Exam assignment
SD22

Rasmus Møller
Rasmus Trap

Backend URL:
https://github.com/RasmusKEA/crispy-funicular
In the above repository are all three back-ends found as well as database dump files.

20/12-22

1 Introduction

1.1 Problem description

The goal of this assignment is to design and create a database which works according to best practice. We are going to make an application where you can post reviews of video games and give them a rating. This review can be commented on and updated by the author. There will therefore be assigned 3 different roles to a user. User, staff and admin. The user role can view the reviews and comment, where the staff role can post the reviews and update them. Admin has access to the data and can post and delete, he also allocates the roles.
We will create three different databases and a dedicated CRUD backend to each. A relational database in MySQL, a GraphQL database in neo4j and a document database in MongoDB.

In this report we will go over how we design our relational database, the normalization of the database and what tools we are using. We will go over what transactions are and what kind of stored objects we are using. What do we use an ORM for and what kind of ORM we are using.
We will explain the three kinds of databases we have constructed and what kind of similarities and differences they may have.

1.2 Explanation of choices for databases and programming languages and other tools

For our final assignment we are asked to create a functional CRUD-backend with three different databases. We will create three similar backends with three different types of databases. A GraphQL database using neo4j. The Document database will be created with Mongoose as an ODM for MongoDB as the database platform. The SQL database will be created with MySQL workbench and sequelize as the ORM for the application.
We have chosen Node.js as our backend. This is where we feel the most comfortable and when we have to implement this many new terms into our program. We felt that the best outcome will come if we use what we are comfortable with. We discussed other languages but we like the structure behind Node.js, so we used it to create all three types of CRUD applications.

1.3 Explanation of application

This application will be a platform that contains reviews on video games. The content will be an article with a review score and a comment section. We will accomplish this using Node.js, and different kinds of databases. We will provide different roles to users using JWT.

1.4 API end-point description

End-point	**Method**	**Description**
/api/review/	POST	End-point for creating a review. Body should include: idUser: String review: String title: String rating: Number ratingReasoning: String platform: String image: String featured: Number
/api/review/	GET	Retrieves all reviews
/api/review/:id	GET	Retrieves a single review by ID
/api/review/:id	PUT	Updates one or more fields in an existing review. Body should contain at least one or more of the following: idUser: String review: String title: String rating: Number ratingReasoning: String platform: String image: String featured: Number
/api/review/:id	DELETE	Deletes a review by ID
/api/review/featured	GET	Retrieves the currently featured review. This is defined where the field *'featured'* is equal to 1.
/api/comment/	POST	Creates a new comment. Body should contain: idUser: String idReview: String userComment: String
/api/comment/	GET	Retrieves all comments
/api/comment/all/:id	GET	Retrieves all comments based on idReview
/api/comment/:id	GET	Retrieves one comment based on the comments ID
/api/comment/:id	PUT	Updates one or more fields in an existing comment. Body should contain at least one or more of the following: idUser: String idReview: String userComment: String

/api/comment/:id	DELETE	Deletes a comment by ID
/api/auth/signup	POST	Creates a new user. Body should contain: username: String email: String password: String
/api/auth/signin	POST	Logs a user in. Body should container: username: String password: String

