CIT/P Portfolio Project Requirements

Complex IT Systems – Practice

September 18, 2025

This document describes the requirements to the three portfolio subprojects that are included in the mandatory Project Portfolio.

The overall goal of these portfolio subprojects is to build a system that allows multiple users to view information about, browse, search for, rate, and compare movies and actors. The system must keep track of users' search history and must support rating and bookmarking of movies.

Each portfolio subproject corresponds to one of the topics covered in the three main sections of the CIT/T course. The three subprojects will aim at, respectively, (1) building a data repository with embedded functionality for unified access, (2) developing services to access the repository and exposing these services through a well-defined interface, and (3) developing a web-based client that gives uses access to the provided services via a browser.

The database developed in the first subproject is based on data from the Internet Movie Database (IMDB). IMDB is an online database of information mainly related to movies and TV Series (but includes also information about other types of published material such as games and documentaries). It is the largest and most comprehensive movie database on the web and it includes more than 24 million titles (movies, TV-series and other) and more than 14.5 million personalities. IMDB was launched in 1990 and is now owned by Amazon.com. We will use a reduced version of IMDB's dataset. (See the *CITP Project Portfolio Source Data* found on Moodle for a description of the provided data.)

The solution to each subproject consists of models, scripts and programs that implement part of the system and a report that documents this. The results of subprojects 1 and 2 must be handed in as stand-alone project reports on the course's Moodle page. (The deadlines and instructions for handing in are described on Moodle.) The result of subproject 3 must be handed in via `eksamen.ruc.dk`. (The deadline is found on `study.ruc.dk` under the description of Practice".) This final report must also include one so-called *individual reflection* for each member of the group. (See 3-G.1 on page 21.) You should not include the answers to one subproject in the reports for subsequent subprojects. If you find the need to modify a previous subproject during the development of the current subproject, then explain those modifications in the report covering the current subproject.

Each of the three reports must document the parts of the system you implement. These

reports must follow standard principles of academic and technical writing. Document the (relevant) parts of the system (procedures, functions, methods, classes, components, tables, etc.) by analyzing their requirement and the context in which they are used, designing a solution, implementing the proposed design, evaluate the merits of the implementation, and reflecting on the result (thus following a standard approach to software engineering).

A report should not be a log documenting what you did when. Present the final product, instead of all the steps taken to develop it. A report should not just list the artifacts (procedures, functions, methods, classes, components, tables, etc.) that implement the system. Instead supplement the descriptions of these artifacts by motivating, explaining, and justifying the decisions that resulted in them. Avoid explaining solely by giving examples. Instead give general explanations and then (if needed) supplementary examples. The reports should be precise, consistent, concise, systematic, and readable. The reports must be formatted consistently. All sections must be numbered. All pages must be numbered. The report must contain a table of contents.

This document contains 21 pages. It consists three chapters documenting the requirements to the three subprojects. We recommend that you read or skim the set of requirements for all three portfolio subprojects before getting started on the first.

Contents

Subproject 1

The database

The objective of Portfolio Subproject 1 is to deliver a database for the movie application and to set up key application functionality. The database should be designed around two partially independent data models: a Movie Data Model for storing movie information gathered from external sources, and a Framework Model for handling the application framework (users, bookmarks, local ratings, history). These two models must be integrated into a single database during implementation.

1-A Application design

Sketch a preliminary design of the application you intend to develop and the features that you aim to provide. Study the domain by considering the provided data and by browsing movie apps on the web. Based on this, develop your own ideas and describe these in brief. Develop and describe also a first sketch of how to show the history, ratings and bookmarkings in your application. Give arguments for your design decisions and discuss and describe the implications on your data model. Since this is a preliminary design, it may obviously be subject to later changes — including later simplifications due to time limitations.

1-B The Movie Data Model

The data model for the movie data part must be designed so that the provided data can be represented. The provided data is described in the note *CITP Project Portfolio Source Data* that also explains how to get access to the data and load the data into a database in your own database server.

You can claim that the provided tables already comprise a movie data model as requested, but if you want to achieve a good design that doesn't violate common conventions concerning relational database design, a redesign is needed.

1-B.1 Create a data model to represent the provided data. Aim to design the model

independently of the original source structure, while ensuring that all information can still be represented. Use the database design methods and theory covered in the database section of the CITT course as guidance. Be sure to document both intermediate and final versions of your models, consider alternative designs, and justify the decisions you make. *Present an ER diagram* of your final model *using the notation specified in the DB-book*.

