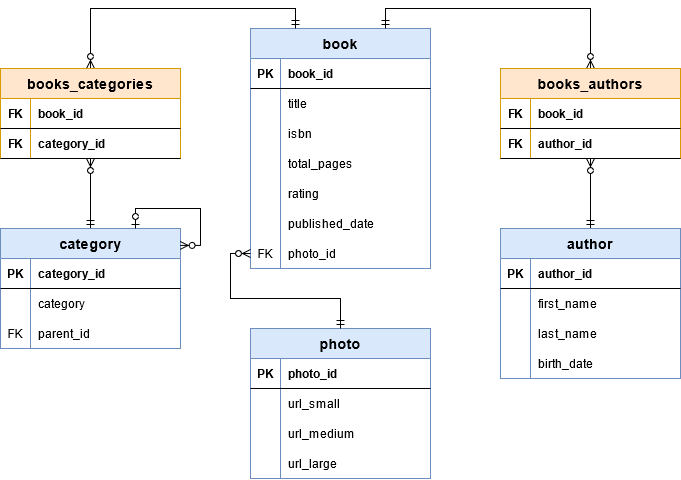
<https://medium.com/huawei-developers/database-relationships-in-spring-data-jpa-8d7181f50f60>

Database Relationships in Spring Data JPA  
**Introduction**

I created an example ER diagram for book database. I will try to explain the relations by including the scenario in this diagram in the project.



In this diagram, there is a **Many-to-Many** relationship between books and categories. A book can have more than one category, and a category can have more than one book. The same is true for the relationship between the book and the author.

Categories can have subcategories. These subcategories must be linked to a parent category. In this case, the **One-to-Many** relationship emerges.

There is also a photo table attached to the books. This table keeps the cover photos of the related book in different sizes. The relationship here will be **One-to-One**.

**OneToOne Relation**

*Relationship between Book and Photo is an example to****OneToOne****relation.*

For that relation, we need to create a foreign key for **photo\_id**parameter in Book table. **OneToOne** annotation will create this relation for us. We should add this annotation to both entity classes that’s because this is a bidirectional relationship.

We want to create a column in Book table. So, we will use **JoinColumn**annotation in Book class. On the Photo class, simply put **OneToOne** annotation and fill**mappedBy**field by variable name on the owning side. In this example, this value will be ‘photo’.

@Entity

public class Book {

...

*@OneToOne(cascade = CascadeType.ALL)*

*@JoinColumn(name = "photo\_id")*

*private Photo* ***photo****;*

}

...

@Entity

public class **Photo** {

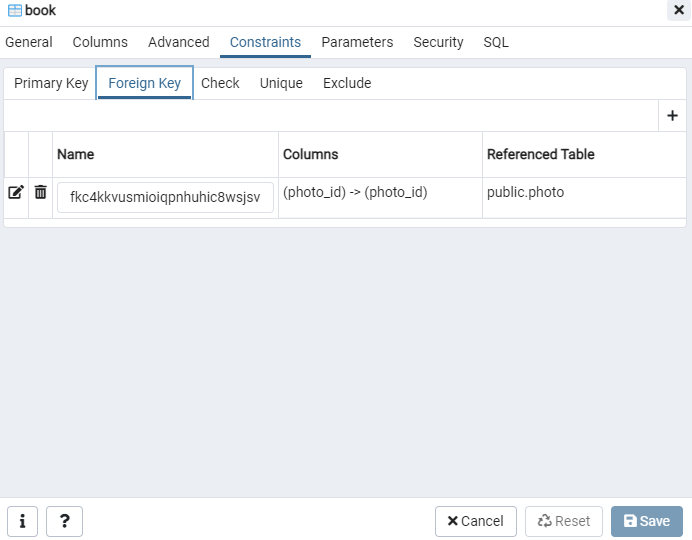
...

*@OneToOne(mappedBy = "****photo****")*

*private Book book;*

}

Rerun the program and you will see the **photo\_id** in columns and the **foreign key** in the properties of the Book table.

  
  
  
  
**OneToMany Relation**

Relationship between parent category and children categories in Category table is an example to **OneToMany**relation.

In this case, both parent and children tables are same, we will add both **OneToMany** and **ManyToOne** annotations to same class. We should join a column for **parent\_id**.

@Data

@Entity

@Table(name = "category")

public class Category {

...

