# JPA Repository End-to-End Application

jim stafford

Fall 2022 v2021-03-21: Built: 2022-12-07 06:14 EST

# Table of Contents

# Chapter 1. Introduction

This lecture takes what you have learned in establishing a RDBMS data tier using Spring Data JPA and shows that integrated into an end-to-end application with API CRUD calls and finder calls using paging. It is assumed that you already know about API topics like Data Transfer Objects (DTOs), JSON and XML content, marshalling/unmarshalling using Jackson and JAXB, web APIs/controllers, and clients. This lecture will put them all together.

## 1.1. Goals

The student will learn:

- to integrate a Spring Data JPA Repository into an end-to-end application, accessed through an API
- to make a clear distinction between Data Transfer Objects (DTOs) and Business Objects (BOs)
- to identify data type architectural decisions required for a multi-tiered application
- to understand the need for paging when working with potentially unbounded collections and remote clients
- to setup proper transaction and other container feature boundaries using annotations and injection

## 1.2. Objectives

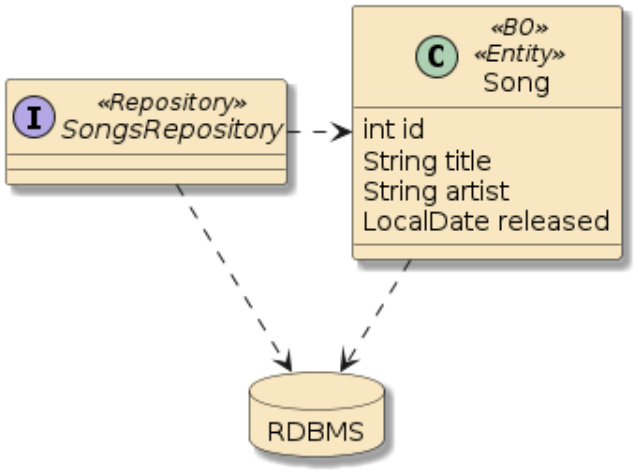
At the conclusion of this lecture and related exercises, the student will be able to:

1. implement a BO tier of classes that will be mapped to the database
2. implement a DTO tier of classes that will exchange state with external clients
3. implement a service tier that completes useful actions
4. identify the controller/service layer interface decisions when it comes to using DTO and BO classes
5. determine the correct transaction propagation property for a service tier method
6. implement a mapping tier between BO and DTO objects
7. implement paging requests through the API
8. implement page responses through the API

# Chapter 2. BO/DTO Component Architecture

## 2.1. Business Object(s)/@Entities

For our Songs application — I have kept the data model simple and kept it limited to a single business object (BO) `@Entity` class mapped to the database using JPA and accessed through a Spring Data JPA repository.




The business objects are the focal point of information where we implement our business decisions.

*Figure 1. BO Class Mapped to DB as JPA @Entity*

The primary focus of our BO classes is to map business implementation concepts to the database.

The following snippet shows some of the required properties of a JPA `@Entity` class.

*BO Class Sample JPA Mappings*

```
@Entity
@Table(name="REPOSONGS_SONG")
@NoArgsConstructor
...
public class Song {
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private int id;
 ...
```

## 2.2. Data Transfer Object(s) (DTOs)

The Data Transfer Objects are the focal point of interfacing with external clients. They represent state at a point in time. For external web APIs, they are commonly mapped to both JSON and XML.

For the API, we have the decision of whether to reuse BO classes as DTOs or implement a separate set of classes for that purpose. Even though some applications start out simple, there will come a point where database technology or mappings will need to change at a different pace than API technology or mappings.

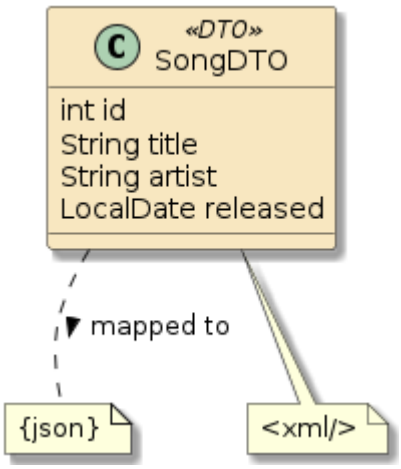



*Figure 2. DTO*

For that reason, I created a separate SongsDTO class to represent a sample DTO. It has a near 1:1 mapping with the `Song` BO. This 1:1 representation of information makes it seem like this is an unnecessary extra class, but it demonstrates an initial technical separation between the DTO and BO that allows for independent changes down the road.

The primary focus of our DTO classes is to map business interface concepts to a portable exchange format.

The following snippet shows some of the annotations required to map the `SongDTO` class to XML using Jackson and JAXB. Jackson JSON requires very few annotations in the simple cases.

*DTO Class Sample JSON/XML Mappings*

```
@JacksonXmlRootElement(localName = "song", namespace = "urn:ejava.db-repo.songs")
@XmlRootElement(name = "song", namespace = "urn:ejava.db-repo.songs") ②
@NoArgsConstructor
...
public class SongDTO { ①
    @JacksonXmlProperty(isAttribute = true)
    @XmlAttribute
    private int id;
```

① Jackson JSON requires very little to no annotations for simple mappings

② XML mappings require more detailed definition to be complete

## 2.3. BO/DTO Mapping

With separate BO and DTO classes, there is a need for mapping between the two.

- map from DTO to BO for requests
- map from BO to DTO for responses

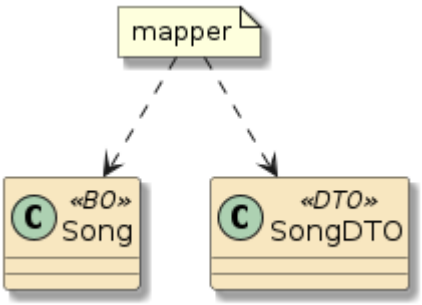



*Figure 3. BO to DTO Mapping*

We have several options on how to organize this role.

### 2.3.1. BO/DTO Self Mapping

- The BO or the DTO class can map to the other
  - Benefit: good encapsulation of detail within the data classes themselves
  - Drawback: promotes coupling between two layers we were trying to isolate

Avoid unless users of DTO will be tied to BO and are just exchanging information.