1-B.2 Implement the model developed in 1-B.1 as a relational database in PostgreSQL. Start by creating a new database and importing all source data as outlined in *CITP Project Portfolio Source Data*. Then, define the tables that implement your 1-B.1 model and transfer the data from the imported source tables into these new structures. After the data has been successfully migrated, remove the original source tables. (Notice that the values of `nconst` are needed by the frontend; these should be available in the data model. See 3-D.2.)

Collect all your commands for creating tables, distributing data and deleting source data in a single SQL script called `B2_build_movie_db.sql`. Make this script such that it can be modified and executed repeatedly with `psql` until you are satisfied with the result and have tested that all data is in the right place. To make a script so that it can be executed repeatedly, simply include "wipe-out" of all the old stuff — either by `dropping` things to start with or by using `create or replace` rather than just `create`. The major part of your SQL script will be `create table` and `insert` statements. (Maybe like this: `insert into xxx select ... from ...`). However, you may also need to include `do`-blocks in your SQL script, for instance if you need to iterate through a source table and split data. When you are done, generate a *reverse engineered ER diagram*[1] and include this as well as your SQL script in your documentation.

If your considerations from Section 1-A calls for changes to the Movie-data model, you can just include these changes in 1-B.1 and 1-B.2. Just remember to add a note about what the changes are.

1-C Framework Model

In essence the framework model should support the following: registration of users of your application and, for users individually, storage of search history, storage of rating history, bookmarking of titles and names (personalities). Feel free to extend the Framework with your own additions (such as support addition of personal notes to titles).

1-C.1 Develop a data model that is appropriate for the purpose of the framework. Provide (and include in your report) an ER diagram of your framework model. Indicate how your Framework Model connects to the Movie Data Model by

[1] Open your database in Navicat and right click on the schema (probably called "public"). Then select "Reverse Schema to Model ..." and click the Diagram tab (or double click the diagram line). You can edit the diagram by moving boxes and connectors, if you would like to change the layout. When you are done, you can save it or just copy the graphics by taking a screenshot and paste this into your report.

including the entities from your Movie Data Model in 1-B.1 that are directly connected with relationships from the Framework Model. It must be clear how the two models combine. Optionally, provide a complete ER diagram with all entities and relationships from both models. Use the ER notation described in the DB-book.

1-C.2 Implement the Framework Model. Write an SQL script that adds the Framework Model to an already created Movie Data Model database (the result from 1-B.2). Name this script `C2_build_framework_db.sql`. Since there is no data for the Framework part yet, the script will be mostly `create table` statements. Generate a reverse engineered ER-diagram of your full (Movie Data + Framework) database and include this as well as your SQL script in your documentation.

1-D Functionality

In addition to modeling and implementing the database, an essential part of this subproject is to develop core functionality that the data layer can expose for use by the service layer. The objective is to provide this functionality through an Application Programming Interface (API), consisting of a set of functions and procedures implemented in PostgreSQL.

The initial requirements focus on three areas: framework support, a basic search function, and title rating functionality.

1-D.1 Basic framework functionality: Consider what is needed to support the framework and develop functions for that. You will need functions for managing users and for bookmarking names and titles. You could also consider developing functions for adding notes to titles and names and for retrieving bookmarks as well as search history and rating history for users.

1-D.2 Simple search: Develop a simple search function called `string_search()`. This function should, given a search string S as parameter, find all movies where S is a substring of the title or a substring of the plot description. For the movies found, return id and title (`tconst` and `primarytitle`, if you kept the attribute names from the provided dataset). Make sure to bring the framework into play, such that the search history is updated as a side effect of the call of the search function.

1-D.3 Title rating: Introduce functionality for rating by a function, called `rate()`, that takes a title and a rate as an integer value between 1 and 10, where 10 is best.[2] The function should update the (average) rating appropriately taking the new vote into consideration. Make sure to bring the framework into play, such that the rating history is updated as a side effect of the call of the rate function. Also make sure to treat multiple calls of `rate()` by the same user

[2] See the detailed interpretation of IMDB ratings at help.imdb.com/article/imdb/track-movies-tv/ratings-faq/G67Y87TFYYP6TWAV.

consistently. This could be for instance by ignoring or blocking an attempt to rate, in case a rating of the same movie by the same user is already registered. Alternatively, an update with the new rate can be preceded by a "redrawing" of the previous rating, recalculating the average rating appropriately.

The following section extends the concept of search beyond basic string matching. Update the database and implement a set of functions and procedures that fulfill the specified requirements. When doing so, consider and discuss possible alternatives, justify your design decisions, and, where relevant, refer to applicable methodology and theory. Ensure to bring the framework into play where appropriate.

