- Hibernate properties for Spring Boot

```
# DataSource configuration
spring.datasource.url=jdbc:mysql://localhost:3306/your_database
spring.datasource.username=your_username
spring.datasource.password=your_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Hibernate properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

# Naming strategy for database tables
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

# Enable Hibernate second-level cache and specify the cache region factory
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

# Specify the JDBC batch size
spring.jpa.properties.hibernate.jdbc.batch_size=20

# Control fetching strategy for associations
spring.jpa.properties.hibernate.default_batch_fetch_size=16
spring.jpa.properties.hibernate.max_fetch_depth=3

# Enable automatic session context management
```

```
spring.jpa.properties.hibernate.current_session_context_class=thread

# Specify the isolation level for database transactions
spring.jpa.properties.hibernate.transaction.isolation=READ_COMM
ITTED

# Enable statistics for Hibernate
spring.jpa.properties.hibernate.generate_statistics=true

# Control the format of SQL statements
spring.jpa.properties.hibernate.format_sql=true
```

JPA VS SPRING DATA JPA

JPA (Java Persistence API):

1. **Standardized Java Specification:**
    - JPA is a standardized Java specification that defines an API for object-relational mapping in Java applications.
    - It provides a set of interfaces and annotations that developers can use to map Java objects to database tables and vice versa.
2. **ORM Functionality:**
    - JPA serves as a standard interface for developers to interact with relational databases using Java objects.
    - It abstracts away the details of SQL queries and database-specific intricacies, allowing developers to focus on the Java code.
3. **Provider-Neutral:**
    - JPA is provider-neutral, meaning that it defines a standard interface, and different JPA providers (like Hibernate, EclipseLink, etc.) can implement this specification.
    - Developers can switch between JPA providers without changing their application code, promoting flexibility.

Spring Data JPA:

1. **Higher-Level Abstraction:**
    - Spring Data JPA is a part of the broader Spring Data project, which aims to simplify data access in Spring applications.

    - It builds on top of JPA and provides a higher-level abstraction, making it easier to perform common data access operations without boilerplate code.
2. **Repository Abstraction:**
    - Spring Data JPA introduces the concept of repositories, which are interfaces that extend JpaRepository or one of its subinterfaces.
    - These repositories provide methods for common CRUD (Create, Read, Update, Delete) operations, and Spring Data JPA automatically generates the necessary implementations at runtime.
3. **Query Methods and Custom Queries:**
    - Spring Data JPA allows developers to define query methods in repository interfaces by following a naming convention. These methods are automatically translated into SQL queries.
    - It also supports the use of custom JPQL (Java Persistence Query Language) queries when more complex queries are needed.
4. **Integration with Spring Ecosystem:**
    - Spring Data JPA seamlessly integrates with the broader Spring ecosystem, including features like transaction management, dependency injection, and declarative configuration.

Importance and Use Cases:

- **JPA:**
    - Essential for providing a standardized and vendor-agnostic way of performing object-relational mapping in Java applications.
    - Suitable for applications where direct control over SQL queries and database interactions is necessary or when using a specific JPA provider.
- **Spring Data JPA:**
    - Ideal for Spring-based applications where developers want to leverage higher-level abstractions and reduce boilerplate code.
    - Significantly reduces the amount of code needed for common data access tasks, making development more efficient.
    - Well-suited for projects that benefit from Spring's features and conventions.

Combined Usage:

- **Many projects use both JPA and Spring Data JPA together:**
    - Developers can use JPA for low-level interactions with the database, custom queries, or cases where they need to work directly with the EntityManager.
    - Spring Data JPA is often used for simplifying common CRUD operations and leveraging its repository abstraction.

MYBATIS VS OTHER STRATEGIES

MyBatis:

Key Points:

- **SQL-Centric Approach:** MyBatis is known for its SQL-centric approach. It allows developers to define SQL queries in external XML files or annotations, providing more control over the SQL being executed.
- **Dynamic SQL:** MyBatis supports dynamic SQL, making it easier to build queries based on dynamic conditions or criteria.
- **Flexibility:** Developers have more control over the SQL and can optimize queries as needed. It's a good fit for projects where fine-tuned control over SQL is essential.
- **Less Abstraction:** Compared to full-fledged ORM frameworks, MyBatis provides less abstraction over the database, giving developers more control over the SQL execution.

2. EclipseLink:

Key Points:

- **ORM Framework:** EclipseLink is a comprehensive ORM framework that implements the JPA (Java Persistence API) specification. It aims to simplify the interaction between Java objects and relational databases.
- **Annotation-Based Mapping:** EclipseLink uses annotations to map Java entities to database tables. It reduces the need for XML configuration files, making it more convention-based.
- **Advanced Features:** EclipseLink provides advanced features such as caching, lazy loading, and support for advanced JPA features like inheritance mapping.
- **JPA Standard:** EclipseLink adheres to the JPA standard, making it a good choice for projects that follow the JPA specifications. It's often used in Java EE (Enterprise Edition) applications.

