

Stack Overflow Viewer Application (SOVA)



Project group: raw4

Özge Yaşayan

Shushma Devi Gurung

Ivan Spajić

Manish Shrestha

MSc. in Computer Science

Date: 18/11/2019

Link to GitHub repository: <https://github.com/ivanspajic/SOVA/tree/2.0/Subproject2>

Table of Contents

Table of Contents	1
1. Introduction	2
2. Application design	2
2.1 Overview of requests to the web service	3
2.2 Architecture of the backend system	3
2.2.1. Three-tier architecture	3
2.2.2. Class diagrams	4
2.2.3. Data Access Layer	5
2.2.4. Business Layer	6
2.2.5. Web Service Layer	6
2.2.5.1. Requesting for Stack Overflow data	6
2.2.5.2. CRUD operations on Framework data	7
3. Security	8
3.1. Passwords	8
3.2. Authentication and authorization	9
3.3. Visual summary	10
4. Testing	12
5. Refactor of Subproject 1	14
5.1. Updating Users table to include salt column	14
5.2. Updating users table to rename name column to username	14
5.3. Updating stored procedure which updates history table to take in an optional userId parameter	15
5.4. Updating search functions to also take in an optional user_id parameter to pass to the stored procedure	16
6. Future work	17
7. Appendices	17
7.1. User stories	17
7.2. API Documentation	18

1. Introduction

The section of Project Portfolio, subproject 2 is the continuation of subproject 1 of Stack Overflow Viewer Application (SOVA). Previously, we had built two different data models: QA data model and Framework model. The first model is for a keyword-based search that can be used to find answers to questions related to programming. The second model is meant to support the framework which includes users, markings, annotations and history.

The purpose of subproject 2 is to apply a RESTful web service interface to the SOVA application and to extend its functionality. We aimed at keeping the architecture maintainable, testable, extendable and scalable. For subproject 2, we are using three-tier architecture. The three layers refer to View Layer (UI), Database Access Layer (DAL) and Business Logic Layer (BLL). It is built using ASP.NET Core framework.

2. Application design

Sketch of revised preliminary application design.

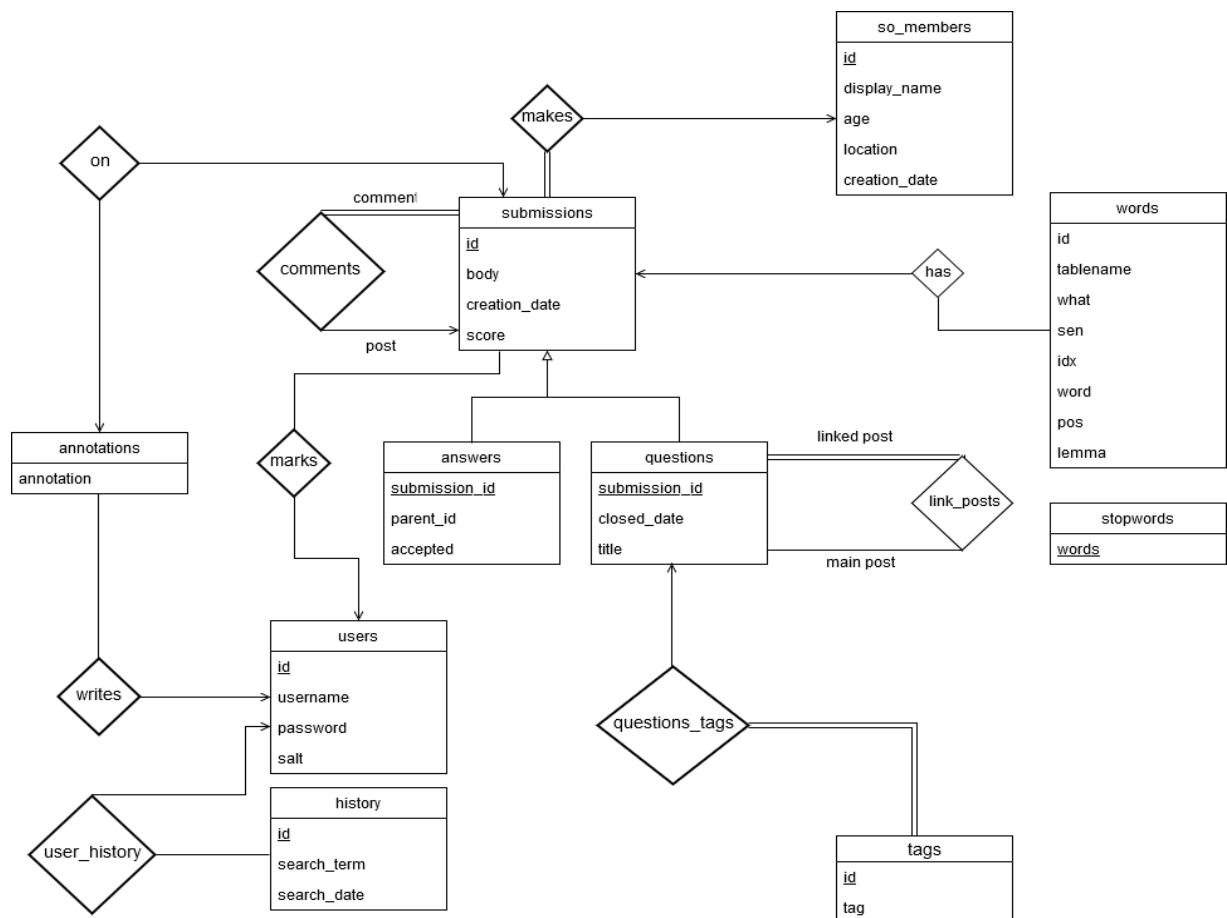


Fig 1. Revised preliminary application design

2.1 Overview of requests to the web service

Based on the user stories¹, we can extract the needed requests to be made to the web service. The API documentation² is provided in the appendix which describes how to interact with the webservice to make the following requests.

