Documentation: CA 5

Software Development and Network

12.12.2014

Names:

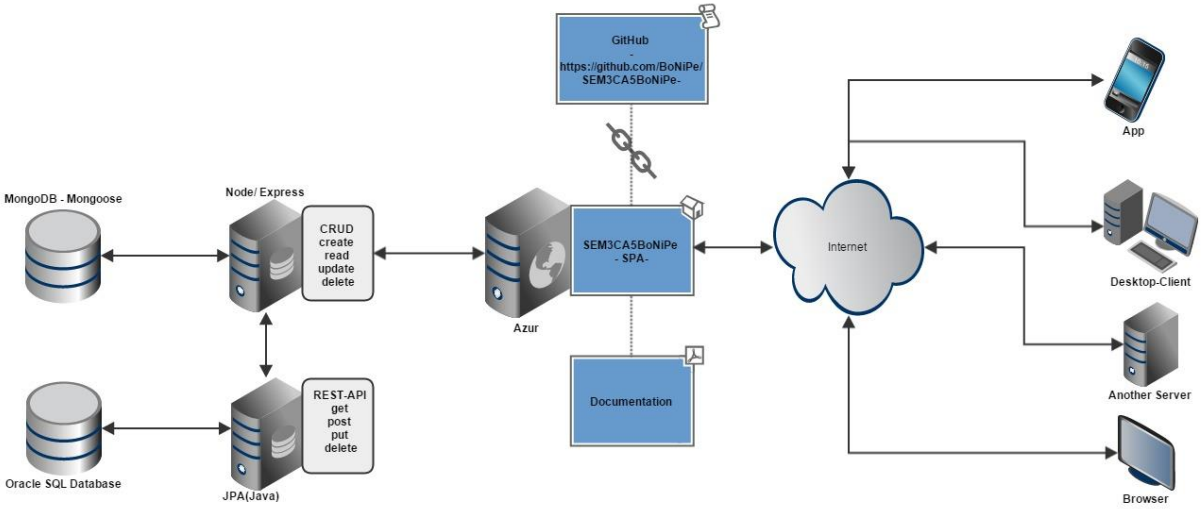
Boyko Surlev

Nikolaj Desting

Peter Tomascik

An overall architecture diagram

- diagram of the overall distributed system

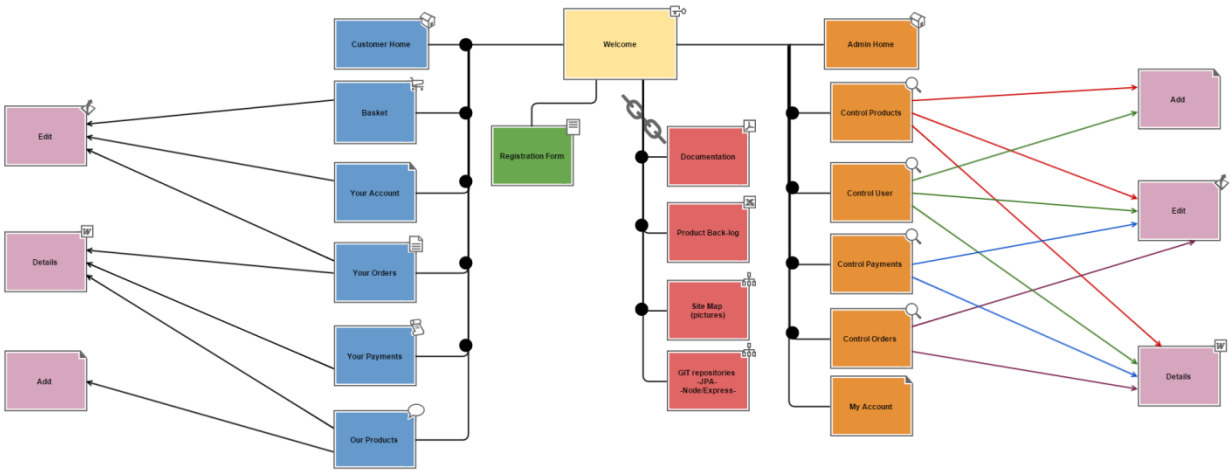


The purpose of this system is to make communication simple between a warehouse and companies that wants to order products from the warehouse stock.
For example if a restaurant wants to order 20 boxes of napkins, 20 boxes of carrots and 20 boxes of buns, they would browse this web application, find what they need in the stock of the warehouse and add it to a basket which in the end will be a full order. The guys in the warehouse will get a notification when a new order is pending, and they should be able to accept or decline the order.
This means that we have 2 types of functionality in the application: Admin and User.
Users (Client-side) are the guys in a company, who wants to order products. Their functionality should be:
• Browse stock
• Adding products to basket
• Placing order
• View recent orders placed from their company
Admins are the guys working at the warehouse. Their functionality should be following:

