BlogChamp

Software Architecture

Github Repository: https://github.com/IzMo2000/BlogChamp

AUTHOR

Kane Davidson, Max Poole, Meaghan Freund, Izaac Molina, Josh VanderMeer

Table of Contents

1 Introduction

1.1 Motivation & Objectives

Our primary objective is to help provide an easy way for NAU students to connect and communicate with each other. Through BlogChamp, they will be able to create posts on the public feed or their personal blog, reply to each other, like other posts, and send friend requests to other users.

This small social network of NAU students is what motivates us, since it can inevitably lead to people becoming friends, communicating, and a more connected campus community. Ideally, users will be able to find others with similar interests, hobbies, and majors.

1.2 Introduce your team

Izaac Molina - Computer Science and Biomedical Science Major, Fall ‘24

Maximilian Poole - Computer Science Major, Spring ‘25

Meaghan Freund - Computer Science Major, Spring ‘25

Kane Davidson - Computer Science Major, Fall ‘24

Joshua VanderMeer - Computer Science Major, Spring ‘25

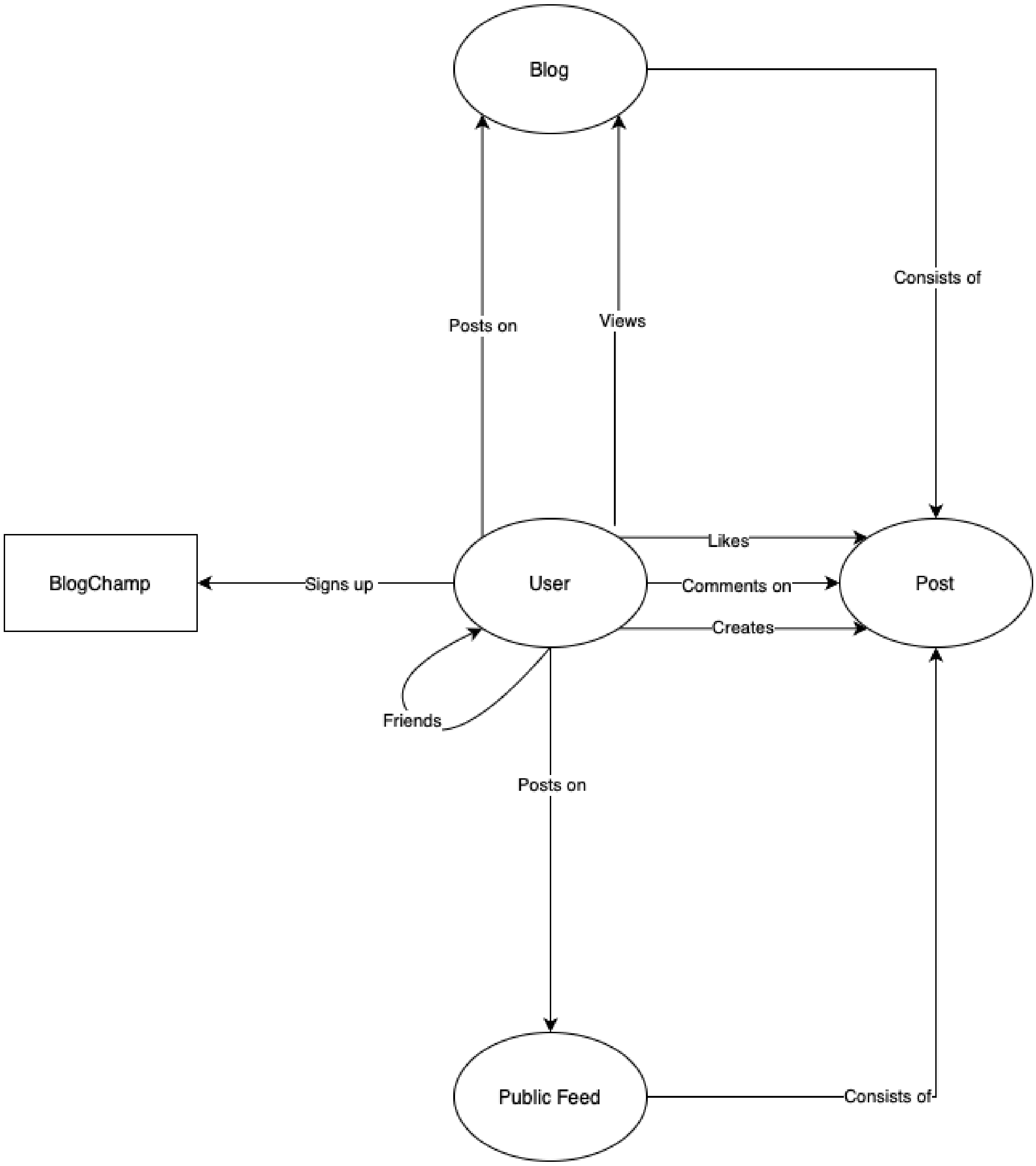
2 Requirement Analysis

2.1 Stakeholder Analysis

Our stakeholders will consist of: users, NAU students, other competing social media platforms such as Twitter/X, and the developers of these competitors. Our primary stakeholders are the users of BlogChamp, since they are who the blog is made for. Because they are our primary stakeholders, we will prioritize the users' feedback and create BlogChamp based on their needs.

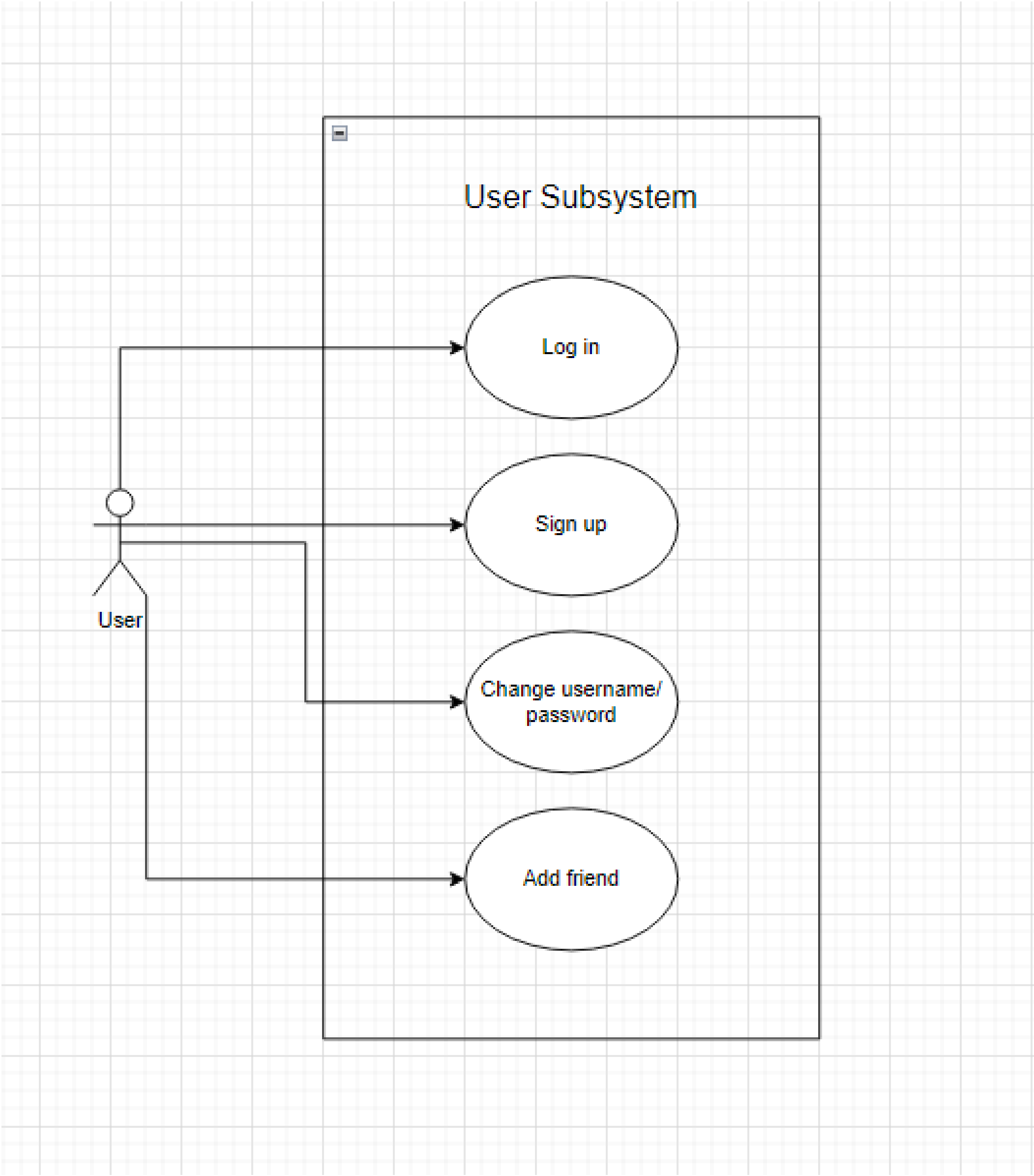
Throughout our development process, we will address our stakeholders' needs, interests, and concerns. We can do this by conducting interviews and surveys with potential users, and make changes to BlogChamp when it is appropriate. Ideally, BlogChamp will act as a place where users can provide feedback directly, whether it be through a private survey mechanic or by simply posting their feedback publicly.

2.2 Scope Modeling



2.3 Business Use Case Definition

2.3.1 Use Case Model – User Subsystem



2.3.2 Business Use Case Description – User Subsystem

Log In

Use Case Name	Log in	System Analysis
Use Case ID	BC001	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	This use case describes the event of a user logging in to their account	
Precondition	User has an account	
Trigger	The user is going to log in to their account	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects the login button on the web page Step 3: User fills in the username and password input field and submits	Step 2: System allows the user to access the username and password input fields Step 4: System checks if the username and password are correct Step 5: System allows the user access to their page
Alternate Courses	User inputs the wrong username or password	
	Actor Action	System Response
	Step 1: User selects the login button on the web page Step 3: User fills in the username and password input field and submits Step 6: User re-enters their username and password	Step 2: System allows the user to access the username and password input fields. Step 4: System checks if the username and password are correct Step 5: System clears the input fields and displays a warning message
Conclusion	This use case concludes when the user successfully logs in or stops attempting	
Business Rules	The user had been properly registered in the user database	

Implementation Constraints and Specifications	Frequency - It is estimated that this use case will be executed 100 times a day. Support: Up to 30 concurrent users
Assumptions	None
Open Issues	None

Sign Up

Use Case Name	Sign up	System Analysis
Use Case ID	BC002	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	This use case describes the event of a user signing up for an account	
Precondition	none	
Trigger	The customer is going to create an account	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects the sign up button Step 3: User fills in the username and password field and submits	Step 2: System allows the user to access a username and password field Step 4: System stores the username and password in the database
Alternate Courses	The user tries to sign up with an existing account	
	Actor Action	System Response
	Step 1: User selects the sign up button Step 3: User fills in the username and password field and submits	Step 2: System allows the user to access a username and password field. Step 4: System detects that the username already exists

		Step 5: System redirects the user to the login page
Conclusion	This use case concludes when the user successfully creates a new account or gets directed to the login page	
Business Rules	none	
Implementation Constraints and Specifications	Frequency - It is estimated that this use case will be executed 30 times a day. Support: Up to 10 concurrent users	
Assumptions	None	
Open Issues	None	

Change Username/Password

Use Case Name	Change username/password	System Analysis
Use Case ID	BC003	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	This use case describes the event of a user changing their username and/or password	
Precondition	User is logged in	
Trigger	User selects option on their page to start a blog	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects the change password button Step 3: User fills in the password field Step 5: User retypes their password in the confirm password field	Step 2: System allows the user to access a password field. Step 4: System allows access to the confirm password field Step 6: System checks if both passwords are the same and updates the database

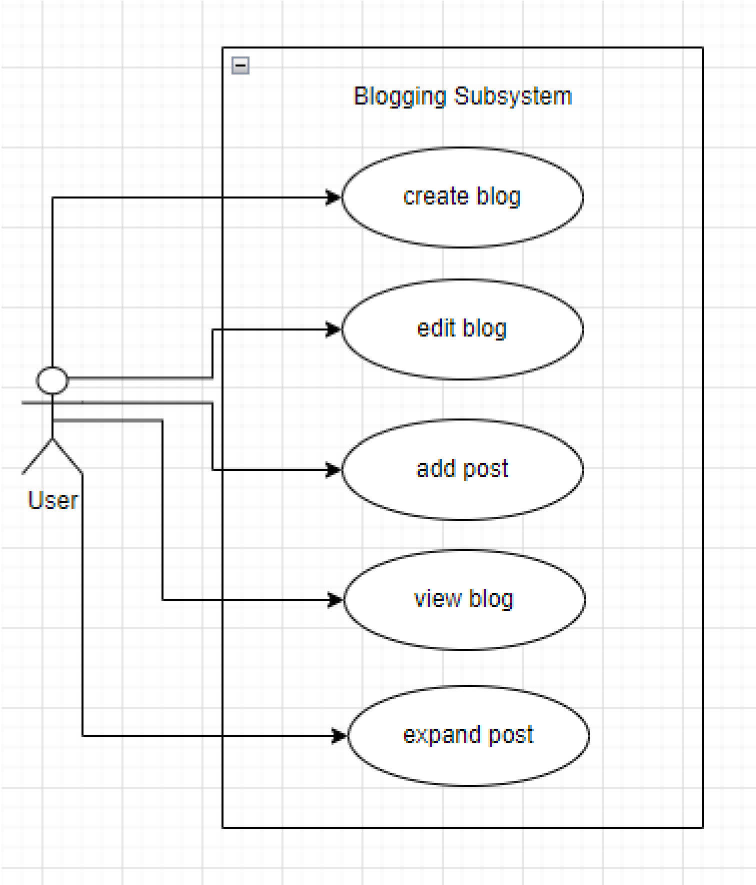
Alternate Courses	User just wants to update their username	
	Actor Action	System Response
	Step 1: User selects the change username field Step 3: User inputs their new password Step 4: User submits the change	Step 2: System allows the user to access a change username field Step 5: System updates the database with the new username
Conclusion	This use case concludes when the user successfully updates their username or password or backs out	
Business Rules	The user had been properly registered in the user database	
Implementation Constraints and Specifications	Frequency - It is estimated that this use case will be executed 30 times a day. Support: Up to 10 concurrent users	
Assumptions	None	
Open Issues	None	

Add Friend

Use Case Name	Add friend	System Analysis
Use Case ID	BC004	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	This use case describes the event of a user adding a friend	
Precondition	User is logged in User knows the friends username	
Trigger	The user is going to add another friend	

Typical Course of Events	Actor Action	System Response
	Step 1: User selects add friend option on their page Step 3: User fills in the field with the username of the person they want to friend	Step 2: System allows access to an input username field Step 4: System checks for valid username Step 5: System adds the friend
Alternate Courses	User wants to remove a friend	
	Actor Action	System Response
	Step 1: User selects add friend option on their page Step 3: User fills in the field with the username of the person they want to friend	Step 2: System allows access to an input username field Step 4: System checks for valid username Step 5: System removes the friend
Conclusion	This use case concludes when the user successfully adds or removes a friend or backs out	
Business Rules	The user had been properly registered in the user database	
Implementation Constraints and Specifications	Frequency - approx. executed 100 times a day. Support: Up to 40 concurrent users	
Assumptions	None	
Open Issues	None	

