СБЕР

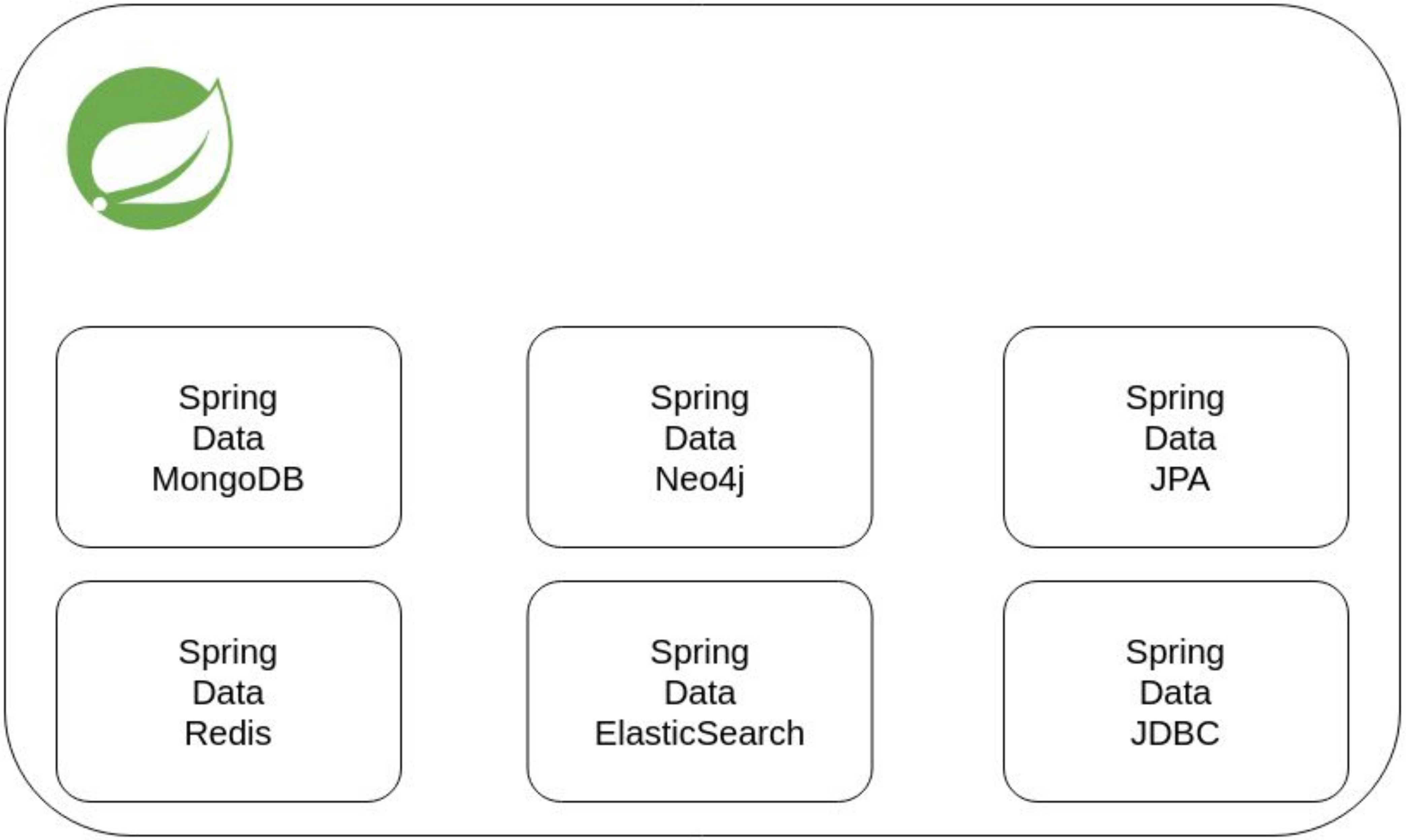
Spring Data JDBC. Problems known, problems unknown

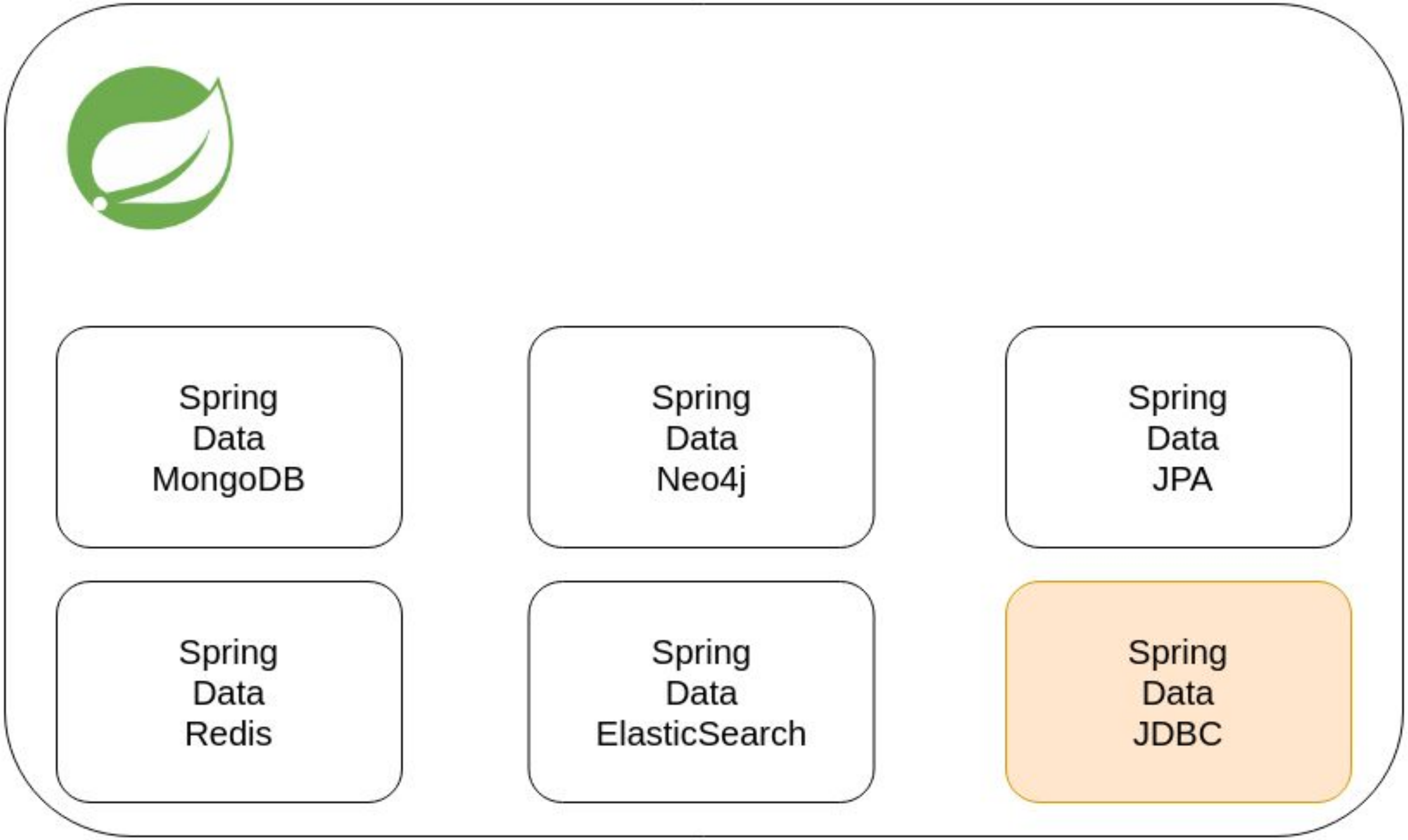
Поливаха Михаил (@mipo256)

