Final project
Databases for Developers

The final project assignment is to develop the backend part of a web application and deploy the whole solution to the cloud. You need to implement 3 solutions with following databases:

- relational database (MySQL is recommended),
- document database (MongoDB is recommended),
- graph database (Neo4j is recommended).

Recommended database cloud services: Azure MySQL server, MongoDB Atlas, Neo4j AuraDB.

Examples of the possible projects – you can come up with your own idea

- RPG – role-playing game
- Stock brokerage web app
- Logistics & Package Tracking System
- Hospital & Patient Record System
- Hotel booking & room allocation system
- Payroll & employee management system
- Restaurant ordering & kitchen workflow system
- Warehouse & inventory optimization system
- Ride-sharing & trip tracking system
- Social media content & moderation backend
- Chess platform
- Real-estate listing & rental system
- Food delivery & courier dispatch system
- Manufacturing production tracking system
- Banking system
- Movie rental / streaming system
- Car rental system
- Art collection system
- System for reviews of concerts and music releases
- Web shop
- Airline reservation system
- Library management system
- …

Databases should be complex enough to cover the curriculum of the whole course.

Domain model complexity

The relational database should have at least 10 main entities (join tables do not count).

The other 2 solutions must be able to store the same data and query the same information – of course, the design and structure will be different.

We are working with a “monolithic” design. We deliberately avoid more complex architectures such as microservices. Instead, we want to focus on dealing with a complex databases from the software developer perspective.

Substituting the frontend with Postman or Swagger UI:

The focus will be on the database and the backend server. Use Postman, Swagger UI, etc. to simulate the interaction with the API. If you happen to have a frontend, it is fine, but do not cover it in the report because it is not relevant for this course.

API documentation

Most frameworks allow you to use Swagger to create a documentation for your APIs, so you can have a swagger with 3 sets of endpoints. This will give an overview of the available database resources and actions – one set for each database:

Example:

- …/mysql/products
- …/mongodb/products
- …/neo4j/products

Project will consist of the following:

1. **CRUD Application (Backend)**
2. **Relational Database**
3. **Document Database**
4. **Graph Database**
5. **Migrator Application**

You are free to choose whichever tools you consider appropriate. The recommendations are MySQL as a relational database, MongoDB as a document database, and Neo4j as a graph database, because we will use them during the lectures. However, you may choose any other databases of the same type if all the requirements are fulfilled.

For the CRUD application, you are free to choose any development stack – for example JavaScript or TypeScript for Node.js / Express, Java / Spring Boot, C# / ASP .NET, Python / Django / Fast API, PHP / Laravel, …

NOTE: You must develop your own backend. You are not allowed to use firebase, supabase, or similar platforms as you API. You can use them as a database service but not as an API.

Mandatory project deliverables:

- Backend CRUD application
- Relational database solution
- Document database solution
- Graph database solution
- Migrator application (one-time migration from RDBMS)
- Development environment using docker-compose
- Cloud deployment using managed database services
- Integration tests (application ↔ database)
- AI-based data enrichment feature

Test data

At the delivery time, the databases should contain some meaningful amount of realistic data. Aim for having at least 100 records for each entity – like 100 rows in each table, 100 documents in each collection, etc. It should be possible to generate the test data with the help of AI tools like ChatGPT.
NOTE: You only need to generate the test data for your relational database as that is your starting point. The other databases will be seeded by the migrator application.

AI Integration (Data Enrichment)

The system must integrate with an external AI service to generate derived data based on information stored in the database. To avoid costs, a local model can be used in the development environment. The deployed production version may have the AI feature disabled.

The AI feature must:

- Process and aggregate data queried from the database (e.g. reviews, descriptions, metadata)
- Persist the AI-generated result back into the database
- AI-generated data must be stored and treated as part of the domain model and must not be generated only transiently in memory.

Database user privileges

Users will be defined at database level. There will be, at least:

- ➢ A user for the application (with the minimum privileges it needs)
- ➢ A user with full database admin privileges
- ➢ A user with read-only privileges
- ➢ A user with restricted reading privileges, which will be unable to see some data

NOTE: We need to define a user for the CRUD application itself for connecting to the database server. In production, the application should not have admin rights, but it should only have the minimum rights it needs to perform its functionality.

MySQL solution (or another relational database)

A CRUD application (create/read/update/delete) functionality to implement can be:

- Login/logout.
- Query data from the tables.
- CRUD Functionality depending on the business logic
- Typical API features like pagination, filtering, and sorting. You should not have any GET endpoints that query unlimited amount of data.
- Security measures to prevent typical attacks and non-authorized access to data.

MongoDB solution (or another document database)

The goal is to implement the same functionality (or at least most of it) like with the relational database.

