EXAM NUMBER

Y3864454

TABLE OF CONTENTS

1. SYSTEM DESIGN

Nowadays people of different occupations (students, teachers, managers, waiters and more others) need to make notes while working: for instance, students while listening a lecture, teachers while working on a presentation, managers while communicating with consumers and waiters while accepting an order. Keeping all information in one place would significantly contribute to people's performance while they work and assist them to process the information in a convenient and quick way. What is more, having an information splitting on different categories, i.e. notebooks, would help people to store data in a well-organized way. Thus, a note keeping system "Nevernote" would support people in storing information.

The first step in a service lifecycle is service oriented analysis. As we need to develop a new system, we need to identify stakeholders requirements. All stakeholders requirements are listed in the assessment paper.

According to Erl, "a large problem can be better constructed, carried out, and managed if it is decomposed into a collection of smaller, related pieces. Each of these pieces addresses a concern or a specific part of the problem" [1, p. 3]. It means that if we split the server side of the application into independent web-services and each of them is be responsible for one individual unit of logic, it would have significant advantages:

- web-services could be running on different machines;
- web-services could be implemented using different programming languages;
- each web-service could have its own database;
- each web-services could be developed with different technologies.

We are going to use SOAP [2] approach in implementing the server side of the application. In comparison with other approaches such as CORBA, file-based communication, sockets, common database, SOAP is more suitable, manageable and flexible for developing client-server application because it has a list of considerable pluses:

- it has a specification document which defines a SOAP design, messages format, envelope, encoding rules and remote procedure call;
- can work over different application level protocols (HTTP/ SMTP);
- provides standard facilities for reporting faults;
- real-time communication between client and web-service in comparison with file-based communication;
- support multiple clients;
- if one of web-services stops working, others will not be affected and, thus, user will be able to continue invoking service operations;
- it has a support for different handlers (e.g. authorization handler) in comparison with CORBA "where messages hardly pass through the security barrier like a firewall" [3, p. 195].

What is more, SOAP message format is easy to understand XML document which consists of only four parts: envelop which is a container for a whole message, header which is composed of important meta-information, body which is an XML formatted document and optional part - fault which describes an error occurred on a server side. As Erl emphasizes, "the SOAP messaging framework fulfils the need for SOA's reliance on "independent units of communication," by supporting the creation of intelligence-heavy, document-style, and highly extensible messages." [1, p. 154]. What it means that as a SOAP message is just a simple XML document, a lot of programming languages can process it and, thus, communicate with each other through SOAP messages which is resulted in SOAP approach has become one of the most reliable way of communication in service-oriented context.

The second step in a service lifecycle is to create a design of the system with UML class diagram. As we have three logic parts in a subject area of a note keeping system (people who use the application, i.e. users, notebooks where people store notes and notes themselves which have the description of a particular activity that needs to be stored), we will split the application into 3 independent web-services: "Users Web-Service", "Notebooks Web-Service" and "Notes Web-Service".

To implement web-services, entities and faults we are going to use "ArgoUML" application which allows us to define web-services through SOAMLite profiles [4]. With this application we can easily define web-services, their operations, entities and faults through stereotypes. With the help of stereotypes, the class diagram becomes easy to follow and understand. What is more, constraints of SOAMLite profiles enables to develop a class diagram that inherits the principles of service-oriented architecture [1]. Let's start with defying the "Users Web-Service". Firstly, we will describe web-service operations and, secondly, implement them in "ArgoUML".

1.1 Users Web-Service

"Users Web-Service" is responsible for managing users, storing information about them, deleting and updating existing accounts. According to the application requirements, some operations need user to be authorized with the system and other operations are open to everyone. Having this restriction in mind, we will split users web-service into two web-services: users web-service and public users web-service which contains operations that can be invoked by not authorized users (this will be further discussed in details in).

The main entity in both web-services is a "User" entity. As in the context of web-services entities cannot have methods, entity "User" has only properties: its id, name, nick name which must be unique and consists of no more than 16 characters, email and password.

Let's identify service operations in the "Public Users Web-Service" (Table 1) and then in the "Users Web-Service" (Table 2).

Table 1 Public users web-service operations

Public Users Web-Service Operations					
№	**Operation**	**Description**	**Input**	**Output**	**Potential faults**
1.	create a new user account	when a new user wants to register with the application	the entity "User"	–	- a user with such nick name already exists in the system; - a nick name too long (more than 16 characters); - a email is not valid; - a password is weak (less than 7 characters).
2.	get all registered users	this operation can be used in the case if a client application has an opportunity to show a particular user current participants of this web-application	–	a list of registered users	–
3.	find user	when a particular user authorizes with an application, we can find such user in a database and if user with such nickname exist, we allow them to log in	a user's nickname	the entity "User"	- a user with such nickname doesn't exist

Table 2 Users web-service operations

Users Web-Service Operations					
№	**Operation**	**Description**	**Input**	**Output**	**Potential faults**
1.	update a user account	can be used when a user wants to change their private information	the entity "User"	–	- a user with such id doesn't exist in the system; - a user with such nick name already exists in the system; - a nick name too long (more than 16 characters); - a email is not valid, email should be kind of "name@gmail.com"; - a password is weak (less than 7 characters).
2.	delete a user	can be used by admins or if users want to deactivate their accounts	a user's id	–	- a user with such id doesn't exist in the system
3.	find user by Id	can be invoked by other service operations, for example, in update method	a user's id	the entity "User"	- a user with such id doesn't exist in the system

As each web-service has a list of potential faults, we need to define them. As Erl emphasizes, faults "store a simple message used to deliver error condition information when an exception occurs." [1, p. 148]. It means that a client application can extract a message from a fault that has been sent by server and identify an occurred error. From defined operations we can specify two faults: "*InvalidUserFault"* which happens when a user doesn't exist in the system and "*InvalidUserDetailsFault"* which contains a list of restriction for a user information (used nick name, long nick name, invalid email, weak password) and happens when a user information is not valid.

Finally, to demonstrate faults occurrence, let's show a business process in a context of users web-service – a user registration (Fig. 1).

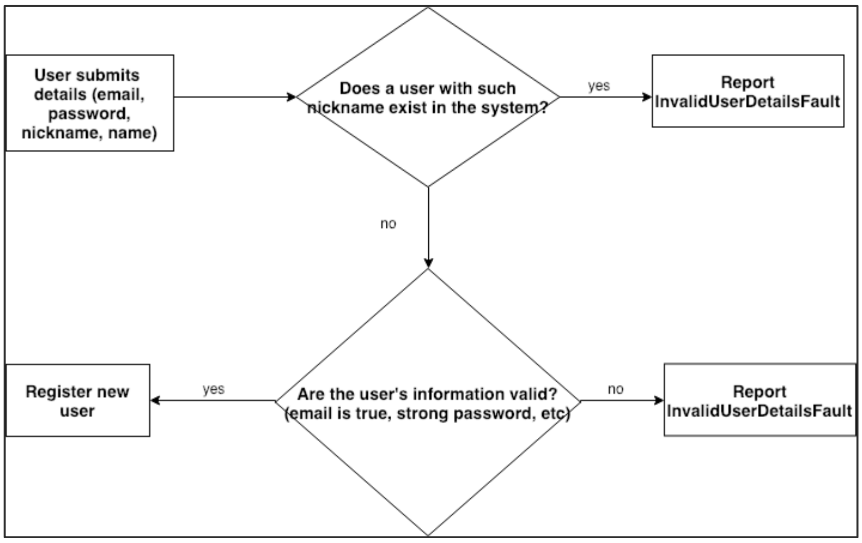

User submits details (email, password, nickname, name)
Does a user with such nickname exist in the system?
yes
Report InvalidUserDetailsFault
no
Are the user's information valid? (email is true, strong password, etc)
yes
Register new user
no
Report InvalidUserDetailsFault

Fig. 1 The user's registration process

1.2 Notebooks Web-Service

"Notebooks Web-Service" is responsible for managing notebooks, updating and deleting them from the system. Similarly, like "Users Web-Service" it is divided into two logical parts: "Public Notebooks Web-Service" and "Notebooks Web-Service".