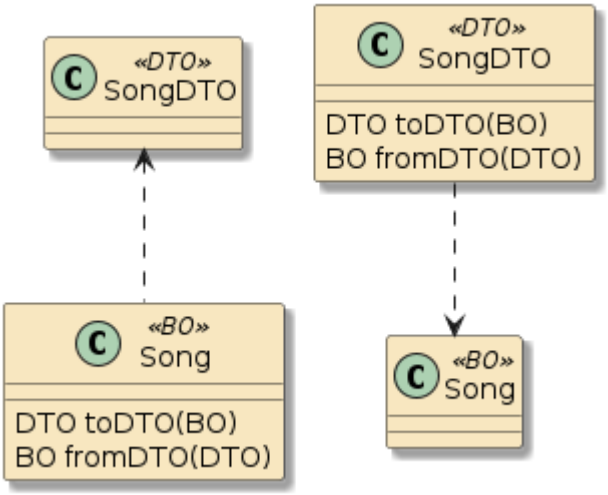



*Figure 4. BO to DTO Self Mapping*

### 2.3.2. BO/DTO Method Self Mapping

- The API or service methods can map things themselves within the body of the code
  - Benefit: mapping specialized to usecase involved
  - Drawback:
    - mixed concerns within methods.
    - likely have repeated mapping code in many methods

Avoid.

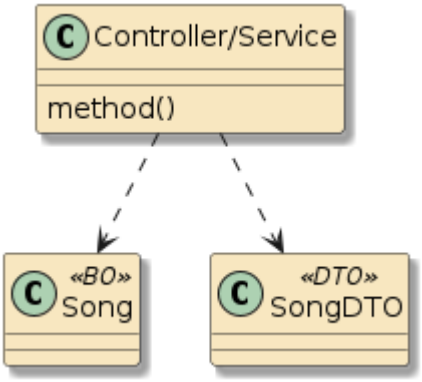



*Figure 5. BO to DTO Method Self Mapping*

### 2.3.3. BO/DTO Helper Method Mapping

- Delegate mapping to a reusable helper method within the API or service classes
  - Benefit: code reuse within the API or service class
  - Drawback: potential for repeated mapping in other classes

This is a small but significant step to a helper class

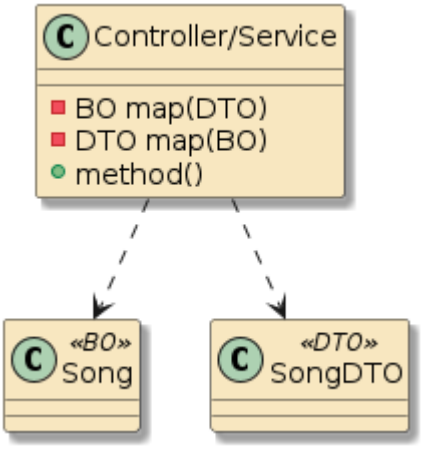



*Figure 6. BO/DTO Helper Method Mapping*

### 2.3.4. BO/DTO Helper Class Mapping

- Create a separate interface/class to inject into the API or service classes that encapsulates the role of mapping
  - Benefit: Reusable, testable, separation of concern
  - Drawback: none

Best in most cases unless good reason for self-mapping is appropriate.

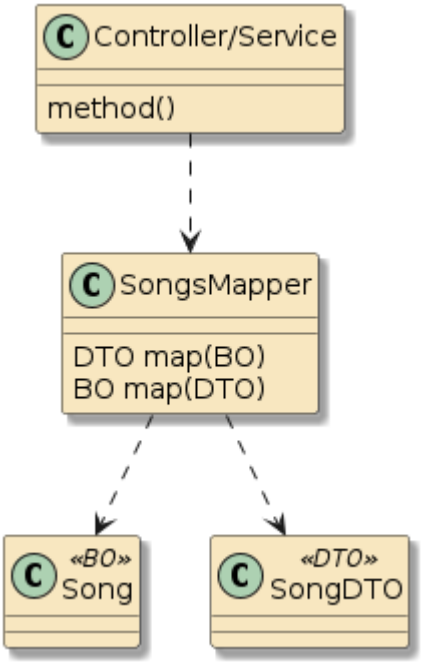



*Figure 7. BO/DTO Helper Class Mapping*

### 2.3.5. BO/DTO Helper Class Mapping Implementations

Mapping helper classes can be implemented by:

- brute force implementation
  - Benefit: likely the fastest performance and technically simplest to understand
  - Drawback: tedious setter/getter code
- off-the-shelf mapper libraries (e.g. Dozer, Orika, MapStruct, ModelMapper, JMapper) [1] [2]
  - Benefit: declarative language and inferred DIY mapping options
  - Drawbacks:
    - relies on reflection and other generalizations for mapping which add to overhead
    - non-trivial mappings can be complex to understand

[1] *"Performance of Java Mapping Frameworks"*, Baeldung

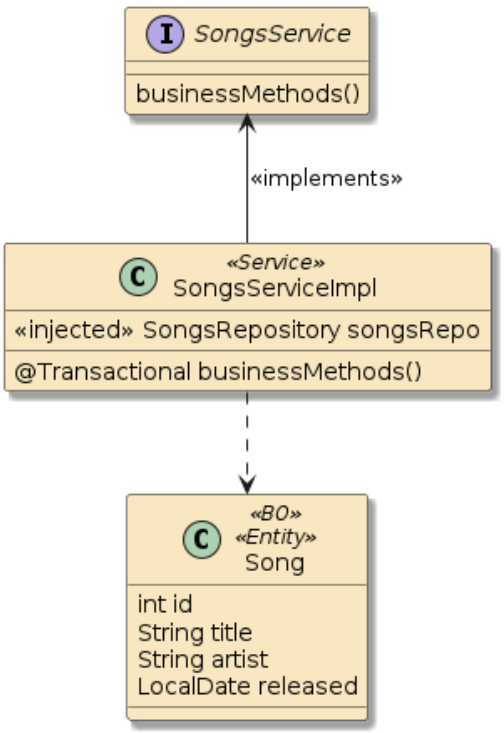
[2] *"any tool for java object to object mapping?"*, Stack Overflow

# Chapter 3. Service Architecture

Services — with the aid of BOs — implement the meat of the business logic.

