like struct, XML and JSON

A JSON evolution story
at a startup

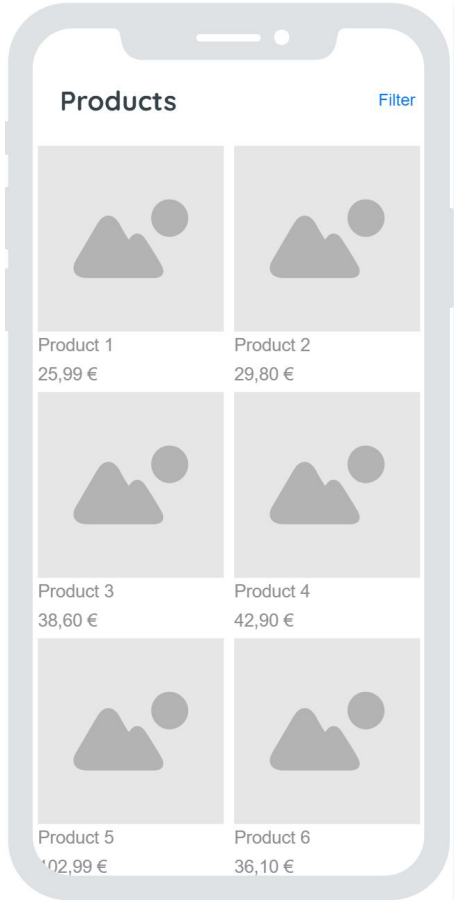
Once upon a time was a startup

Startup for fashion product platform

App for product search

Show offers from different stores

Integrate product catalogues from affiliate partners



First contact with product catalogue

JSON data for products


```
{
  "products": [
    {
      "id": 1,
      "title": "Product 1",
      "description": "This is a good product.",
      "category": "clothing",
      "price": 29.99,
      "discountPercentage": 7.17,
      "rating": 4.94,
      "stock": 5
      "tags": [
        "clothing",
        "vintage"
      ],
      "brand": "ABC",
      "sku": "ABC123",
      "weight": 2,
      "dimensions": {
        "width": 23.17,
        "height": 14.43,
        "depth": 28.01
      },
      "warrantyInformation": "1 month warranty",
      "shippingInformation": "Ships in 1 month",
```

Initial Requirements

- Extract certain data (EAN, price, URL, etc.) into columns
- Match same products based on EAN etc.
- Regular data synchronization
- Also store JSON as blob “just in case”
- Quick time-to-market

Fast and easy solution

- Read JSON into Map to extract data for columns
- Schedule cron job for synchronization

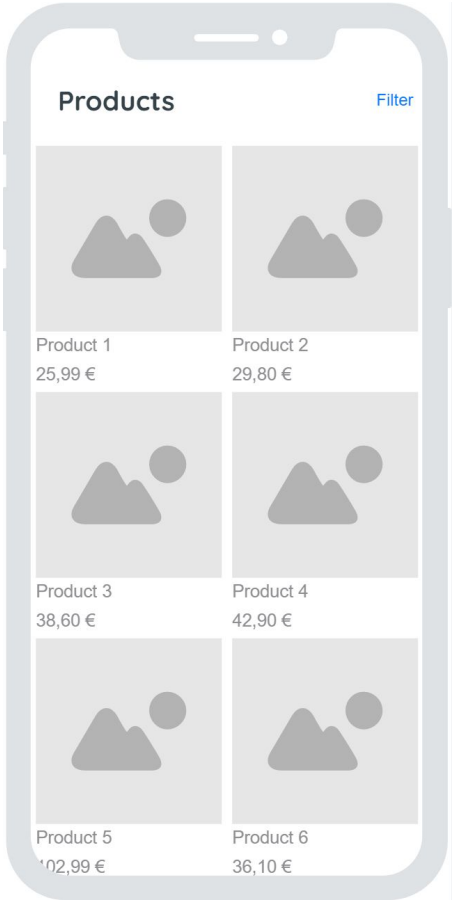
```
ObjectMapper mapper  = new ObjectMapper();

map = mapper.readValue(json, Map.class);
```

```
@Entity
public class Product {
   @Id @GeneratedValue
   private Long id;
   private String name;
   private String description;
   private String[] categories;
   private BigDecimal price;
   private Color[] colors;
   private Size[] sizes;
   @OneToMany(mappedBy = "product")
   private Set<ProductOffer> offers;
   @JdbcTypeCode(SqlTypes.JSON)
   private String json;
}
```

First app version

- After a couple of hours of import... it works :)
- We published the app and wanted to present it
- But we forgot to extract a field...