1-D.4 Structured string search: Create a search function that accepts four string parameters and name the function `structured_string_search()`. The function should:

- Return movies where the four provided parameters match the four fields: title, plot, characters, and person names, respectively.
- Matching should be case-insensitive.
- Parameter values should be treated as substrings, meaning a match occurs if the value is contained anywhere in the corresponding column.

For each movie found, the function should return its ID (`tconst` in the source data) and title (`primarytitle`). Make sure to bring the framework into play, such that the search history is stored as a side effect of the call of the search function.

1-D.5 Finding names: The above search functions are focused on finding titles. Try to add to these by developing one or two functions aimed at finding names (of, for instance, actors).

1-D.6 Finding co-players: Make a function that, given the name of an actor, will return a list of actors that are the most frequent co-players to the given actor. For the actors found, return their `nconst`, `primaryname` and the frequency (number of titles in which they have co-played).

Hint: You may, for this as well as for other purposes, find a view helpful to make query expressions easier (to express and to read). An example of such a view could be one that collects the most important columns from title, principals and name in a single virtual table.

1-D.7 Name rating: Derive a rating of names (just actors or all names, as you prefer) based on ratings of the titles they are related to. Modify the database to store also these name ratings. Make sure to give higher influence to titles with more votes in the calculation. You can do this by calculating a weighted average of the `averagerating` for the titles, where the `numvotes` is used as weight.

1-D.8 Popular actors: Propose and implement a function that utilizes the name ratings. One option is a function that takes a movie as input and returns the list of

actors in that movie, ordered by decreasing popularity. Another could be a similar function that takes an actor and lists the co-players in order of decreasing popularity.

1-D.9 Similar movies: Discuss and suggest a notion of similarity among movies. Design and implement a function that, given a movie as input, will provide a list of other movies that are similar.

1-D.10 Frequent person words: The `wi` table provides an inverted index for titles using the four columns: `primarytitle`, `plot` and, from persons involved in the title, `characters` and `primaryname`. So, given a title, we can from `wi` get a lot of words, that are somehow characteristic for the title. To retrieve a list of words that are characteristic for a person we can do the following: find the titles the person has been involved in, and find all words associated with these titles (using `wi`). To get a list of unique words, you can just group by word in an aggregation by `count()`. Thereby you'll get a list of words together with their frequencies in all titles the person has been involved in. Use this principle in a function `person_words()` that takes a person name as parameter and returns a list of words in decreasing frequency order, limited to fixed length (e.g. 10). Optionally, add a parameter to the function to set a maximum for the length of the list. You can consider the frequency to be a weight, where higher weight means more importance in the characteristics of the person.

1-D.11 Exact-match querying: Introduce an exact-match querying function that takes one or more keywords as arguments and returns titles that match all of these. Use the inverted index `wi` for this purpose. You can find inspiration on how to do that in the slides on Textual Data and IR.

1-D.12 Best-match querying: Develop a refined function similar to 1-D.11, but now with a "best-match" ranking and ordering of objects in the answer. A best-match ranking simply means: the more keywords that match, the higher the rank. Titles in the answer should be ordered by decreasing rank. See also the Textual Data and IR slides for hints.

1-D.13 Word-to-words querying: An alternative, to providing search results as ranked lists of posts, is to provide answers in the form of ranked lists of words. These would then be *weighted keyword lists* with weights indicating relevance to the query. Develop functionality to provide such lists as answer. One option to do this is the following: (1) Evaluate the keyword query and derive the set of all matching titles, (2) count word frequencies over all matching titles (for all matching titles collect the words they are indexed by in the inverted index `wi`), and (3) provide the most frequent words (in decreasing order) as an answer (the frequency is thus the weight here).

Consider, if time allows, the following issues.

1-D.14 Weighted indexing [Optional]: Build a new inverted index for weighted indexing similar to the `wi` index, but now with added weights. A weight for an entry in the index should indicate the relevance of the title to the word.

As weighting strategy, a good choice would probably be a variant of TFIDF. Ranked weighted querying: Develop a refined function similar to 1-D.12, but now with a ranking based on a relevance weighting (TFIDF or similar) provided by the weighted indexing.

Finally, feel free to elaborate.

1-D.15 **Own ideas [Optional]:** If you have some ideas of your own, you can plug them in here.

1-E Improving performance by indexing

A key advantage of using a relational database is that query performance can be dynamically optimized at any stage of development and tailored to actual system usage, particularly for the most frequent or critical queries — even after deployment. Such performance improvements can be made by adding or adjusting indexes on database tables, without requiring changes to other parts of the system.

