

**Solvinery**

**ADD**

# Contents

<b>Preface.....</b>	<b>3</b>
Glossary.....	3
<b>Chapter 1 Use Cases.....</b>	<b>4</b>
<b>Chapter 2 System Architecture.....</b>	<b>12</b>
The Client Architecture.....	12
The Server Architecture.....	13
<b>Chapter 3 Data Model.....</b>	<b>14</b>
<b>Chapter 4 Behavioral Analysis.....</b>	<b>18</b>
4.1 Sequence Diagrams.....	18
4.2 Events.....	20
4.3 States.....	20
<b>Chapter 5 Object-Oriented Analysis.....</b>	<b>20</b>
5.1 Classes Diagram.....	20
5.2 Class Description.....	20
5.3 Packages.....	22
5.4 Unit Tests.....	23
<b>Chapter 6 User Interface Draft.....</b>	<b>23</b>
<b>Chapter 7 Testing.....</b>	<b>33</b>

# Preface

## Glossary

### Zimpl

Zimpl (Zuse Institut Mathematical Programming Language) is a high level, lightweight programming language designed to model mathematical problems into a linear or nonlinear mixed integer mathematical program expressed in .lp or .mps file format which can be read and (maybe) solved by a LP or MIP solver. Zimpl supports multiple solvers and is well integrated with the Scip solver, who's designed by the same German organization.

### Scip

Scip (Solving Constraint Integer Programs) is a solver for LP, MIP and mixed integer nonlinear programming (MINLP), and is one of the quickest FOSS solver frameworks available, designed for absolute control of the solution process and access detailed information deep inside the solution.

### Kafka

Kafka is an event stream platform designed for publishing events, storing them for however needed and processing them through the method of subscription. Kafka's functionality is provided in a distributed, highly scalable, elastic, fault-tolerant, and secure manner.

In our project, we'll use Kafka on a smaller scale. In bigger projects it's designed to have a cluster of brokers, each responsible for managing event streaming (thus providing easy scalability and distributed deployment, and fault tolerance if a broker fails)- but we'll use a single broker, for simplicity reasons.

# Chapter 1 Use Cases

## 1. Create image from text

Description: Given Zimpl code, create a new image in the server representing it. If the code is valid and compiles, move to the next screen. Else, return an error message.

Preconditions:

- The server is running, and able to communicate with the current client app.
- The user is logged in.

Actors: Advanced User, Zimpl interpreter.

Postconditions:

- The user is logged in.
- If the code compiles, the server now has a new image belonging to the user and a model matching the code, inside said image. Else, the server state remains unchanged.

Course of Action: The code text is sent to the server. The server verifies it compiles using the Zimpl interpreter. If it does, it parses the code into a model matching the code. Else, it shows an error message given by the interpreter.

## 2. Create image from path

Description: Given path to a text file, read and load the file and create a new image in the server representing it. If the code is valid and compiles, move to the next screen. Else, return an error message.

Preconditions:

- The server is running, and able to communicate with the current client app.
- The user is logged in.
- Given path is valid in the user's file system.

Actors: Advanced User, Zimpl interpreter

Postconditions:

- The user is logged in.

- If the code compiles, the server now has a new image belonging to the user and a model matching the code, inside said image. Else, the server state remains unchanged.

Course of Action: The client app reads the file from the path and loads it into the text box. If file reading fails for any reason (invalid path, permissions, etc.) show an error message. The user can then send it to the server or edit it. The code text is sent to the server. The server verifies it compiles using the Zimpl interpreter. If it does, it parses the code into a model matching the code. Else, it shows an error message given by the interpreter.

### 3. Choose variables

Description: After valid image creation, The user can choose which variables will be part of the solution. For each variable chosen, the user can choose sets and parameters with which this variable is constructed from (if any exist) and choose which of these are mutable and visible, or immutable and invisible, essentially constants within the problem, from the perspective of the basic user.

Preconditions:

- The user is logged in.
- If the code compiles, the server now has a new image belonging to the user and a model matching the code, inside said image. Else, the server state remains unchanged.
- The server contains a valid image which the client app is working on.

Actors: Advanced User.

Postconditions: The server accepted the choice and has saved it as part of the image configuration.

Course of Action:

### 4. Define constraints

Description: After choosing variables, the user is able to define constraints. Meaning he can create constraints' modules, in which he selects constraints to be part of, gives the module a name and description.

Preconditions:

- The user is logged in.
- The server contains a valid image which the client app is working on

Actors: Advanced User.

Postconditions: The server accepted the choice and has saved it as part of the image configuration.

Course of Action: The GUI shows the user list of available constraints. The user creates a new module and gives it a name. Then the user gives it a description, and chooses which of the available constraints to link to the module. The GUI then shows the dependent sets and params to the constraints, now being dependent to the module.

## **5. Define preferences**

Description: After defining constraints, the user is able to define preferences. Meaning he can create preferences' modules, in which he selects preferences to be part of, gives the module a name and description.

Preconditions:

- The user is logged in.
- The server contains a valid image which the client app is working on

Actors: Advanced User.

Postconditions: The server accepted the choice and has saved it as part of the image configuration.

Course of Action: The GUI shows the user list of available preferences. The user creates a new module and gives it a name. Then the user gives it a description, and chooses which of the available preferences to link to the module. The GUI then shows the dependent sets and params to the preferences, now being dependent to the module.

## **6. Alias Sets/Params**

Description: for each set and param that are part of a module, they can be assigned custom names and optionally descriptions. Their original names (Those given in zimpl code) will be hidden from the basic user. If no alias then names will be extracted from zimpl code directly

Preconditions:

- The user is logged in.
- The server is running, and able to communicate with the current client app.

- The user's server session contains an image.
- The user's image contains a module, with at least one param or set in it.

Actors: Advanced user

Postconditions:

- The user is logged in.
- The server contains a valid image which the client app is working on.
- A valid module, with at least one param or set in it.
- The module has at least one set or param aliased.

Course of Action: The user chooses a module he has created, and within it navigates to the option of renaming sets or prams. Simple data can be renamed directly, while complex data (such as a list of pairs) needs to have some or all of its parts renamed individually. The user confirms his choice, and the server updates the image accordingly, after verifying the names are legal.

## 7. Control preferences bar

Description: A preference can be assigned to the bar. All preferences will be assigned to the same bar, and their locations will have values normalized according to other preferences, ranging from most to least important compared to others on the bar.

Preconditions:

- The user is logged in.
- The server contains a valid image which the client app is working on. (In this case a valid image contains a valid zimpl program, which has to have exactly one max/min function which isn't empty, from which we define our preferences)

Actors: Basic User.

Postconditions:

- The user is logged in.
- The server contains a valid image which the client app is working on.
- At least one preference weight has been adjusted at the server.

Course of Action: The user drags draggable shapes around the bar which represent different preferences. The server gets automatically updated.

## **8. Edit parameter/set values**

Description: Given sets and parameters, the user can assign values to them. Functionality will be present to help manage data in sets and parameters, e.g. automatically inserting a list of the current month's dates, make a parameter always be the current date, etc.

Preconditions:

- The server is running, and able to communicate with the current client app.
- The user is logged in.
- The module has at least one set or param aliased.

Actors: Basic User.

Postconditions:

- The server is running, and able to communicate with the current client app.
- The user is logged in.
- The module has at least one set or param aliased.
- The server receives values for the parameters and sets.

Course of Action: In the final screen, the available sets and parameters are shown. The user can (for each set or parameter) insert a value. The user sends the values to the server and it is stored there.

## **9. Solve problem (Send the problem to the engine to be solved)**

Description: Given the defined variables, constraints' modules, preferences' modules, parameters and set values, the user can send the above data to the server and should be able to receive a solution to its problem, which satisfies the data that he defined earlier.

Preconditions:

- The user is logged in.
- The server contains a valid image which the client app is working on.
- The user defined variables, constraints' modules, preferences' modules.
- The user supplied values to sets and parameters.

Actors: Basic user. Scip engine, Zimpl interpreter



Postconditions: The user should be able to see the solution that was received by the server.

Course of Action: Given the problem's definition and data, and a timeout choice, the user selects the button to solve the problem, and awaits the solution. When the problem is solved the screen is switched to that of the solution screen.

## **10. Register**

Description: A basic user can register to the system, adding a new account to it with his information.

Preconditions:

- The server is running, and able to communicate with the current client app.
- The user is not logged in.

Actors: Basic User.

Postconditions: The server has a new account, defined to be basic, and with the information that the user supplied.

Course of Action: The user goes to the registration page. There he has to supply a valid email address and a password. If the email is not taken yet, a new account is created, otherwise the user has to choose another email address.

## **11. Log in**

Description: A basic user can log in to the system. By doing so he is able to perform new actions.

Preconditions:

- The server is running, and able to communicate with the current client app.
- The user is not logged in.

Actors: Basic user.

Postconditions: The user is logged in.

Course of Action: The user goes to the logging in page. The user types the email address and password that he registered with. In case the credentials

are correct the user is logged in. Otherwise, the user has to correct the data that he typed.

## **12. Log out**

Description: A logged in user can log out from the system. By doing so he is able to register or log into a new account.

Preconditions:

- The server is running, and able to communicate with the current client app.
- The user is logged in.

Actors: Basic User.

Postconditions: The user is now logged out.

Course of Action: The user clicks on the logout button, and he is directed to the home page.

## **13. Publish Image**

Description: Given an image the current user has in his sessions, he has the option to publish it to other users of the system. A published image's name and description will be visible to other users, and they can download a copy of the image to use or edit as they see fit.

Preconditions:

- The user is logged in.
- The server is running, and able to communicate with the current client app.
- The user's server session contains an image.

Actors: Advanced user

Postconditions: The image is published, and the number of published images in the server grows by one.

Course of Action: Upon selecting the option to publish an image, the user received a confirmation prompt with a preview of the image (for now its name and description, maybe some other yet to be decided info)- after the user confirms the image is sent to the server and is fetched next time a user wishes to view published images.

## **14. View Images**

Description: The user can view a list of all published images. The view option will fetch a set amount of images (either at random or via some criteria), and the viewer will load more images using pages or dynamically as the user scrolls down. Each image will show its name, description and publish date.

Preconditions:

- The user is logged in.
- The server is running, and able to communicate with the current client app.

