Ticker´s Generation
Working with two transactions

Design and Testing II

Group 6

Ángel Delgado Luna

Belen Garrido López

Ezequiel Portillo Jurado

Alejandro Rodríguez Díaz

Mª de Gracia Piñero Pastor

Index

Foundations 3
Versions 3
First Version. Unique value into entity. 3
Second Version. Setting the unique value as an entity with the unique value 3
Third Version. According to the last version, abstract the object into a generical class. 5
Implementation – Controlling unique values 5
Deliverable D01 and D02 5
Deliverable D03 and D04 6
Deliverable D05 8
Implementation 1 – Ticker System for one entity 8
Implementation 2 – Ticker System for any entity 10

Foundations

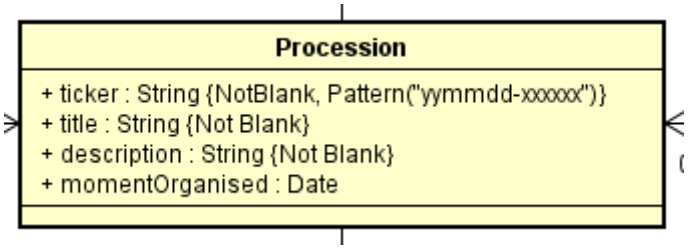
When we are working with objects, any of them may have any relation. For example:

- An actor with an user account whose username is a unique value.
- An entity which contains a unique value. For example: tickers.

Which is the problem to resolve? When working with entities with unique values, the transactions can set into risk cause of the unicity from any of the values. Throwing as consequence an exception from database and retrieving all data to its initial process. So, the point is… How can we avoid this? Let´s go ahead this document!

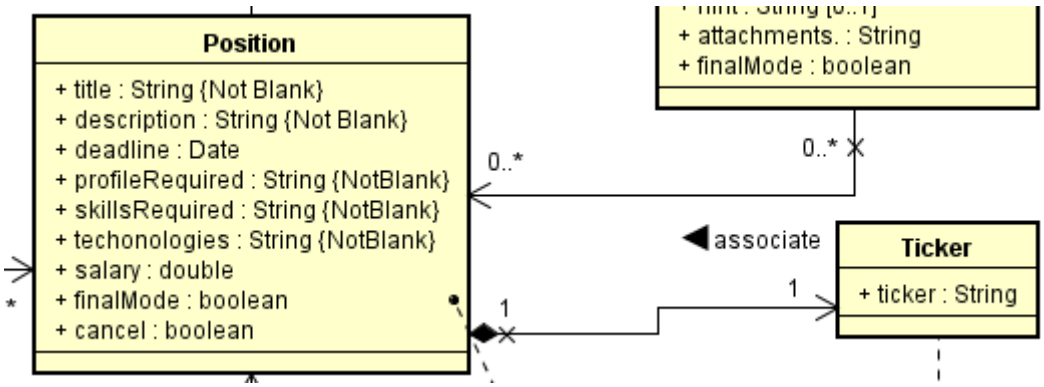
Versions

First Version. Unique value into entity.



This version has the problem mention in the foundations section. It´s considered as a unique value which, in case of error, retrieves all entity data. This can be avoid using the next versions.