- Get question by id
- Search question by query string
- Filter question by tag
- Get answers for a question by question id
- Get answer by answer id
- Get all comments by post id
- Create a user
- Update an existing user by id
- Get authentication token
- Get annotation for a post
- Create annotation for a post
- Update annotation for a post
- Get user's history
- Get user's bookmarked posts
- Update bookmark by post Id

2.2 Architecture of the backend system

Backend is the server side of the website where data is stored, manipulated, or updated and sent to client on request. The backend serves data to the frontend. In our backend system, we used C# to build functionalities behind the scenes. Security part was also handled here.

2.2.1. Three-tier architecture

The idea of using three-tier architecture is mainly used to implement "high cohesion and low coupling" for each module of the project.

In the process of project development, we may need to repeatedly make operations to the database, repeatedly use a certain method, etc. It may be that the operation parameters are different. If we write code for each use, it will undoubtedly increase the task for the developer, make the code vulnerable to bugs and it also violates the Don't Repeat Yourself (DRY) principle. In order to maintain reusability, these methods are abstracted into classes that developers can call elsewhere. The three layers refer are Presentation Layer (UI), Business Logic Layer (BLL) and Data Access Layer. Data is accessed to presentation layer by Business Logic Layer through API calls. There are many advantages of using 3 layer architecture. It helps in doing development faster, better, performance, scalability and availability. While one layer is upgraded the other layers will be less impacted. Separating

¹ Appendix 7.1. User Stories

² Appendix 7.2. API Documentation

different parts of an application helps minimize performance issues when a server goes down.

2.2.2. Class diagrams

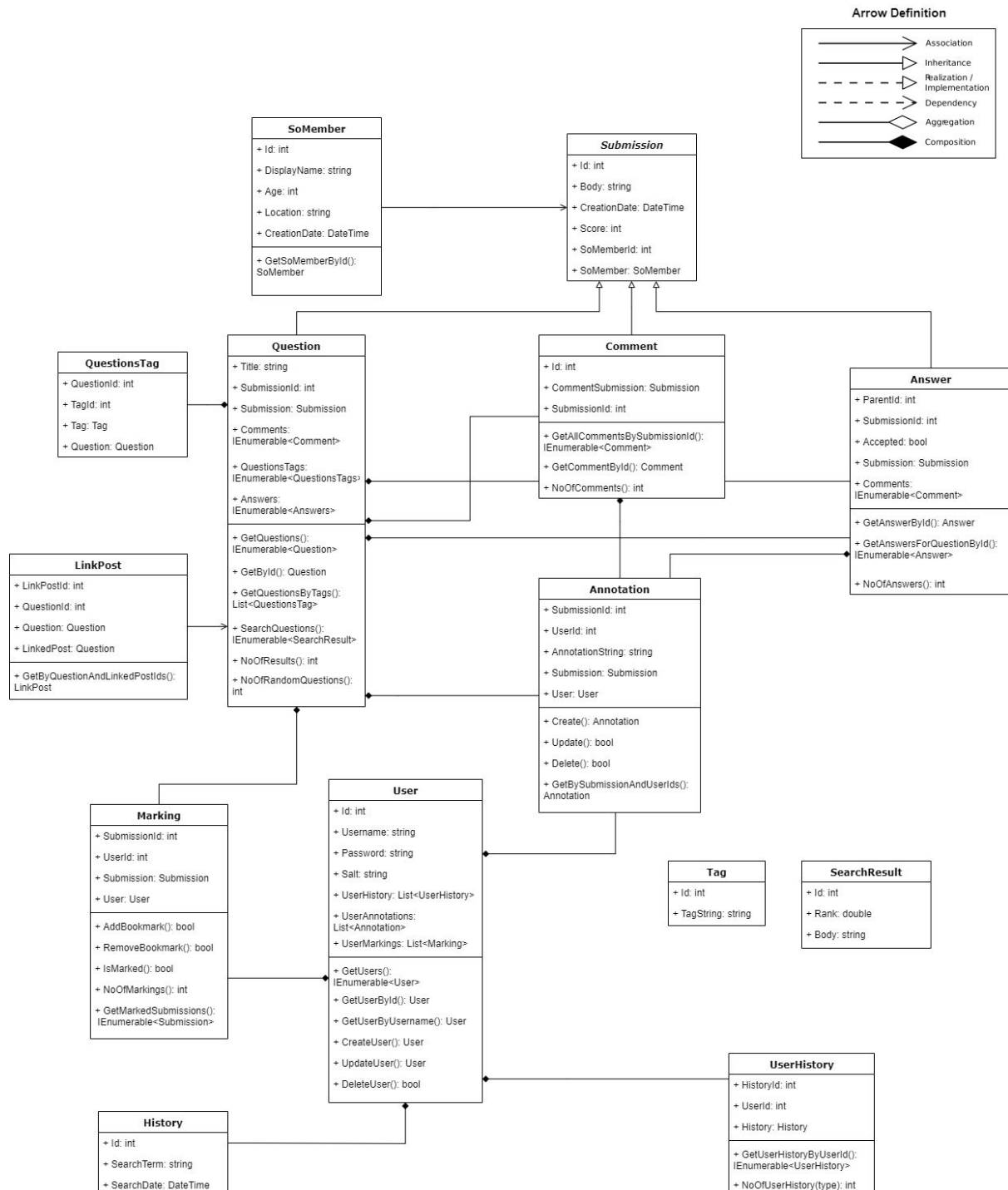


Fig 2. Class diagram³

³ [Hosted on GitHub for enlarged view.](#)

The models can be observed in our class diagram.

We have DTOs for the following along with the information they contain:

- Annotation: the link to the annotation and the annotation text.
- Answer: the link to the answer, the parent post's id, whether the answer is the accepted answer and the Submission object that contains other information related to that answer.
- Comment: the link to the comment, the id of the comment and the Submission object that contains other information related to that comment.
- Question: the link to the question, the title of the question, the date when the question was closed, the Submission object that contains other information related to that question, and the id of the question.
- Submission: the link to the submission, the id of the submission, the body/text of the submission, the creation date of the submission, the score of the submission, the id of the Stackoverflow member that posted it, the Stackoverflow member object that contains the information about that member.
- User: the id of the user, the link to the user's profile, the username and the password of the user.
- User history: the link to the user's history, the id of the user history, the id of the user, and a History object that contains the information about the search term that the user searched for and the date of the search.

