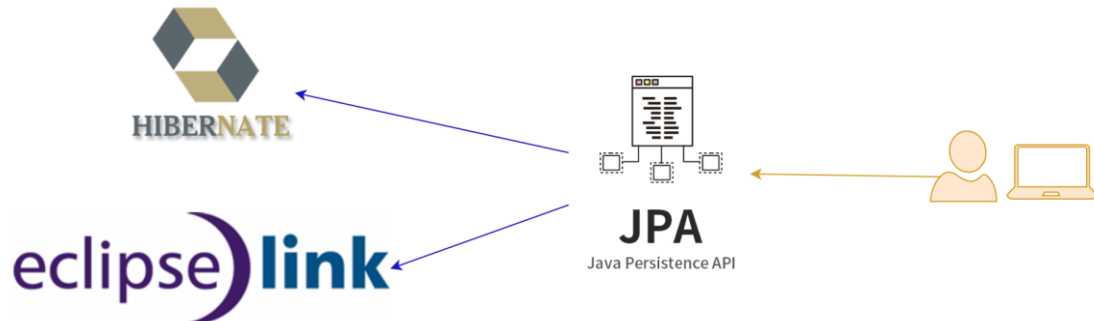


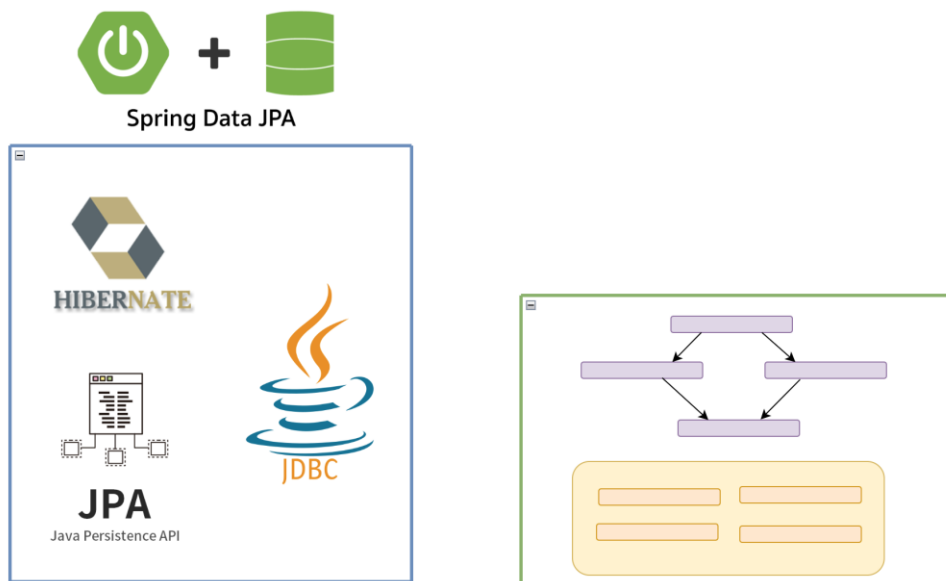
JPA e ORM

JPA é uma especificação do Java EE, ORM são frameworks para mapeamento de objeto relacional que implementam as especificações do JPA:



Spring Data JPA

O **projeto SDJ do Spring** é uma facilitação de interação com operações de banco de dados, então ele **reúne vários frameworks desse contexto (JPA, Hibernate)** e ainda fornece **Interfaces com implementações em tempo de execução** que evitam código boiler plate:



Quando a gente usa o **SDJ** é importante configurar um **DataSource** (*é um objeto que contém as informações de conexão com o database*) que pode ser configurado como classe ou direto no properties:

```
spring.datasource.url=${DB_URL}
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format-sql=true
```

Configuração do DataSource

Configurações adicionais sobre o JPA, útil para depurar queries automáticas

```
@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource dataSourceConfig(){
        return DataSourceBuilder.create().build();
    }
}
```

Geração de PK

Você pode passar a responsabilidade da **geração da PK** pro banco de dados para que ele faça isso de maneira automatizada, existem alguns valores constantes para geração de chaves primárias no JPA, **you define o valor de acordo com o banco de dados que está usando**:

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
private Long id;
```

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE)
private Long id;
```

AUTO = o DB se virá

IDENTITY = A PK vai ser um campo auto_increment

SEQUENCE = você usa a estratégia de SEQUENCE do database, é preciso dizer ao Hibernate a estratégia.