You add the document database as a parallel data source to the existing backend (and create parallel REST APIs or GraphQL API).

To move from the relational model to document design, we usually need to de-normalize our relational model. For example, if we have tables like customers, orders, order_items, products we can have all this information in a customer document which will contain multiple levels of embedded documents – a customer document will contain an array of order documents. Each order can contain multiple order_item documents…

Neo4j solution (or another graph database)

The goal is to implement the same functionality (or at least most of it) as with the relational and document database.
You add the graph database as another data source to the existing backend (and create parallel REST APIs or GraphQL API).

A graph database consists of nodes and relationships which makes it easy to traverse between nodes – for example if we want to know who a friend of a friend of a friend is… (social media app) or if we want to know what other products were purchased by customers who ordered or looked at a particular product (product recommendation).

Migrator application

Application responsible for one-time migration from the relational database to MongoDB and Neo4j.
It will simulate a situation where the company decides to migrate the production database with existing data into another database technology. That is why it will be aiming for one-time migration once your relational database is finished and seeded with test data that will represent the production data.

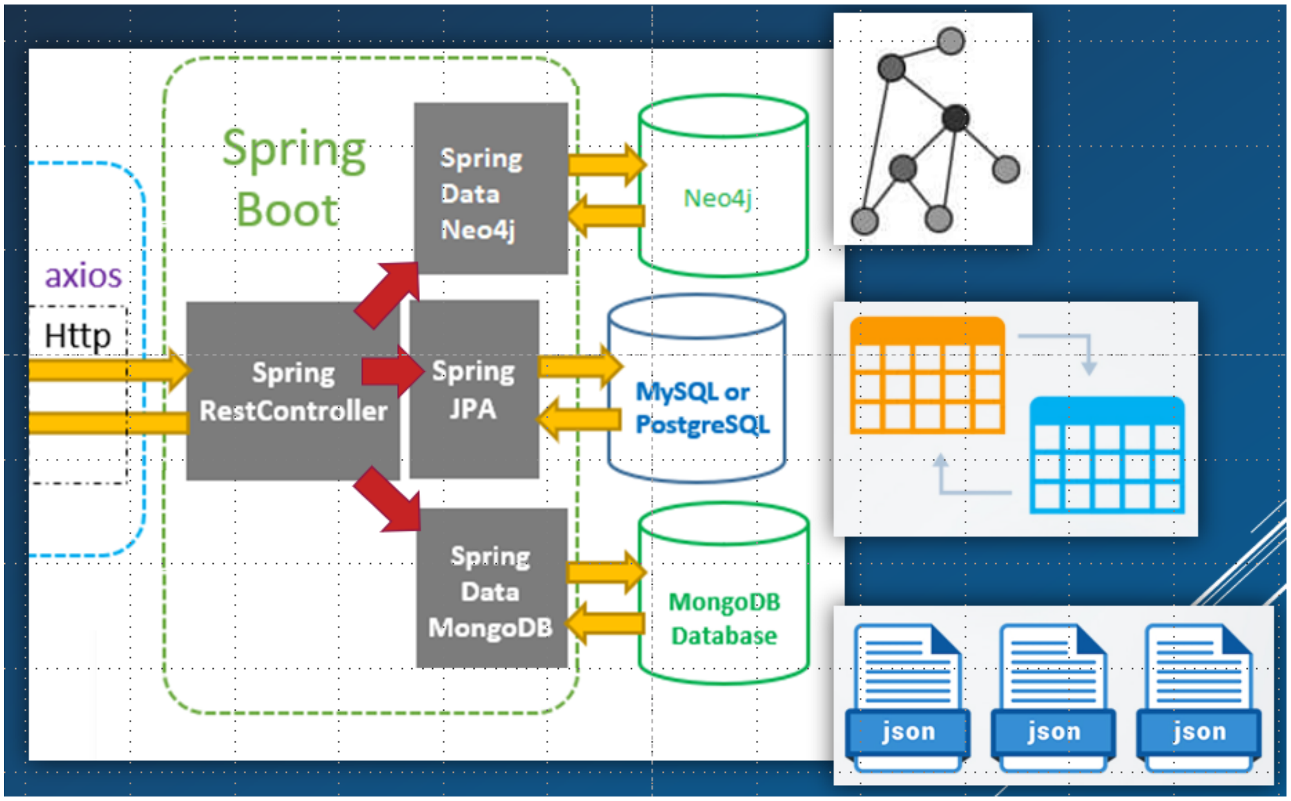
Summary

- We start with creating a system which uses an RDBMS like MySQL.
- Then we implement the same functionality with MongoDB (or similar DB).
- Then we implement the same functionality with Neo4j (or similar DB).
- Implement a migrator application for one-time migration from RDBMS.