When the data gets sent to the clients, they receive it as JSON objects.

2.2.3. Data Access Layer

Object-relational mapping

In our project, we used Entity Framework which took care of the data mapping. We also used repositories. Between the models and the data layer, we had data layer abstractions which contained the interfaces for our repositories. Through the repositories, we were able to handle the CRUD operations in the web service layer. In short, the repositories contained many functions related to the transactions and they helped abstract the SQL away.

In the database context, we mapped the entities to tables and properties to columns. We also indicated the database sets, primary keys, foreign keys and mapped SQL query results from the SQL functions we used.

The data access layer was also prepared for authentication. The functions in the repositories that required a sign-in were adjusted to take the necessary parameters to handle authentication.

2.2.4. Business Layer

In our application, business layer is merged with the data access layer. However, for more complicated applications, it is important to have a business layer that can handle the business logic as well as manage behavior of the application.

2.2.5. Web Service Layer

2.2.5.1. Requesting for Stack Overflow data

A web service layer is an interface to the resources it exposes to the client. Since Stack Overflow platform revolves around questions, that's how most of our interfaces relating to Stack Overflow data are designed. The standard interface follows the following convention within our backend.

`https://<serverAddr>:<port>/api/<endpoint>`

To explore the data from Stack Overflow, all the requests made to the backend will be a **GET** request as we will be simply fetching the data without providing any data in the body. Based on the user stories, we designed the web service interface as shown in the table below.

Get question by id	GET	/api/questions/{questionId}
Search question by query string	GET	/api/questions/query/{queryString}
Filter question by tag	GET	/api/questions/tag/{tagString}
Get answers for a question by question id	GET	/api/questions/{questionId}/answers
Get answer by answer id	GET	/api/answers/{answerId}
Get all comments by post id	GET	/api/{postId}/comments

As aforementioned, most of it (if not all) is based on questions. Hence, the URI begins with /api/questions/ and extends on it based on the requirement. To get a question by id, we need to provide the question id as a query param. For example, if we would like a question with id 19, the URI would look like:

`https://<serverAddr>:<port>/api/questions/19`

Similarly, we can search for questions by a search term, filter them by tags, get answers, comments and linked post using the interface defined on the table above. The interface is well-defined, self explanatory and user friendly as with a quick glance, it becomes obvious that the values in red needs to be changed to appropriate id or a string with the help of interface description.

2.2.5.2. CRUD operations on Framework data

Prerequisite: For any CRUD operation within the Framework data, the user needs to be authenticated as all of this data is linked to the user id. The process to get authenticated and providing authentication token is described in details in “Security” section below.

As an extension, a user within our application can have annotations, history and bookmarks. This consists of more operations than simply making a **GET** request like in Stack Overflow data. Users can do CRUD operations with data which they own. By CRUD operations, we mean CREATE, READ, UPDATE and DELETE operations which translates to HTTP methods **POST**, **GET**, **PUT/PATCH** and **DELETE**. After all, we are building a RESTful web service.

For now, let's focus on CRUD operation of annotations. Annotations are personal comments which the users can add to Stack Overflow data (questions, answers and comments) for future reference or note-taking. As with questions interface, the interface for annotations follow a similar approach. We do want it to be user and (more importantly) developer friendly. The interface for all the CRUD operations for annotations starts with:

`https://<serverAddr>:<port>/api/annotations/`

Depending on what action a user want to do, the correct verbs, in RESTful API terms, has to be chosen. Here is a handy list of interface with descriptions and request type to perform CRUD operations by an authorized user.

Get annotation for a post	GET	<code>/api/annotations/{postId}</code>
Create annotation for a post	POST	<code>/api/annotations/{postId}</code>
Update annotation for a post	PUT	<code>/api/annotations/{postId}</code>
Delete annotation for a post	DELETE	<code>/api/annotations/{postId}</code>

In our application, we have designed it such that a user can only have a single annotations per post. It was a group decision.

Let us take question with id 19 again for this example. If an authorized user would like to create an annotation on this question, they would make a **POST** request with a JSON body which houses the annotation.

- Provide authentication token in the header (described in “Security” section)
- Provide a JSON body with Annotation

```
{  
  "AnnotationString": "This is an important question."  
}
```

- Make a **POST** request to URI:
`https://<serverAddr>:<port>/api/annotations/19`

The process to update the annotation is very similar. It would be **PUT** request with an updated JSON body. For requesting and deleting operations, the body is not necessary. Simply changing the request type with the query parameter (question id 19, in this case) is sufficient.

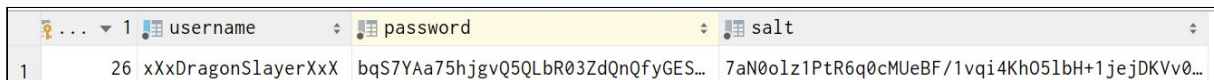
The rest of the interfaces for different endpoints are listed and described in the appendix for reference.

3. Security

Within our application, we have taken steps towards security for database connection, user creation, authorization and user data storage. For the basics, when connecting to the database, we fetch the connection string (which includes personal credentials when working in a group) from a JSON file located individually on every group member's local machines and not checked in to the source control.

3.1. Passwords

When creating a user, the user credentials are sent through an encrypted connection using HTTPS. The password is then hashed with their unique salt making it incomprehensible. Only then it is written onto the database. See figure xxx for clearer picture.



	username	password	salt
1	26 xXxDragonSlayerXxX	bqS7YAa75hjgvQ5QLbR03ZdQnQfyGES...	7aN0o1z1PtR6q0cMUeBF/1vqi4Kh051bH+1jejDKVv0...