2.3.3 Use Case Model – subsystem 2



2.3.4 Business Use Case Description – subsystem 2

Create Blog

Use Case Name	Create blog	System Analysis
Use Case ID	BC005	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	Describes the event of a user deciding to start a blog on their page	

Precondition	User is logged in	
Trigger	User selects option on their page to start a blog	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects “add blog” option on their page Step 4: User fills in initialization info for blog, (i.e. name, description) Step 6: User sees new blog page	Step 2: System checks to make sure the blog limit is not reached Step 3: System throws prompt for blog initialization Step 5: System adds blog to database, loads and displays new blog page
Alternate Courses	User has reached the maximum number of blogs	
	Actor Action	System Response
	Step 1: User selects “add blog” option on their page Step 4: User sees warning that limit has been reached, must close warning to proceed	Step 2: System checks to make sure the blog limit is not reached. Step 3: System detects blog limit reached, throw message to user
Conclusion	Concludes when the user has created a new blog or has been notified they cannot create any more blogs.	
Business Rules	The user had been properly registered in the user database	
Implementation Constraints and Specifications	Frequency - approx. executed 30 times a day. Support: Up to 10 concurrent users	
Assumptions	None	
Open Issues	None	

Edit Blog

Use Case Name	Edit blog	System Analysis
Use Case ID	BC006	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	

Stakeholders	Users (NAU Students)	
Description	Describes the event of a user deciding to edit one of their preexisting blogs	
Precondition	User is logged in Blog to be edited exists in system	
Trigger	User selects option on the blog page to edit their blog	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects "edit blog" option on the blog page Step 3: User fills in updated blog info. Step 5: User sees success message and new updated blog info	Step 2: System receives request, throws prompt for edit dialog with name and description Step 4: System updates blog information in database
Alternate Courses	None	
Conclusion	Concludes when the user has updated their blog information successfully.	
Business Rules	The user had been properly registered in the user database, blog was set up in database	
Implementation Constraints and Specifications	Frequency - approx. executed 30 times a day. Support: Up to 15 concurrent users	
Assumptions	None	
Open Issues	None	

Add Post

Use Case Name	Add post	System Analysis
Use Case ID	BC007	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	

Description	Describes the event of a user adding a post to their blog	
Precondition	User is logged in Blog to be added onto exists in the system	
Trigger	User selects option to add post to blog	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects “add post” option on the blog page Step 4: User is directed to “create post” use case in the Post subsystem	Step 2: System receives request, checks to ensure post limit on blog not reached Step 3: Check passes, system directs user to Post subsystem
Alternate Courses	Blog has reached post limit	
	Actor Action	System Response
	Step 1: User selects “add post” option on the blog page Step 4: User is notified of failure to add a post.	Step 2: System receives request, checks to ensure post limit on blog not reached Step 3: Check fails, system notifies user that limit has been reached.
Conclusion	Concludes when the user has been directed to Post subsystem to create the post or when user has been notified that post limit has been reached	
Business Rules	The user had been properly registered in the user database, blog was set up in database	
Implementation Constraints and Specifications	Frequency - approx. executed 100 times a day. Support: Up to 30 concurrent users	
Assumptions	None	
Open Issues	None	

View Blog

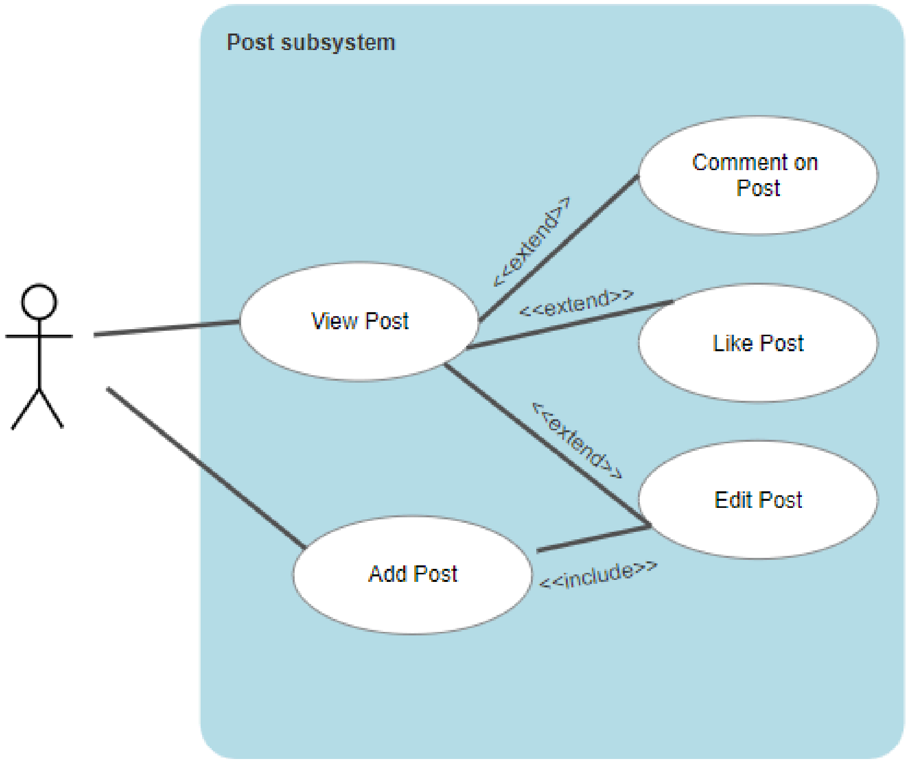
Use Case Name	View blog	System Analysis
Use Case ID	BC008	
Primary Business Actor	User	
Other Participating Actor	None	

Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	Describes the event of a user attempting to view a blog	
Precondition	User is logged in Blog to be viewed exists in the system	
Trigger	User selects a blog to view	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects the blog name, requesting to open the blog page Step 4: User is directed to the blog page, is now viewing the page	Step 2: System receives request, checks to ensure user has authorization to view the blog. Step 3: Check passes, system directs user to blog page
Alternate Courses	User does not have authorization to view blog (i.e. not the creator and is not friends with the creator)	
	Actor Action	System Response
	Step 1: User attempts to access blog through untraditional means, i.e. using a link to it Step 4: User is redirected to page, notified that they do not have permission to view that blog	Step 2: System checks user authorization, sees that user is not authorized Step 3: System rejects user, redirects them to their page
Conclusion	Concludes when the user sees the requested blog, or has been redirected due to restricted access.	
Business Rules	The user had been properly registered in the user database, blog was set up in database, user must be friends with blog creator (or the creator themselves) to view their blog	
Implementation Constraints and Specifications	Frequency - approx. executed 500 times a day. Support: Up to 50 concurrent users	
Assumptions	None	
Open Issues	None	

Expand Post

Use Case Name	Expand post	System Analysis
Use Case ID	BC009	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	Describes the event of a user expanding a post on a given blog page	
Precondition	User is logged in Blog to be added onto exists in the system User is viewing the blog page (and therefore has permission to access its posts)	
Trigger	User selects the post to expand	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects post name on the given blog page, requesting to expand and view the post Step 4: User is directed to "view post" use case in the Post subsystem	Step 2: System receives request, finds post Step 3: System finds post information, directs user to Post subsystem
Alternate Courses	None	
Conclusion	Concludes when the user has been directed to Post subsystem to view the given post	
Business Rules	The user had been properly registered in the user database, blog was set up in database, user must have authorized access to the blog (and therefore post)	
Implementation Constraints and Specifications	Frequency - approx. executed 150 times a day. Support: Up to 40 concurrent users.	
Assumptions	None	
Open Issues	None	

2.3.5 Use Case Model – subsystem 3



2.3.6 Business Use Case Description – subsystem 3

View Post

Use Case Name	View Post	System Analysis
Use Case ID	BC0010	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	This use case describes the event of a user opening a post.	
Precondition	User is logged in.	

	User wants to look at a post	
Trigger	User is going to like, comment, or edit a post	
Typical Course of Events	Actor Action	System Response
	Step 1: User selects sees a list of posts from post page Step 2: User clicks on a post	Step 3: System opens up the post page
Alternate Courses	User has reached the maximum number of blogs	
	Actor Action	System Response
	Step 1: User selects “add blog” option on their page Step 4: User sees warning that limit has been reached, must close warning to proceed	Step 2: System checks to make sure the blog limit is not reached. Step 3: System detects blog limit reached, throw message to user
Conclusion	Concludes when the user has created a new blog or has been notified they cannot create any more blogs.	
Business Rules	The user had been properly registered in the user database, user has permission to view post	
Implementation Constraints and Specifications	Frequency - approx. executed 30 times a day. Support: Up to 10 concurrent users	
Assumptions	None	
Open Issues	None	

Edit Post

Use Case Name	Edit Post	System Analysis
Use Case ID	BC0011	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	

Description	This use case describes the event of a user editing a post.	
Precondition	User is logged in. Post to be edited exists in the system	
Trigger	User has selected a post to edit	
Typical Course of Events	Actor Action	System Response
	Step 1: The user clicks the edit button Step 3: The user edits the caption or photo Step 4: The user clicks confirm to finish editing	Step 2: The system opens the edit post options Step 5: The system accepts the edit and solidifies the post.
Alternate Courses	User cancels their edit	
	Actor Action	System Response
	Step 1: User edits their post Step 2: User exits without saving	Step 3: System does not detect changes, leaves post as it was prior
Conclusion	The use case concludes when a user completes editing their post or exits the edit post page.	
Business Rules	The user had been properly registered in the user database, user has ownership over their post	
Implementation Constraints and Specifications	Frequency - approx. executed 30 times a day. Support: Up to 10 concurrent users	
Assumptions	None	
Open Issues	None	

Comment on Post

Use Case Name	Comment on post	System Analysis
Use Case ID	BC0012	
Primary Business Actor	User	

Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	This use case describes the event of a user commenting on a post.	
Precondition	User is logged in. Post exists in the system and allows comments. User has permission to comment	
Trigger	User has selected a post to comment on	
Typical Course of Events	Actor Action	System Response
	Step 1: The user clicks the comment button Step 3: The user types out their comment Step 4: The user clicks the confirm/send button	Step 2: The system opens up a bar to type a comment Step 5: The system ends the ability to type and confirms the comment
Alternate Courses	User cancels their comment	
	Actor Action	System Response
	Step 1: User types out their comment Step 2: User exits without saving	Step 3: System does not detect new comment, does not post comment
Conclusion	The use case concludes when a user completes commenting on a post or cancels their comment.	
Business Rules	The user had been properly registered in the user database, user has permission to comment	
Implementation Constraints and Specifications	Frequency - approx. executed 30 times a day. Support: Up to 10 concurrent users	
Assumptions	None	
Open Issues	None	

Like Post

Use Case Name	Like post	System Analysis
Use Case ID	BC0013	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	This use case describes the event of a user liking a post.	
Precondition	User is logged in. Post exists in the system. User has permission to like post.	
Trigger	User has selected a post to like	
Typical Course of Events	Actor Action	System Response
	Step 1: The user clicks the like button	Step 2: The system accepts the like and adds the number to the like count
Alternate Courses	User unlikes a post	
	Actor Action	System Response
	Step 1: The user clicks the like button Step 3: The user clicks the like button a second time	Step 2: The system accepts the like and adds the number to the like count Step 4: The system accepts the second click, takes out 1 from the like count
Conclusion	The use case concludes when a user likes a post or unlikes a post.	
Business Rules	The user had been properly registered in the user database, user has permission to like.	
Implementation Constraints and Specifications	Frequency - approx. executed 30 times a day. Support: Up to 10 concurrent users	
Assumptions	None	

Open Issues	None

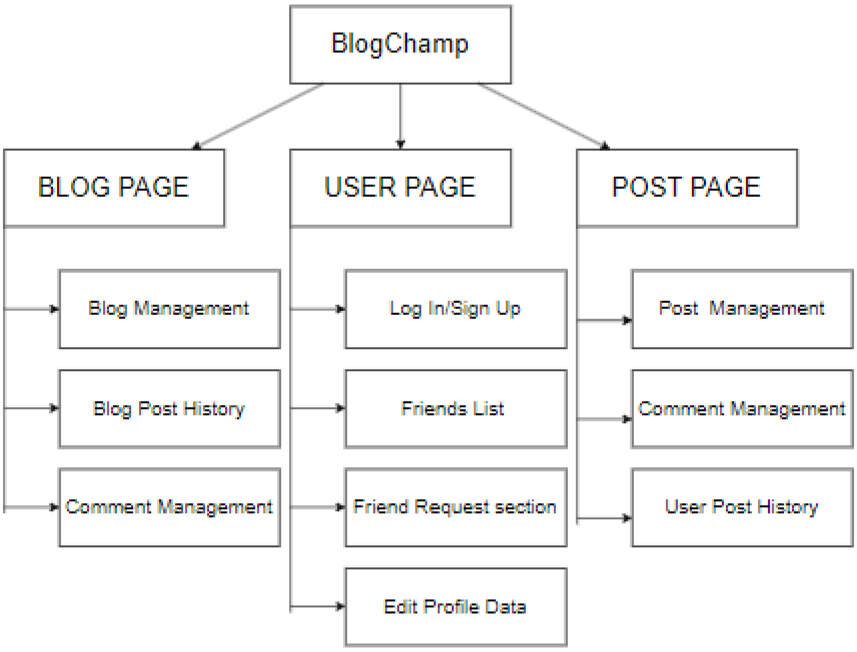
Add Post

Use Case Name	Add post	System Analysis
Use Case ID	BC0014	
Primary Business Actor	User	
Other Participating Actor	None	
Other Interested	None	
Stakeholders	Users (NAU Students)	
Description	This use case describes the event of a user adding a post.	
Precondition	User is logged in.	
Trigger	User selects to create a new post.	
Typical Course of Events	Actor Action	System Response
	Step 1: The user clicks the add post button Step 3: The user edits their post as they wish Step 4: The user confirms their post	Step 2: The system opens up the add post page Step 5: The system accepts their post and uploads it to their page.
Alternate Courses	User cancels adding a post	
	Actor Action	System Response
	Step 1: The user creates the details of their post Step 2: The user leaves the post without saving	Step 3: The system does not detect a change to the user's account, does not create a new post
Conclusion	The use case concludes when a user creates a post	
Business Rules	The user had been properly registered in the user database, user has not reached maximum capacity for their blog.	

