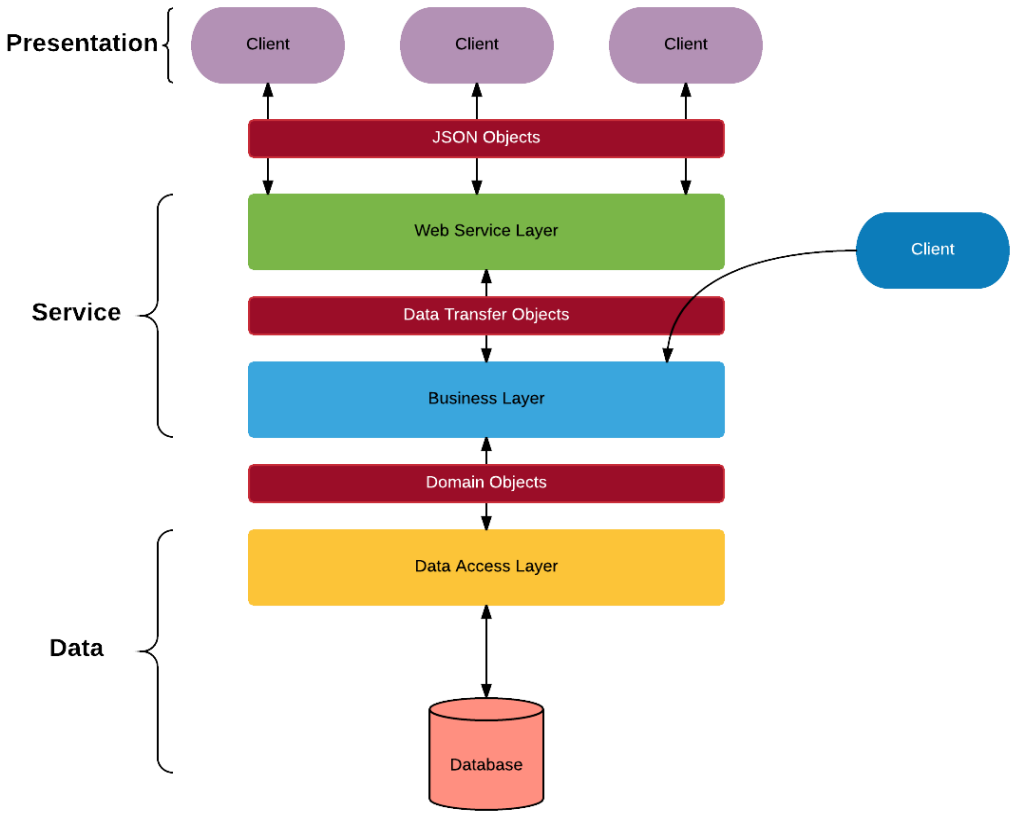
RAWDATA Portfolio-project 2 requirements

(Please refer to the note: *RAWDATA Portfolio project* for a general introduction)

The goal of this portfolio subproject 2 is to add a restful web service interface to the SOVA application (Stack Overflow Viewer Application) and to extend the functionality of this. We want the architecture to be as maintainable, testable, extendable, and scalable as possible, thus do it the "right way" even through the complexity of the model and application for now not necessary justify this. We assume, of course, that the system will need expansion in the future, due to all the nice features.

The overall architecture of the part of the application to be developed in this subproject is shown in the following figure:



The clients in the top will be the aim for the next subproject. The blue client to the right just illustrate that there might be other components over time, that need access to the system. The red boxes between the layers show the kind of data flowing between the components (layers), e.g. between the business layer and the web service layer Data Transfer Objects (DTO) will be used. The reason for using different data structures between the layers in the system is to decouple the elements, such that change in the underlying layer not necessary imply changes in the layers higher up the stack.