Figure 3: User stored in the database

As you can see, the id of the user is 26 and the username is xXxDragonSlayerXxX but password and salt is a long, arbitrary string. The following methods are used to create the salt and hashed password.

```
// Subproject2/SOVA/Service/PasswordService.cs

public static string GenerateSalt(int size)
{
    var buffer = new byte[size];
    _rng.GetBytes(buffer);
    return Convert.ToBase64String(buffer);
}

public static string HashPassword(string pwd, string salt, int size)
{
    return Convert.ToBase64String(KeyDerivation.Pbkdf2(
```

```
        pwd,
        Encoding.UTF8.GetBytes(salt),
        KeyDerivationPrf.HMACSHA256,
        10000,
        size));
    }
```

The variable `size` is being read from another JSON file within the project (`Subproject2/SOVA/appsettings.json`) and the algorithm to hash the password and the iteration count (10000 in this case) is set by the developers. So even if the database was to be compromised, the hackers would have a hard time decrypting users' passwords.

3.2. Authentication and authorization

We have implemented token based authentication/authorization in our application using JSON Web Token (JWT) as it is widely used and accepted. JWT.io defines these tokens as a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.⁴

When a user makes a request for authentication, the provided password is hashed using the initially created salt for this user, and then matches the hashed password stored in the database. If it does not match, the backend responds with 400 Bad Request.

```
// Subproject2/SOVA/Controllers/AuthController.cs

// Get the user from the database
var user = _userRepository.GetUserByUsername(dto.Username);
if (user == null)
{
    return BadRequest();
}
// Hash the provided password with salt from the database
var pwd = PasswordService.HashPassword(dto.Password, user.Salt, _size);

// Match the above password with password in the database.
if (user.Password != pwd)
{
    return BadRequest();
}
```

⁴ Source: JWT.io, What is JSON Web Token? <https://jwt.io/introduction/>

If the password matches, the next step is to create an authentication token (JWT). We include the user id and the token expiry in the token claims. JSON Web Token (JWT) claims are pieces of information asserted about a subject.⁵ This token is then signed with a key which is also stored in the previously mentioned JSON file within the project which is an arbitrary string and HMAC SHA256 hash function.

```
// Subproject2/SOVA/Controllers/AuthController.cs

var tokenHandler = new JwtSecurityTokenHandler();
// Fetch the key from the JSON file within the project folder
var key = Encoding.UTF8.GetBytes(_configuration["Auth:Key"]);

// Set the token claims and sign the token with this key and HMAC SHA256
// hash function
var tokenDescription = new SecurityTokenDescriptor
{
    Subject = new ClaimsIdentity(new Claim[]
    {
        new Claim(ClaimTypes.Name, user.Id.ToString()),
    }),
    Expires = DateTime.Now.AddDays(1),
    SigningCredentials = new SigningCredentials(
        new SymmetricSecurityKey(key),
        SecurityAlgorithms.HmacSha256Signature)
};

// Create the token and respond to the user
var securityToken = tokenHandler.CreateToken(tokenDescription);
var token = tokenHandler.WriteToken(securityToken);
return Ok(new { user.Username, token });
```

The backend responds with an authentication token which the user can then add it to the request header when making requests.

3.3. Visual summary

1. The user sends a request with credentials in the request body.

⁵ Source: Auth0 Docs, JSON Web Token Claims <https://auth0.com/docs/tokens/jwt-claims>