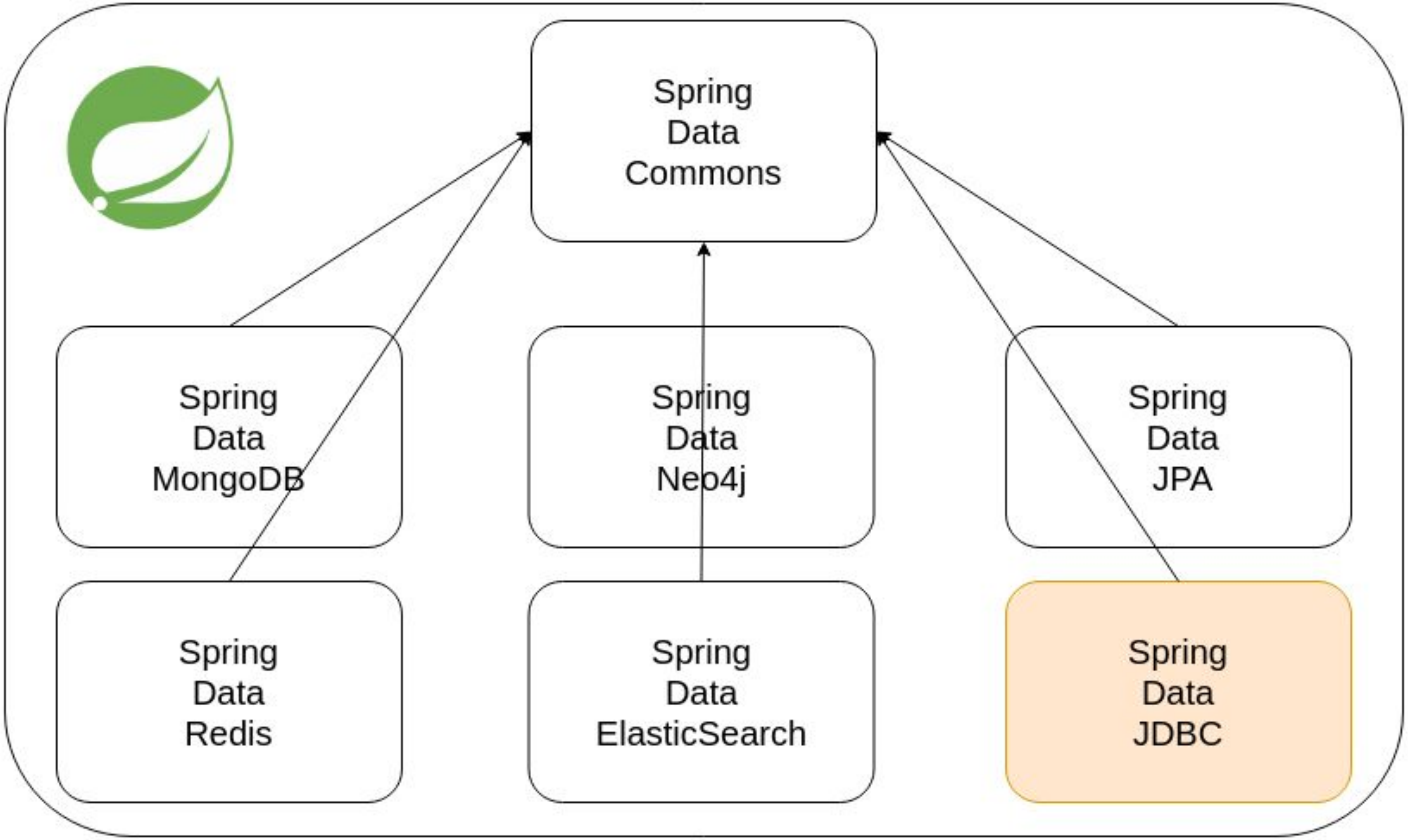
- **Certified AWS Solution Architect**
- **Skilled Software Engineer**
- **Active OpenSource member (including Spring projects)**
- **Technical Writer**

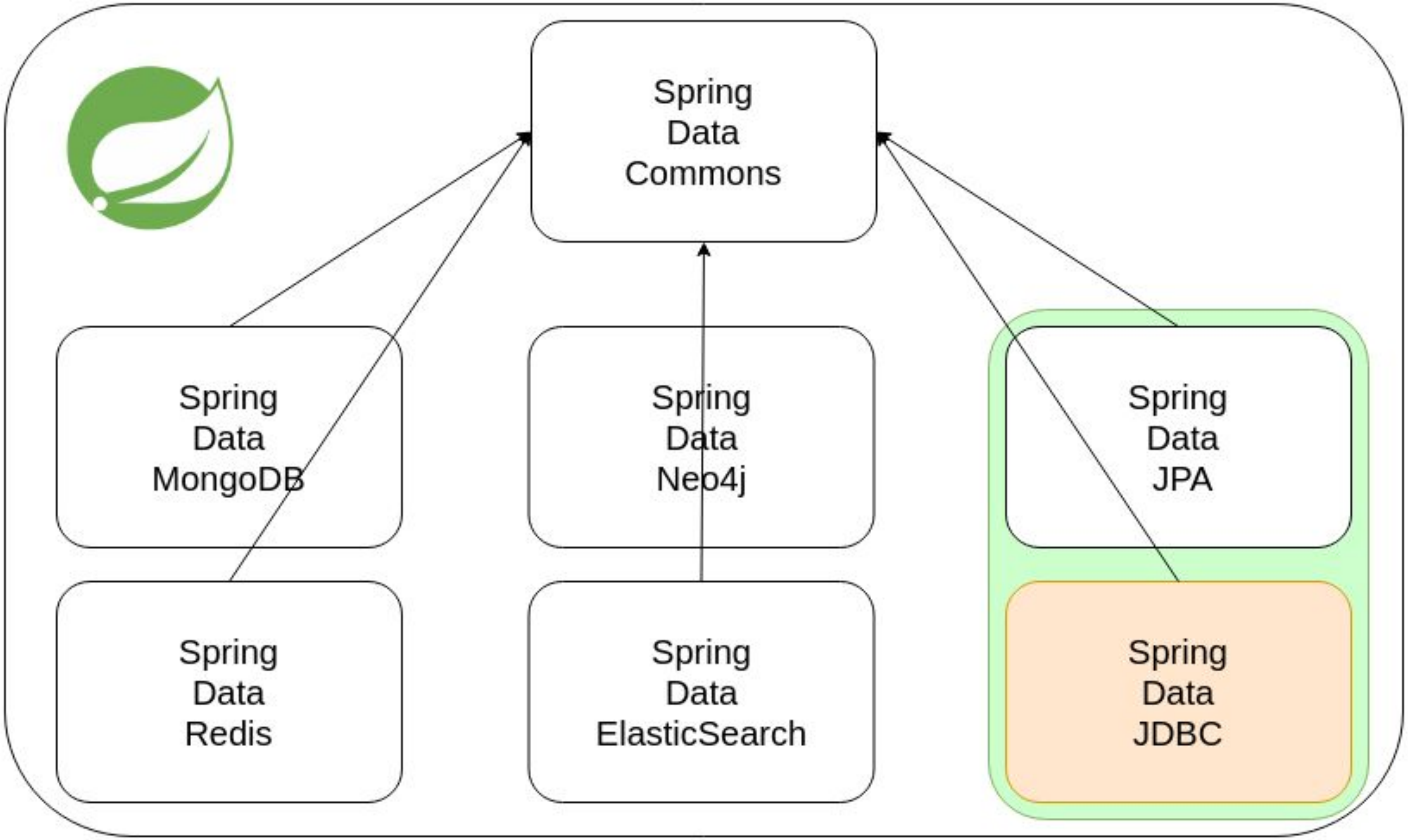
Contacts:

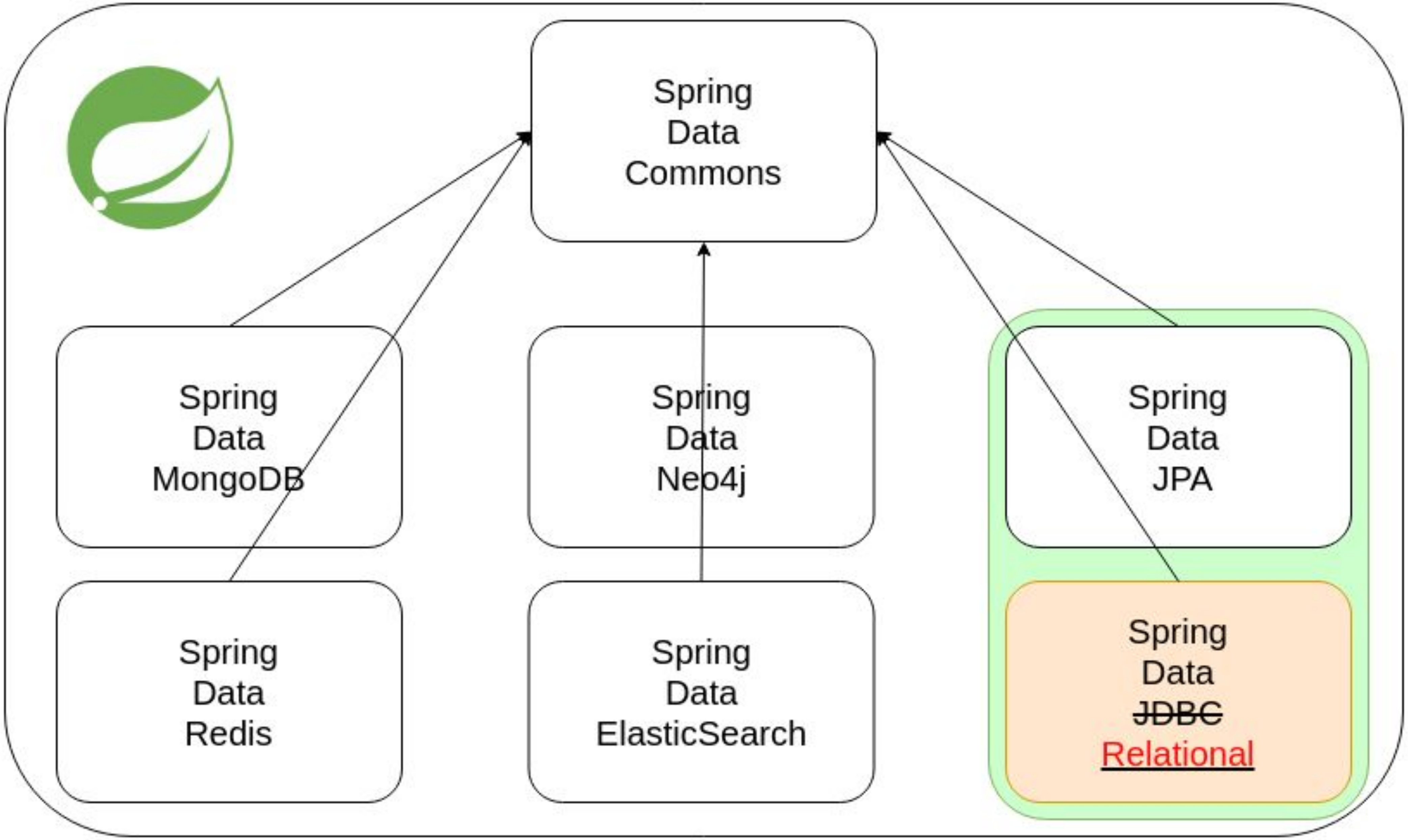
- **Telegram: @mipo256**
- **GitHub: mipo256**
- **Email: mikhailpolivakha@gmail.com**