2 Relational database

2.1 Intro to relational database

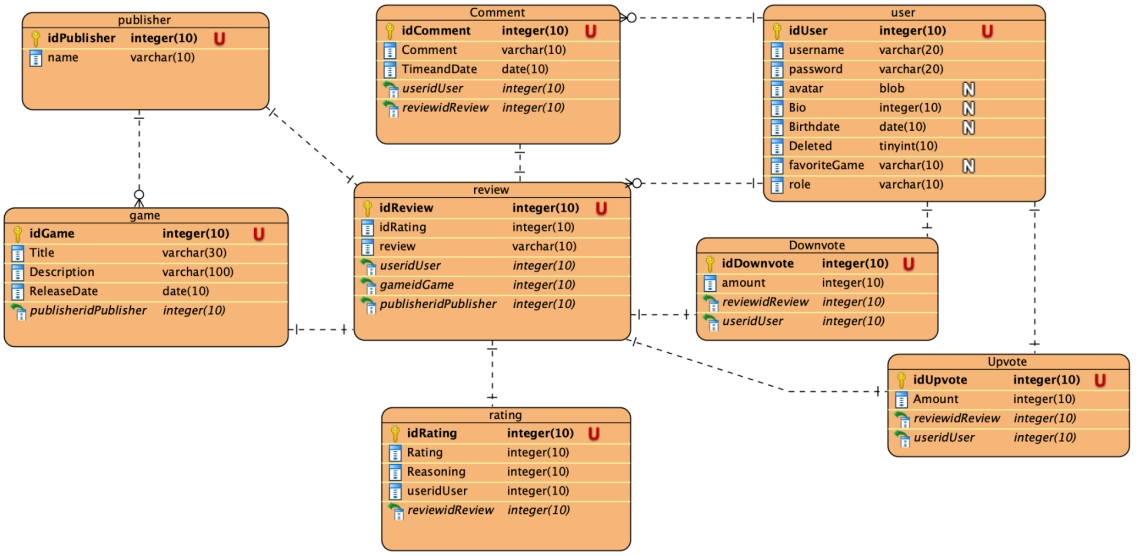
A relational database is a collection of data that, as the name indicates, has predefined relationships describing where and how data is stored in one or more tables.
These relationships are logical connections between these tables based on the basis of interaction between these tables.

2.2 Database design

2.2.1 Entity/Relationship model (Conceptual -> Logical -> Physical)

User	- UserID - Username - Password(hashed) - Avatar (optional/default) - Bio (optional) - Birthday (optional) - Deleted (boolean) - FavoriteGame/FavoritePublisher
Game	- GameID - PublisherID - Title - Description - ReleaseDate
Publisher	- Publisher ID - Name
Review	- ReviewID - GameID - PublisherID - UserID - RatingID - Review
Rating	- RatingID - ReviewID - UserID - Rating - Reasoning
Upvote	- UpvoteID - ReviewID - Amount
Downvote	- DownvoteID - ReviewID - Amount
Comment	- CommentID - UserID - ReviewID - Comment - TimeAndDate

The table above shows what our initial idea was regarding the database. It is our version of the initial conceptual ER-diagram. This shows no actual relations except for how we thought the foreign keys would be.



This is the initial er-diagram
Ignore the length of the integers, visual paradigm bug.

Our logical ER diagram shows the relations between our tables. One user can do several reviews and comments, but a comment can for example only be linked to one user. One publisher can also have several games.

We also had a rating table, because we wanted to have a rating of the game for every review. That did not make sense though, so we decided to put the rating inside of the review table, because every rating would have had the same review id anyway. So the primary key would always have been the same on both tables.

2.2.2 Normalization process

Normalization. The practice and design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies. Normalization rules divide larger tables into smaller tables and link them using relationships.

In theory there are 6 normalization forms, however, in most practical applications, normalization achieves its best in 3rd Normal Form.
The first normalization form says:

- Each table cell should contain a single value.
- Each record needs to be unique.

We've achieved this. An example is looking at our comments-table. We have every single comment as an individual entry in our database. In violation to the first normalization form, we could have had all comments made by one user comma separated in a row.

The second normalization form says:

- Be in 1NF
- Single Column Primary Key that does not functionally dependant on any subset of candidate key relation

