



[New Guide] Download the 2018 Guide to Containers: Development and Management

[Download Guide ▶](#)

# Java: Using the Specification Pattern With JPA

by **Michael Scharhag** MVB · Dec. 30, 13 · Java Zone · Tutorial

Learn how to build stream processing applications in Java-includes reference application. Brought to you in partnership with Hazelcast.

This article is an introduction to using the specification pattern in Java. We also will see how we can combine classic specifications with JPA Criteria queries to retrieve objects from a relational database.

Within this post we will use the following Poll class as an example entity for creating specifications. It represents a poll that has a start and end date. In the time between those two dates users can vote among different choices. A poll can also be locked by an administrator before the end date has been reached. In this case, a lock date will be set.

```
@Entity
public class Poll {

    @Id
    @GeneratedValue
    private long id;

    private DateTime startDate;

    private DateTime endDate;

    private DateTime lockDate;

    @OneToMany(cascade = CascadeType.ALL)
    private List<Vote> votes = new ArrayList<>();
}
```

For better readability I skipped getters, setters, JPA annotations for mapping Joda DateTime instances and fields that aren't needed in this example (like the question being asked in the poll).

Now assume we have two constraints we want to implement:

- A poll is *currently running* if it is not locked and if startDate < now < endDate
- A poll is *popular* if it contains more than 100 votes and is not locked

We could start by adding appropriate methods to Poll like: poll.isCurrentlyRunning(). Alternatively we could use a service method like pollService.isCurrentlyRunning(poll). However, we also want to be able to query the database to get all currently running polls. So we might add a DAO or repository method like pollRepository.findAllCurrentlyRunningPolls().

If we follow this way we implement the *isCurrentlyRunning* constraint two times in two different locations. Things become worse if we want to combine constraints. What if we want to query the database for a list of all popular polls that are currently running?

This is where the specification pattern come in handy. When using the specification pattern we move business rules into extra classes called specifications.

To get started with specifications we create a simple interface and an abstract class:

```
public interface Specification<T> {
    boolean isSatisfiedBy(T t);
    Predicate toPredicate(Root<T> root, CriteriaBuilder cb);
    Class<T> getType();
}

abstract public class AbstractSpecification<T> implements Specification<T> {
    @Override
    public boolean isSatisfiedBy(T t) {
        throw new NotImplementedException();
    }

    @Override
    public Predicate toPredicate(Root<T> poll, CriteriaBuilder cb) {
        throw new NotImplementedException();
    }
}
```

```

@Override
public Class<T> getType() {
    ParameterizedType type = (ParameterizedType) this.getClass().getGenericSuperclass();
    return (Class<T>) type.getActualTypeArguments()[0];
}
}

```

Please ignore the `AbstractSpecification<T>` class with the mysterious `getType()` method for a moment (we come back to it later).

The central part of a specification is the `isSatisfiedBy()` method, which is used to check if an object satisfies the specification. `toPredicate()` is an additional method we use in this example to return the constraint as `javax.persistence.criteria.Predicate` instance which can be used to query a database. For each constraint we create a new specification class that extends `AbstractSpecification<T>` and implements `isSatisfiedBy()` and `toPredicate()`.

The specification implementation to check if a poll is currently running looks like this:

```

public class IsCurrentlyRunning extends AbstractSpecification<Poll> {
    @Override
    public boolean isSatisfiedBy(Poll poll) {
        return poll.getStartDate().isBeforeNow()
            && poll.getEndDate().isAfterNow()
            && poll.getLockDate() == null;
    }

    @Override
    public Predicate toPredicate(Root<Poll> poll, CriteriaBuilder cb) {
        DateTime now = new DateTime();
        return cb.and(
            cb.lessThan(poll.get(Poll_.startDate), now),
            cb.greaterThan(poll.get(Poll_.endDate), now),
            cb.isNull(poll.get(Poll_.lockDate))
        );
    }
}

```

Within `isSatisfiedBy()` we check if the passed object matches the constraint. In `toPredicate()` we construct a `Predicate` using JPA's `CriteriaBuilder`. We will use the resulting `Predicate` instance later to build a `CriteriaQuery` for querying the database.

The specification for checking if a poll is popular looks similar:

```

public class IsPopular extends AbstractSpecification<Poll> {
    @Override
    public boolean isSatisfiedBy(Poll poll) {
        return poll.getLockDate() == null && poll.getVotes().size() > 100;
    }

    @Override
    public Predicate toPredicate(Root<Poll> poll, CriteriaBuilder cb) {
        return cb.and(
            cb.isNull(poll.get(Poll_.lockDate)),
            cb.greaterThan(cb.size(poll.get(Poll_.votes)), 5)
        );
    }
}

```

If we now want to test if a Poll instance matches one of these constraints we can use our newly created specifications:

```

boolean isPopular = new IsPopular().isSatisfiedBy(poll);
boolean isCurrentlyRunning = new IsCurrentlyRunning().isSatisfiedBy(poll);

```

For querying the database we need to extend our DAO / repository to support specifications. This can look like the following:

```

public class PollRepository {
    private EntityManager entityManager = ...

    public <T> List<T> findAllBySpecification(Specification<T> specification) {
        CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
        // use specification.getType() to create a Root<T> instance
        CriteriaQuery<T> criteriaQuery = criteriaBuilder.createQuery(specification.getType());
        Root<T> root = criteriaQuery.from(specification.getType());
        // get predicate from specification
        Predicate predicate = specification.toPredicate(root, criteriaBuilder);
        // set predicate and execute query
        criteriaQuery.where(predicate);
        return entityManager.createQuery(criteriaQuery).getResultList();
    }
}

```

Here we finally use the `getType()` method implemented in `AbstractSpecification<T>` to create `CriteriaQuery<T>` and `Root<T>` instances. `getType()` returns the generic type of the `AbstractSpecification<T>` instance defined by the subclass. For `IsPopular` and `IsCurrentlyRunning` it returns the `Poll` class. Without `getType()` we would have to create the `CriteriaQuery<T>` and `Root<T>` instances inside `toPredicate()` of every specification we create. So it is just a small helper to reduce boiler plate code inside specifications. Feel free to replace this with your own implementation if you come up with better approaches.

Now we can use our repository to query the database for polls that match a certain specification:

```
List<Poll> popularPolls = pollRepository.findAllBySpecification(new IsPopular());
List<Poll> currentlyRunningPolls = pollRepository.findAllBySpecification(new IsCurrentlyRunning());
```

At this point the specifications are the only components that contain the constraint definitions. We can use it to query the database or to check if an object fulfills the required rules.

However one question remains: How do we combine two or more constraints? For example we would like to query the database for all popular polls that are still running.

The answer to this is a variation of the composite design pattern called composite specifications. Using a composite specification we can combine specifications in different ways.

To query the database for all running and popular pools we need to combine the *isCurrentlyRunning* with the *isPopular* specification using the logical and operation. Let's create another specification for this. We name it `AndSpecification`:

```
public class AndSpecification<T> extends AbstractSpecification<T> {
    private Specification<T> first;
    private Specification<T> second;

    public AndSpecification(Specification<T> first, Specification<T> second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public boolean isSatisfiedBy(T t) {
        return first.isSatisfiedBy(t) && second.isSatisfiedBy(t);
    }

    @Override
    public Predicate toPredicate(Root<T> root, CriteriaBuilder cb) {
        return cb.and(
            first.toPredicate(root, cb),
```

```

        second.toPredicate(root, cb)
    );
}

@Override
public Class<T> getType() {
    return first.getType();
}
}

```

An AndSpecification is created out of two other specifications. In isSatisfiedBy() and toPredicate()we return the result of both specifications combined by a logical and operation.

We can use our new specification like this:

```

Specification<Poll> popularAndRunning = new AndSpecification<>(new IsPopular(), new IsCurrentlyRunning());
List<Poll> polls = myRepository.findAllBySpecification(popularAndRunning);

```

To improve readability we can add an and() method to the Specification interface:

```

public interface Specification<T> {
    Specification<T> and(Specification<T> other);
    // other methods
}

```

and implement it within our abstract implementation:

```

abstract public class AbstractSpecification<T> implements Specification<T> {
    @Override
    public Specification<T> and(Specification<T> other) {
        return new AndSpecification<>(this, other);
    }
    // other methods
}

```

Now we can chain multiple specification by using the and() method:

```

Specification<Poll> popularAndRunning = new IsPopular().and(new IsCurrentlyRunning());
boolean isPopularAndRunning = popularAndRunning.isSatisfiedBy(poll);

```

```
List<Poll> polls = myRepository.findAllBySpecification(popularAndRunning);
```

When needed we can easily extend this further with other composite specifications (for example `OrSpecification` or `NotSpecification`).

## Conclusion

When using the specification pattern we move business rules in separate specification classes. These specification classes can be easily combined by using composite specifications. In general, specification improve reusability and maintainability. Additionally specifications can easily be unit tested. For more detailed information about the specification pattern I recommend this article by Eric Evans and Martin Fowler.

You can find the source of this example project on [GitHub](#).

Learn how to build distributed stream processing applications in Java that elastically scale to meet demand- includes reference application. Brought to you in partnership with Hazelcast.

## Like This Article? Read More From DZone



**JPA Implementation Patterns: Saving (Detached) Entities**



**JPA, Asynchronous Processing, and "Leaky Abstractions"**



**Domain Object Persistence**



**Free DZone Refcard  
Getting Started With Vaadin 10**

Topics: [JAVA](#) , [JPA](#) , [SPECIFICATION PATTERN](#)


Published at DZone with permission of Michael Scharhag , DZone MVB. [See the original article here.](#)

Opinions expressed by DZone contributors are their own.


Opinions expressed by DZone contributors are their own.

# Java Partner Resources

Building Real-Time Data Pipelines with a 3rd Generation Stream Processing Engine  
Hazelcast

 Level up your code with a Pro IDE  
JetBrains

 Modern Java EE Design Patterns: Building Scalable Architecture for Sustainable Enterprise Development  
Red Hat Developer Program

 Build vs Buy a Data Quality Solution: Which is Best for You?  
Melissa Data

---