VIRTUALMIND

1) SQL

```
CREATE TABLE `topic` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `post` (
    `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
    `topic_id` int(11) unsigned NOT NULL,
    `title` varchar(16536) NOT NULL,
    `text` varchar(16536) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `topic_id` (`topic_id`),
  CONSTRAINT `post_ibfk_1` FOREIGN KEY (`topic_id`) REFERENCES `topic` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Considering the schema definition above, please write the queries that return:

A. The first 10 topics in alphabetical order
```
SELECT t.*
FROM topic AS t
ORDER BY name
LIMIT 10;
```
B. The name of each topic with 5 or more posts.
```
SELECT t.name
FROM topic AS t
INNER JOIN (
        SELECT topic_id, count(topic_id) AS occurrences
        FROM post
        GROUP BY topic_id
        HAVING occurrences >= 5) AS p
ON t.id = p.topic_id;
```
C. All the topics without any post.
```
SELECT t.*
FROM topic AS t
LEFT JOIN post AS p ON p.topic_id = t.id
WHERE p.id IS NULL;
```

2) Java

A. Using the tables from above, write the code of the Java classes to model them, including the JPA annotations needed for persisting them. Getters and setters can be omitted.

```
/**CLASS TOPIC**/
package com.example.demo.virtual_mind.entities;

import java.io.Serializable;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "topic")
public class Topic implements Serializable {

        /**
         *
         */
        private static final long serialVersionUID = 154178329911147264L;

        @Id
        private Integer id;

        @Column(nullable = false)
        private String name;

        @OneToMany(fetch = FetchType.LAZY, mappedBy = "topic")
        private List<Post> posts;

        public Topic() {
        }

        public Integer getId() {
                return id;
        }
```

```
	public void setId(Integer id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public List<Post> getPosts() {
		return posts;
	}

	public void setPosts(List<Post> posts) {
		this.posts = posts;
	}
}


/** CLASS POST **/
package com.example.demo.virtual_mind.entities;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "post")
public class Post implements Serializable {

	/**
	 *
	 */
	private static final long serialVersionUID = -2772115536871352240L;
```

```
@Id
private Integer id;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "topic_id", nullable = false)
private Topic topic;

@Column(nullable = false)
private String title;

@Column(nullable = false)
private String text;

public Post() {
}

public Integer getId() {
		return id;
}

public void setId(Integer id) {
		this.id = id;
}

public Topic getTopic() {
		return topic;
}

public void setTopic(Topic topic) {
		this.topic = topic;
}

public String getTitle() {
		return title;
}

public void setTitle(String title) {
		this.title = title;
}

public String getText() {
		return text;
}
```

```
        public void setText(String text) {
                this.text = text;
        }
}
```

B. Analyze the following code:

```
1   public class TopicService {
2
3       @PersistanceContext
4       private EntityManager entityManager;
5
6       @Transactional
7       public void updateTopic(long topicId, TopicDTO updated) {
8           Topic topic = entityManager.find(Topic.class, topicId);
9           if (topic != null) {
10               topic.setName(updated.getName());
11          } else {
12               throw new EntityNotFoundException(Topic.class, topicId);
13          }
14      }
15  }
```

1. Please, enumerate all the queries executed when the `updateTopic` method is called, including transactional sentences. Point out in which line code of the aforementioned method each query is executed.

(1) In line number 8 a select query is executed:
(2) After finishing the execution of the method, the @Transactional annotation gives the instruction to flush the topic entity, so, an update query is executed.

2. Imagine that the method `updateTopic` is called from an `update` method of a RESTful MVC controller named `TopicController`. Please, point out the first line of the HTTP request that would be the result of calling such method (specify HTTP method and URL where you would have mapped the controller method).

```
@RestController
@RequestMapping("topics")
public class TopicRestCtrl {

        @Autowired
        TopicService topicService;

        @PutMapping("/{id}")
        ResponseEntity<Topic> update(@Param("name") String name) { … }
```

3. In the event that the result is an `EntityNotFoundException` being throw, what would be the HTTP status that the `TopicController` should return to its clients?

It should be HTTP 404 NOT FOUND
For example, if the resource with id 5 doesn't exist, It means that the resource requested by the url http://{ip}:{port}/topics/5 is not found

C. Analyze the following code:

