Fetch Modes: Eager and Lazy Loading in Spring Boot

In Spring Boot, managing the loading strategy of related entities is crucial for application performance and data consistency. JPA offers two main fetch modes: eager and lazy loading. Understanding how and when to use these fetch modes can significantly impact the efficiency of your application.

Fetch Modes Overview

Eager Loading

Eager loading is a strategy where related entities are loaded immediately with their parent entity. This approach is useful when you know you will need the related entities and want to avoid multiple queries.

Lazy Loading

Lazy loading, on the other hand, delays the loading of related entities until they are explicitly accessed. This can improve performance by reducing the number of initial queries and minimizing the amount of data fetched from the database.

Fetch Modes in Action

Entity Definitions

Let's consider a real-life example of an e-commerce application with `Order` and `Customer` entities. An order is placed by a customer, and we have a one-to-many relationship between `Customer` and `Order`.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    private String name;

    @OneToMany(mappedBy = "customer", fetch = FetchType.LAZY)
    private List<Order> orders;

    // Getters and setters
}

@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String product;

    private Double price;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;

    // Getters and setters
}
```

Eager Loading Example

Suppose you want to retrieve a customer and their orders simultaneously because you know you will need both pieces of information immediately. You can use eager loading by setting the `fetch` attribute to `FetchType.EAGER`.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "customer", fetch = FetchType.EAGER)
    private List<Order> orders;

    // Getters and setters
}
```

Usage Example:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {}

@Service
public class CustomerService {
    @Autowired
    private CustomerRepository customerRepository;

    public Customer getCustomerWithOrders(Long customerId) {
        return customerRepository.findById(customerId).orElseThrow(() -> new Runtimel
    }
}
```

When you fetch a customer using `getCustomerWithOrders`, the associated orders are loaded immediately with the customer.

Lazy Loading Example

Now, let's consider a scenario where you often need to fetch customers without needing their orders. Loading orders only when necessary can improve performance by reducing the amount of data fetched initially.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "customer", fetch = FetchType.LAZY)
    private List<Order> orders;

    // Getters and setters
}
```

Usage Example:

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {}

@Service
public class CustomerService {
    @Autowired
    private CustomerRepository customerRepository;

    public Customer getCustomer(Long customerId) {
        return customerRepository.findById(customerId).orElseThrow(() -> new Runtimel
```

```
    }

    public List<Order> getCustomerOrders(Long customerId) {
        Customer customer = customerRepository.findById(customerId).orElseThrow(() -
        return customer.getOrders(); // Orders are fetched lazily here
    }
}
```

In this case, the `getCustomer` method fetches the customer without loading the orders. The orders are only loaded when `getCustomerOrders` is called.

Real-Life Scenario

Imagine an e-commerce dashboard where administrators frequently view customer details without needing order information. However, when analyzing a specific customer's purchasing history, they need detailed order data.

- **Eager Loading:** When an admin views a detailed report of a customer's purchase history, eager loading can fetch the customer and orders in a single query, providing all necessary information at once.
- **Lazy Loading:** When an admin views a list of customers, lazy loading avoids fetching order data, reducing the initial query's load and improving performance. If the admin clicks on a customer to view detailed order information, the orders are fetched at that moment.

Conclusion

Choosing between eager and lazy loading depends on your application's specific needs. Eager loading is suitable when you know you'll need related entities immediately, while lazy loading is beneficial for reducing initial query load and improving performance when related data is not always required. By understanding and strategically applying these fetch modes, you can optimize your Spring Boot application's performance and resource utilization.