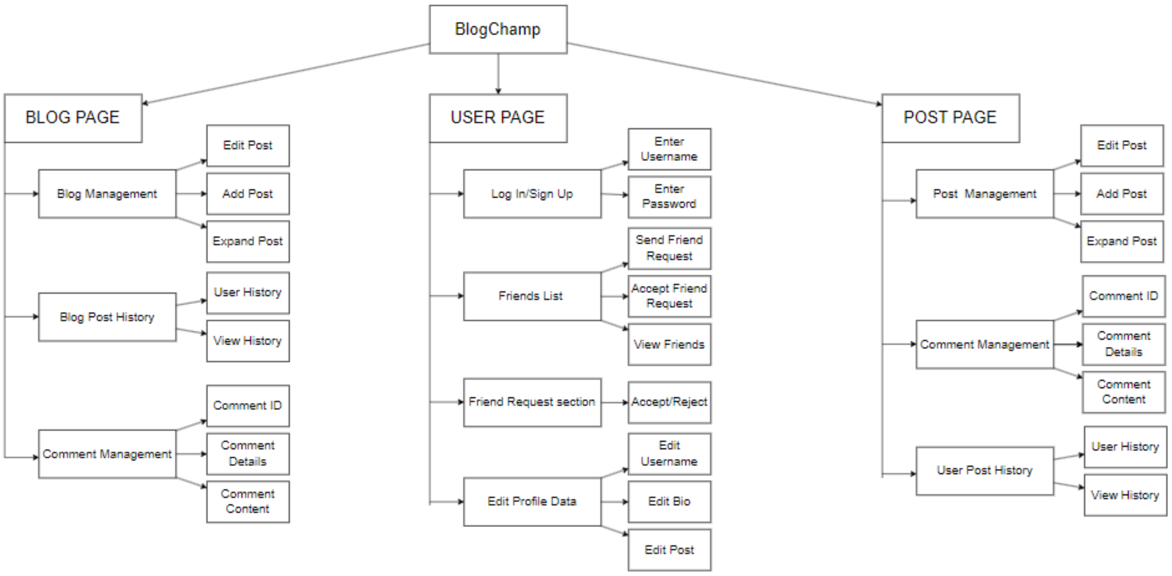
Implementation Constraints and Specifications	Frequency - approx. executed 100 times a day. Support: Up to 10 concurrent users
Assumptions	None
Open Issues	None

2.4 Functional Requirements

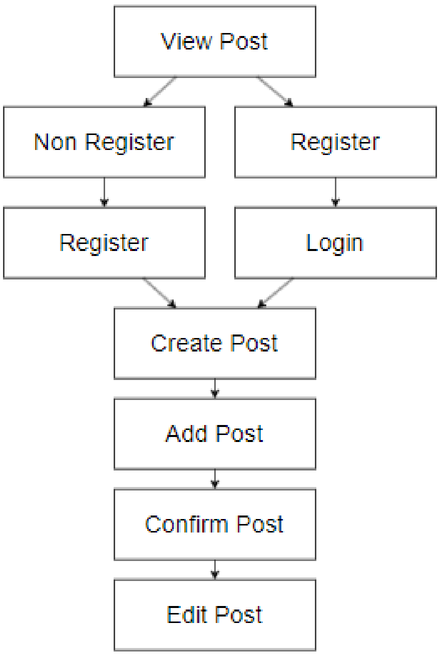
2.4.1 Functional Decomposition



2.4.2 Functional Decomposition (include action)



2.4.3 Process Model

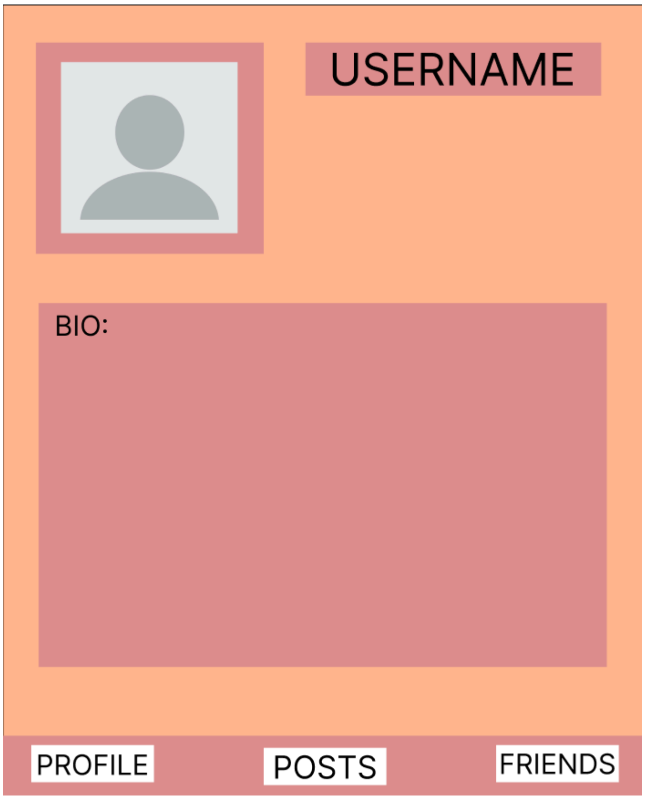


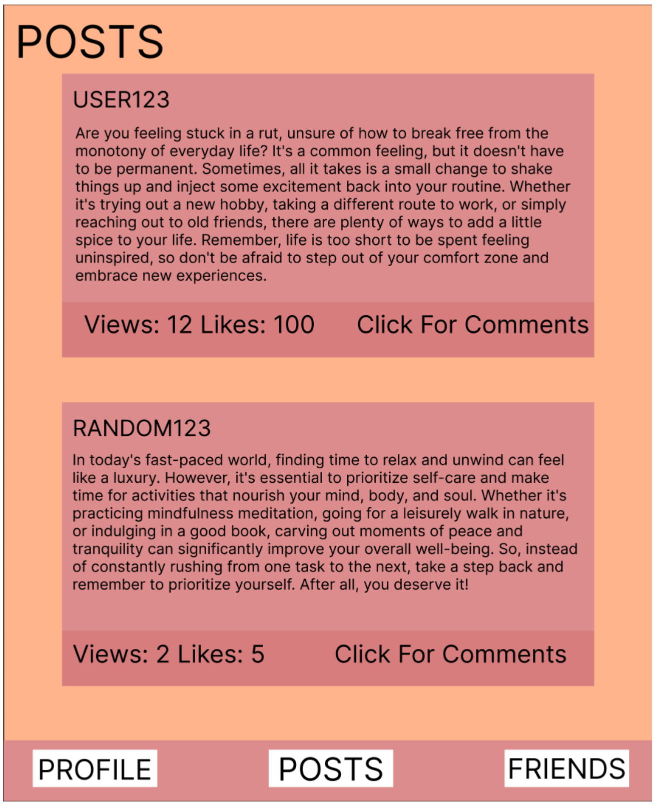
-

2.4.4 Web Page resource

The user profile page will be connected to the posts page and vice-versa. From the posts page a user will be able to jump to their profile page and from the profile they will be able to go right to the posts page. The blog page will not be visible from the posts page because it is only accessible to friends so a user will get to it by viewing their friends from their profile and selecting a blog. From the blog page a user will be able to access the posts and profile page.

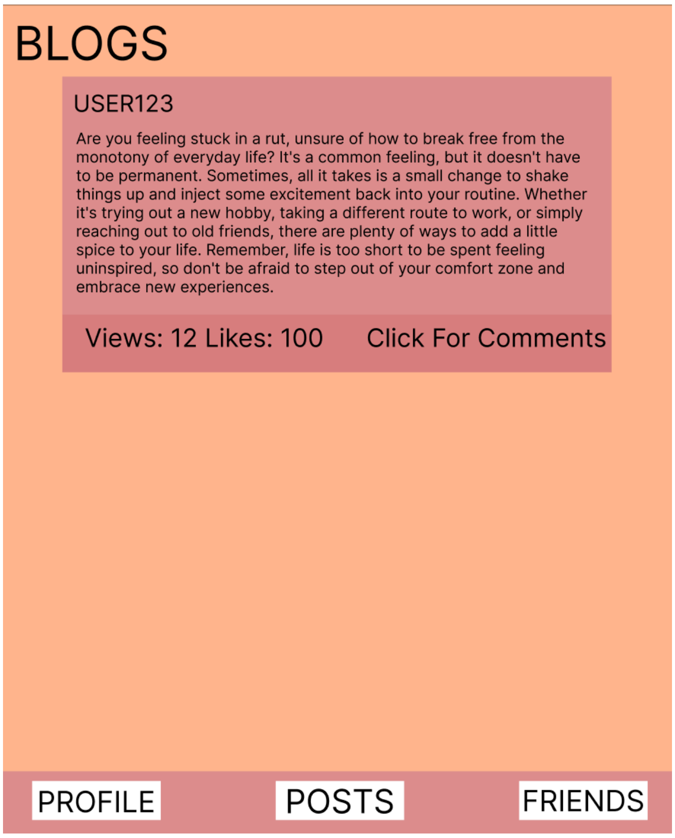
2.4.5 User Interface design





Are you feeling stuck in a rut, unsure of how to break free from the monotony of everyday life? It's a common feeling, but it doesn't have to be permanent. Sometimes, all it takes is a small change to shake things up and inject some excitement back into your routine. Whether it's trying out a new hobby, taking a different route to work, or simply reaching out to old friends, there are plenty of ways to add a little spice to your life. Remember, life is too short to be spent feeling uninspired, so don't be afraid to step out of your comfort zone and embrace new experiences.

In today's fast-paced world, finding time to relax and unwind can feel like a luxury. However, it's essential to prioritize self-care and make time for activities that nourish your mind, body, and soul. Whether it's practicing mindfulness meditation, going for a leisurely walk in nature, or indulging in a good book, carving out moments of peace and tranquility can significantly improve your overall well-being. So, instead of constantly rushing from one task to the next, take a step back and remember to prioritize yourself. After all, you deserve it!



BLOGS

USER123

Are you feeling stuck in a rut, unsure of how to break free from the monotony of everyday life? It's a common feeling, but it doesn't have to be permanent. Sometimes, all it takes is a small change to shake things up and inject some excitement back into your routine. Whether it's trying out a new hobby, taking a different route to work, or simply reaching out to old friends, there are plenty of ways to add a little spice to your life. Remember, life is too short to be spent feeling uninspired, so don't be afraid to step out of your comfort zone and embrace new experiences.

Views: 12 Likes: 100

2.5 Non-Functional Requirements (bonus)

2.5.1 Performance

Source	Stimulus	Environment	Artifact	Response	Response Measure
Internal	All Users online	Normal Operation	System	<=1000	Capacity

2.5.2 Scalability

Source	Stimulus	Environment	Artifact	Response	Response Measure
Internal	Data Size is too much	Normal Operation	Hardware	Hardware is added to allow for more data storage	No Downtime

2.5.3 Modifiability

Source	Stimulus	Environment	Artifact	Response	Response Measure
Developer	Wishes to add a feature	Post product launch	Code	Modification is made with no side effects on the site	Depending on the feature between 5-10 hours

2.5.4 Security

Source	Stimulus	Environment	Artifact	Response	Response Measure
External	User tries to change or delete data	Online	Data within the system	System blocks access to the data	Extent to which the data is damaged

2.5.5 Availability

Source	Stimulus	Environment	Artifact	Response	Response Measure
Internal	Too many users	Normal Operation	Process	Inform operator to continue to operate	No Downtime

2.5.6 Usability

Source	Stimulus	Environment	Artifact	Response	Response Measure
End User	User wants to learn all of the system features	At runtime	System	System provides a help menu to give the user assistance	User satisfaction

2.5.7 Reliability

Source	Stimulus	Environment	Artifact	Response	Response Measure
Internal	System goes down	At runtime	System	The system is rebooted and comes back online	At most 2-3 hours of downtime

2.5.8 Maintainability

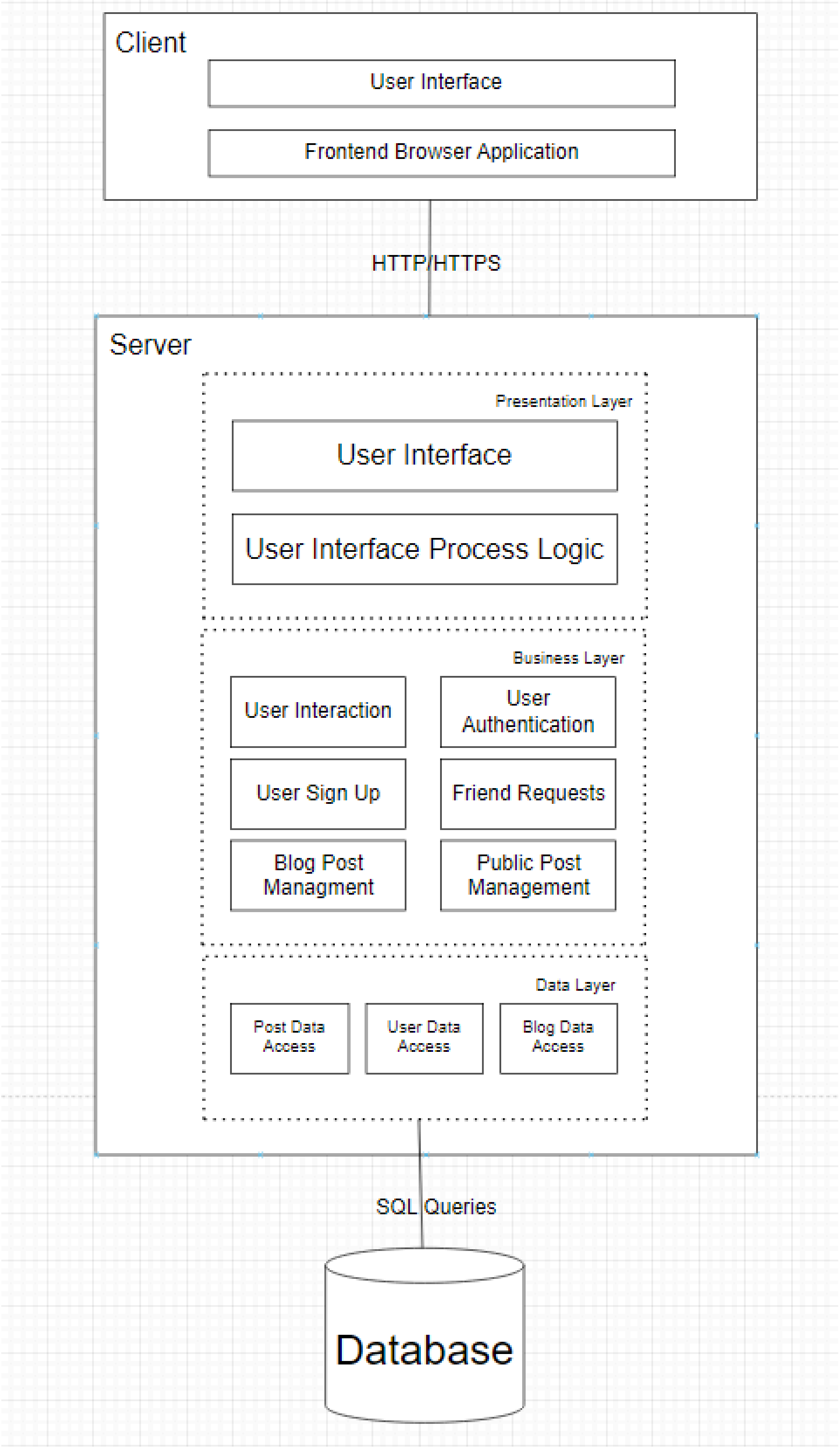
Source	Stimulus	Environment	Artifact	Response	Response Measure
Developer	Wants to update the code	At runtime	Code	Update is made with no side effects	No downtime

2.5.9 Flexibility

Source	Stimulus	Environment	Artifact	Response	Response Measure
Developer	Wants to disable a feature	At runtime	Code	Feature is disabled with no side effects	No downtime

