**Software specification requirements**

**Blog**

|  |  |  |
| --- | --- | --- |
| Name | Contact | Work |
| Zeljko Jelicic | zeljko.jelicic@eng.it | Spring Framework |
| Stefan Atanaskovic | stefan.atanaskovic@eng.it | Angular |
| Nikola Djekic | nikola.dekic@eng.it | Html, CSS, Bootstrap, Angular |
| Tamara Dragovic | tamara.dragovic@eng.it | Database, JPA |
| Milorad Dakic | milorad.dakic@eng.it | Database, JPA |

Table of Contents

[1.Introduction 1](#_Toc3841508)

[1.1 Purpose 1](#_Toc3841509)

[1.2 Scope 1](#_Toc3841510)

[1.3 Definitions, acronyms 1](#_Toc3841511)

[1.4 References 2](#_Toc3841512)

[1.5 Overview 2](#_Toc3841513)

[2.Overall description 2](#_Toc3841514)

[2.1 Product functions 2](#_Toc3841515)

[2.2 User characteristics 3](#_Toc3841516)

[2.3 Constraints 3](#_Toc3841517)

[2.4 UML Diagrams 4](#_Toc3841518)

[2.4.1 Use Case Diagram 4](#_Toc3841519)

[2.4.2 Activity Diagram 6](#_Toc3841520)

[2.4.3 Sequence Diagram 7](#_Toc3841521)

[2.4.4 Class diagram 8](#_Toc3841522)

[3. Database details 8](#_Toc3841524)

[3.1 Database connection 8](#_Toc3841525)

[3.2 Creating database user(admin) 9](#_Toc3841526)

[3.3 Database modeling in SQL Modeler 13](#_Toc3841528)

[3.4 Creating tables in SQL Developer 17](#_Toc3841529)

[3.5 JPA entities from database 18](#_Toc3841530)

[4. Specific requirements 21](#_Toc3841531)

[4.1 Interface Requirements 21](#_Toc3841532)

[4.1.1 Blogger interfaces 21](#_Toc3841533)

[4.1.2 Admin interfaces 22](#_Toc3841534)

[5. Functional requirement 25](#_Toc3841535)

[6. Communication between the different parts of the system 36](#_Toc3841536)

[7.Security 43](#_Toc3841537)

# 1.Introduction

This section gives a scope description and overview of everything included in this Software Specification Requirements. Also, the purpose for this document is described and application functionality with realized plan is provided.

## Purpose

The purpose of this SRS document is to provide a detailed overview of our software product, its parameters and goals. This document describes the project's target audience and its user interface and software requirements. It defines how our client, team can see the product and its functionality. Nonetheless, it helps any designer and developer to assist in software delivery lifecycle (SDLC) processes.

## Scope

This SRS is also aimed at specifying requirements of software to be developed but it can also be applied to assist in the selection of in-house and the standard can be used to create software requirements specifications directly or can be used as a model for defining organization or project specific standard. It does not identify any specific method, nomenclature or tool for preparing a Software Specification Requirements.

## Definitions, acronyms

|  |  |
| --- | --- |
| Term | Definition |
| Admin/Administrator | System administrator who is given specific permission for managing and controlling the system. |
| User | Someone that interact with “Blog” application. User creates, edits, deletes own blogs. |
| Web visitor | Someone that interact with “Blog” application. Someone who can view blogs and write comments. |

## 

## 1.4 References

*(1) IEEE Software Engineering Standards Committee, “IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications”, October 20, 1998.*

*(2) Burleson, K. D., Physical Database Design Using Oracle (Foundations of Database Design), 1st Edition, Auerebach Publications, 2006, p. 247.*

*(3) Oracle Academy courseware (Database Foundations and Database Design and Programming with SQL), 2015.*

*(4) Pro JPA 2, Mike Keith and Merrick Schnicariol, 2009.*

## 1.5 Overview

The remaining sections of this document provide a general description, including characteristics of the users of this project and the functional and data requirements of the product.  General description of the project is discussed in section 2 of this document.  Section 3 is about databases and gives the functional requirements, data requirements and constraints and assumptions made while designing the “Blog”.  Section 4 gives the admin viewpoint and user viewpoint of product with their abilities to interact with application.  Section 4 also gives the specific requirements of the product.  Section 5 describes functional requirements. Section 6 gives details about communication between two different parts of the system.

Section 7 is about security of our system.

# 2.Overall description

## 2.1 Product functions

There are several search criteria and it will be possible for the administrator of the system to manage the options for those criteria that have that.

The result of the search will be viewed either in a list view, depending on what criteria included in the search. User can search by: keyword, tag, date (from date to date), by topic, by domain, by user.

When user goes on HOME page he will see the list of blogs sorted by date from newest to oldest. Once user clicks on blog he can see all details about that blog (read, see number of views, leave a comment).

## 2.2 User characteristics

There are two types of users that interact with the system: users of the “Blog” application and administrator. Each of these two types of users have different use of the system so each of them have their own privileges.

**The administrator** interacts with the web application. He is managing the overall system so there is no incorrect information within it. The administrator can manage the information for each blog, creates new domain, add new user, approves comments on each blog. Admin has hidden URL path to log in and also has privilege to disable user's account.

**User of “Blog”** also interact with web application. He can create new blog, delete, edit and comment own blogs.

We also have web application **guest**-user can only read blogs and leave comments.

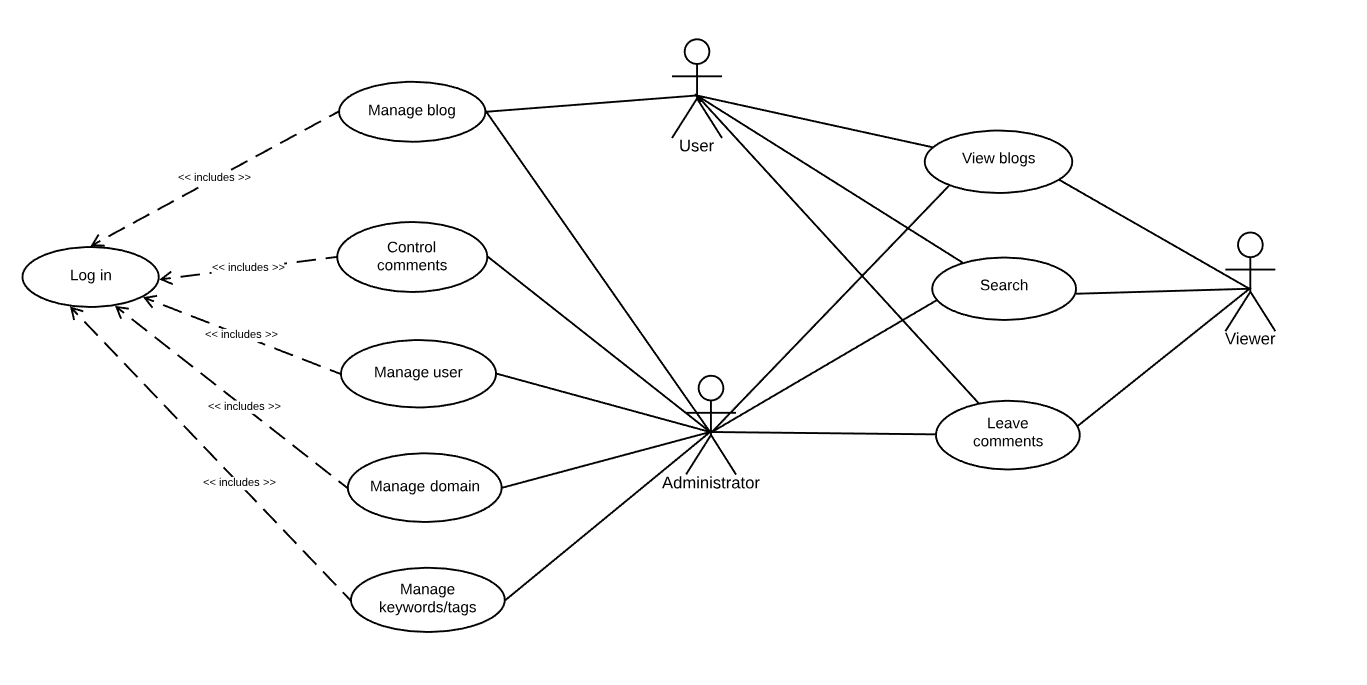
## 2.3 Constraints

The Internet connection is a constraint for the application. Web application will be constrained by the capacity of the database, since the application fetches data from the database over the Internet, it is crucial that there is an Internet connection for the application to function.

## 2.4 UML Diagrams

On the following pictures we will show you few examples of diagrams in our project. We choose the most important one`s for project functionality.

### 2.4.1 Use Case Diagram



*Picture 1. Use Case Diagram*

First one, use case diagram describes privileges that user and administrator have. We also have web user-viewer without any privileges but certain options when he/she navigates to our web site.

**Use Case Manage blog**

**Users:** User-blogger, Admin

**Flow:**  After user is logged in, he has the following options: list all blogs and create new blog. If user decides to list all blogs following options are: edit and delete blog. Other possibility is to create new blog where user is setting title, subtitle, text, URL for the blog picture, choose blog topic, keywords, tags and post this blog. If user is logged in like user-blogger, he has access only to his blogs, while admin has access to all blogs.

**Pre-condition:** user must be logged in.

**Use Case Manage user**

**User:** Administrator

**Flow:**  After user is logged in has the following options: create new user (blogger) or disable(block) user.

**Pre-condition:** user must be logged like admin

**Use Case Control comments**

**User:** Administrator

**Flow:**  After user is logged in has the following options: approve comment or disable comment. If comment is approved it will be displayed automatically bellow that blog other wise it will be removed.

**Pre-condition:** user must be logged like admin

**Use Case Manage domain**

**User:** Administrator

**Flow:**  After user is logged in, he can create domain. If domain is created it will be stored in database.

**Pre-condition:** user must be logged like admin

**Use Case Manage keywords/tags**

**User:** Administrator

**Flow:**  After user is logged in, he can create lists of keywords and tags.

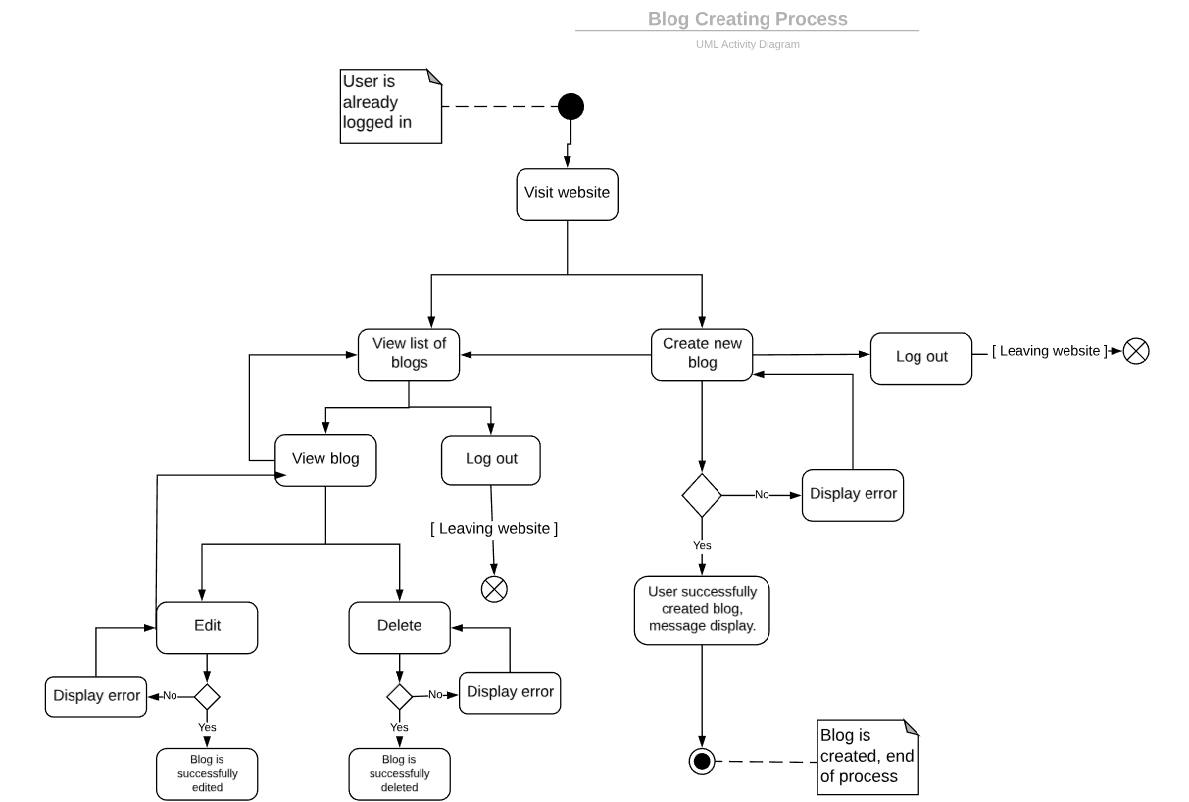
**Pre-condition:** user must be logged like admin