Actors: Advanced User, Simple User (?)

Postconditions: No state changes.

Course of Action: The user selects the option to view, the view screen changes to the list of images.

## 15. Search Image

Description: In the view images screen, the user can look up images via a search bar, filtering by name.

Preconditions:

- The user is logged in.
- The user is in Images view screen

Actors: Advanced User, Basic User (?)

Postconditions: image list now shows only images containing looked up string

Course of Action: The user types his input in the search field, then selects the filter option. The list then updates according to his query.

## 16. Load Image

Description: In the images view screen, the user can select an image to load into his own session. All aspects of the image are loaded, including its problem model (the one defined by Zimpl code), all its modules, aliases, etc.

Preconditions:

- The user is logged in.
- The user is in Images view screen

Actors: Advanced User, Basic User

Postconditions: A new image is added to the user's session.

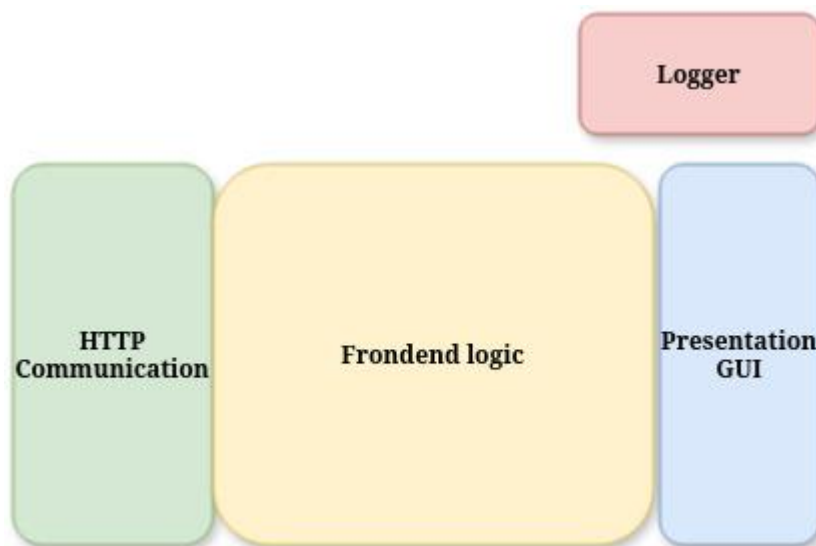
Course of Action: The user finds the image he wants, and selects the load option on it. The server fetches the image, makes a copy of it, assigns it to the user and adds it to his session. The user can now view the image in his images section

# Chapter 2 System Architecture

## The Client Architecture

Clients is a React based web client, accessed from a public domain yet to be acquired. Communication with the server will be through HTTP requests with json encoding/decoding. The system will support both remote local and remote solver options- to provide flexibility in testing and running the solver . By default, the client will communicate with our server when it is run, but an option will be given to set a custom address for it to communicate with (which will be localhost if the server is run locally).

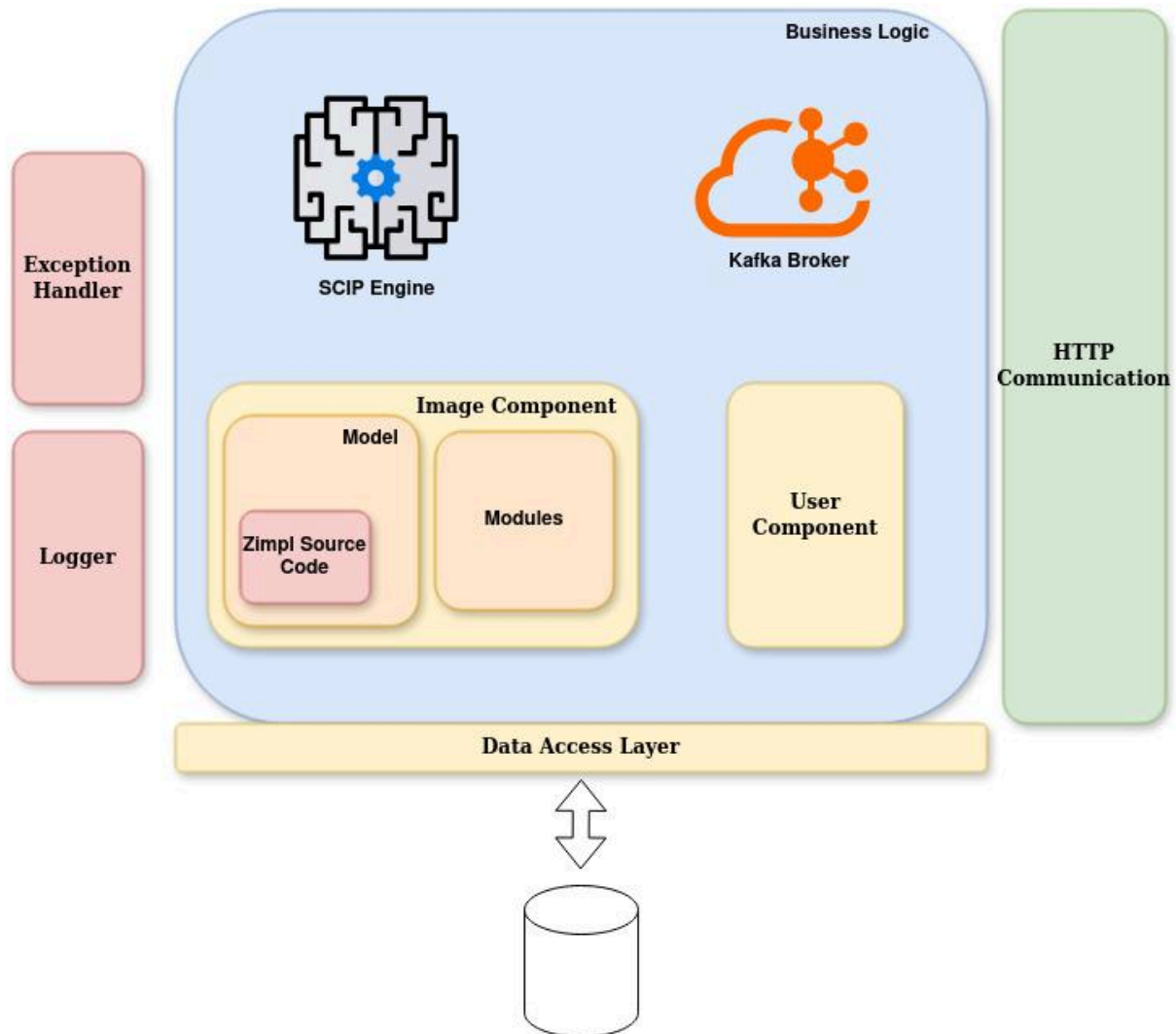
The remote server will be restricted to some maximum timeout for solving client problems, which override the client's set timeout if it's smaller then it. This is to prevent overloading the server, intentionally or not. The timeout may be adjusted manually or automatically according to server load.



- **HTTP Communication**- An API layer responsible for decoding and encoding json I/O, and internal objects.
- **Frontend Logic**- the JS code responsible for the core functionality of the client app.
- **Presentation GUI**- The CSS code responsible for designing a proper GUI interface for the client to use.
- **Logger**- The logger component monitoring the client side actions. Will have its own logging methods, and will share data with the server periodically or as needed (on error), for ease of client side debugging.

## The Server Architecture

The server will be built according to the Modular Monolith Architecture, a monolithic program with its core design focus being modularity between its different components:



The architecture focuses on modularity between different parts of the system, limited communication to pure API calls between different parts of it. Communication will be direct API calls for light operations, and asynchronous events driven for heavier or repetitive operations, such as logging or operations with the engine. Asynchronous events will be streamed between services using [Kafka](#).

Below is an explanation of the diagram:

- HTTP Communication includes the API gateway and all java objects and JSON encoding/decoding classes.
- Logger- Will be implemented asynchronously utilizing Kafka and no other additional libraries.

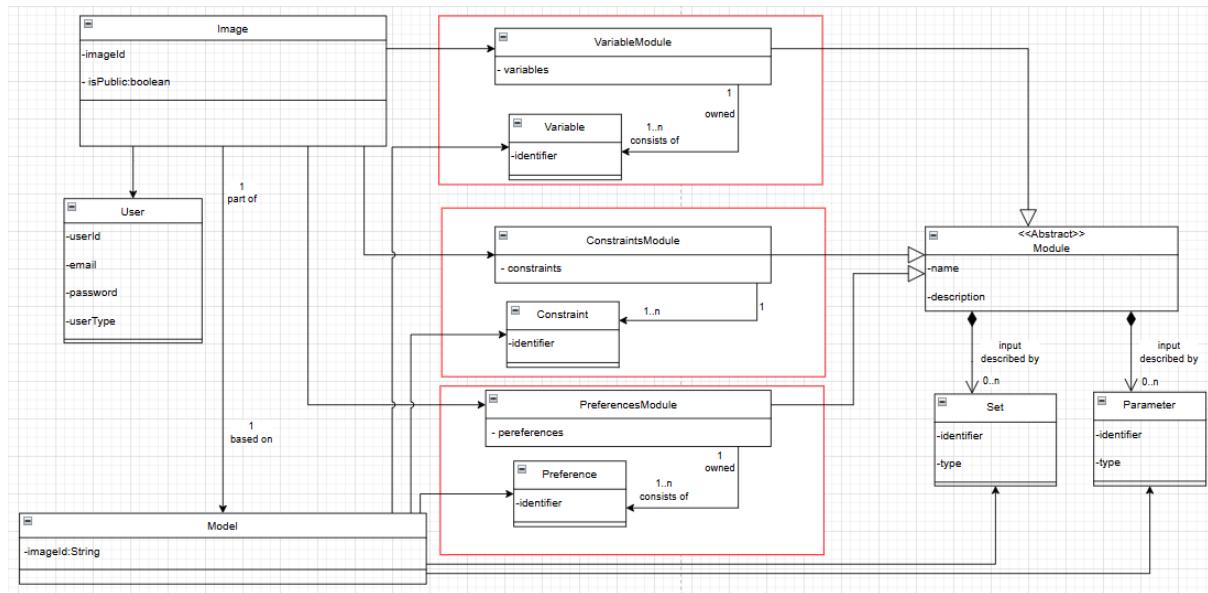
- Exception handler- The fallback classes which build exception messages depending on the exceptions, and log them.
- Data Access Layer- The API for database calls. Communication between it and the rest of the software will be done through DAO objects.
- For the database itself, we'll use PostgreSQL, accessed through Spring Data JPA (Java Persistence API), a Spring wrapper for Hibernate.
- Kafka Broker- The broker managing event queues between producers and consumers. Only a single one is expected to be used, although the library is designed to be scalable if needed.
- SCIP Engine- the engine that receives Zimpl code as input and output solutions. Multiple will be utilized at the same time, and will be called through Bash scripts.