The service

- implements an interface with business methods
- is annotated with `@Service` component in most cases to self-support auto-injection (or use `@Bean` factory)
- injects repository component(s)
- declares transaction boundaries on methods
- interacts with BO instances




*Example Service Class Declaration*

```
@RequiredArgsConstructor
@Service
public class SongsServiceImpl
     implements SongsService {
  private final SongsMapper mapper;
  private final SongsRepository songsRepo;
  ...
```

## 3.1. Injected Service Boundaries

Container features like `@Transactional`, `@PreAuthorize`, `@Async`, etc. are only implemented at component boundaries. When a `@Component` dependency is injected, the container has the opportunity to add features using *"interpose"*. As a part of interpose — the container implements proxy to add the desired feature of the target component method.

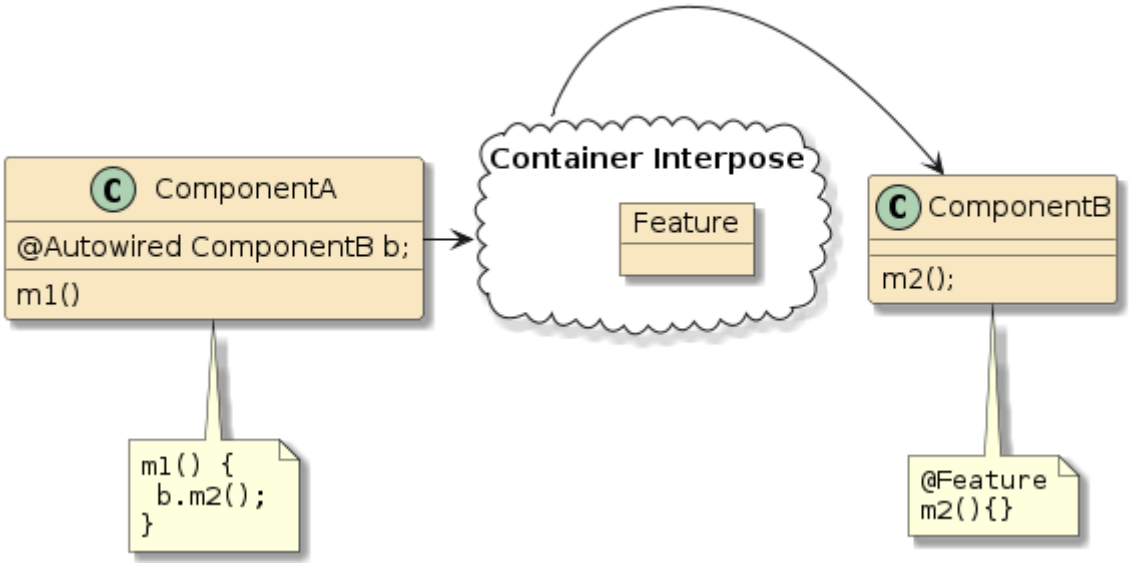



*Figure 8. Container Interpose*

Therefore it is important to arrange a component boundary wherever you need to start a new characteristic provided by the container. The following is a more detailed explanation of what not to do and do.

### 3.1.1. Buddy Method Boundary

The methods within a component class are not typically subject to container interpose. Therefore a call from m1() to m2() within the same component class is a straight Java call.

*No Interpose for Buddy Method Calls*

Buddy method calls are straight Java calls without container interpose.

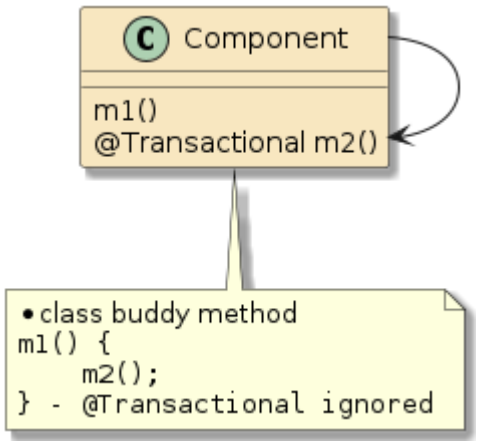



*Figure 9. Buddy Method Boundary*

### 3.1.2. Self Instantiated Method Boundary

Container interpose is only performed when the container has a chance to decorate the called component. Therefore, a call to a method of a component class that is self-instantiated will not have container interpose applied — no matter how the called method is annotated.

*No Interpose for Self-Instantiated Components*

Self-instantiated classes are not subject to container interpose.

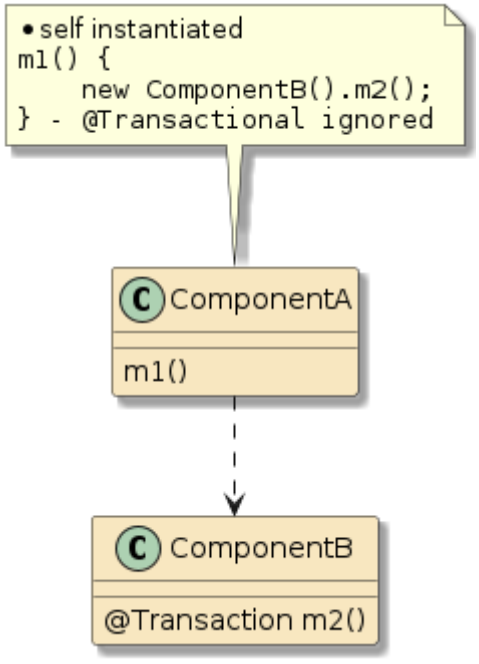



*Figure 10. Self Instantiated Method Boundary*

### 3.1.3. Container Injected Method Boundary

Components injected by the container are subject to container interpose and will have declared characteristics applied.

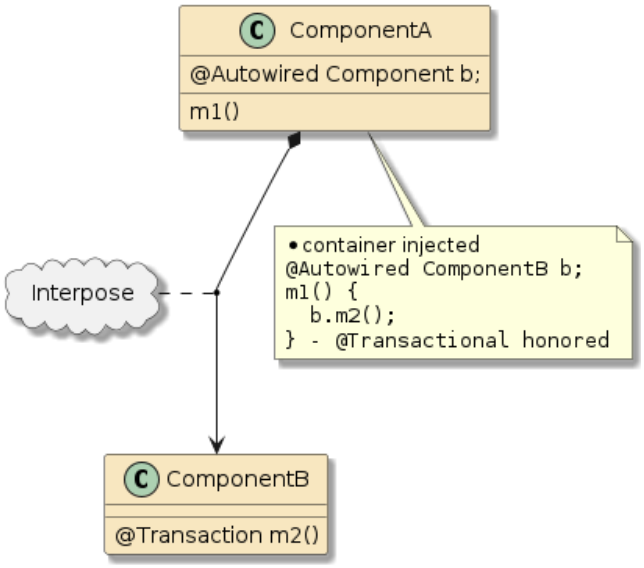



*Figure 11. Container Injected Method Boundary*

> *Container-Injected Components have Interpose*
>
> Use container injection to have declared features applied to called component methods.