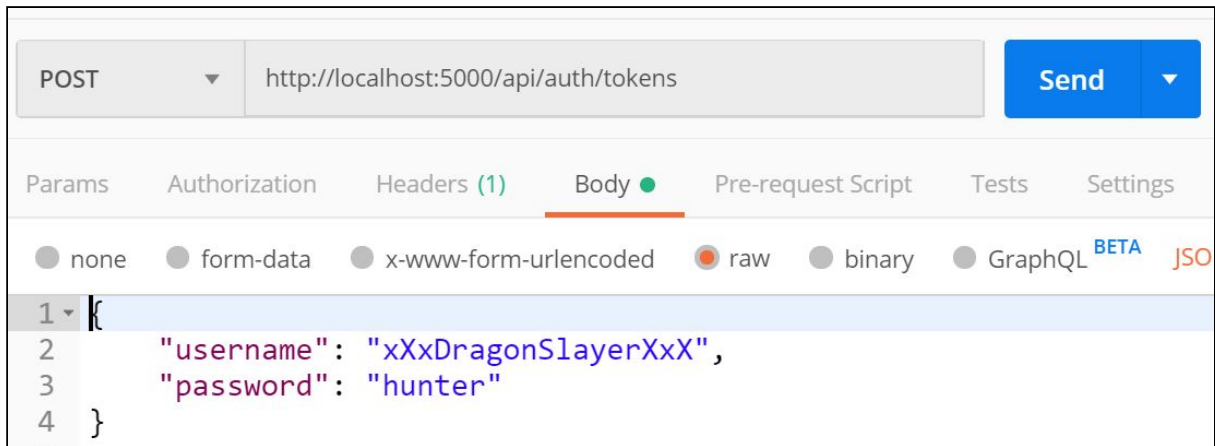


Fig 4. Request with credentials

2. The backend matches credentials, signs an authentication token (JWT) and responds with said token.

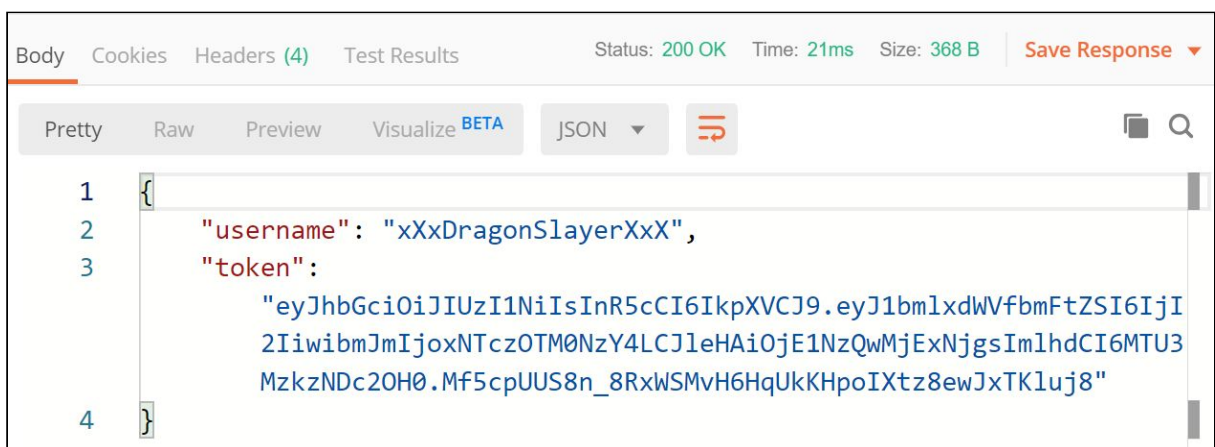


Fig 5. Response with authentication token

3. User can send this token in the header for making requests requiring authentication.

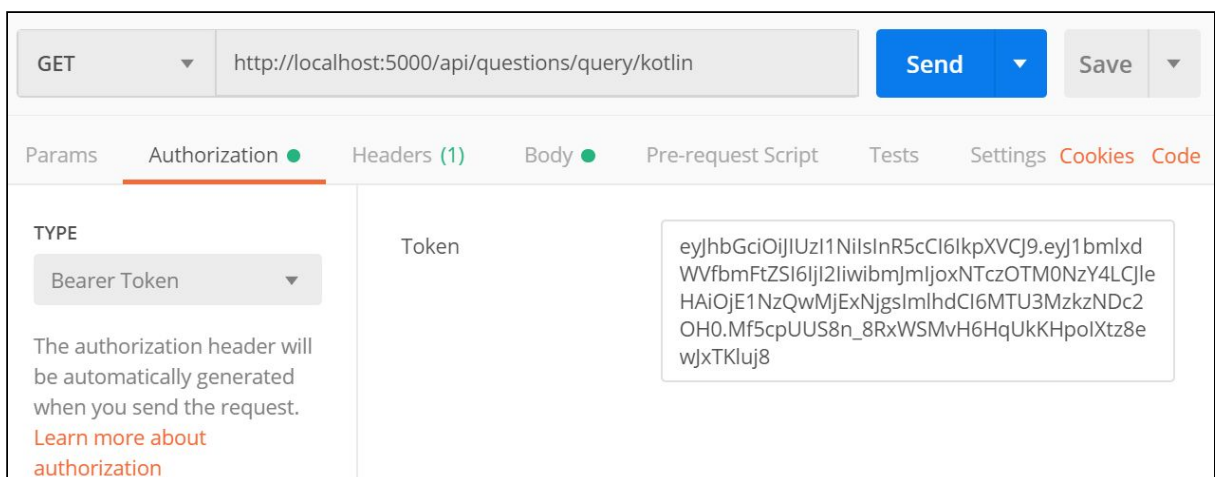


Fig 6. Request with authentication token in header

4. This token gets used in methods like searching to populate user history for example.

```
// Subproject2/SOVA/Controllers/QuestionsController.cs

// This method searches and adds the search to user's history. Hence we
// need user to be authenticated.

[Authorize] // User needs to be authorized to access this method.
[HttpGet("query/{queryString}", Name = nameof(SearchQuestion))]
public ActionResult SearchQuestion([FromQuery] PagingAttributes
pagingAttributes, string queryString)
{
    // If the authorization is successful, the userId gets saved in the
    // userId variable.
    int.TryParse(HttpContext.User.Identity.Name, out var userId);

    // Perform search which also populates the user's history
    var searchResults = _questionRepository.SearchQuestions(queryString,
userId, pagingAttributes);

    return Ok(CreateSearchResult(searchResults, queryString, userId,
pagingAttributes));
}
```

4. Testing

As is the case with any software solution, testing must be done to ensure the requirements were met as well as the quality of the code. This solution is comprised of multiple tiers integrated to work together as a whole, however they cannot be tested together from the very beginning. Newly developed software must first be tested in small functional units, covering the essential functions, serving as mere parts of actual user stories, before moving up on the scale and testing everything as a whole. For this purpose alone, unit tests were utilized for the second part of the project portfolio.

As mentioned, the second part of the project covered both the data services (repositories) serving as a link between the Web API controllers and the database, as well as the controllers themselves. Both layers contained their own functionalities that could be tested with unit tests, however due to time constraints, unit tests were written only for the data service layer while the Web API layer was tested through usability testing, mostly by using the Postman tool.

When writing unit tests, one must have certain principles in mind to ensure maximal protection against faulty cases of code working accidentally or just not at all. All unit tests

must thus adhere to certain design rules. Firstly, it is important that each test method is written with a specific test case in mind. Unit tests that cover multiple different tests are not exact and may result in faulty positives due to certain state combinations. Unit tests must also be independent of each other and themselves, such that it should be possible to run unit tests in a different order multiple times without their results varying.

All unit tests should follow the AAA (Arrange, Act, Assert) design pattern to ensure simplicity and structure. The first step is to arrange the proper variables and dependencies for the creation of the appropriate test case conditions. Secondly, the test must act and invoke to acquire the results of what it should test. Lastly, the test should perform a truth assertion on some data to conclude the correctness of the tested functionality.