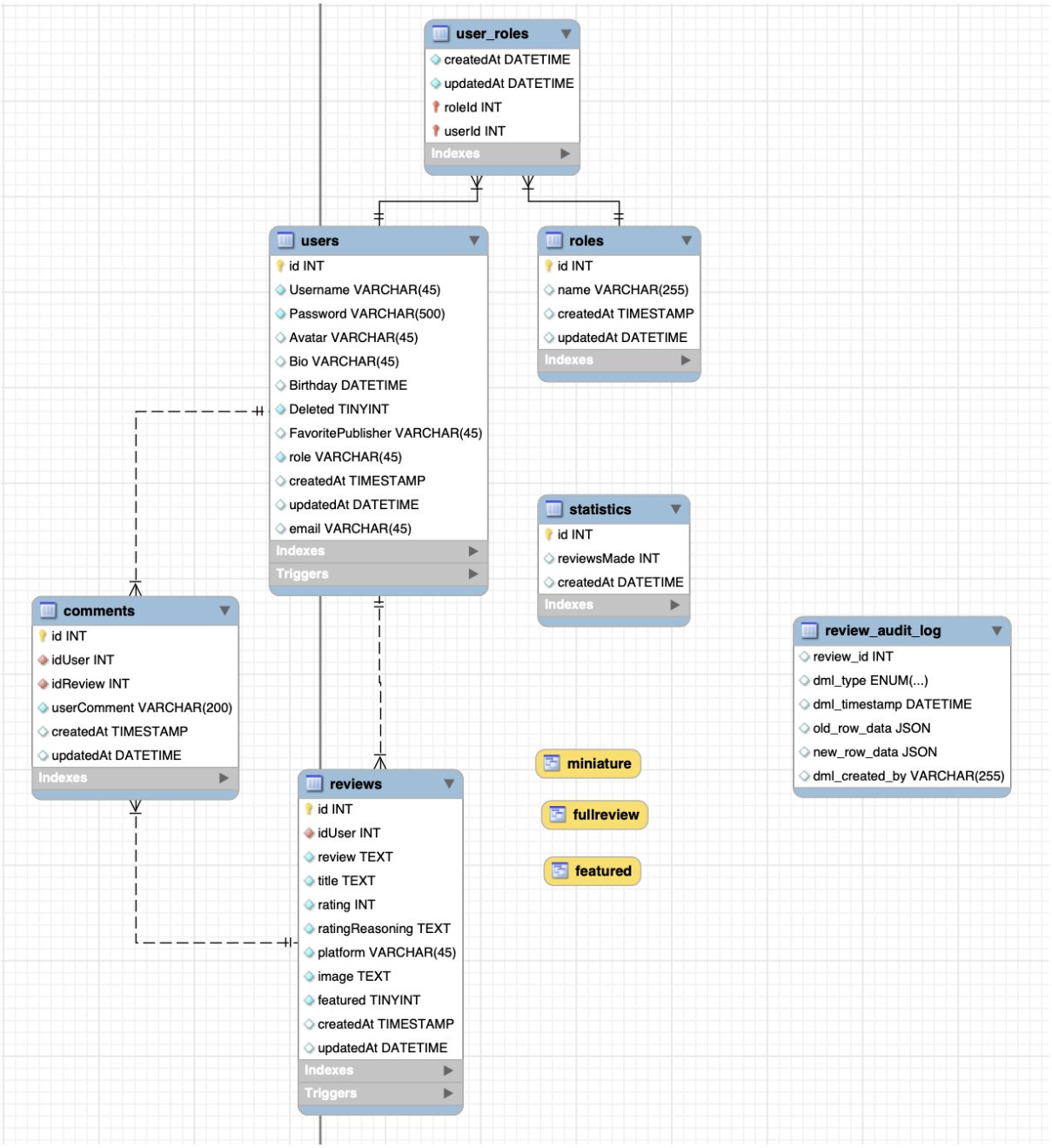
All of our tables only include relevant information to each corresponding table in order to fulfill the second normalization form as well as foreign keys to establish a relation between tables.

The third normalization form says:

- Be in 2NF
- Has no transitive functional dependencies

We have achieved no transitive functional dependencies as we have no columns that may change due to an update in another column.

2.3 Physical data model



On the table above we see our physical ER-diagram which is made using MySQL Workbenchs tool. This shows our actual relations, tables, views and our audit-table. It is quite obvious that this is quite far from our logical ER-diagram, which is the result of many iterations to achieve what worked best for our application.

2.3.1 Data types

When setting data types; e.g. looking at any table we have set "*createdAt*" and "*updatedAt*" as a DATETIME, to ensure the integrity of the data in our database. In theory, it could've been a VARCHAR, but that is a vague and indefinite description of what that column actually is.
When ensuring the data integrity, we also increase the security and minimize the risk of users inputting something that can or may break the database or the program code.

2.3.2 Foreign Keys

When setting foreign keys upon creating a new table in a relational database like MySQL, we also ensure the integrity of the data in our database. The foreign key constraint is used to prevent actions that would destroy links between tables, as the foreign key is a field in one table, that refers to the primary key in another table. The foreign key is also a part of creating the relations between tables; an example is the two one-to-many relations that goes from our comment table to both the user table and the review table.