Other Strategies:

1. **Hibernate:**
    - Hibernate is another popular and powerful ORM framework that also implements the JPA specification. It provides a high level of abstraction over the database, allowing developers to work with Java objects directly.

2. **Spring Data JPA:**
   - Spring Data JPA is part of the broader Spring Data project. It simplifies data access in Spring applications, including support for JPA. It provides repository abstractions and query methods, reducing boilerplate code.
3. **JDBC Template (Spring):**
   - For cases where a more manual and low-level approach is needed, the JDBC template in the Spring framework allows developers to interact with databases using plain SQL and JDBC operations.

Importance and Considerations:

- **Choosing Between MyBatis and EclipseLink:**
  - Choose MyBatis when you need more control over SQL queries and a SQL-centric approach.
  - Choose EclipseLink when you prefer a more standard JPA approach with higher-level abstractions and features.
- **Hibernate and Spring Data JPA:**
  - Hibernate is a powerful and feature-rich ORM framework widely used in enterprise applications.
  - Spring Data JPA is well-suited for Spring-based applications where developers want to leverage Spring's features and conventions.

SPRINGBOOT DEVELOPER WANTS TO KNOW ANOTATION-LEVEL

Spring Boot provides a wide range of annotations that simplify the development of applications by handling common tasks and configurations. Below is a list of commonly used Spring Boot annotations along with explanations of when to use them at the project level:

1. **`@SpringBootApplication`**:
   - **When to use:** This annotation should be used at the main class level. It combines three commonly used annotations (`@Configuration`, `@EnableAutoConfiguration`, `@ComponentScan`) to mark the main configuration class of a Spring Boot application.
2. **`@Controller`**, **`@RestController`**:

- **When to use:** These annotations are used to define controllers. Use `@RestController` if you want the controller methods to return the response directly, usually in the form of JSON.

3. **`@Service`**:
    - **When to use:** Use this annotation to mark a class as a service. Services contain business logic and are typically used to perform operations on data retrieved from repositories.
4. **`@Repository`**:
    - **When to use:** This annotation is used to indicate that a class is a repository, which is responsible for interacting with the database. Spring Boot provides additional functionality, such as exception translation for JPA, when this annotation is used.
5. **`@Component`**:
    - **When to use:** Use this annotation to indicate that a class is a Spring component. It is a generic stereotype for any Spring-managed component, and it can be used for classes that don't fit into more specific stereotype categories like `@Service` or `@Repository`.
6. **`@Configuration`**:
    - **When to use:** Use this annotation on classes that define Spring bean configurations. It's often used in combination with `@Bean` methods to define custom bean configurations.
7. **`@EnableAutoConfiguration`**:
    - **When to use:** Typically, this annotation is not explicitly used at the project level. Spring Boot automatically applies sensible default configurations based on the dependencies included in your project. However, you might use it to fine-tune auto-configuration if needed.
8. **`@EnableWebMvc`**:
    - **When to use:** Use this annotation if you want to customize the default Spring MVC configuration. For example, you may want to add custom converters or configure message converters.
9. **`@EntityScan`**:
    - **When to use:** Use this annotation to specify the base packages to scan for JPA entities. This is useful when your entity classes are not located in the same package or subpackages as your main application class.
10. **`@EnableJpaRepositories`**:
    - **When to use:** Use this annotation to enable JPA repositories in your Spring Boot application. It's typically placed on the main configuration class.
11. **`@ConfigurationProperties`**:

- **When to use:** Use this annotation to bind external configuration properties to a Java class. It's useful when you want to centralize configuration properties in a custom configuration class.

12. **`@Value`**:
    - **When to use:** Use this annotation to inject values from property files or environment variables into a Spring bean. It's a concise way to inject simple values.
13. **`@Qualifier`**:
    - **When to use:** Use this annotation to disambiguate bean resolution when multiple beans of the same type are present in the application context.
14. **`@Autowired`**:
    - **When to use:** Use this annotation for automatic dependency injection. It can be used on fields, setter methods, and constructors.
15. **`@PathVariable`**, **`@RequestParam`**, **`@RequestBody`**:
    - **When to use:** Use these annotations in controller methods to extract values from the URL path, query parameters, or request body.
16. **`@RequestMapping`**:
    - **When to use:** Use this annotation to map web requests to controller methods. It can be used at the class level and method level.
17. **`@GetMapping`**, **`@PostMapping`**, etc.:
    - **When to use:** These annotations are shortcuts for `@RequestMapping(method = RequestMethod.GET)`, `@RequestMapping(method = RequestMethod.POST)`, and so on. Use them for more concise mapping of HTTP methods.
18. **`@Transactional`**:
    - **When to use:** Use this annotation to mark a method, or all methods of a class, as transactional. It ensures that the method is executed within a transaction, and the transaction is committed if the method completes successfully or rolled back if an exception is thrown.