Since the entire solution was developed with the .NET Core framework, the Xunit testing framework was chosen for the deployment of unit tests. Typically, test methods would carry a “Test” attribute in .NET to signify that they are unit tests, however Xunit uses the “Fact” attribute instead. Xunit does however offer a neat solution regarding edge cases, where the developer can save time. Usually, a developer would have to write very similar methods for slightly different edge cases, or they would perhaps develop a method that the different test methods would then provide with different arguments. Xunit combats this with the “Theory”, and “InlineData” attributes. Much like “Fact”, “Theory” signifies that a method is a unit test, however it will be run multiple times during the testing cycle. The number of times depends on the number of “InlineData” attributes provided. Each “InlineData” attribute provided usually carries a different set of arguments representing a specific edge case which will be executed by the Xunit testing framework.

```
[Theory]
[InlineData("", 19, 1)]
[InlineData(" ", 19, 1)]
[InlineData(null, 19, 1)]
[InlineData("Test Annotation", 0, 1)]
[InlineData("Test Annotation", -1, 1)]
[InlineData("Test Annotation", 19, 0)]
[InlineData("Test Annotation", 19, -1)]
[InlineData("", 0, 0)]
public void CreateAnnotationOnSubmissionForUser_InvalidArguments(string annotation, int submissionId, int userId)
{
    // Arrange
    SOVAContext databaseContext = new SOVAContext(_connectionString);
    AnnotationRepository annotationRepository = new AnnotationRepository(databaseContext);

    // Act
    Annotation actualAnnotation = annotationRepository.Create(annotation, submissionId, userId);

    // Assert
    Assert.Null(actualAnnotation);
}
```

Fig 7. XUnit test

Since unit tests must be independent from each other and must also contain their scopes within testing a single function within the software solution, it is imperative that there are no dependencies on unnecessary modules. In the case of testing repositories, it is important to test a single repository and (if possible) a single method within that repository, representing a single unit of functionality. For example, a repository that handles user CRUD operations should optimally have its methods tested without the need for invoking other methods in the

same unit test. If the “Create” method is to be tested, the assertion for a recently created object should not use the “Read” method from the same repository, since these two should work independently, and the test should not have to rely on two functional units working to test just one. Such cases were bypassed in the SOVA solution by performing checks directly with the database contexts used, while only invoking repository methods when necessary for testing.

```
[Theory]
[InlineData("John Milla", null, null)]
[InlineData(null, "hunter1", "newsaltwhodis")]
public void UpdateUser_ValidArguments_SomeAlwaysNotNull(string username, string password, string salt)
{
    // Arrange
    SOVAContext databaseContext = new SOVAContext(_connectionString);
    UserRepository userRepository = new UserRepository(databaseContext);

    int userId = 1;

    // Act
    User actualUser = userRepository.UpdateUser(userId, username, password, salt);

    // Assert
    User expectedUser = databaseContext.Users.Find(userId);

    Assert.Equal(expectedUser, actualUser);
}
```

Fig 8. XUnit test

5. Refactor of Subproject 1

5.1. Updating Users table to include salt column

When working in the database section of the project, subproject 1, we were unaware of the implication of introducing authentication and authorization would mean we would have to store the salt as well. We have updated the users table to accommodate this new entity.

5.2. Updating users table to rename name column to username

In our project, we decided that a user would simply have a username and that would be their display name as well. No separate entity for display name. To make that consistent and make that knowledge easily transferable between the backend and the database, we updated the previously labelled name attribute to username in the users table.

5.3. Updating stored procedure which updates history table to take in an optional userId parameter

Previously, we had developed a stored procedure named `log_search` whose responsibility was to update the history table with the user's search query.

```
create or replace function log_search(string_to_log text)
    returns void as
$$
begin
    insert into history(search_term, search_date)
        select string_to_log, now();
end
$$ language plpgsql;
```

As can be seen, it did not relate the search term with the user in any way. It only took the search term as an argument and popped it onto the history table with current date. We updated this stored procedure to take in an additional argument `user_id`. With this additional argument, if it is not null, we can now update the `user_history` table with `user_id` and `history_id`.

```
create or replace function log_search(string_to_log text,
authenticated_user_id integer)
    returns void as
$$
declare
    search_history_id int;
begin
    insert into history(search_term, search_date)
        select string_to_log, now();
    -- Users can still search without being logged in. But it won't get
    recorded in users_history since no user_id is present.
    if authenticated_user_id notnull then
        select id
        into search_history_id
        from history
        where search_term = string_to_log
        order by search_date desc
        limit 1;
        perform log_search_for_user(search_history_id,
authenticated_user_id);
    END IF;
```

```

end
$$ language plpgsql;

create or replace function log_search_for_user(history_id int,
authenticated_user_id integer)
    returns void as
$$
begin
    insert into user_history(history_id, user_id) values (history_id,
authenticated_user_id);
end
$$ language plpgsql;

```

5.4. Updating search functions to also take in an optional user_id parameter to pass to the stored procedure

Since the stored procedures are called from within other functions (search functions in our case), we needed to refactor these search functions to take in the user id to be able to pass it to the stored procedure explained above. However, its default value is null in our system, as users are still allowed to search without being logged in.

Before refactor:

```

CREATE OR REPLACE FUNCTION best_match_weighted(VARIADIC w text[])
.
.
.
perform log_search(w_elem);

```

After refactor:

```

CREATE OR REPLACE FUNCTION best_match_weighted(authenticated_user_id
integer default null, VARIADIC w text[] default null)
.
.
.
perform log_search(w_elem, authenticated_user_id);

```

6. Future work

In this section, we discuss about the things we have missed due to lack of understanding during the development process or due to lack of time and resources.

- Integration tests: Currently, we have unit tests which covers the entirety of the data layer. Due to the lack of time, we were unable to add integration tests into the application.
- Search with multiple query strings: The data is prepared to handle search with multiple query strings but our current API for searching is designed to handle only single query strings.