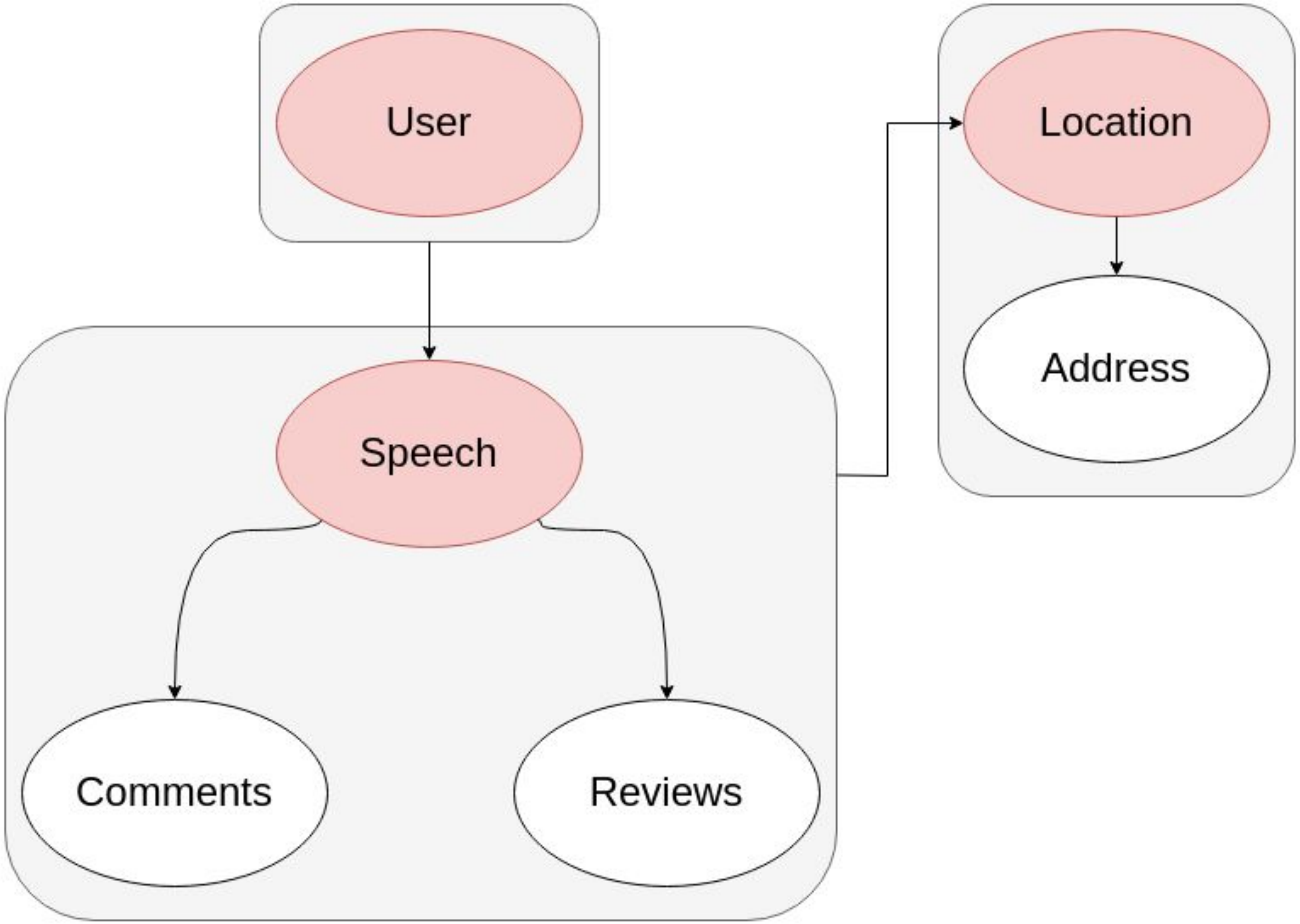

Spring Data MongoDB
Spring Data Neo4j
Spring Data JPA
Spring Data Redis
Spring Data ElasticSearch
Spring Data JDBC

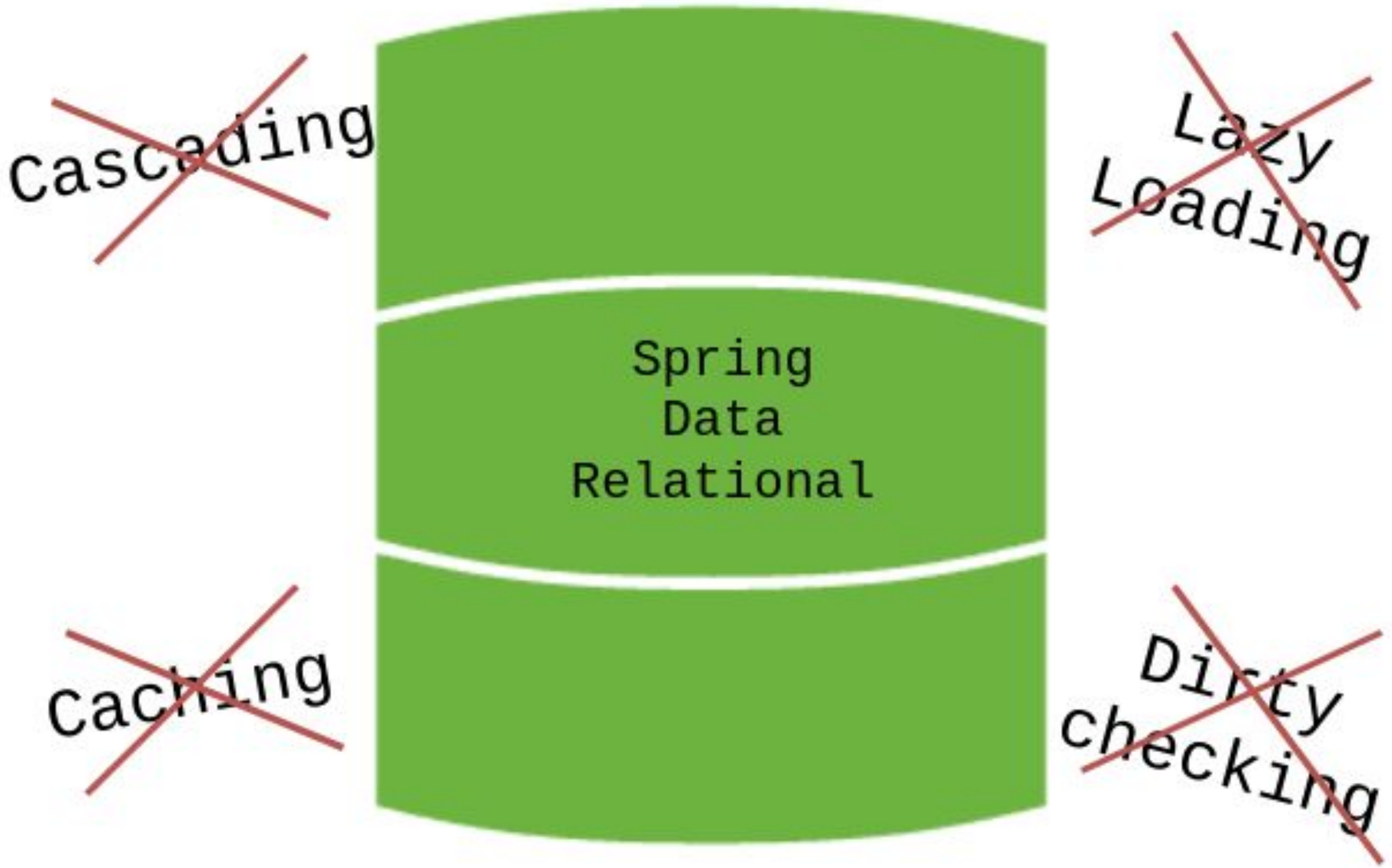

Spring Data MongoDB
Spring Data Neo4j
Spring Data JPA
Spring Data Redis
Spring Data ElasticSearch
Spring Data JDBC


Spring Data Commons
Spring Data MongoDB
Spring Data Neo4j
Spring Data JPA
Spring Data Redis
Spring Data ElasticSearch
Spring Data JDBC


Spring Data Commons
Spring Data MongoDB
Spring Data Neo4j
Spring Data JPA
Spring Data Redis
Spring Data ElasticSearch
Spring Data JDBC