19. **`@EnableCaching`**, **`@Cacheable`**, **`@CacheEvict`**:
    - **When to use:** Use these annotations for caching. `@EnableCaching` is used at the configuration class level to enable caching, and `@Cacheable` and `@CacheEvict` are used on methods to control caching behavior.
20. **`@ConditionalOnProperty`**:
    - **When to use:** Use this annotation to conditionally enable or disable certain configurations based on the values of properties in the application's configuration.

1. `@EnableCaching`:

Purpose:

Enables Spring's caching support in the application.

Sample Implementation:

```
java

@SpringBootApplication
@EnableCaching
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

2. `@Cacheable`:

Purpose:

Indicates that the result of the annotated method should be cached. The value returned from the method is stored in the cache, and subsequent calls to the same method with the same arguments return the cached result.

Sample Implementation:

```
java

@Service
public class MyService {

    @Cacheable("myCache")
    public String getFromDatabase(String key) {
        // Simulating fetching data from the database
        return "Data for key: " + key;
    }
}
```

3. `@CacheEvict`:

Purpose:

Indicates that the specified cache(s) should be cleared (evicted) either before or after the method is executed.

Sample Implementation:

```
java

@Service
public class MyService {
    @CacheEvict(value = "myCache", allEntries = true)
    public void clearCache() {
        // This method will clear all entries in the "myCache" cache.
    }
}
```

DEMO CODE SPRING CACHE

(I) Dependency

```
<dependencies>
 <!-- Spring Boot Starter -->
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter</artifactId>
 </dependency>

 <!-- Spring Boot Starter Data JPA -->
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-jpa</artifactId>
 </dependency>

 <!-- Spring Boot Starter Cache -->
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-cache</artifactId>
 </dependency>

 <!-- Caffeine Cache -->
 <dependency>
 <groupId>com.github.ben-manes.caffeine</groupId>
 <artifactId>caffeine</artifactId>
 </dependency>

 <!-- H2 Database (for simplicity) -->
 <dependency>
 <groupId>com.h2database</groupId>
 <artifactId>h2</artifactId>
 <scope>runtime</scope>
 </dependency>
</dependencies>
```

(II)Configure Caffeine Cache in application.properties:

```
properties

# Enable caching with Caffeine
spring.cache.type=caffeine

# Caffeine Cache Settings
spring.cache.caffeine.spec=maximumSize=100,expireAfterWrite=10m
```

In this configuration:

- `maximumSize=100` sets the maximum size of the cache to 100 entries.
- `expireAfterWrite=10m` sets a time-to-live of 10 minutes for each entry in the cache.

(III)

Create a service that uses caching with the @Cacheable annotation.

```java
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class BookService {

    private final BookRepository bookRepository;

    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    @Cacheable(value = "booksCache", key = "#genre")
    public List<Book> findBooksByGenre(String genre) {
        System.out.println("Fetching books from the database for genre: " + genre);
        return bookRepository.findByGenre(genre);
    }

    @CacheEvict(value = "booksCache", allEntries = true)
    public void evictAllBooksCache() {
        System.out.println("Evicting all entries from the booksCache");
    }
}
```

4. `@EnableTransactionManagement`:

Purpose:

Enables Spring's annotation-driven transaction management.

Sample Implementation:

```java
@SpringBootApplication
@EnableTransactionManagement
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}

@Service
public class MyService {

    @Transactional
    public void performTransactionalOperation() {
        // This method is now executed within a transaction.
    }
}
```

5. `@EnableScheduling`:

Purpose:

Enables Spring's scheduled task execution capabilities. It allows the use of `@Scheduled` annotations on methods.

Sample Implementation:

```
java

@SpringBootApplication
@EnableScheduling
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}

@Service
public class MyScheduledService {

    @Scheduled(fixedRate = 5000) // Run every 5 seconds
    public void performScheduledTask() {
        // This method will be executed at the specified interval.
    }
}
```

6. `@EnableAsync`:

Purpose:

Enables support for asynchronous method execution.

Sample Implementation:

```
java

@SpringBootApplication
@EnableAsync
```

```
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}

@Service
public class MyAsyncService {

    @Async
    public CompletableFuture<String> performAsyncOperation() {
        // Simulating an asynchronous operation
        return CompletableFuture.completedFuture("Async operation completed");
    }
}
```

These annotations provide powerful features for caching, transaction management, scheduling, and asynchronous processing in a Spring Boot application. Make sure to include the relevant dependencies in your project, such as `spring-boot-starter-cache`, `spring-boot-starter-data-jpa` (for transactions), and others, depending on your specific use case.