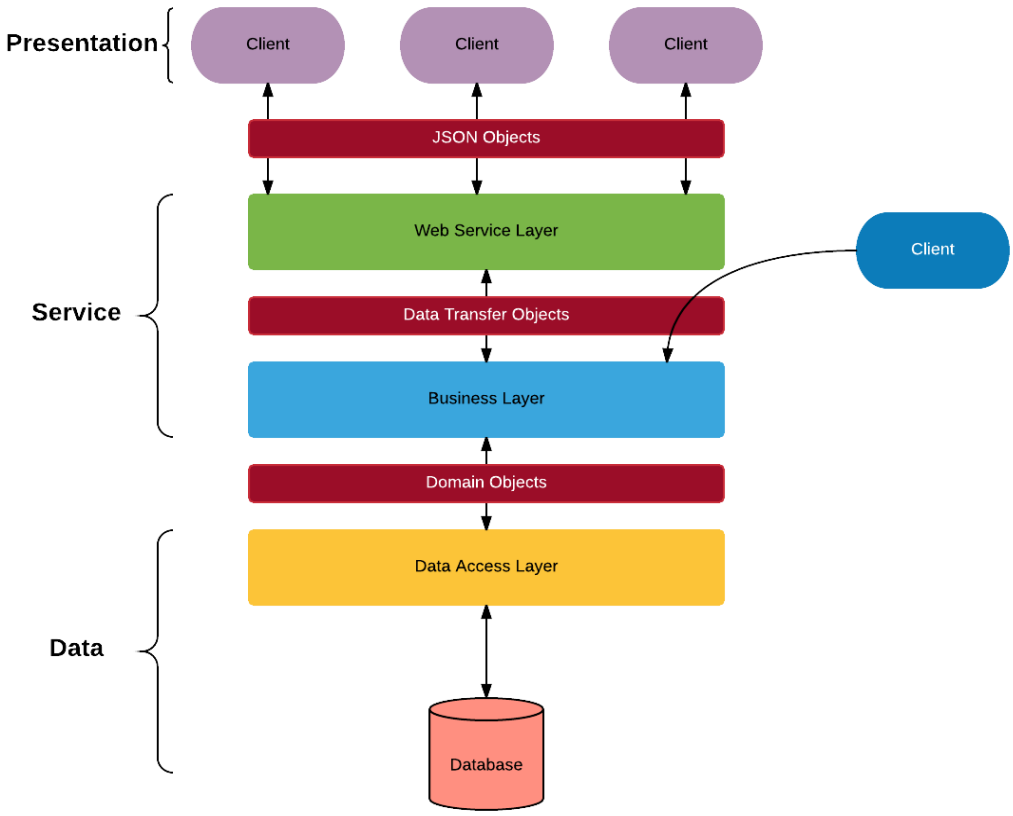
CITP Portfolio Subproject 2 requirements

(Please refer to the note: *CITP Introduction to Project Portfolio* for a general introduction)

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 below illustrates the overall architecture of the portion of the application to be worked on in this subproject:



In the figure above, the primary focus for the next subproject will be the clients located at the top. 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

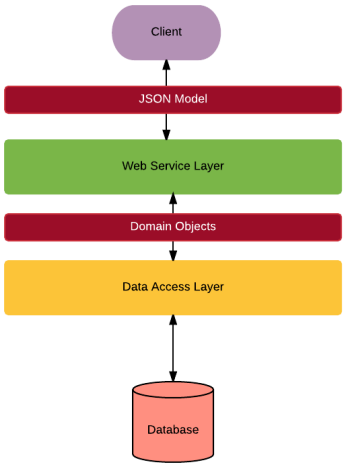
The Data Access Layer[1] (DAL) serves as the gateway to the database, conceals the intricacies of SQL, manages transactions, and facilitates the conversion between the relational model and the object-oriented model, as well as the reverse transformation when necessary.

The key design patterns employed within the data access layer include the Data Mapper[2], Repository[3], and Unit of Work[4] patterns, aimed at establishing a robust and adherent layer (SOLID[5]) on top of the database. Additionally, several other patterns, such as Façade[6], Singleton[7], Factory[8], and Bridge[9], 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.

Business Layer

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 (DAL), 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 on the right-hand side.

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.

---

[1] https://en.wikipedia.org/wiki/Data_access_layer
[2] http://martinfowler.com/eaaCatalog/dataMapper.html
[3] http://martinfowler.com/eaaCatalog/repository.html
[4] http://martinfowler.com/eaaCatalog/unitOfWork.html
[5] https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
[6] https://en.wikipedia.org/wiki/Facade_pattern
[7] https://en.wikipedia.org/wiki/Facade_pattern
[8] https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)
[9] https://en.wikipedia.org/wiki/Bridge_pattern

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 (WSL) is responsible 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 WSL 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.

A. Application design

Revising the initial application design presented in Portfolio 1 Subproject, we aim to define the use cases[10]/user stories[11] that the system must accommodate. From these, we will extract a high-level overview of the required requests to the web service.

A.1. Outline the architecture of the backend[12] system. [Provide an overview of the overall system architecture]
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.
A.3. Document the structure of the domain objects, data transfer objects, and the JSON objects (the Outer facing contract).
- 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.

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

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.

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 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.
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 Create, Read, Update, and Delete (CRUD) operations as necessary. Ultimately, the repositories or services' purpose is to define an interface that streamlines communication between the Web Service Layer

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

(WSL)/Business Layer (BL) and the Data Access Layer (DAL) by furnishing the essential functionality.

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

C. Web Service Layer

The gateway to the backend system is the Web Service Layer (WSL). 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 A.1-3 and 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 (BL)/Data Access Layer (DAL). Importantly, the WSL should remain independent of the data sources, ensuring that modifications to the data sources do not necessitate alterations in the WSL.

Adhering to the principles of the Representational State Transfer (REST) architectural style, while maintaining flexibility, is key. The desired interface should be stateless, uniform, self-descriptive, 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.

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 A. Thoroughly document the characteristics of this interface.

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

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

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

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.

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.

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.

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 (WSL) and Data Access Layer (DAL). 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 WSL and DAL. 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 (JWT)[13]. 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 this tutorial.

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 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 (WSL), 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.

[13] https://jwt.io/introduction/

The project report

You are expected to continue collaborating within the same groups established during Portfolio 1. Your task for Portfolio 2 involves implementing the updated database, which includes new functionality and any necessary updates, on the course server (cit.ruc.dk). Following this implementation, commit your solution to GitHub or a similar version control resource. Additionally, you are required to submit a Portfolio 2 report in PDF format. This report should include an URL linking to the source code. If security features have been incorporated, provide the login information within the report.

The report should have a length of approximately 8-12 standard pages[14], excluding any appendices. The submission deadline for both the report and the project itself is November 6, 2023.

Please note that the report submitted for Portfolio 2 is not meant to be revised later. However, if there are valid reasons for making design and implementation changes in the next section, you can document these alterations in the subsequent report. Therefore, the division of the Portfolio into subprojects should not hinder iterative development and improvement.

[14] A normal page corresponds to 2400 characters (including spaces). Images and figures are not counted.