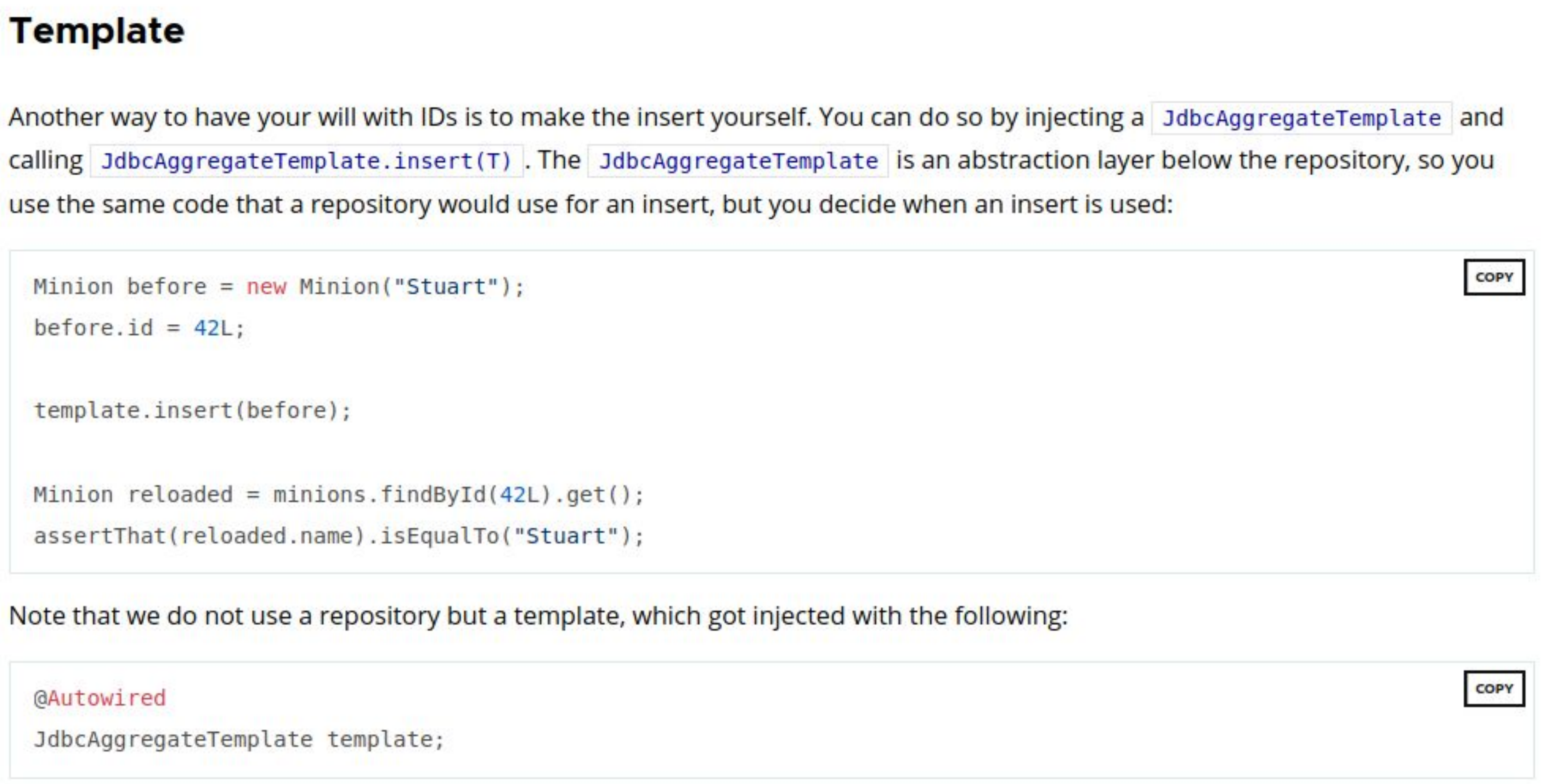

Spring Data Commons
Spring Data MongoDB
Spring Data Neo4j
Spring Data JPA
Spring Data Redis
Spring Data ElasticSearch
Spring Data ~~JDBC~~ Relational


User
Speech
Comments
Reviews
Location
Address


~~Cascading~~
~~Lazy Loading~~
Spring Data Relational
~~Caching~~
~~Dirty checking~~

ID GENERATION

Just use JdbcAggregateTemplate[1]



Template

Another way to have your will with IDs is to make the insert yourself. You can do so by injecting a `JdbcAggregateTemplate` and calling `JdbcAggregateTemplate.insert(T)`. The `JdbcAggregateTemplate` is an abstraction layer below the repository, so you use the same code that a repository would use for an insert, but you decide when an insert is used:

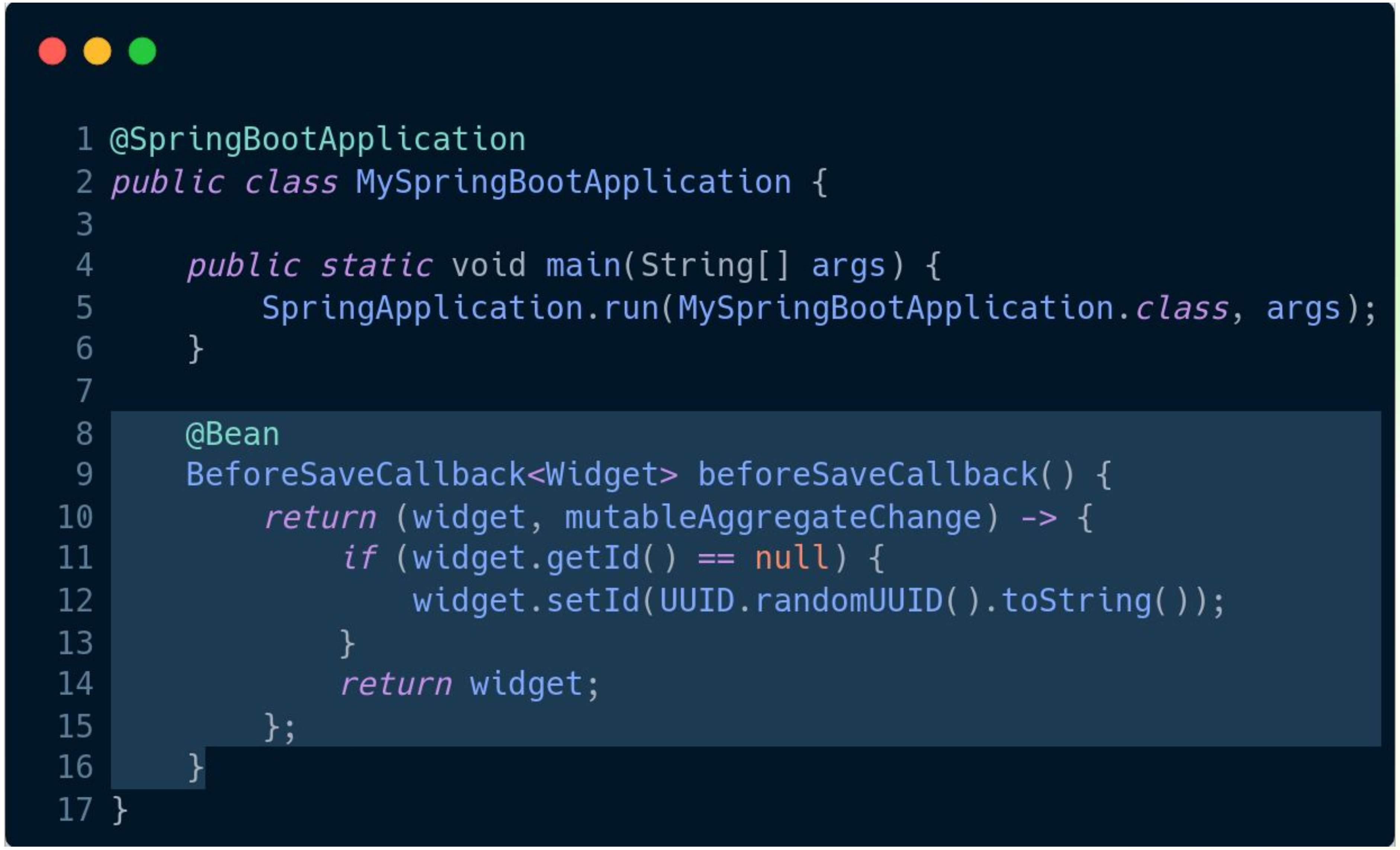
```
Minion before = new Minion("Stuart");
before.id = 42L;

template.insert(before);

Minion reloaded = minions.findById(42L).get();
assertThat(reloaded.name).isEqualTo("Stuart");
```

Note that we do not use a repository but a template, which got injected with the following:

```
@Autowired
JdbcAggregateTemplate template;
```

[1]https://spring.io/blog/2021/09/09/spring-data-jdbc-how-to-use-custom-id-generation