Implementing the same functionality with different database technologies simulates a scenario when a company decides to switch from one database technology to another which we will achieve using our migrator application.

It will also help to compare the strengths and weaknesses of used database technologies.

Example of all 3 solutions in one Spring Boot project (service layer is not visible)



Apply layered architecture: controllers -> services -> models / repositories. If your backend supports generics, you can create a single set of controllers and services working with generic types and switch between the databases using a variable. Otherwise, you can make 3 sets of modules. Also, use DTO objects in the controllers. Do not expose the models to the outside.

Final Project Delivery

The final group project will be uploaded individually to WISEflow. It will consist of a report and a series of artifacts.

Final Project Artifacts – public code repository

- Relational database scripts (one or several), including:
    - Database creation, including tables, keys, indexes, constraints, and referential integrity checks.
    - Load of test data
    - Stored procedures
    - Triggers
    - Views
    - Events
    - Creation of users and privileges
- The source code of the CRUD application and the migrator - included as a link to an external public code repository (like GitHub).
- For MongoDB system:
    - dump file of the document database
    - Script for loading the test data
    - The source code of the CRUD application.
- For Neo4j system:
    - dump file of the graph database
    - Script for loading the test data
    - The source code of the CRUD application.
- The source code of the migration application (link to GitHub or another public repo).
- A brief installation procedure that specifies how to organize the code and import the databases in a test environment with full operational capabilities. For this, it is recommended to use a containerized solution with docker-compose.

Final Project Report

The final report will have a big influence on the exam grade. It is the only information that is accessible to the external censor prior to the oral exam.

The report should have this structure:

Cover page, including Title, Full names of all students in the group, Group number, Date of delivery
List of figures
List of appendices
Table of contents (paginated index – the whole document should be paginated)
1. Introduction
1.1. System overview and cloud architecture: This section must include:
- Architecture diagram showing the system deployed in the cloud, and
- Diagram showing the local development deployment using docker-compose.

1.2. Explanation of choices for databases and programming languages, and other tools.
2. **Relational database**
2.1. Intro to relational databases

2.2. Database design
2.2.1. Entity/Relationship Model (Conceptual -> Logical -> Physical model)
2.2.2. Normalization process
2.3. Physical data model
2.3.1. Data types
2.3.2. Primary and foreign keys
2.3.3. Indexes
2.3.4. Constraints and referential integrity
2.4. Stored objects – stored procedures / functions, views, triggers, events
2.5. Transactions. Explanation of the structure and implementation of transactions
2.6. Auditing. Explanation of the audit structure implemented with triggers
2.7. Security.
2.7.1. Explanation of users and privileges
2.7.2. SQL Injection – what is it and how it is dealt with in the project?
2.8. Description of the CRUD application for RDBMS

- Mainly the data layer – models, repositories, etc. but also briefly mention the other layers like services and controllers.
- Graphical schema of the backend modules (model, repository, service, controller, etc).
- Other relevant topics like transactions, calling the stored procedures, security, integration testing, AI integration

3. **Document database**
3.1. Intro to document databases
3.2. Database design – also in a graphical form showing all the collections and embeddings.
3.3. Short descriptions of features like indexes, transactions, PKs, constraints, stored objects (if stored objects are not available, how were they replaced?)
3.4. Description of the CRUD application for the document database

- same as for the RDBMS – if you have identical solution where the only difference is the data layer, you can just focus on the parts that are different.

4. Graph database
4.1. Intro to graph databases
4.2. Database design - also in a graphical form (show the database model – not a screenshot of the data)
4.3. Short descriptions of features like indexes, transactions, PKs, constraints, stored objects (if stored objects are not available, how were they replaced?)
4.4. Description of the CRUD application for the graph database

- same as for the RDBMS – if you have identical solution where the only difference is the data layer, you can just focus on the parts that are different.

5. Discussion: similarities and differences between used database types, how each handled: Data modeling, Transactions, Queries, Performance, Constraints & integrity, Which database fits the domain best — and why, Trade-offs and compromises
6. Reflection: What was difficult about: Modeling the same domain in three databases, implementing transactions, auditing, and security, What design mistakes were made and fixed, what would be done differently next time, what you learned about databases in practice.
7. Conclusion: short high-level summary of the project, results, main insights.

8. References: All major design choices, technical claims, and comparisons must be supported by references. Sources must be cited in the text and listed in the References section. References should primarily be official documentation, textbooks, or academic or technical publications. Use standard formats like APA, IEEE, or Harvard.
9. Appendix - optional

Report formal requirements

You should work in groups of 4 - 5 students.
The formal requirements for the report can be found in the course catalog, in the exam section:
https://katalog.kea.dk/course/preview/6584/da-DK