**Use Case Search, View blog, Leave comments**

**Users:** Administrator, user, viewer

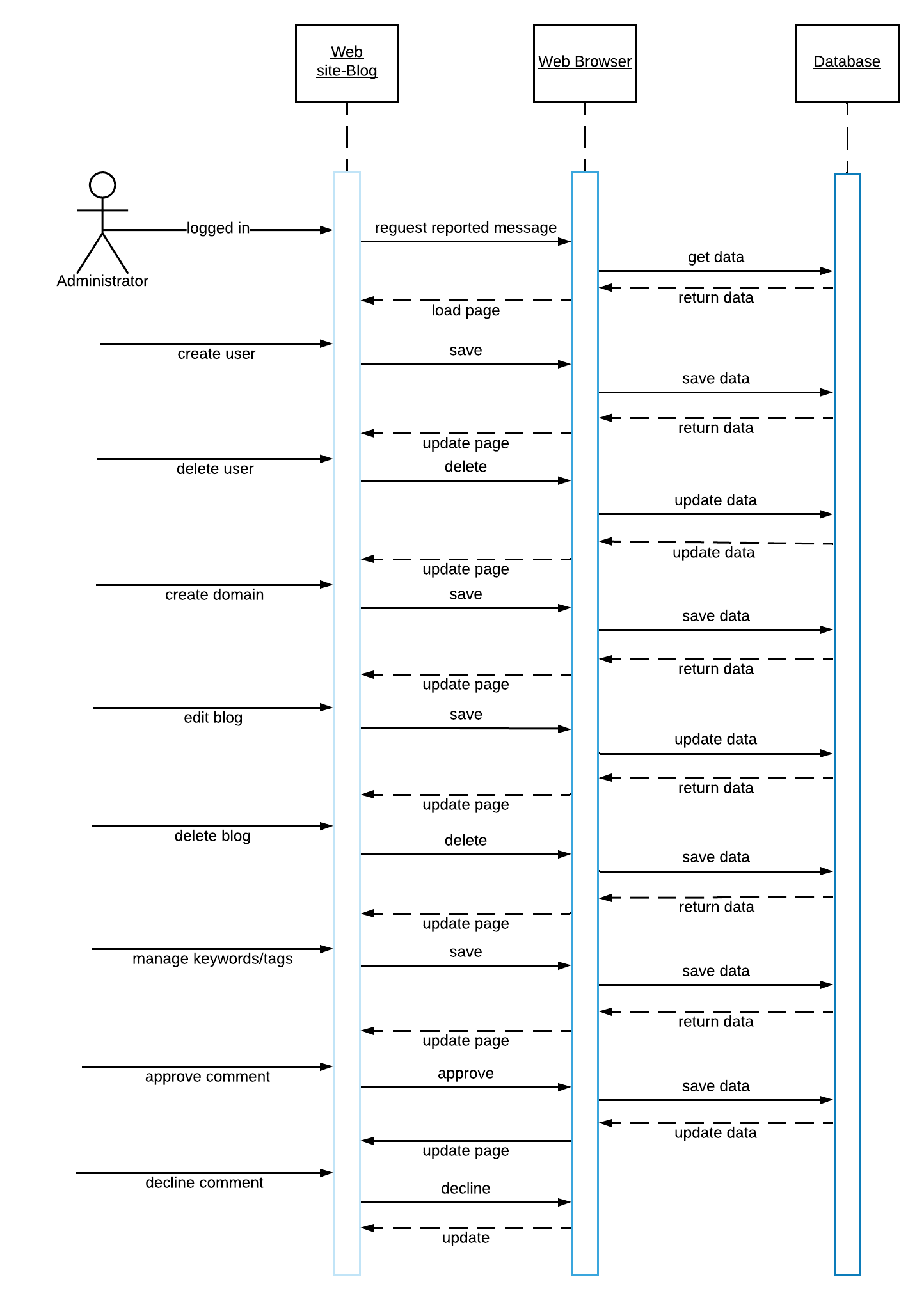
**Flow:**  After page is loaded users can search, view, and leave comments on Blog.

### 2.4.2 Activity Diagram



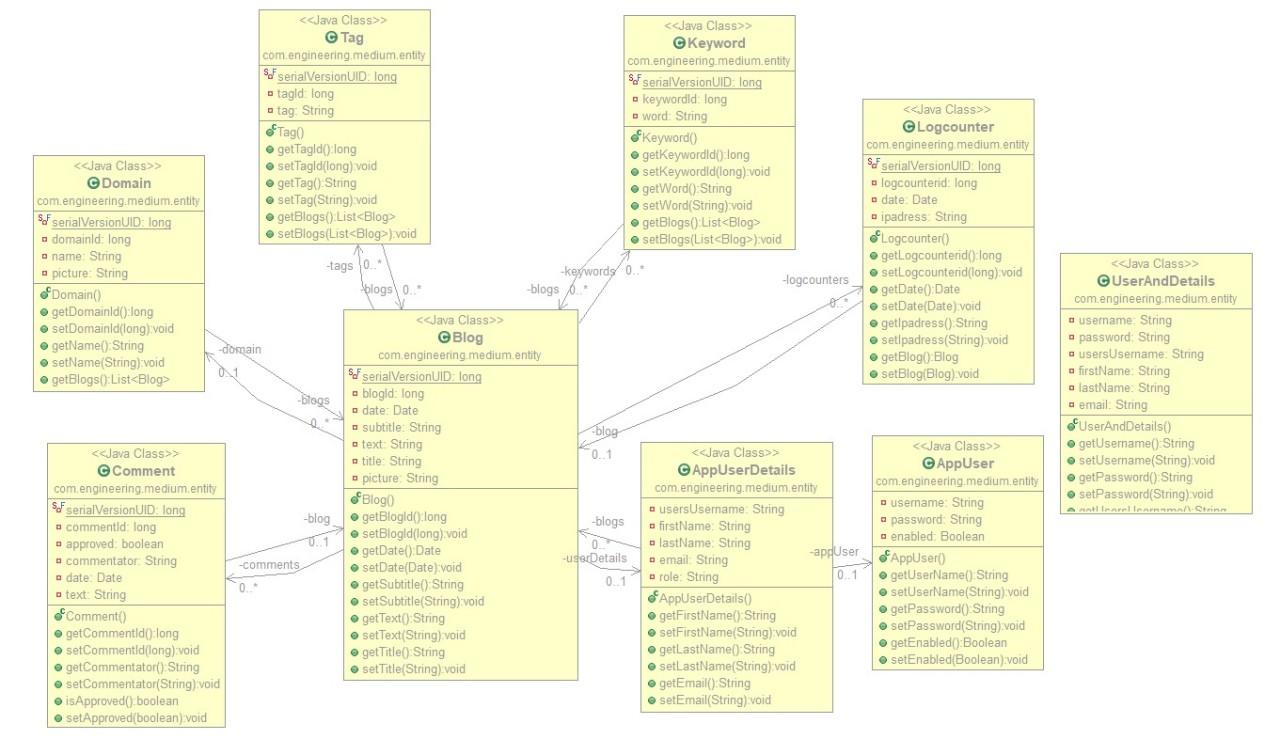
*Picture 2. Activity Diagram - Creating Blog process*

### 2.4.3 Sequence Diagram



*Picture 3. Sequence Diagram - Administrator*

### 2.4.4 Class diagram

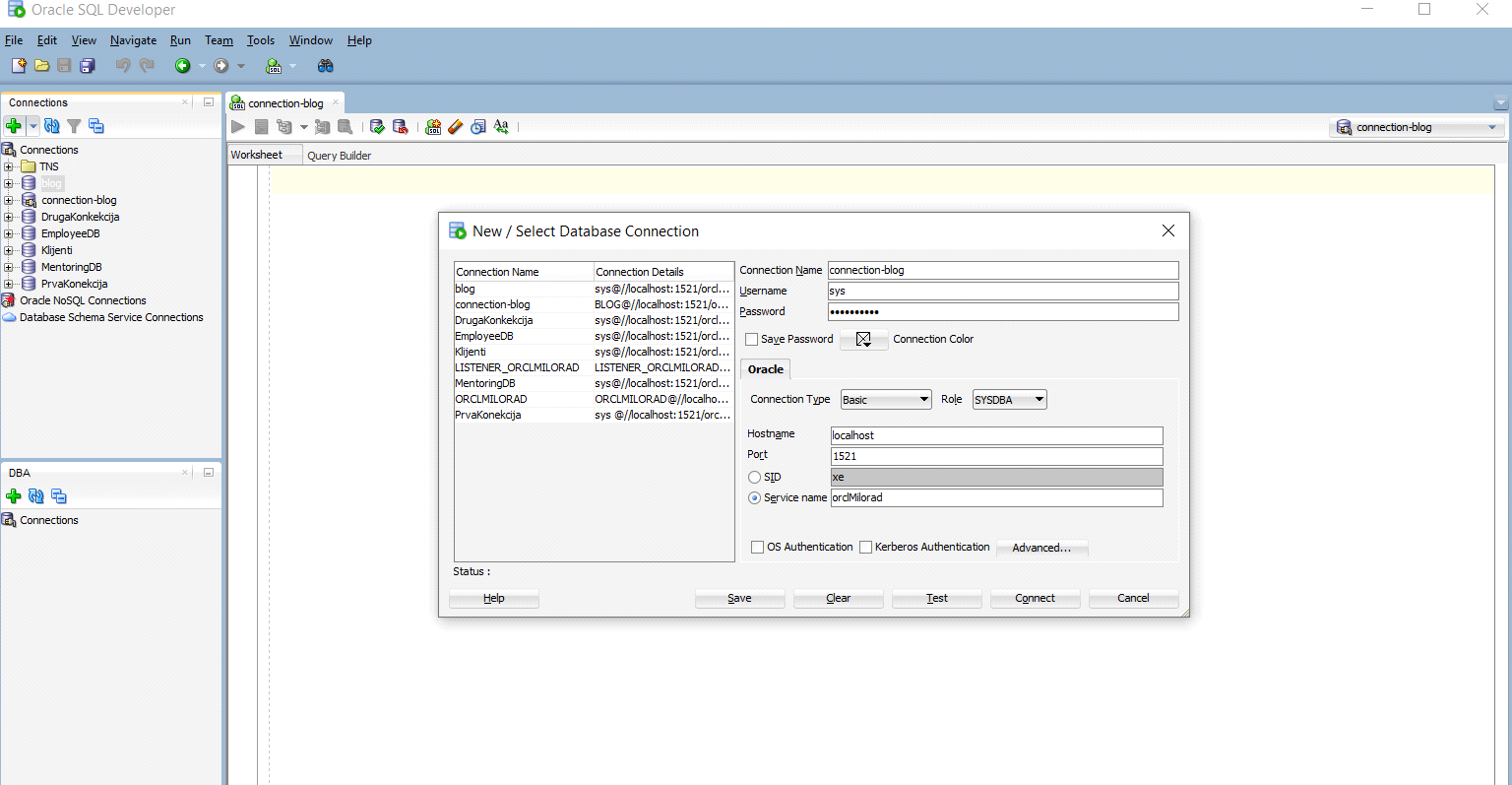


# Picture 4. class diagram

# 3. Database details

## 3.1 Database connection

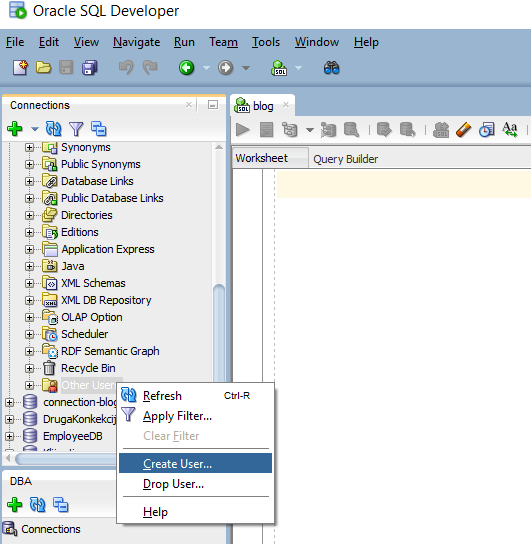
First thing we have to do is to open SQL developer and to create new connection. We will name it connection-blog, user name is sys and password is the one that you declared once SQL was installed on your computer, optionally you can save your password. Connection type is basic by default and you have to choose SYSDBA as a role in order to be able to have all system roles. We are connected via local host on port 1521 and SID is ‘xe’ by default. After that you can click on test connection to verify if everything is set in place and finally you can click on CONNECT button.



*Picture 5. Connection*

## 3.2 Creating database user(admin)

Once we are connected need to create a new user. In connection menu find a folder 'Other users' and right click on that folder and select option Create User.



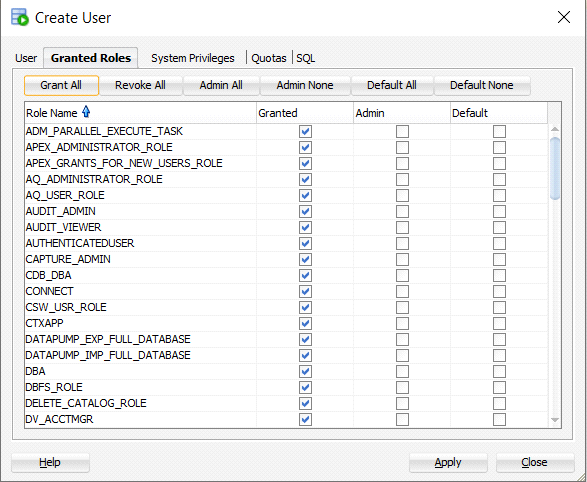
*Picture 6.Creating user*

After this we need to set up username, password and to choose SYSTEM as default Tablespace and TEMP as Temporary Tablespace.

## 

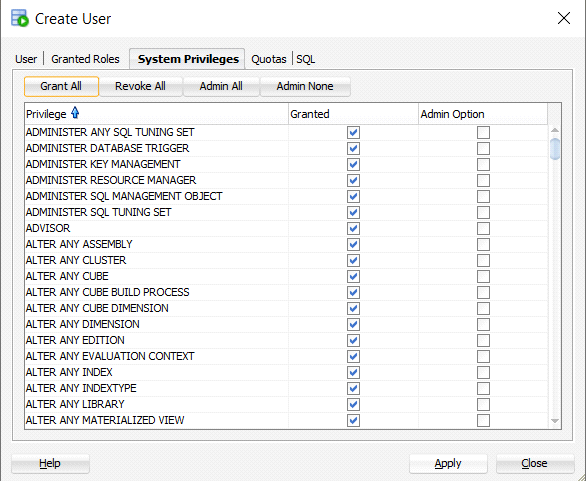
*Picture 7. Generating new user*

Next step is to click on option Granted Roles and choose Grant All.