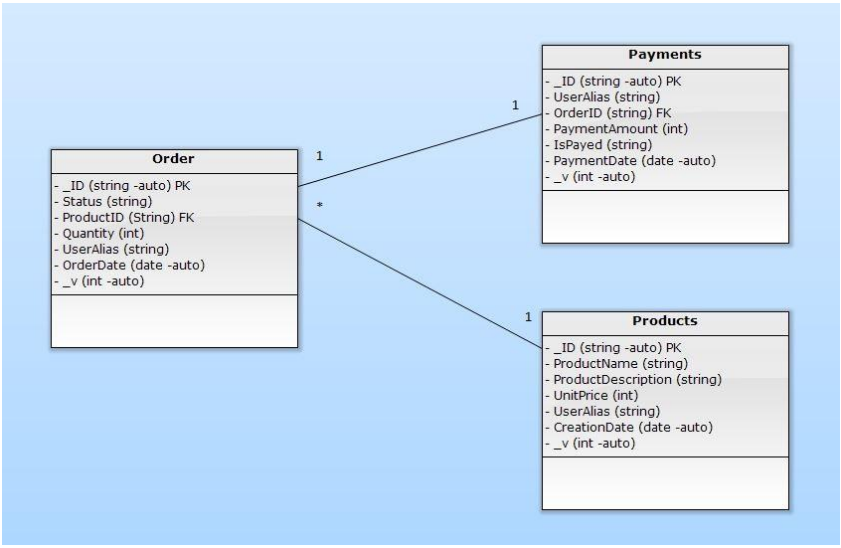
- CRUD company profiles and their users
- CRUD stock
- View order history
- View pending requests
- Accept or decline orders

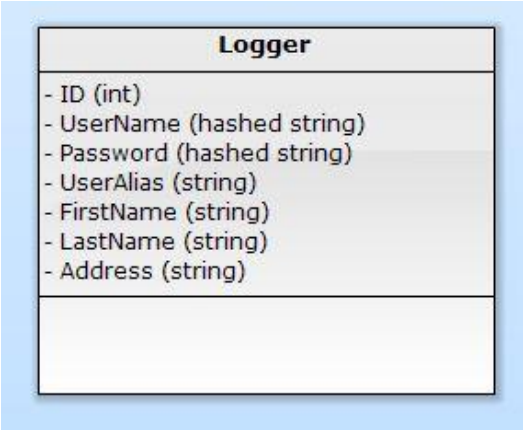
Can be edited – was not sure if this has to be general overview of idea or program func.

- site diagram of the frontend pages



- database diagram





A description of the (REST) interfaces between client - server and server - server

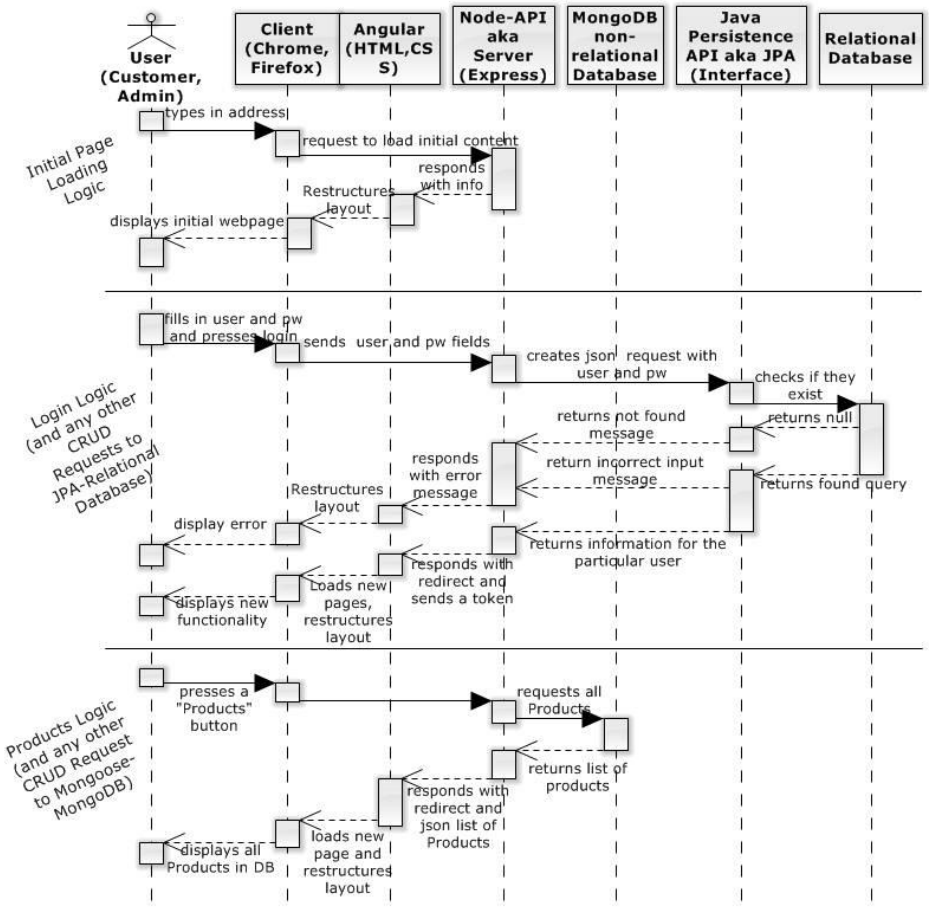
Before starting with describing how our program is actually working we prefer to explain what REST actually is. “Representation State Transfer” is abstraction of the World Wide Web. It works like an interface between the clients and the server. It is used in the development of web services.
Since we are using two databases our team has two REST servers. One of them is coded in Node.js and the other is coded in JAVA (JPA).
The “Node.js” interface is the one which handles the communication between the actual client and the Express server. It is separated in 3 departments. One of them is handling the admin logic and another one is dealing with the data which logically should be requested by the customer. In order to receive an actual response from the server, a token will be required. There is a different token for admin and customer, therefore none of them can access the functionality of the other.
The customer is able to see all products, create orders and also have full control over his/her already created orders. On the top of that, the customer will also be able to control his own account.
On the other hand, the admin has the right to control all the data which is stored in both MongoDB and the Relational Database. However the admin is not able to create “orders” or any “paymentDetails”, because this functionality is reserved for the customer. The administrator is also able to control all users.
The 3rd “department” is the one, which handles the logic between the two REST services. The secondary REST service is located between the Node.js and respectfully the JPA server. The second mentioned handles the Relational database. Our team decided that the user functionality should be set in the relational database. Every single user is able to authenticate himself. In order to do so, several requirements should be fulfilled. The client should make a request to the Node.js REST service. Whenever the Node REST server receives the request, it will automatically generate another request to the secondary server (JPA). The JPA returns a response to the Node server, which will return a response to the client. This description shortly represents the whole logic in our program and all the components in it.

Handling Failures

Whenever a user makes a request to the databases the DataLayers, in Node.js, and the Facade, in JPA, he/she will trigger a specifically designed function. Every function from those two classes can return two types of response – either the actual data, which means that the request was successful, or the response

will represent an error message, which could be triggered by an incorrect request of incorrect input data which cannot be found and confirmed by the database information. For instance, if a user tries to log into the account he/she will either receive a confirmation, which will actually represent a redirect to a particular page depending on the type of the logger, otherwise the logger will receive an error message stating the type of problem which has occurred. The frontend error handling is underdeveloped. We decided that since we have only 3 weeks to develop entire program, we should concentrate on the functionality. However the “customer frontend menu” has full frontend error handling. We accept the idea that the administrator understands the importance of his/her work and will not fill in the database with useless and incorrect information. As a conclusion on that topic, we believe that the worst error which could occur and maybe the only one, which can actually crash the program is the lack of any function, which can check if the MongoDB is actually running or not. This is not tested and we are not certain what the results would be.

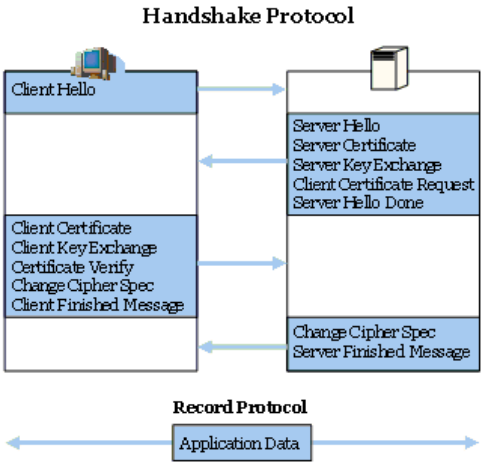
Sequence diagram



description

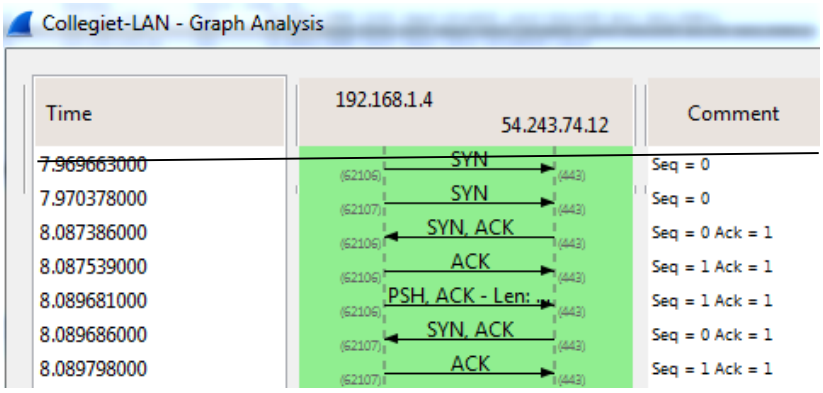
A Wireshark sample covering a REST call

Three way handshake:



This describes communication between client (IP= 137.135.180.134) and our server (IP = 54.243.74.12).

90	7.97037800	192.168.1.4	54.243.74.12	TCP	66 62107→443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
131	8.08738600	54.243.74.12	192.168.1.4	TCP	66 443→62106 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460 SACK_PERM=1 WS=256
132	8.08753900	192.168.1.4	54.243.74.12	TCP	54 62106→443 [ACK] Seq=1 Ack=1 Win=65700 Len=0
133	8.08968100	192.168.1.4	54.243.74.12	TLSv1.2	227 Client Hello
134	8.08968600	54.243.74.12	192.168.1.4	TCP	66 443→62107 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460 SACK_PERM=1 WS=256
135	8.08979800	192.168.1.4	54.243.74.12	TCP	54 62107→443 [ACK] Seq=1 Ack=1 Win=65700 Len=0
136	8.09180100	192.168.1.4	54.243.74.12	TLSv1.2	227 Client Hello
140	8.20656600	54.243.74.12	192.168.1.4	TCP	60 443→62106 [ACK] Seq=1 Ack=174 Win=15872 Len=0
141	8.20816900	54.243.74.12	192.168.1.4	TCP	60 443→62107 [ACK] Seq=1 Ack=174 Win=15872 Len=0
142	8.20956300	54.243.74.12	192.168.1.4	TLSv1.2	1514 Server Hello
143	8.21042600	54.243.74.12	192.168.1.4	TLSv1.2	1514 Certificate
144	8.21053000	192.168.1.4	54.243.74.12	TCP	54 62106→443 [ACK] Seq=174 Ack=2921 Win=65700 Len=0
145	8.21133200	54.243.74.12	192.168.1.4	TLSv1.2	138 Server Key Exchange
146	8.21212900	54.243.74.12	192.168.1.4	TLSv1.2	1514 Server Hello
147	8.21296300	54.243.74.12	192.168.1.4	TLSv1.2	1514 Certificate
148	8.21306600	192.168.1.4	54.243.74.12	TCP	54 62107→443 [ACK] Seq=174 Ack=2921 Win=65700 Len=0
149	8.21310800	54.243.74.12	192.168.1.4	TLSv1.2	138 Server Key Exchange
150	8.23644800	192.168.1.4	54.243.74.12	TLSv1.2	180 Client Key Exchange, Change Cipher Spec, Hello Request, Hello Request
151	8.35427700	54.243.74.12	192.168.1.4	TLSv1.2	296 New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
152	8.35585900	192.168.1.4	54.243.74.12	TLSv1.2	513 Application Data
153	8.41144100	192.168.1.4	54.243.74.12	TCP	54 62107→443 [ACK] Seq=174 Ack=3005 Win=65616 Len=0
154	8.47661200	54.243.74.12	192.168.1.4	TLSv1.2	377 Application Data
156	8.50331400	192.168.1.4	54.243.74.12	TLSv1.2	180 Client Key Exchange, Change Cipher Spec, Hello Request, Hello Request
157	8.51062700	192.168.1.4	54.243.74.12	TLSv1.2	510 Application Data



