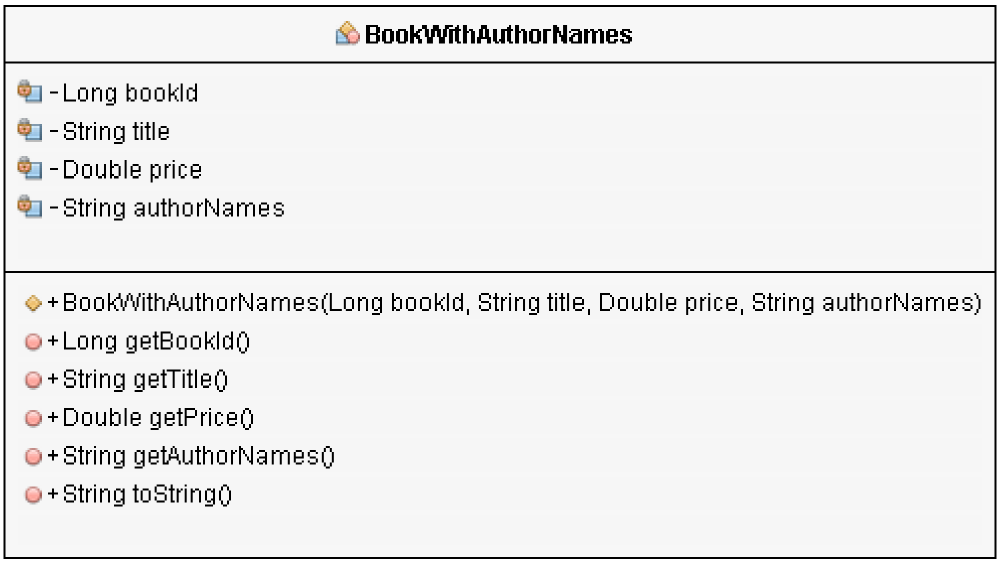
Why, When and How to Use DTO Projections

DTOs are easy to use and the most efficient projection for read-only operations. So, whenever you don't need to change the requested information, you should prefer a DTO projection.

What is a DTO?

In its modern interpretation, a DTO is used as a specialized class to transfer data that you selected in a database query. Now, the goal of a DTO is to read the required information with as few database queries as possible and to provide it in an efficient and easy to use form.

A DTO is a simple Java class that you can specifically design for a use case. The *BookWithAuthorNames* class, for example, only has the 4 attributes that are required to show a list of search results. These are the *id*, *title*, and *price* of the book and a *String* with the name of the author.



How do DTO projections work with JPA and Hibernate

Your database and the SQL language don't know about your Java classes. They only know tables, columns, views, stored procedures, and other database-related concepts.

So, your persistence provider, e.g., Hibernate or EclipseLink, need to handle the DTO projection. It does that when it processes the result set of your query. Instead of mapping each row to an *Object[]*, your persistence provider calls the constructor of your DTO to instantiate a new object. So, you need to make sure that your DTO always has a constructor that matches the columns selected by your query. But more about that later.

Using DTO projections with JPA and Hibernate

After you defined your DTO class, you can use it as a projection with JPQL, criteria and native queries. For each kind of query, you need to define the DTO projection differently, but the result is always the same. Your persistence provider instantiates a new DTO object for each record in the result set.

DTO projections in JPQL

JPQL queries offer a feature called constructor expression. With such an expression you can define a constructor call with the keyword new followed by the fully qualified class name of your DTO and a list of constructor parameters in curly braces.

```
TypedQuery<BookWithAuthorNames> q = em.createQuery(
        "SELECT new org.thoughts.on.java.model.BookWithAuthorNames”
        + “(b.id, b.title, b.price, concat(a.firstName, ' ', a.lastName)) “
        + “FROM Book b JOIN b.author a WHERE b.title LIKE :title",
        BookWithAuthorNames.class);
q.setParameter("title", "%Hibernate Tips%");
List<BookWithAuthorNames> books = q.getResultList();
```

DTO projections in criteria queries

You define a DTO projection in a *CriteriaQuery* in a pretty similar way as you do in JPQL. But instead of using the new keyword to specify the constructor call in a query *String*, you call the *construct* method on the *CriteriaBuilder* with a reference to the DTO class and multiple constructor parameters.

```
// Create query
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<BookWithAuthorNames> cq = cb
        .createQuery(BookWithAuthorNames.class);
// Define FROM clause
Root<Book> root = cq.from(Book.class);
Join<Book, Author> author = root.join(Book_.author);

// Define DTO projection
cq.select(cb.construct(
        BookWithAuthorNames.class,
        root.get(Book_.id),
        root.get(Book_.title),
        root.get(Book_.price),
        cb.concat(author.get(Author_.firstName), ' ',
                author.get(Author_.lastName))));

// Define WHERE clause
ParameterExpression<String> paramTitle = cb.parameter(String.class);
cq.where(cb.like(root.get(Book_.title), paramTitle));

// Execute query
TypedQuery<BookWithAuthorNames> q = em.createQuery(cq);
q.setParameter(paramTitle, "%Hibernate Tips%");
List<BookWithAuthorNames> books = q.getResultList();
```

DTO projections for native SQL queries

Hibernate sends native SQL queries directly to the database and doesn't parse them. That's one of the reasons why you can use all the features supported by your database in a native query.