Handle missing field

- Luckily data is in JSON Blob
- But data extraction is part of import code...
- So add a new column and backfill through re-import?
- Or can we work with the JSON Blob directly?

Our friend the Blob

JSON functions to the rescue

- Adding new column is painful as it requires re-import
- Don’t know if the data is really useful yet
- HQL supports extraction with `JSON_VALUE`/`JSON_QUERY`
- So let’s adapt the query!

Our friend the Blob

JSON_VALUE function

- JSON_VALUE extracts scalar value with JSON Path (RFC 9593)
- Optional RETURNING clause to specify type, String is default

```
"json_value(" expression "," expression passingClause? ( "returning" castTarget)?
onErrorClause? onEmptyClause?  ")"

passingClause
   : "passing" expression "as" identifier ("," expression "as" identifier)*;

onErrorClause
   : ( "error" | "null" | ( "default" expression ) ) "on error";

onEmptyClause
   : ( "error" | "null" | ( "default" expression ) ) "on empty";
```

Our friend the Blob

JSON_QUERY function

- JSON_QUERY extracts JSON value with JSON Path (RFC 9593)
- Extract JSON object or array

```
"json_query(" expression "," expression passingClause? wrapperClause?
onErrorClause? onEmptyClause?  ")"

wrapperClause
   : "with" ("conditional"|"unconditional")? "array"? "wrapper"
   | "without" "array"? "wrapper"

passingClause
   : "passing" expression "as" identifier ("," expression "as" identifier)*

onErrorClause
   : ( "error" | "null" | ( "empty" ( "array" | "object" )? ) ) "on error";

onEmptyClause
   : ( "error" | "null" | ( "empty" ( "array" | "object" )? ) ) "on empty";
```

Our friend the Blob

HQL query

- Pass through extracted values

```
select new ProductDto(
   p.id, /* other fields */
   json_value(p.json, '$.terms'),
   json_query(p.json, '$.tags')
)
from Product p
where p.id in :ids
```

Our friend the Blob

`JSON_VALUE` and `JSON_QUERY` support

- Simple JSON Path `$.persons[0].id`
- Wildcard support `$.persons[*].id`

Database	`JSON_VALUE`	`JSON_QUERY`	JSON Path
DB2	✓	✓	✓ Only simple and wildcard
PostgreSQL	✓	✓	✓
MySQL / MariaDB	✓	✓	✓ Only simple and wildcard
Oracle	✓	✓	✓
SQL Server	✓	✓	✓ Only simple
H2	✓	✓	✓ Only simple
…			

Going deeper
Taming complexity

Handling complex JSON extraction

Challenges

- How to extract data matching a filter?
- How to extract array elements as rows?
- Are multiple JSON functions going to perform well?
- What about JSON Path database portability?

Handling complex JSON extraction

JSON Path

- JSON Path filters
- “Pipe” values into downstream expression
- Functions to operate on values

```
$.array[*] ? (@.id == 1).name
$.array[*] ? (@.id == 2).tags.size()
```

Handling complex JSON extraction

JSON Path disadvantages

- JSON Path filter is mighty, but support varies between DBs
- Also lacks aggregation support
- What about JSON parsing/processing performance?