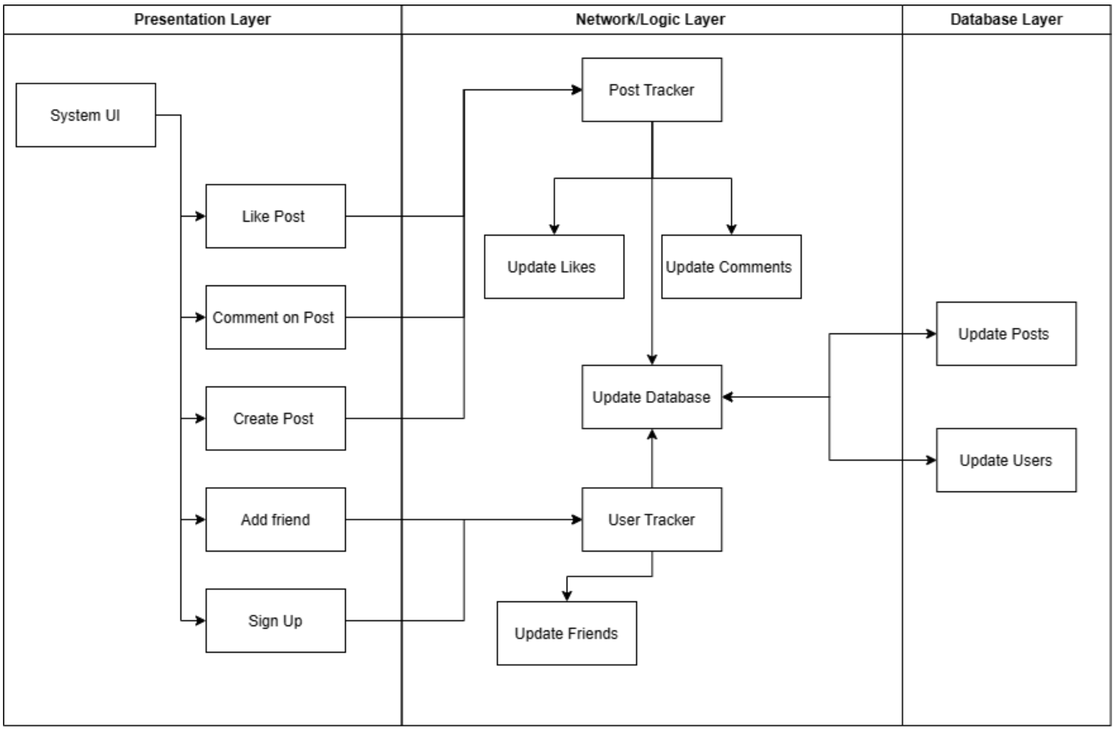
3.1 Architecture Style

We will be using a client server architecture for this project.
The client server architecture will allow multiple clients to access the server and be serviced at the same time. With a blog style application it is important for multiple users to be able to interact with the same server at the same time.

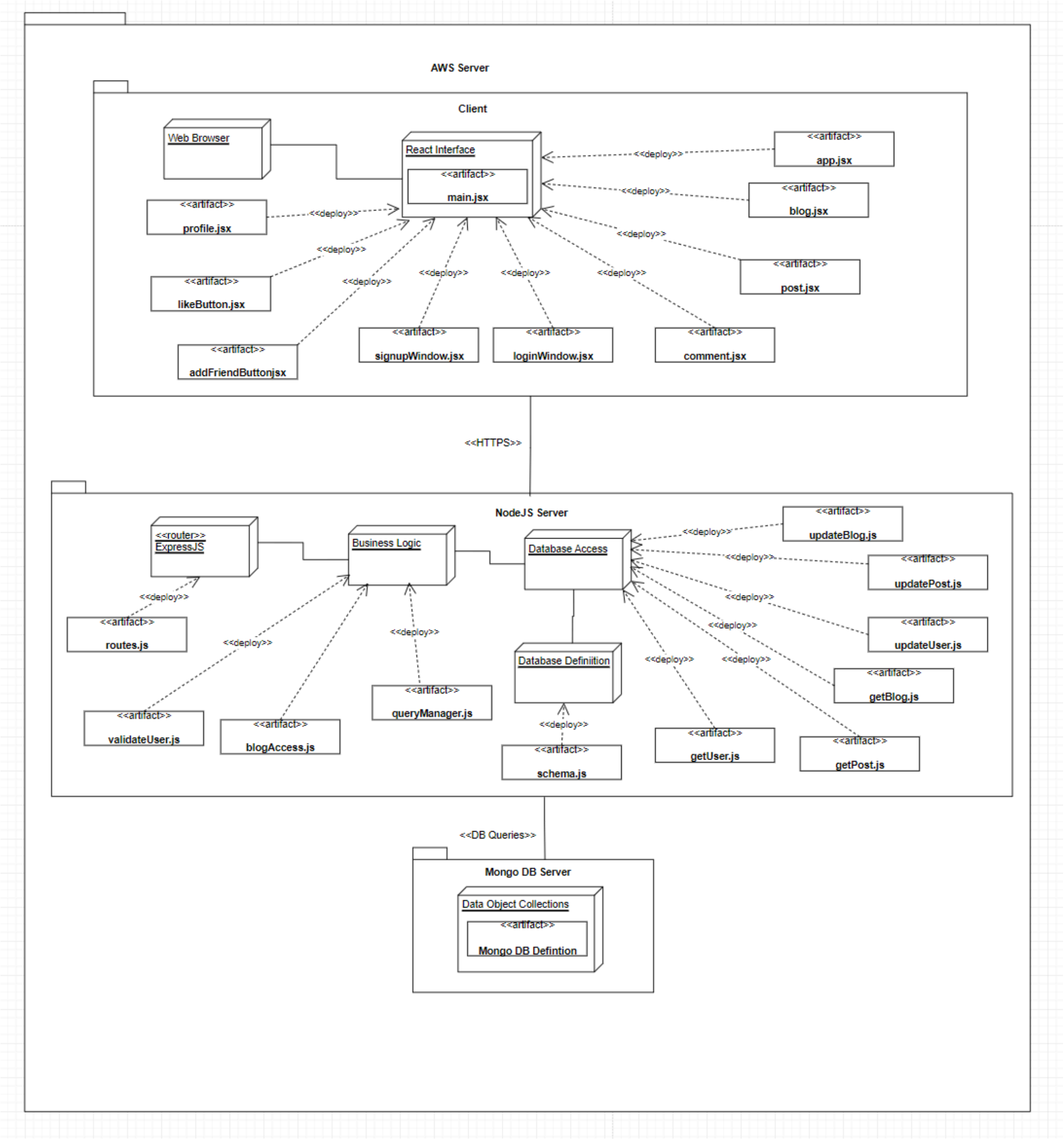

Client
User Interface
Frontend Browser Application
HTTP/HTTPS
Server
Presentation Layer
User Interface
User Interface Process Logic
Business Layer
User Interaction
User Authentication
User Sign Up
Friend Requests
Blog Post Managment
Public Post Management
Data Layer
Post Data Access
User Data Access
Blog Data Access
SQL Queries
Database