1-E.1 Consider the extensions developed under Section 1-D and discuss or explain what may potentially provide significant performance improvements. Describe how your database is indexed. (Observe that this concerns database indexing, not textual inverted indexing — even though the latter may build on the former.)

1-F Testing using the IMDB database

Demonstrate by examples that the results of Section 1-D work as intended. Write a single SQL script that activates all the written functions and procedures and, for those that modify data, add selections to show before and after for the modifications. In this subproject you need only to proof by examples that you code is runnable. A more elaborate approach to testing is an issue in Subproject 2. Generate an output file from running your test script file. (See descriptions in assignment 1 and 2 on how to do this.)

Subproject 2

The backend

The objective of Subproject 2 within this portfolio is to enhance the Movie application by incorporating a RESTful web service interface and expanding its functionality. Our aim is to design an architecture that prioritizes maintainability, testability, extensibility, and scalability. We intend to follow best practices, even though the current complexity of the model and application may not warrant a detailed justification. It is our anticipation that future enhancements and additional features will be necessary for the system.

The diagram in Figure 2-1 illustrates the overall architecture of the portion of the application to be worked on in this subproject.

The primary focus for Subproject 3 will be the clients located at the top in Figure 2-1. The presence of the blue client on the right merely serves as an indication that additional components may require access to the system in the future. The red boxes positioned between the various layers depict the type of data exchanged among the components or layers. Specifically, Data Transfer Objects (DTOs) will be employed for communication between the Business Layer and the web service layer. This approach is adopted to establish a decoupled architecture, ensuring that modifications in the lower layers do not necessitate corresponding alterations in the upper layers of the system.

Data Access Layer

The Data Access Layer[1] (DAL) serves as the gateway to the database, conceals the intricacies of SQL, manages transactions, and facilitates the conversion between the relational model and the object-oriented model, as well as the reverse transformation when necessary.

The key design patterns employed within the data access layer include the Data Map-

[1] See en.wikipedia.org/wiki/Data_access_layer.

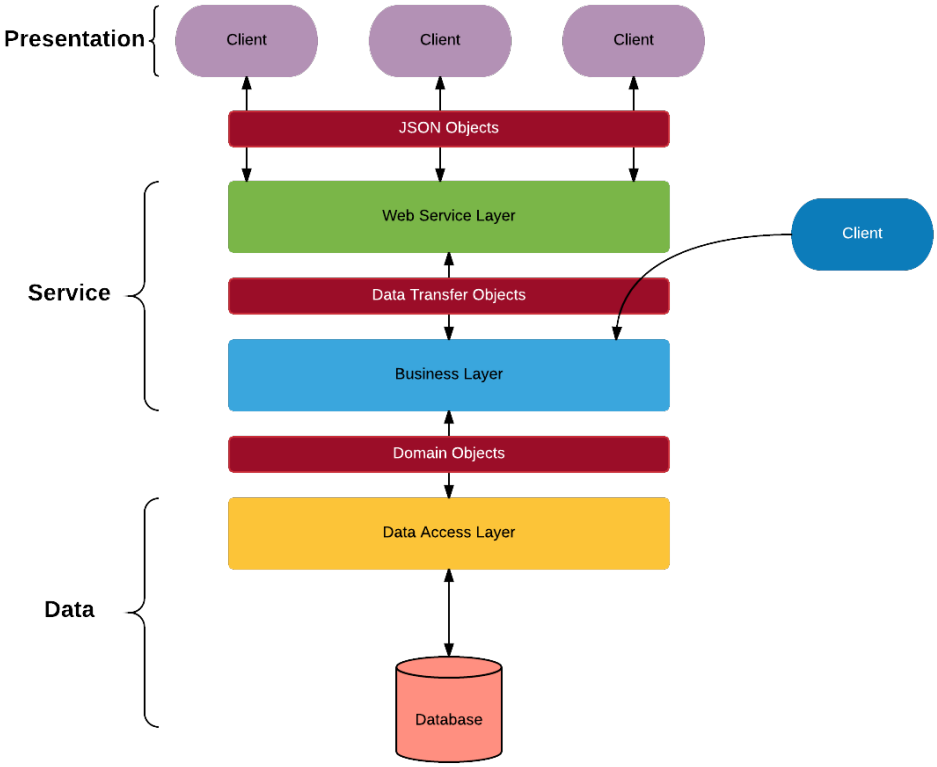

Figure 2-1: Architecture

per[2], Repository[3], and Unit of Work[4] patterns, aimed at establishing a robust and adherent layer following good principles, like Single-responsibility[5] and Dependency inversion[6], on top of the database. Additionally, several other patterns, such as Façade[7], Singleton[8], Factory[9], and Bridge[10], are commonly utilized to bolster sound architectural designs within this layer.