The main Business Logic is separated into 2 components:

- User Component is responsible for all user operations, including managing user data and authentication.
- Image Component is responsible for all the user's image operations, including solving them. Its main components include:
  - The model, which defines the actual problem parsed from the Zimpl source file, and holds objects for each field of the problem, e.g. variables, constraints and preferences.
  - Modules, with defined grouped constraints, preferences or variables that allow abstraction of the problem Model which is fit for users uninformed of how the system actually works.
  - The Zimpl source code is a code part of an image, and resides inside the model. any changes in the problem definition of the data it works with will, eventually, be written in that file. It is also the data sent to the SCIP engine when solving the problem.

# Chapter 3 Data Model

For our application we will use a postgresSQL database, which works together with spring. The following diagram shows the relationships between the classes that represent data in the application:



A description follows:

**Model** - This object holds a reference to the file specifying the Image's mathematical code. All fields this object uses, are derived from the file. The main fields it parses from the file are: parameters, sets, variables, constraints, and an objective function (which is divided into preferences - each term in the objective function as if it is a polynomial). All of these properties are parsed dynamically, the zpl file is the only value being stored persistently. This class holds its `imageId` which is also the primary key for the relevant document in the key-value store.

**Image** - This object describes a meta structure over the raw **Model**. It defines groupings of constraints, preferences and variables that are in the **Model**. It holds a reference to its corresponding **Model** and **User** (owner).

**ConstraintsModule** - This object groups together 'raw' constraints from the zpl file to form a composite, use-case oriented, constraint. It has a name and a description of what this module is about.

**PreferencesModule** - This object groups together 'raw' preferences from the zpl file to form a composite, use-case oriented, preference. It has a name and a description of what this module is about.



VariablesModule - This module assembles and describes the meaningful variables that present in the Model. Not all variables are meaningful for the user of the image, some are used only for intermediary calculations.

Set, Parameter - Represented by an identifier and a type, which is the input form for that Set or Parameter. Sets and Parameters describe the inputs for a particular module.

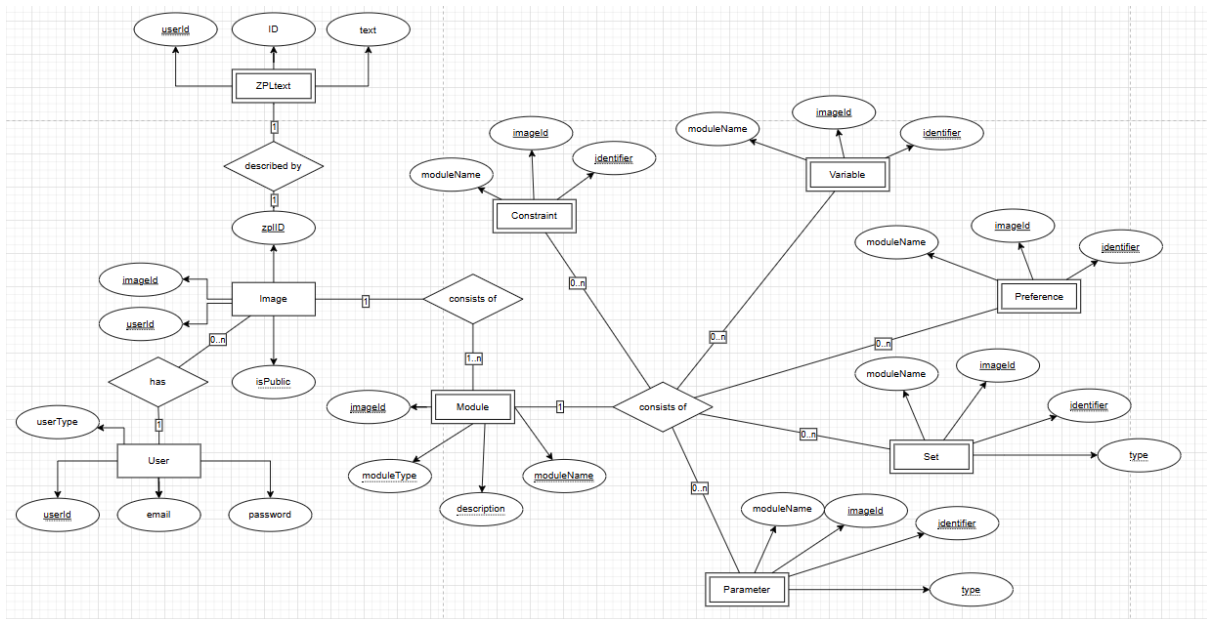
Constraint, Preference, Variable - Represent the 'atoms' of ConstraintsModule/PreferenceModule/VariablesModule respectively.

User - This object describes the data about the users in our system. It will contain identification fields such as email password and type (advanced/basic).

ZPLtext - The zpl file that any model is built from will be stored in our database. This will be used both to rebuild images if the system crashes, and to show the zpl file for users who want to use other users' images.

Some of the relationships are enforced at the database, while some are enforced asynchronously, as these are cross-database relationships. For example, each Image has modules, which are stored in one database. When an image's module is being modified, a transaction is carried out adhering to all ACID principles. On the other hand, Image holds a reference to its owner (user) and Model (described by a zpl File), which are maintained by different services on different databases. Therefore the reference is maintained asynchronously, meaning, for example, if the user deletes its own account (assuming the application will support such functionality, currently it doesn't), all of his proprietary images still reference the deleted user. The image will figure out their owner is deleted only when they need to access it, leading to eventual consistency.

Next follow the ERD and the Tables diagrams



## Transactions:

**Register** - The request will be sent to the Keycloak service which will handle the request while complying with ACID principles.

**Creation of an Image** - The request will arrive to the Images service, it will send the .zpl file to the solver service, and expect it to save the file in its document database, with the key being the imageId. The Image service will create a default (empty) image in its database, by creating a tuple in Images table only (no modules) with the imageId, the requesting user, and isPublic=false. The lock for the added row in Images will be held until the solving service will confirm a successful write in his DB.

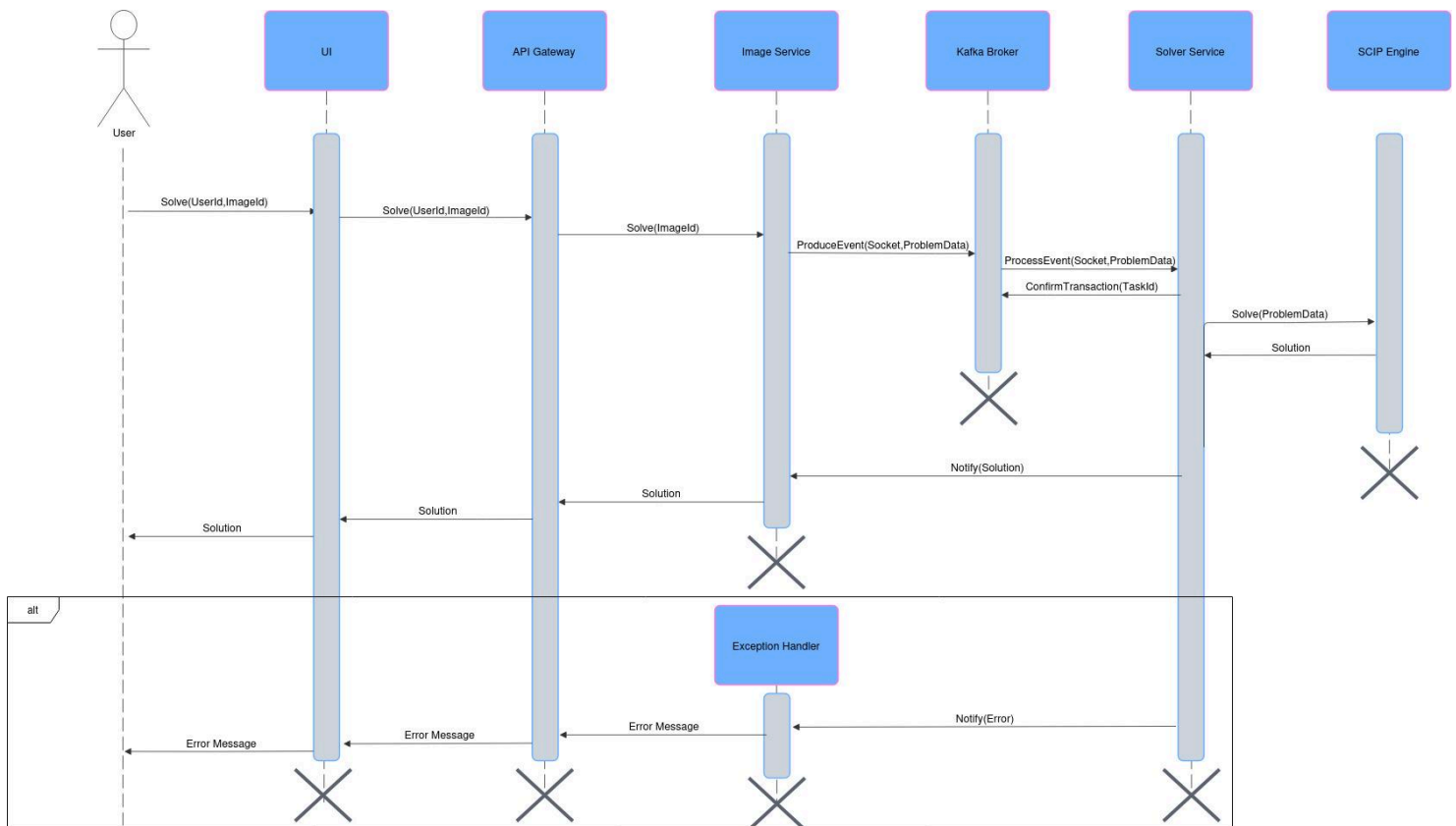
**Searching an Image** - A read will be issued on the entire Images table, looking for an image that has the required keywords in its name or description that were prompted. Isolation level will be set to "Committed Read" for this transaction.