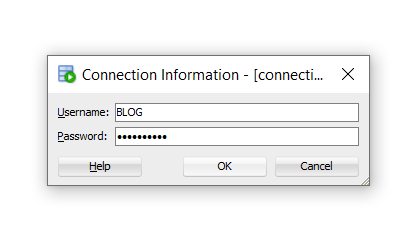
*Picture 8. User - roles*

After that you need to click on System Privileges and choose option Grant All. Once you have done with setting all parameters you can click on Apply button in bottom right corner of this window and you have successfully created a new user.



*Picture 9. User privileges*

Our next step is to connect as a new user. Choose username and password that you defined in previous steps and click OK to connect.



*Picture 10. User Log in*

## 3.3 Database modeling in SQL Modeler

First table we created was "USERS". It has following attributes: username(type of varchar 64), password(type of char 68, due to security reasons we decided to use password generator ‘BCrypt’ which is fixed size 68) and enabled (type of tinyint, by default is 1) this attribute means if user is not active he will not be erased but this field will be set to 0. Each password that we hash is going to have the same prefix ‘$2a$’, a unique salt and a unique hash. Salt represents random data that is used as an additional input to one-way function, it is used to safeguard passwords in storage. Hashing represents function that turns password into another string, meaning that is impossible to go the other way around.

Second table is "USERDETAILS" which has attributes: firstname(type of varchar 64), lastname(type of varchar 64), email(varchar 64), role(type of varchar 64,in this field we will set current role for a user for instance blogger, director, manager, employee), and this table has foreign key 'username' as a reference to table "USERS" field ‘username’.

Third table is "AUTHORITIES" which has attributes: authority(admin, blogger) and foreign key 'username' as a reference to table "USERS" field ‘username’.

Forth table is "BLOG" which has attributes: blog\_id(type of integer), title(type of varchar 255), subtitle(type of varchar 255), text(type of text), date(type of datetime), picture(type of varchar 255, which is path to picture in our project folder). This table has two foreign keys: The first one is domain\_id as reference to "DOMAIN" table field domain\_id. Second foreign key is username as reference to "USERDETAILS" table field username.

Fifth table is "DOMAIN" which has attributes: domain\_id(type of integer), name(type of varchar 64, which represents specific topic for our blogs).

Sixth table "COMMENTS" which has attributes: comment\_id(type of integer), text(type of text), date(type of datetime), approved(type of tinyint by default is 0, after admin approves comment it will be converted to 1), commentator(type of varchar 64, by default is "anonymous", this means that commentator don’t have to insert name it will be anonymous by default, but if he inserts, it will be saved into this field). This table has foreign key blog\_id as reference to "BLOG" table field blog\_id.

Seventh table "TAG" which has attributes: tag\_id(type of integer), tag(type of varchar 64). While user is creating blog, he defines tags for blog. Tag is used for search, sort and detailed blog description.

Eighth table is "KEYWORD" which has attributes: keyword\_id(type of integer), keyword(type of varchar 64). While user is creating blog, he defines keywords for blog. Keyword is important word that is repeating through the text, it is used for search and sort.

Ninth table "BLOG\_KEYWORDS" is binding table between "BLOG" and "KEYWORD".

Tenth table "BLOG\_TAGS" is binding table between "BLOG" and "TAG".

Eleventh table "LOGCOUNTER" which has attributes: logcounter\_id(type of integer), IPaddress(type of varchar 64, which is IP address from user that is visiting particular blog), date(type of datetime – show exact date and time when user saw particular blog). This table has foreign key blog\_id as reference to "BLOG" table field blog\_id.

**Relation between tables:**

One user may have one or more authority so relation is 1:N.

One user may have one details and relation between these two entities is 1:1.

One user may write more blogs so relation between them is 1:N optional.

One domain may contain more blogs. One blog must belong to only one domain.

Relation is 1:N.

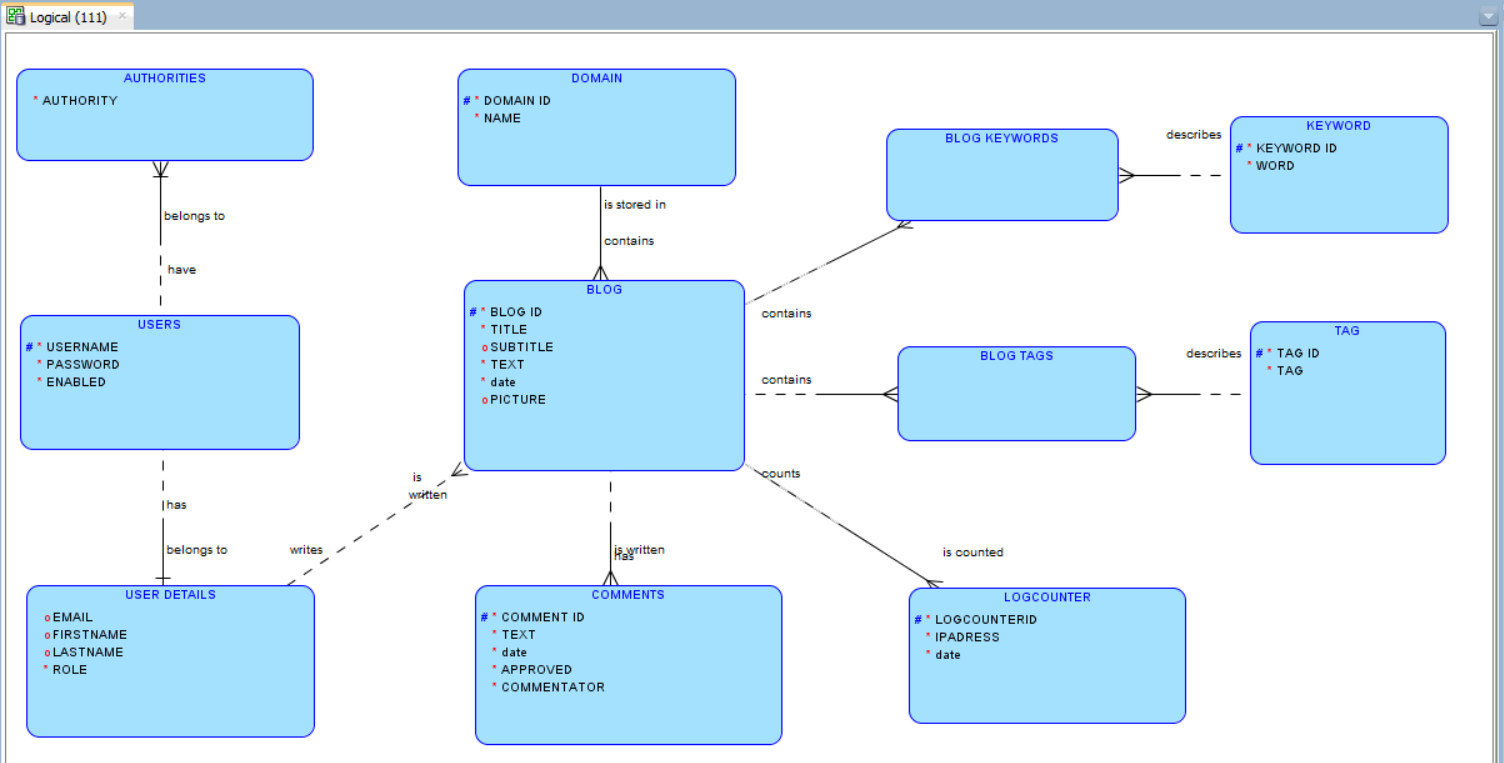
One blog may have more comments 1:N, optional.

One blog may have more logcounters(IPaddresses of blog visitors) 1:N, optional.

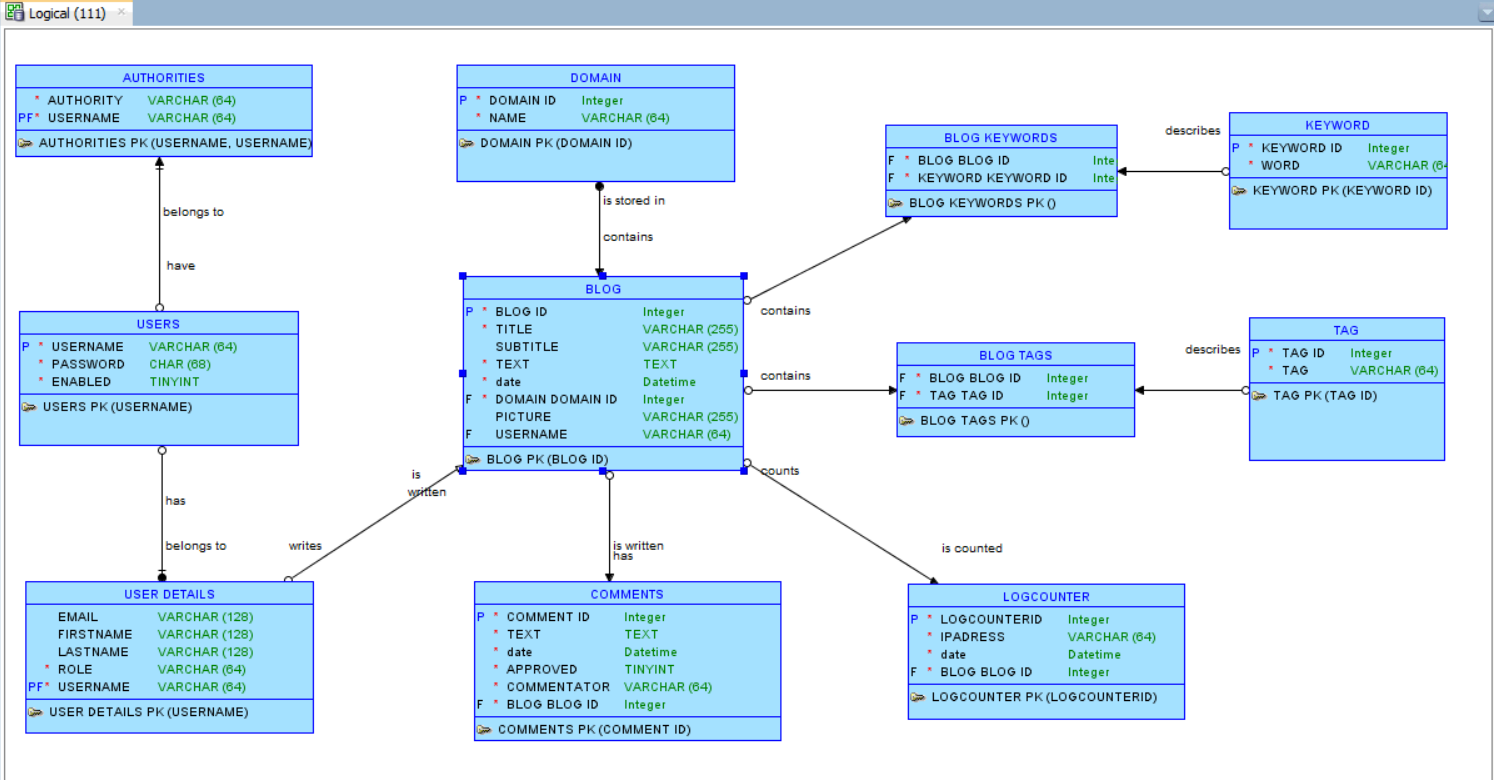
Relation between blog and tag is M:N, optional, it is defined in binding table “BLOG\_TAGS”. One blog may have more tags. “BLOG\_TAGS” table allows as to have a list of multiple tags and blogs.

Relation between keyword and blog is M:N, optional, it is defined in binding table “BLOG\_KEYWORDS”. One blog may have more keywords. “BLOG\_KEYWORDS” table allows as to have a list of multiple keywords and blogs.

