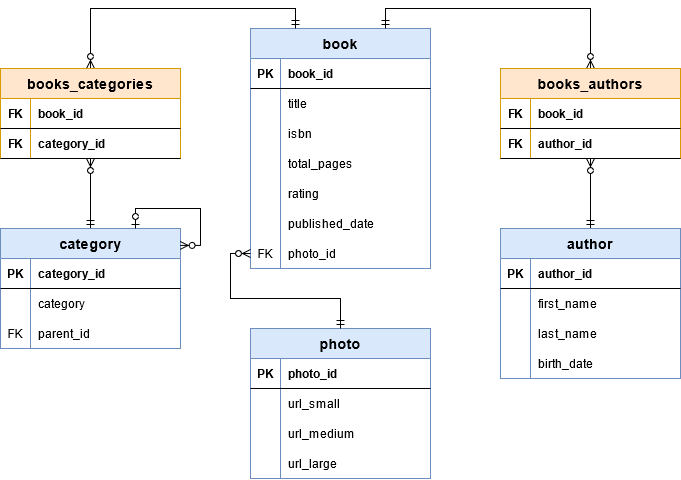
<https://medium.com/huawei-developers/database-relationships-in-spring-data-jpa-8d7181f50f60>

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** {

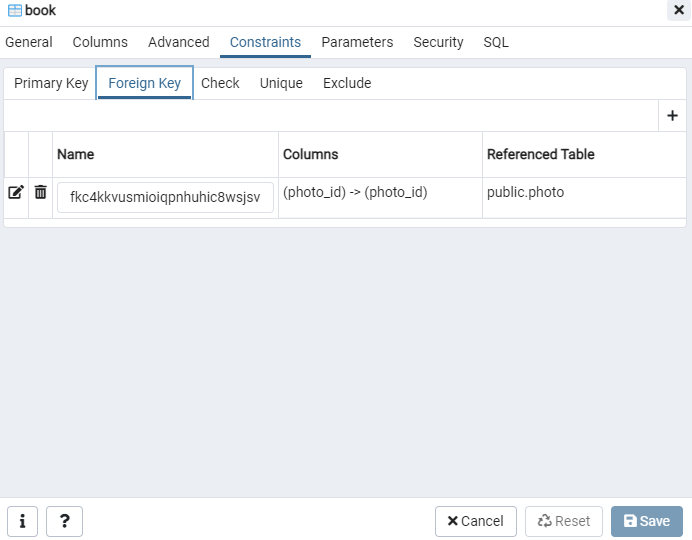
...

*@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")*

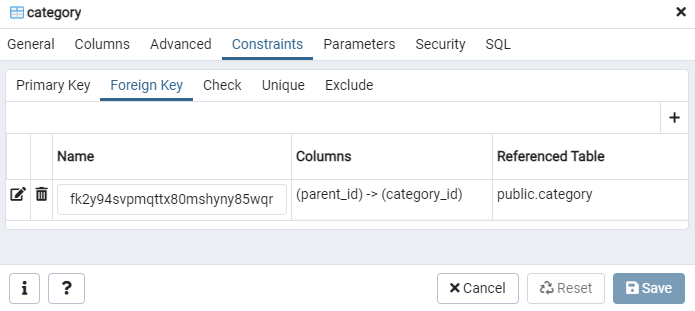
*private Category parent;*

*@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 {

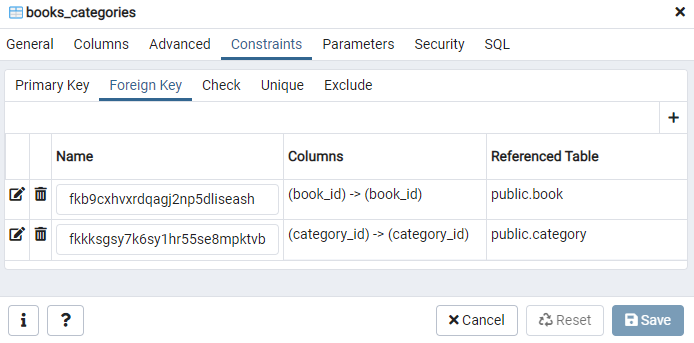
...

*@ManyToMany(mappedBy = "categories")*

*private Collection<Book> books;*

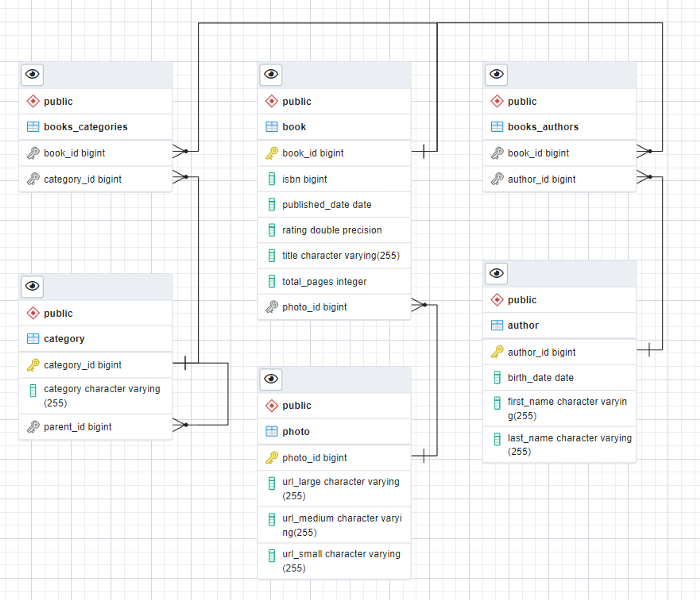
}

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**

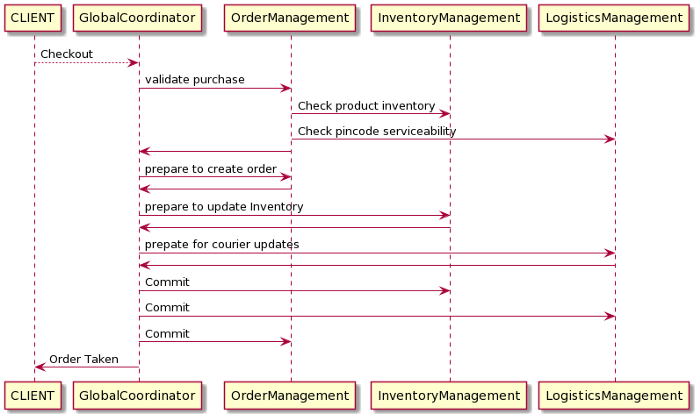


# *Two-phase commit (2pc)*

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:**

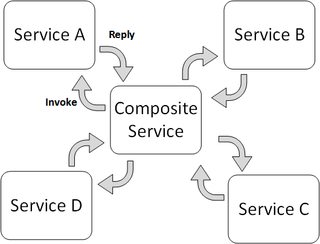
* The biggest disadvantage of 2pc is that it blocks the resource for a long duration, all the individual services need to keep a lock on their resources until the transaction is complete. Multiple involved microservices add to the increased time of the overall transaction leading to long locks on the objects. This is why it's only useful in certain scenarios and not recommended to use otherwise. This approach can not 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

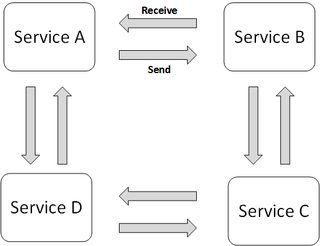
Service orchestration represents a ***single centralized executable business process*** (the **orchestrator**) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

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);   
}