**Solve** - The solver service will read the file with the imageId provided in the request.

# Chapter 4 Behavioral Analysis

## 4.1 Sequence Diagrams

Solve Diagram:

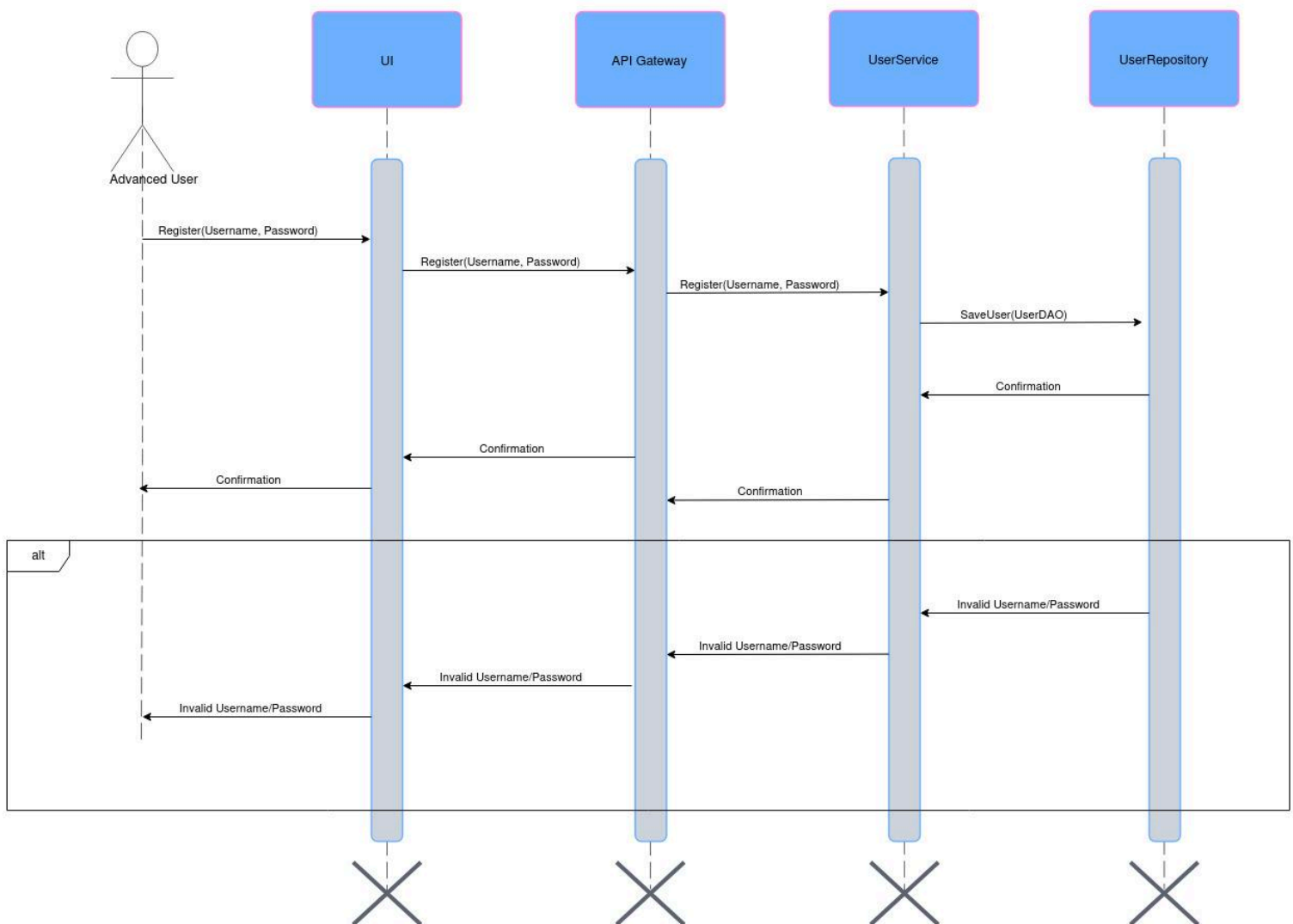


Step by step explanation:

1. The client selects the option to solve in the frontend UI.
2. The frontend UI sends the request to the server with the appropriate user and image information.
3. The gateway parses the request and forwards the request to the Image Service.
4. The image Service fetches the image from the repository, validates it and assembles the Zimpl code corresponding to it.
5. The code representing the problem is sent to the Kafka Broker.
6. Solver Service, through a transaction (A tool provided by Kafka for thread safe operations) changes the solve event status to taken.
7. A Solve Service, one of many, accepts the event from the broker.
8. The service executes the solving process on an SCIP engine instance.
9. The solution or an error message is then propagated back to the user.

10. If an error occurs at any stage, it's passed to the exception handler, who formats an error message to show to the users, logs the error and propagates it to the UI.

### Register Diagram:

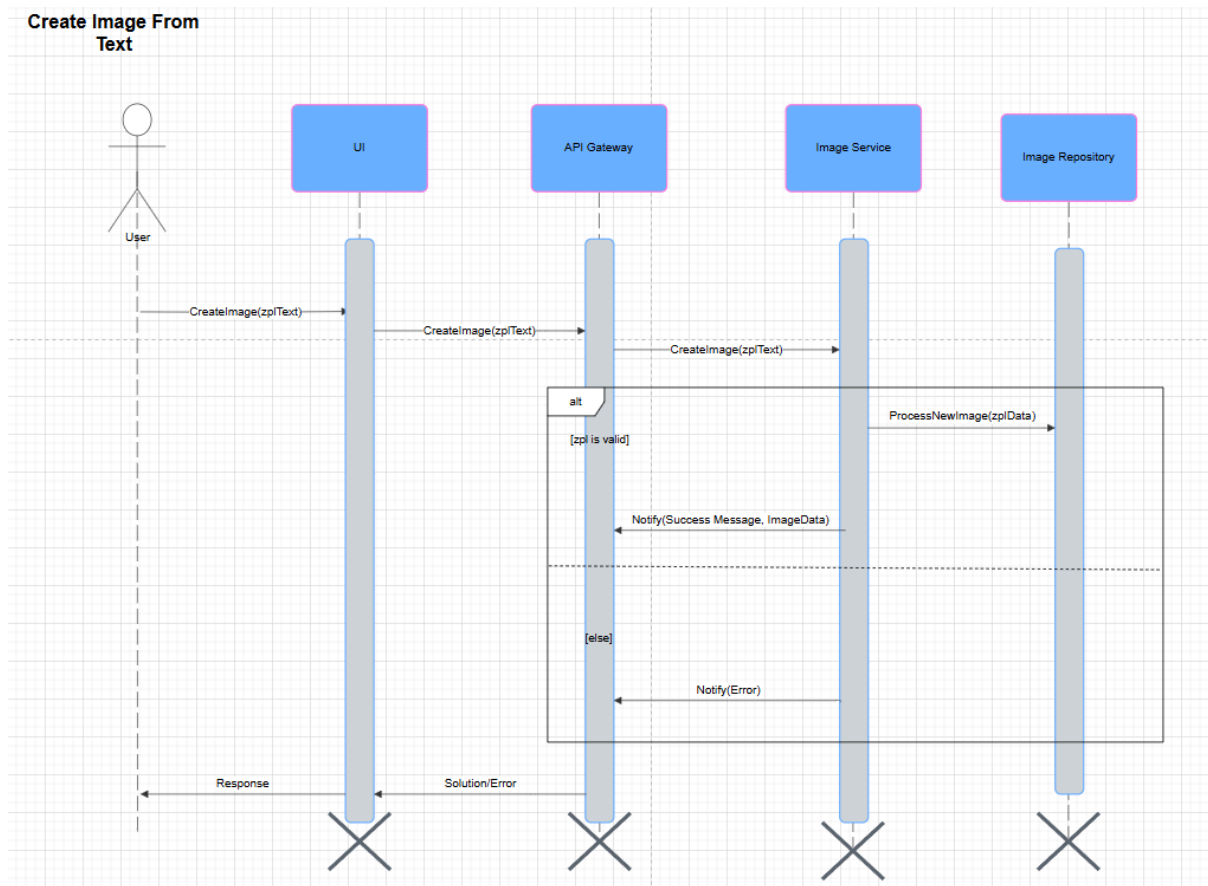


### Step by step explanation:

1. The user inserts his desired username and password into the UI, and clicks on the register button.
2. The frontend GUI sends the request to the server.
3. The API gateway parses the request and sends it to the User Service.

4. User Service validates the request, creates a UserDAO, and calls UserRepository with it.
5. UserRepository inserts a new user entry to the Users database. Data validation occurs at this stage as well, with database parameters.
6. A confirmation or failure propagates back to the end user.