Leveraging the Entity Framework simplifies the implementation of data mapping, unit of work, and the repository pattern to a certain extent. However, the provided repository pattern by Entity Framework is quite general, offering an interface to all collections within the data context. In our specific development context, it becomes evident that a more tailored and fine-grained interface is required to effectively support the unique requirements of the system under development.

[2] See martinfowler.com/eaaCatalog/dataMapper.html.
[3] See martinfowler.com/eaaCatalog/repository.html.
[4] See martinfowler.com/eaaCatalog/unitOfWork.html.
[5] See en.wikipedia.org/wiki/Single-responsibility_principle.
[6] See en.wikipedia.org/wiki/Dependency_inversion_principle.
[7] See en.wikipedia.org/wiki/Facade_pattern.
[8] See en.wikipedia.org/wiki/Singleton_pattern.
[9] See en.wikipedia.org/wiki/Factory_(object-oriented_programming).
[10] See en.wikipedia.org/wiki/Bridge_pattern.

Business Layer

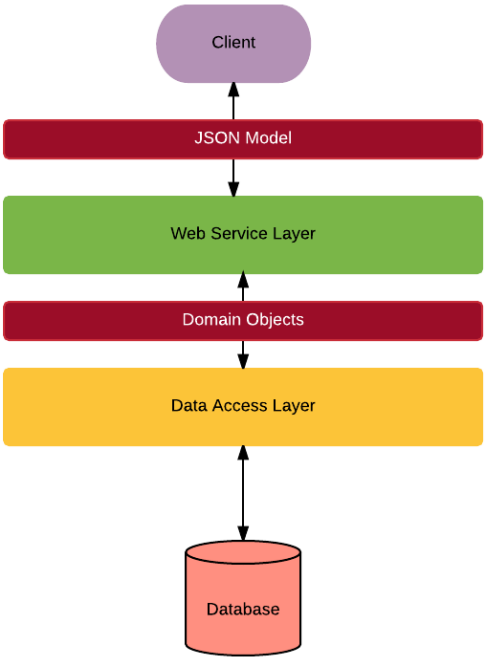

Figure 2-2: Simplified architecture

The heart of the application resides in the Business Layer (BL), which orchestrates the business logic and oversees system behaviors. This layer is responsible for encapsulating all the system logic, excluding the aspects that are specifically delegated to the database. In simpler applications, the Business Layer may amalgamate with other layers, such as the Data Access Layer, effectively serving as a bridge to the database. However, in more intricate applications, it abstracts the database and operates on an autonomous domain model.

For this project, it appears likely that the Business Layer can be seamlessly integrated with the other layers, resulting in a merger of the Business Layer with the Data Access and Web Service layers. Consequently, the architecture will assume a configuration akin to the diagram depicted in Figure 2-2.

Web Service Layer

The Web Service Layer (WSL) is required to offer a consistent and standardized interface for the resources it makes accessible. In other words, if a client possesses the knowledge of accessing one resource, it should be able to apply the same approach to access other resources as well.

As an illustration, suppose the URI for accessing the movie with ID 123 is `/api/movies/123`. In a similar manner, the same pattern can be applied to retrieve other entities, such as the actor with ID 234 using `/api/actors/234`. The Web Service Layer is respon-

sible for delivering responses in the form of objects, adhering to either JSON or XML format, as determined by client-specified content negotiation. While this project primarily requires us to offer responses in JSON format, feel free to extend support to other formats like XML if desired.

The Web Service Layer plays a pivotal role in establishing the link between the URI interface and the controllers that possess the knowledge to access the underlying layers for data retrieval or modification.

2-A Application design

Revising the initial application design presented in Subproject 1, we aim to define the use cases[11] and/or user stories[12] that the system must accommodate. By doing this we can ensure that functional requirements that we expect the application to fulfill are captured, and we can extract a high-level overview of the required requests to the web service.

2-A.1 Outline the architecture of the backend[13] system. Provide an overview of the overall system architecture. This is basically to state how your system is structured. It may be as shown in Figure 2-2, and then you can elaborate on the specific components and their interactions as a starting point.

2-A.2 Create class diagrams to illustrate dependencies within and between layers. Present visual representations of how classes within the application's layers are interrelated. We do not aim to create a complete UML model, but rather to provide a high-level overview of the class relationships. Pick the most important classes and show how they relate to each other. You can choose to create separate diagrams for each layer or combine them into a single diagram, depending on what you find most effective for conveying the structure of your application.