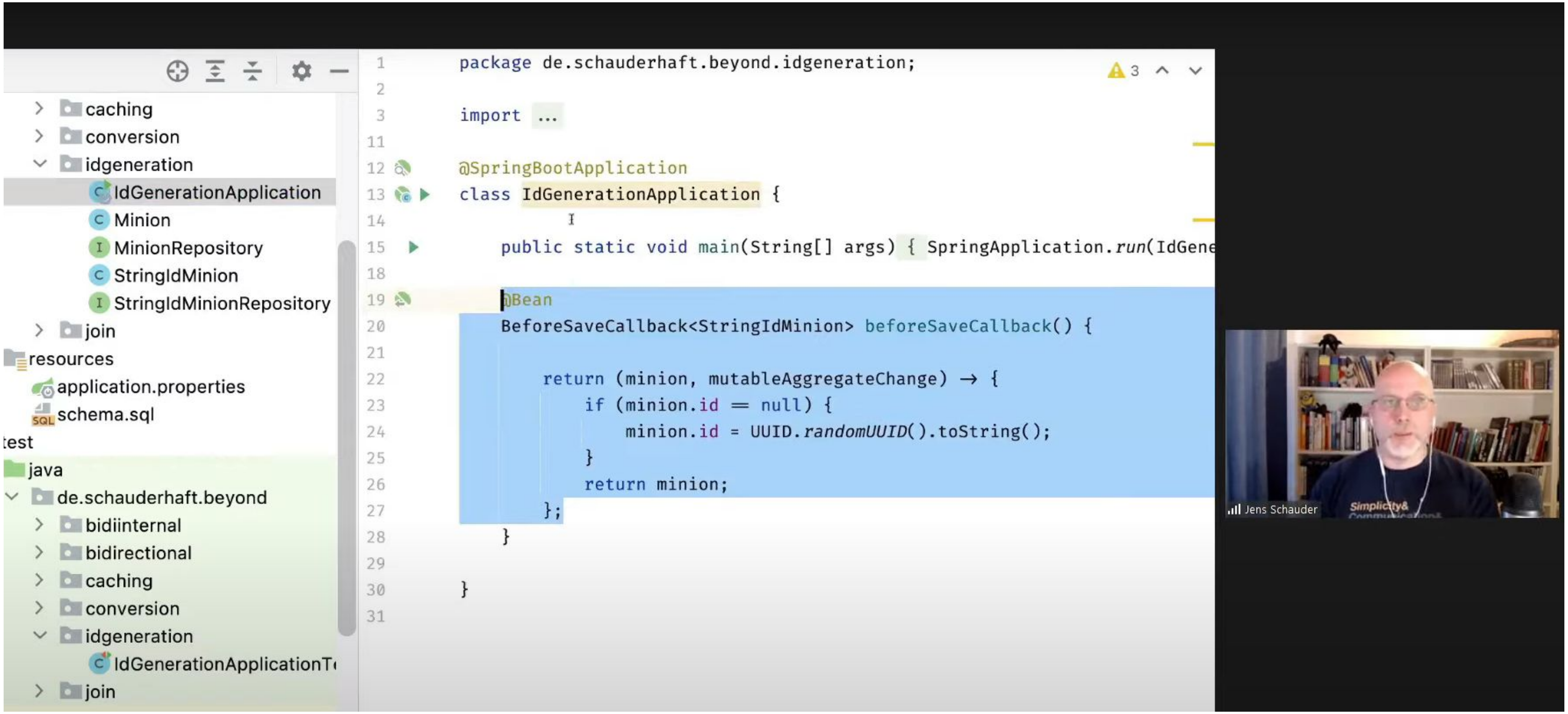
Adding the callback should save the day[1]



```java
@SpringBootApplication
public class MySpringBootApplication {

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

    @Bean
    BeforeSaveCallback<Widget> beforeSaveCallback() {
        return (widget, mutableAggregateChange) -> {
            if (widget.getId() == null) {
                widget.setId(UUID.randomUUID().toString());
            }
            return widget;
        };
    }
}
```

[1]https://codetinkering.com/spring-jdbc-beforesavecallback-example/

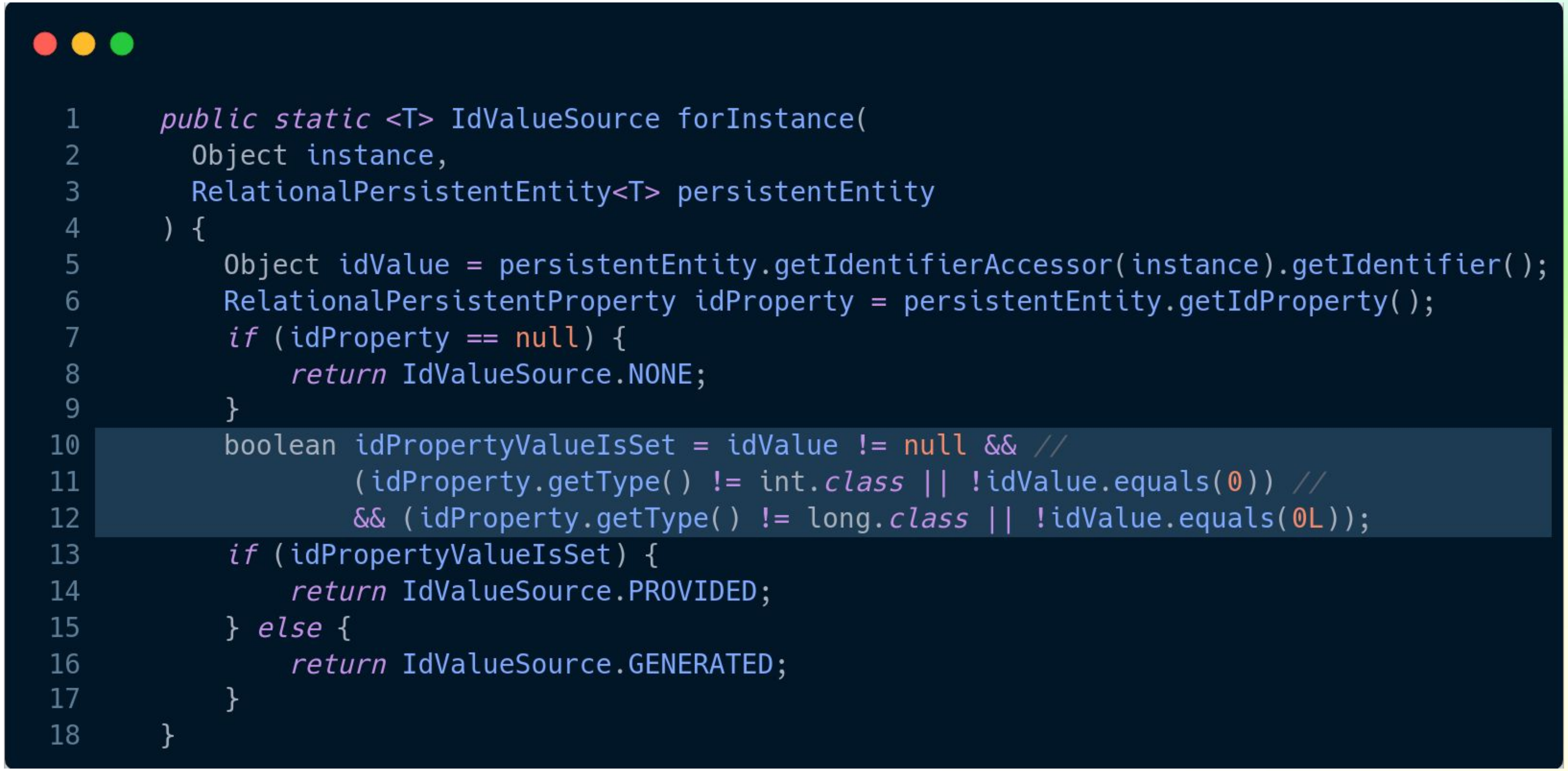
Demo by Technical Lead[1]



```
package de.schauderhaft.beyond.idgeneration;

import ...

@SpringBootApplication
class IdGenerationApplication {

    public static void main(String[] args) { SpringApplication.run(IdGene

    @Bean
    BeforeSaveCallback<StringIdMinion> beforeSaveCallback() {

        return (minion, mutableAggregateChange) → {
            if (minion.id == null) {
                minion.id = UUID.randomUUID().toString();
            }
            return minion;
        };
    }

}
```