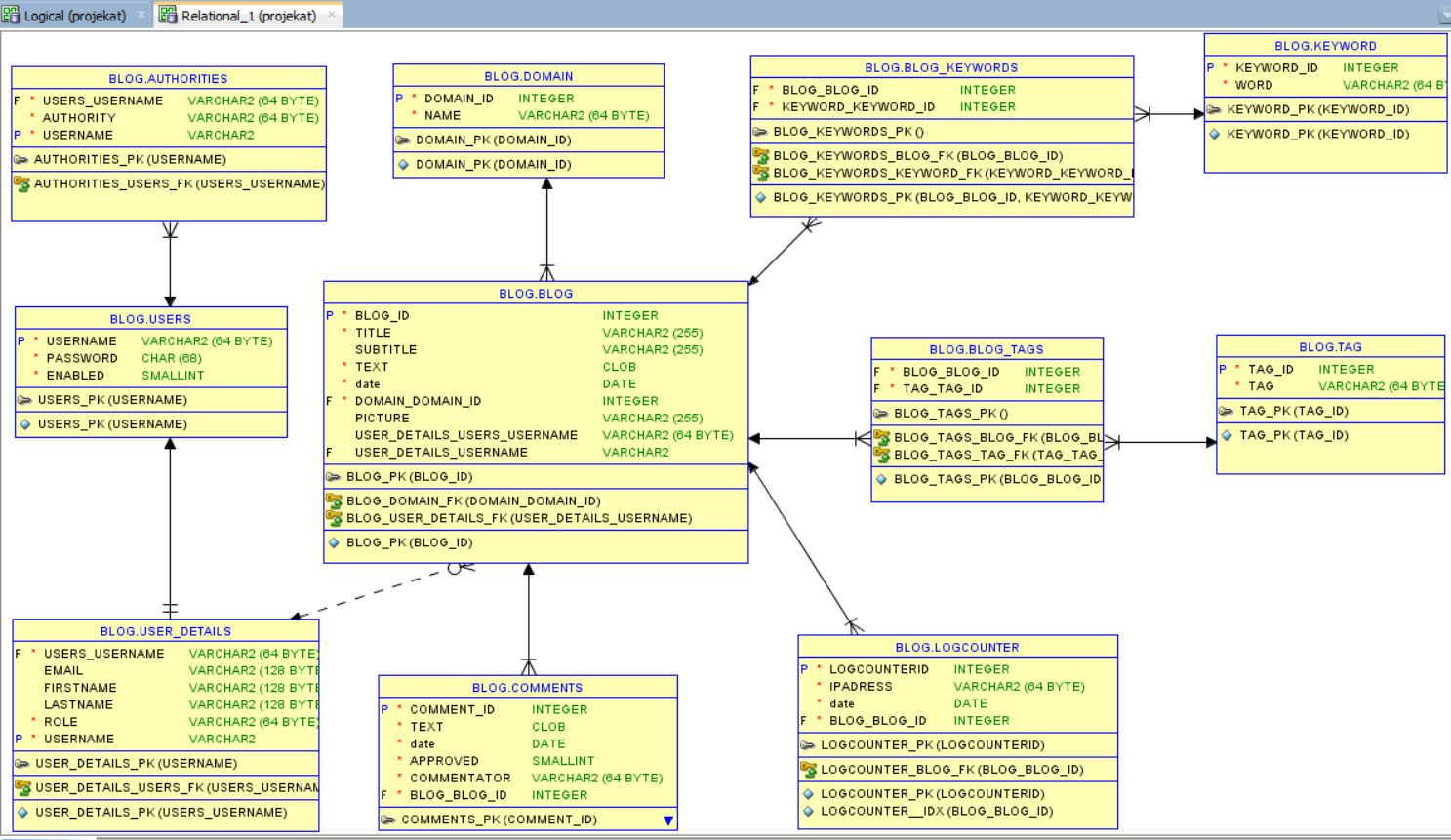
Final database model with all relations is defined in the following pictures



*Picture 11. Logical Model 1*

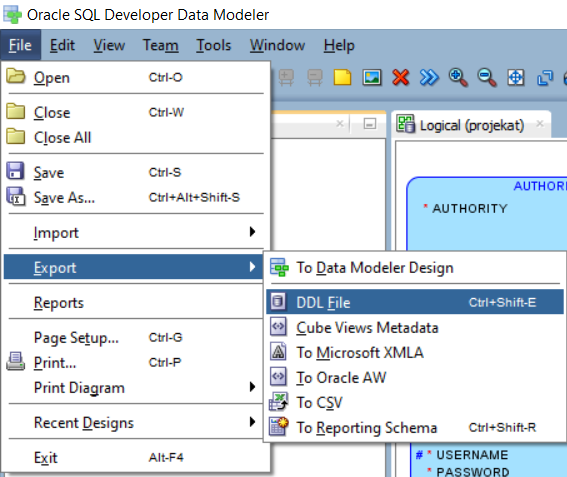


*Picture 12. Logical Model 2*



*Picture 13. Entity Relation Diagram (ERD)*

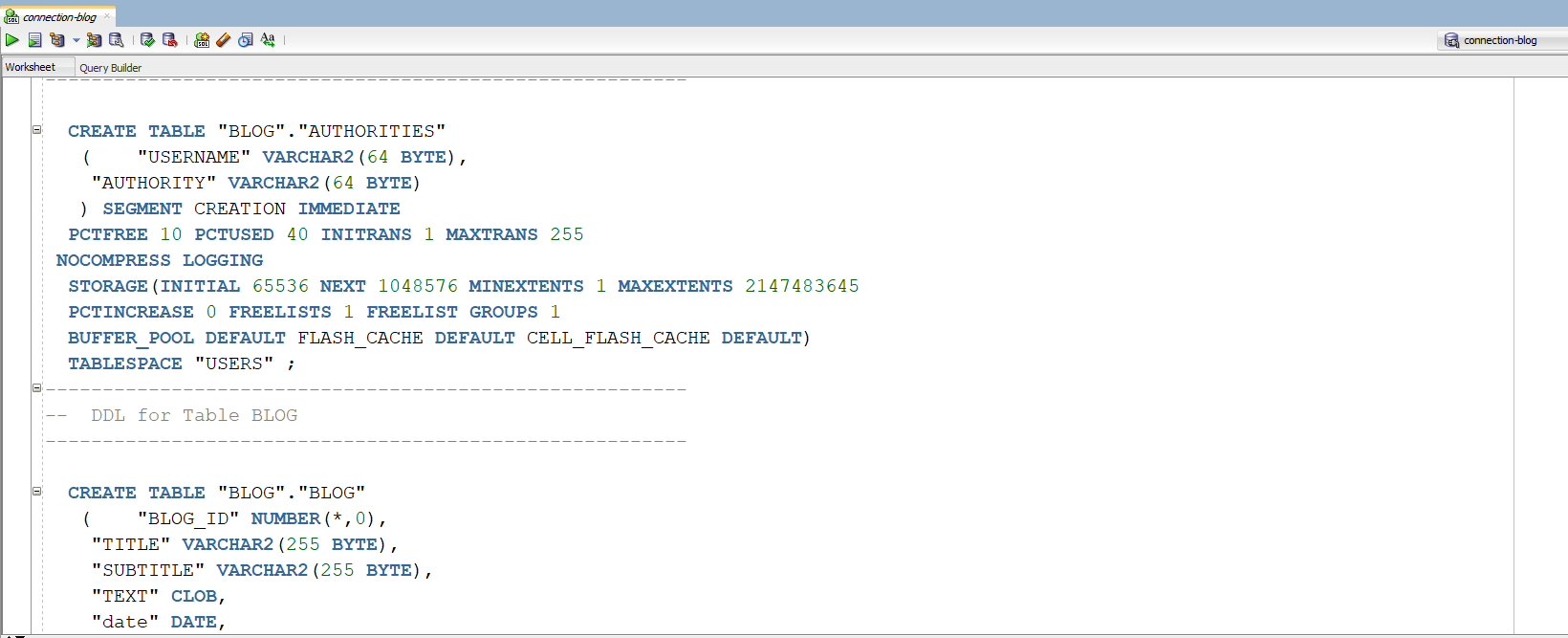
After model has been created I have to save design and to export it to DDL file which we I will later use to make tables. Next step is to choose option Generate and click OK button.



*Picture 14. Exporting to DDL file*

## 3.4 Creating tables in SQL Developer

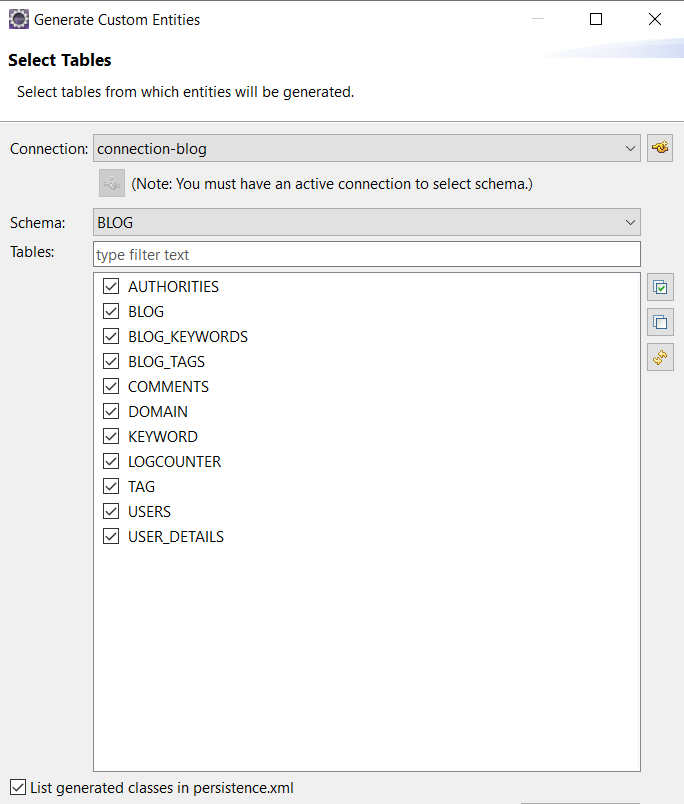
We need to create schema from DDL file. Find our file in project which is called medium.ddl, select all from that file and paste it into SQL developer and then click on run script button which is on top left corner. Sequences will be implemented in this file.(MINVAL =1, INCREMENT BY 1, cache =20).



*Picture 15. Creating schema*

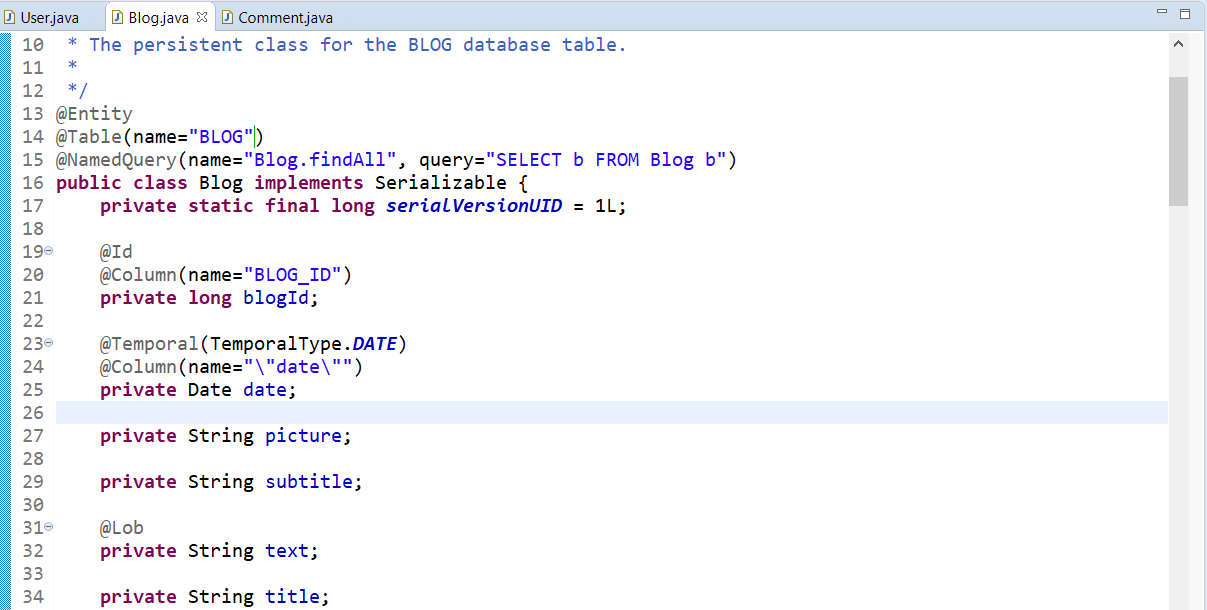
## 3.5 JPA entities from database

In Eclipse IDE I connected to our existing SQL connection-blog in order to create entities from our database. On project I just ran command New =>JPA Entities from Tables. Select all tables from our schema and click FINISH. These classes we used to create entities however Eclipse made few mistakes with relations between them and with data type so we had to correct them manually. Finall version of entities is described in section 5.2



*Picture 16. Generating entities from tables*

After all tables are generated example of one entity looks like picture below.

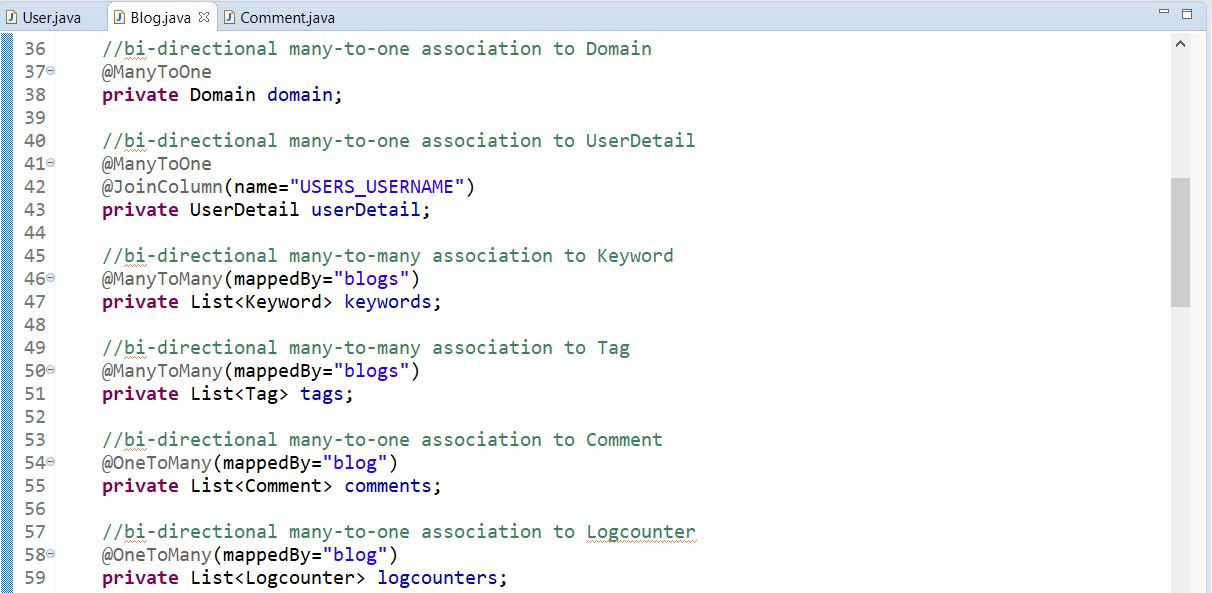


*Picture 17. Relations between entities 1*

@Entity annotation means this class is entity in our database.