2-A.3 Document the structure of the domain objects, data transfer objects, and the JSON objects (the Outer facing contract). This requirement is closely related to 2-A.2, but should concentrate on the actual objects and their structure. Again, we do not aim to create a complete UML model, but rather to provide a high-level overview of the object structures. Pick the most important objects and show how they are composed.

- Describe the composition and relationships of domain objects.
- Explain the structure and purpose of data transfer objects.
- Define the format and contents of JSON objects used for external communication (the outer facing contract).

[11] See en.wikipedia.org/wiki/Use_case.
[12] See en.wikipedia.org/wiki/User_story.
[13] Assuming that the web pages created in the next section will form the frontend, the backend is everything from the first two sections.

These revisions will help refine the understanding and documentation of the application design, ensuring it aligns with the project's requirements and goals.

2-B The Data Access Layer

In environments characterized by numerous and potentially changing data sources, it becomes imperative for the system to abstract the concrete implementations and offer a more generic interface that isn't tied to specific sources. Achieving this goal can be realized through the implementation of a data service, leveraging the Repository pattern or similar techniques, thereby establishing an abstract interface for data that can be adapted for diverse resources.

2-B.1 Define and document the domain model, with a specific emphasis on how to transform the relational model into an object-oriented model. This requirement closely relates to 2-A.3 but should concentrate on the object-relational mapping, delineating how the relational model within the database aligns with the object-oriented domain model. We do not aim to create a complete UML model, but examples of interesting mappings in your model, to get an overview of how your solve the mapping.

2-B.2 Construct a database access layer founded upon Object-Relational mapping, utilizing the Entity Framework. Develop the requisite repositories or services to craft an abstract interface for the database, facilitating CRUD operations (Create, Read, Update, and Delete) as necessary. Ultimately, the repositories or services' purpose is to define an interface that streamlines communication between the Web Service Layer or Business Layer and the Data Access Layer by furnishing the essential functionality. [Be aware that the IMDB data is read-only, so we will not support creation, update or deletion of IMDB data.]

2-B.3 Prepare the data access layer to incorporate authentication, including the addition of necessary parameters to methods for handling authentication-related processes, like register and login.

2-C Web Service Layer

The gateway to the backend system is the Web Service Layer. The objective in this layer is to create an interface that exposes the necessary functionalities required for the frontend, which will be developed in the subsequent section. Given that the precise requirements are yet to be determined, it is crucial to incorporate the insights from Section 2-A adhere to sound design principles that accommodate changes and the addition of new features.

The implementation of RESTful web services will be carried out using ASP.NET Web API. The overarching structure of this component will involve responding to client requests through the interface provided by the Business Layer or Data Access Layer. Importantly, the Web Service Layer should remain independent of the data sources,

ensuring that modifications to the data sources do not necessitate alterations in the Web Service Layer.

Adhering to the principles of the Representational State Transfer (REST) architectural style, while maintaining flexibility, is key. The desired interface should be stateless, uniform, selfdescriptive, and centered on resources. This resource-centric approach emphasizes nouns over verbs, resulting in URLs like `.../movies/123` rather than the RPC-style `getMovie(123)` to retrieve the movie with ID 123.

2-C.1 Design a web service interface, including the specification of URIs, for accessing read-only IMDB data. Envision the requirements of the user interface, drawing insights from the use cases/user stories outlined in Section 2-A. Thoroughly document the characteristics of this interface.

2-C.2 Implement the interface as defined in 2-C.1, ensuring that it provides the desired access to read-only IMDB data.

2-C.3 Develop a web service interface, specifying URIs, for accessing framework data. This interface should support a comprehensive set of CRUD operations for all resources. Again, take into consideration the anticipated needs of the user interface and reference the use cases/user stories from Section 2-A to formulate the interface's requirements. Document the details of this interface.

2-C.4 Implement the interface as defined in 2-C.3, ensuring that it encompasses the full range of CRUD operations for framework data.

2-C.5 Ensure that all responses conform to the concept of a self-descriptive interface by providing self-references, i.e., including the URI to the specific objects rather than just their bare IDs. This requirement also extends to lists of objects, where each element in the list must contain a reference to access that object.

2-C.6 Implement paging for all listing operations, incorporating a default page size while allowing clients to specify their preferred page size. In cases where the client's requested page size is too large or not defined, the default size should be applied. Additionally, the page results should include links to access the previous and next pages, if they exist, enhancing the overall user experience.