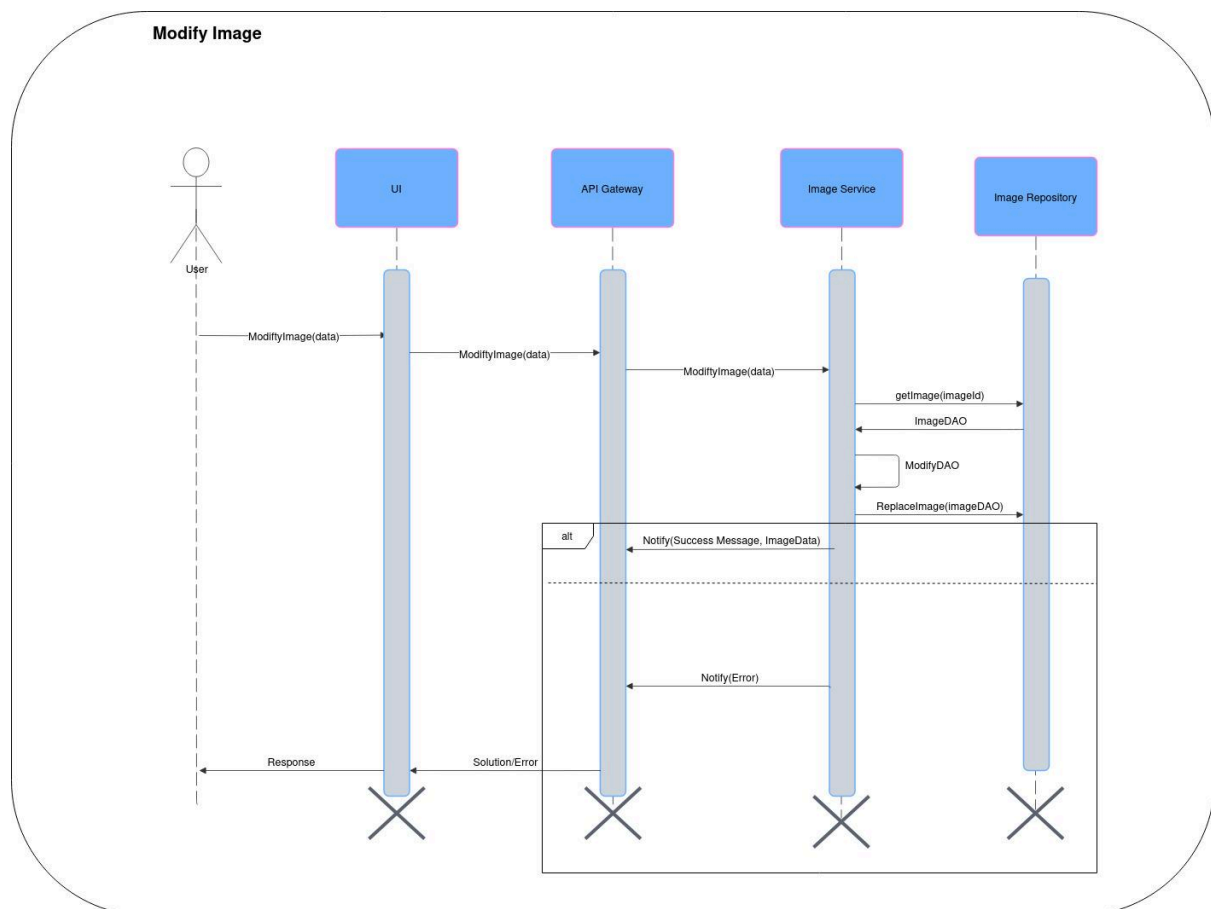
### CreateImage Diagram :



### Step by step explanation:

1. The user inserts his ZPL Text into the UI, and clicks on the register button.
2. The frontend GUI sends the request to the server.
3. The API gateway parses the request and sends it to the Image Service.
4. Image Service validates the request, creates an ImageDAO, and calls ImageRepository with it.
5. ImageRepository inserts a new image entry to the Image database. Data validation occurs at this stage as well, with database parameters.
6. A confirmation or failure propagates back to the end user.

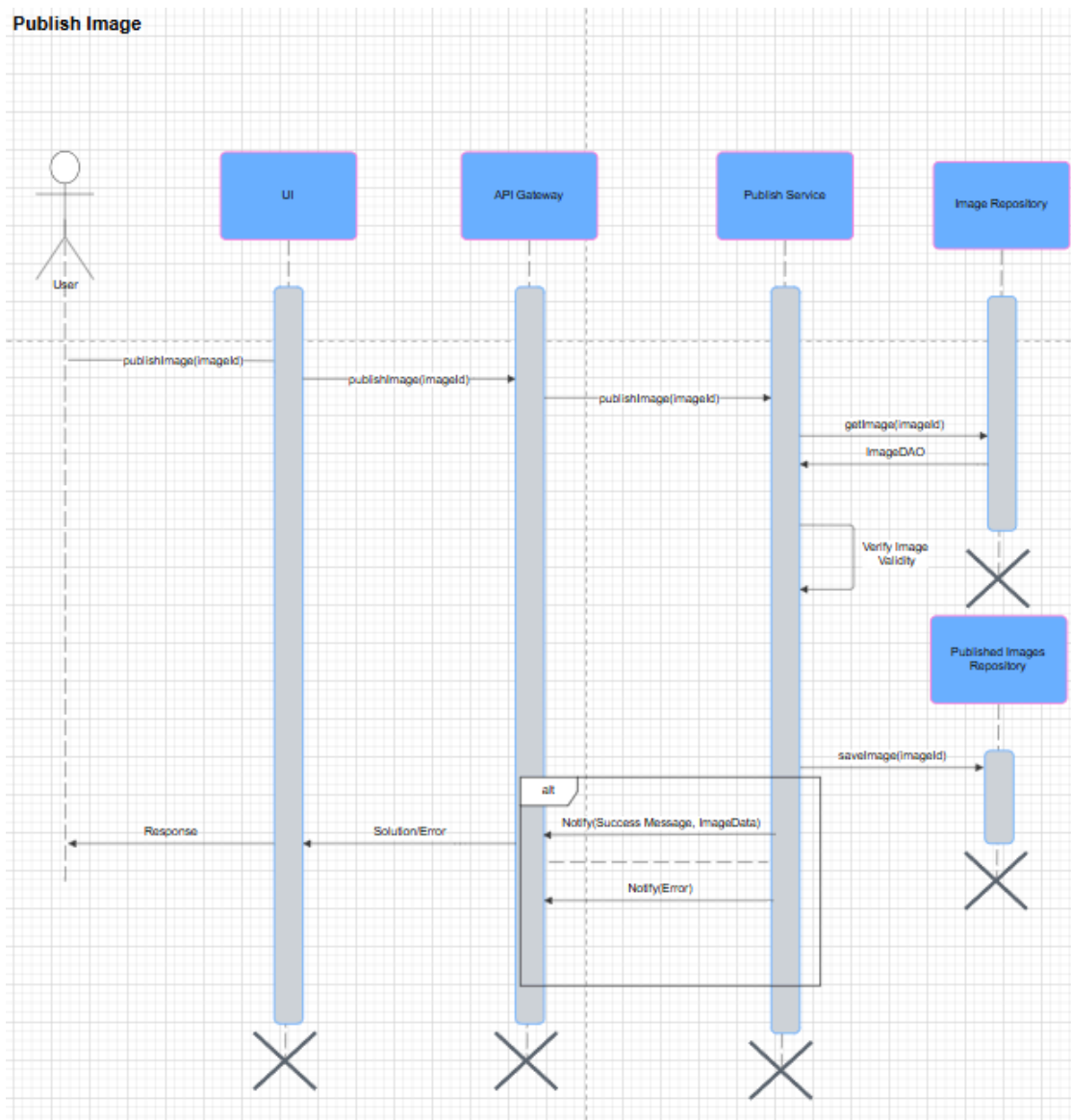
## Modify Image:



### Step by step explanation:

1. The user modifies some data in the image, and selects the option to save it.
2. The image is sent to the publishing service.
3. The service queries the image repository for the image object.
4. The changes and the images pass validation, and a DAO of the modified image is created.
5. The is saved again with the same ID, overriding the old image
6. A success message is propagated to the user.
7. If at any point an error occurs, an error message propagates to the user.

## Publish Image :



Step by step explanation:

1. The user is in the "MyImages" page, after selecting a specific image.
2. The user clicks on the "publish" button in the UI.
3. The frontend GUI sends the request to the server.
4. The API gateway sends the request to the Publish service.
5. The Publish service takes the Image's DAO from the Image Repository.
6. The service validates the image, its existence, and that it's properly configured.
7. The service sends the DAO to the Published Images Repository, so it'll be saved there.
8. Lastly, the Publish Service sends back a response (success message and the Image's data if everything was fine, or error message else).

## 4.2 Events

A user may issue several commands which may be separated into event types:

- The user may mutate data relating to himself, by changing his own user data or doing an action to one of the images that belong to him, such as adding a constraint module.
- The user may publish an image to the general user base, making it appear when searching for an image.
- The user may solve a problem defined in an image, which invokes the SCIP engine and returns a solution, if one is found.

## 4.3 States

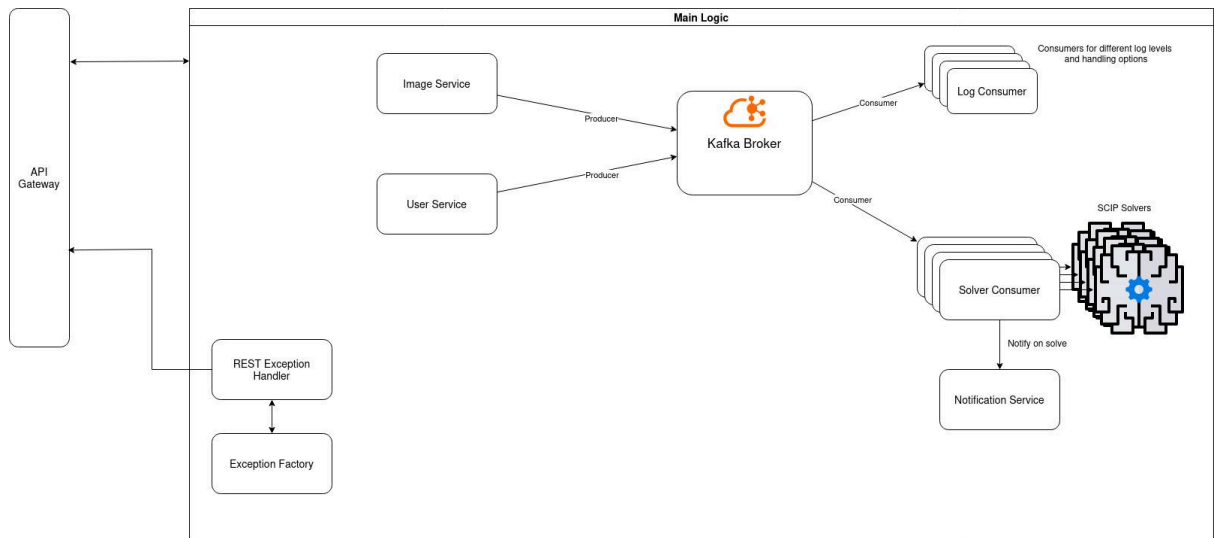
Our system is for the designed to be almost stateless, user requests arrive with all the data needed to process them (in part identifiers to database entries), only some limited data will be kept on RAM- a user to timestamp map to track timeouts for inactive users, and reject their request to demand a fresh login if their timeout expires. A server restart would incur a deletion of that map, and force all users to relogin.