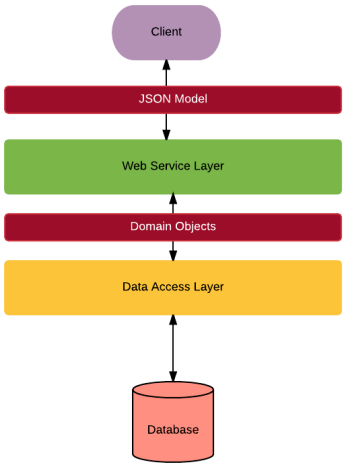
Data Access Layer

The Data Access Layer[1] (DAL) provides the interface to the database, abstracts away the SQL, handles transactions, and adapts the relational model to the object-oriented model, and vice versa.

The primary patterns used in the data access layer are the Data Mapper[2], Repository[3] and Unit of Work[4] patterns, in order to create a SOLID[5] layer on top of the database. Other patterns that are commonly used to support good designs in this layer is Façade[6], Singleton[7], Factory[8] and Bridge[9]. Using the Entity Framework takes care of the implementation of the data mapping and the unit of work. The repository pattern defines the interface to the data, but the one Entity Framework provides is very general, just providing an interface to all the collections in the data context. We obviously need a more specific and fine-grained interface that support the specific system under development.

Business Layer

The business layer (BL) is the center of the application that handles the business logic and manages behaviors. This layer should cover all the logic of your system, except, of course, for the part that is delegated to the database. For simple applications the business layer could merge with the DAL, thus just be an interface to the database, while it for more complicated applications, will abstract away the database and work on an independent domain model. For this project it is most likely that the former is adequate, i.e. that we can merge the BL and DAL layers, thus the architecture will look like the figure to the right.



Web Service Layer

The Web Service Layer (WSL) must provide a uniform interface to the resources that it exposes, i.e. if the client knows how to get to one resource it should be able to follow the same pattern to get to other resources. For example, if the URI to the post with id 123 is `http://server/api/posts/123` then the same pattern can be used to retrieve the comment with id 234 `http://server/api/comments/234`. The output from WSL should be objects in either JSON or XML format, as specified by the clients through content negotiation.

The WSL defines the routing between the URI interface and the controllers that knows to access the BL/DAL to retrieve or update data.

---

[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/Singleton_pattern
[8] https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)
[9] https://en.wikipedia.org/wiki/Bridge_pattern

A. Application design

Sketch a revision of your preliminary application design presented in portfolio 1 subproject. Try to define the use cases[10]/user stories[11] that the system must support. From these, try to extract an overview of the needed requests to the web service.

A.1. Document the architecture of your "backend"[12] system.
A.2. Draw class diagrams to show the dependencies within the layers and between layers.
A.3. Document the structure of the domain objects, the data transfer objects, and the JSON/XML objects

B. The Data Access Layer

In environments where the data sources are many and may change over time, the system need to abstract away the concrete implementations and provide a more generic interface not bound to the sources. The pattern commonly used to solve this is the Repository pattern, by defining an abstract interface to data that can be implemented for various resources.

B.1. Define and document the domain model, i.e. how to map the relational model into an object-oriented model. This requirement is closely related to A.3, but should focus on the object-relational mapping, i.e. how is the relational model in the database mapped into the object-oriented domain model.
B.2. Create a database access layer based on the Object-Relational mapping by use of the Entity Framework. Create the necessary repositories/service(s) to create an abstract interface to the database for the CRUD operations needed. Eventually, the purpose of the repositories/service(s) is to define an interface that will simplify the communication between WSL/BL and the DAL, by providing the necessary functionality.
B.3. Prepare the data access layer for authentication, i.e. adding needed parameters to methods to handle authentication.

C. Web Service Layer

The interface to the backend is the web service layer (WSL). Here the system must expose what is needed for the frontend to be developed in the next section. The restful web services must be implemented with use of ASP.NET Web API. The overall structure of this part will be to reply to requests from clients by use of the interface provided by the BL/DAL. WSL should not have any dependencies on the data sources. It is important to follow the ideas from the Representational State Transfer architecture style, but not to be dogmatic. We want an interface that is stateless, uniform, self-descriptive, and resource-oriented. By resource-oriented we want the interface to be focused on the underlying resources, i.e. focus on nouns and not verbs, e.g. …/posts/123 instead of the RPC style getPost(123) to get the post with id 123.