## 3.2. Compound Services

With `@Component` boundaries and interpose constraints understood — in more complex transaction, security, or threading solutions, the **logical** `@Service` many get broken up into one or more **physical** helper `@Component` classes.

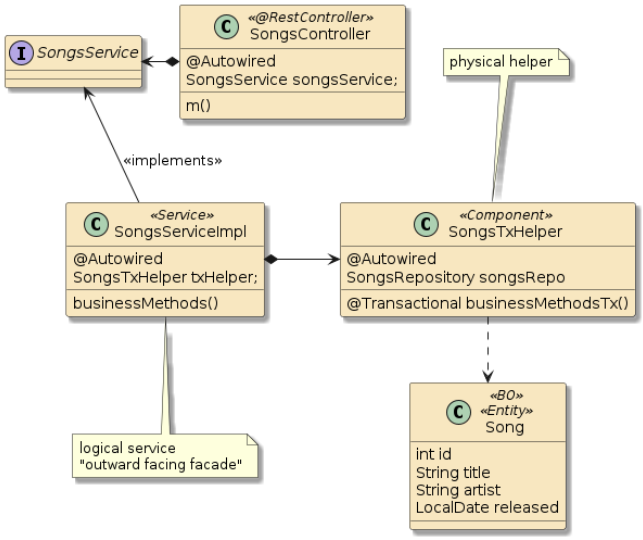



*Figure 12. Single Service Expressed as Multiple Components*

Each physical helper `@Component` is primarily designed around container augmentation (ex. action(s) to be performed within a single `@Transaction`). The remaining parts of the logical service are geared towards implementing the outward facing facade, and integrating the methods of the helper(s) to complete the intended role of the service. An example of this would be large loops of behavior.

```
for (...) { txHelper.txMethod(); }
```

To external users of `@Service` — it is still logically, just one `@Service`.

*Conceptual Services may be broken into Multiple Physical Components*

Conceptual boundaries for a service usually map 1:1 with a single physical class. However, there are cases when the conceptual service needs to be implemented by multiple physical classes/`@Components`.

# Chapter 4. BO/DTO Interface Options

With the core roles of BOs and DTOs understood, we next have a decision to make about where to use them within our application between the API and service classes.

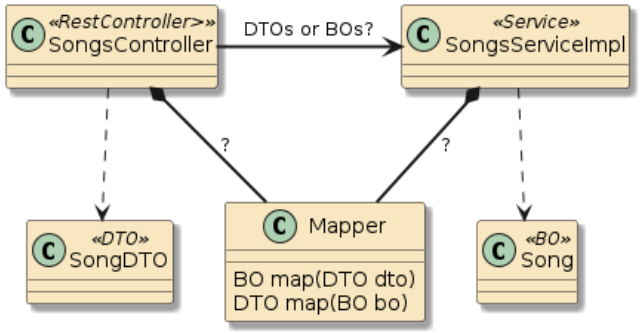



*Figure 13. BO/DTO Interface Decisions*

- Controller external interface will always be based on DTOs.
- Service's internal implementation will always be based on BOs.
- Where do we make the transition?

## 4.1. API Maps DTO/BO

It is natural to think of the `@Service` as working with pure implementation (BO) classes. This leaves the mapping job to the `@RestController` and all clients of the `@Service`.

- Benefit: If we wire two `@Services` together, they could efficiently share the same BO instances between them with no translation.
- Drawback: `@Services` should be the boundary of a solution and encapsulate the implementation details. BOs leak implementation details.

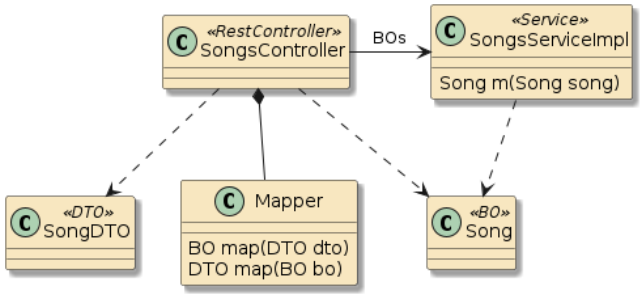



*Figure 14. API Maps DTO to BO for Service Interface*

## 4.2. @Service Maps DTO/BO

Alternatively, we can have the `@Service` fully encapsulate the implementation details and work with DTOs in its interface. This places the job of DTO/BO translation to the `@Service` and the `@RestController` and all `@Service` clients work with DTOs.

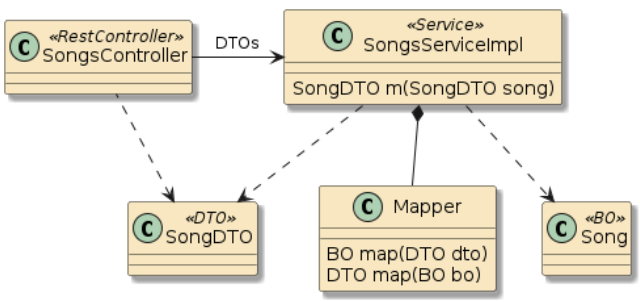
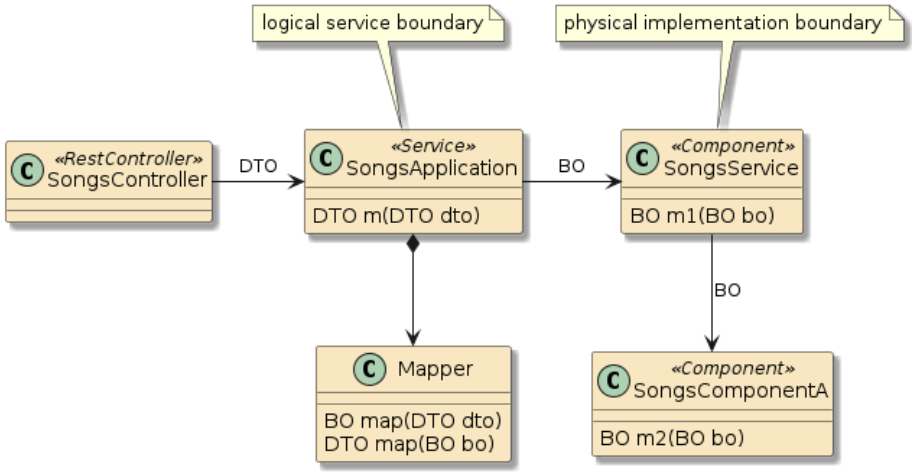



*Figure 15. Service Maps DTO in Service Interface to BO*

- Benefit: `@Service` fully encapsulates implementation and exchanges information using DTOs designed for interfaces.
- Drawback: BOs go through a translation when passing from `@Service` to `@Service` directly.