The main entity in both web-services is "Notebook" which has the following properties: its id, name which must be unique, type which can be "regular" or "starred", read only flag which is needed for specifying access level when a user shares a notebook, created date attribute, shared flag which is true when a notebook is in public access and user id attribute which is the id of the owner of this notebook. Let's identify service operations in the "Public Notebooks Web-Service" (Table 3) and then in the "Notebooks Web-Service" (Table 4).

Table 3 Public Notebooks web-service operations

Public Notebooks Web-Service Operations					
№	**Operation**	**Description**	**Input**	**Output**	**Potential faults**
1.	get all public notebooks	is used to display all notebooks which have been shared by their owners	–	a list of public notebooks	–

Table 4 Notebooks web-service operations

Notebooks Web-Service Operations					
№	**Operation**	**Description**	**Input**	**Output**	**Potential faults**
1.	create a new notebook	when a user adds a new notebook to his collection of notebooks	the entity "Notebook"	–	- a notebook with such name already exists in the system
2.	find a notebook	is used by other operations, for example, in create method when an application need to check whether a notebook with such name exists	a notebook's name	the entity "Notebook"	- a notebook with such name doesn't exist in the system
3.	find a notebook by Id	is used by other operations, for example, in update method	a notebook's id	the entity "Notebook"	- a notebook with such id doesn't exist in the system
4.	delete a notebook	when a user deletes a notebook from his collection of notebooks	a notebook's name, a nick name of the user that deletes the notebook	–	- a notebook with such name doesn't exist in the system
5.	make a notebook public	when a user shares a notebook	a notebook's id, a nick name of the user that updates the notebook	–	– (as a user has successfully logged in, the situation that a user with such name doesn't exist is impossible)
6.	get all user's notebooks	to display all existing notebooks in own user's profile	a user's id	a list of user's notebooks	– (as a user has successfully logged in, the situation that a user with such id doesn't exist is impossible)

7.	update a notebook	when a user wants to change an information about notebook	the entity "Notebook", a nick name of the user that updates the notebook	–	- a notebook with such id doesn't exist in the system; - a notebook with such name already exists in the system
8.	delete all user's notebooks	when a user wants to delete all notebooks	a user's id	–	– (as a user has successfully logged in, the situation that a user with such id doesn't exist is impossible)
9.	delete notebook from public notebooks	when a user wants to make a notebook private (unshare a notebook)	a notebook's id, a nick name of the user that deletes the notebook	–	– (as a user has successfully logged in, the situation that a user with such nick name doesn't exist is impossible)
10.	push notification	this method is used by other service methods when they perform actions in shared notebooks	a nick name of the user that makes updates, the entity "Notebook", the flag which tells whether a notebook has been deleted, the flag which tells whether a notebook has been made private or public	–	–
11.	tryAuth	when a user tries to authenticate with a system	–	–	–

There are two faults: "*InvalidNotebookFault*" which happens when a notebook doesn't exist in the system and "*InvalidNotebooksDetailsFault*" which happens when a notebook with such name already exists in the system.

Finally, to demonstrate the faults occurrence, let's show process in the context of "Notebooks Web-Service"– a notebook update process (Fig. 2).

1.3 Notes Web-Service

"Notes Web-Service" is responsible for managing notes, updating and deleting them from the system. The main entity is "Note" which has the following properties: its id, title which must be unique, description, starred flag which specifies the type of a note, created date attribute, notebook id which specifies the parent notebook of this note and user id attribute which is the id of the owner of this note.

As the operations in this web-service are almost similar to the operations defined in the "Notebooks Web-Service" as well as faults, we are not going to describe them here.

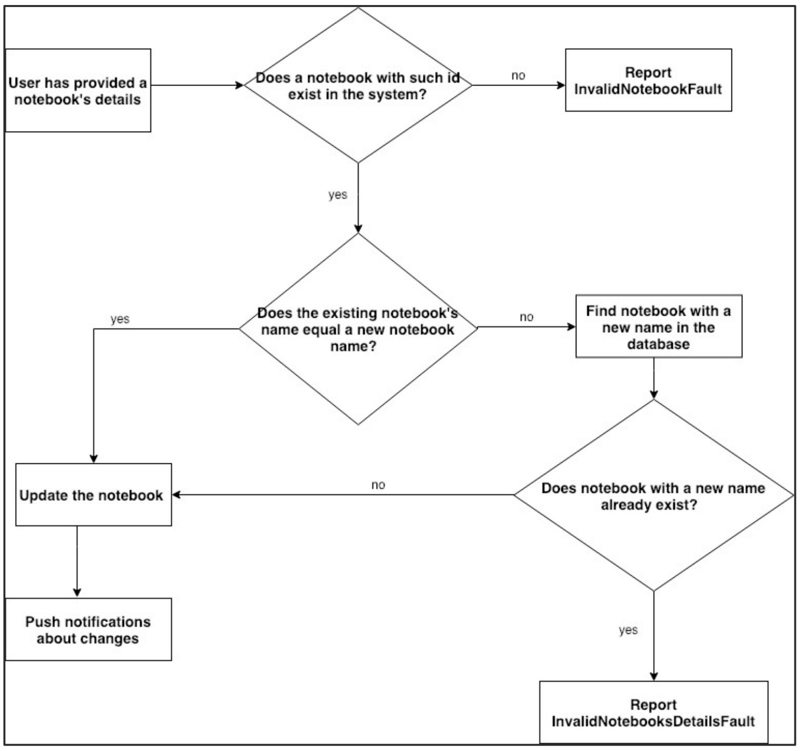

User has provided a notebook's details
Does a notebook with such id exist in the system?
no
Report InvalidNotebookFault
yes
Does the existing notebook's name equal a new notebook name?
yes
no
Find notebook with a new name in the database
Does notebook with a new name already exist?
no
Update the notebook
Push notifications about changes
yes
Report InvalidNotebooksDetailsFault

Fig. 2. The notebook's update process

1.4 ArgoUML Profile implementation

The final step is to implement web-services through UML class diagrams with “ArgoUML” application. The model has five class diagrams which conforms to the web-services defined before (notebooks, public_notebooks, users, public_users and notes web-services). In the class diagram, services have a stereotype <<service>> which identifies them as a service, methods have a stereotype <<so>> (service operation), entities have a stereotype <<entity>> and faults is simply <<faults>>. What is more, for a specific service operation we can define faults which this method can throw. For example, “updateNotebook()” method throws two fauls - “*InvalidNotebookFault”* and “*InvalidNotebooksDetailsFault”* .You can find the class diagram implementation in the file “nevernote-keeping-system-services.zargo”.