@Table annotation describes the name of this entity in database.

@Lob annotation describes field text which is LARGE OBJECT.



*Picture 18. Relations between entities 2*

First relation is between Blog and Domain. It is ManyToOne which means that many blogs are stored in one domain.

Second is between Blog and UserDetails, ManyToOne which means that one user may write more Blogs.

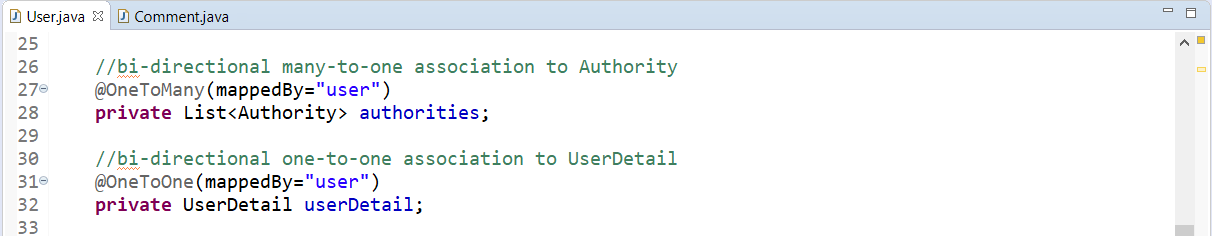
Third is between Blog and Keywords. Relation is ManyToMany because more blogs may have more keywords.

Forth is between Blog and BlogTag with binding table Keyword. This means that more Blog may have more tags.

Fifth is between Blog and Comments. Relation is OneToMany which means one blog contains more comments.

Sixth is between Blog and LogCounter. OneToMany relation. One Blog may be seen by more users that are counted with LogCounter.

On this picture I will describe other relations that we have between entities



*Picture 19. Relations between entities 3*

Relation between User and Authority is OneToMany. This means that one user may have more authorities(admin, blogger).

Relation between User and UserDetail is OneToOne. One user has one user\_details.

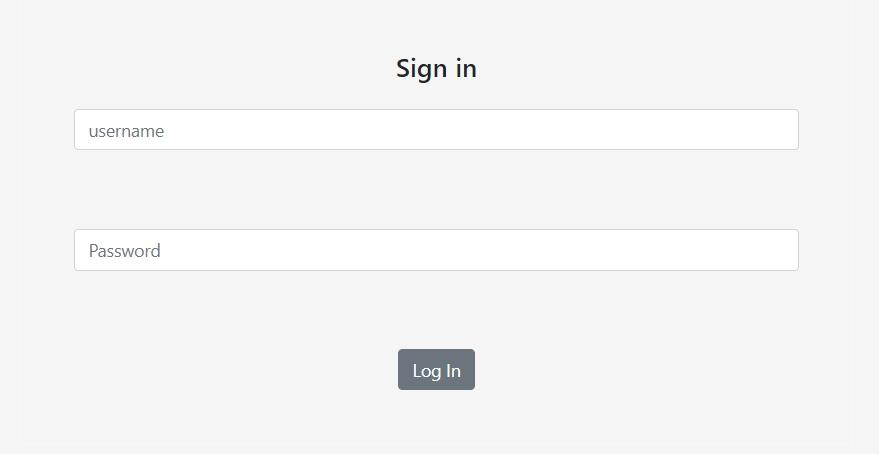
# 4. Specific requirements

## 4.1 Interface Requirements

## 

## 4.1.1 Blogger interfaces

Once user/blogger visits web site with hidden path (localhost:4200/login)he/she must log in with his credentials in order to use all of his/her privileges as user. Minimum three characters for the username and minimum eight characters for the password.



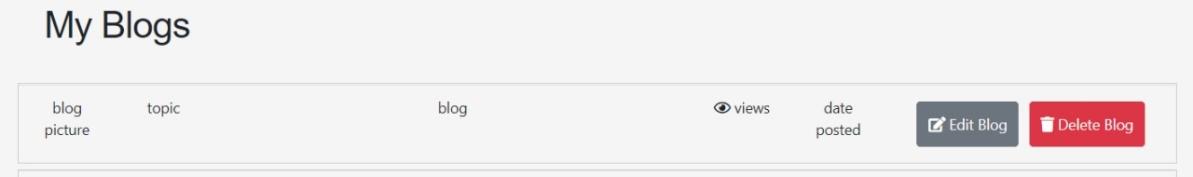
*Picture 20. Log in page on hidden path*

After user is logged in he/she has following choices: write new blog or to see the list of own blogs.

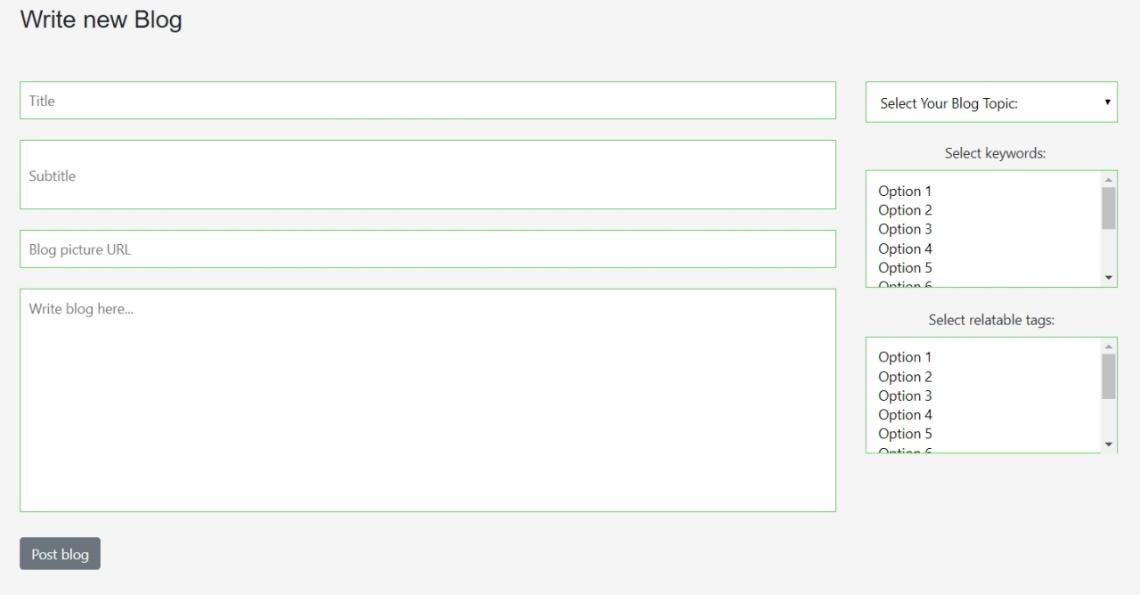


*Picture 21. User - first view*

If user selects list of blogs it will be displayed blog picture, topic/domain, blog text, number of people who viewed that blog and date. Also user can edit and delete blog.



*Picture 22. List of user’s blogs*

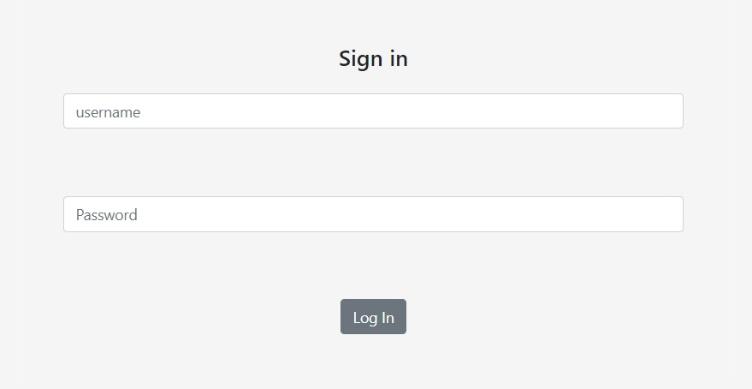


*Picture 23. User creating/editing new blog*

If user selects option Write new Blog he/she has following input fields: Title, subtitle, Blog picture URL(which is a path to a picture that is stored in database). Optional field are: select topic(domain), keywords, tags. After user/blogger finishes with creating blog next step is to click on Post blog button in order to make it public.

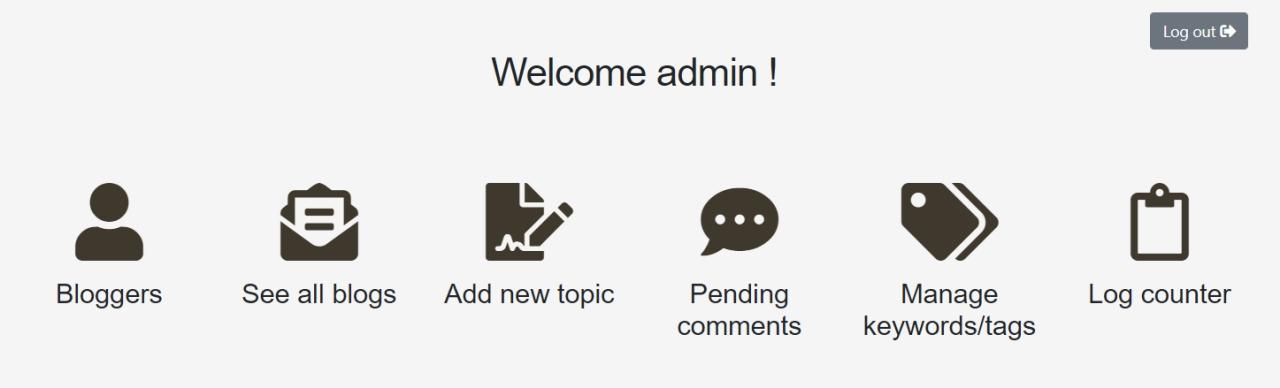
## 4.1.2 Admin interfaces

Admin has own hidden path of web site where he/she has to put credentials in order to log in and to use all privileges that admin has. Minimum three characters for the username and minimum eight characters for the password.



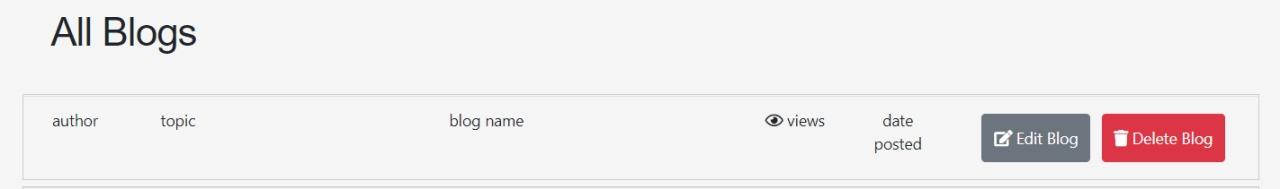
*Picture 24. Log in page on hidden path*

After being logged in admin has the following options: see all bloggers, see the list of all blogs, to add new topic(domain), approve/decline comment, manage keywords/tags and admin can also see how many times each blog has been viewed.



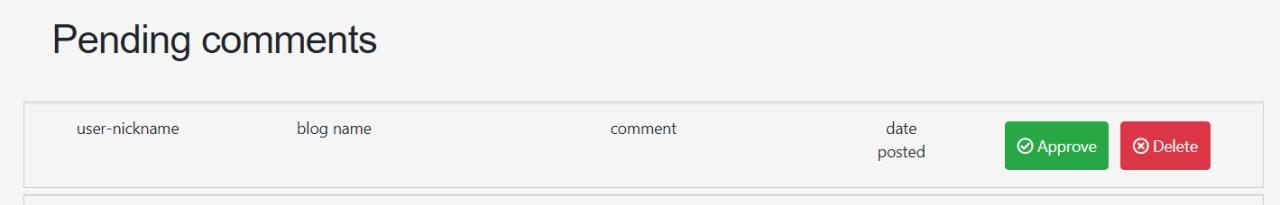
*Picture 25. Admin - first view*

On this page admin can see the list of all blogs which are contained of: Author, topic, blog title, number of people who viewed that blog, date when blog was posted, admin can also edit blog and delete blog from the list.



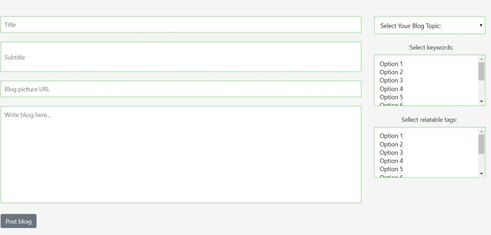
*Picture 26. List of all blogs*

In this section admin can see all comments that are still pending. Options that admin has is to approve it and make it visible on that blog or to delete(decline) and remove it from pending list.



*Picture 27. Admin - approving comments*

On this page admin has privilege to edit particular blog. The only section admin cannot change is user who created blog.



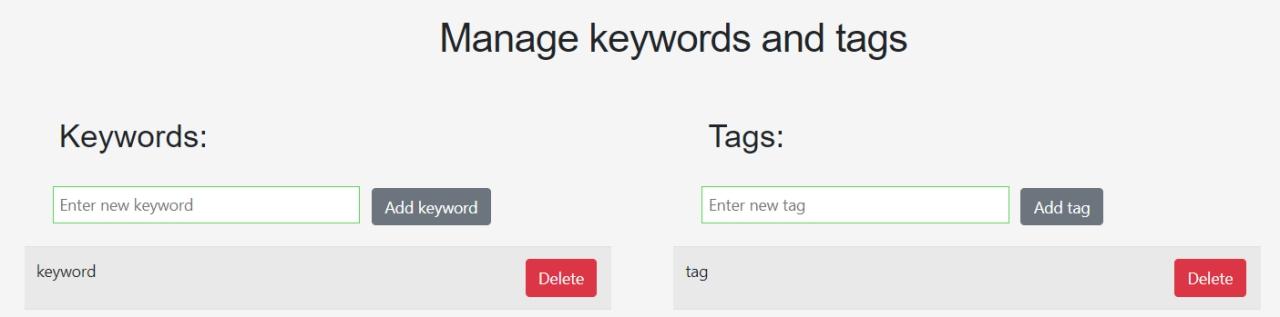
Picture 28. Admin – edit form

If admin clicks on log counter list will be displayed with: log ID, date when blog was viewed, IP address from person that viewed blog and blog title.