*@ManyToOne(cascade = CascadeType.ALL)*

*@JoinColumn (name = "parent\_id")*

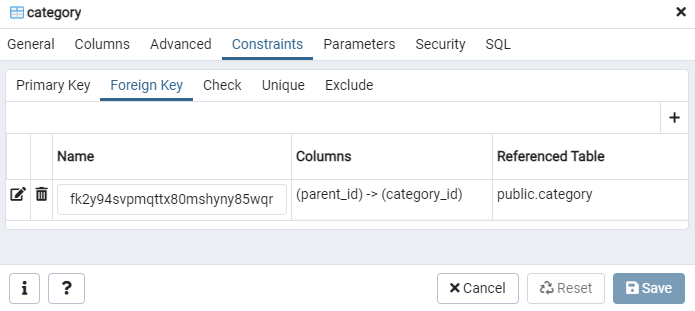
*private Category parent;*

*@OneToMany (mappedBy = "parent")*

*private Collection<Category> children;*

}

After rerunning the program, we will see a foreign key for **parent\_id**in properties of Category table.



**ManyToMany Relation**

Relationship between book and category is an example to **ManyToMany** relation.

In this relation, we need to create a new table to handle **ManyToMany** relationship. This new table will hold foreign keys for both **book\_id** and **category\_id** fields. We will follow the relationship between books and categories by this table.

...

@Entity

public class Book {

...

*@ManyToMany(cascade = CascadeType.ALL)*

*@JoinTable(*

*name = "books\_categories",*

*joinColumns = @JoinColumn(name = "book\_id"),*

*inverseJoinColumns = @JoinColumn(name = "category\_id")*

*)*

*private Collection<Category> categories;*

}

...

@Entity

public class Category {

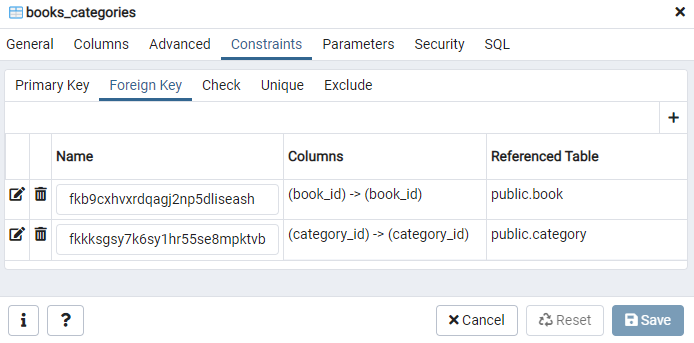
...

*@ManyToMany(mappedBy = "categories")*

*private Collection<Book> books;*

}

After running the code, a new table called **books\_categories**will be exist and specified foreign keys will be shown in properties of this new table.



We need same relation for book and author. After adding that, we can test our database.

**Testing**

## https://miro.medium.com/max/700/1*CmRgtP5qJ-9X9jx8MmEWOA.png **Stored Procedure Creation a stored procedure can have parameters** so that we can get different results based on the input. For example, we can create a stored procedure that takes an input parameter of integer type and returns a list of cars:

**CREATE** **PROCEDURE** ***FIND\_CARS\_AFTER\_YEAR***(**IN** year\_in **INT**)

**BEGIN**

**SELECT** \* **FROM** car **WHERE** **year** >= year\_in **ORDER** **BY** **year**;

**END**

**Reference Stored Procedures in Repository   
In Spring Data JPA, repositories are where we provide database operations.** We can construct a repository for the database operations on the *Car* entity, and reference stored procedures in this repository:

@Repository

**public** **interface** **CarRepository** **extends** **JpaRepository**<Car, Integer> {

}

***Map a Stored Procedure Name Directly***

**We can define a stored procedure method using the *@Procedure*annotation, and map the stored procedure name directly.**

There are four equivalent ways to do that. For example, we can use the stored procedure name directly as the method name:

@Procedure

**int** **GET\_TOTAL\_CARS\_BY\_MODEL**(String model);

If we want to define a different method name, we can put the stored procedure name as the element of the *@Procedure*annotation:

@Procedure("GET\_TOTAL\_CARS\_BY\_MODEL")

**int** **getTotalCarsByModel**(String model);

**We can also use the *procedureName* attribute to map the stored procedure name:**

@Procedure(procedureName = "GET\_TOTAL\_CARS\_BY\_MODEL")

**int** **getTotalCarsByModelProcedureName**(String model);

Finally, we can use the *value* attribute to map the stored procedure name:

@Procedure(value = "GET\_TOTAL\_CARS\_BY\_MODEL")

**int** **getTotalCarsByModelValue**(String model);

***Reference a Stored Procedure Defined in Entity***

**We can also use the *@NamedStoredProcedureQuery* annotation to define a stored procedure in the entity class:**

@Entity

@NamedStoredProcedureQuery(name = "Car.getTotalCardsbyModelEntity",

procedureName = "GET\_TOTAL\_CARS\_BY\_MODEL", parameters = {

@StoredProcedureParameter(mode = ParameterMode.IN, name = "model\_in", type = String.class),