1.5 The use of message queues

As we need to let client applications know when users make changes in shared notebooks, we need to implement a communication between servers and clients. To do that, we are going to use Active MQ MOM [5] which provides us with two types of channels: topics which are intended for publish and subscribe channel and queues which enable point-to-point communication. As we have a set of web-services which is one logic application and a multiple client applications, we use “Topics”. With “Topics” we have one publisher and a lot of subscribers. As Eugster emphasizes, “the strength of this event-based interaction style lies in the full decoupling in time, space and synchronization between publishers and subscribers” [6]. What it means that client applications (e.g. subscribers) don’t need to poll web-servers periodically, they just subscribe on a particular event and wait when it happens. Polling the system periodically results in many unnecessary calls which add to the workload of web-servers and consume its bandwidth. With “publish-subscribe” approach we avoid polling the system which makes a specific application more flexible and scalable. The design of the communication between clients and web-services is shown in Fig. 3.

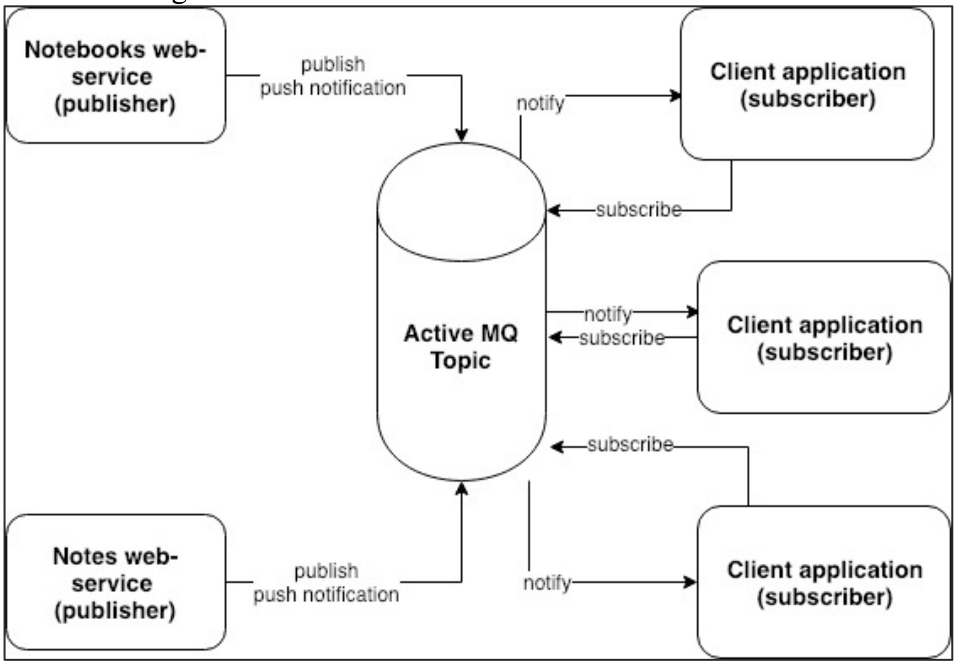

Notebooks web-service (publisher)
publish push notification
notify
Client application (subscriber)
subscribe
Active MQ Topic
notify
subscribe
Client application (subscriber)
subscribe
Notes web-service (publisher)
publish push notification
notify
Client application (subscriber)

Fig. 3. Publish-subscribe approach

1.6 Conclusion

In the analysis phase we have defined operations, faults and entities of each web-services. For each method we defined its input, output and faults. In conclusion, we have developed five class diagrams with argoUML Profiles application.

To conclude, each web-service has its own logic and inherits the properties of web-services design principles: stateless – they don’t have attributes, i.e. they don’t need to remember past/cater for future requests, reusable – duplicated behaviour is extracted in reusable methods, autonomous – each web-service is independent, discoverable – their description and names of operations are understood by humans and other web-services and, finally, conforming to the Erl, web-services are loosely coupled, i.e. they have been “designed to interact without the need for tight, cross-service dependencies” [1, p. 291].

2. IMPLEMENTATION

2.1 Application architecture

Users interact with the system through a graphical client which is implemented with Java Swing. The clients invokes different web-service's operations and, thus, obtain necessary information. Web-service's operations extract information from one shared database which is implemented with H2. The architecture of the application is shown in Fig. 4.

Another implementation decision which was rejected is to create another web-service which performs the role of intermediary – API Gateaway. In other words, clients would not know anything about services, they would know only about the one gateaway service which manages all incoming requests and distribute them among web-services. As Newman emphasizes, such approach has a significant list of advantages: "this can marshal multiple backend calls, vary and aggregate content if needed for different devices, and serve it up" [7]. However, according to Newman, such implementation decision will lead to a huge layer of code which will be difficult to test, maintain and manage. It would be better to create a backend layer for every device. In other words, "to restrict the use of these backends to one specific user interface or application"[7]. For this reason, we decided to directly use each web-service by backend part of a particular device (in our case, by java swing desktop application).

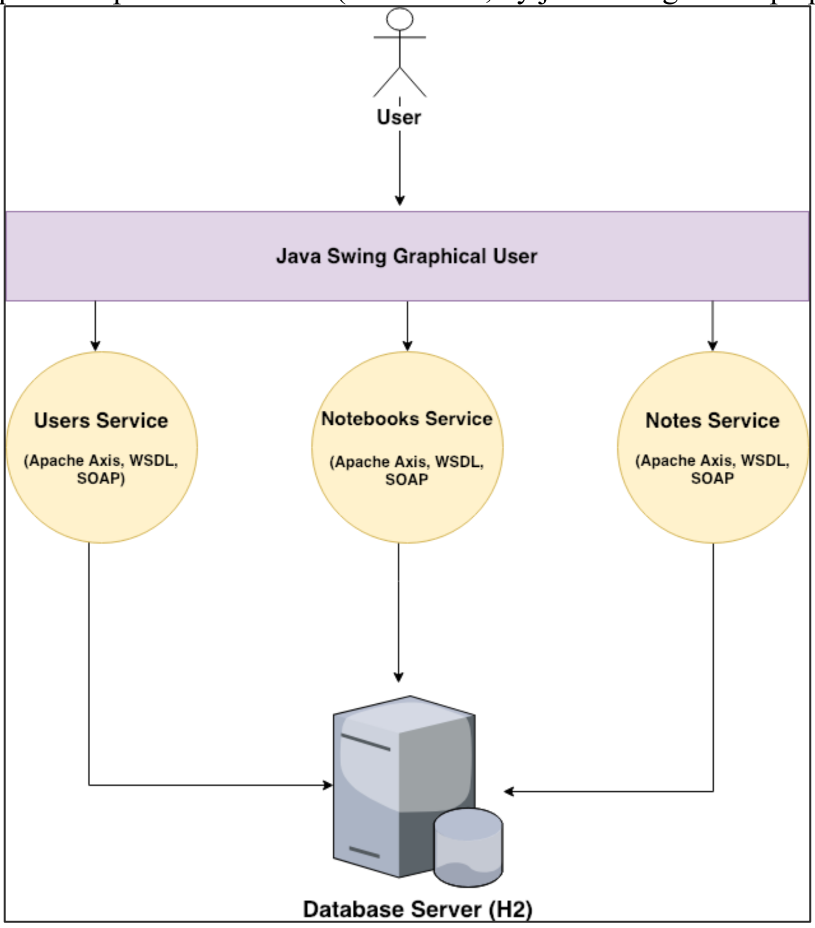


Fig. 4. The application architecture

2.2 Entity model

From the defined application requirements, methods and entities we can create an entity model – a database storage. To store data we will use "H2 database" in an embedded mode [8]. It means that we can't create more than one database connection in the particular moment. For this reason, we will use a pattern called "Singleton" and in a service side we will define a Database class which is responsible for establishing/closing database connection, performing select and update queries. It means that every time when a particular service operation tries to create a new Database class, it will receive an existing instance of a Database class instead of creating a new one.