2-D Security

Certainly, applications like the Movie application must prioritize security, especially when dealing with the storage of personal information. Security can be quite complex, particularly if you aim for a high level of protection. However, in this project, the primary emphasis does not lie in achieving an exceptionally high level of security. Instead, the focus remains on comprehending, designing, and implementing a full-stack solution.

2-D.1 Nonetheless, the backend must have the capability to manage users, ensuring that all aspects of the user-related interface, such as search history, rankings,

and bookmarks, maintain a record of user actions. The database and its interface have been prepared to accommodate this, necessitating similar preparations in the Web Service Layer and Data Access Layer. In the future, the frontend will offer user login functionality, requiring us to capture and utilize this information in communication between clients and our service. As previously mentioned, RESTful web services are stateless, meaning that authentication details must be included as part of the requests since user states are not stored in the backend. There are various strategies you can adopt for user management in your project, ranging from straightforward to highly secure solutions.

i. The simplest approach is to hardcode one specific user into the backend and overlook the login process for different users. While this sets up a system prepared for "users," it doesn't employ user authentication for resource access.

ii. A step up would involve sending the username as part of the request using the HTTP Authentication header field. With this approach, you can handle basic authentication and respond to unauthorized requests, applying authentication logic within Web Service Layer and the Data Access Layer. However, this system will lack robust security, as anyone can manipulate the request to gain access.

iii. The conventional method for addressing authentication in RESTful web services is to use tokens in conjunction with HTTPS (secure HTTP). Tokens are employed to transmit signed information from the client to the server. The token encapsulates encoded and signed details about the user (such as username) and a timeout (validity period), allowing the server to verify whether the user should have access to the requested resource. One popular token system is JSON Web Tokens[14] (JWT).

    The server generates the token upon login, and the client saves and presents the token in the HTTP Authentication header field during requests. ASP.NET offers support for authentication by integrating middleware and utilizing annotations. The middleware is equipped to decode the information contained within the token and authenticate the user. Annotations, such as `[Authorize]`, can be utilized to specify which parts of the system the user is authorized to access (whether it's a method or a class). Additional guidance can be found in a tutorial.[15]

2-E Testing

Thorough testing is essential for each aspect of the implementation, including both unit-level and integration testing. When we emphasize testing every part, we do not

[14] See jwt.io/introduction/.
[15] See jasonwatmore.com/post/2021/04/30/net-5-jwt-authentication-tutorial-with-example-api.

mean conducting exhaustive tests on each individual class or method. Instead, we suggest providing illustrative examples that demonstrate the testing of different layers and their respective components. For instance, if there are multiple repositories or services, it is adequate to showcase how to test one of them, and if the services contain several similar methods, testing one of each kind is sufficient. Additionally, within the Web Service Layer, it is essential to furnish UI tests. These UI tests validate that the service consistently delivers the expected data in the correct format, ensuring that the user interface functions as intended.

Subproject 3

The frontend

The goal of Portfolio Subproject 3 is to design and implement a frontend that provides a graphical user interface to the movie application. This frontend must run in a user's browser and it must communicate with the backend that you have implemented as part of Portfolio Subproject 2.

You should structure the frontend so that it consists of decoupled components organized into well-defined parts or layers with separate responsibilities. For example, the frontend may be internally organized into a Presentation Layer responsible for how data is presented to the user, a Business Logic Layer containing any logic independent of the presentation layer and data access, and a Data Access Layer implementing the communication with the backend web service.

Besides communicating with your own backend, the frontend must fetch pictures of persons from The Movie Database (TMDB), by making requests similar to those of Assignment 6. (See 3-D.2.)

3-A User interface design

You must design and document the user interface and the information architecture of the frontend.

You should design the user interface such that it complies with Jacob Nielsen's 10 Usability Heuristics for User Interface Design.[1]

3-A.1 Design the intended site structure of the frontend. Document this design with a site structure diagram that shows the different "pages" or "views" that the frontend provides and how the user can transition and navigate between them.[2]

[1] See www.nngroup.com/articles/ten-usability-heuristics/.
[2] See www.bluehost.com/resources/website-structuring/.

3-A.2 Design the intended user interfaces of the individual “pages” and components of the frontend. Document them using wireframe models.[3]

3-A.3 Design and document the routes that can be used to navigate the application.

3-B Presentation Layer

The actual graphical user interface must be implemented in JavaScript using the React library and Bootstrap (but see 3-F.1). It should implement the design from Section 3-A.

3-B.1 Document React components that you have developed and their roles; document the component hierarchy.