@StoredProcedureParameter(mode = ParameterMode.OUT, name = "count\_out", type = Integer.class)})

**public** **class** **Car** {

// class definition

}

Then we can reference this definition in the repository:

@Procedure(name = "Car.getTotalCardsbyModelEntity")

**int** **getTotalCarsByModelEntiy**(@Param("model\_in") String model);

**We use the *name* attribute to reference the stored procedure defined in the entity class.** For the repository method, we use *@Param*to match the input parameter of the stored procedure. We also match the output parameter of the stored procedure to the return value of the repository method.

***Reference a Stored Procedure With the @Query Annotation***

We can also call a stored procedure directly with the *@Query* annotation:

@Query(value = "CALL FIND\_CARS\_AFTER\_YEAR(:year\_in);", nativeQuery = true)

List<Car> **findCarsAfterYear**(@Param("year\_in") Integer year\_in);

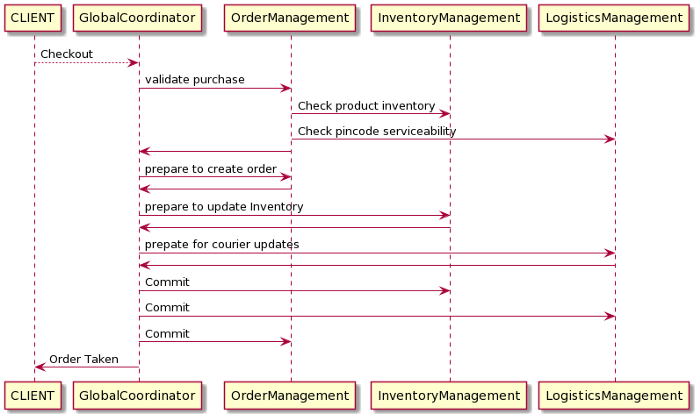
In this method, we use a native query to call the stored procedure. We store the query in the *value* attribute of the annotation.

Similarly, we use *@Param*to match the input parameter of the stored procedure. We also map the stored procedure output to the list of entity *Car* objects.  
  
https://www.baeldung.com/spring-data-jpa-stored-procedures  
  
  
  
  
  
  
  
  
  
  
  
  
*Two-phase commit (2pc)*

The ***two-phase commit protocol*** breaks a database commit into two phases to ensure correctness and fault tolerance in a distributed database system.

As the name suggests, 2pc divides the transaction into two phases — The Prepare phase and the Commit phase. In the prepare phase, all the involved services are asked to prepare for a transaction, once all the services acknowledge the prepare they are asked to commit the transaction. If any service responds with a negative acknowledgement (in case of any internal error), the transaction is aborted globally in all the involved services.

***To achieve this a transaction coordinator is required which manages the entire transaction lifecycle, it issues prepare commands to the individual microservices and listens to their response.***

  
  
If for example, the Logistics management service responds with a negative acknowledgement to the Global Coordinator's (GC) prepare call (this could be due to any internal service failure) the GC would issue abort to all the involved microservice and the transaction will be rolled back globally. If on the contrary all the prepare calls are positively acknowledged by the services the GC calls to commit these changes and the transaction is committed globally.

**Advantages of 2pc:**

* If even one service fails to prepare the whole transaction is aborted otherwise the whole transaction is a success. This ensures that the transactions globally are always ATOMIC in nature.
* The chances of data corruption are low.
* Read-write isolation, since data is only available when the global coordinator commits the changes.
* If the sync call responds with success this means that the transaction is successful overall involved microservices.

**Disadvantages of 2pc:**

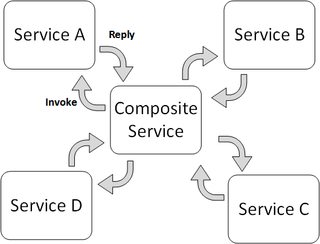
* The biggest disadvantage of 2pc is that it blocks the resource for a long duration, all the individual services need to keep a lock on their resources until the transaction is complete. Multiple involved microservices add to the increased time of the overall transaction leading to long locks on the objects. This is why it's only useful in certain scenarios and not recommended to use otherwise. This approach cannot be used on a high throughput transactional system.
* 2pc also needs a separate service to act as a transaction coordinator, which means added responsibility of managing and scaling this service.
* The transaction coordinator is the single point of failure in 2pc

# The Saga Pattern

# The major problem in 2pc was its synchronous nature which results in the long locks, the Saga pattern, on the other hand, is asynchronous (reactive) in nature thereby making the local transactions run quickly on individual microservices. Service orchestration

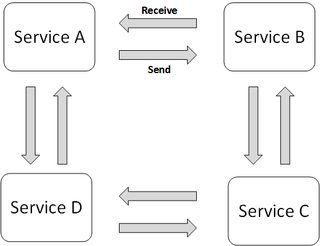
Service orchestration represents a ***single centralized executable business process*** (the **orchestrator**) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

The relationship between all the participating services are described by a single endpoint (i.e., the composite service). The orchestration includes the management of transactions between individual services. Orchestration employs a centralized approach for service composition.