It should be noted that if a new instance of the system is initialised (for example, an application is launched on a particular computer for the first time), we have developed a special method which initializes a new instance of a database and injects required data to it programmatically so that one can play with the system without having to manually create users, notebooks, notes. For instance, a person can log in to the application with the login “user” and password “1234567” or with the login “user2” and password “1234567”. Thus, the person can test the application without spending the time on registering new users manually. To initialise a database the application requires an SQL script which generates a new database. This script is located under “src/script.sql” which contains SQL statements such as “CREATE TABLE, ALTER TABLE and others”. If a database has already been initialised, then the application skips this step and starts to work with the existing instance of the database.

The application “Nevernote” has three entities – users, notebooks and notes. Similarly, we will define three tables with the same names and fields (class attributes) like in these three entities. Additionally, in order to let users see public notebooks, we need to create another table called “Public_Notebooks” which stores the identifiers of shared notebooks. Users may have no notebooks which is displayed in the diagram as “0..*”. However, each notebook must have its owner which is displayed in the diagram as “1”. Similarly, each note has a parent notebook (one to many relationship). The entity model is shown in Fig. 5.

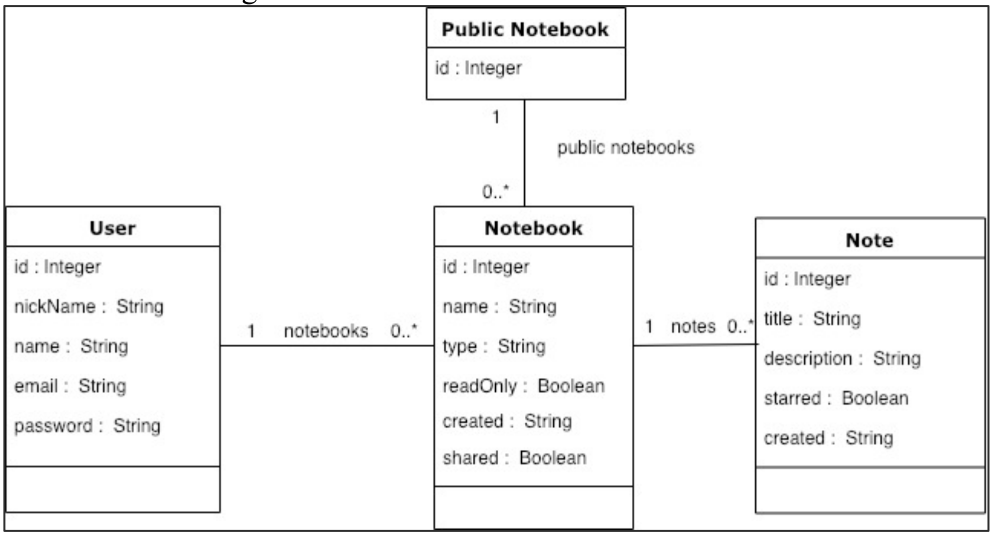


Fig. 5. The entity model of the application “Nevernote”

As we can see from Fig. 4, web-services use a shared database. Such approach is popular because it is the fastest approach of implementation to start with and is easy to understand by beginners developers. However, if we suddenly need to change the logic in a particular web-service and this influences a database, we need to be sure that we don’t spoilt parts of the schema which is used by other web-services. According to Earl, “this situation normally results in requiring a large amount of regression testing” [7]. Nevertheless, in our example we will use a shared database because this implementation decision is a good point to start with and not so time-consuming which results in we will be able to focus on other no less important parts of our application.

To conclude, when users need to change some data, web-services update the information in the shared database which guarantees a real-time communication between clients and web-services.

2.3 Services implementation

Let’s implement defined web-services. We are going to implement them using SOAP, Apache Axis, Apache Tomcat and WSDL. We will use a bottom up approach [9] in developing web-services. It means that, firstly, we will create JavaBeans – java classes which represent entities, input and output objects identified before with “ArgoUML” application. Secondly, we will develop web-service java classes which implement business logic and operations defined before. The next step is to generate WSDL document from existing web-services using Apache Axis [10]. Apache Axis allows us to generate WSDL from Java service implementations/interfaces, generate Java client stubs from WSDL, process SOAP messages in a modular way. It should be noted that according to SOAR principles we can use only primitive types (String, Integer, Float etc.) as arguments and return types in operations or classes (entities) we have already defined. Now, having WSDL document which describes abstract interfaces and concrete implementation, we can generate clients which will invoke web-services and operations defined in WSDL. WSDL-file specifies the operations defined in web-services, structure of input and return parameters, operation’s names, structure of faults.

Such bottom up approach has a number of advantages: it is has a low entry barrier. In other words, people don’t have to know a lot about WSDL or XML because the WSDL file is generated automatically. What is more, if people want to add new operations, they just regenerate WSDL document which is no time-consuming and allow them to focus on more priority tasks. However, one of the most significant disadvantage which I have faced is the loose of handlers which I added to the WSDL while regeneration. Every time when I change operations or add new ones, I have to regenerate WSDL document and, thus, all handlers which have been included to WSDL-file just disappear (for example, security handler).

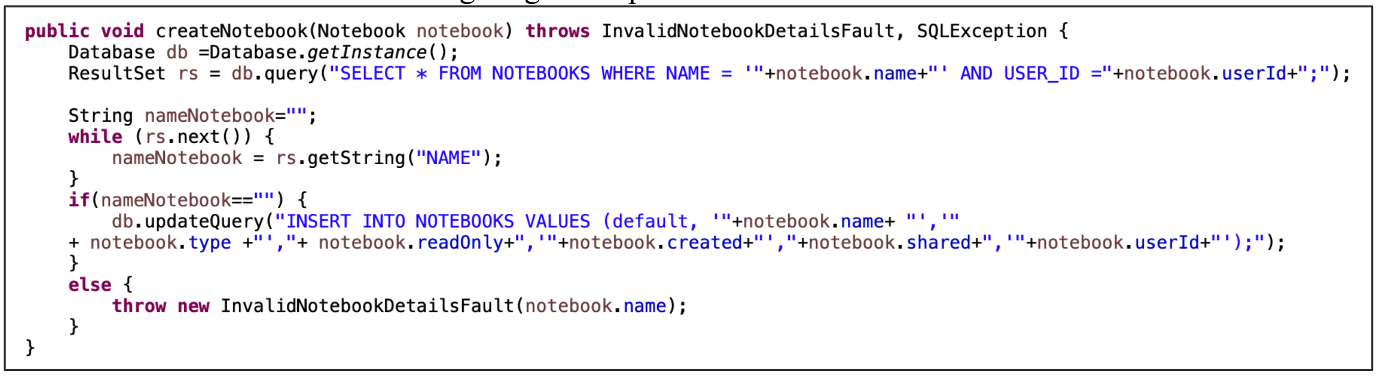
Let’s dig deeper into web-services implementation. We have one project which has four packages. Each package is responsible for its own logic: notebooks server, notes server, users server and database class. Each service includes entity class, service class, and faults classes. To let services interact with a H2 database, we added a database library under folder “Webcontent/WEB_INF/lib”. It should be noted that in the beginning of each service operation we retrieve an existing instance of a database class to let the server make updates in the database.