2.3.4 Constraints and referential integrity

As for constraints and referential integrity we've used foreign keys as described above. We have also decided to CASCADE on delete when deleting a review. This makes sure that we delete all comments on the review at the same time, so that we are not using any space or memory in our database for no reason.

2.3.5 Transactions

Transactions is grouping of statements that run when more than one operation needs to be completed simultaneously. It is a necessary tool for developers to know, to simplify our code and upkeep the database integrity. It is used to account for errors that can happen within a database.
In our project we have not found it relevant to use transactions. But the tool is necessary to know as developers since we will come across databases that more than one operation needs to be run simultaneously.

2.3.6 User Privileges

In our application we have three different roles we assign. User is our first role and the user has the least access to the application. A user can view the different reviews and comment on them. The user can also update his profile information. Staff is our next role, the staff role can post reviews and update them. The staff role also has all the access that the User Role possesses. The admin role is the one who assigns the roles, he also has access to the data. He has the most Read and Write privileges.

2.3.7 Security

SQL injections is a vulnerability within an application, an attacker can interfere with the database queries. It can allow the attacker to view data they normally are not allowed to. An SQL injection can even allow the attacker to modify and delete data. This is something you have to take into account as a developer.
To secure our application for such attacks did we use encryption of a user's passwords. In case of a SQL injection of the user table. This allows us to ensure that only a user knows what their password is. Not even us as developers can see the password.
We also used JWT as a personal token to ensure that a user cannot access all of the data. We as developers can protect different endpoints from unauthorized users using JWT.
Another layer of security is that we use the Sequelize ORM. This prevents us from having raw SQL-statements in our code which could be manipulated. Instead we use the built-in functions Sequelize provides.

2.3.8 Indexes

An index is a data structure available in MySQL, which allows us to locate specific data without scanning all rows in a table for a given query. In all simpleness; creating an index helps us retrieve data faster.
Indexes make the best sense when having a large set of data, as its effect scales with the amount of data in a table. We have however implemented an index on our *review* column in our *reviews* table.

```
ALTER TABLE reviews ADD INDEX review_idx (review(25));
```

This is our *review_idx*-index on our *reviews*-table. The point of this index is to search through entire reviews faster, with a maximum search-key of 25 characters.
An example query could be:

```
SELECT * FROM rategame.reviews WHERE review LIKE "%War%";
```

Which returns all reviews written about games including the word "War". This would become handy with a large dataset.

2.4 Stored objects - stored procedures / functions, views, triggers, events

1) Stored procedures

So far we have implemented two stored procedures. One named *CommentsOnReviews* which takes a parameter *reviewID*. This gathers all comments regarding one review, as we recognize that we are going to use this call quite a few times.
We also have a *UserProfileComments* stored procedure, which gathers all comments made by one user. This is for the same reason as the other one.

```
CREATE DEFINER=`user`@`%` PROCEDURE `CommentsOnReviews`(
    IN reviewID INT
)
BEGIN
SELECT UserComment, createdAt FROM comments WHERE idReview = reviewID;
END
```

2) Stored functions

A stored function in MySQL is a set of SQL statements that perform some task and return a single value. We have created a single stored function, which sets a users “writers level” depending on the amount of reviews one has written.
The stored function looks like this:

```
CREATE DEFINER=`root`@`localhost` FUNCTION `WritersLevel`(
    amountOfReviews int
) RETURNS varchar(20) CHARSET utf8mb4
    DETERMINISTIC
BEGIN
    DECLARE writers_level VARCHAR(20);
    IF amountOfReviews > 50 THEN
        SET writers_level = 'Advanced';
    ELSEIF (amountOfReviews <= 50 AND
            amountOfReviews >= 20) THEN
        SET writers_level = 'Intermediate';
    ELSEIF amountOfReviews < 20 THEN
        SET writers_level = 'Novice';
    END IF;
    -- return the customer occupation
    RETURN (writers_level);
END
```

3) Views

A view is a virtual table based on the result of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database. We've created three different views. A view that returns the featured review, a view that returns a full review containing all details about a review, and lastly a view that returns less details about a review.