3.2 View Models

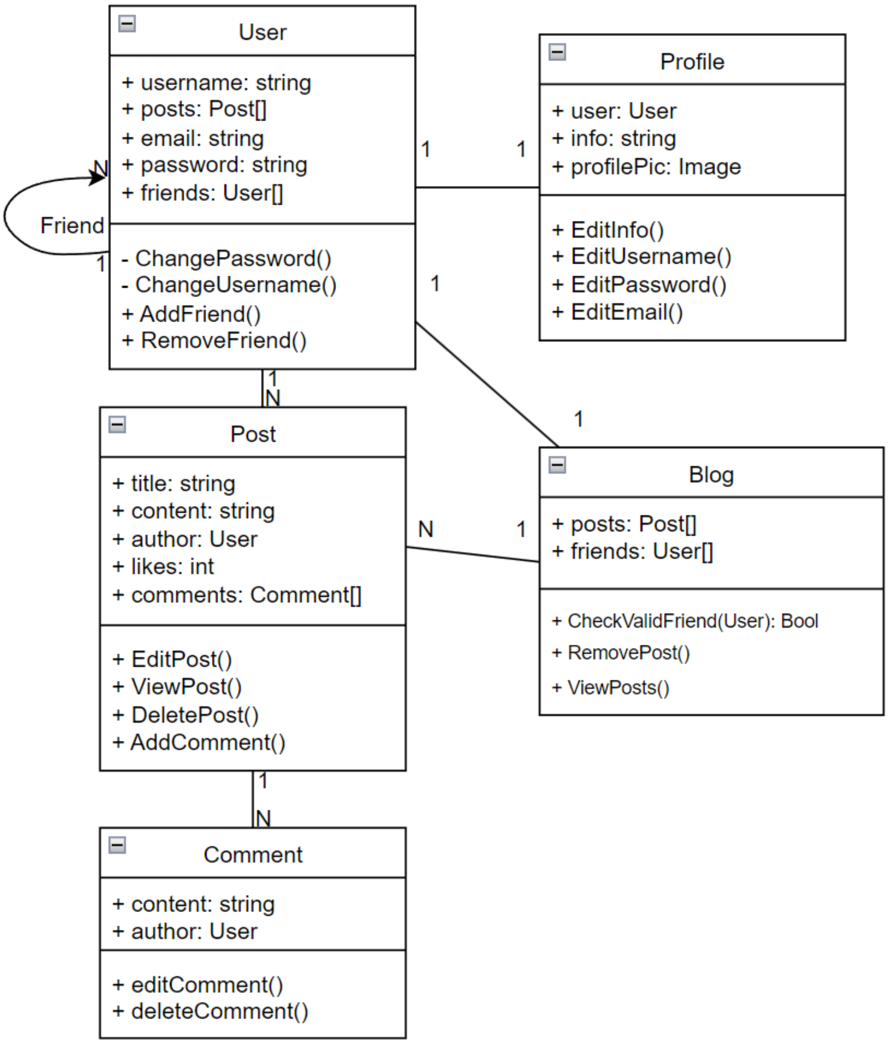
3.2.1 Implementation View



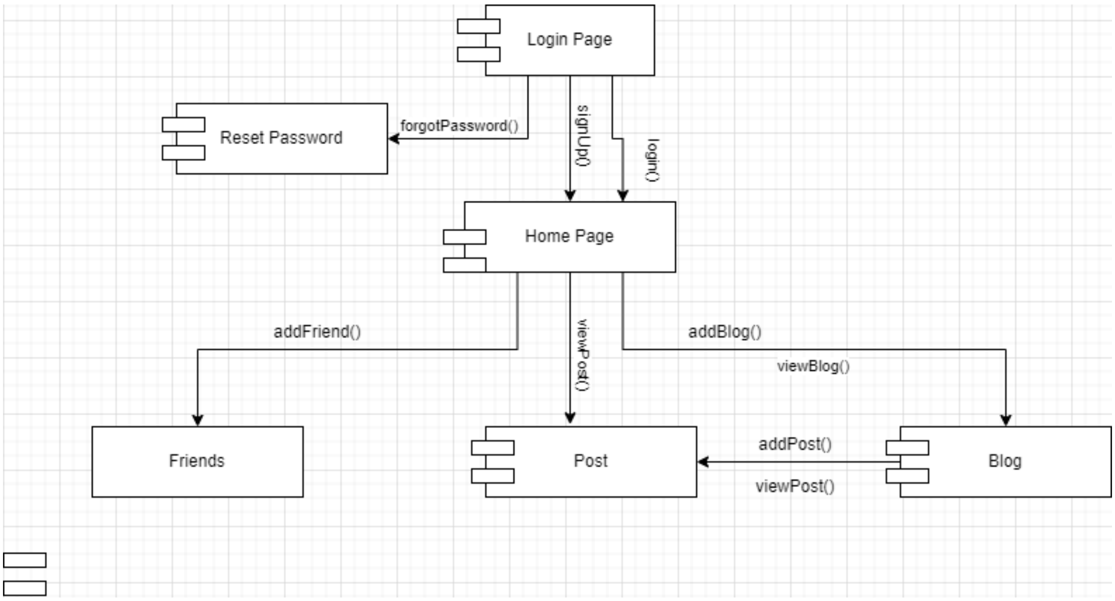
3.2.2 Deployment View

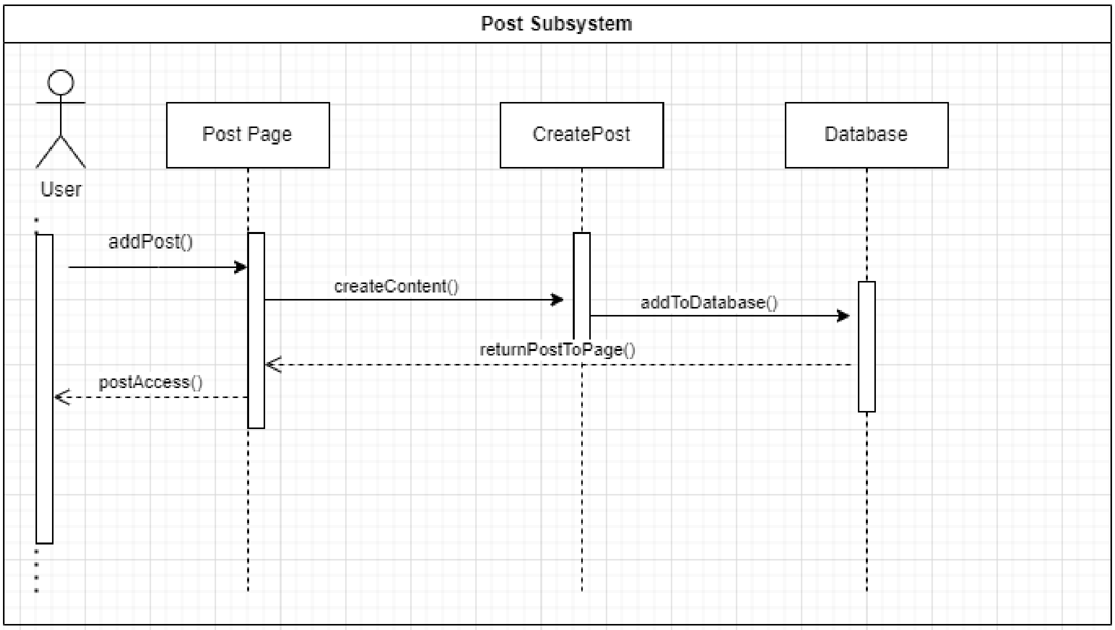


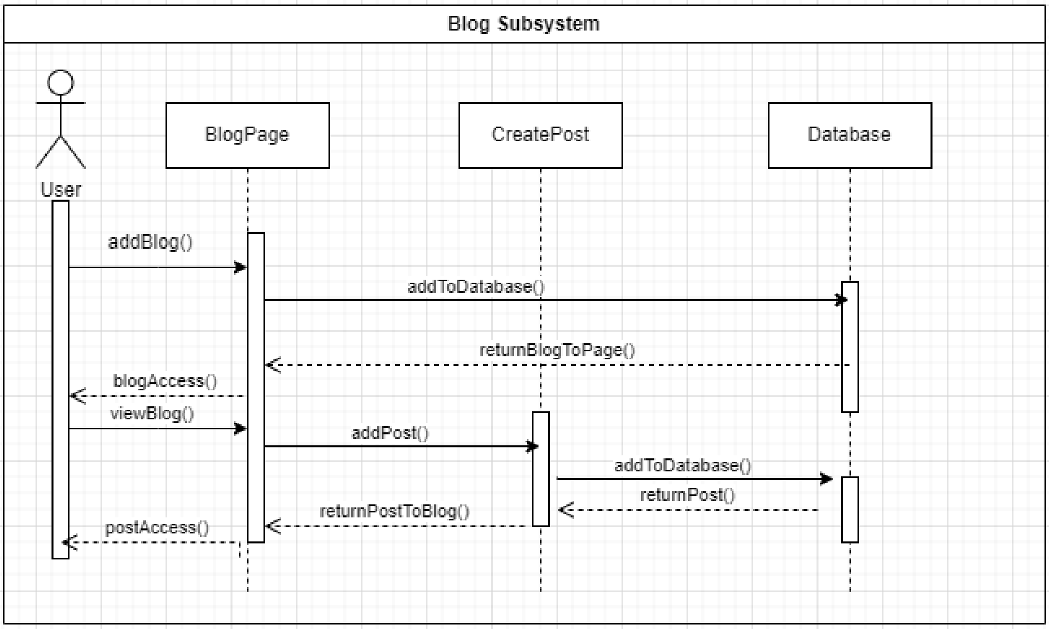
3.2.3 Logical View

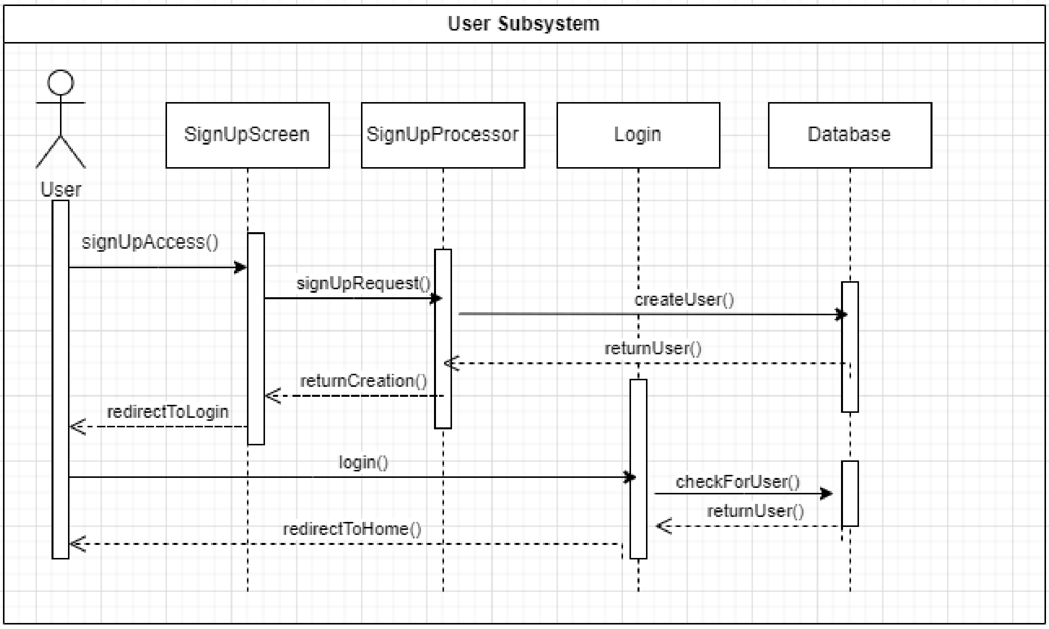


3.2.4 Process View

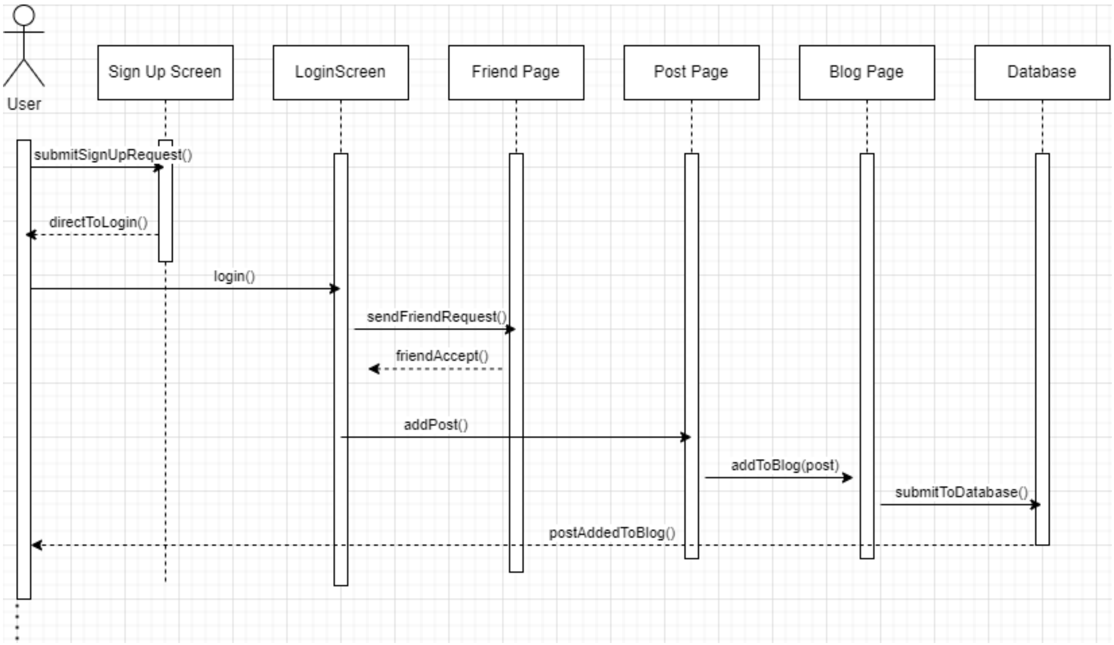



Post Subsystem
User
Post Page
CreatePost
Database
addPost()
createContent()
addToDatabase()
returnPostToPage()
postAccess()

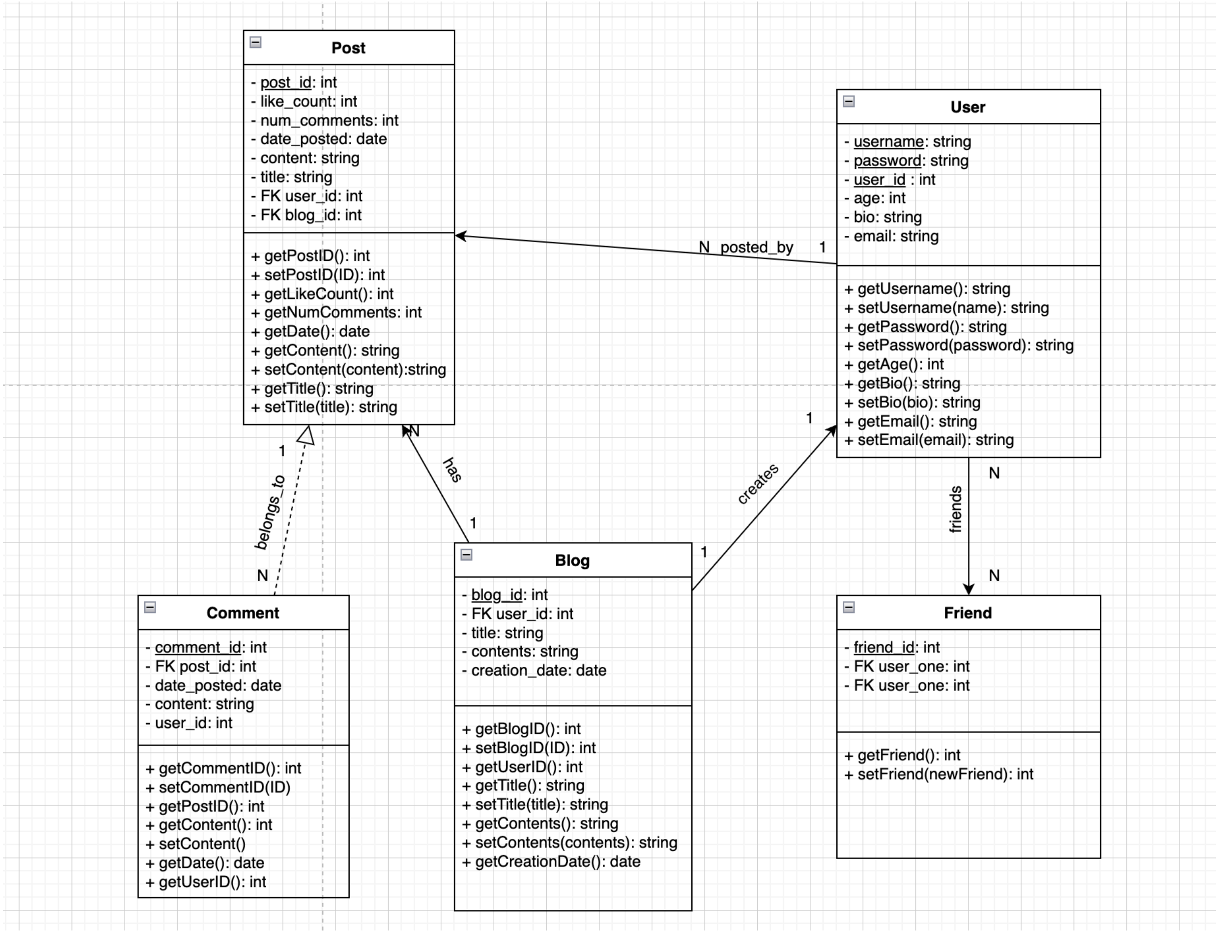

Blog Subsystem
User
BlogPage
CreatePost
Database
addBlog()
addToDatabase()
returnBlogToPage()
blogAccess()
viewBlog()
addPost()
addToDatabase()
returnPost()
returnPostToBlog()
postAccess()


User Subsystem
User
SignUpScreen
SignUpProcessor
Login
Database
signUpAccess()
signUpRequest()
createUser()
returnUser()
returnCreation()
redirectToLogin
login()
checkForUser()
returnUser()
redirectToHome()