TABLE = usa uma tabela especial para gerenciar as PK, ele salva o valor numa tabela e usa como referência para criar o próximo valor da PK.

Uma chave composta pode ser representada de algumas maneiras.

@IdClass

@Embeddable

@EmbeddedID

Mapeando Herança em ORM

Existem algumas abordagens para mapear entidades que tem herança. Uma delas é a **SINGLE TABLE**, onde todos os atributos da herança vão para uma tabela e cada linha vai ser discriminada por uma coluna que representa a classe específica de toda a herança:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "pmode", discriminatorType = DiscriminatorType.STRING)
public abstract class Payment {
```

Classe mãe, com a coluna que tem o valor discriminatório sendo "pmode"

O valor que vai ser inserido na coluna "pmode"

```
@Entity
@DiscriminatorValue("ch")
public class Check extends Payment {
```

```
@Entity
@DiscriminatorValue("cc")
public class Card extends Payment {
```

O valor que vai ser inserido na coluna "pmode"

TABLE PER CLASS: Aqui a estratégia é fazer uma tabela para cada classe concreta da herança, basta definir na classe mãe a estratégia e mapear normalmente as classes filhas como tabelas próprias:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Payment {
```

JOINED: Aqui cada classe da hierarquia tem sua própria tabela, cada uma carregando seus respectivos atributos. A vantagem aqui é ter tabelas que guardam valores mínimos e a desvantagem

é que o **ORM** vai precisar fazer um join em todas que ele precisar pra um determinado retorno:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Payment {
```

```
@Entity
@Table(name = "CHECK")
@PrimaryKeyJoinColumn(name = "id")
public class Check {

    private String check;
```

Referência a PK da classe mais acima na herança

Mapeando Composição

As composições podem ser mapeadas através de Embeddable e Embedded classes, essas classes se tornam uma única tabela no banco de dados:

```
@Entity
@Table(name = "goal")
public class Goal {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Long objective;

    @Embedded
    private DatesHistory dateHistory;
```

Objeto incorporado

Significa que esse objeto pode ser incorporado em outro objeto

```
@Embeddable
public class DatesHistory {

    @JsonFormat(pattern = "dd/MM/yyyy HH:mm")
    private LocalDateTime creationDate = LocalDateTime.now();

    @JsonFormat(pattern = "dd/MM/yyyy HH:mm")
    private LocalDateTime expectedFinalizationDate;

    @JsonFormat(pattern = "dd/MM/yyyy HH:mm")
    private LocalDateTime realFinalizationDate;

    @JsonFormat(pattern = "dd/MM/yyyy HH:mm")
    private LocalDateTime stopDate;

    @JsonFormat(pattern = "dd/MM/yyyy HH:mm")
    private LocalDateTime retakenDate;
```

Cuidados com non readable queries

Queries que não são só readable precisam ser mapeadas com **@Modifying**. Queries que não são apenas consultas também precisam de **@Transactional**, isso garante que a operação deve acontecer (*abre uma transação e efetua commit apenas no final, se uma exceção for lançada da rollback*):


```
@Transactional
public Goal saveGoal(User user, Goal goal) {
    goal.setUser(user);
    return goalRepository.save(goal);
}
```

```
@Modifying
@Transactional
@Query("UPDATE WalletModel W SET W.limitValue = :limit")
fun resetWalletsLimits(limit : BigDecimal)
```

Paginação

A paginação vem de uma interface específica de `PaginAndSorting`, se você estender de `JpaRepository` já vai ter acesso a ela, buscas paginadas precisam de um objeto de **Paginação**:

O objeto de paginável pode conter atributos como o número da pagina, tamanho dela, tipo de ordenação baseado em que propriedades



```
Pageable pageable = PageRequest.of( page: 1, size: 22, Sort.Direction.ASC, ...properties: "name");
Page<Goal> goals = goalService.findAllById(userId, pageable);
```