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = `root`@`localhost`
    SQL SECURITY DEFINER
VIEW `featured` AS
    SELECT
        `reviews`.`title` AS `title`,
        `reviews`.`rating` AS `rating`,
        `reviews`.`image` AS `image`,
        `reviews`.`review` AS `review`,
        `reviews`.`featured` AS `featured`
    FROM
        `reviews`
    WHERE
        (`reviews`.`featured` = 1)
```

4) Triggers

A trigger is a stored program invoked automatically in response to an event such as insert, update, or delete that occurs in the associated table. For this assignment we have implemented a trigger before INSERT and before UPDATE that makes any username to lowercase.

5) Events

MySQL Events are tasks that run according to a schedule. Scheduled tasks of this type are also sometimes known as "temporal triggers", implying that these are objects that are triggered by the passage of time. For this assignment, we have implemented an event that once a week inserts into our statistics table the amount of reviews created during the past week.

Db	Name	Definer	Time zone	Type	Execute at	Interval value	Interval field	Starts	Ends	Status
rategame	weeklyReviews	root@localhost	SYSTEM	RECURRING	NULL	1	WEEK	2022-11-17 16:52:21	NULL	ENABLED

2.6 Auditing. Explanation of the audit structure implemented with triggers

Auditing is when validating that changes are made correctly. Triggers are ideal for this, as they are invoked when a change is made by either INSERT, UPDATE or DELETE. As these executions are critical and can mess up the entire database, auditing makes it possible to ensure the integrity of the data and verifying that the execution has gone well.

Trigger	Event	Table	Statement	Timing
review_insert_audit_trigger	INSERT	reviews	INSERT INTO review_audit_log (review_i…	AFTER
review_update_audit_trigger	UPDATE	reviews	INSERT INTO review_audit_log (review_i…	AFTER
review_delete_audit_trigger	DELETE	reviews	INSERT INTO review_audit_log (review_i…	AFTER

2.7 ORM - Sequelize

We have used the Sequelize ORM to develop our database's backend. An ORM makes it possible for the developer to work with a database in a somewhat object-related manner.
The biggest benefit we've seen is that we are not required to write pure SQL-statements only. This is because Sequelize comes with some built-in functionality to query for data - e.g. *findByPk()* which is all you need to write in order for Sequelize to find one entry where id is given as a parameter.
This is also helpful to prevent SQL-injection, as when we are not writing raw SQL-statements, it isn't as easy to tamper with.

```
exports.create = (req, res) => {
  // Validate request
  if (!req.body.review) {
    res.status(400).send({
      message: "Content can not be empty!",
    });
    return;
  }

  // Create a Review
  const review = {
    idUser: req.body.idUser,
    review: req.body.review,
    title: req.body.title,
    rating: req.body.rating,
    ratingReasoning: req.body.ratingReasoning,
    platform: req.body.platform,
    image: req.body.image,
  };

  // Save Review in the database
  Review.create(review)
    .then((data) => {
      res.send(data);
    })
    .catch((err) => {
      res.status(500).send({
        message:
          err.message || "Some error occurred while creating the Review.",
      });
    });
};
```

3 GraphQL - Neo4j

3.1 What is Neo4j

Neo4j is a database platform that stores the data it is provided. The data will be stored in nodes and maintain data relationships. It is a good tool when working with graph databases, and can help you visualize the data in a manageable way.

The GraphQL database is different from the relational database. Instead of creating the tables and columns before you create the backend. The database runs alongside your code, meaning that it is an API query language. That means that we create our data in runtime.

3.2 Cypher

Cypher is the graph query language that neo4j uses, and is the way you retrieve the data. Cypher provides a visual way of handling the data relationships and patterns.
Example: "*MATCH (u:User) WHERE u.username = "${req.body.username}" OR u.email = "${req.body.email}" return u*"
This is an example from our backend where we define a user. So we start defining user and then we give the user its labels. So the query language is very intuitive and can be very helpful if the queries get complicated.

3.3 Code examples