# Chapter 5 Object-Oriented Analysis

## 5.1 Classes Diagram

Our project is and will be build and the classical layered fashion:



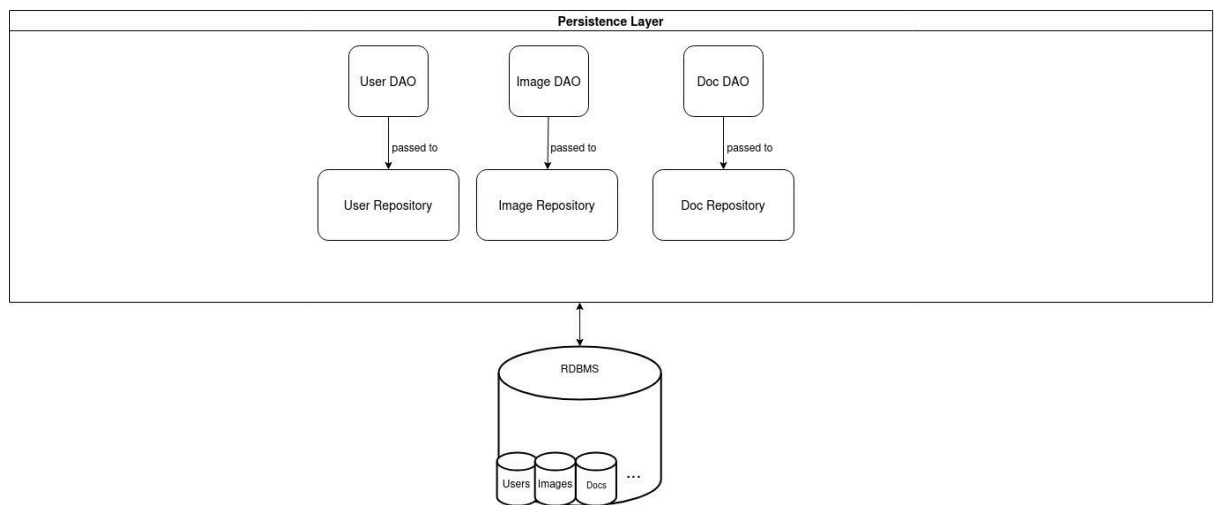
## 5.2 Class Description

In general, the diagram does not attempt to be comprehensive but present the general idea of how the system will look at its core:

1. API Gateway - The top most layer of the server will be a RESTful API, utilizing Spring. Data transferring between the incoming data and the server is done purely with DTOs, implemented as java records. complex data is made with DTOs using other DTOs as fields, all of which is converted into json format using Sprint's default parser Jackson. DTO creation is abstracted from the rest of the project and is centralized in a RecordFactory, featuring many static methods converting inner classes into DTOs.
2. Most of the core logic will happen in Service classes, separated by responsibility. Image Service and User Service are examples of it. All service classes are Spring services, meaning their instances are managed through the Spring container. They are, in essence, singletons, with optimistic synchronization with most of the concurrency concerned delegated to Database restrictions or handled using Kafka's tools for it.
3. The main heavy operations of using the SCIP engine to solve problems will be done only in async using Kafka's broker. Image Service will produce events to solve a problem, with the events containing the problem itself and all its surrounding data such as user data. A Solver Service will take the event, process it by sending it to the engine, and return the result either through

Kafka with an additional event, or a websocket, whose data it got from the solve event it got to process.

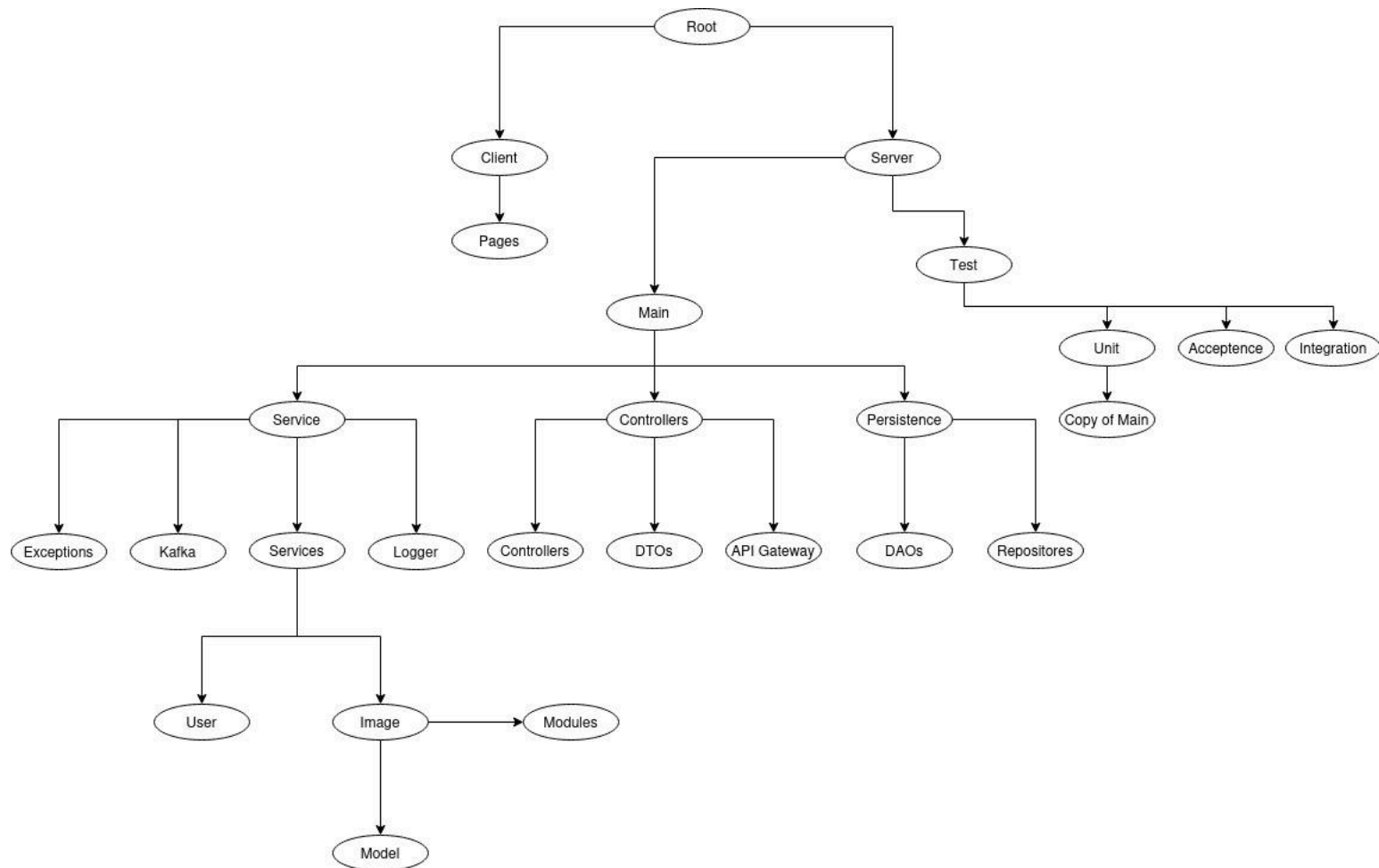
4. Logging will be done through Kafka- Any object can send log events to Kafka, and those events will be process by different Log Consumers, several of those consumers will exist, each with the task of processing different log events, or several to process the same log event differently (i.e print to console and save to file).
5. All exceptions will be handled via a centralized exception handling factory and handlers. Different exception types will exist for different user or software errors, all caught, and either attempt a recovery or gracefully pass them to a factory converting the exception to a DTO with a specialized message to the user, which is then sent to the client.



6. Persistence will be done using Spring Data JPA- operations will be done through Repository classes, which are spring tools to abstract away database operations, one will exist for each table. Operation data and table definition will exist in entity DAO classes, which are created outside the layer and passed to it via the API it publishes.

## 5.3 Packages

Predicted Package Tree:

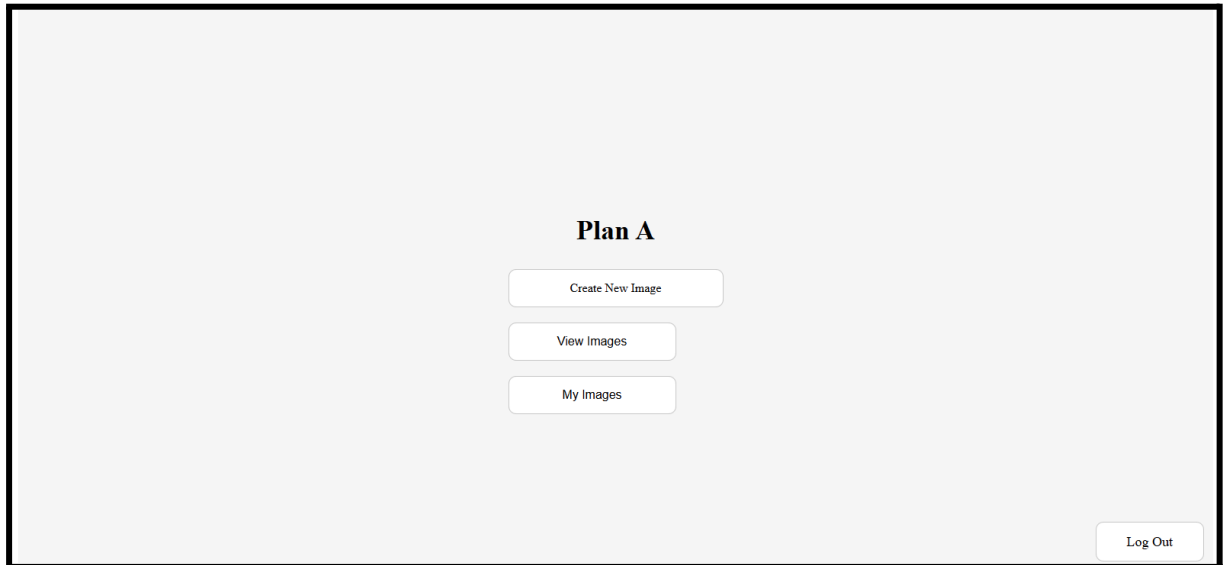


## 5.4 Unit Tests

Each class under the main package will have a corresponding class under the Test package with the same path, that includes unit tests for said class. That includes DTO and DAO classes, since they include data validation logic. Specific implementation techniques will be expanded on in the Testing chapter.