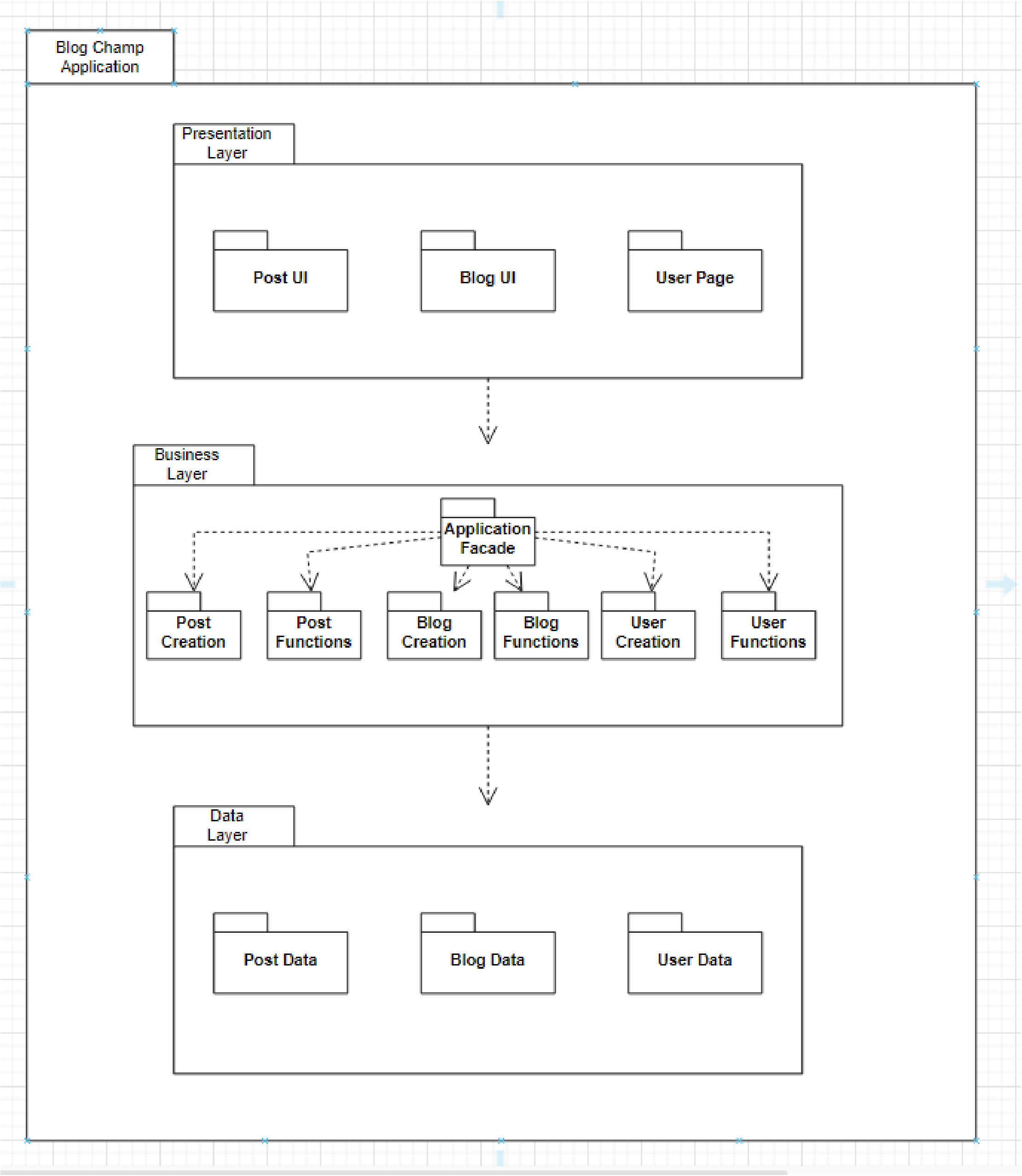
3.2.5 Scenarios View



4.1 Database Design

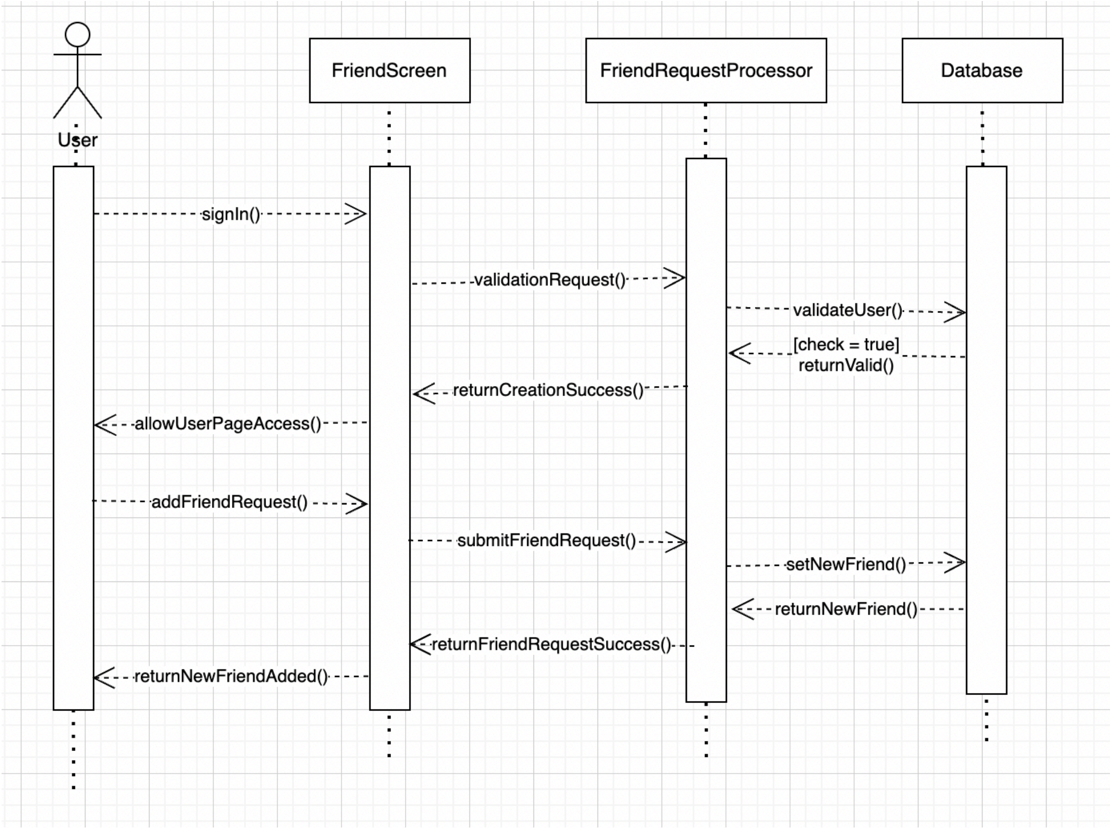


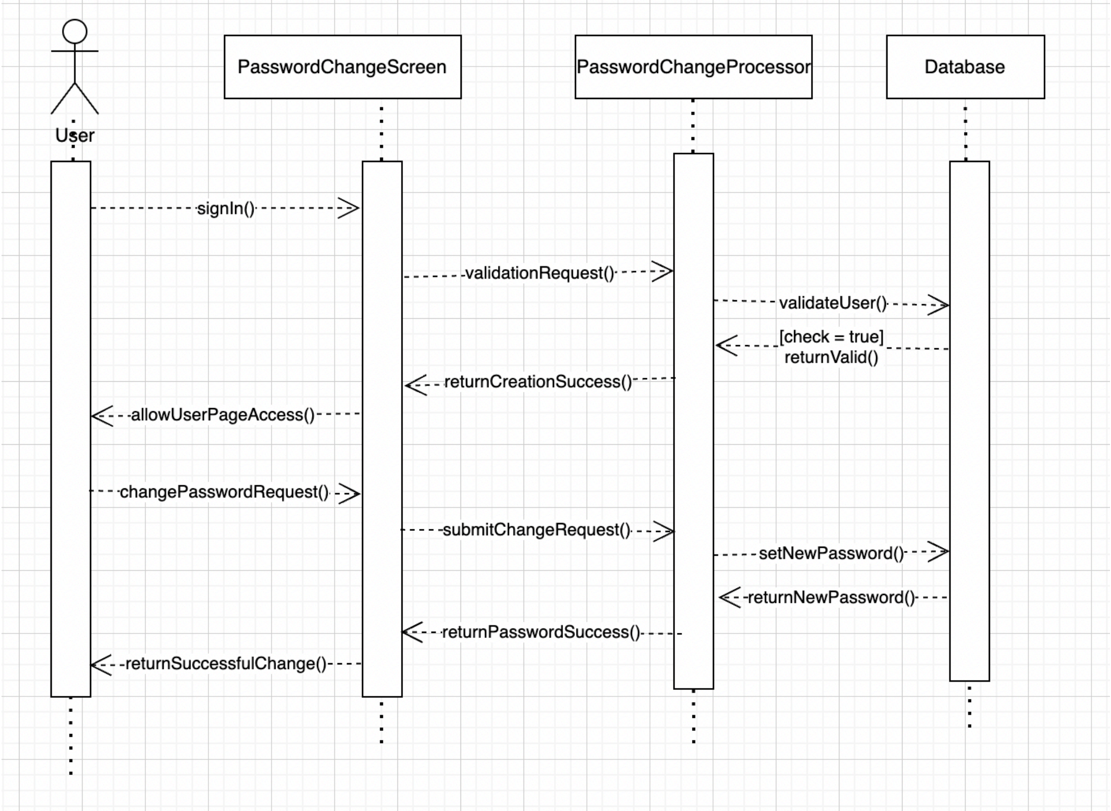
4.2 Package/Module Diagram of the system - front/end sides

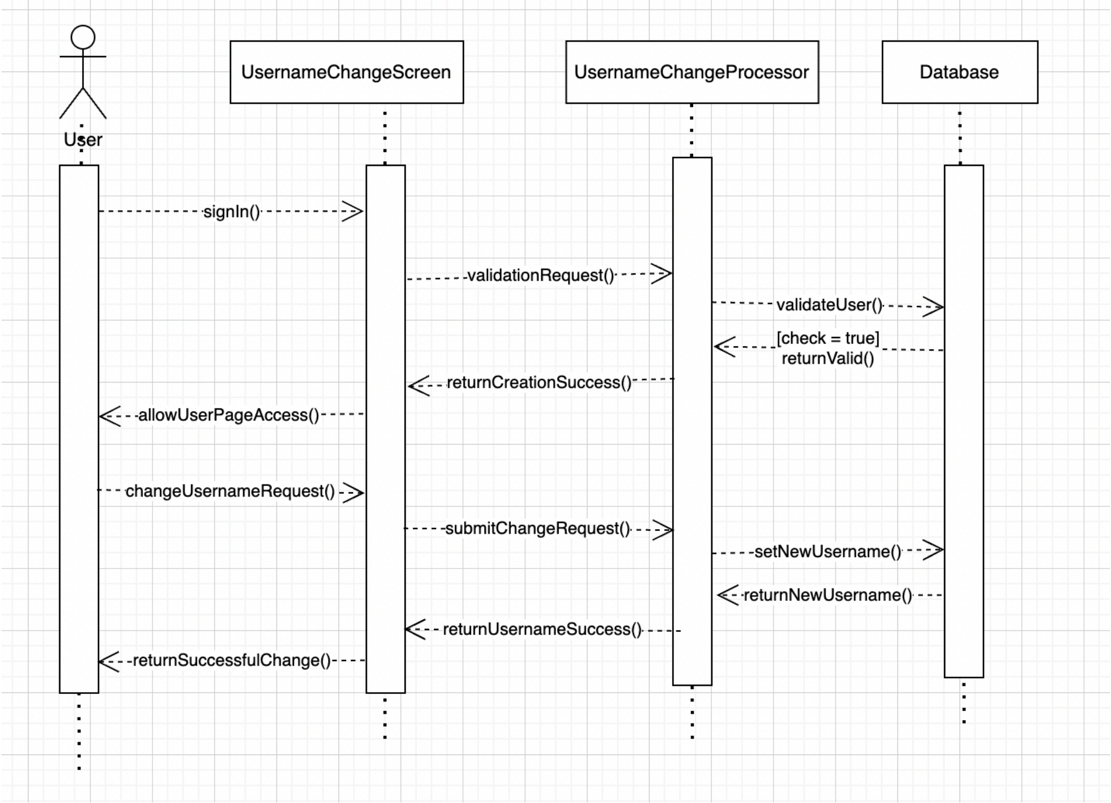


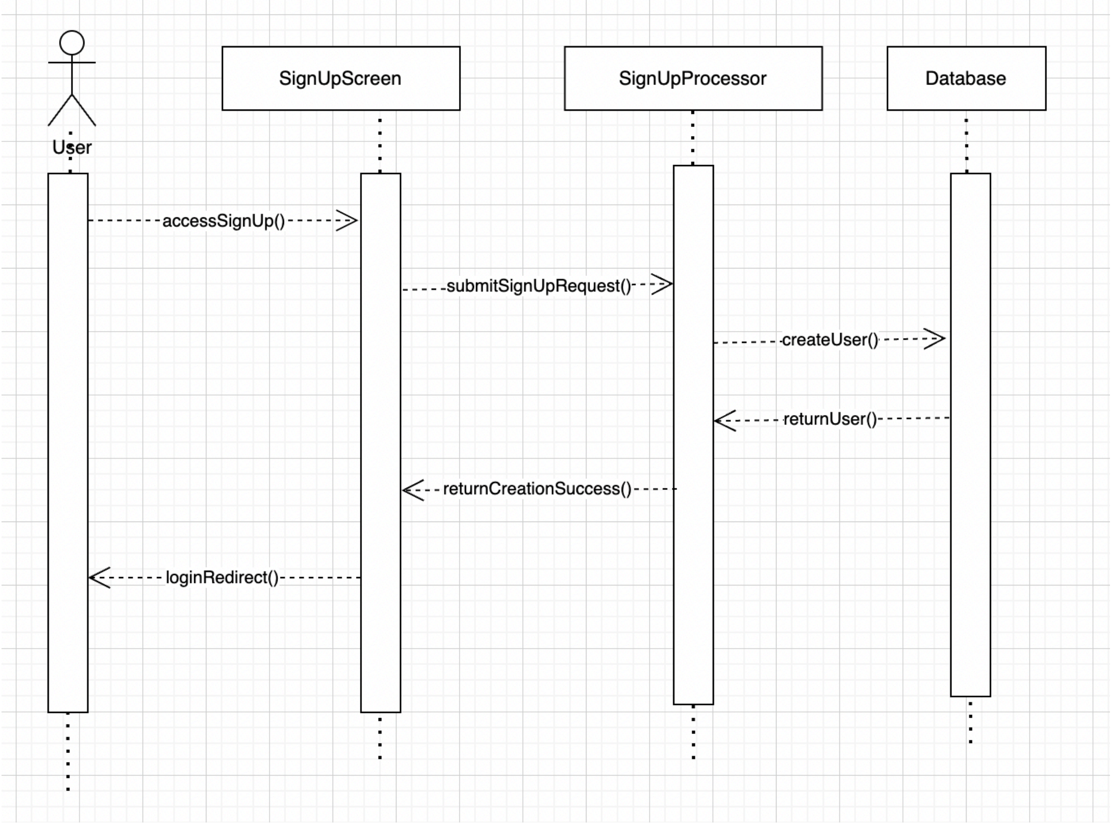
4.3 Detailed Design

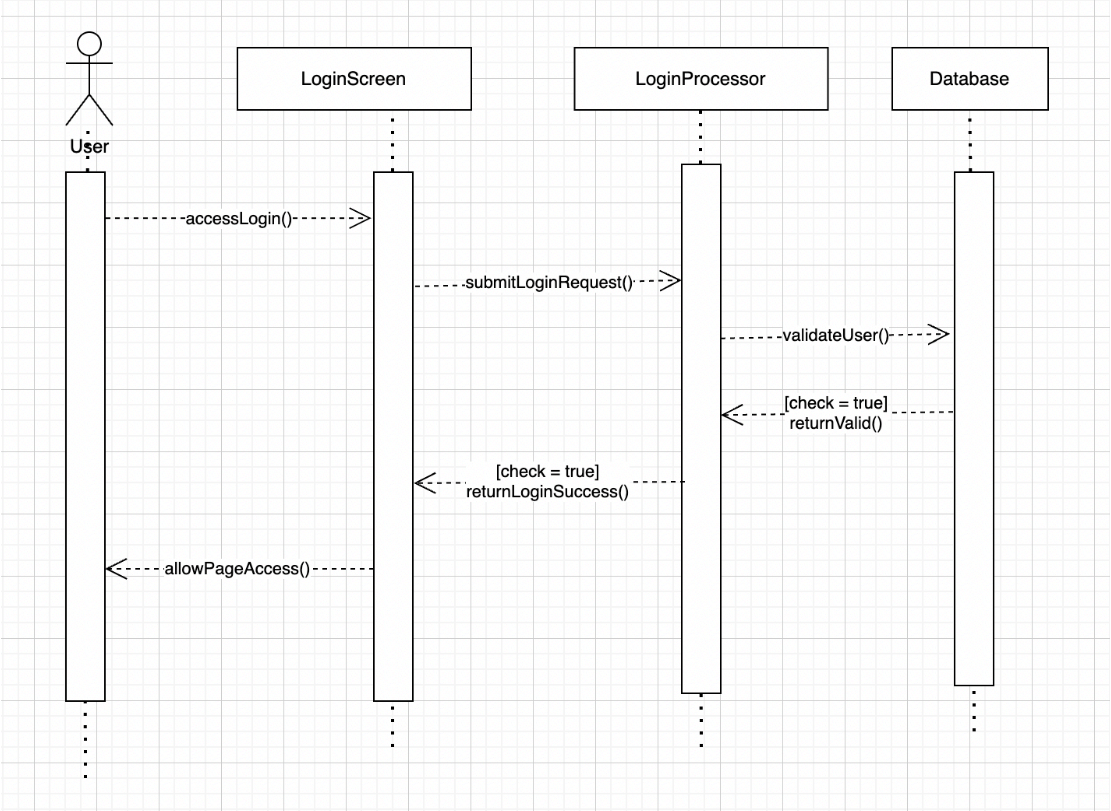
User subsystem:

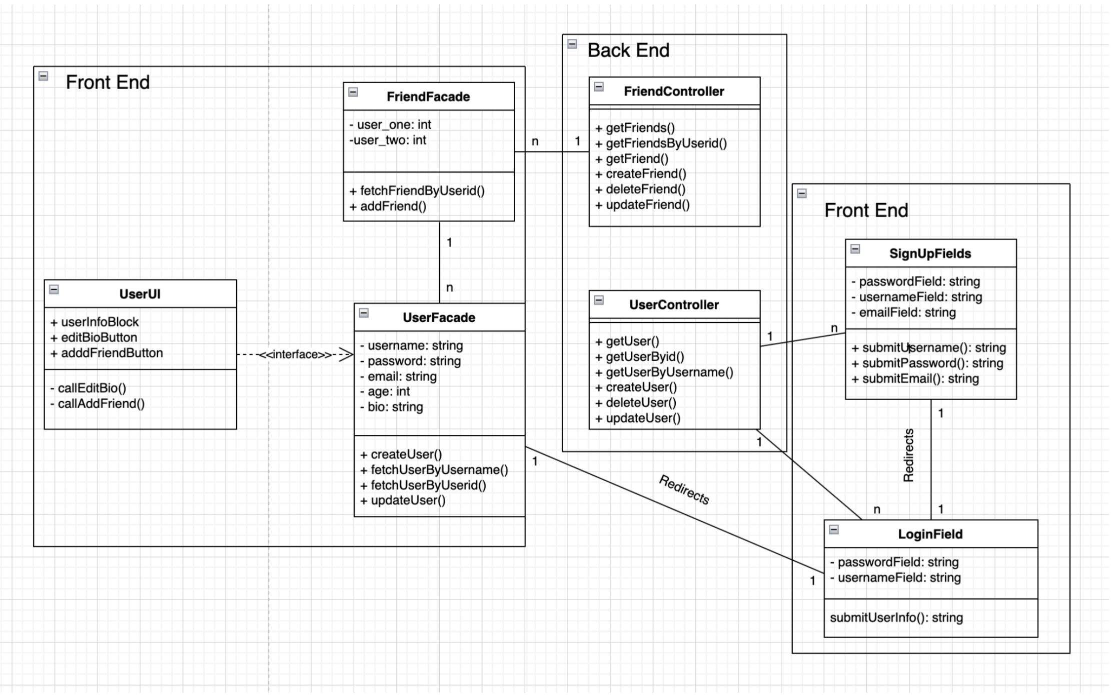



User
PasswordChangeScreen
PasswordChangeProcessor
Database
signIn()
validationRequest()
validateUser()
[check = true]
returnValid()
returnCreationSuccess()
allowUserPageAccess()
changePasswordRequest()
submitChangeRequest()
setNewPassword()
returnNewPassword()
returnPasswordSuccess()
returnSuccessfulChange()