```
const router = require("express").Router();
const Review = require("../model/review.model.js");

module.exports = (app) => {
  router.post("/signup", async (req, res, next) => {
    // Create User

    let existingUser = await req.neo4j.read(
      `MATCH (u:User) WHERE u.username = "${req.body.username}" OR u.email = "${req.body.email}" return u`
    );

    if (!existingUser.records.length) {
      req.neo4j
        .write(
          `
        CREATE (u:User{
          id: randomUUID(),
          username: "${req.body.username}",
          password: "${req.body.password}",
          email: "${req.body.email}"})
          return u
      `
        )
        // Convert to user entity
        .then((res) => {
          const review = new Review(res.records[0].get("u"));

          return {
            ...review.toJson(),
          };
        })
        // Return the output
        .then((user) => res.json({ user }))
        // Pass any errors to the next middleware
        .catch(next);
    } else {
      res.status(400).send({
        message: "Failed! Email or username is already in use!",
      });
    }
  });
```

In our user route, we create a node of data by giving the user a unique id, username, password and email. The data will be saved in our neo4j dbms, and create a node with all the data stored into it.
neo4j library allows us as developers to run the queries in the api instead of handling it in the database itself.

```
89
90      router.delete("/:id", (req, res, next) => {
91        console.log(req.params);
92        let { id } = req.params;
93        console.log(req.body);
94        req.neo4j
95          .write(
96            `MATCH (r {id: "${id}"})
97            DELETE r`
98          )
99          // Return the output
100         .then(res.status(200).send("Deleted succesfully"))
101         // Pass any errors to the next middleware
102         .catch(next);
103     });
104
105     app.use("/api/review", router);
106   };
```

```
16  router.get("/:id", (req, res, next) => {
17    let { id } = req.params;
18    req.neo4j
19      .read(`Match (r:Review {id: "${id}"}) return r`)
20      .then((result) => {
21        return result.records.map((i) => i.get("r").properties);
22      })
23      .then((json) => res.send(json))
24      // Pass any errors to the next middleware
25      .catch(next);
26  });
27
```

```
59    router.put("/:id", (req, res, next) => {
60      console.log(req.params);
61      let { id } = req.params;
62      console.log(req.body);
63      req.neo4j
64        .write(
65          `MATCH (r {id: "${id}"})
66          SET r.idUser = "${req.body.idUser}",
67          r.review = "${req.body.review}",
68          r.title = "${req.body.title}",
69          r.rating = "${req.body.rating}",
70          r.ratingReasoning = "${req.body.ratingReasoning}",
71          r.platform = "${req.body.platform}",
72          r.image = "${req.body.image}",
73          r.featured = "${req.body.featured}"
74          return r`
75        )
76        // Convert to user entity
77        .then((res) => {
78          const user = new Review(res.records[0].get("r"));
79
80          return {
81            ...user.toJson(),
82          };
83        })
84        // Return the output
85        .then((user) => res.json({ user }))
86        // Pass any errors to the next middleware
87        .catch(next);
88    });
```

Here are some examples of the review.routes to show how we do the, create, read, update and delete with neo4j. This is how we do the data changes in our graph database. Now we can fill up the neo4j database with test data by running the application and using postman.

4 Document database - MongoDB

4.1 What is MongoDB

MongoDB is a document-oriented NoSQL database, meaning that it stores data in flexible, JSON-like documents which maps to objects in our code. These documents are stored in so-called collections, which translates into tables known from MySQL.

Some primary features and benefits of MongoDB is:

1) The documents (which would translate into rows in SQL) doesn't need to have a schema beforehand and can therefore be created on the go.
2) Scalability - As there's no actual structure within the database itself, it is easy to scale up and have enormous clusters.
3) Aggregation - It allows to perform operations on the grouped data and get a single result or computed result.

4.2 Mongoose

Mongoose is an ODM (Object Data Model) library for MongoDB and NodeJS which we've used for our MongoDB application. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in our MongoDB-database.

4.3 Code examples

To give an example from our code, we will show how a review is posted and traverse through the code.

At first an HTTP POST-request is sent to our end-point "*/api/review/*". Upon reaching this end-point the controller-method *"create"* is called.