*Picture 29. Logcounter view*

Here admin can add or remove keywords and tags.



*Picture 30. Manage keywords/tags*

## 

# 5. Functional requirement

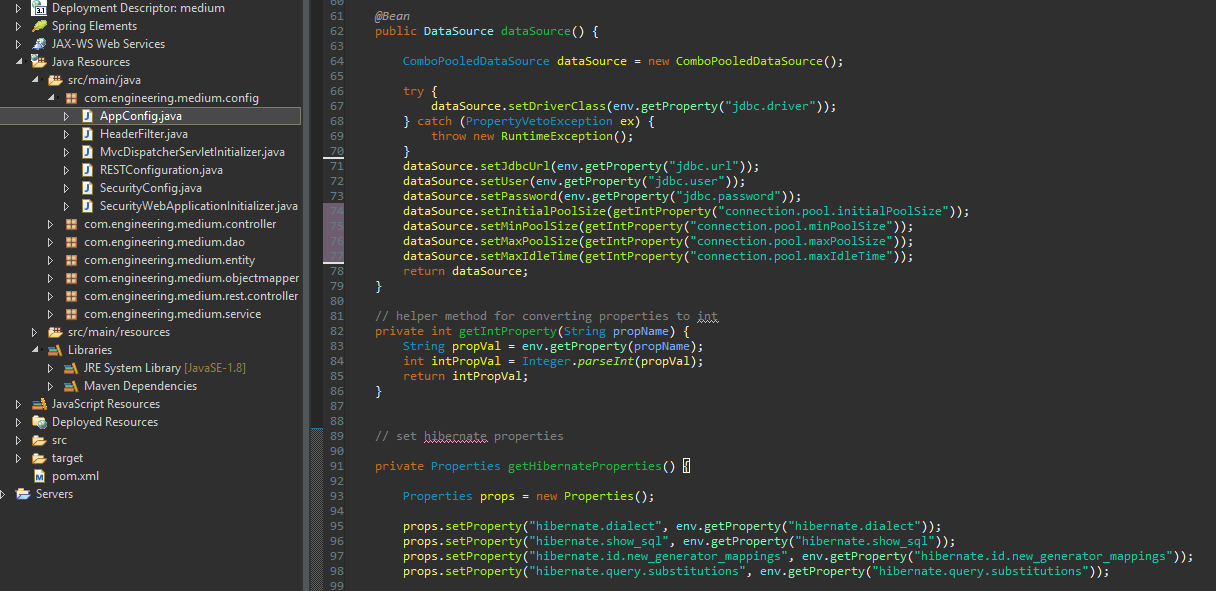
Application back-end is based on Java technology and it has been implemented using Spring framework which is running on Apache Tomcat application server. Considering that the front-end is based on different technology (Angular), communication between front and back is based on RESTful API through web services. Server is waiting for Http requests, converts JSON files to appropriate objects, depending on routes executes implemented logic and process data from database. After that processed data is being converted to JSON files and returned to the front-end.

5.1 App Configuration

Application configuration is completely based on Java, no XML files. For build tool it is used Maven, so all dependencies (library's) are contained in pom.xml in the root of application. In that way its insured that migration of application hosting requires minimal modifications.

All classes are organized in seven packages based on concerns of its content (config, controller, dao, service, entity, objectmapper and rest.controller).

**Config package** holds all configuration classes (AppConfig, HeaderFIlter, MvcDispatcherServletInitializer, RESTConfiguration, SecurityConfig and SecurityWebApplicationInitializer).

Core settings of application it is placed in **AppConfig** class (View Resolver, Data Source, Hibernate properties, Session Factory and Transaction Manager). 

*Picture 31. Part of the code from AppConfig.class*

**ViewResolver** bean settings do not have an important role because this applications main goal is handling Http request. At this moment when server is up and running we have index page shown at the root of localhost on port 8080 where Tomcat is running.

**DataSource** bean is loading all setting which are crucial for database connection (location, user, password... ). All those properties are placed in persistence-orcl.properties file which is placed in medium/src/main folder in application. In this way all setting for jdbc and hibernate are centralized so if we ever need to relocate our database or change RDBMS from Oracle Database to MySQL or any other, it requires only few lines of code to be changed.

**LocalSessionFactoryBean** is in charge for creating Hibernate Session Factory. Whenever its needed to communicate with DB Session Factory creates new session in which all queries are performed and automatically session is being closed. Location of all entities is read from same properties file.

**HibernateTransactionManager** binds a Hibernate Session from the specified factory to the thread, potentially allowing for one thread-bound Session per factory. It supports custom isolation levels, and timeouts that get applied as Hibernate transaction timeouts.

**HttpMessageConverters** bean configure the HttpMessageConverters to use for reading or writing to the body of the request or response. If no converters are added, a default list of converters is registered.

Also, we have override those methods **addResourceHandlers** and **addCorsMappings**. AddResourceHandlers adds handlers to serve static resources such as images, js, and, css files from specific locations under web application root, the classpath, and others. AddCorsMappings configure cross origin requests processing, we needed to allow certain Http methods which we are using later in project.

**HeaderFilter** classhave only one concern, to add 'Access-Control-Allow-Origin' header to every response header which is required by CORS policy. Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin.

**RestDispatcherServletInitializer** is used to to register a DispatcherServlet and use Java-based Spring configuration as opposed to the traditional web.xml-based approach.

We created an additional configuration class **RESTConfiguration** which holds Spring MVC View that renders JSON content by serializing the model for the current request using Jackson 2's ObjectMapper. By default, the entire contents of the model map (with the exception of framework-specific classes) will be encoded as JSON.

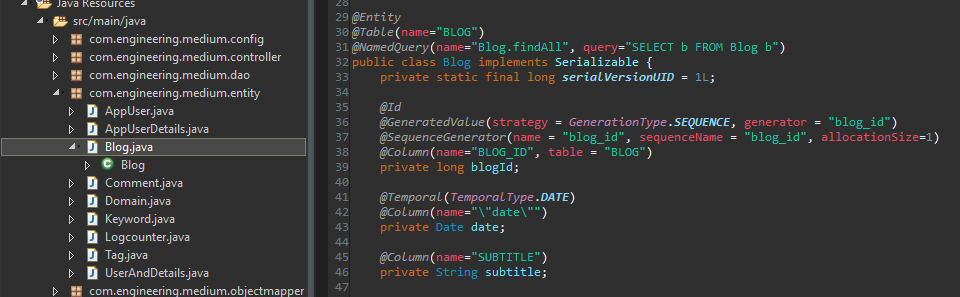
**SecurityWebApplicationInitializer** classregisters the DelegatingFilterProxy to use the springSecurityFilterChain before any other registered Filter. It will also register a ContextLoaderListener.

**SecurityConfig** provides a base class for creating a WebSecurityConfigurer instance. The implementation allows customization by overriding methods. Again, we have used DataSource as a factory for connections to the physical data source. An alternative to the DriverManager facility, a DataSource object is the preferred means of getting a connection. We will talk more about security in Security chapter.

5.2 Entities

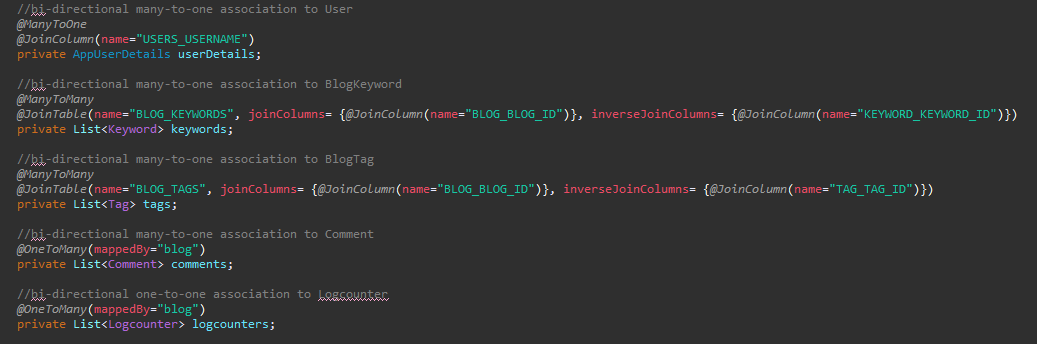
For every table in DB there is corresponding POJO class or better said Entity. All entity's are placed in the package which holds the same name. For every column in DB table there is corresponding property in the class. Relations between tables are represented as a references to the other class (table). In this project we are using Hibernate as Object/Relational Mapping (ORM) framework. In addition to its own "native" API, Hibernate is also an implementation of the Java Persistence API (JPA) specification.

Every class which represents table in DB holds *@Entity* and *@Table* annotation where we defining which table this class represents. Every property has annotation *@Column* where we are defining the name of the column in DB table. For unique identifiers *@Id* annotation is used. In case where Oracle DB is used it is necessary to define which Id generator we want to use. So in our case, we have created specific generator (sequence) for each Id in DB tables so now we are just telling Hibernate which one to use when creates Id.



Picture 32. Part of the code from Blog entity

Every class property which represents reference to some other class (table) have additional annotations which tells Hibernate how to act when object of that class needs to be persisted, deleted, read or updated. For example if we have one-to-many relationship the owner of the foreign key is at the “target” table, but we also have reference to that object at “owner” class. In that way we've insured bi-directional binding so we can get to all references using single object. Are job is to put appropriate annotation and Hibernate does the rest for us.



Picture 33. Example of Hibernate annotations

In case we have many-to-one relation we use two annotations: *@ManyToOne* and *@JoinColumn* where we define column name which holds foreign key. On the other side of relation we use *@OneToMany* annotation and just telling Hibernate which property of reference class to use for mapping using “mappedBy”.

In case of many-to-many relation in DB we are using join tables which consists only of composite primary keys. Those table aren't presented as classes. We have presented those relations just through annotations. With *@ManyToMany* we defined type of relation and using *@JoinTable* we have defined name of join table and then using “joinColumns” we are telling Hibernate which column represents “owner” Id and using “inverseJoinColumns” for the reference of the “target” we have created bi-directional many-to-many relation.

Entities **AppUser, AppUserDetails** and **UserAndDetails** are managed a little bit different than other. AppUser entity is managed by UserDetailsManager. Table User, coresponding to AppUser entity, and Authorities in DB are managed by UserDetailsManager and JdbcUserDetailsManager. Using Jdbc user management service, CRUD operations are provided for both users and authorities. So when new user is created at the same time authority for that user has been assigned. Relation between those entities is one-to-many, so one user can have multiple roles in authority table. You can find detailed information about these tables in chapter 3.3.

**AppUserDetails** is a class with detailed information about specific user (first name, last name, email... ). Relation between AppUser and AppUserDetails is one-to-one. Knowing that primary key of AppUser table is username property, usersUsername represents foreign key in AppUserDetails entity. There can no be multiple users with same username, username is unique on application level.

**UserAndDetails** just represents model which is used in rest controller for receiving data from register new user form. It collects all data for new user, user data and user details. After that two objects are created, AppUser and AppUserDetails, with collected information and they are being stored in DB.

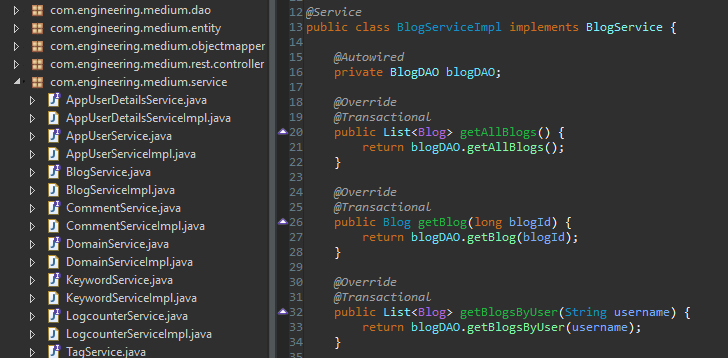
5.3 DAO and Services

For every entity there is corresponding DAO interface. DAO interface contains method signatures for all CRUD and all other operations which are available for specific entity. Every DAO interface has its implementation annotated with *@Repository (@Component* specification) where Hibernate query's are executed. This is the place where Hibernate SessionFactory instance is injected and sessions are managed for each query. After creating session operations on DB are available. Query's are executed, data is collected or modified and results are returned. After that Hibernate automatically closes session. No query's is possible to happen out of the session so usual problem in this part of application flow is when lazy data loading is activated and you are trying to get those data. You'll be facing LazyInitializationException. In parts of execution, where we needed those lazy load data, we have used Hibernate static method Hibernate.initialize() to force Hibernate to load those data inside the existing session.



Picture 34. Example of using Hibernate.intialize() method

Every entity has also a Service interface which contains same method signatures as its DAO interface. In service implementation, annotated with *@Service (@Component* specification), we have injected DAOInterface, that way we achieved that communication between user (controller) is not being executed directly and that every query has been isolated in single transaction. Also, we accomplished loose coupling between objects based on IoC concepts and higher level of security because service layer has no relation to the DB.