## Chapter 6 User Interface Draft

- MainPage



A wireframe for the MainPage. It features a light gray background with a black border. In the center, the text "Plan A" is displayed in bold. Below it are three vertically stacked buttons: "Create New Image", "View Images", and "My Images". In the bottom right corner, there is a "Log Out" button.

**Plan A**

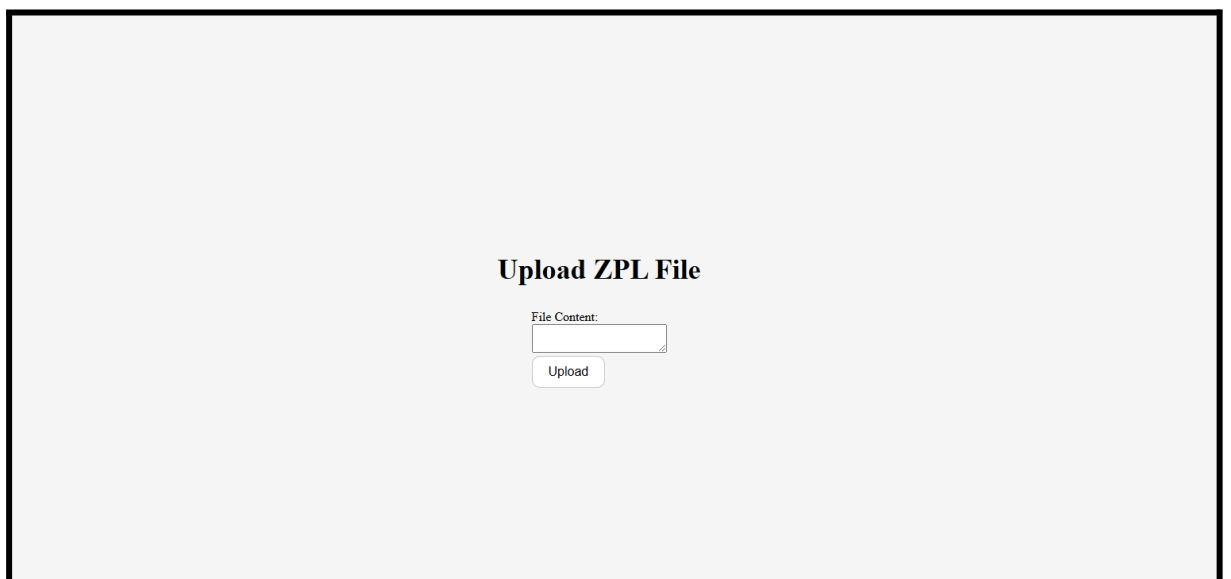
Create New Image

View Images

My Images

Log Out

- UploadZPLPage



A wireframe for the UploadZPLPage. It features a light gray background with a black border. In the center, the text "Upload ZPL File" is displayed in bold. Below it is a text input field labeled "File Content:". Below the input field is an "Upload" button.

**Upload ZPL File**

File Content:

Upload

- ConfigureVariablesPage

### Configure Variables

#### Available Variables

☐ Edge

☐ maxGuards

☐ minGuards

☒ minimalSpacing

☒ NeighbouringShifts

#### Involved Sets

☒ Stations

alias

☐ Times

alias

#### Involved Parameters

☐ NumberOfSoldiers

[Continue](#)

- ConfigureConstraintsPage

### Configure High-Level Constraints

#### Constraint Modules

Module Name

Add Constraint Module

Module1

#### Define Constraint Module

Module1

Description:

This is the module's description

This module's constraints:

trivial3

Involved Sets

• Stations

• Times

#### Available Constraints

minGuardsCons

trivial7

Soldier\_Not\_In\_Two\_Stations\_Concurrently

maxGuardsCons

trivial4

All\_Stations\_One\_Soldier

trivial5

minimalSpacingCons

trivial1

[Continue](#)

[Back](#)

- ConfigurePreferencesPage

Configure High-Level Preferences

Back

Preference Modules

Module NameAdd Preference Module

Preference Module 1

Define Preference Module

Preference Module 1

Description:

Description

This module's preferences:

sum<i,a,b>inSoldiersToShifts sum<m,n>inShift (Edge[i,a,b]\*Edge[i,m,n]\*(b-n))

Involved Sets

Stations

Continue

Available Preferences

(minimalSpacing)\*\*2

((maxGuards-minGuards)+weight)\*\*3

- SolutionPreviewPage

Solution Preview

Back

Constraints

Module1

Module Description: This is the module's description

Constraints

trivial3

Involved Sets

Stations

Times

Involved Parameters

NumberOfSoldiers

Preferences

Preference Module 1

Module Description: Description

Preferences

Preferences

Preference Module 1

Module Description: Description

Preferences

$$\text{num} = \langle \text{La}, \text{J} \rangle \rightarrow \text{inSoldier} \rightarrow \text{ToShift} : \text{num} = \langle \text{m}, \text{u} \rangle \rightarrow \text{inShift} (\text{m} = \text{u} \text{ or } \text{b} = \text{u}) \wedge (\text{Edge}(\text{La}, \text{b}) \wedge \text{Edge}(\text{J}, \text{m}, \text{u}) \wedge \text{Q}(\text{u}))$$