# 4.3. Layered Service Mapping Approach

The later DTO interface/mapping approach just introduced — maps closely to the Domain Driven Design (DDD) "Application Layer". However, one could also implement a layering of services.




- outer `@Service` classes represent the boundary to the application and interface using DTOs
- inner `@Component` classes represent implementation components and interface using native BOs

*Layered Services Permit a Level of Trust between Inner Components*

When using this approach, I like:

- all normalization and validation complete by the time DTOs are converted to BOs in the Application tier
- BOs exchanged between implementation components assume values are valid and normalized

# Chapter 5. Implementation Details

With architectural decisions understood, lets take a look at some of the key details of the end-to-end application.

## 5.1. Song BO

We have already covered the `Song` BO `@Entity` class in a lot of detail during the JDBC, JPA, and Spring Data JPA lectures. The following lists most of the key business aspects and implementation details of the class.

*Song BO Class with JPA Database Mappings*

```
package info.ejava.examples.db.repo.jpa.songs.bo;
...
@Entity
@Table(name="REPOSONGS_SONG")
@Getter
@ToString
@Builder
@With
@AllArgsConstructor
@NoArgsConstructor
...
public class Song {
    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @Column(name="ID", nullable=false, insertable=true, updatable=false)
    private int id;
    @Setter
    @Column(name="TITLE", length=255, nullable=true, insertable=true, updatable=true)
    private String title;
    @Setter
    private String artist;
    @Setter
    private LocalDate released;
}
```

## 5.2. SongDTO

The SongDTO class has been mapped to Jackson JSON and Jackson and JAXB XML. The details of Jackson and JAXB mapping were covered in the API Content lectures. Jackson JSON required no special annotations to map this class. Jackson and JAXB XML primarily needed some annotations related to namespaces and attribute mapping. JAXB also required annotations for mapping the LocalDate field.

The following lists the annotations required to marshal/unmarshal the SongsDTO class using Jackson and JAXB.

*SongDTO Class with JSON and XML Mappings*

```
package info.ejava.examples.db.repo.jpa.songs.dto;
...
@JacksonXmlRootElement(localName = "song", namespace = "urn:ejava.db-repo.songs")
@XmlRootElement(name = "song", namespace = "urn:ejava.db-repo.songs")
@XmlAccessorType(XmlAccessType.FIELD)
@Data @Builder
@NoArgsConstructor @AllArgsConstructor
public class SongDTO {
    @JacksonXmlProperty(isAttribute = true)
    @XmlAttribute
    private int id;
    private String title;
    private String artist;
    @XmlJavaTypeAdapter(LocalDateJaxbAdapter.class) ①
    private LocalDate released;
 ...
}
```

① JAXB requires an adapter for the newer LocalDate java class

### 5.2.1. LocalDateJaxbAdapter

Jackson is configured to marshal LocalDate out of the box using the ISO_LOCAL_DATE format for both JSON and XML.

*ISO_LOCAL_DATE format*

```
"released" : "2013-01-30"     //Jackson JSON
<released xmlns="">2013-01-30</released> //Jackson XML
```

JAXB does not have a default format and requires the class be mapped to/from a string using an `XmlAdapter`.

*LocalDateJaxbAdapter Class*

```
@XmlJavaTypeAdapter(LocalDateJaxbAdapter.class)
private LocalDate released;

public static class LocalDateJaxbAdapter extends XmlAdapter<String, LocalDate> {
    @Override
    public LocalDate unmarshal(String text) {
        return LocalDate.parse(text, DateTimeFormatter.ISO_LOCAL_DATE);
    }
    @Override
    public String marshal(LocalDate timestamp) {
        return DateTimeFormatter.ISO_LOCAL_DATE.format(timestamp);
    }
```

```
}
```

## 5.3. Song JSON Rendering

The following snippet provides example JSON of a `Song` DTO payload.

*Song JSON Rendering*

```
{
  "id" : 1,
  "title" : "Tender Is the Night",
  "artist" : "No Doubt",
  "released" : "2003-11-16"
}
```

## 5.4. Song XML Rendering

The following snippets provide example XML of `Song` DTO payloads. They are technically equivalent from an XML Schema standpoint, but use some alternate syntax XML to achieve the same technical goals.

*Song Jackson XML Rendering*

```
<song xmlns="urn:ejava.db-repo.songs" id="2">
  <title xmlns="">The Mirror Crack'd from Side to Side</title>
  <artist xmlns="">Earth Wind and Fire</artist>
  <released xmlns="">2018-01-01</released>
</song>
```

*Song JAXB XML Rendering*

```
<ns2:song xmlns:ns2="urn:ejava.db-repo.songs" id="1">
    <title>Brandy of the Damned</title>
    <artist>Orbital</artist>
    <released>2015-11-10</released>
</ns2:song>
```

## 5.5. Pageable/PageableDTO

I placed a high value on paging when working with unbounded collections when covering repository find methods. The value of paging comes especially into play when dealing with external users. That means we will need a way to represent Page, Pageable, and Sort in requests and responses as a part of DTO solution.