```
router.post("/", reviews.create);
```

When the "*create*" method is called, a body is expected. This body is then made into an instance of a Review-object which is modeled for a mongoose schema. The model looks like this

```
module.exports = (mongoose) => {
  var schema = mongoose.Schema(
    {
      idUser: String,
      review: String,
      title: String,
      rating: Number,
      ratingReasoning: String,
      platform: String,
      image: String,
      featured: Number,
    },
    { timestamps: true }
  );

  schema.method("toJSON", function () {
    const { __v, _id, ...object } = this.toObject();
    object.id = _id;
    return object;
  });

  const Review = mongoose.model("review", schema);
  return Review;
};
```

:

We can then call the mongoose method "*save*" on this object in order to save the review which has been POST'ed to our end-point.

```
exports.create = (req, res) => {
  const review = new Review({
    idUser: req.body.idUser,
    review: req.body.review,
    title: req.body.title,
    rating: req.body.rating,
    ratingReasoning: req.body.ratingReasoning,
    platform: req.body.platform,
    image: req.body.image,
    featured: req.body.featured,
  });

  review
    .save(review)
    .then((data) => {
      res.send(data);
    })
    .catch((err) => {
      res.status(500).send({
        message:
          err.message || "Some error occurred while creating the Review.",
      });
    });
};
```

If no error occurs, the inserted data is then returned to where the POST-request was sent from as an indication that the review has been successfully POST'ed.

5 Conclusion

The MySQL database is relational, meaning that the data in one table can contain data that relates to data in another table. In our case we have a user table, and that user table is related to the review table which contains which user wrote the review. But sometimes you need to use complex joins between tables which can be a hard task and can require more to perform than other database formats.

The graph database does not require joins and runs faster. Graph databases are usually easier to navigate when the datasets get bigger and more complex. In neo4j the data visualization is also a bit easier to understand than looking at a lot of tables and rows of data. When we chose to go for SQL we did that because we had an idea that it was a better fit for our application. Meaning that we think the relations between user, review and comments was a better fit when using sql. Through SQL we could also ensure that our data was correct. It was also easier for us, since the application is not that big to use foreign keys and joins between tables. If the application had been bigger, maybe another option would have been smarter.

The document database is a little less ensuring of the data, and is better at larger scales. There is also less structure meaning that it is a more flexible way of storing data. The data is stored in JSON documents which are objects in our code. The documents get stored into a collection which is the equivalent of a Node in graph database or table in SQL. But the lack of structure in the document database and the fact that our application does not need to scale at a large level. We chose the relational database as our choice, that does not mean that it is not possible to create our backend with the other database choices we were provided.

In conclusion it is our feeling that the difference between using MySQL and MongoDB is less than between the two and Neo4j. All three types of databases have their advantages and disadvantages, even though on a performance level you can argue that MySQL has a lot of disadvantages.

Setting up a Neo4j or MongoDB environment takes little to no time, as you do not have to define your schemas as thoroughly as it is required for a MySQL database.

The query-languages are quite similar with the largest difference being with Neo4j yet it is quite intuitive.

6 References

https://neo4j.com/

https://graphql.org/

https://www.mongodb.com/

https://sequelize.org/

https://mongoosejs.com/

https://www.mysql.com/

https://nodejs.org/en/

Installation procedure

1) MySQL

To get started with our database, open up MySQL Workbench and establish a connection. Click “File” and then “Open SQL Script”, select our .sql file “Mandatory 2”. Run this script and our database will be set up.

2) Neo4j

In order to create your own database from our Neo4j-dump-file, you have to open up your Neo4j Desktop application. Once it is open, you create a new project, then click “Add” and select “File” and choose our dump-file. Once it has been added to your project, click on the three dots to the most right of your filename and click “Create new DBMS from dump”. Assign your new DBMS a name and a password and click “Create”.

3) MongoDB

As we have our MongoDB hosted on MongoDBs service “MongoDB Cloud” it is available for use everywhere without an actual dump-file.
The connection string for the database is:
mongodb+srv://username:Password@cluster0.ficlyk4.mongodb.net/?retryWrites=true&w=majority

We’ve made the username “username”, and password “Password”. So the exact string above works.

In order to start the backend, open the folder for the desired project in VSCode or any editor of your choice. While in that folder in any cmd/terminal write “*npm i*” to install dependencies. Once done, write “*node server.js*” to start the backend. Now it is possible to make requests to any end-point of your choice.