[1]https://www.youtube.com/watch?v=SJlKBkZ2yAU&t=870s

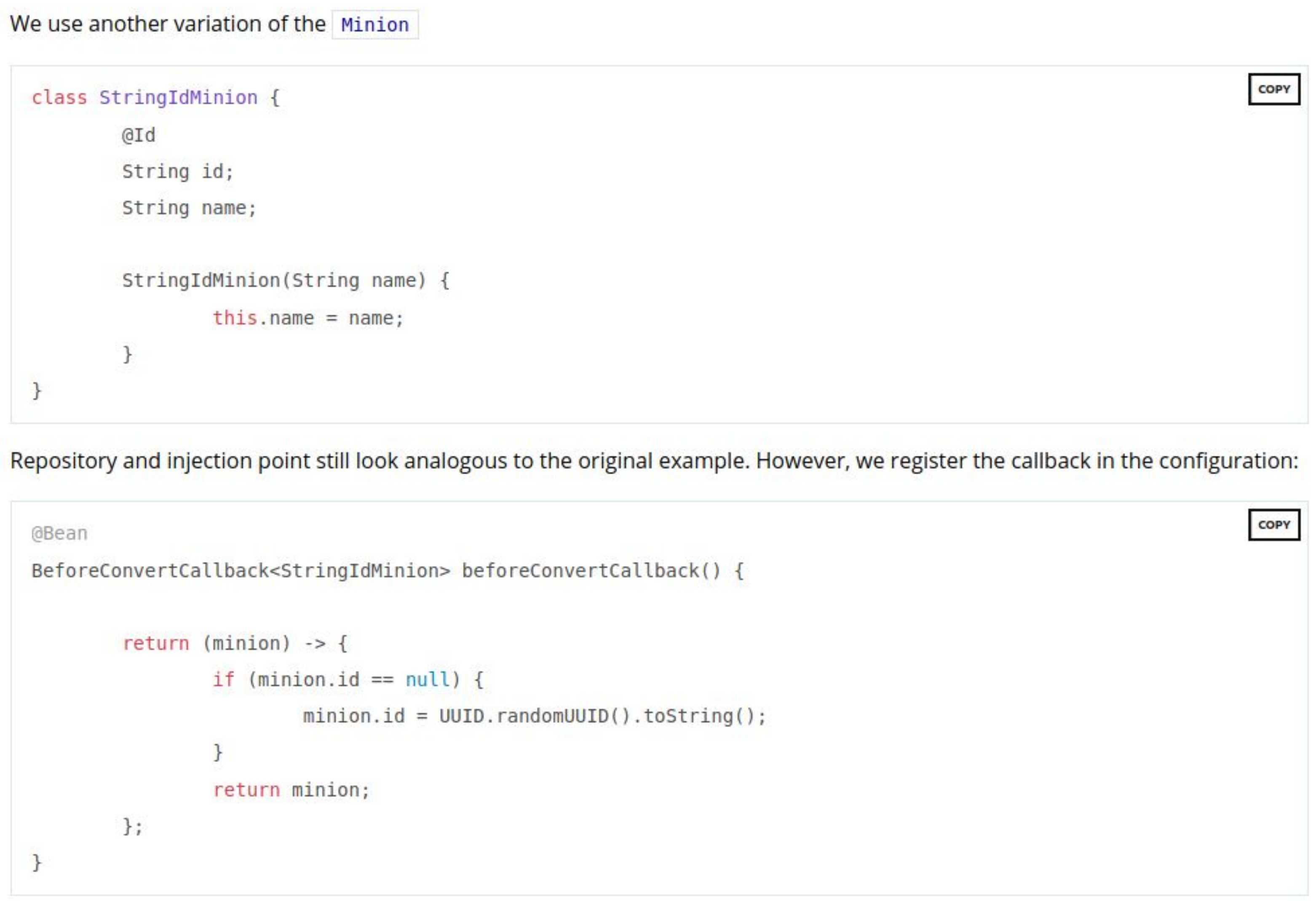
How do we know the aggregate is new?[1]



```java
public static <T> IdValueSource forInstance(
  Object instance,
  RelationalPersistentEntity<T> persistentEntity
) {
    Object idValue = persistentEntity.getIdentifierAccessor(instance).getIdentifier();
    RelationalPersistentProperty idProperty = persistentEntity.getIdProperty();
    if (idProperty == null) {
        return IdValueSource.NONE;
    }
    boolean idPropertyValueIsSet = idValue != null && //
            (idProperty.getType() != int.class || !idValue.equals(0)) //
            && (idProperty.getType() != long.class || !idValue.equals(0L));
    if (idPropertyValueIsSet) {
        return IdValueSource.PROVIDED;
    } else {
        return IdValueSource.GENERATED;
    }
}
```

[1]https://github.com/spring-projects/spring-data-relational/blob/main/spring-data-relational/src/main/java/org/springframework/data/relational/core/conversion/IdValueSource.java#L49

Documentation is (almost) fine![1]



We use another variation of the `Minion`

```
class StringIdMinion {
        @Id
        String id;
        String name;

        StringIdMinion(String name) {
                this.name = name;
        }
}
```

Repository and injection point still look analogous to the original example. However, we register the callback in the configuration:

```
@Bean
BeforeConvertCallback<StringIdMinion> beforeConvertCallback() {

        return (minion) -> {
                if (minion.id == null) {
                        minion.id = UUID.randomUUID().toString();
                }
                return minion;
        };
}
```

[1] https://spring.io/blog/2021/09/09/spring-data-jdbc-how-to-use-custom-id-generation

Key takeaways

1. Use BeforeConvertCallback - it is going to work

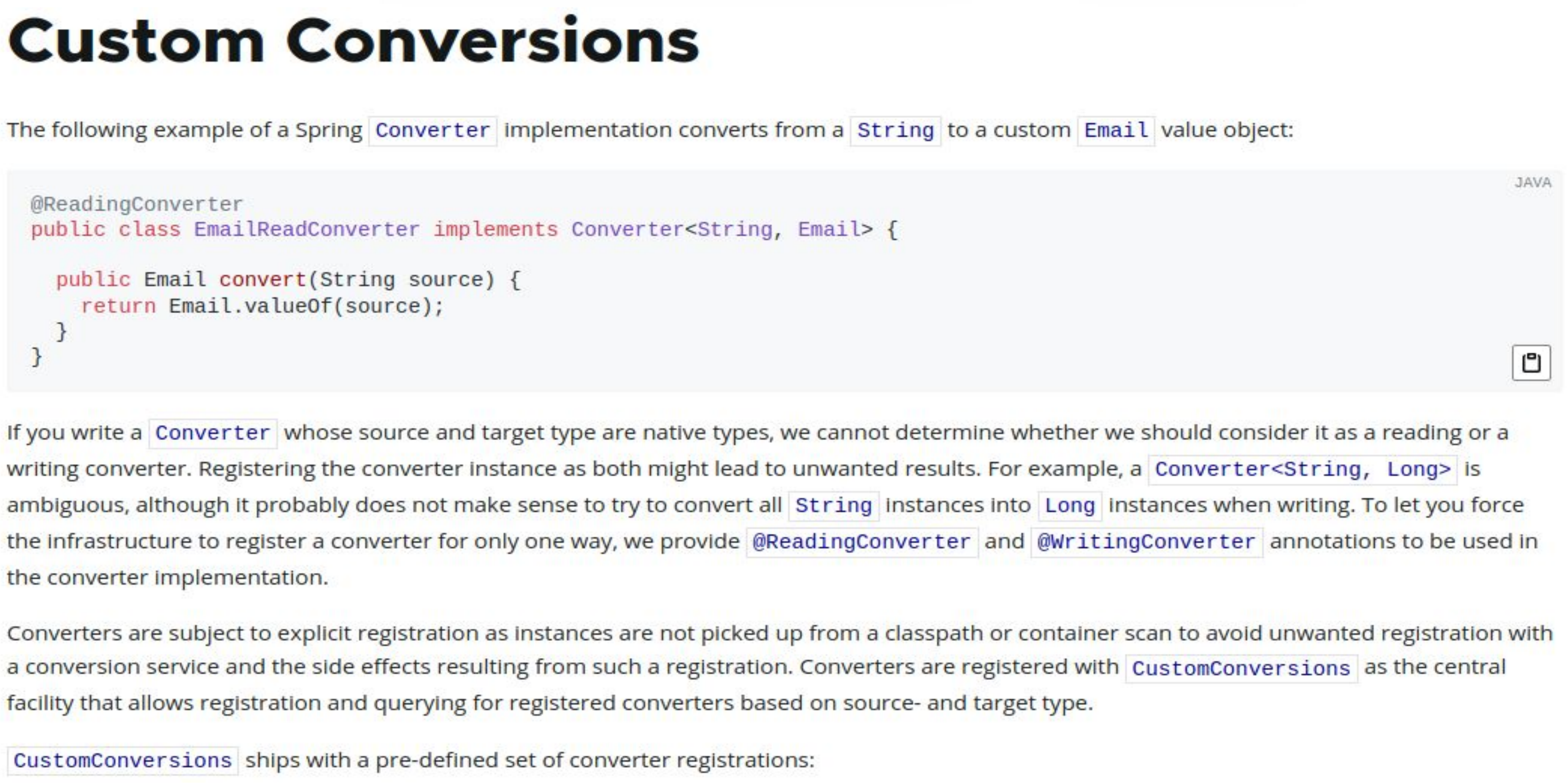
2. Now, BeforeSaveCallback is not working and we will probably leave it as is

3. There is no easy, out of the box way to generate IDs (However, there is a ticket for that)

#1169

Custom Conversions

Custom Conversions are great[1]



Custom Conversions

The following example of a Spring `Converter` implementation converts from a `String` to a custom `Email` value object:

JAVA

```java
@ReadingConverter
public class EmailReadConverter implements Converter<String, Email> {

  public Email convert(String source) {
    return Email.valueOf(source);
  }
}
```

If you write a `Converter` whose source and target type are native types, we cannot determine whether we should consider it as a reading or a writing converter. Registering the converter instance as both might lead to unwanted results. For example, a `Converter<String, Long>` is ambiguous, although it probably does not make sense to try to convert all `String` instances into `Long` instances when writing. To let you force the infrastructure to register a converter for only one way, we provide `@ReadingConverter` and `@WritingConverter` annotations to be used in the converter implementation.