7. Appendices

7.1. User stories

We came up with user stories in order to have an overview of what we needed to implement in our web service. Here are the user stories:

- As a user, I want to be able to search for questions so I can find answers.
- As a user, I want to be able to search and get the most relevant posts so that I can find the most relevant answers.
- As a user, I want to be able to see questions on the home page so I can explore.
- As a user, I want to be able to authenticate so that I can have a sense of security about my personal details and activities.
- As a user, I want to be able to update my username and password so that my personal data can be safe in case of user data being compromised.
- As a user, I want to be able to sign in so I can annotate on a submission.
- As a user, I want to be able to sign in so I can remove my annotation..
- As a user, I want to be able to sign in so I can bookmark a submission.
- As a user, I want to be able to sign in so I can remove my bookmark on a submission.
- As a user, I want to be able to sign in so I can have a history of my searches.
- As a user, I want to be able to search for questions even when I'm not signed in so I won't have to sign up.
- As a user, I want to be able to filter questions by tags so I can find out more about certain topics and improve my skills.
- As a user, I want to be able to get posts in a paginated view so that I can keep track of how many posts have been retrieved and visited.
- As a user, I want to be able to see a list of related posts so that I can find more relevant answers.

7.2. API Documentation

Get questions with random offset	GET	/api/questions/
Get question by id	GET	/api/questions/{questionId}
Search question by query string	GET	/api/questions/query/{queryString}
Filter question by tag	GET	/api/questions/tag/{tagString}
Get answers for a question by question id	GET	/api/questions/{questionId}/answers
Get answer by answer id	GET	/api/answers/{answerId}
Get all comments by post id	GET	/api/{postId}/comments
Create a user	POST	/api/auth/users
Update an existing user by id	PATCH	/api/auth/users/{userId}
Get authentication token	POST	/api/auth/tokens
Get all users	GET	/api/users
Get user by id	GET	/api/users/{userId}
Get annotation for a post	GET	/api/annotations/{postId}
Create annotation for a post	POST	/api/annotations/{postId}
Update annotation for a post	PUT	/api/annotations/{postId}
Delete annotation for a post	DELETE	/api/annotations/{postId}
Get user's history	GET	/api/history
Get user's bookmarked posts	GET	/api/bookmarks
Update bookmark by post Id	PUT	/api/{submissionId}/bookmarks

Note: We will be using Postman to demonstrate the API requests. You can use any similar software of your liking.

We have developed endpoints which is exposed for clients to interact with the system. The users can read StackOverflow data and perform CRUD operations on the framework data model which we developed on the first section of the project. The details of the development is described on the Web Service Layer section above. Here, we will expand on how to interact with the system using these endpoints.

Get answers for a specific question

If a user wants to read the StackOverflow data, they need not be authenticated. That is, retrieving the questions, answers, comments and searching them using keywords or tags can be performed without the user having to log in.

For example, to get the answers for question with id 1101818, we simply make a **GET** request like described in the documentation above, i.e to the endpoint `/api/questions/1101818/answers`. The response from the backend looks like:

```
{
  "totalItems": 8,
  "numberOfPages": 1,
  "prev": null,
  "next": null,
  "items": [
    {
      "link":
"http://localhost:5000/api/questions/1101818/answers?answerId=1108349",
      "parentId": 1101818,
      "accepted": false,
      "submission": {
        "id": 1108349,
        "body": "<p>Did you look into Umbraco (<a
href=\"http://umbraco.org/\" rel=\"nofollow\">http://umbraco.org/</a>). Almost
out of the box .NET CMS. Really easy in use (for content manangers and devs).
And allot of options to extend it with webcontrols, .NET libraries and
xslt.</p>&#xA;",
        "creationDate": "2009-07-10T08:14:43",
        "score": 2,
        "soMemberId": 76031,
        "soMember": null
      }
    }, ... ]
  ]
}
```

Following the API documentation above, similar results can be retrieved for questions, comments, search results and tag filter. The interesting part is to be able to perform operations while being authenticated which allows users to have their own profile, history, annotations and marking. Let's create a user first. The image below highlights the bits in the request needed to create a user.

Creating a user

Following the above API documentation, we should make a **POST** request to the endpoint `/api/auth/users`. It expects a JSON body with username and password.

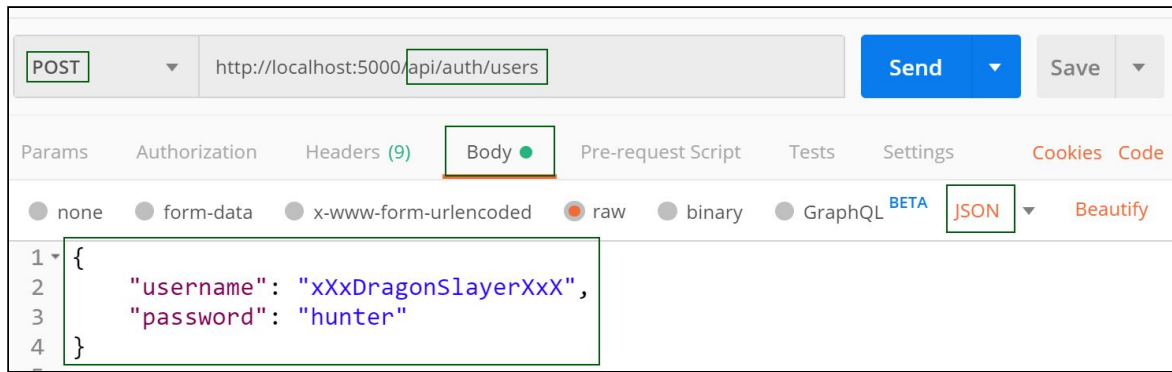


Figure 9: Creating a user with Postman

When we send this request, our backend creates a user and responds with status code **201 Created** and the username in the body.

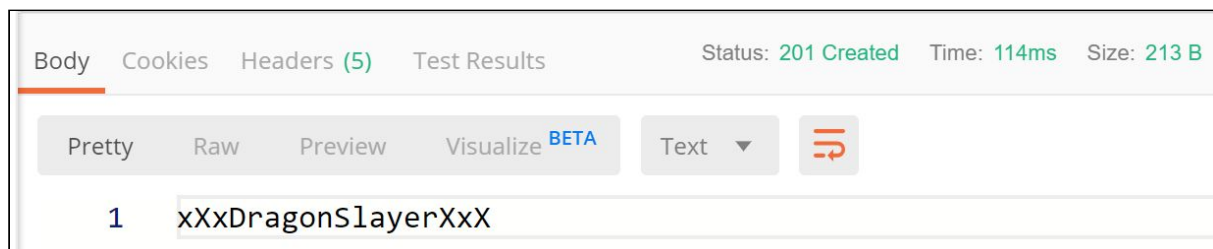


Figure 10: Response from the backend

And to be 100% confident that the user was actually created, we can have a look at our database to confirm. In this case the user was created with id 26 with its hashed password and salt.

	id	username	password	salt
1	25	tesitng	VrM00Jg/90jomfXyMrXC0ihpTRrRad...	eCLomEMRRDsRsv0WWYISsq7WxTh0iq/iYEHaVRb10Z1...
2	26	xXxDragonSlayerXxX	bqS7YAa75hjgvQ5QLbR03ZdQnQfyGES...	7aN0o1z1PtR6q0cMUeBF/1vqi4Kh051bH+1jejDKVv0...

Figure 11: User created in the database

Now that we have a user created, we can log in as that user and have our own personal space with history, annotations and markings. Authenticating into the application as a user with an authentication token is discussed in detail in the Security section.

User History

Since we have a user in the database, we can authenticate and create our personal space. The user stories talks about having search history, annotations and bookmarks. So let us do a search. That means a **GET** with query term **vim** for example, to the endpoint `/api/questions/query/vim`.

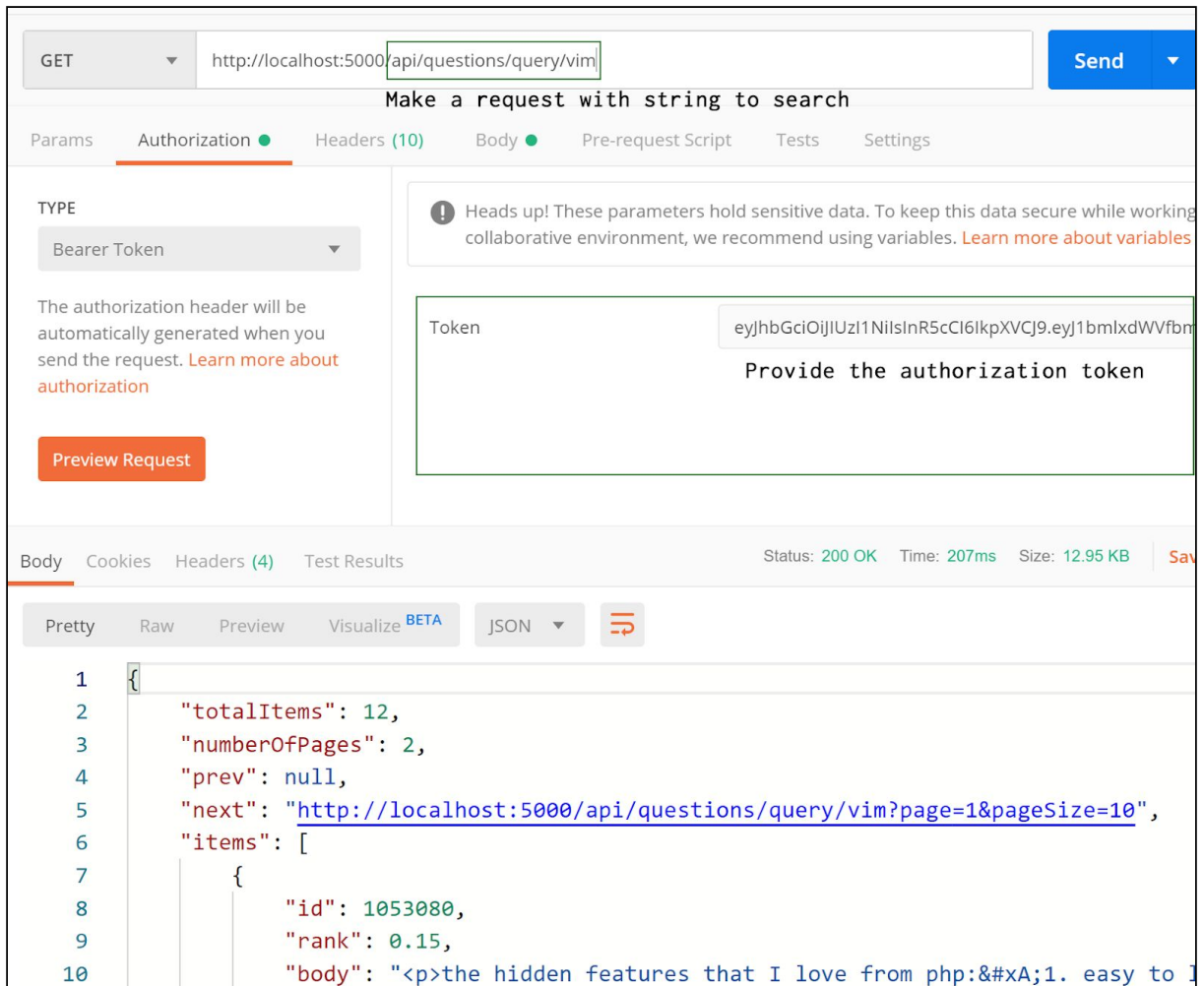


Figure 12: Making a request to search

Since we have stored procedures set up in the first section of the project, it should have updated our database table history with the search term **vim** and also have the authenticated user's id and the history's id in the user_history table. Let's check that out.

user			
	username	password	salt
1	26 xXxDragonSlayerXxX	bqS7YAa75hjgvQ5QLbR03ZdQnQfyGES...	7aN0olz1PtR6q0cMUeBF/1vqi4Kh051bH+1jejDKVv0...

history		
	id	search_term
1	108	vim

user_history	
	user_id
1	26

Figure 13: History for a user

Similarly, the user can perform CRUD operation on annotation in a post and bookmark of a post. Refer the API documentation above.