Second Version. Setting the unique value as an entity with the unique value

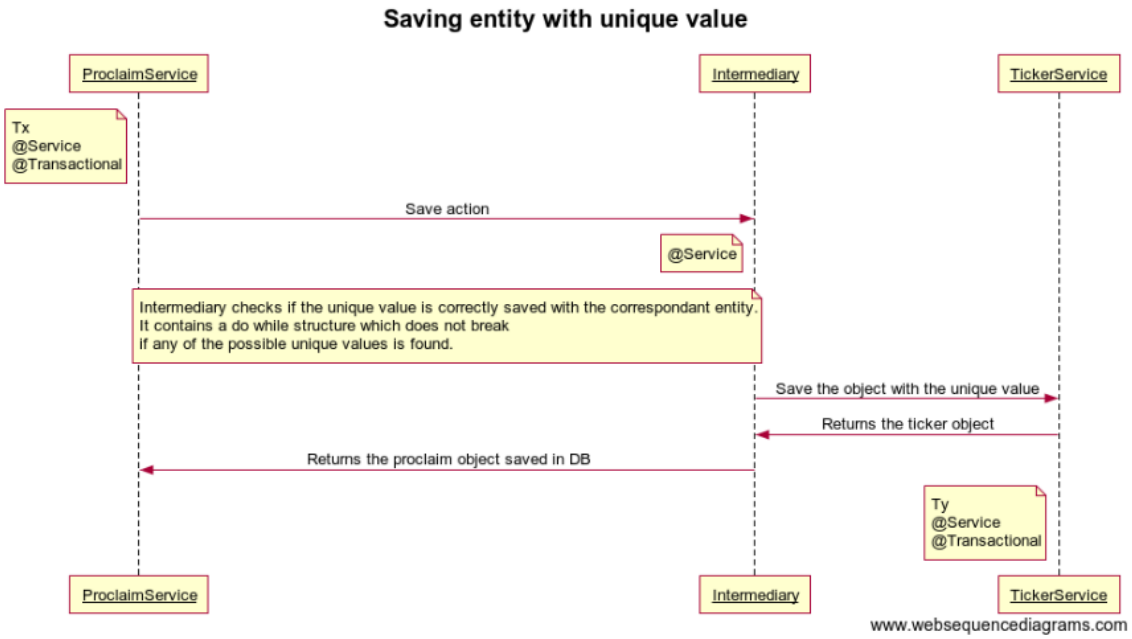


As It can be seen, ticker parameter (considered as an unique string) is now an attribute from an entity. Is the problem completely resolved? No at all. Because, even we can avoid crashing the main object, we have to build a system which lets keeping two transactions working at the same time in a certain operation. We will call this system as an intermediary service. This will apply the save actions for both entity. Considering some details:

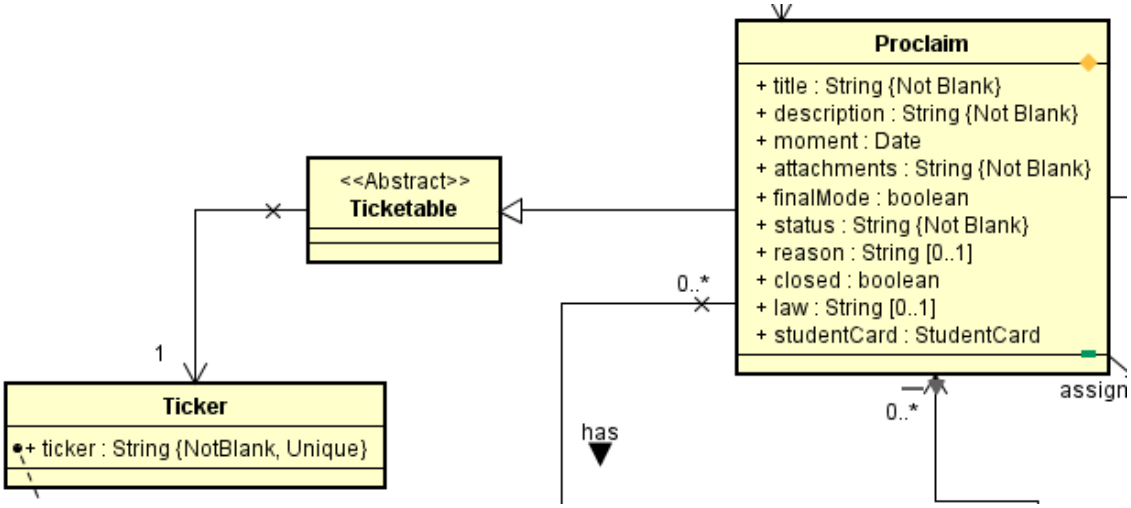
- This new class, is considered a service. But no transactional. This is a service which controls that transactions do not crash when persisting.
- A new transaction for the new entity (In our example: Ticker) is required. With the technology we are using this can be done thanks to the following hibernate annotation:

  @Service
  @Transactional(value = TxType.REQUIRES_NEW)

Graphic



Third Version. According to the last version, abstract the object into a generical class.



The same as before, but it is usable by any entity. It just has to extends ticketable class in order to use this mechanism.