Converters are subject to explicit registration as instances are not picked up from a classpath or container scan to avoid unwanted registration with a conversion service and the side effects resulting from such a registration. Converters are registered with `CustomConversions` as the central facility that allows registration and querying for registered converters based on source- and target type.

`CustomConversions` ships with a pre-defined set of converter registrations:

[1] https://docs.spring.io/spring-data/relational/reference/commons/custom-conversions.html

Key takeaways

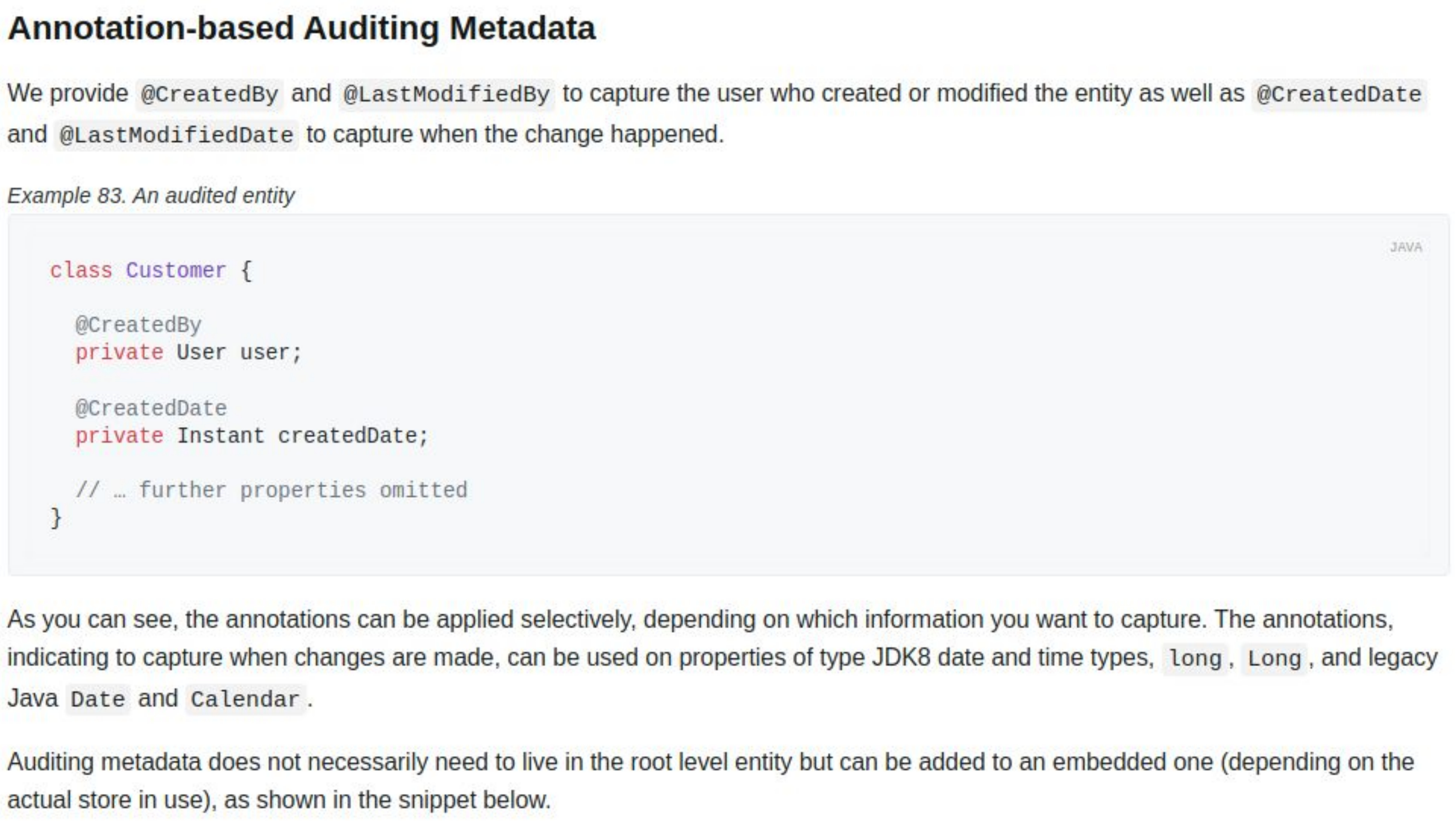
1. Currently, there is no way to specify on what fields converter should apply and on what fields should not

2. This issue is known to developers, but currently nobody is working on it

#1602

Auditing Annotations

We support common's audit annotations!



Annotation-based Auditing Metadata

We provide `@CreatedBy` and `@LastModifiedBy` to capture the user who created or modified the entity as well as `@CreatedDate` and `@LastModifiedDate` to capture when the change happened.

Example 83. An audited entity

JAVA

```java
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private Instant createdDate;

  // … further properties omitted
}
```

As you can see, the annotations can be applied selectively, depending on which information you want to capture. The annotations, indicating to capture when changes are made, can be used on properties of type JDK8 date and time types, `long`, `Long`, and legacy Java `Date` and `Calendar`.

Auditing metadata does not necessarily need to live in the root level entity but can be added to an embedded one (depending on the actual store in use), as shown in the snippet below.

[1] https://docs.spring.io/spring-data/jdbc/docs/3.1.9/reference/html/#auditing.annotations

Key takeaways

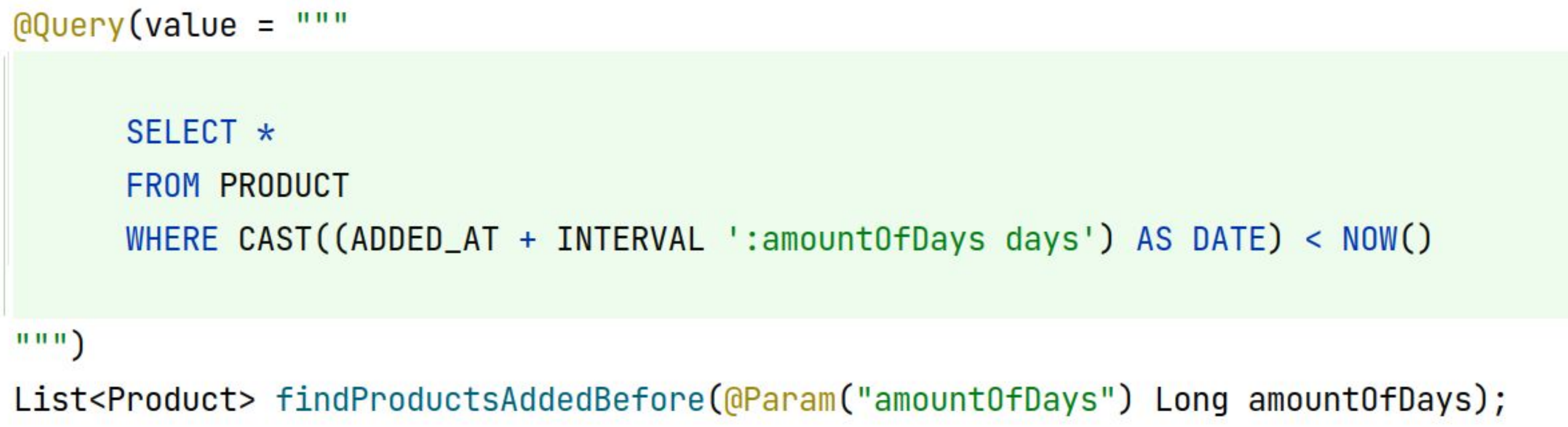
1. Features, described in Spring-Data-Common documentation, are not always implemented in the exactly described way in child modules