3-B.2 Describe React patterns, principles, and techniques that you have used (e.g., prop drilling, lifted state, `useState`, `useEffect`, `useContext`, and any other hooks applied, DOM router, controlled components, and how communication with backend is implemented).

3-B.3 Describe the purpose of third-party JavaScript, React, or Bootstrap libraries or components that your application uses.

3-B.4 Your solution must implement the user interface with (React) Bootstrap. Describe how (and if) Bootstrap helps you implement the user-interface design from Section 3-A.

3-C Business Logic Layer

You may chose to represent data in the frontend differently from the representation in the backend. Such an internal representation may, for example, reduce coupling to the backend and may ease combining data from different sources or servers. (See Section 3-D.)

3-C.1 Describe the internal representation of data as objects and describe any classes or functions that you have implemented to help maintain these representations.

3-C.2 Describe other helper functions for manipulating data in the frontend.

3-D Data Access Layer

A data access layer may provide an interface to the backend: Instead of connecting to the web service offered by the backend directly from the presentation layer (i.e., React components), you may implement a *data service* that defines an internal interface (for example, a set of functions or one or more classes) that the business logic layer can use to communicate (indirectly) with the backend.

This architecture prevents future changes in the backend from propagating into the business logic layer and the presentation layer of the frontend.

[3] See `en.m.wikipedia.org/wiki/Website_wireframe`.

The data service should communicate asynchronously with the backend.

3-D.1 Show where and how data access is implemented and document the functionality it provides.

3-D.2 The frontend must fetch pictures of persons from The Movie Database (TMDB), following the approach outlined below. (These pictures should, of course, be used where appropriate by the Presentation Layer.)

The attribute `nconst` in table `name_basics` uniquely identifies a person. We can use this IMDB-specific ID to find the corresponding person in TMDB, by issuing a request to TMDB with parameter `external_source=imdb_id`, as follows (where *N* is this IMDB-specific ID and *K* is your TMDB API key).

`https://api.themoviedb.org/3/find/`*N*`?external_source=imdb_id&api_key=`*K*

The result of this request is a JSON-encoded object *r* whose `person_results` field contains an array of *person objects* (in the format described in Assignment 6). Therefore, we can get the TMDB-specific ID of the (first) person of this response by the JavaScript expression *r*`.person_results[0].id`. (For example, Steven Spielberg has `nconst` `nm0000229` in IMDB and ID 488 in TMDB.) Given this TMDB-specific ID, you can get profiles (and therefore `file_path`s and URLs) of pictures of a person, as described in Task 7 in Assignment 6.

3-E Functional requirements

The frontend must provide a user interface to the movie application. The user interface must be designed such that it appears consistent and coherent.

3-E.1 Your solution should be a single-page application. (If you divert from this design, then explain where and why.)

3-E.2 The frontend must provide a navigation bar to transition between different types of content. It may also provide links to transition between content where you find it relevant.

3-E.3 Content with listed information must be provided with pagination when possible.

3-E.4 Features supported by the framework model should also be available from the frontend, such as registration of users and bookmarking of titles and people. (See Section 1-C.)

3-E.5 The functional requirements listed under Section 1-D should be supported by the frontend. In particular, the frontend should support searching for movies and people, and rating titles. Optionally, your frontend may display word clouds.

3-F Non-functional requirements

3-F.1 You should implement the frontend primarily using React, JavaScript, HTML, CSS, and (React) Bootstrap.

You are, however, allowed to implement part (but not all) of the frontend in TypeScript. In that case, you should document advantages and disadvantages resulting from the use of TypeScript over JavaScript. (Notice that we do not evaluate a solution more favorably just because part of it is implemented in TypeScript.)

You are also allowed to encode part (but not all) of the styling of HTML using other CSS frameworks than (React) Bootstrap. In that case, you must document significant differences, advantages, and disadvantages between the two frameworks and you must motivate the choice of a different framework. (Again, we do not evaluate a solution more favorably just because it uses a different CSS framework.)

3-F.2 You should aim for the implementation being modular, scalable, maintainable, testable, fault tolerant, and extendable. In particular, you should split the solution into separate (JavaScript and JSX or TypeScript and TSX) files with well-defined interfaces, and export only relevant components, functions, variables, constants, and classes from these files.

3-G Individual reflections

In addition to the implementation and project work done in your group, the final report must also include so-called *individual reflections.*

3-G.1 Each group member must write two pages that include his or her own reflections discussing one or more concepts from the course (taken from any of the four sections) and relate these to the group's product design or product implementation. If you select concepts not addressed by your design and implementation, you should explain how they might be relevant.