Involvd Sets

Stations

Times

Involvd Parameters

NumberOfSoldiers

Variable Sets

Stations

Type: TEXT

Variable Sets

Stations

Type: TEXT

Station1

Station2

Times

Type: INT

4

8

12

Parameters

Solve

- SolutionResultsPage

Solution Results

Select Variable:  
Edge

TIMES \ STATIONS	STATION1	STATION2
2	✓	✗
6	✗	✓
5	✓	✗
10	✗	✓
3	✓	✓
1	✓	✗
9	✓	✗
4	✗	✓
7	✗	✓
8	✗	✓

- ViewImages

Plan A

Search Bar

Image 1

Image 2

Image 3

Image 4

Image 5

Image 6

Image 7

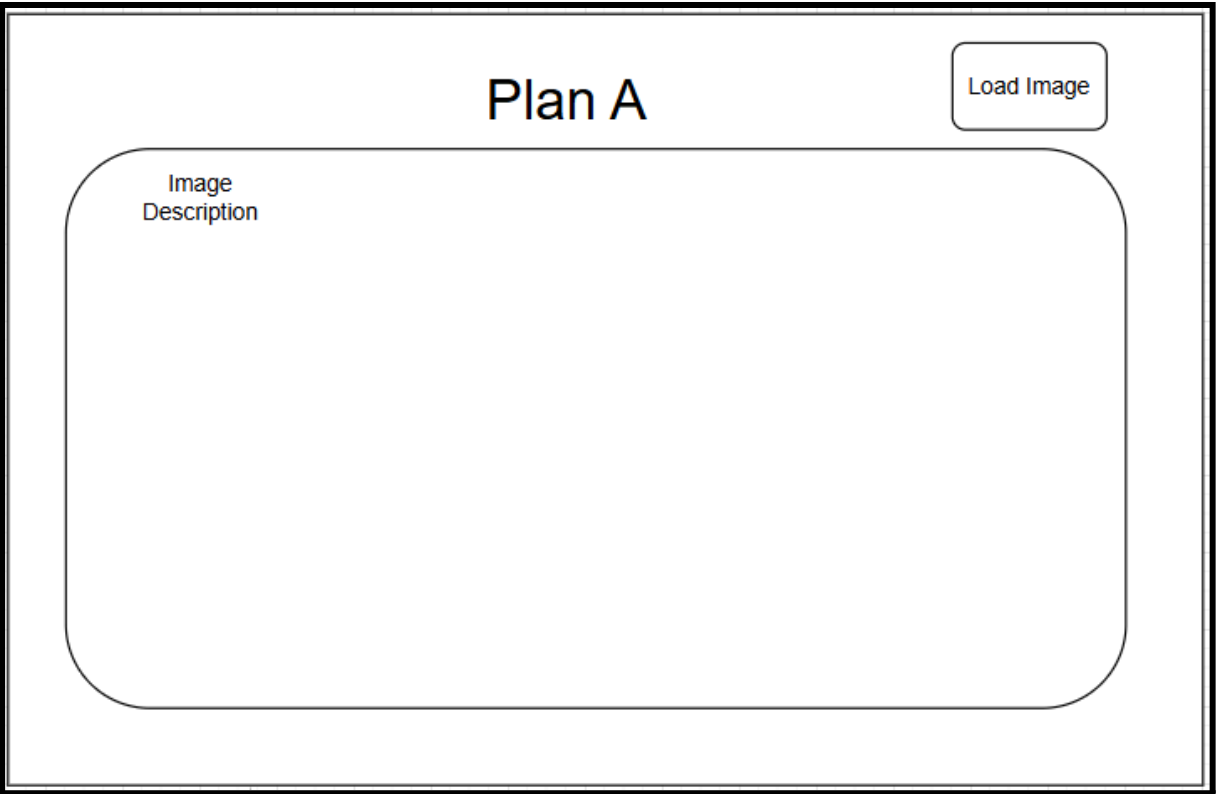
Image 8

Image 9

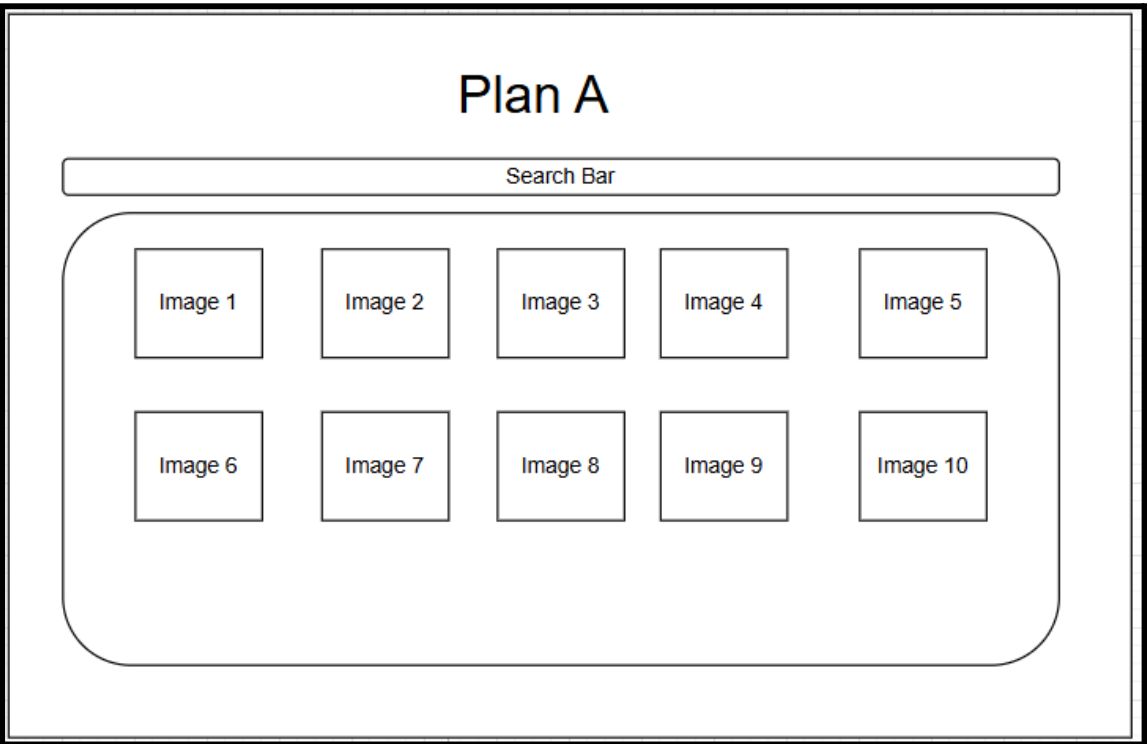
Image 10



- ImageInfo



- MyImages



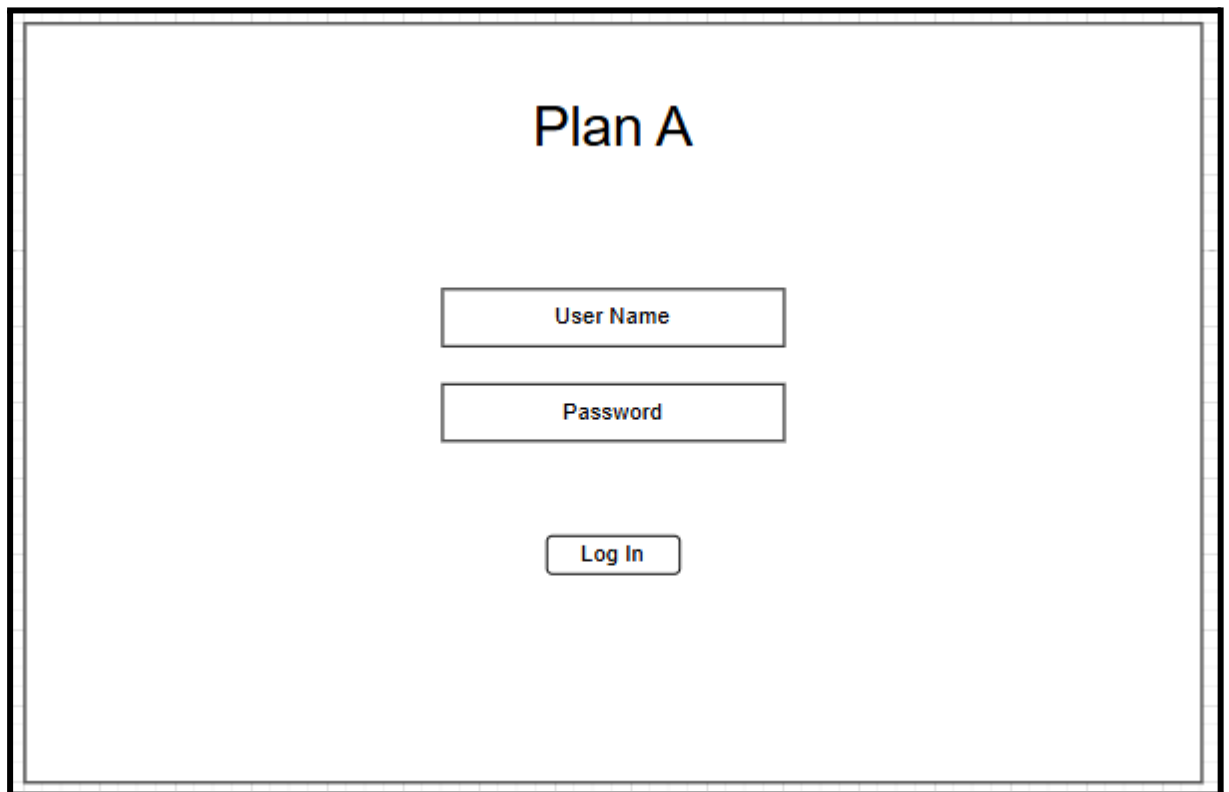
- MyImageInfo

Plan A

Delete ImageLoad Image

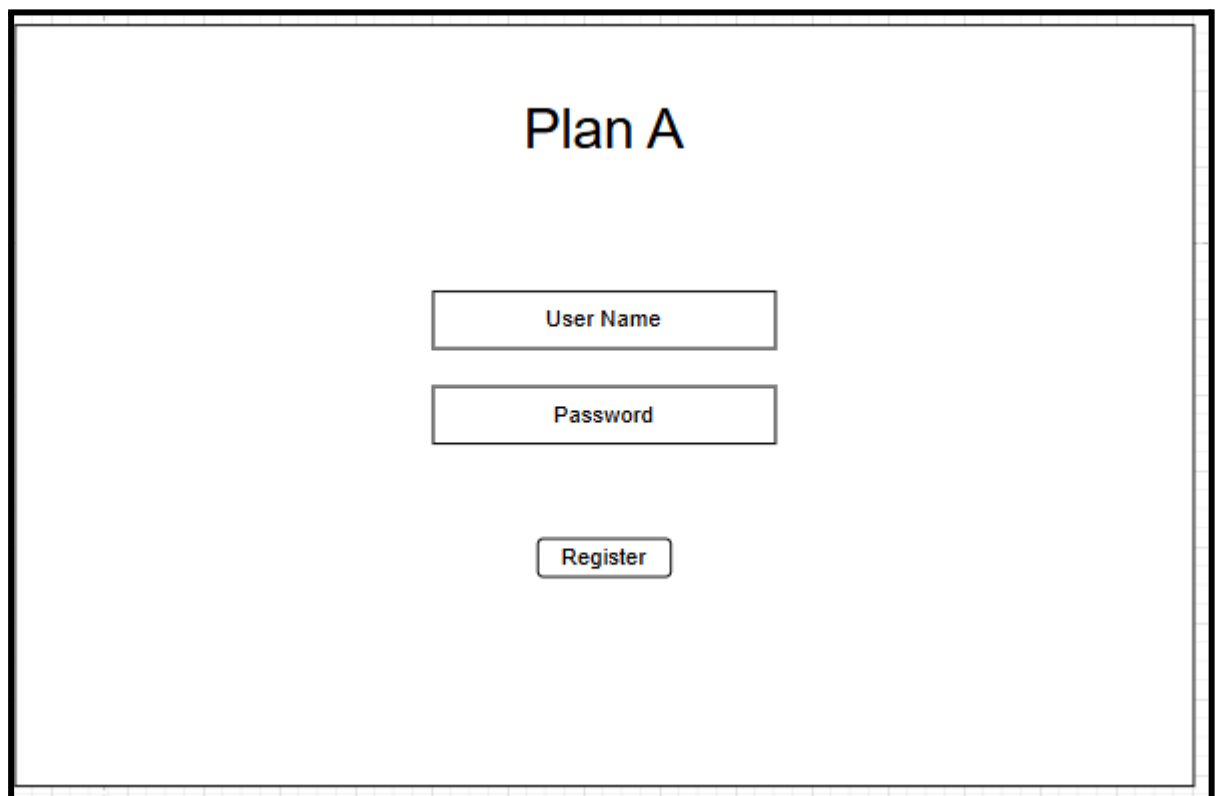
Image  
Description

- LoginPage



A mockup of a login page titled "Plan A". The page features a large, light gray rectangular area with a thin black border. Inside this area, the text "Plan A" is centered at the top in a large, black, sans-serif font. Below the title, there are two input fields stacked vertically, each with a thin black border and the placeholder text "User Name" and "Password" respectively. Below these fields is a single button with a thin black border and the text "Log In".

- RegisterPage



A mockup of a register page titled "Plan A". The page features a large, light gray rectangular area with a thin black border. Inside this area, the text "Plan A" is centered at the top in a large, black, sans-serif font. Below the title, there are two input fields stacked vertically, each with a thin black border and the placeholder text "User Name" and "Password" respectively. Below these fields is a single button with a thin black border and the text "Register".

## Chapter 7 Testing

Since our server will need to support managing multiple users, and with the (probable) inclusion of a mix of async and synchronous operations, concurrency and stress testing is a major priority to ensure proper functionality:

Concurrency Testing- A must have to catch issues like race conditions, deadlocks, starvation, data inconsistency and so on. We'll probably use Apache Jmeter for that purpose, which has plugins for testing async execution with Kafka as well.

Stress Testing- Jmeter be used here as well. Although we don't have a specific goal to support a big amount of users, we do build our system around scalability, so capability to handle a decent amount of load is expected, that of course will be determined by these tests.

Beyond These, There of course will be tests any major software has:

Unit Testing- We'll be using the normal Junit for unit tests, with mocks using Mockito. Spring's functionalities (like repositories, services, and so on) will be tested using Spring Boot Test, which is a Junit wrapper with support for testing of Spring utilities. Test coverage will be measured.

Integration Tests- Testing for this will be done mostly Junit and Spring Boot Test as well, with probably Testcontainers for database simulation during testing, and RestAssured for testing API calls.

Acceptance Test- For each use case there will be a Selenium script simulating it in the frontend, and verification of functionality.