But that also means that you can’t use a constructor expression to define the DTO projection within your query. You need to define an *@SqlResultSetMapping* instead.

```
@Entity(name = "Book")
@SqlResultSetMapping(
        name = "BookWithAuthorNamesMapping",
        classes = @ConstructorResult(
                targetClass = BookWithAuthorNames.class,
                columns = { @ColumnResult(name = "id", type = Long.class),
                        @ColumnResult(name = "title"),
                        @ColumnResult(name = "price"),
                        @ColumnResult(name = "authorName")}))
public class Book { ... }
```

After you defined the *@SqlResultSetMapping*, you can implement your native query and provide the name of the mapping as the 2nd parameter of the *createNativeQuery* method.

```
Query q = em.createNativeQuery(
                "SELECT b.id, b.title, b.price, a.firstName || ' ' || a.lastName as authorName ”
                + “FROM Book b JOIN Author a ON b.author_id = a.id “
                + “WHERE b.title LIKE :title",
                "BookWithAuthorNamesMapping");
q.setParameter("title", "%Hibernate Tips%");
List<BookWithAuthorNames> books = q.getResultList();
```

DTO projections without a custom DTO class

All of the previous examples used a custom DTO class. That’s a good approach, if you want to send the result to a client or if you process it in a different part of your application.

But it’s also an unnecessary overhead if you only want to execute a query and immediately process the result. In that case, a *Tuple* projection might be the easier option.

JPA's *Tuple* interface provides a generic and comfortable way to access the elements of a query result. You can use it to access the elements by their index or alias, and you can provide additional type information to cast them automatically.

Using the Tuple interface with JPQL

If you want to use the *Tuple* interface in a JPQL query, you need to provide a reference to the interface as the 2nd parameter of the *createQuery* method. And you should also define an alias for each selected entity attribute. You can then provide this alias to the get method of the Tuple interface to retrieve the selected value.

```
TypedQuery<Tuple> q = em.createQuery(
      "SELECT b.id as id, b.title as title, b.price as price, “
      + “concat(a.firstName, ' ', a.lastName) as authorName “
      + “FROM Book b JOIN b.author a WHERE b.title LIKE :title",
      Tuple.class);
q.setParameter("title", "%Hibernate Tips%");
List<Tuple> books = q.getResultList();

for (Tuple b : books) {
    log.info("ID: " + b.get("id"));
    log.info("Title: " + b.get("title"));
    log.info("Price: " + b.get("price"));
    log.info("Author: " + b.get("authorName"));
}
```

Using the Tuple interface with a CriteriaQuery

A *CriteriaQuery* that returns a *Tuple* interface is pretty similar to one that returns a DTO projection. Instead of the DTO class, you now create a query that returns a *Tuple* interface. You can then use the *multiselect* method of the *CriteriaQuery* interface to select multiple entity attributes. If you want to access the elements of your *Tuple* by their alias, you need to specify the alias while selecting them.

```
// Create query
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createQuery(Tuple.class);
// Define FROM clause
Root<Book> root = cq.from(Book.class);
Join<Book, Author> author = root.join(Book_.author);

// Define Tuple projection
cq.multiselect(root.get(Book_.id).alias("id"),
        root.get(Book_.title).alias("title"),
        root.get(Book_.price).alias("price"),
        cb.concat(author.get(Author_.firstName), ' ',
                author.get(Author_.lastName)).alias("authorName"));

// Define WHERE clause
ParameterExpression<String> paramTitle = cb.parameter(String.class);
cq.where(cb.like(root.get(Book_.title), paramTitle));

// Execute query
TypedQuery<Tuple> q = em.createQuery(cq);
q.setParameter(paramTitle, "%Hibernate Tips%");
List<Tuple> books = q.getResultList();
```

Using the Tuple interface with a native SQL query

You don't need to provide an *@SqlResultSetMapping* if you want to use a *Tuple* projection with a native SQL query. You only need to reference the *Tuple* interface as the 2nd parameter of the *createNativeQuery* method. And to make it even better, in contrast to the previously discussed JPQL and Criteria queries, you don't need to provide an alias, if you select a simple database column.

```
Query q = em.createNativeQuery(
    "SELECT b.id, b.title, b.price, a.firstName || ' ' || a.lastName as authorName “
    + “FROM Book b JOIN Author a ON b.author_id = a.id WHERE b.title LIKE :title",
    Tuple.class);
q.setParameter("title", "%Hibernate Tips%");
List<Tuple> books = q.getResultList();
```

Conclusion

DTOs are the most efficient projection for read operations. You should, therefore, use it whenever you don’t need to change the requested data.

As you have learned in this article, you can use DTO projections with all 3 kinds of queries supported by JPA:

- In JPQL, you can use the new keyword to define a constructor expression.
- The *construct* method of the *CriteriaBuilder* interface enables you to specify a constructor expression for a *CriteriaQuery*.
- Native SQL queries require an *@SqlResultSetMapping* to define the mapping of the query result to one or more DTO instances.

If you don’t want to implement a custom DTO class, you can use the Tuple interface with all 3 kinds of queries.