2. Auditing annotations work only on root properties

3. This is by design, and we are not going to fix it

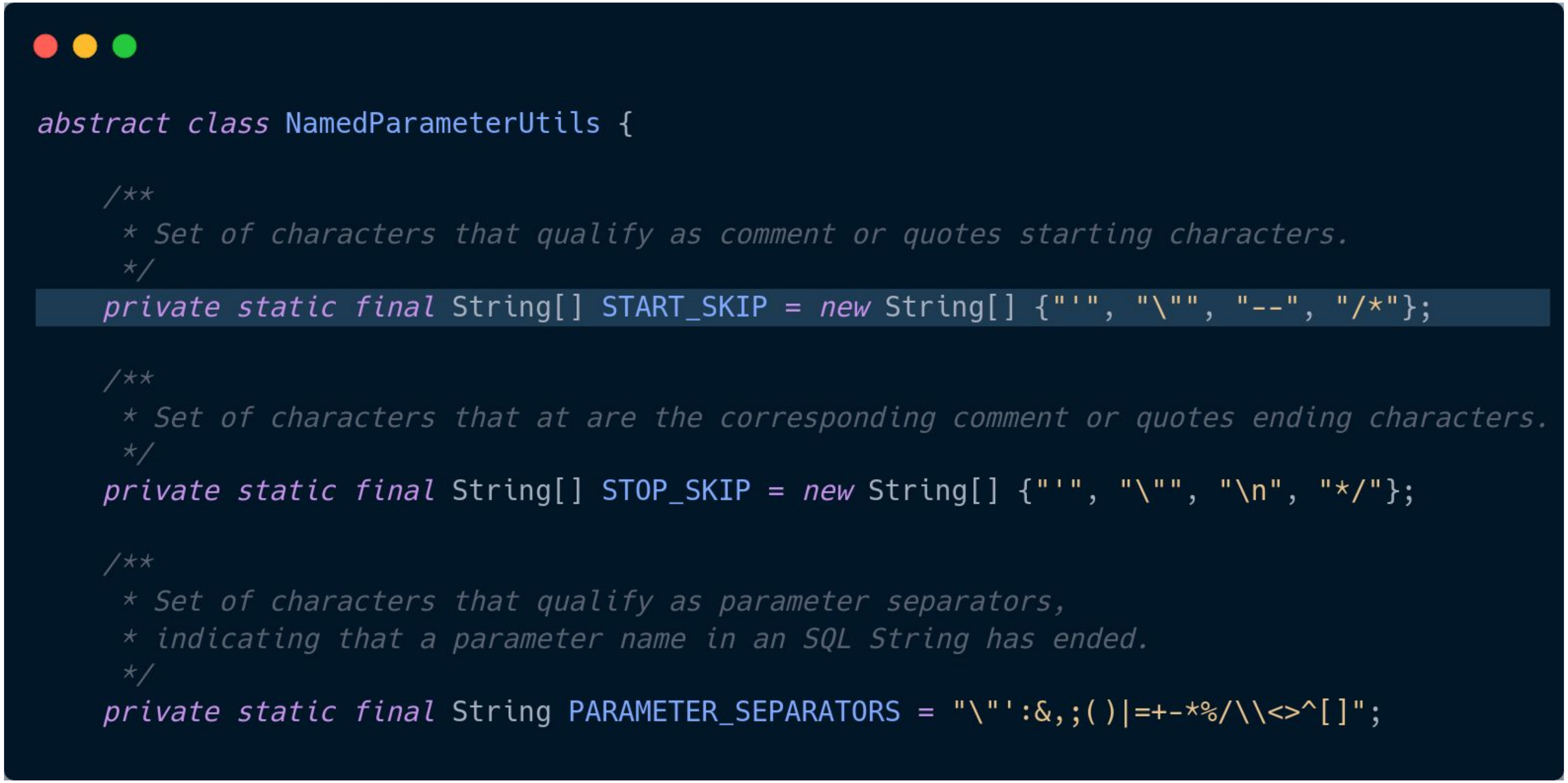
Named Parameters Resolution

Using INTERVAL type in @Query



```java
@Query(value = """

    SELECT *
    FROM PRODUCT
    WHERE CAST((ADDED_AT + INTERVAL ':amountOfDays days') AS DATE) < NOW()

""")
List<Product> findProductsAddedBefore(@Param("amountOfDays") Long amountOfDays);
```

Named parameters resolution[1]



```java
abstract class NamedParameterUtils {

    /**
     * Set of characters that qualify as comment or quotes starting characters.
     */
    private static final String[] START_SKIP = new String[] {"'", "\"", "--", "/*"};

    /**
     * Set of characters that at are the corresponding comment or quotes ending characters.
     */
    private static final String[] STOP_SKIP = new String[] {"'", "\"", "\n", "*/"};

    /**
     * Set of characters that qualify as parameter separators,
     * indicating that a parameter name in an SQL String has ended.
     */
    private static final String PARAMETER_SEPARATORS = "\"':&,;()|=+-*%/\\<>^[]";
```

[1]https://github.com/spring-projects/spring-framework/blob/main/spring-r2dbc/src/main/java/org/springframework/r2dbc/core/NamedParameterUtils.java

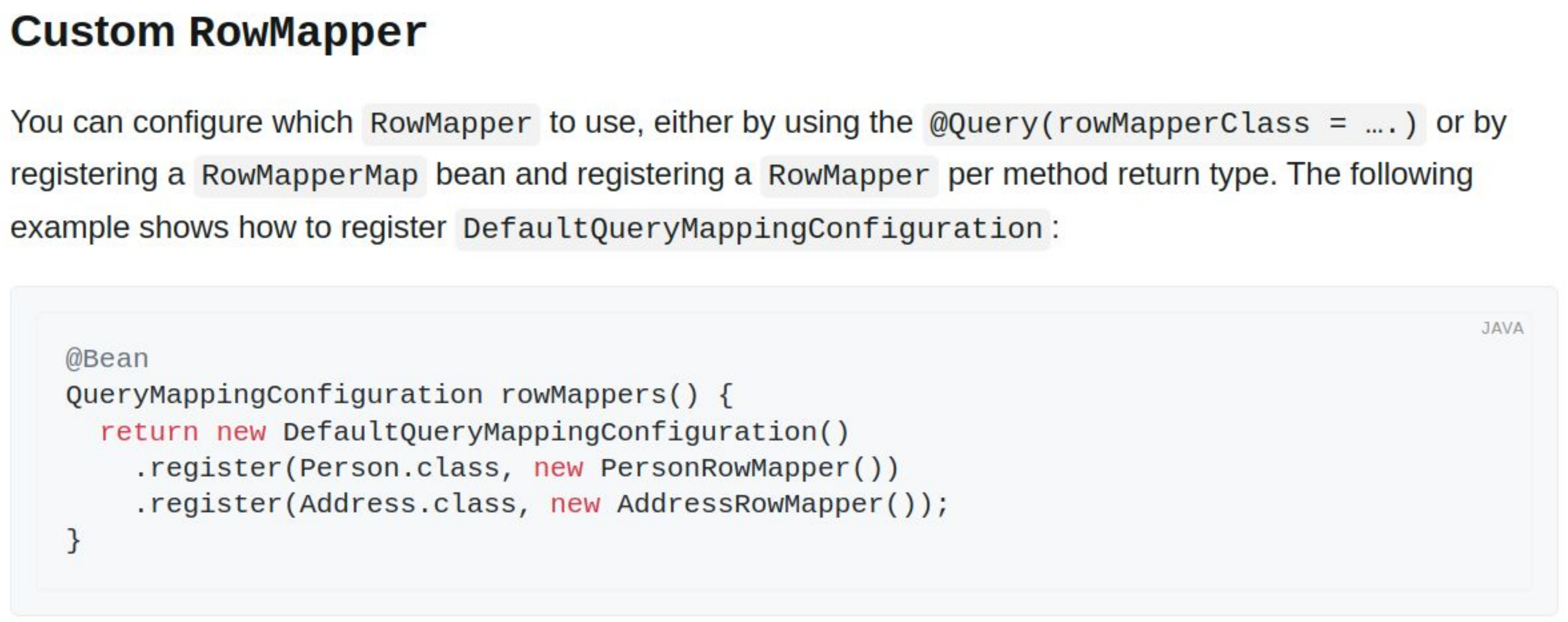
Key takeaways

1. SQL named parameter e not interpreted due to escape sequences

2. Spring Data Relati much, since we just rely on NamedParameterJ

Custom Row Mapping

Custom mapping of ResultSet[1]



Custom RowMapper

You can configure which `RowMapper` to use, either by using the `@Query(rowMapperClass = ….)` or by registering a `RowMapperMap` bean and registering a `RowMapper` per method return type. The following example shows how to register `DefaultQueryMappingConfiguration`:

```java
@Bean
QueryMappingConfiguration rowMappers() {
  return new DefaultQueryMappingConfiguration()
    .register(Person.class, new PersonRowMapper())
    .register(Address.class, new AddressRowMapper());
}
```

[1]https://docs.spring.io/spring-data/jdbc/docs/3.1.9/reference/html/#jdbc.query-methods.at-query.custom-rowmapper

Key takeaways

1. QueryMappingConfiguration for now works only for string-based queries.

2. To solve for now, use rowMapperRef/rowMapperClass annotation attributes

3. There is a ticket to make QueryMappingConfiguration work for all queries

#1006

JSR310 Timestamps

Key takeaways

1. Timestamps conversion are kind of broken now
2. Some JSR310 (New Java Time API) classes also behave weirdly
3. There is a ticket and PR already for this

#1136

What we are about to introduce

- Add support for ValueExpression #1738
- DTO projections causes the converter to be called twice #1725
- Reuse TypeInformation during PersistentPropertyPath and PersistentEntity lookups #1679
- Consider JTS Geometry types as simple types #1711
- io.r2dbc.spi.Parameter not considered a simple type #1696

What we introduced lately

- Schema generation #756
- Load aggregates that have a single one-to-m relationship with a single select #1446
- Allow using JDBC and R2DBC repositories in a single application #1143

Conclusions

1. Spring Data Relational in general is working, but it has some unspoken problems.

2. Some of these problems are known to developers of the framework but unknown to the public.

3. ***We are actively trying to make Spring Data Relational better. For now, it is much, much better than it was 5 years ago.***

What we haven't discussed yet

1. Why development of Spring Data Relational is so slow?
2. What is the priority of tickets selection for Spring Data Relational (probably for other modules as well)?
3. Why some submitted tickets get declined?
4. What can you try to do to ensure that your ticket get accepted?
5. What we need help on right now?