Implementation – Controlling unique values

Deliverable D01 and D02

```
public static String generateTicker(final Collection<String> tickers) {
		SimpleDateFormat formato;
		formato = new SimpleDateFormat("yyMMdd");

		String formated;

		formated = formato.format(new Date());

		final Character[] ch = {
			'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
		};

		String c = "";

		Random random;

		random = new Random();

		for (int i = 0; i < 6; i++)
```

```
                    c += ch[random.nextInt(ch.length)];

                String result;
                result = formated + "-" + c;

                if (tickers.contains(result))
                        Utiles.generateTicker(tickers);

                return result;
        }
```

1. When saving an entity, a query calls to all the tickers.
2. If the collection given by parameter to the ticker generator method contains the ticker produced, a recursive call is done.
3. The recursive call will stop when any of the generated tickers is not saved in database.

Deliverable D03 and D04

Save method in Service

```
public Position save(final Position arg, final boolean duplicate) {
…
                Ticker ticker;
…

                if (duplicate == false)
                        ticker = arg.getTicker();
                else
                        ticker = this.createTicker(c.getCommercialName());
…

                Ticker saveT;
                saveT = this.repositoryTicker.saveAndFlush(ticker);

                arg.setTicker(saveT);

                saved = this.repository.saveAndFlush(arg);

                return saved;
        }
```

1. Save and Flush operations in order to avoid crashing between transactions.

Save method in controller

```
@RequestMapping(value = "/edit", method = RequestMethod.POST, params = "save")
	public ModelAndView saved(final Position position, final BindingResult binding) {
		ModelAndView result = null;

		boolean res = false;

		if (position.isFinalMode()) {
			if (position.getId() == 0)
				res = false;
			else
				res = this.service.problemsByPosition(position.getId()).size()
>= 2;
		} else
			res = false;

		position.setFinalMode(res ? position.isFinalMode() : false);

		if (position.getDeadline().after(new Date()))
			do {
				result = super.save(position, binding,
"position.commit.error", "position", "position/edit", "position/company/edit.do",
"/position/company/list.do", "redirect:list.do");
				if (this.control) {

					Map<String, Object> map;
					map = result.getModel();
					if (map.containsKey("duplicate"))
						this.duplicate = (boolean)
map.get("duplicate");
					else
						this.duplicate = false;

				} else
					this.duplicate = false;
			} while (this.duplicate == true);
		else
			result = super.createAndEditModelAndView(position,
"deadline.future", "position", "position/edit", "position/company/edit.do",
"/position/company/list.do");

		return result;
	}
```

2. If it is tried to save a position with a ticker contained in database, an exception is thrown and caught by the controller. In this case, we have a controller which calls to the save function with a little complement (reached in blue). Blue code checks if a variable named by “duplicate” set to true or false. This value is received from the generic controller built. If it is true, the save action is repeated; else, it get outside from do while structure and finishes the action.

Domain Entity - Position

```
@OneToOne(cascade = {
        CascadeType.ALL
}, optional = false)
@JoinColumn(name = "ticker")
public Ticker getTicker() {
        return this.ticker;
}

public void setTicker(final Ticker ticker) {
        this.ticker = ticker;
}
```

3. JoinColumn joins two columns from different tables into one. This lets us to work with a transaction which has an operation working over another related object. In this case, one to one. Ticker column from proclaim with the id from ticker´s table.

Deliverable D05

Implementation 1 – Ticker System for one entity

Proclaim

```
@OneToOne(cascade = {
        CascadeType.ALL
}, optional = false)
@JoinColumn(name = "ticker")
public Ticker getTicker() {
        return this.ticker;
}

public void setTicker(final Ticker ticker) {
        this.ticker = ticker;
}
```

TickerRepository

```
@Repository
public interface TickerRepository extends JpaRepository<Ticker, Integer> {

        @Query("select t from Ticker t where t.ticker = ?1")
        Ticker findTickerByCode(String code);
}
```

Intermediary

```
@Service
public class ProclaimTickerServiceInter {

   @Autowired
   private TickerService      serviceTicker;

   @Autowired
   private ProclaimRepository  proclaimRepository;



   public Proclaim proclaimWithTicker(final Proclaim without) {

      Proclaim result = null;

      Ticker aux;

      boolean value = false;

      do
         try {
            aux = this.serviceTicker.saveTicker(without.getTicker());
            without.setTicker(aux);
            result = this.proclaimRepository.save(without);
            value = true;
         } catch (final Throwable oops) {
            value = false;
            aux = this.serviceTicker.create();
            without.setTicker(aux);
         }
      while (value == false);

      return result;
   }
```

TickerService

```
@Service
@Transactional(value = TxType.REQUIRES_NEW)
public class TickerService {

	@Autowired
	private TickerRepository	repository;


	public Ticker saveTicker(final Ticker saveTo) {

		Ticker result = null;

		result = this.repository.saveAndFlush(saveTo);

		return result;
	}
```

Implementation 2 – Ticker System for any entity

Ticketable

```
@Entity
@Access(AccessType.PROPERTY)
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Ticketable extends DomainEntity {

	private Ticker	ticker;


	@OneToOne(optional = false, fetch = FetchType.LAZY)
	public Ticker getTicker() {
		return this.ticker;
	}

	public void setTicker(final Ticker ticker) {
		this.ticker = ticker;
	}
```

```
}
```

This class will be the parent class from every entity which wants to use its properties. In our case, the ticker object.

Proclaim

```
@Entity
@Access(AccessType.PROPERTY)
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Table(indexes = {
        @Index(columnList = "title, description, moment, attachments, finalMode")
})
public class Proclaim extends Ticketable {
```

Proclaim, as child class, extends to Ticketable.

GenericRepository

```
@NoRepositoryBean
public interface GenericRepository<K extends Ticketable> extends JpaRepository<K,
Integer> {

        @Query("select t from Ticker t where t.ticker = ?1")
        Ticker findTickerByCode(String code);
}
```

This repository contains a query which checks if any ticker object exists with the given string.

@NoRepositoryBean annotation is put in order to indicate hibernate that this repository is not used at first by any entity.

ProclaimRepository

```
@Repository
public interface ProclaimRepository extends GenericRepository<Proclaim> {
```

TickerRepository

```
@Repository
public interface TickerRepository extends JpaRepository<Ticker, Integer> {

	@Query("select t from Ticker t where t.ticker = ?1")
	Ticker findTickerByCode(String code);
}
```

This query is repetitive into this repository due to the tickers conversion system. TickerToString and StringToTicker require of this query from this repository in order to convert the entity from string to this and viceversa.

ProclaimService

```
	@Autowired
	private TickerServiceInter	interm;

public Proclaim save(final Proclaim aux) {

		Proclaim result;
/…/
		result = this.interm.withTicker(aux, this.repository);

		return result;
	}
```

Intermediary

```
	@Autowired
	private TickerService	service;

	public <K extends Ticketable> K withTicker(final K without, final GenericRepository repository) {

		K result = null;

		Ticker aux = null;

		boolean value = false;

		do
			try {

				if (without.getId() != 0) {

					K auxFromDB;
					auxFromDB = (K) repository.findOne(without.getId());

					boolean check;
```

```
                                        check =
auxFromDB.getTicker().getTicker().equals(without.getTicker().getTicker());

                                        if (!check)
                                                without.setTicker(auxFromDB.getTicker());
                                }

                                if (without.getId() == 0) {
                                        Ticker findByCode;
                                        findByCode =
repository.findTickerByCode(without.getTicker().getTicker());

                                        if (findByCode != null)
                                                throw new IllegalArgumentException();
                                        else {
                                                findByCode = this.service.saveTicker(without.getTicker());
                                                without.setTicker(findByCode);
                                        }
                                }

                                result = (K) repository.save(without);
                                value = true;
                        } catch (final Throwable oops) {
                                value = false;
                                aux = this.create();
                                without.setTicker(aux);
                        }
                while (value == false);

                return result;
        }
```

1. It receives a “Ticketable” object and its repository.
2. Do while iterations are produced until a valid ticker is found. For this, depending of the id of the entity will do one thing or another.
    a. If id is equal to zero. The ticker given in create function to the entity is checked. If this query returns a non-null element, then an exception is thrown and a new do while iteration is produced.
    b. If id is not equal to zero. It´s checked if the ticker of the object is the same as it has saved in database. If not, a new ticker is generated. (This method in the user logic is not necessary because the object is reconstructed. But, in functional testing, it is necessary.)

TickerService

```
@Service
@Transactional(value = TxType.REQUIRES_NEW)
public class TickerService {

        @Autowired
        private TickerRepository        repository;
```

```
public Ticker saveTicker(final Ticker saveTo) {

	Ticker result = null;

	result = this.repository.saveAndFlush(saveTo);

	return result;
}
```