User
UsernameChangeScreen
UsernameChangeProcessor
Database
signIn()
validationRequest()
validateUser()
[check = true]
returnValid()
returnCreationSuccess()
allowUserPageAccess()
changeUsernameRequest()
submitChangeRequest()
setNewUsername()
returnNewUsername()
returnUsernameSuccess()
returnSuccessfulChange()

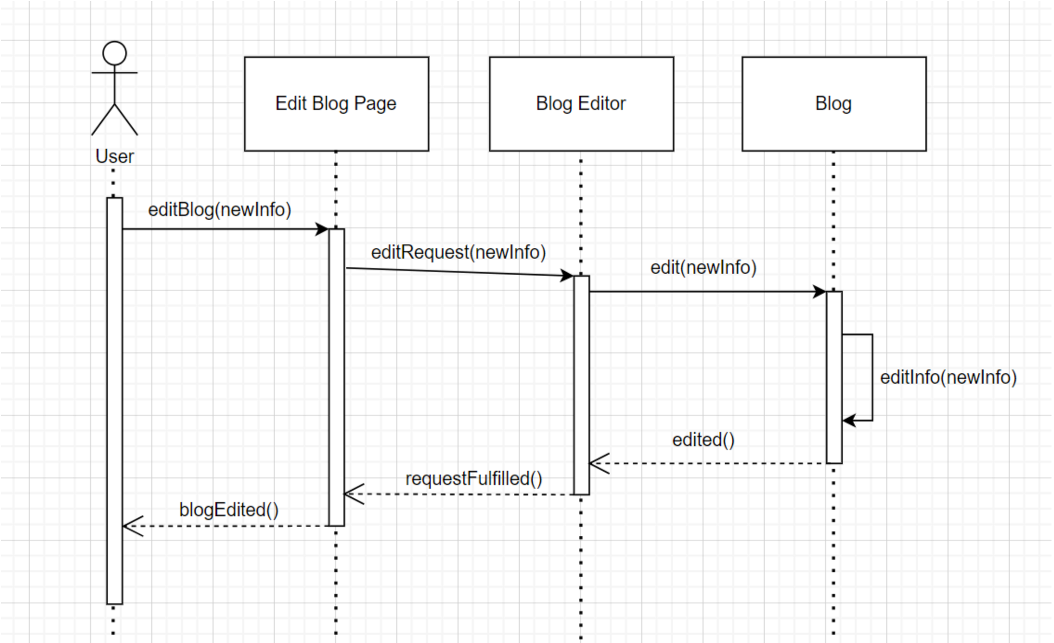

User
SignUpScreen
SignUpProcessor
Database
accessSignUp()
submitSignUpRequest()
createUser()
returnUser()
returnCreationSuccess()
loginRedirect()

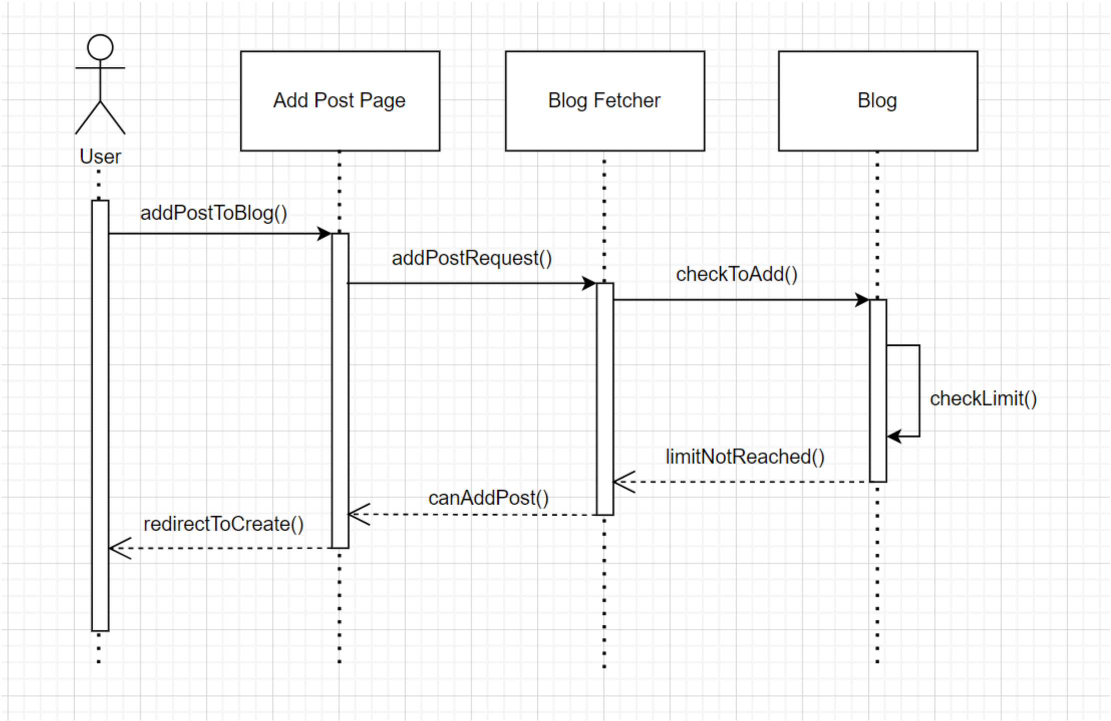

User
LoginScreen
LoginProcessor
Database
accessLogin()
submitLoginRequest()
validateUser()
[check = true]
returnValid()
[check = true]
returnLoginSuccess()
allowPageAccess()

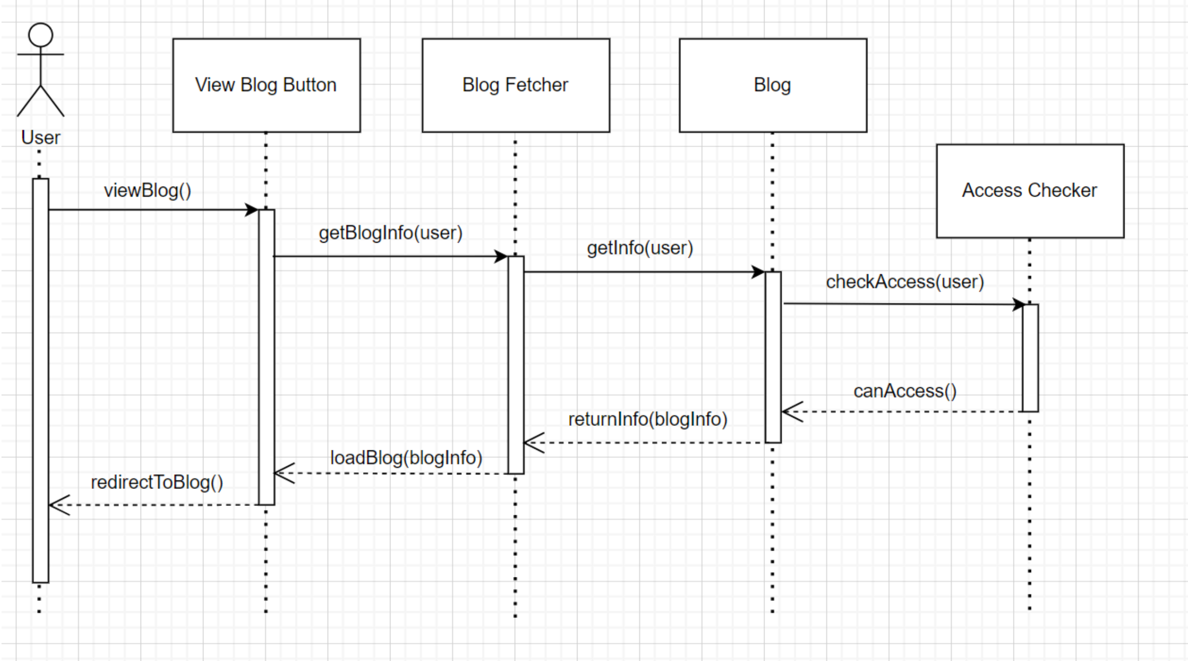


The subsystem for the user will have seven classes involved. The user needs the capability to sign up or log in and the first step to that is through the user UI. They will select the login option and if the user does not have a login they will be redirected to the sign up page that contains the sign up fields. This is where the user will be able to input their accounts username and password to sign up and create an account. The password fields class will then access the UserController class in order to check if the username, password, and email are valid before submitting them to the database. If the user does have an account they will be able to login with their username and password again through the UserController. This class will access the user database to confirm the username and password before getting access to the web page. The user will also need to be able to change their information which is the updateUser part of the UserController class. Lastly the user class itself holds the functionality to add another user as a friend.

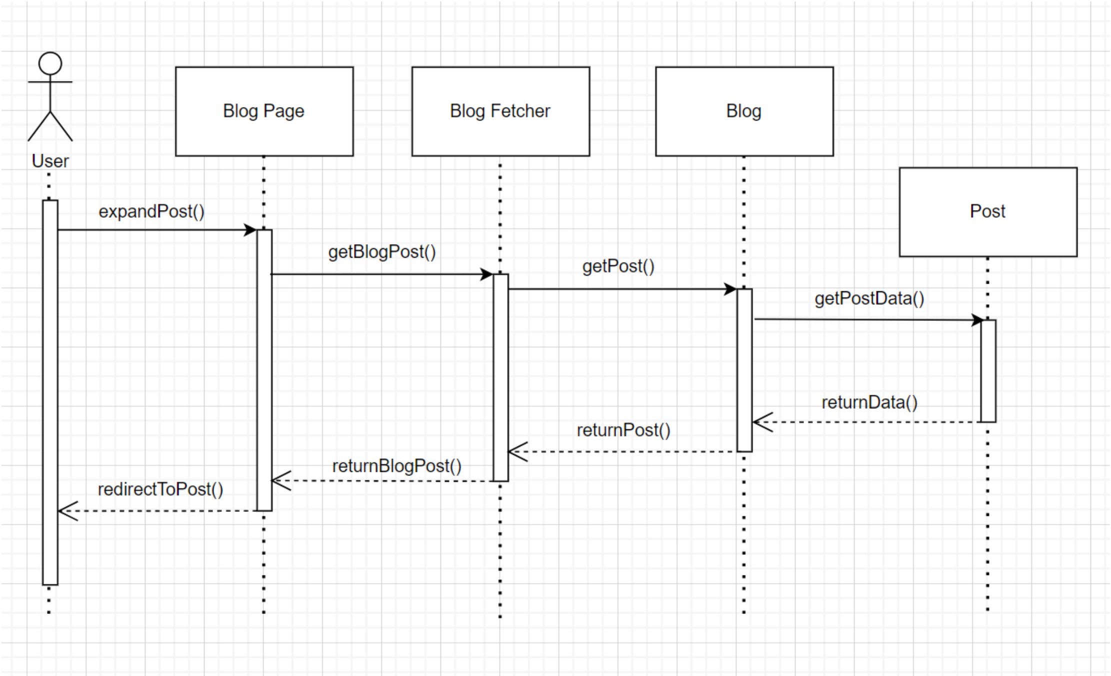
Blog Subsystem:

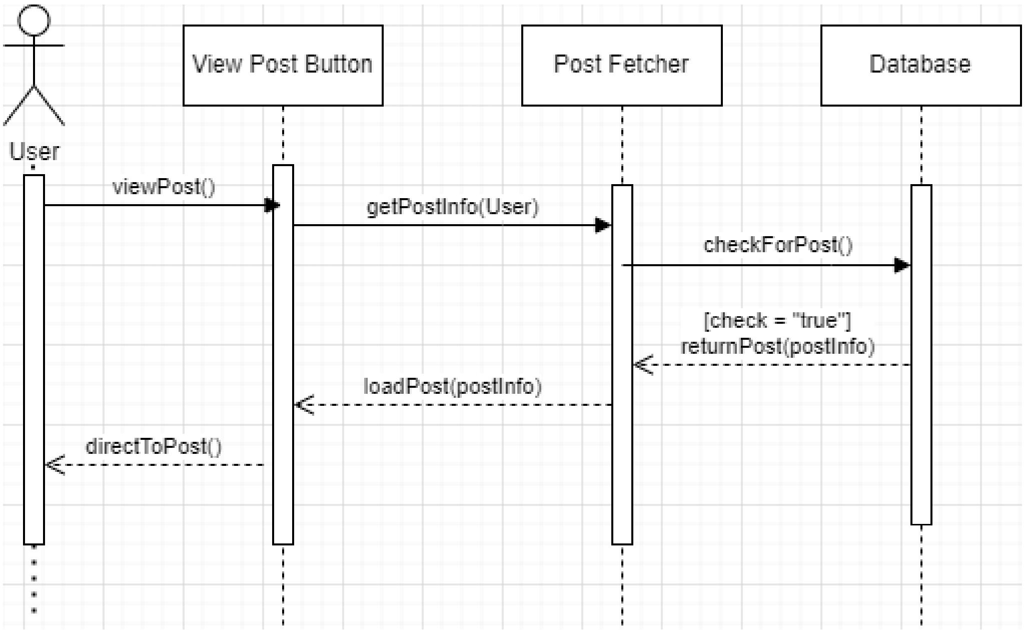


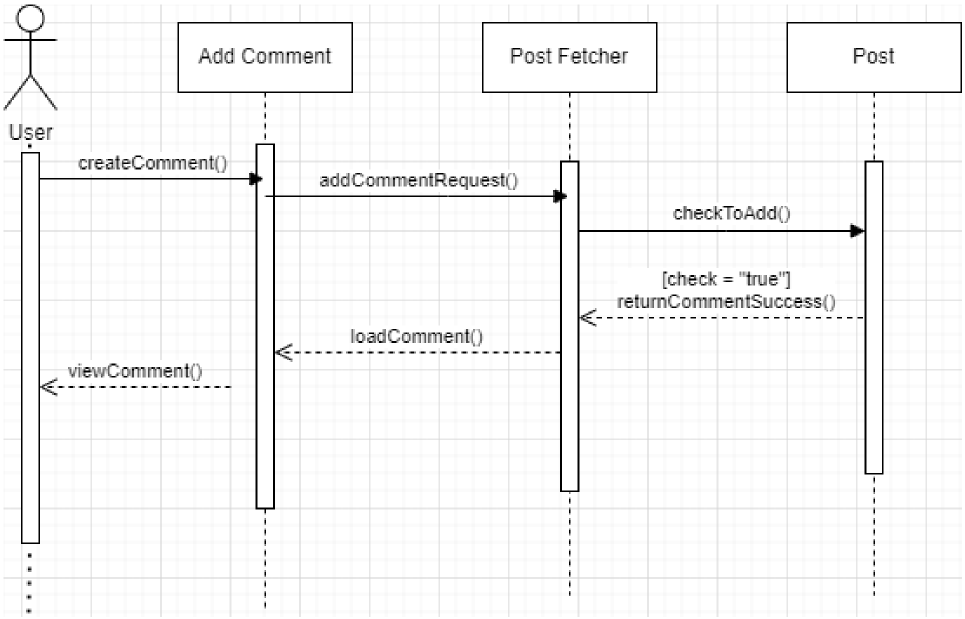

User
Add Post Page
Blog Fetcher
Blog
addPostToBlog()
addPostRequest()
checkToAdd()
checkLimit()
limitNotReached()
canAddPost()
redirectToCreate()