You will notice that I made a few decisions on how to implement this interface

1. I am assuming that both sides of the interface using the DTO classes are using Spring Data. The DTO classes have a direct dependency on their non-DTO siblings.
2. I am using the Page, Pageable, and Sort DTOs to directly self-map to/from Spring Data types. This makes the client and service code much simpler.

```
Pageable pageable = PageableDTO.of(pageNumber, pageSize, sortString).toPageable();
①
Page<SongDTO> result = ...
SongsPageDTO resultDTO = new SongsPageDTO(result); ①
```

① using self-mapping between paging DTOs and Spring Data (`Pageable` and `Page`) types

3. I chose to use the Spring Data types (`Pageable` and `Page`) in the `@Service` interface when expressing paging and performed the Spring Data/DTO mappings in the `@RestController`. The `@Service` still takes DTO business types and maps DTO business types to/from BOs. I did this so that I did not eliminate any pre-existing library integration with Spring Data paging types.

```
Page<SongDTO> getSongs(Pageable pageable); ①
```

① using Spring Data (`Pageable` and `Page`) and business DTO (`SongDTO`) types in `@Service` interface

I will be going through the architecture and wiring in these lecture notes. The actual DTO code is surprisingly complex to render in the different formats and libraries. These topics were covered in detail in the API content lectures. I also chose to implement the PageableDTO and sort as immutable — which added some interesting mapping challenges worth inspecting.

### 5.5.1. PageableDTO Request

Requests require an expression for Pageable. The most straight forward way to accomplish this is through query parameters. The example snippet below shows pageNumber, pageSize, and sort expressed as simple string values as part of the URI. We have to write code to express and parse that data.

*Example Pageable Query Parameters*

```
①
/api/songs/example?pageNumber=0&pageSize=5&sort=released:DESC,id:ASC
②
```

① `pageNumber` and `pageSize` are direct properties used by `PageRequest`

② `sort` contains a comma separated list of order compressed into a single string

Integer `pageNumber` and `pageSize` are straight forward to represent as numeric values in the query. Sort requires a minor amount of work. Spring Data Sort is an ordered list of "property and direction". I have chosen to express property and direction using a ":" separated string and concatenate the ordering using a ",". This allows the query string to be expressed in the URI without special characters.

### 5.5.2. PageableDTO Client-side Request Mapping

Since I expect code using the PageableDTO to also be using Spring Data, I chose to use self-mapping between the PageableDTO and Spring Data Pageable.

The following snippet shows how to map `Pageable` to PageableDTO and the PageableDTO properties to URI query parameters.

*Building URI with Pageable Request Parameters*

```
PageRequest pageable = PageRequest.of(0, 5,
    Sort.by(Sort.Order.desc("released"), Sort.Order.asc("id")));
PageableDTO pageSpec = PageableDTO.of(pageable); ①
URI uri=UriComponentsBuilder
   .fromUri(serverConfig.getBaseUrl())
   .path(SongsController.SONGS_PATH).path("/example")
   .queryParams(pageSpec.getQueryParams()) ②
   .build().toUri();
```

① using PageableDTO to self map from Pageable

② using PageableDTO to self map to URI query parameters

### 5.5.3. PageableDTO Server-side Request Mapping

The following snippet shows how the individual page request properties can be used to build a local instance of PageableDTO in the `@RestController`. Once the PageableDTO is built, we can use that to self map to a Spring Data `Pageable` to be used when calling the `@Service`.

```
public ResponseEntity<SongsPageDTO> findSongsByExample(
        @RequestParam(value="pageNumber",defaultValue="0",required=false) Integer
pageNumber,
        @RequestParam(value="pageSize",required=false) Integer pageSize,
        @RequestParam(value="sort",required=false) String sortString,
        @RequestBody SongDTO probe) {

    Pageable pageable = PageableDTO.of(pageNumber, pageSize, sortString) ①
                                   .toPageable(); ②
```

① building PageableDTO from page request properties

② using PageableDTO to self map to Spring Data Pageable

### 5.5.4. Pageable Response

Responses require an expression for Pageable to indicate the pageable properties about the content returned. This must be expressed in the payload, so we need a JSON and XML expression for this. The snippets below show the JSON and XML DTO renderings of our Pageable properties.

*Example JSON Pageable Response Document*

```
  "pageable" : {
    "pageNumber" : 1,
    "pageSize" : 25,
    "sort" : "title:ASC,artist:ASC"
  }
```

*Example XML Pageable Response Document*

```
<pageable xmlns="urn:ejava.common.dto" pageNumber="1" pageSize="25" sort=
"title:ASC,artist:ASC"/>
```

# 5.6. Page/PageDTO

`Pageable` is part of the overall `Page<T>`, with contents. Therefore, we also need a way to return a page of content to the caller.

## 5.6.1. PageDTO Rendering

JSON is very lenient and could have been implemented with a generic `PageDTO<T>` class.

```
{"content":[ ①
  {"id":10, ②
   "title":"Blue Remembered Earth",
   "artist":"Coldplay",
   "released":"2009-03-18"}],
 "totalElements":10, ①
 "pageable":{"pageNumber":3,"pageSize":3,"sort":null} ①
}
```

① `content`, `totalElements`, and `pageable` are part of reusable PageDTO

② `song` within `content` array is part of concrete Songs domain

However, XML — with its use of unique namespaces, requires a sub-class to provide the type-specific values for content and overall page.

```
<songsPage xmlns="urn:ejava.db-repo.songs" totalElements="10"> ①
  <wstxns1:content xmlns:wstxns1="urn:ejava.common.dto">
    <song id="10"> ②
      <title xmlns="">Blue Remembered Earth</title>
      <artist xmlns="">Coldplay</artist>
      <released xmlns="">2009-03-18</released>
    </song>
  </wstxns1:content>
  <pageable xmlns="urn:ejava.common.dto" pageNumber="3" pageSize="3"/>
```

```
</songsPage>
```

① `totalElements` mapped to XML as an (optional) attribute

② `songsPage` and `song` are in concrete domain `urn:ejava.db-repo.songs` namespace

### 5.6.2. SongsPageDTO Subclass Mapping

The `SongsPageDTO` subclass provides the type-specific mapping for the content and overall page. The generic portions are handled by the base class.