Handling complex JSON extraction

`JSON_TABLE` function

- `JSON_TABLE` turns JSON data to relational representation
- Takes JSON, JSON Path and column list
- Ensures JSON is only parsed/processed once

Handling complex JSON extraction

JSON_TABLE function

- JSON_TABLE is a set-returning function, added in version 7.0
- Usage in the FROM clause, with LATERAL for a JOIN

```
"json_table(" expression ("," expression)? passingClause? columnsClause errorClause?  ")"

columnsClause
   : "columns(" column ("," column)* ")"

column
   : "nested" "path"? STRING_LITERAL columnsClause
   | attributeName "json" wrapperClause? ("path" STRING_LITERAL)? queryOnErrorClause?
queryOnEmptyClause?
   | attributeName "for ordinality"
   | attributeName "exists" ("path" STRING_LITERAL)? existsOnErrorClause?
   | attributeName castTarget ("path" STRING_LITERAL)? valueOnErrorClause? valueOnEmptyClause?

queryOnErrorClause
   : ( "error" | "null" | ( "empty" ( "array" | "object" )? ) ) "on error";
…
```

Handling complex JSON extraction

Extracting maximum value

```
select new ProductDto(
   p.id, /* other fields */
   t.tags,
   t.terms,
   max(t.price)
)
from Product p
left join lateral json_table(p.json, '$' columns(
   tags JSON,
   terms String,
   nested '$.variants[*]' columns(
      price BigDecimal
   )
)) t
where p.id in :ids
group by p, t.tags, t.terms
```

Handling complex JSON extraction

`JSON_TABLE` support

- `JSON_TABLE` transformation into `JSON_VALUE` and `JSON_QUERY`
- Sometimes requires joining N generated rows for array unwrapping

Database	`JSON_TABLE`	Note
DB2	✓	Transformed, max. 10000 array elements
PostgreSQL	✓	Native since 17, transformed before
MySQL / MariaDB	✓	Native support
Oracle	✓	Native support
SQL Server	✓	Transformed
H2	✓	Transformed, max. 1000 array elements
…		

Handling complex JSON extraction

`JSON_TABLE` advantages

- Database portability
- Relational data can be aggregated
- Read/Process JSON only once
- Easier to read

Handling complex JSON extraction

`JSON_TABLE` challenges

- JSON functions allow making use of the Blob, but…
  - It’s data modeling at query time
  - Repetitive when needed often
  - No data conformance guarantee

- So, back to relational?

Going beyond
Modeling JSON

Modeling structured data as “document”

- Hibernate ORM aggregate embeddables, since version 6.2
- Model contents of `JSON`/`XML` or `STRUCT` column
- Constraints for nested fields
- Canonical Java model for querying
- Can be combined with embeddable inheritance

Modeling structured data as “document”

Entity model

```
@Entity
@Table(name = "product")
public class ProductForQuery {
    @Id @GeneratedValue
    private Long id;
    private String name;
    private String description;
    private String[] categories;
    private BigDecimal price;
    private Color[] colors;
    private Size[] sizes;
    @OneToMany(mappedBy = "product")
    private Set<ProductOfferForQuery> offers;
    @JdbcTypeCode(SqlTypes.JSON)
    private ProductEmbeddable json;
}
```

```
@Embeddable
public class ProductEmbeddable {
    @Column(nullable = false)
    private String terms;
    private String tags;
    @JdbcTypeCode(SqlTypes.JSON_ARRAY)
    private List<ProductVariantEmbeddable>
variants;
}
@Embeddable
public class ProductVariantEmbeddable {
    private BigDecimal price;
}
```

Modeling structured data as “document”

PostgreSQL DDL generation

```
create table product (
   id          bigint not null,
   description varchar(255),
   name        varchar(255),
   categories  varchar(255) array,
   colors      smallint array,
   json        jsonb,
   sizes       smallint array,
   primary key (id),
   check (json is null
      or (cast(json ->> 'terms' as varchar(255)) is not null)
   )
)
```

Modeling structured data as “document”

HQL query

```
select new ProductDto(
   p.id, /* other fields */
   p.json.tags,
   p.json.terms,
   max(v.price)
)
from ProductForQuery p
left join p.json.variants v
where p.id in :ids
group by p, p.json.tags, p.json.terms
```

Modeling structured data as “document”

Database support

Database	JSON	XML	Struct	Note
DB2	✓	✓	✓	Struct requires XML transformation
PostgreSQL	✓	✓	✓	
MySQL / MariaDB	✓	✗	✗	
Oracle	✓	✓	✓	
SQL Server	✓	✓	✗	Struct needs CLR
H2	✓	✗	✗	
…				

But what about filtering?

- Accessing what wasn’t extracted is nice, but…
  - Can we implement filtering?
  - How to filter arrays?
  - What about performance?

But what about filtering?

How it can be done

- Basic filtering works fine with paths

```
p.json.terms like :terms||'%'
```

- Array filtering requires “unnest” through join

```
exists (
   select 1
   from p.json.tags t
   where t in :tags
)
```

But what about filtering?

Basic filter performance

```
select … from product pfq1_0
where cast(pfq1_0.json->>'terms' as varchar(255)) like ('%'||?||'%') escape ''

create index product_terms on product using gin (cast(json->>'terms' as varchar(255))
gin_trgm_ops);

Bitmap Heap Scan on product pfq1_0  (cost=640.37..5969.43 rows=1600 width=8)
  Recheck Cond: ((((""json"" ->> 'terms'::text)) ~~ '%These%'::text)
  ->  Bitmap Index Scan on product_terms  (cost=0.00..639.97 rows=1600 width=0)
        Index Cond: ((((""json"" ->> 'terms'::text)) ~~ '%These%'::text)
```

Working fine, but recheck might be expensive

Consider a generated stored column for that

But what about filtering?

Generated column

```
alter table product add column generatedTerms varchar(255) generated always as (json->>'terms')
stored;
```

And add to entity model

```
@GeneratedColumn ("json->>'terms'")
private String generatedTerms;
```

Generated column can be overridden with `DialectOverride.GeneratedColumn`

But what about filtering?

Array filter performance

```
select … from product pfq1_0
where exists(
     select 1
     from lateral json_table(pfq1_0.json->'tags','$[*]' columns(
          t1_0 text path '$'
     )) t1_0
     where t1_0.t1_0 in (?)
)
```

```
Seq Scan on product pfq1_0  (cost=0.00..1304538.00 rows=500000 width=8)
  Filter: EXISTS(SubPlan 1)
  SubPlan 1
    ->  Table Function Scan on ""json_table"" t1_0  (cost=0.01..1.26 rows=1 width=0)
          Filter: (t1_0 = 'a'::text)
```

Not so good

But what about filtering?

Array filter performance

- Need a trick to make it work nicely

```
create or replace function  json_to_string_array( jsonb) returns varchar [] as $$
select coalesce(
   array_agg(x),
   case when $1 is null then null else array []::varchar[] end
)
from jsonb_array_elements_text($1) t( x);
$$ language sql immutable ;

create index product_tags  on product using gin (json_to_string_array( json->'tags'));
```

But what about filtering?

Array filter performance

- Make Hibernate aware of this function

```
public class MyFunctionContributor implements FunctionContributor {
   @Override
   public void contributeFunctions(FunctionContributions functionContributions) {
      TypeConfiguration typeConfiguration = functionContributions.getTypeConfiguration();
      BasicType<String[]> stringArrayType =
           typeConfiguration.getBasicTypeForJavaType( String[]. class );
      functionContributions.getFunctionRegistry()
           .registerNamed( "json_to_string_array", stringArrayType );
   }
}
```

But what about filtering?

Array filter performance

- Then we can filter easily with index support

```
array_contains(
   json_to_string_array(p. json.tags),
   :tags
)

Bitmap Heap Scan on product pfq1_0  (cost=48.26..1556.10 rows=500 width=8)
  Recheck Cond: (json_to_string_array((""json"" -> 'tags'::text)) @> '{a}'::character varying[])
  ->  Bitmap Index Scan on product_tags  (cost=0.00..48.14 rows=500 width=0)
        Index Cond: (json_to_string_array((""json"" -> 'tags'::text)) @> '{a}'::character
varying[])
```

Again, recheck might be expensive

Consider a generated stored column for that

Conclusion

- Hibernate ORM has mighty tools to handle JSON
- Good performance sometimes requires extra work
- Monitor performance and careful with index rechecks
- Relational modeling can still be faster
- But sometimes time to market is more important

Questions?

github.com/beikov

x.com/c_beikov

christian.beikov@ibm.com

Examples: github.com/beikov/beyond-the-blob

What happened to the startup?

- Like many other startups, dissolved
- But not because of JSON ;)
- Didn’t find the right investor in time
- Couldn’t make enough money to sustain
- Dispute over future of company
- Chat with me if you want to know more!