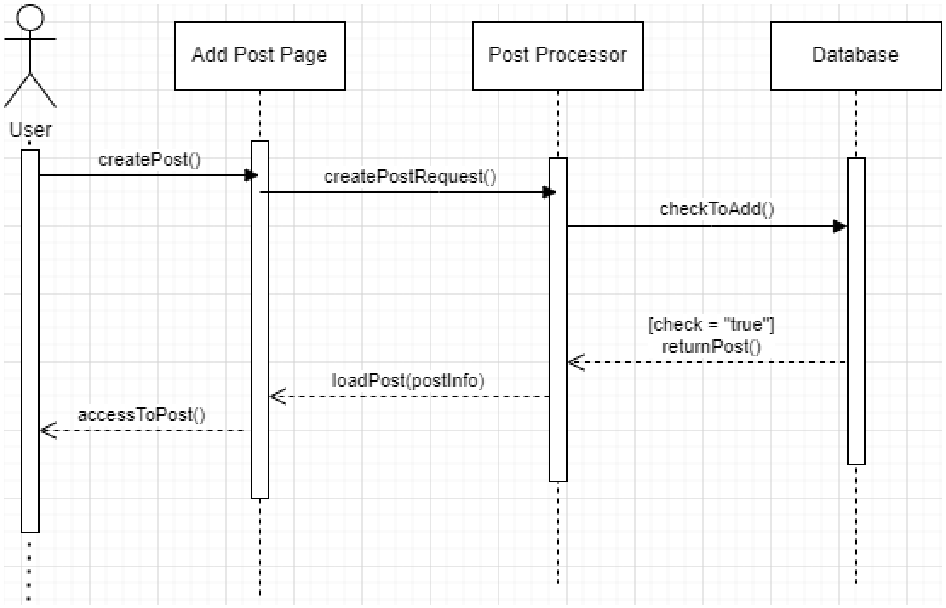

User
View Blog Button
Blog Fetcher
Blog
Access Checker
viewBlog()
getBlogInfo(user)
getInfo(user)
checkAccess(user)
canAccess()
returnInfo(blogInfo)
loadBlog(blogInfo)
redirectToBlog()

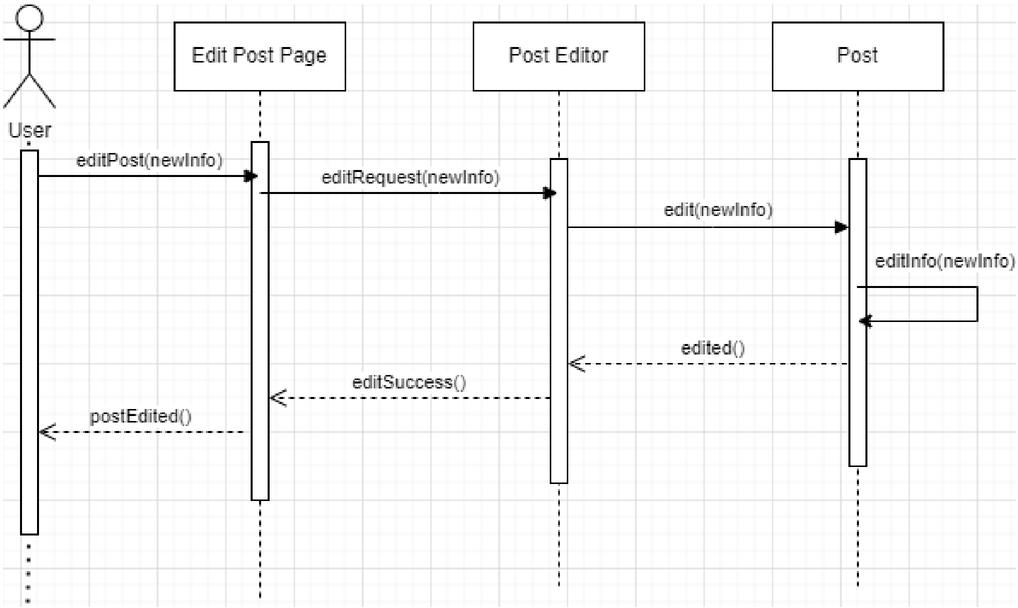
Post Subsystem:



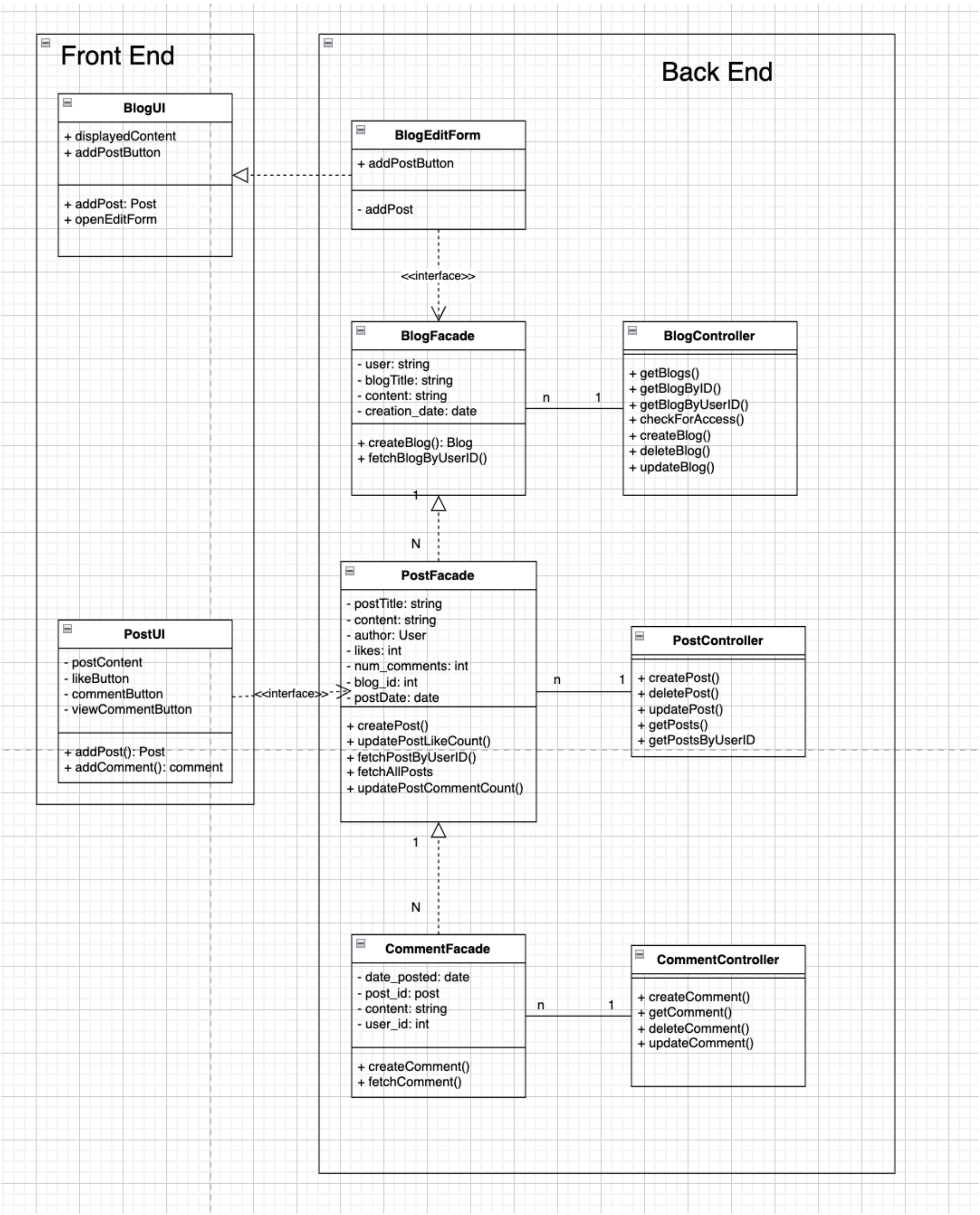



User
Add Comment
Post Fetcher
Post
createComment()
addCommentRequest()
checkToAdd()
[check = "true"]
returnCommentSuccess()
loadComment()
viewComment()


User
Add Post Page
Post Processor
Database
createPost()
createPostRequest()
checkToAdd()
[check = "true"]
returnPost()
loadPost(postInfo)
accessToPost()


User
Edit Post Page
Post Editor
Post
editPost(newInfo)
editRequest(newInfo)
edit(newInfo)
editInfo(newInfo)
edited()
editSuccess()
postEdited()

Blog/Post Class Diagram:



To simplify the relationship between the post and blog subsystem, they have been combined in the above diagram. The blog class contains a title and related posts, which can be mutated by the addPost method which will be managed by the Post Controller class. The user will be able to select from the like button or post content edit option to call the functionalities of the controller. Selecting the edit options calls the PostController class which allows for adding, deleting, liking, commenting, and deleting a comment for posts. The user will then input the information they wish to be updated and the controller will modify it in the database. Blogs are then populated with posts, which contain a title, content, author, likes, comments, and a post date. The separate utility functions in the Post and Blog subsystems enable modularity in the software design by providing extendable classes that enable post and blog creation and editing.

4.4 Design Principle

4.4.1 Single Responsibility Principle (SRP)

In our user class diagram, we separate the utility classes that each user will interact with. For example, we have “SignUpFields” and “LoginFields” classes each with their own utility classes. We chose to separate these classes into multiple classes in order to follow the single responsibility principle, which states that a class should only have a single responsibility or task.

The SRP is important to follow since it ensures our system is focused and easier to maintain, test, and understand. When creating our user class diagram, we started with one large login and signup class, which each handled all of the functionality for logging in and signing up. However, we noticed that this violated the SRP, and had to be broken down into multiple smaller classes.

4.4.2 Open-Closed Principle (OCP)

In our user class diagrams, the post class is separated into smaller subclasses that have utilities that interact with the main post class. These subclasses are extended off the main Post class and use the details from this class to perform individual functionality. This is good for the Open Close Principle because the Post class is open to extension and closed to modification through this set up.

The Open Close Principle is important to follow since it ensures that our system is focused on keeping source information from being altered by external classes which allows for better reliability when it comes to data. This is why we separated these tasks out and allowed them to extend the main class. The goal is to pair this with SRP to allow for optimal data storage and flow to create a reliable website.

4.4.3 Liskov's Substitution Principle (LSP)

In our user class diagrams, the post subsystem is a superclass to the class comments that allows for the comments to inherit different components like comment ID, User ID, Post ID, and post data and be able to use this to help the comment class work. This helps to keep the system flowing and optimal to allow for the comments and posts to work well together.

The Likov’s Substitution Principle is important to follow since it ensures the out system is focused on keeping the main functionality and balance of the post class at its full potential and capacity while dealing with the subclasses that handles the comments fully. This way our system runs efficiently and smoothly for all users that interact with the systems that involve comments and posts.

4.4.4 Dependency-Inversion Principle (DIP)

In our user class diagram, the SignUpFields and LoginFields are not dependent on one another, despite being tied with the use of usernames and passwords. They are associated by the LoginField redirecting to the SignUpField when necessary, but the utilities for each are an extension of themselves respectively. Further, in our post and blog class diagram, the Post and Blog do not depend on one another, with blog being a higher-level since it is made up of many posts. They simply have an association that makes them related but not dependent on each other.

Implementing the DIP, dependencies are not connected to concrete classes, ensuring loose coupling rather than unnecessary tight coupling. The design we have in place, we make sure that our classes can be edited without changing an unrelated principle within our structure, such as the change of a blog utility needing the restructure of the post class as well.

4.4.5 Interface-Segregation Principle (ISP)

In our deployment view and layering diagram, we follow the interface-segregation principle by separating our entire system into distinct layers. This ensures that users and each layer only depends on interfaces that they use and interact with. For example, the user will only see and interact with our presentation layer. This layer handles all of the front end interaction that happens between users, and then communicates any changes that need to be made to the business layer. The business layer then communicates these changes to the data layer, which primarily stores all of the blog, post, and user data.

By following the interface-segregation principle, we are able to prevent large interfaces within our system. Instead, we have multiple smaller interfaces that contain only relevant information for each layer. This also helps promote readability and reduces cohesion between each class and layer. By having low cohesion, we lower the chances of a layer or class breaking the entire system whenever classes get modified.

4.5.1 Facade Pattern (structural pattern)

To properly update and acquire data throughout our application, we will need interfaces that link user actions in the UI with database operations. We are using the facade pattern in our BlogFacade, PostFacade, CommentFacade, and UserFacade classes that allow the front end to both make requests to update the database, as well as pull data from the database to update the UI content. This allows the data to be updated in our frontend components without the need to construct api calls and response checks every time. The post and blog facades also enable communication between the subsystems, as when the user wants to add a post to a blog, this action in the UI is controlled by the blog facade, which then communicates with the post facade to add a post in the database. The facade pattern can be found in the Facade.js file inside the frontend folder (frontend/src/pages).

4.5.2 Singleton Pattern (creational pattern)

In order to properly handle data in our application, we create single instances of facades in order to handle each data structure. These facades contain the functions that handle the major data operations such as creating and updating a user and post, adding a friend, creating a blog, etc. The facades are located in the Facades.js file (frontend/src/pages/Facades.js). Since we only instantiate one instance of each facade, there is always only one global point of access to any objects inside of the facade. By following the singleton pattern, we ensure that our code is efficient by only creating one instance of our facades in addition to promoting modularity by providing a centralized point of the major data functions.

4.5.3 Template Pattern (behavioral pattern)

To properly display and show all the contents of each post in both the post page and individual blog pages we use a template pattern to build each post. This pattern can be found in the Front end folder inside components. It is titled PostDetails.jsx and this file grabs all the data of a given post and formats it to fit the website theme and style. This allows for a fast and snappy feel with each post loading and updating very quickly and smoothly. In this file you can see some functionalities like fetchUser and handleLike that are used to get, update, and display the post data. Finally it returns a build post to be displayed on whatever page is requesting this post. This template is accessed by the Post.jsx file to keep it constantly updating the page with new posts and updated data. The IndividualBlog.jsx page does a similar process of using this template to update posts contained in a user's blog. Both of those files are located in the frontend/src/pages directory.