All operation identified before in the part *1. System Design* have been successfully implemented in each of web-service. The example of the method “createNotebook()” of “Notebooks Web-Service” is shown in Fig. 6.

All faults are separate classes which inherit the same structure. They have getters, setters and get message method which is invoked by client application when an error occurs. Let’s look at the faults management. Faults may occur in different circumstances (an entity doesn’t exist in the database, a user has provided invalid information etc.). We use two approaches to let a client application know that an error occurred in the server side: first way is to check whether information provided by client conforms to the information extracted from a database (e.g., name/title should be unique). We use the condition operator (*if, else)* to throw a fault if the client’s information is not valid. The other way is to surround database queries while its execution with *try, catch* operators (e.g., find a notebook/note/user by id). If a server was unable to execute query and find an entity with provided by client id, we again throw an exception that contains a description of a particular error. Once a fault has occurred, SOAP mechanism adds the fault block to a SOAP envelop and pass to client which parses message.

All faults has an argument which is a name of a particular entity. However, fault “InvalidUserDetailsFault” has five arguments which let “Users web-service” identify client applications what went wrong (e.g., invalid username, weak password etc).

In the “Users web-service” package we have another class called “EmailValidator” which is responsible for validating emails while a usersregistration process. Once a user has submitted their details, in the service operation “*createUser*” email validator class checks whether provided by a user email satisfies email structure through regular expressions.



```
public void createNotebook(Notebook notebook) throws InvalidNotebookDetailsFault, SQLException {
    Database db =Database.getInstance();
    ResultSet rs = db.query("SELECT * FROM NOTEBOOKS WHERE NAME = '"+notebook.name+"' AND USER_ID ="+notebook.userId+";");

    String nameNotebook="";
    while (rs.next()) {
        nameNotebook = rs.getString("NAME");
    }
    if(nameNotebook=="") {
        db.updateQuery("INSERT INTO NOTEBOOKS VALUES (default, '"+notebook.name+ "','"
    + notebook.type +"',"+ notebook.readOnly+",'"+notebook.created+"',"+notebook.shared+",'"+notebook.userId+"');");
    }
    else {
        throw new InvalidNotebookDetailsFault(notebook.name);
    }
}
```

Fig. 6 The example of the method “createNotebook()” of “Notebooks Web-Service”

3. AUTHENTICATION

According to Chase, "each message has to travel through one or more intermediate nodes, any one of which can read and/or alter a message" [11]. It can result in private users information can be viewed by no unauthorized people. As we need to protect some methods from impersonation, we need to add support for authentication. Earlier, we have described the need to split two web-services ("Users" and "Notebooks") into two parts: public and private web-services. For example, in the web-service "Public Users" we use the same entities and the same faults as in the private web-service "Users". It means that we understand "Users" and "Public Users" web-services as one logic unit just spitted into two classes. The need to split services is due to inability to add authentication handlers to different operations in the WSDL file. When we add authentication, we add this handler to entire web-service rather than to a particular service operation. Thus, we decided to split web-services into public and private.

As we need to demonstrate the ability to maintain handlers support, we simplified the process of authorization. What is means that in a real-world, applications use a password digest which is sent as a plain text through the wire. In our case, we sent login and password as a plain text. Once a user provided login and password, client application saves his credentials into a file called *"credentials"* + *username* (in a real world we are likely to save username token and digest to a client database).

All authentication process can be described in the following steps:

- a particular user provides his credentials (nickname which is a login and a password);
- next, client application sets its username, saves credential to a file called *"credentials"* + *username* and sends message to a client "*PasswordHandler*" class;
- "*PasswordHandler*" intercepts outgoing messages, extract password from *"credentials"* + *username* file, sets password to the message and forwards it to the server side;
- on the server side, password handler intercepts incoming messages, looks in the server database for a password, verifies it, and if success, approves the authorization;
- after authentication process is completed, server process only authenticated requests.

In the server side we have a class called "*PasswordHandler*" which is actually the authentication handler. To force all messages go through this handler, we have added an authentication handlers to each web-service that need to be protected in the WSDL file. All requests that need to be protected go through an authentication handler.

To let a client application know that all requests should go through a "*PasswordHandler*" class, we added "secure-notebook-client.wsdd" file which has the reference to a client's "*PasswordHandler*" class. Having implemented this wsdd file, we can be sure that all request will go through this security handler.

When a user provides their credentials (login and password), client application invokes the web-service operation "*tryAuth()*" which on the server side prints that a user has been authenticated successfully. We need to do that in order to save a username and a password to *"credentials"* + *username* file and set a username for the security handler. When a client invokes secure operations, it uses this username that has already been set.

As we have two public web-services, let's look at the operations they have which don't require users to be authenticated. First, we look at *"Public Users"* web-service. This web-service has three operations:

1. *createUser (User user).* This method is public because everyone can have an account in "Nevernote" application. In other words, every person who wants to register in the application can do that without authentication.
2. *getAllUsers ().* This method is also open to everyone. For example, a particular person need to know who use "Nevernote" application, i.e. its participants. Having this information in mind, they can decide whether they want to register or not. They can view users who have registered with the application without having their own account – without authentication.
3. *findUser (String nickname).* Similarly, this operation is also public because people who don't have a profile in the application can view information about other users (like in Facebook user's information are open on the Internet).

Other operations, which have been implemented in *"Users web-service",* need users to be authenticated. Methods *deleteUser() and updateUser()* change user's private information in a database.

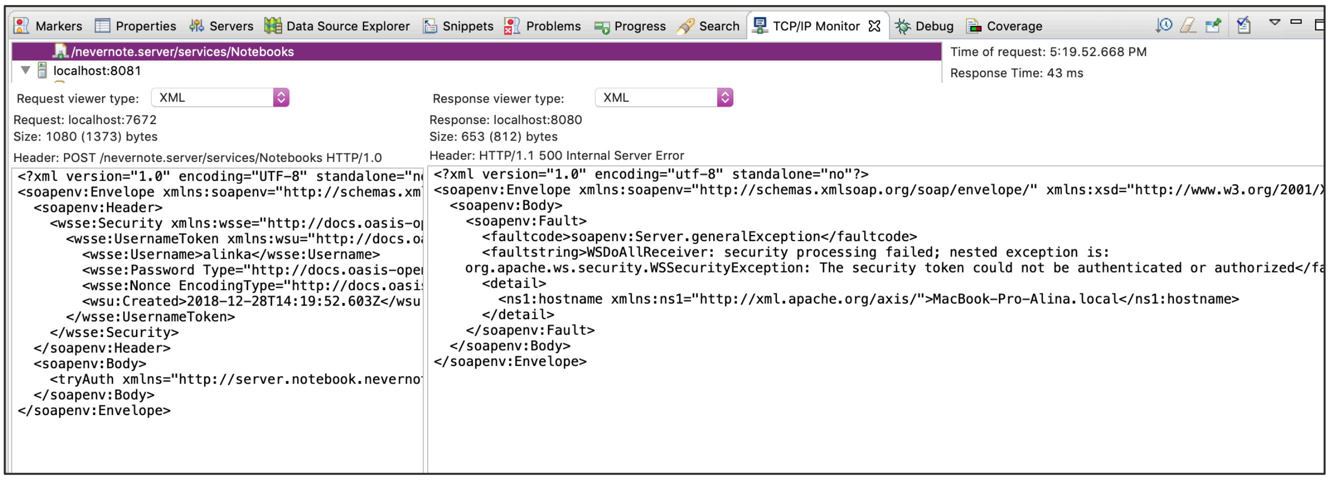
It means that users who are not registered with the application can influence database. They can have access only to public user's information (for example, people's nicknames). In the case of *findUserById()* operation, users need to be authenticated because this method require user's identifier which people usually don't know.