```
public class PostService {

    @PersistenceContext
    private EntityManager entityManager;

    private int pageSize = 25; // In real life would be configurable

    public List<PostDTO> listPostTitlesAndTopics(int pageNumber) {
        List<Post> posts = entityManager
            .createQuery("SELECT p FROM Post p")
            .setFirstResult((pageNumber - 1) * pageSize)
            .setMaxResults(pageSize)
             .getResultList();
        List<PostDTO> result = new ArrayList(posts.size());

        for(Post post : posts) {
            PostDTO postDto = new PostDTO();
            postDto.setId(post.getId());
            postDto.setTitle(post.getTitle());
            postDto.setTopicName(post.getTopic().getName());
            result.add(postDto);
        }

        return result;
    }
}
```

Which bottlenecks towards the DB does the `listPostTitlesAndTopics` method present, and how would you fix them?

(1) It is important that in the definition of the Post Entity we take care of making the @ManyToOne relationship with Topic, with a Lazy fetching policy, that way we avoid the overhead of querying additional information that is not being used
(2) For this case it's preferable to make a custom query like this:

SELECT p.id, p.title, t.name
FROM Post p
JOIN p.topic t

Or inclusive, if we need special performance in these query, we can use native sql in conjunction with plain JDBC and in the same way as shown above, only retrieving the needed fields.

(3) It’s also better to fill the DTO in the same step where we parameterized the query like this:
SELECT new PostDTO(p,id, p.title, t.name)
FROM Post p
JOIN p.topic t

D. During a code review, we found the following code fragment:

```
@lombok.Data
class DateBucket {
      final Instant from;
      final Instant to;

      public static List<DateBucket> bucketize(ZonedDateTime fromDate,
                  ZonedDateTime toDate,
                  int bucketSize,
                  ChronoUnit bucketSizeUnit) {
          List<DateBucket> buckets = new ArrayList<>();
          boolean reachedDate = false;
          for (int i = 0; !reachedDate; i++) {
              ZonedDateTime minDate = fromDate.plus(i * bucketSize, bucketSizeUnit);
              ZonedDateTime maxDate = fromDate.plus((i + 1) * bucketSize, bucketSizeUnit);
              reachedDate = toDate.isBefore(maxDate);
              buckets.add(new DateBucket(minDate.toInstant(), maxDate.toInstant());
          }

          return buckets;
      }
}
```

1. Explain briefly what is the `bucketize` method trying to do.
   It is forming a list of buckets, each bucket having a starting time and an ending time.
   Starting and ending time in a specific bucket stands for a starting time lets say x

and an ending time equals to x + (one bucketSizeUnit).
The first bucket will coincide with fromDate parameter in its from attribute.

In general terms what this method does is to take two dates: startDate and endDate and generate buckets for successive intervals equals to bucketSizeUnit (also provided), until the last generated bucket exceeds the endDate.

2. Refactor the method to use Java 8 Streams.

```
public static List<DateBucket> bucketizeWithStreams(ZonedDateTime fromDate,
ZonedDateTime toDate, int bucketSize,
                ChronoUnit bucketSizeUnit) {
        List<DateBucket> buckets = new ArrayList<>();
        Stream<Integer> stream = Stream.iterate(0, i -> i + 1);
        stream.map((i) -> {
                ZonedDateTime minDate = fromDate.plus(i * bucketSize, bucketSizeUnit);
                ZonedDateTime maxDate = fromDate.plus((i + 1) * bucketSize,
bucketSizeUnit);
                boolean reachedDate = toDate.isBefore(maxDate);
                buckets.add(new DateBucket(minDate.toInstant(), maxDate.toInstant()));
                return reachedDate;
        }).takeWhile((p) -> !p);
        return buckets;
    }
```

E. Describe a design pattern that you have used before (other than Singleton), the problem you had to resolve, and how the pattern helped you to fix such a problem.

   The Adapter pattern: the main idea with this pattern is to make an incompatible interface or functionality, usable for us.
   For example, if our program understands and deals with JSON but a functionality we know that works great only understands XML, we can apply the adapter pattern to wrap the XML functionality, decouple the concerns of parsing from and to JSON and XML, and that way, providing the JSON version of the functionality (which is just a wrapper) and which we can cleanly operate with.
   A real world problem I fixed with the implementation of this pattern was:

   - In a Java back-end we had some services that many clients used to produce electronic invoices, each client being a company with many users indeed.
   - After the generation of each electronic invoice, a service that communicates with the DIAN (National Customs and Taxes Address) has to be consumed. The trouble here is that each client (company) has its own implementation of this

service, so, for each client we have to give support to consume the particular service, having the same base information, but using the specific format supported by each implementation variant of the service.

- As a summary of the problem, we have a set of information in the form of Java Beans (entities) and we needed to set up a specific format (JSON, XML, CSV, Key value pairs, among others formats), a specific name for properties and specific headers for the request, for each variant that each company provided for the service.
- The solution was to implement an Adapter pattern, in fact, there were many adapters that wrap the information in the format we already manage and then each adapter converts the information in the specific format requested for the service being used.
- Additionally, on top of the adapters, we used a strategy pattern to organize and dynamically decide which adapter to use depending on which version of the service we had to consume.

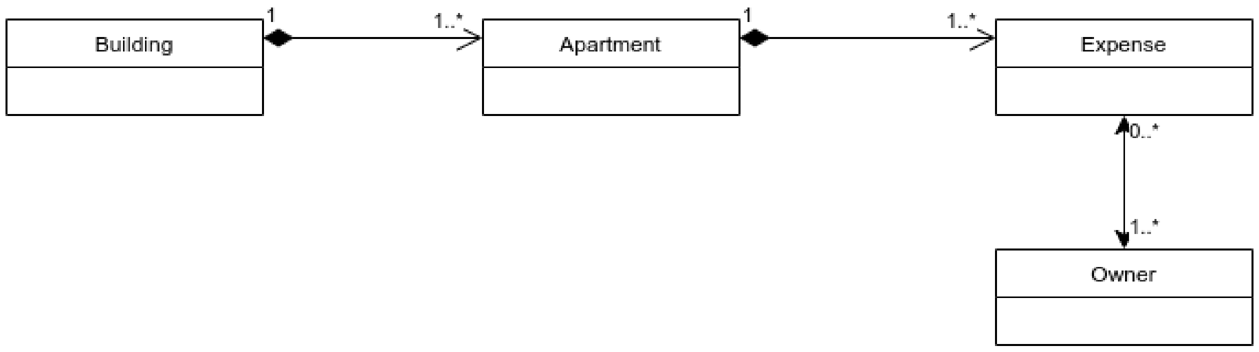
VIRTUALMIND

3) Modeling

In an already existing system for apartment buildings expenses management, the following classes were found:

- Building
- Apartment
- Owner
- Expense

A. Which relationships do you consider should be among the classes described above? You may add a diagram.



B. It was necessary to repair the stair landing zone, with a cost of $10.000. Which class would you assign the responsibility of registering and storing that expense to? It may not be among the 4 initial classes. Write the method signature.