C.1. Create a web service interface (design the URIs) to the read only Stack Overflow data. Imagine the needs of the user interface. Use the use cases/user stories from A to get a picture of the requirements. Document the interface.
C.2. Implement the interface defined in C.1
C.3. Create a web service interface (design the URIs) to the annotation data. This should implement the full set of CRUD operations for all the resources. Again, imagine the

[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.

needs of the user interface and use the use cases/user stories from A to grasp the requirements. Document the interface.

C.4. Implement the interface defined in C.3.

C.5. All responses must support the idea of having a self-descriptive interface by providing self-references, i.e. the URI to the objects themselves. This also means that when returning a list of objects, each object must contain a reference to get that specific object.

C.6. All listing operations must implement paging, with a default page size and the possibility to set the size from the clients (using the default size if the request is too large or not defined). The page result must contain links to the previous and next page (if such exists).

C.7. [OPTIONAL] Use ETags to support caching.

C.8. [OPTIONAL] Use ETags to support optimistic locking.

D. Security

Obviously, applications like SOWA need also to be secure, since personal information will be stored. Security is complicated, especially if you want to do it on a reasonable level. However, in this project "high level" security is not part of the goal. The focus is primary on understanding, designing and implementing a full stack.

D.1. The backend must nevertheless handle users, such that all parts of the interface related to users keep track of which user did what, i.e. search history, markings, notes, etc. Your database and its interface are prepared for this, and thus the WSL and DAL must also be. Later, in the frontend, you can provide users login to the system, thus we need to capture and use this information in the communication between clients and our service. As mentioned earlier, restful webservices are stateless and thus authentication information must be sent to the WSL as part of the requests (i.e. the state of users are not stored in the backend). You can choose different strategies for handling users in your project, from a very simple to a real secure solution.

i. The simplest solution is to hardcode one specific user into the backend and ignore login of different users. This will give a system prepared for "users", but not actually using them for authenticated access to resources.

ii. The next level would be to send the username inside requests that need authentication by use of the HTTP `Authentication` header field[13]. With this solution you can handle simple authentication and respond to unauthorized request and apply the logic for handling authentication in WSL and DAL. Obviously, the system will not be secure, since anybody can just fiddle with the request to gain access to the system.

iii. The common way to solve the authentication problem in restful webservices is to use tokens and HTTPS[14] (secure HTTP). A token is way to transmit signed information from the client to the server. The token is encoded and signed information about the user (username) and timeout (validity) such that the server can verify whether the user should gain access to the requested resource or not. One such popular token system is the JSON Web Tokens (JWT)[15]. The server will generate the token at login, and the client will save and provide the token in the HTTP `Authentication` header field on requests. ASP.NET supports authentication by adding middleware and use of annotations. The middleware

[13] https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication
[14] https://en.wikipedia.org/wiki/HTTPS
[15] https://jwt.io/introduction/

knows how to unfold the information packed into the token an verify the user, and the annotations, e.g. [Authorize], specify that the user need to be authorized to get access to the part annotated (method or class). You can find a small tutorial here.

E. Testing

Every single part of the implementation must be tested, both on the unit level and for integration. By every part we do not mean every class, but examples on testing the different layers and the different elements in the layers, i.e. if you have more repositories/services you only need to show how to test one of them. For the WSL part, UI test must be provided, i.e. testing that the service returns the expected data in the right format.

The project report

You are supposed to continue in the groups from portfolio 1 and submit your portfolio 2 result by implementing the revised database (including new functionality and updates if any) on the course server (rawdata.ruc.dk) and committing the solution, including projects for each part(layer) to GitHub (or similar resource) with a Section2 tag. The Portfolio project 2 report (in pdf format), including an URL to the source must be uploaded to the course website on Moodle.

The report should in size be around 10-12 normal-pages[16] excluding appendices. The submission deadline for the report as well as the product is 7/11-2018.

Notice that the report you hand in for Portfolio project 2 is not supposed to be revised later. However, if you find good reasons for this, your design and implementation can be subject to revision later. Documentation for such changes can be included in the following reports. Thus, the division of the Portfolio in subprojects is not intended to stand in the way for iterative development.

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