"Public Notebooks" has only one operation *getAllPublicNotebooks()*. This operation returns all notebooks which have been shared by users, i.e. they have become public. From my understanding, people who are not registered with the application can view public notebooks (like wall in Facebook). However, to make changes in public notebooks (e.g. add new notes or view existing notes) people should be authenticated because other users should know who have changed a particular notebook.

Other operations have been implemented in "Notebooks Web-Service" because they can be invoked only by authenticated users. For example, delete notebooks, update them and view all private notebooks can only users who has a profile in the application. For this reason, we have made these methods not in public access.

The last web-service "Notes" are entirely closed for users who don't have profiles in the application "Nevernote". This is because all notes are the integral part of notebooks, i.e. they are listed under notebooks. They cannot be viewed without notebooks. In public notebooks, notes are visible only for authenticated users. In other words, people who are not registered with the application can view existing notebooks, but not notes in them.

If a user provides invalid credentials or they are not registered with the system, the server will throw a security exception. To view all incoming messages, we forward all requests to a particular port (in our case "8080"). With this specification, we can see every request/response and investigate the problems which can occur. In Fig. 7 we can see the security fault. This message is available through TCP/IP monitor.



Time of request: 5:19.52.668 PM
Response Time: 43 ms

Request viewer type: XML
Request: localhost:7672
Size: 1080 (1373) bytes
Header: POST /nevernote.server/services/Notebooks HTTP/1.0

```
<?xml version="1.0" encoding="UTF-8" standalone="n
<soapenv:Envelope xmlns:soapenv="http://schemas.xm
  <soapenv:Header>
    <wsse:Security xmlns:wsse="http://docs.oasis-o
      <wsse:UsernameToken xmlns:wsu="http://docs.o
        <wsse:Username>alinka</wsse:Username>
        <wsse:Password Type="http://docs.oasis-ope
        <wsse:Nonce EncodingType="http://docs.oasi
        <wsu:Created>2018-12-28T14:19:52.603Z</wsu
      </wsse:UsernameToken>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <tryAuth xmlns="http://server.notebook.neverno
  </soapenv:Body>
</soapenv:Envelope>
```

Response viewer type: XML
Response: localhost:8080
Size: 653 (812) bytes
Header: HTTP/1.1 500 Internal Server Error

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/
  <soapenv:Body>
    <soapenv:Fault>
      <faultcode>soapenv:Server.generalException</faultcode>
      <faultstring>WSDoAllReceiver: security processing failed; nested exception is:
    org.apache.ws.security.WSSecurityException: The security token could not be authenticated or authorized</f
      <detail>
        <ns1:hostname xmlns:ns1="http://xml.apache.org/axis/">MacBook-Pro-Alina.local</ns1:hostname>
      </detail>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

Fig. 7 Security fault example

4. CLIENT IMPLEMENTATION

To let remote users invoke the web services we have already implemented, we need to develop a client graphical application. We are going to use Java Swing which "provides many features for writing large-scale applications in Java" [12].

The entry point to the client application is "App.java" file where we initialize all important object: session for receiving new messages from the server, a special Api class that is needed to invoke web-services, and main frame which is the main graphical window of the application. Also in here we add a listener on close window event: when such event happens, we close all sessions and connections and exit from the application.

We have a kind of an API class – "*NeverNoteApi.java*" – which contains a set of methods which invoke web-services. What is more, in this class we initialize web-services connections, define ports from which users can invoke web-services (e.g. 8081). Having this class, we don't need to initialize web-service in every graphical component, i.e. class (e.g. LoginPanel.java, NotebookMenu.java, NotePanel.java etc.), we do that only one time when the client application starts working.

The client application can be spitted to four logical parts: login/register page, user's notebooks page, shared notebooks page and notifications page. Each page is a separate Java class which are implemented through JPanel graphical component which includes other graphical components such as JButtons, JLabels and others. Let's look at each page in more details.

4.1 Login/Register Page

When users launche the *"Nevernote"* application, they see this page. In here, they should provide their credentials to log in or they can create a new account. When they select one of the two buttons (log in/register), a special dialog appears where they provide their information. The dialog checks whether users have filled all fields and, if so, return "true" to the parent JPanel component which is "*Login Page"* itself. In other case, an error dialog notifies users that they are not allowed to leave fields empty. When users provided all required data, the client application invokes a *"tryAuth()"* web-service method to check whether user's information is valid and, if it is so, users see a "successful registration dialog". After they have authenticated, they are allowed to use web-services methods from *"User Notebooks Page"*. "*Login Page"* is shown in Fig. 8.

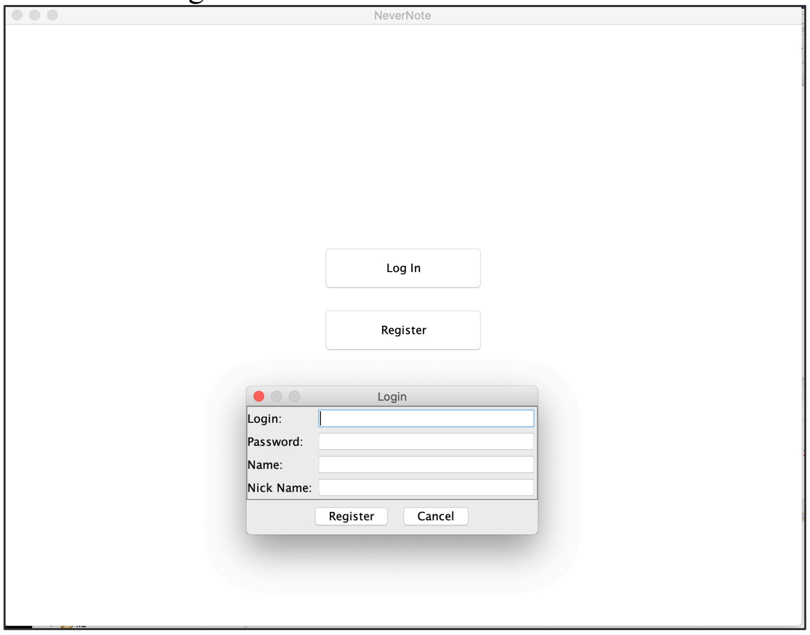
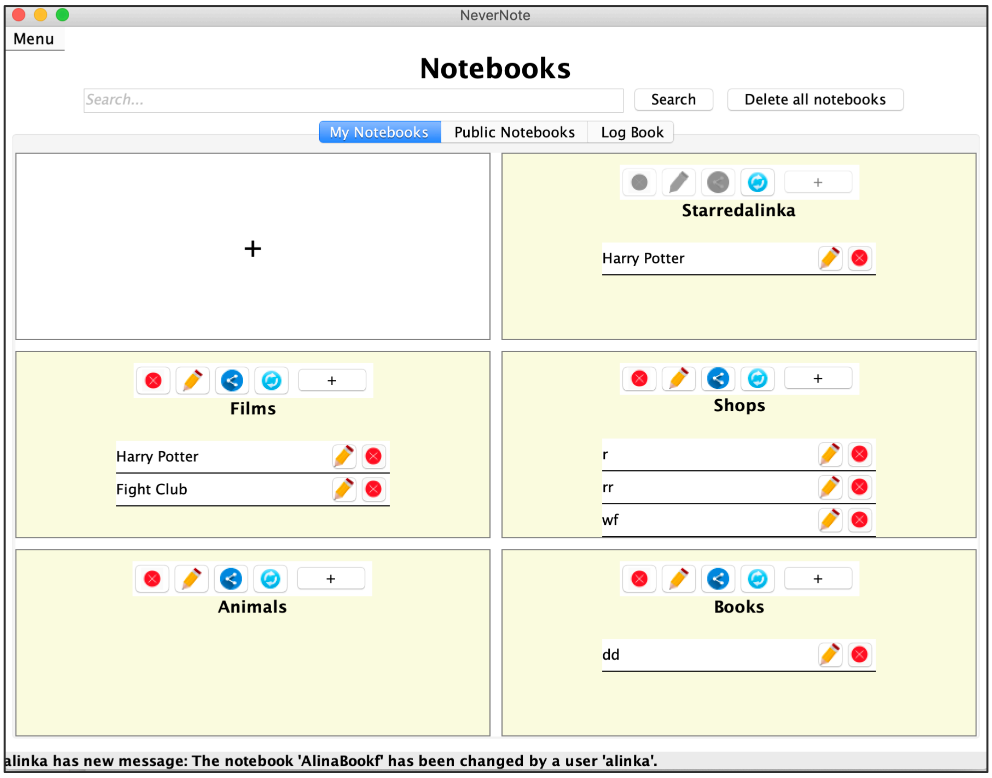