```
public interface ExpenseService {

        //THIS IS THE METHOD SIGNATURE ASKED
        Expense register(Expense expense);

        //other method signatures belonging to ExpenseService
}

public class ExpenseServiceImpl implements ExpenseService {

        @Autowired ExpenseRepository expenseRepo;

        @Override
        public Expense register(Expense expense) {
            return expenseRepo.save(expense);
        }

        //other method definitions belonging to ExpenseServiceImpl
    }
```

C. The end of the month is near, and expenses have to be settled. The settlement process consists of:

   - Calculate the total expenditures of the period.
   - Divide them among the apartments.
   - Issue the billing documents.
   - Notify the owners.

   Which classes would you assign to be responsible for each step? They may not be among the 4 initial classes, in which case you will have to name and describe briefly the fundamental responsibility of the new classes.

```
/**This class will be in charge of calculating the total expenditures of the period, for that, it will take into account a given cutoff date to start the computation**/
@Service
public class GeneralExpendituresCalculator { … }

/** This class will be in charge of distributing the expenses among the apartments given the business rules **/
@Service
public class ExpenseDistributor { … }

/**This class will be in charge of generating the billing documents, for that, to this class it is parameterized a document template and the arguments to the query for retrieving the information to generate the billing documents, for example, a parameter could be when expenseDocumentGenerated=false, that way, this class will generate billing documents**/
@Service
public class BillingGenerator { … }


/** This class will be in charge of the notification process which only consists in receiving an owner and the corresponding Set of expenses to be notified to that owner.
Here we don't care about querying specific owners to be notified. We keep this class as simple as possible just providing the service of notification and that way it comply with SOLID (reusability)**/
@Service
```

public class ExpenseNofifier { … }

Additional Note: specially the classes BillingGenerator and GeneralExpendituresCalculator could involve intensive processing and for that reason, these two classes could be good candidates for being treated as asynchronous services (consumers) each one consuming from its respective queue service and then notifying about the finalization of a process instance.

4) English

Write a few lines below to describe the biggest challenge you had in your professional life and how you solved it.

I haven't had only one biggest challenge I had to solve, better, I think there are many big challenges I can speak of:

- One big challenge was to process an input stream of images in very limited hardware.
  There were about 30 images coming per second and we had to process all of them in almost real time. We optimize as much as we could using multithreading to squeeze at most the whole processor, and many other techniques for performance improvement, the idea was to discard as many useless images as possible as soon as possible in the pipeline, we also started to prioritize the different algorithms that discard images running first those with lower computational complexity. At the end of the process we got the images we needed for conforming a face model for a face recognition process we did with a third-party provider.
- Another big challenge was to make an arithmetic formula editor and solver which included sum, subtraction, multiplication, division, equality, greater than, lower than and negation operators; it also contained conditional structures and variable resolution for types boolean and numeric. The purpose was that the user could set variables in the database and then use them in a dynamically created formula editor and solver.
  We build that solver using ANTLR (Another Tool For Language Recognition), defining our own grammar for a formula and integrating it with JPA in order to resolve variable values coming from the database.