# Service Choreography

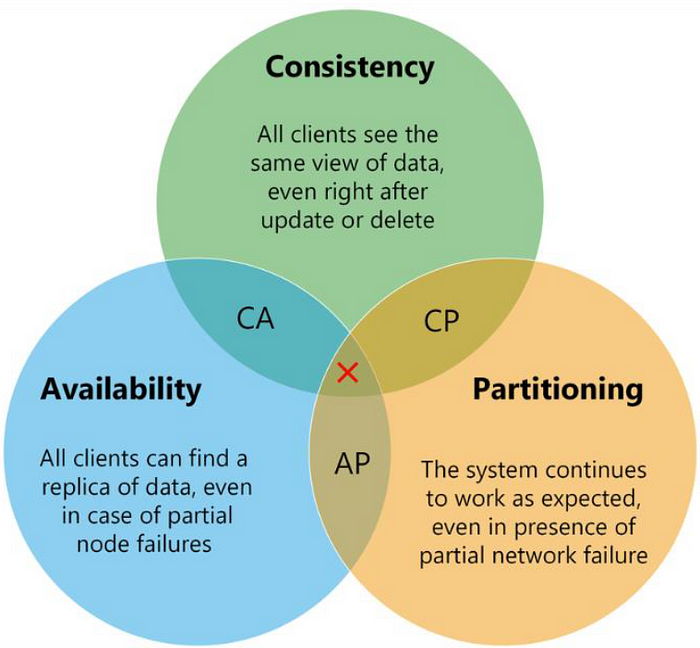
Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints. Choreography employs a decentralized approach for service composition.



***The choreography describes the interactions between multiple services, whereas orchestration represents control from one party's perspective.*** This means that a **choreography** differs from an **orchestration** with respect to where the logic that controls the interactions between the services involved should reside.

Catching the exception will ensure our code executes and transaction won’t rollback both for checked and unchecked exceptions. This could make our bank customers happy, but sometimes you’d rather rollback, and a way to configure it, is to simply add ‘**rollbackFor**’ property to @Transactional like this:  
  
@Transactional(rollbackFor = Exception.class)   
public void transferFunds(String from, String to, Integer amount) throws Exception {   
 Account fromAccount = accountRepository.findByName(from);   
 Account toAccount = accountRepository.findByName(to);   
 takeFrom(fromAccount, amount);   
 giveTo(toAccount, amount);   
}

# CAP Theorem

CAP theorem try to prove that in a distributed system, Consistency, Availability, and Partition Tolerance cannot all be achieved at the same time.  
  
  
So according to CAP Theorem, distributed systems should sacrifice between consistency, availability, and partition tolerance. And, any database can only guarantee two of the three concepts; consistency, availability, and partition tolerance.  
  
**Consistency**  
Consistency means that if the system get any read request, the data should return last updated value from database under all circumstances. If the data cannot be retrieved, an error should be throw and if data is not up-to-date, then it should never be returned. So, when consistent not provide, the system must block the request until all replicas update.

**Availability**The ability of a distributed system to respond to requests at any time. If distributed system can respond all request any time, we can say that the system has high availability. Even if one node in any cluster is down, the system should be able to survive with other nodes. Also high available systems can be fault-tolerance in order to accommodate all requests. Availability in a distributed system ensures that the system remains operational 100% of the time.

**Partition Tolerance**Partition Tolerance is actually network partitioning. That means, parts of your system are located in different networks. Partition Tolerance is the ability of the system to continue its life in case of any communication problem that may occur between the nodes. It’s basically guarantees the system continues to operate even if one data node is down.

If a system is to be fully consistent, it must be sacrifice that always available. Otherwise, even if it is desired to be accessible at all times, the consistency should be sacrificed. ***Mostly in microservices architectures choose Partition Tolerance with High Availability and follow Eventual Consistency for data consistency***.

<https://medium.com/design-microservices-architecture-with-patterns/how-to-choose-a-database-for-microservices-cap-theorem-d1585bf40ecd>