Fig. 8 Login Page

4.2 User's Notebooks Page

This page is the most important part of the application. In the very top it has a menu where a user can change his email or real name and also exit from the application. Next we can observe a search box which helps user to search a particular book. To implement search function, we use the following approach: in the JPanel components we just looks for a Notebook JPanel with the same label as in the searchbox. If the label of the notebook doesn't match the label in the searchbox, we hide this notebook. Having this function, users can quickly find a required notebook. A search box followed by another graphical component - a JButton which deletes all user's notebooks. When a user wants to delete all notebooks, the client application, firstly, deletes all user's notebooks from public notebooks, secondly, deletes all notes in notebooks and, lastly, deletes all user's notebooks.

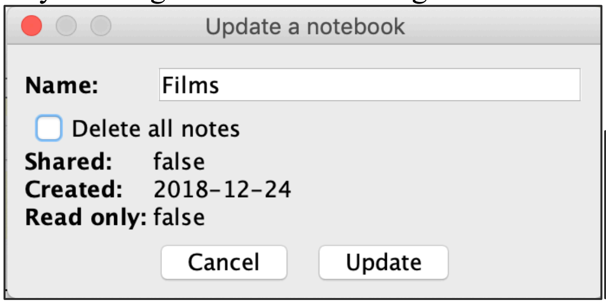
In the middle of the page we can view a main part – user's notebooks. To display notebooks in a user-friendly way, we used a GridLayout which places elements in a virtual table. This layout enables us to easily delete components and add new ones (e.g, when a user adds notebook, it easily placed in the table).



alinka has new message: The notebook 'AlinaBookf' has been changed by a user 'alinka'.

Fig. 9 A user's notebooks page

Each notebook is a Java class which extends JPanel. It has a header where user can delete, edit, update, share a particular notebooks and also add new notes. Each button in the header has its own event listener which is responsible for performing specific actions. When a mouse click event happens, a client application invokes a specific web-service method and updates a notebook panel to display information which have been returned by a web-service. For example, when a user adds a new note in a notebook, it immediately appears in the notebook panel. If a user wants to edit a particular notebook, he sees an edit dialog which contains all information about a particular notebook. From this dialog, a user can edit a notebook's title and delete all notes in this notebook Fig. 10. To share a particular notebook, a user need to specify an access level for other users (read only or read/write). If a notebook has already been shared, then, user views a message which explains that notebook is shared now and that he can unshare a notebook by clicking the same button Fig. 11 .



Update a notebook

Name: Films

Delete all notes

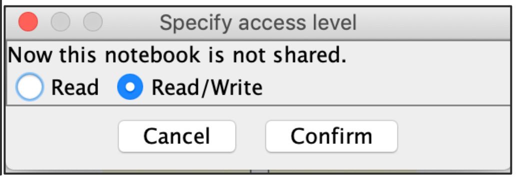
Shared: false

Created: 2018-12-24

Read only: false

Cancel Update

Fig. 10 Edit a notebook dialog



Specify access level

Now this notebook is not shared.

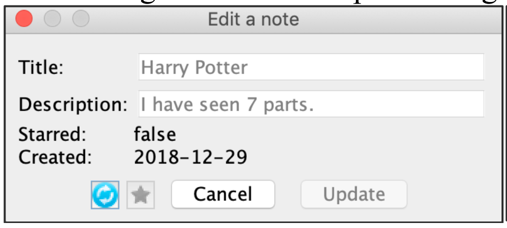
Read [✓ Read/Write]

Cancel Confirm

Fig. 11 Access level dialog

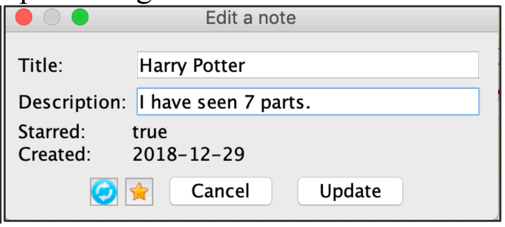
When a user successfully registered with the application, a "starred" notebook automatically appears in the notebooks of the user. It has a name "Starred" + user's nickname. Users can't modify this notebook, they are allowed only to read information. When users make a particular note starred, they should update a "starred" notebook to view new starred notes. This notebook is shown in Fig. 9. Update button in a notebook header is needed to update shared notebooks when someone make changes in them.

Each notebook contains a list of notes which are also JPanels. Users can edit notes and delete them. Before edit a particular note, users should update them in order to have the latest version of this note Fig. 13 (e.g., if this note is in a shared notebook and someone has changed this note). In other case, they are not allowed to change a note before performing updates Fig. 12.



Title: Harry Potter
Description: I have seen 7 parts.
Starred: false
Created: 2018-12-29

Fig. 12 Edit a note dialog before updates

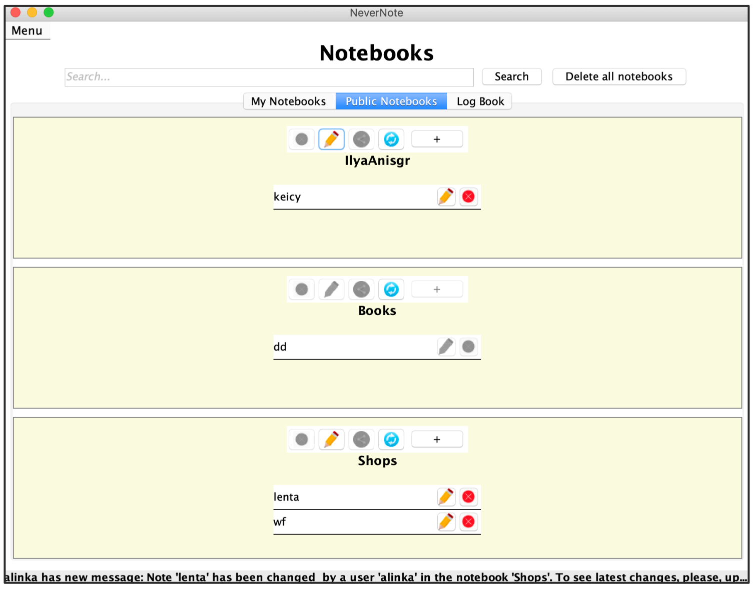


Title: Harry Potter
Description: I have seen 7 parts.
Starred: true
Created: 2018-12-29

Fig. 13 Edit a note dialog after updates

4.3 Public (shared) Notebooks Page

This page is similar to “User’s Notebooks Page” and lists all notebooks which have been shared by users. If an access level is read only, users can’t make changes in such notebook (buttons are disabled). Only “update” button is enabled to view changes, if a notebook has been changed by someone else. This page is shown in Fig. 14.