*SongsPageDTO Subclass Mapping*

```
@JacksonXmlRootElement(localName = "songsPage", namespace = "urn:ejava.db-repo.songs")
①
@XmlRootElement(name = "songsPage", namespace = "urn:ejava.db-repo.songs") ①
@XmlType(name = "SongsPage", namespace = "urn:ejava.db-repo.songs")
@XmlAccessorType(XmlAccessType.NONE)
@NoArgsConstructor
public class SongsPageDTO extends PageDTO<SongDTO> {
    @JsonProperty
    @JacksonXmlElementWrapper(localName = "content", namespace = "
urn:ejava.common.dto")②
    @JacksonXmlProperty(localName = "song", namespace = "urn:ejava.db-repo.songs") ③
    @XmlElementWrapper(name="content", namespace = "urn:ejava.common.dto") ②
    @XmlElement(name="song", namespace = "urn:ejava.db-repo.songs") ③
    public List<SongDTO> getContent() {
        return super.getContent();
    }
    public SongsPageDTO(List<SongDTO> content, Long totalElements, PageableDTO
pageableDTO) {
        super(content, totalElements, pageableDTO);
    }
    public SongsPageDTO(Page<SongDTO> page) {
        this(page.getContent(), page.getTotalElements(),
             PageableDTO.fromPageable(page.getPageable()));
    }
}
```

① Each type-specific mapping must have its own XML naming

② "Wrapper" is the outer element for the individual members of collection and part of generic framework

③ "Property/Element" is the individual members of collection and interface/type specific

### 5.6.3. PageDTO Server-side Rendering Response Mapping

The `@RestController` can use the concrete DTO class (SongPageDTO in this case) to self-map from a Spring Data `Page<T>` to a DTO suitable for marshaling back to the API client.

*PageDTO Server-side Response Mapping*

```
Page<SongDTO> result=songsService.findSongsMatchingAll(probe, pageable);

SongsPageDTO resultDTO = new SongsPageDTO(result); ①
ResponseEntity<SongsPageDTO> response = ResponseEntity.ok(resultDTO);
```

① using SongsPageDTO to self-map Sing Data `Page<T>` to DTO

### 5.6.4. PageDTO Client-side Rendering Response Mapping

The `PageDTO<T>` class can be used to self-map to a Spring Data `Page<T>`. Pageable, if needed, can be obtained from the `Page<T>` or through the `pageDTO.getPageable()` DTO result.

*PageDTO Client-side Response Mapping*

```
SongsPageDTO pageDTO = request.exchange()
        .expectStatus().isOk()
        .returnResult(SongsPageDTO.class)
        .getResponseBody().blockFirst();
Page<SongDTO> page = pageDTO.toPage(); ①
Pageable pageable = ... ②
```

① using `PageDTO<T>` to self-map to a Spring Data `Page<T>`

② can use `page.getPageable()` or `pageDTO.getPageable().toPageable()` obtain `Pageable`

# Chapter 6. SongMapper

The `SongMapper` `@Component` class is used to map between `SongDTO` and `Song` BO instances. It leverages Lombok builder methods — but is pretty much a simple/brute force mapping.

## 6.1. Example Map: SongDTO to Song BO

The following snippet is an example of mapping a `SongDTO` to a `Song` BO.

*Map SongDTO to Song BO*

```
@Component
public class SongsMapper {
    public Song map(SongDTO dto) {
        Song bo = null;
        if (dto!=null) {
            bo = Song.builder()
                    .id(dto.getId())
                    .artist(dto.getArtist())
                    .title(dto.getTitle())
                    .released(dto.getReleased())
                    .build();
        }
        return bo;
    }
    ...
```

## 6.2. Example Map: Song BO to SongDTO

The following snippet is an example of mapping a `Song` BO to a `SongDTO`.

*Map Song BO to SongDTO*

```
    ...
    public SongDTO map(Song bo) {
        SongDTO dto = null;
        if (bo!=null) {
            dto = SongDTO.builder()
                    .id(bo.getId())
                    .artist(bo.getArtist())
                    .title(bo.getTitle())
                    .released(bo.getReleased())
                    .build();
        }
        return dto;
    }
    ...
```

# Chapter 7. Service Tier

The SongsService `@Service` encapsulates the implementation of our management of Songs.

## 7.1. SongsService Interface

The `SongsService` interface defines a portion of pure CRUD methods and a series of finder methods. To be consistent with DDD encapsulation, the `@Service` interface is using DTO classes. Since the `@Service` is an injectable component, I chose to use straight Spring Data pageable types to possibly integrate with libraries that inherently work with Spring Data types.

*SongsService Interface*

```
public interface SongsService {
    SongDTO createSong(SongDTO songDTO); ①
    SongDTO getSong(int id);
    void updateSong(int id, SongDTO songDTO);
    void deleteSong(int id);
    void deleteAllSongs();

    Page<SongDTO> findReleasedAfter(LocalDate exclusive, Pageable pageable);②
    Page<SongDTO> findSongsMatchingAll(SongDTO probe, Pageable pageable);
}
```

① chose to use DTOs for business data (`SongDTO`) in `@Service` interface

② chose to use Spring Data types (`Page` and `Pageable`) in pageable `@Service` finder methods

## 7.2. SongsServiceImpl Class

The `SongsServiceImpl` implementation class is implemented using the `SongsRepository` and `SongsMapper`.

*SongsServiceImpl Implementation Attributes*

```
@RequiredArgsConstructor ① ②
@Service
public class SongsServiceImpl implements SongsService {
    private final SongsMapper mapper;
    private final SongsRepository songsRepo;
```

① Creates a constructor for all final attributes

② Single constructors are automatically used for Autowiring

I will demonstrate two types of methods here — one requiring an active transaction and the other that only supports but does not require a transaction.

## 7.3. createSong()

The `createSong()` method

- accepts a `SongDTO`, creates a new song, and returns the created song as a `SongDTO`, with the generated ID.
- declares a `@Transaction` annotation to be associated with a Persistence Context and propagation `REQUIRED` in order to enforce that a database transaction be active from this point forward.
- calls the mapper to map from/to a `SongsDTO` to/from a `Song` BO
- uses the `SongsRepository` to interact with the database

*SongsServiceImpl.createSong()*

```
@Transactional(propagation = Propagation.REQUIRED) ① ② ③
public SongDTO createSong(SongDTO songDTO) {
    Song songBO = mapper.map(songDTO); ④

    //manage instance
    songsRepo.save(songBO); ⑤

    return mapper.map(songBO); ⑥
}
```

① `@Transaction` associates Persistence Context with thread of call

② `propagation` used to control activation and scope of transaction

③ `REQUIRED` triggers the transaction to start no later than this method

④ mapper converting DTO input argument to BO instance

⑤ BO instance saved to database and updated with primary key

⑥ mapper converting BO entity to DTO instance for return from service

## 7.4. findSongsMatchingAll()

The `findSongsMatchingAll()` method

- accepts a `SongDTO` as a probe and `Pageable` to adjust the search and results
- declares a `@Transaction` annotation to be associated with a Persistence Context and propagation `SUPPORTS` to indicate that no database changes will be performed by this method.
- calls the mapper to map from/to a `SongsDTO` to/from a `Song` BO
- uses the `SongsRepository` to interact with the database

*SongsServiceImpl Finder Method*

```
@Transactional(propagation = Propagation.SUPPORTS) ① ② ③
public Page<SongDTO> findSongsMatchingAll(SongDTO probeDTO, Pageable pageable) {
    Song probe = mapper.map(probeDTO); ④
```

```
    ExampleMatcher matcher = ExampleMatcher.matchingAll().withIgnorePaths("id"); ⑤
    Page<Song> songs = songsRepo.findAll(Example.of(probe, matcher), pageable); ⑥
    return mapper.map(songs); ⑦
}
```

① `@Transaction` associates Persistence Context with thread of call

② `propagation` used to control activation and scope of transaction

③ `SUPPORTS` triggers the any active transaction to be inherited by this method but does not proactively start one

④ mapper converting DTO input argument to BO instance to create probe for match

⑤ building matching rules to include an ignore of `id` property

⑥ finder method invoked with matching and paging arguments to return page of BOs

⑦ mapper converting page of BOs to page of DTOs

# Chapter 8. RestController API

The `@RestController` provides an HTTP Facade for our `@Service`.

*@RestController Class*

```
@RestController
@Slf4j
@RequiredArgsConstructor
public class SongsController {
    public static final String SONGS_PATH="api/songs";
    public static final String SONG_PATH= SONGS_PATH + "/{id}";
    public static final String RANDOM_SONG_PATH= SONGS_PATH + "/random";

    private final SongsService songsService; ①
```

① `@Service` injected into class using constructor injection

I will demonstrate two of the operations available.

## 8.1. createSong()

The `createSong()` operation

- is called using `POST /api/songs` method and URI
- passed a SongDTO, containing the fields to use marshaled in JSON or XML
- calls the `@Service` to handle the details of creating the Song
- returns the created song using a SongDTO

*createSong() API Operation*

```
@RequestMapping(path=SONGS_PATH,
        method=RequestMethod.POST,
        consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<SongDTO> createSong(@RequestBody SongDTO songDTO) {

    SongDTO result = songsService.createSong(songDTO); ①

    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()
            .replacePath(SONG_PATH)
            .build(result.getId()); ②
    ResponseEntity<SongDTO> response = ResponseEntity.created(uri).body(result);
    return response; ③
}
```

① DTO from HTTP Request supplied to and result DTO returned from `@Service` method

② URI of created instance calculated for `Location` response header

③ DTO marshalled back to caller with HTTP Response

## 8.2. findSongsByExample()

The `findSongsByExample()` operation

- is called using "POST /api/songs/example" method and URI
- passed a SongDTO containing the properties to search for using JSON or XML
- calls the `@Service` to handle the details of finding the songs after mapping the `Pageable` from query parameters
- converts the `Page<SongDTO>` into a `SongsPageDTO` to address marshaling concerns relative to XML
- returns the page as a `SongsPageDTO`

*findSongsByExample API Operation*

```
@RequestMapping(path=SONGS_PATH + "/example",
        method=RequestMethod.POST,
        consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<SongsPageDTO> findSongsByExample(
        @RequestParam(value="pageNumber",defaultValue="0",required=false) Integer
pageNumber,
        @RequestParam(value="pageSize",required=false) Integer pageSize,
        @RequestParam(value="sort",required=false) String sortString,
        @RequestBody SongDTO probe) {

    Pageable pageable=PageableDTO.of(pageNumber, pageSize, sortString).toPageable();①
    Page<SongDTO> result=songsService.findSongsMatchingAll(probe, pageable); ②

    SongsPageDTO resultDTO = new SongsPageDTO(result); ③
    ResponseEntity<SongsPageDTO> response = ResponseEntity.ok(resultDTO);
    return response;
}
```

① `PageableDTO` constructed from page request query parameters

② `@Service` accepts DTO arguments for call and returns DTO constructs mixed with Spring Data paging types

③ type-specific `SongsPageDTO` marshalled back to caller to support type-specific XML namespaces

## 8.3. WebClient Example

The following snippet shows an example of using a WebClient to request a page of finder results form the API. WebClient is part of the Spring WebFlux libraries — which implements reactive streams. The use of WebClient here is purely for example and not a requirement of anything created. However, using WebClient did force my hand to add JAXB to the DTO mappings since

Jackson XML is not yet supported by WebFlux. RestTemplate does support both Jackson and JAXB XML mapping - which would have made mapping simpler.

*WebClient Client*

```
@Autowired
private WebClient webClient;
...
UriComponentsBuilder findByExampleUriBuilder = UriComponentsBuilder
        .fromUri(serverConfig.getBaseUrl())
        .path(SongsController.SONGS_PATH).path("/example");
...
//given
MediaType mediaType = ...
PageRequest pageable = PageRequest.of(0, 5, Sort.by(Sort.Order.desc("released")));
PageableDTO pageSpec = PageableDTO.of(pageable); ①
SongDTO allSongsProbe = SongDTO.builder().build(); ②
URI uri = findByExampleUriBuilder.queryParams(pageSpec.getQueryParams()) ③
                                 .build().toUri();
WebClient.RequestHeadersSpec<?> request = webClient.post()
        .uri(uri)
        .contentType(mediaType)
        .body(Mono.just(allSongsProbe), SongDTO.class)
        .accept(mediaType);
//when
ResponseEntity<SongsPageDTO> response = request
        .retrieve()
        .toEntity(SongsPageDTO.class).block();
//then
then(response.getStatusCode().is2xxSuccessful()).isTrue();
SongsPageDTO page = response.getBody();
```

① limiting query rsults to first page, ordered by "release", with a page size of 5

② create a "match everything" probe

③ pageable properties added as query parameters

*WebClient/WebFlex does not yet support Jackson XML*

WebClient and WebFlex does not yet support Jackson XML. This is what primarily forced the example to leverage JAXB for XML. WebClient/WebFlux automatically makes the decision/transition under the covers once an `@XmlRootElement` is provided.

# Chapter 9. Summary

In this module we learned:

- to integrate a Spring Data JPA Repository into an end-to-end application, accessed through an API
- implement a service tier that completes useful actions
- to make a clear distinction between DTOs and BOs
- to identify data type architectural decisions required for DTO and BO types
- to setup proper transaction and other container feature boundaries using annotations and injection
- implement paging requests through the API
- implement page responses through the API