Picture 35. Part of code from Blog Service example

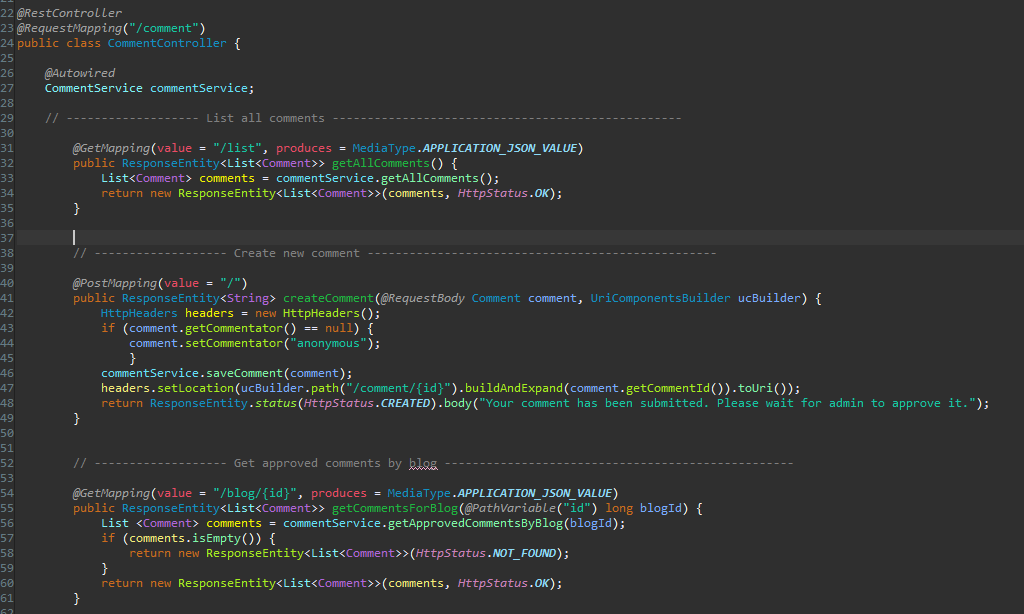
5.4 Object mapper

**Objectmapper** packagecontains only one class which extends ObjectMapper class. This class provides functionality for reading and writing JSON, either to and from basic POJOs, as well as related functionality for performing conversions. It is also highly customizable to work both with different styles of JSON content.

5.5 Controller

**Controller** packagecontains all controller classes. Each controller represents a collection of API endpoints for data manipulation or collecting data. Each entity has its own controller that contains several endpoints. Based on URI, HTTP method or both different methods in controller is being triggered. Each of them has different tasks and returns different information to the client. All of them are returning some kind of HTTP response status codes that indicates whether a specific HTTP request has been successfully completed. Every controller class is annotated with *@RestController*. *@RestController* annotation combines *@Controller* and *@ResponseBody* - which eliminates the need to annotate every request handling method of the controller class with the *@ResponseBody* annotation. Every request handling method of the controller class automatically serializes return objects into *HttpResponse*. Every controller is also annotated with *@RequestMapping* containing path to this specific controller and every method have annotation which defines what kind of request this method is expecting (@GetMapping, @PostMapping...)

Service interface reference of each entity is injected in corresponding controller and it is used to get DAO layer and to access DB. Every controller method returns object of ResponseEntity which represents the whole HTTP response: status code, headers, and body. Because of it, we can use it to fully configure the HTTP response. ResponseEntity is a generic type so we can use any type as the response body. Based on definition of controllers method input parameter we are defining what we are expecting in particular HTTP request. In some cases it is an object and sometimes we just need URI parameter, such as Id of some object. In both cases we are binding that request content to appropriate data type so we can use it latter in our method. When we expect some URI parameter we use *@GetMapping* combined with *@PathVariable* and data type of variable that we are binding to. In other case, expecting POST request we use *@PostMapping* combined *@RequestBody* and we are binding body of that request to the expected object. The body of the request is passed through an HttpMessageConverter to resolve the method argument depending on the content type of the request. We also have methods without input parameters, those are usually methods which returns complete list of objects (whole DB table content).



Picture 36. One of the controllers with different types of methods

**RegistrationController** is in charge for registering new user and it is mainly driven by Spring Security. He has two beans injected: UserDetailsManager and AppUserDetailsService. UserDetailsManager is a part of Spring Security and provides the ability to create new users and update existing ones. AppUserDetailsService collects additional users information and saves it to user\_details table. When data collected from Register New User form ends at this API endpoint its been converted to UserAndDetails object, once we mentioned earlier. This object holds data about AppUser (username, password) and AppUserDetails with additional information (first name, last name, email... ). After that we check if user with same username already exists in DB, if it does we return appropriate HTTP response with message. If everything pass well we encode password with Springs PasswordEncoder using BCRYPT password algorithm and assign role of EMPLOYEE to a new user. In our application every new user is a blogger with role of employee, so if we want to set new user as admin it is necessary to make an additional input in AUTHORITIES table with same USERNAME and AUTHORITY should be ROLE\_ADMIN. Now we create new Spring Security User object with collected data and with Springs UserDetailsManager we save new user into DB. As we said Spring automatically saves data in both tables, USERS and AUTHORITY. All details about new user are also saved in our DB using custom AppUserDetails entity that we have created earlier.

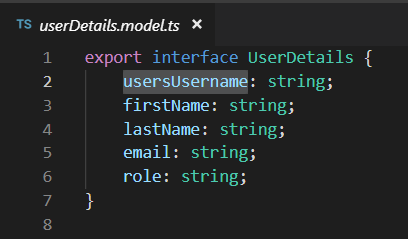
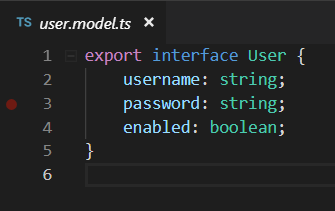
**RestLoginController** is our custom made controller used for login and logout. Main credentials check is done by Spring Security and this controller is just represents entry point for login to a back office. If credentials check pass well, our controller collects that users details and sends it back in HTML response body. After that in clients local storage all details are being temporarily saved and used until session ends or local storage is cleared on demand. You can find more details about application security in Security chapter.

# 6. Communication between the different parts of the system

Application frontend is based in Angular version 7. The angular application requests data from the backend(made with the Spring Framework) via the HttpClientModule, then waits for the json data as the response. Afterword’s the response is transformed from the json to actual JavaScript objects, based on data models inside the app and the data is used to fill up the templates of angular’s components and display it to the user.

**Data Models**

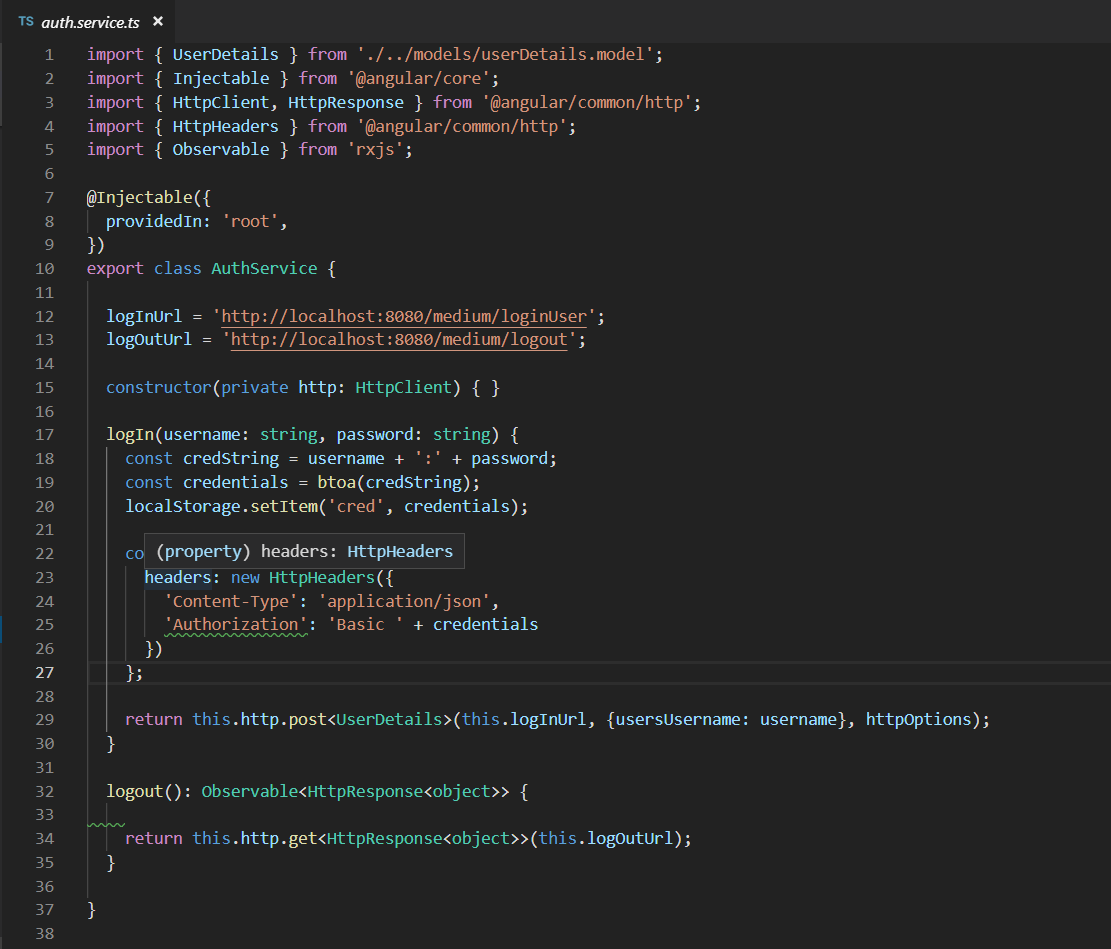
The data models are written as typescript interfaces. The are used to package the JSON data into variables of certain types to make it easier to manipulate and display them to the user.



Picture 37. Models

**Services**

The core functionality of the all the services in this project is provided by the HttpClientModule. Every service has it injected as a dependency, its used to communicate with the back end. Different endpoints supply the services with different JSON data. Some endpoints return actual objects like blogs or comments and others only return the HTTP status(for example delete or post comments), and the based on the status returned the user will receive different feedback. Every service is highly specialized for a certain type of data. For example the blogs service handles all the requests to the backend endpoints that are related to data about blogs and nothing else. It can perform all the CRUD operations with the blogs, get all the blogs by a specific keyword or by a date and so on. The same goes for the most of the other services in the application like the comments, domain tags and user service. The services also fill end clear the local storage with useful data and send authentication parameters to the backend.

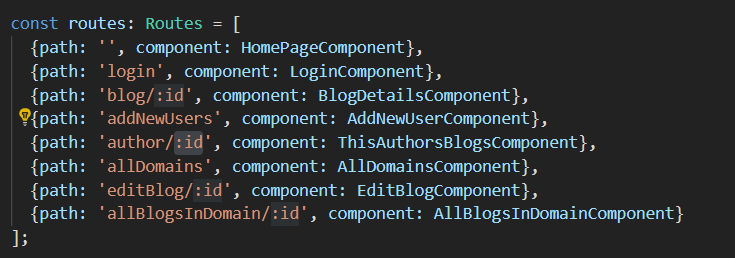


Picture 38. Services

**Routing**

This application has three separate routing modules. One main and two that are used for sub routing: one in the admin options component and the other in the blogger options component.

The main routing module(app-routing.module.ts) is pretty straightforward. It’s the most basic routing module, it just matches specific routes to specific components, some components take parameters and its router outlet is in the app components template.



Picture 39. Routes

The other two routing modules perform the same role, they belong to a parent component(admin options and blogger options) and inside it they render a different child component based on the route selected.

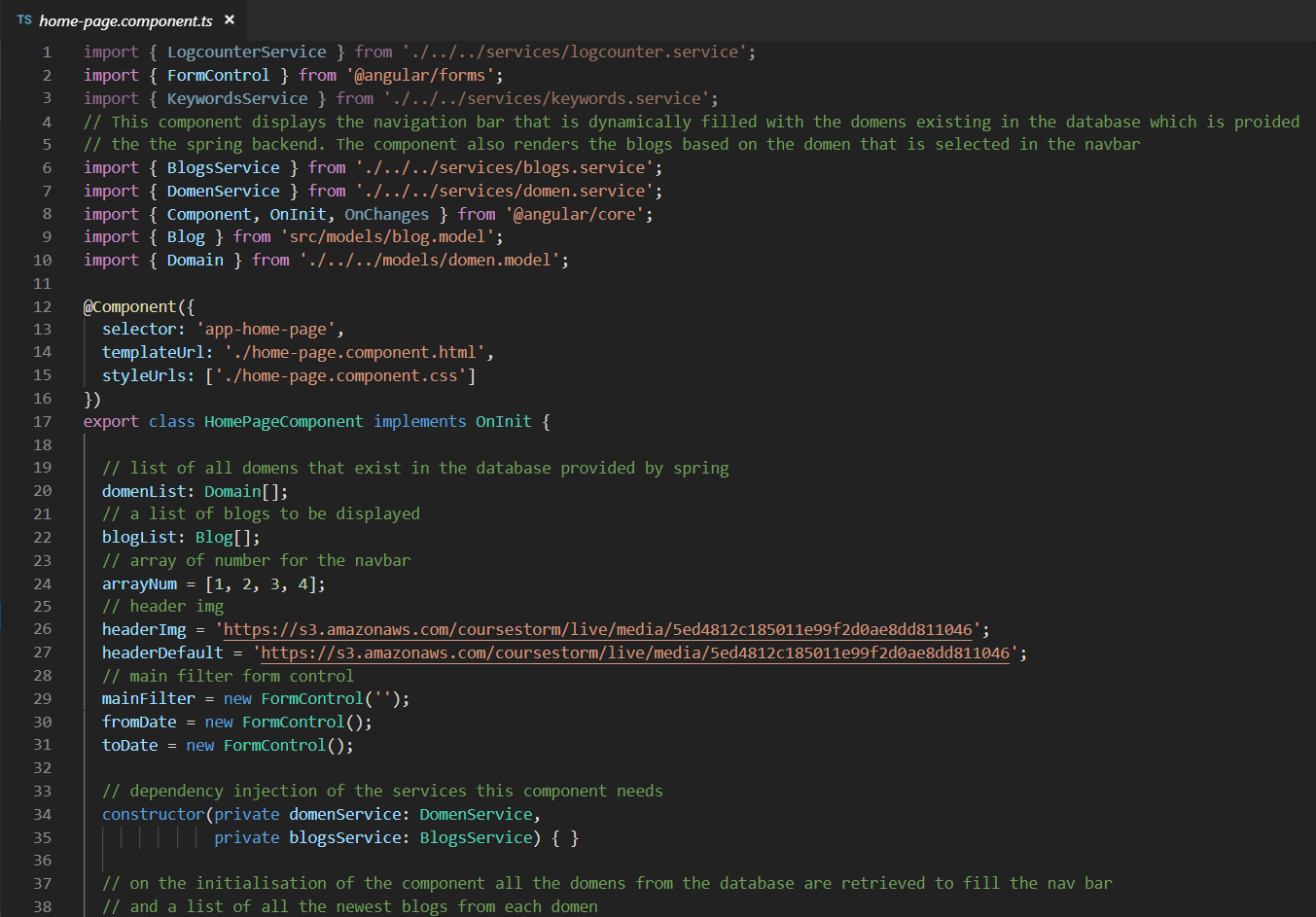
**Components**

The components use the services to get the data and then display it with the use of angular’s directives and bootstrap’s CSS framework.

**Homepage Component**

The homepage component uses the domain and blogs services. The domain service is used to fill out the navigation bar in the header part of the component and the blog service is used to fill out the rest of the page with blogs. The blogs can be searched by domain or by date or by keywords. The header navigation is filled with a home option, a short list of possible domains and a more option.

The Homepage option will list the newest blog from each domain, any domain will list all the blogs from that domain and the more option loads a component that displays all the domains available in the application.



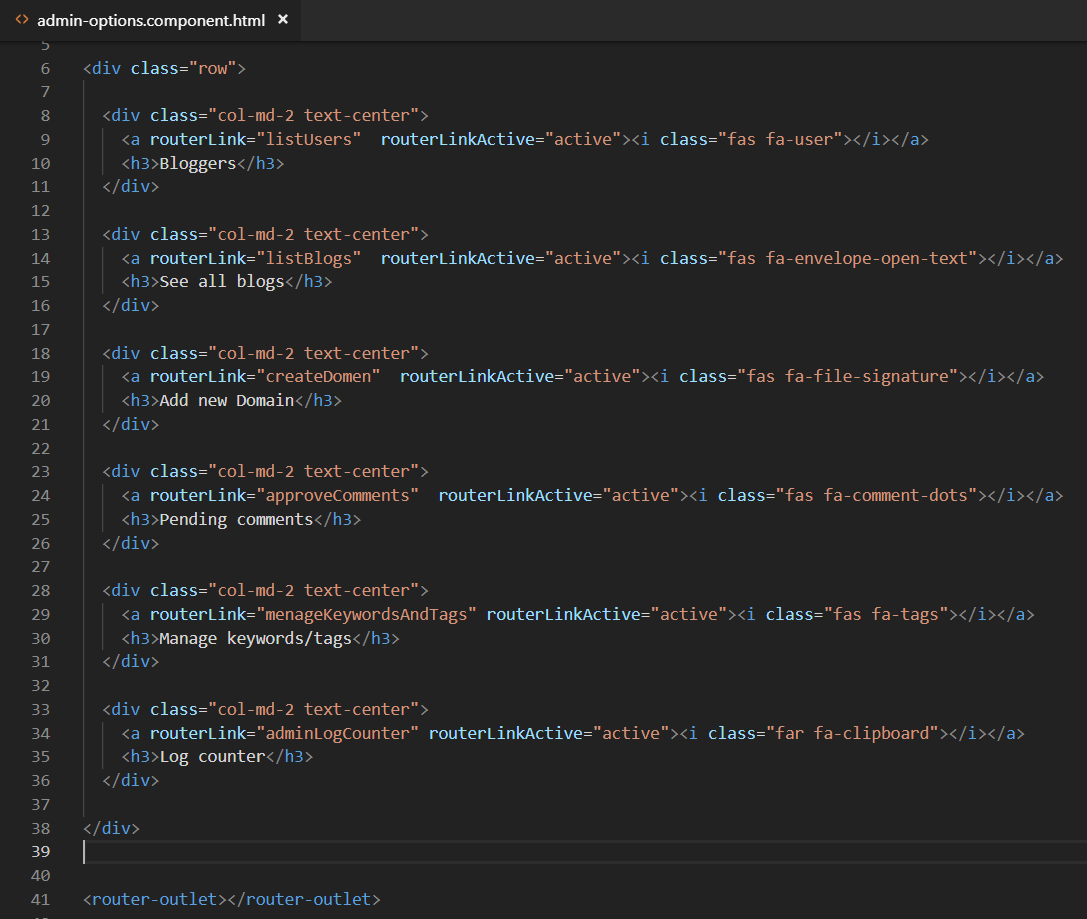
Picture 40. Home Page Component

**Admin Options Component**

Admin options component as mentioned previously has its own routing module, it will render the child components inside it while leaving the top menu for easy navigation through the options.

To get be able to use the admin options component you have to be logged in, if you are not you wont be able to use any of the options in the menu. After you log in you can gat a list of all bloggers where you can remove a blogger, edit a existing one or add a new one. You can also list all the blogs written and update them or delete them. You can add a new domain to the application, add or remove keywords or tags, approve or delete pending comments and see the list of user activity on the blog.

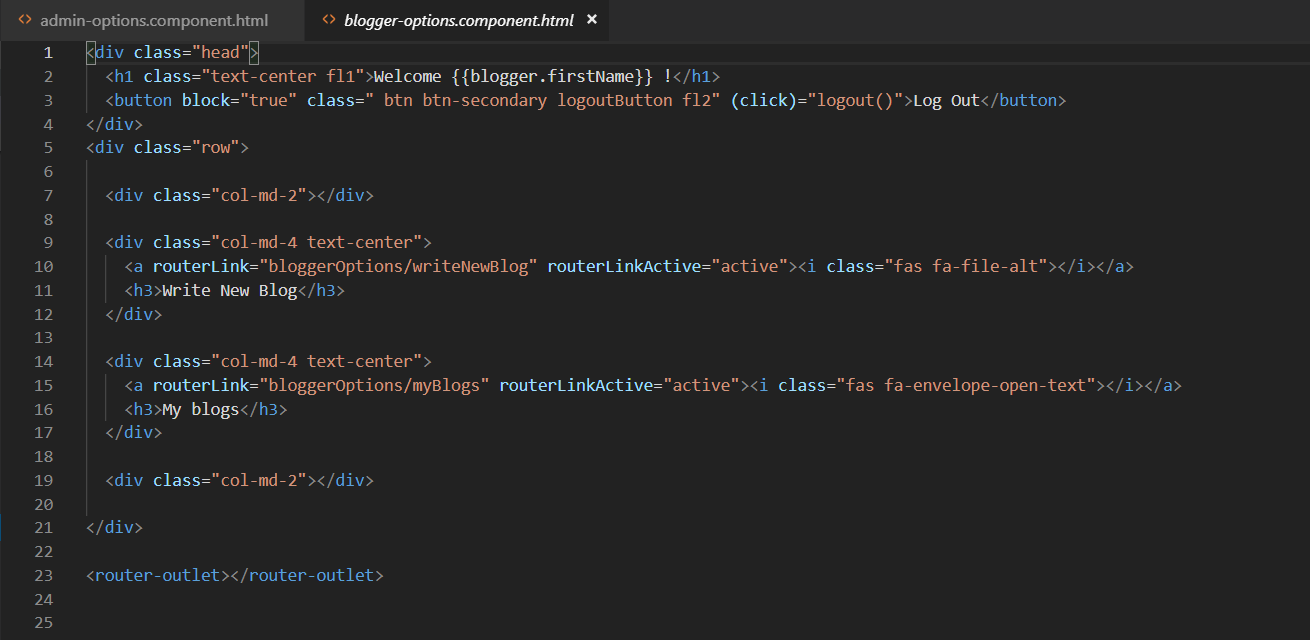
This component and all its children provide the administrator of the app with complete power over the contents and the users of the application, it uses all of the services and models available in the code.



Picture 41. Admin options Component

**Blogger Options Component**

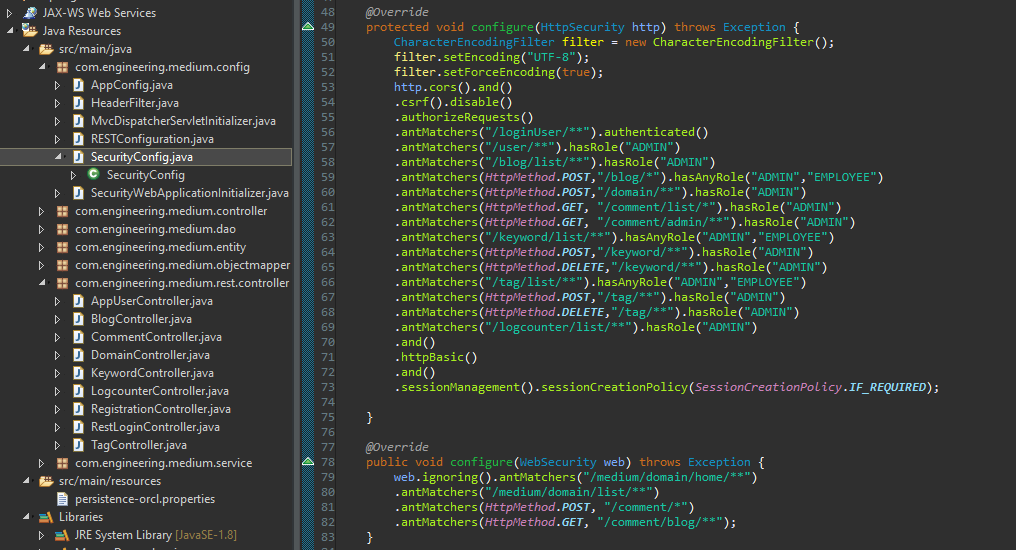
The blogger options component is very similar to the admin options component, it has its own routing module that renders its children in the component itself. This component has two child components, the first will let you write a new blog and post it and the other will show you a list of all the blogs that the logged in blogger has written. The blogger can delete or edit any of the blogs he has written.



Picture 42. Blogger options component

# 7.Security

When common user sends first request to our server it is redirected to the endpoint which returns content of our blog (domains, blogs, approved comments). Those endpoints are excluded from application security and they are available without use of credentials. If user try to access application backoffice it is prompted to enter his credentials. Those credentials are sent in request header as Authorization: Basic [base64 username and password encoded]. Endpoint for login is RestLoginController but as we are using Spring Security every request is first filtered by springSecurityFilterChain. If credentials pass Spring Security we take logged users username, get complete UserDetails object from DB and return it to the client. Now client has at his local storage all details about current user and credentials. Using HTTP interceptor in Angular every outgoing request has been intercepted and Authorization header is being added to the request until users local storage isn't cleaned up. Local storage is forced to cleanup every time logged user send logout request, and session at back-end is being invalidated.

Picture 43. Security configuration

Back-end of application security is based on Spring Security. By registering AbstractSecurityWebApplicationInitializer which extendsWebApplicationInitializer the DelegatingFilterProxy is forced to use the SpringSecurityFilterChain before any other registered Filter. That means that every incoming request to our application must pass Spring security check. Since we are using web services for communication between front-end (Angular) and back-end (Java Spring) and HTTP is stateless protocol, every request requires authentication.

We have created SecurityConfig class and annotated with *@Configuration* and *@EnableWebSecurity* which is needed to have the Spring Security configuration defined in any WebSecurityConfigurer, in our case by extending the WebSecurityConfigurerAdapter base class and overriding individual methods.

We have injected DataSource, a factory for connections to the physical data source that this DataSource object represents. A DataSource object is used for getting a connection. And by overriding configure method we have configured Spring security settings to work with user authentication from database data source.

After we override configure method for the second time we have pased in HttpSecurity object as argument and we have set up some filters. Using HttpSecurity object we have modified security settings and customized it based on our needs. First we have registered CORS filter needed to handle Cross-Origin Resource Sharing and we have disabled Cross-Site Request Forgery (CSRF) tokens. These tokens are used in Spring MVC application, since this is a web service application we dont need them. After that, using RequestMatcher's we have specified which RequestMatcher require a certain level of authorization. AntMatchers accepts HttPMethod and path to the certain endpoint as input parameters and then we can define level of access to this controller, or controller method using roles. We used multiple role access, single role and permitAll which allows everyone to access certain endpoint.

Finally once we override the configure method for the third time we have passed in Web Security object as argument and we have set list of antMatchers for endpoints we want to be excluded from web security like home page, adding new comment, and reading approved comments.

We have created CorsFilter that handles CORS preflight requests and intercepts CORS simple and actual requests in order to add the relevant CORS response headers (like Access-Control-Allow-Origin) using the provided UrlBasedCorsConfigurationSource instance. This is an alternative to Spring MVC Java config and XML namespace CORS configuration, useful for applications depending only on spring-web (not on spring-webmvc) or for security constraints requiring CORS checks to be performed at javax.servlet.Filter level. Here we have allowed use of credentials through HTTP headers and all HTTP methods we are using in our application.

8. TECHNOLOGIES

In our project we have used the following technologies: SQL Oracle Modeler, SQL Oracle Developer, ECLIPSE IDE(Maven, JPA, Hibernate, Apache Tomcat Server), VisualStudioCode(Angular, HTML, Cascade style sheet, Typescript, Bootstrap).