On the first 3 lines you can see the initial three way handshake:
first - the synchronize, with sequence number 0 sent to the server;
second - the response from the server, with sequence number 0, acknowledgment number 1
(value 1 because the server already received the 1st request, which is counted with size 1);

third - the acknowledgment from the client to the server, with sequence number 1 and acknowledgment number 1.

```
Internet Protocol Version 4, Src: 54.243.74.12 (54.243.74.12), Dst: 192.168.1.4 (192.168.1.4)
   Version: 4
   Header Length: 20 bytes
 ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
   Total Length: 363
   Identification: 0x7cd5 (31957)
 ⊞ Flags: 0x02 (Don't Fragment)
   Fragment offset: 0
   Time to live: 45
   Protocol: TCP (6)
```

In this Internet protocol layer we can see size of header and actual size of data. It is obvious that header is just a small piece of actual packet.

```
Secure Sockets Layer
⊟ TLSv1.2 Record Layer: Handshake Protocol: New Session Ticket
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 186
  ⊟ Handshake Protocol: New Session Ticket
      Handshake Type: New Session Ticket (4)
      Length: 182
    ⊞ TLS Session Ticket
⊟ TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
    Content Type: Change Cipher Spec (20)
    Version: TLS 1.2 (0x0303)
    Length: 1
    Change Cipher Spec Message
```

This Secure Socket Layer deals with sessions. We implemented tokens/ sessions in our project and here vie WireShark is nice to see how things actually works and that tokens get transferred successfully.

A summary of the persistence strategies

Since we are using two databases we decided to have two different persistence strategies, so that we can recap everything we learned during this semester. First of all, we decided that most of the data, which should be saved in databases, should be contained in MondoDB. The reason for that is because we believe that JPA's relational database is unstable, it gives random error messages, which are not actually having any impact on the program. On the top of that, if a developer is building a huge project with several tables, which have logical connections, it will become almost impossible to insert changes into the tables, because after every change the whole project should be rebuilt, which does not make any sense to us. After all this persuasive information, we can start with the actual logic behind our code. The mongoDB is using the Table per class persistence strategy. The pros are that the whole information is logically divided and this improves the functionality, because one can easily add new instances. The disadvantages are that, after all, Mongo is non-relational database, so if we want to have any actual

connection between tables we have to delete all entries manually. Also, the whole database is not flexible. Since it's already divided into different parts it will be hard to restructure it. The JPA, Relational Database is completely different. There is a single table in the database, containing all users information, from account and passwords to names and addresses. We are using the single class strategy and we believe that this is suitable solution of the problem, cause we can easily have high amount of users, therefore we need fast access to all information. The positive aspects of the single table strategy is that it is the fastest way of extracting information from the database, however in many situation it can have null values, which could become a problem at one point.

Considerations about reuse

As we already mentioned our project has two servers, which have their own REST interface. Therefore we can always use those REST services to build a whole new structure on the top of them and we can also ensure the reusability of the program. First of all, our JPA Relational database is dealing with the user functionality and the REST server has full CRUD functionality. Therefore this part of our program could be reused to any kind of project in the future. It also has functionality, which we have not used in our actual project, because we decided that we can use the code in the future if we want to further develop that program. The Node-Express server handles both connection with the JPA server and the mongoDB. This REST server also has fully working CRUD functionality, so it can also be reused and developed in order to build a bigger application for distribution websites.

Final thoughts

We know that this project could be made in many different ways and different back end logic. While we were developing this project we found some functionality or logic unnecessary or by looking back we know we could do it differently. Customization of the code was one of our priorities so we can easily oriented in code and also so it looks a bit professional.
If the project would last for longer and we would continue to develop it we would change REST_api logic so functions which is common for admin and user would be in common REST_api.
Also the security of our project is not developed to maximum. We followed teachers' advices so we implement some parts of security logic into the project but we know that it would not be enough for real life system. For example, our hashing logic is seeded in Java server. We did that after discussion with one of the teacher and how he would do it but later on we figure out that hashing should be placed in node server. Same thing is with Java server. Anybody can connect to it anytime and it has no protection from outside internet.
We look on our mistakes as part of this education and so we know where to focus next time while developing comparable system.