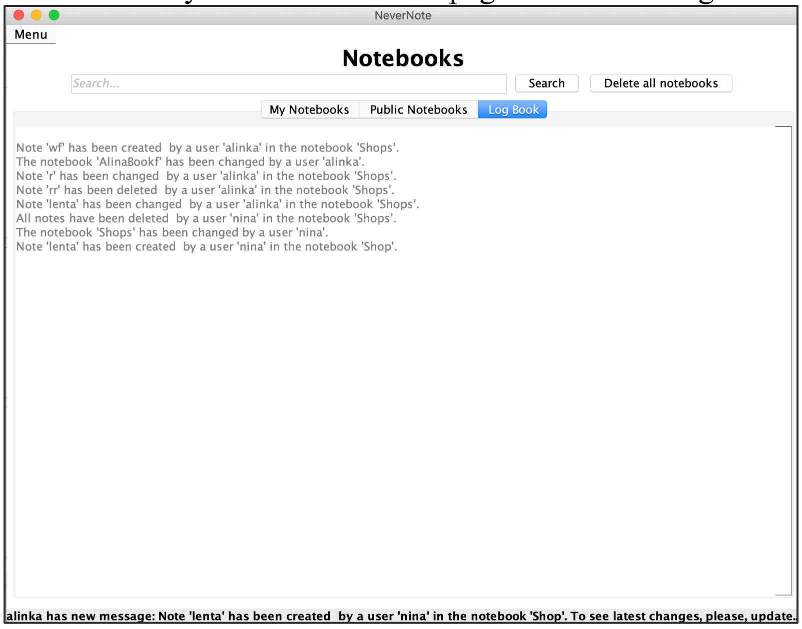


alinka has new message: Note 'lenta' has been changed by a user 'alinka' in the notebook 'Shops'. To see latest changes, please, up...

Fig. 14 A shared (public) notebooks page

4.4 Logbook Page

This page is needed to view all changes that have been made in shared (public) notebooks. Users can’t edit them, they are allowed only to read them. This page is shown in Fig. 15.



```
Note 'wf' has been created  by a user 'alinka' in the notebook 'Shops'.
The notebook 'AlinaBookf' has been changed by a user 'alinka'.
Note 'r' has been changed  by a user 'alinka' in the notebook 'Shops'.
Note 'rr' has been deleted  by a user 'alinka' in the notebook 'Shops'.
Note 'lenta' has been changed  by a user 'alinka' in the notebook 'Shops'.
All notes have been deleted  by a user 'nina' in the notebook 'Shops'.
The notebook 'Shops' has been changed by a user 'nina'.
Note 'lenta' has been created  by a user 'nina' in the notebook 'Shop'.
```

alinka has new message: Note 'lenta' has been created by a user 'nina' in the notebook 'Shop'. To see latest changes, please, update.

Fig. 15 A logbook page

5. PUSH NOTIFICATIONS

As we need to notify client applications about any changes in shared notebooks or notes in shared notebooks, we will create producers in both “Notebooks” and “Notes” web-services. Thus, we have two different topics which are “publicNotebooks” and “publicNotes” respectively. Each web-service has a method which is called “pushNotifications”.

In this method we create a connection factory which use the broker URL which we have to launch as a separate application. As we need to have an application which acts as an intermediary (broker) between service and clients, we implement another standalone application – Broker (*BrokerStart.java*). This application needs to be started only once, when we launch web-services. After the connection was created, we create session and topic which will keep messages. Then we compose a particular message and, add properties (like author of the message) and create a producer which actually sends the composed message to the topic.

This web-service operation is invoked every time when any changes occur in the database (e.g., a particular note has become starred or a notebook has a new title etc). In web-service methods (update, delete, create) we will check whether updates have occur in a shared notebook. If it is true, we will invoke “pushNotification” method which will send a message to a particular channel (“publicNotebooks” or “publicNotes”). Basically, we can describe all process in the following steps:

1. Once changes occur, web-service checks whether a notebook is shared.
2. If a notebook is shared, “pushNotification” method is invoked.
3. In this method, web-service creates a connection and session which are required to connect to a topic manager.
4. The next step is to create a message. Depending on changes (delete/update/create), we set a message.
5. To let a client applications know who was an author of the changes, a web-service set a string property “AUTHOR”. With such client users can easily know who has modified a particular note or a notebook.
6. Next, a web-service creates a producer and sends the message;
7. When the message has been sent, a web-service closes the connection and the session.

Once a message has been sent, all subscribers will receive this message. In the client side, we implement a method “subscribeToNotebooksNotifications()” in the “*NotebookMenu.java*” class. In this method we connect to the topic manager, create two consumers (one for “publicNotebooks” topic and another for “publicNotes” topic) and set message listener to consumers. Once a particular message has been received, we extract all properties which the message has, and display this message in the bottom line of a graphical client. However, if a particular user has deleted all notes in a shared notebook, the line will display only last message (i.e. the name of the last note which has been deleted). To let users track all changes, we have another panel – “Log book”. All changes are displayed here and, thus, users can open it and see all modifications that have been occurred in shared notebooks.

To conclude, the ability to subscribe to a particular event makes applications more independent which support one of the paradigms of service-oriented architecture – service loose coupling. That is to say, applications don’t need to know each other, they just publish and subscribe on an event of their interest which is result in overall application independence.

6. LIST OF RESOURCES

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. 2005.
[2] "Simple Object Access Protocol (SOAP)." [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
[3] W. Chang, Ed., *Advanced Internet Services and Applications*, vol. 2402. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.
[4] "About the Service Oriented Architecture Modeling Language Specification Version 1.0.1." [Online]. Available: https://www.omg.org/spec/SoaML/. [Accessed: 02-Jan-2019].
[5] S. Newman, "Building Microservices," 2015.
[6] "H2 Database Engine." [Online]. Available: http://www.h2database.com/html/main.html. [Accessed: 03-Jan-2019].
[7] Ibm, "Developing Web Services Applications."
[8] "Apache Axis – User's Guide." [Online]. Available: http://axis.apache.org/axis/java/user-guide.html. [Accessed: 26-Dec-2018].
[9] "Understanding Web Services specifications, Part 4: WS-Security," 2006.
[10] B. Cole, R. Eckstein, J. Elliott, M. Loy, D. Wood, and O. ' Reilly, "Java^{TM} Swing, 2nd Edition," 2002.
[11] B. Snyder, D. Bosnanac, and R. Davies, *ActiveMQ in action*. Manning, 2011.
[12] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe."