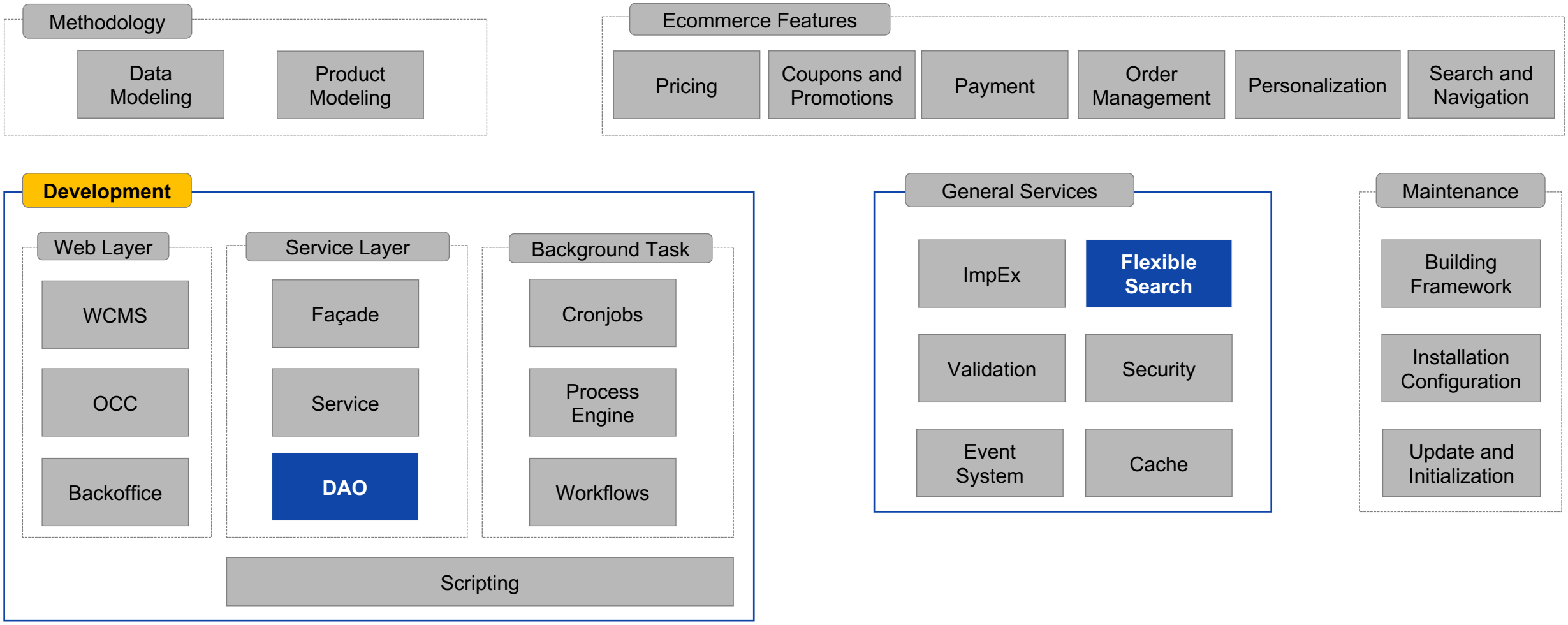
SAP Customer Experience

FlexibleSearch

What we will cover in this topic



We will learn about:

- Overview
- Syntax
- API Examples
- FlexibleSearch Alternatives

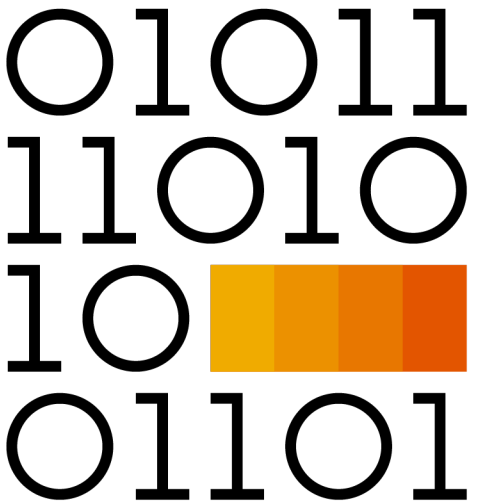
The Context

FlexibleSearch is a built-in **query language using a SQL-based syntax**. It enables searching for items in SAP Commerce Cloud.

Preparation

Complete step P1 of the Flexible Search exercise

(The setup ant target will compile your system during the lecture)



Overview

Overview

- SQL-like syntax
- Abstracts a database query into a Commerce Item query
- Capable of returning a list of item type instances instead of list of individual property values
- Alternatively, attributes of SAP Commerce items are also queryable
- Is translated into native SQL statements on execution
- Allows nearly every feature of SQL SELECT statements
- Queries go through cache

Syntax

Syntax

- Basic Syntax:

```
SELECT {attribute1}, {attribute2}, ... {attributeN} FROM {type} (where <conditions>)?
   (ORDER BY <order>)?
```

- Mandatory:

```
SELECT {attribute1}, {attribute2}, ... {attributeN}
FROM {type}
```

- Optional:

```
WHERE <conditions>
ORDER BY <order>
```

- SQL Command / Keywords:

ASC, DESC, DISTINCT, AND, OR, LIKE, LEFT JOIN, CONCAT, ...

Query examples

- Basic query

```
SELECT {PK} FROM {Car}
```

- Simple queries

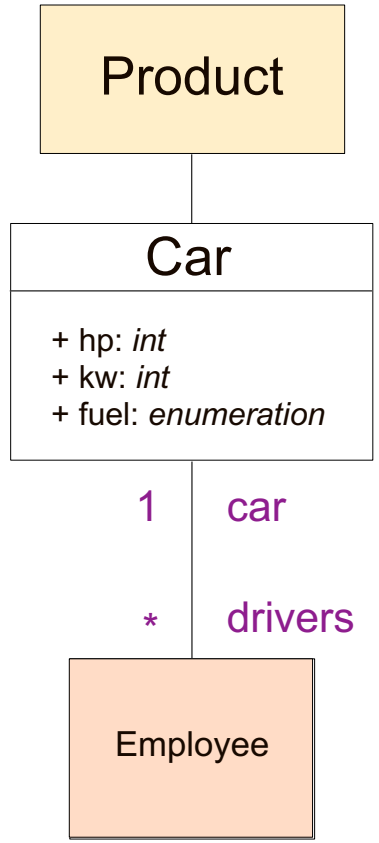
```
SELECT {code}, {hp} FROM {Car}
```

- Single type queries
  - Returns only Product items, not subtypes

```
SELECT {code} FROM {Product!}
```

- Joins

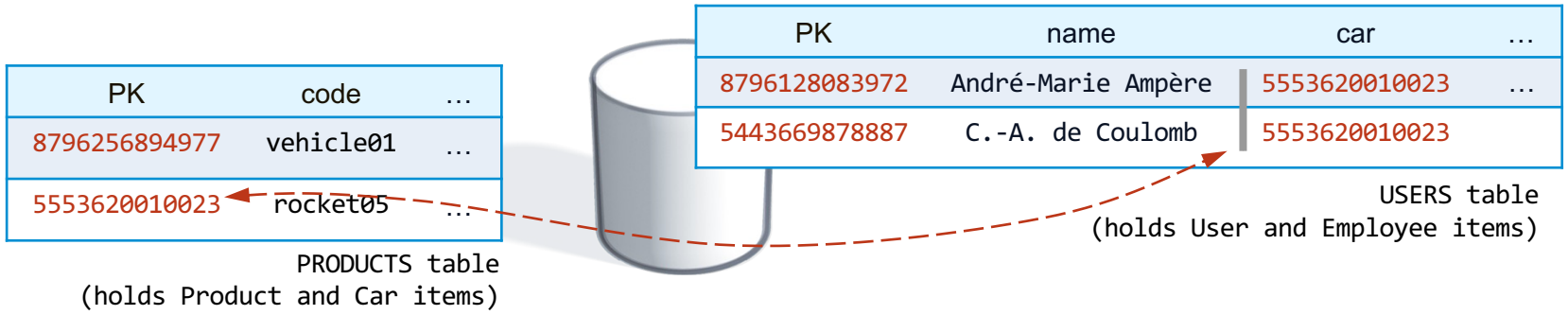
```
SELECT {c.code},{e.uid} FROM {
     Car AS c JOIN Employee AS e
              ON {c.pk} = {e.car}
} WHERE {e.uid} LIKE '%Columbo'
```



Joins for One-to-Many Relations

- Recall the following one-to-many relation from a previous chapter:

```
<relation code="Car2DriversRelation" generate="true" autocreate="true" localized="false" >
      <sourceElement qualifier="car" type="Car" cardinality="one" />
      <targetElement qualifier="drivers" type="Employee" cardinality="many"/>
</relation>
```



- Based on the table deployment of this kind of relation, the JOIN statement would be:
  (note the use of SAP Commerce types instead of table names)

```
SELECT {c.code} as "Car", {e.name} as "Driver"
FROM {  Employee as e JOIN Car as c
                      ON {e.car}={c.PK} }
```

Joins for Many-to-Many Relations

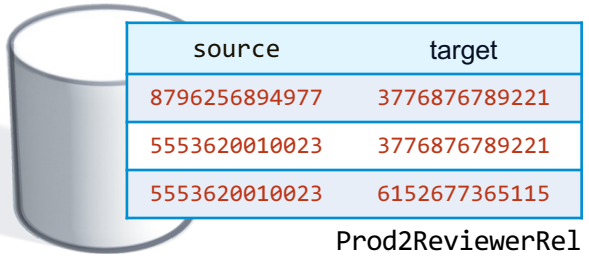
- Recall the following many-to-many relation from a previous chapter:

```
<relation code="Product2ReviewerRelation" autocreate="true" generate="true" localized="false">
    <deployment table="Prod2ReviewerRel" typecode="20123"/>
    <sourceElement qualifier="products" type="Product" cardinality="many" />
    <targetElement qualifier="reviewers" type="Employee" cardinality="many" />
</relation>
```

Relation's "products"

PK	code	…
8796256894977	vehicle01	…
5553620010023	rocket05	…

PRODUCTS table
(holds Product and Car items)



source	target
8796256894977	3776876789221
5553620010023	3776876789221
5553620010023	6152677365115

Prod2ReviewerRel

Relation's "reviewers"

PK	uid	…
3776876789221	ajfoyt	…
6152677365115	mandretti	…

USERS table
(holds User and Employee items)

- Based on the table deployment of this kind of relation, the JOIN statement would be:
  (note the use of SAP Commerce types instead of table names)

```
select {p.code} AS "Product", {e.uid} AS "Reviewer"
from { Employee AS e JOIN Product2ReviewerRelation AS p2r
                     ON {p2r.target}={e.PK}
                     JOIN Product AS p
                     ON {p2r.source}={p.PK}   }
```

More Query Examples

- Inner queries:

```
SELECT {c.code} FROM {Car as c}
   WHERE {c.mechanic} IN
   ({{
      SELECT {PK} FROM {Employee}
         WHERE {uid} LIKE '%Tesla'
    }})
```

- Group functions:
  (notice only attribute names and SAP Commerce types are enclosed in curly braces **{ }**)

```
SELECT count(*) FROM {Car}
```

More Query Examples

- Order By clauses:
  - Ascending order (ASC is default, thus optional)

```
SELECT {code}, {name[en]} FROM {Product} ORDER BY {name[en]}
SELECT {code}, {name[en]} FROM {Product} ORDER BY {name[en]} ASC
```

  - Descending order

```
SELECT {code}, {name[en]} FROM {Product} ORDER BY {name[en]} DESC
```

Using Dates (When In HAC Flexible Search Console)

- Date literals (specific to underlying DB):

```
SELECT {c.code} FROM {Car as c}
   WHERE {Car.warrantyExpiry} < '2020-12-01 0:00:00.0'
```

- Date functions (specific to underlying DB):
  - E.g. in HSQLDB: CURDATE and TODAY are aliases for CURRENT_DATE (SYSDATE also works)

```
SELECT {c.code} FROM {Car as c}
  WHERE {Car.warrantyExpiry} < TODAY
```

API Examples

Querying for SAP Commerce items

In Java, we normally retrieve whole model objects, rather than SELECT a set of individual properties

- A `SearchResult` of PKs is returned by `FlexibleSearchService.search()` when you SELECT the `{PK}` property (and nothing else)

```
"SELECT {PK} FROM {MyCommerceType}"
```

- Transform / "convert" the returned **`SearchResult`** into a Java **`List`** of "whole objects" e.g., `List<`*`MyCommerceTypeModel`*`>`:

```
import de.hybris.platform.servicelayer.search.SearchResult;
. . .
public List<CarModel> getAllCars() {
   String queryStr = "SELECT {PK} FROM {Car}";
   FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
   SearchResult<CarModel> result = getFlexibleSearchService().search( fsq );
   List<CarModel> cars = result.getResult();
   return cars;
}
```

  - Or combine the last three statements, using generics:

```
return getFlexibleSearchService().<CarModel>search( fsq ).getResult();
```

Binding Atomic-Type Parameter Data

- Parameter-value bindings reference map keys using *?key*
  - For **atomic types**, values map conveniently to Java wrapper-class objects, etc.

```
import de.hybris.platform.servicelayer.search.SearchResult;
. . .
public List<CarModel> getCarsByHpRange( Integer minHp, Integer maxHp ) {
    String queryStr = "SELECT {PK} FROM {Car} WHERE {hp} >= ?hpMin AND {hp} <=
?hpMax";
    FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
    fsq.addQueryParameter("hpMin", minHp);
    fsq.addQueryParameter("hpMax", maxHp);
    SearchResult<CarModel> result = getFlexibleSearchService().search( fsq );
    List<CarModel> cars = result.getResult();
    return cars;
}
```

Binding Non-Atomic (Reference-Type) Parameter Data

- Parameter-value bindings reference map keys using *?key*
  - For **non-atomic SAP Commerce types**, item references (PKs) map conveniently to Java Model object references

```
import de.hybris.platform.servicelayer.search.SearchResult;
. . .
public List<CarModel> getCarsByMechanic( EmployeeModel mechanic ) {
    String queryStr = "SELECT {PK} FROM {Car} WHERE {mechanic} = ?mechanic";
    FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
    fsq.addQueryParameter("mechanic", mechanic);
    SearchResult<CarModel> result =
            getFlexibleSearchService().search( fsq );
    List<CarModel> cars = result.getResult();
    return cars;
}
```

Querying for 1 Attribute (Instead of for Whole Model Objects)

- The data type for the single column must be mapped in the form of a 1-element `List<Class>`
  - For a 1-column query, `result.getResult()` still returns a `List<Object>`
  - The `List` is `ArrayList<valueClass>` where each entry is the sole value object representing a result "row"

```
String queryStr = "SELECT {vin} FROM {Car}";
FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
Class[] resultTypesArray = { String.class };
fsq.setResultClassList( Arrays.asList( resultTypesArray ) );
SearchResult<String> result =  getFlexibleSearchService().search( fsq );
List<String> vinList = result.getResult();
for( String carVin : vinList ) {
    logger.info( "VIN: " + carVin );
}
```

Try to use Models whenever it makes sense

Querying for 2+ Attributes (Instead of for Whole Model Objects)

- The column data types must be mapped positionally in the form of a List<Class>
  - For a 2-or-more-column query, `result.getResult()` returns a `List< List<Object> >`
  - Each element of the outer List represents a “query-result row”;
  - Each “query-result row” is a positional List of column values

Try to use Models whenever it makes sense

```
String queryStr = "SELECT {vin}, {weight} FROM {Car}";
FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
Class[] resultTypesArray = { String.class, Integer.class };
fsq.setResultClassList( Arrays.asList( resultTypesArray ) );
SearchResult< List<Object> > result =  getFlexibleSearchService().search( fsq );
List< List<Object> > resultRowList = result.getResult();
for( List<Object> columnValuesForRow : resultRowList ) {
     final String vin = (String)columnValuesForRow.get(0);
     final Integer weight = (Integer)columnValuesForRow.get(1);
     System.out.println( "Car with vin " + vin + " weighs " + weight.intValue() + " kg." );
}
```

Querying Against Today's Date • Date-Only Values (Caching Considerations)

- When comparing with today's date, truncate date value
  - Every FlexibleSearch query is cached, but using the current date/time value — which changes every millisecond — has the effect of never being able to reuse the cached query result
  - Truncate date as needed — for example, to nearest day:
    (if the attribute was meant to be date-only, you would want to truncate the time portion)

```
String queryStr = "SELECT {PK} FROM {Car} WHERE {Car.warrantyExpiry} < ?today";
final Calendar todayCal = Calendar.getInstance();  //initializes to current system time
   todayCal.set(Calendar.HOUR_OF_DAY, 0);
   todayCal.set(Calendar.MINUTE, 0);
   todayCal.set(Calendar.SECOND, 0);
   todayCal.set(Calendar.MILLISECOND, 0);   //VERY easy to forget to zero-out milliseconds
FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
Date todayDate = todayCal.getTime();  // need an instance of java.util.Date for query param value
fsq.addQueryParameter("today", todayDate );
SearchResult<CarModel> searchResult = getFlexibleSearchService().search( fsq );
List<CarModel> cars = searchResult.getResult();
```

Pagination

- Paginate to reduce data-transfer bandwidth

```
final int PAGE_SIZE = 5;
String queryStr = "SELECT {PK} FROM {Car}";
FlexibleSearchQuery fsq;
private void initializeQuery(){ //called by a constructor
      fsq = new FlexibleSearchQuery( queryStr );
      fsq.setNeedTotal( true );//
      fsq.setCount( PAGE_SIZE );
}
. . .
List<CarModel> getCarsByPage( FlexibleSearchQuery fsq, int pageNum ) {
      fsq.setStart( (pageNum – 1) * PAGE_SIZE );
      //Note the Java Generics syntax when both method calls combined in single statement
      return getFlexibleSearchService().<CarModel>search( fsq ).getResult();
}

. . .
List<CarModel> pageOfCars = carDAO.getCarsByPage( fsq, 3 ); //reuse fsq for the 3rd page
. . .
```

Using Model Constants for Attribute and Type Names • Failsafe Approach

- When building a query in Java code, use the static constants defined in each model class to refer to its attribute names
  - Each attribute's name is held by a corresponding all-upper-case constant
  - The name of the SAP Commerce type represented by this model class is contained by `_TYPECODE`
  - Using constants makes your code more difficult to read, but make it impossible to misspell attribute names without causing an immediate compilation error in your DAO classes

- For example, instead of

```
String queryStr = "SELECT {code}, {hp} FROM {Car}";
```

- Use the static constants

```
String queryStr = "SELECT {" + CarModel.CODE + "}, {" + CarModel.HP + "}"
                  + " FROM {" + CarModel._TYPECODE + "}";
```

FlexibleSearch Alternatives

GenericDao

A helper class that dramatically simplifies basic parameter searches

- Use GenericDao as an alternative to FlexibleSearch
  - Configure target item type via `constructor-arg`
  - Tip: use Spring Expression Language (SpEL) shortcut to refer to Model's static typecode variable
  - `DefaultGenericDao` (concrete class → bean) implements `GenericDao` (interface → alias)

```
<alias alias="productDao" name="defaultProductDao">
<bean name="defaultProductDao" class="de.hybris.platform.servicelayer.internal.dao.DefaultGenericDao">
    <constructor-arg value="#{T(de.hybris.platform.core.model.product.ProductModel)._TYPECODE}" />
</bean>
```

  - Performs simple searches and sorts
    - (Optional param) `Map<String, Object>` converts to "`WHERE attrib1=value1 AND attrib2=value2`", ...
    - As a very simple example, to return all products of a particular weight:

```
public GenericDao<ProductModel> productDao;      // Spring bean ref. injected using setProductDao(...)
. . .
public List<ProductModel> getProductsByApprovalStatus( ArticleApprovalStatus approvalStatus ) {

    final Map<String, Object> params = new HashMap<>();
    params.put(ProductModel.APPROVALSTATUS, approvalStatus );  //enum value
    List<ProductModel> products = productDao.find(params);     // Note how productDao.find() returns desired type
    return products;
}
```

Further Alternatives

- Generic Search
- Polyglot Persistence

Flexible Search Alternatives are covered in "SAP Commerce Cloud - Additional Technical Essentials"

References

- FlexibleSearch

  https://help.sap.com/viewer/aa417173fe4a4ba5a473c93eb730a417/latest/en-US/8bc399c186691014b8fce25e96614547.html

- GenericSearch

  https://help.sap.com/viewer/aa417173fe4a4ba5a473c93eb730a417/latest/en-US/8bc5a812866910149a67fcea0efc78cf.html

- Polyglot Persistence

  https://help.sap.com/viewer/aa417173fe4a4ba5a473c93eb730a417/latest/en-US/a94ab1238cc34f94bc9dee639e256440.html

- Polyglot Persistence Query Language

  https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/aa417173fe4a4ba5a473c93eb730a417/651d603ed81247c2be1708f22baed11b.html

- AbstractItemDao API Documentation

  https://help.sap.com/doc/9fef7037b3304324b8891e84f19f2bf3/latest/en-US/de/hybris/platform/servicelayer/internal/dao/AbstractItemDao.html

Key Points

1. FlexibleSearch abstracts a database query into a SAP Commerce Item query
2. It returns a List of **models** (SAP Commerce items) – except in the HAC
3. It is **translated into native SQL** statements on execution
4. Queries go through the **cache** – avoid frequently-changing queries
5. Use joins for 1:n and n:m relations if necessary
6. Use `FlexibleSearchService` to execute queries
7. In queries, refer to Model Attributes (static constants) for a more failsafe approach
8. Use `GenericDao` as a simple, convenient alternative to custom DAOs that use FlexibleSearch

Flexible Search Exercise

Thank you.