PREMIER PROPERTY
MANAGEMENT GROUP

AUBG Property Management

COS 315 - Software Engineering Sprint 1 Report

Martin Nestorov .....

Tatiana Triukhova .....

Edor Kacerja .....

Index

Process Tasks

Product Tasks

3. Code Quality
4. Sequence Diagram

Process Tasks

The following tasks relate to the scrum process.

1. Product Backlog and Sizes

List of the user stories, assigned with *priorities* and estimated sizes in *relative points*.

1.1 As an administrator I want to be able to add new properties for sale or rent to the catalogue.
—(High priority)—
—21 points—

- Design the MySQL database **2pts**
- Create MySQL database **3pts**
- Create user and property tables **2pts**
- Fill it with temporary data to test it **2pt**
- Connect java web app to the DB **4pts**
- Authentication for admin **4pts**
- Add functionality for editing the database by admins **4pts**

1.2 As an administrator, I want to be able to view any offers for a given property.
—(High priority)—
—8 points—

- Create user and offers tables **3pts**
- Fill it with temporary data to test it **2pts**
- Connect java web app to the DB **2pts**
- Grant the authority to read the database by the admin **1pt**

1.3 As a customer I want to be able to browse the catalogue for properties for sale or rent within a certain price range or with a certain number of bedrooms.
—(High priority)—
—21 points—

- Add filters for a price range **8pts**
- Add filter for the number of bedrooms **8pts**
- Restrict access to the DB for normal users **3pts**

1.4 As an administrator I want to be able to delete sold or rented properties from the catalogue.
—(Medium priority)—
—5 points—

- Filter properties in the DB by sold/rented

- Add functionality to delete the properties from the DB

1.5 As an administrator, I want to be able to inform customers that an offer has been accepted or rejected.
—(Medium priority)—
—5 points—

- Add method to send an email to the user who rented/bought a property
- Connect a user account with the property he/she has bought/rented

1.6 As a customer I want to be able to make an offer for a property.
—(Medium priority)—
—5 points—

- Add method to give an offer for a property
- Don't restrict access to it for other users until the offer expires

1.7 As a customer I want to be able to register in the system so that I can make offers for properties.
—(Low priority)—
—3 points—

- Add method to register users when "make an offer" is clicked
- Add method to register at any time during the visit of the app

1.8 As an administrator, I want to be able to divide properties as "for sale" or "for rent".
—(Low priority)—
—2 points—

- Add filtering functionality to administrator for rented/sold properties

1.9 As an administrator, I want to be able to see the profiles of each register customer – name, address, etc.
—(Low priority)—
—2 points—

- Add full reading privileges to admin

1.10 As an administrator, I want to be able to add extra inform, e.g. "No pets" to individual properties.
—(Low priority)—
—1 point—

- Add full editing rights for admin for properties table

2. Sprint Backlog

List of selected user stories to be implemented in Sprint #1.

2.1 As an administrator I want to be able to add new properties for sale or rent to the catalogue. *—(High priority)—*
—21 points—

- Design the MySQL database **2pts**
- Create MySQL database **3pts**
- Create user and property tables **2pts**
- Fill it with temporary data to test it **2pt**
- Connect java web app to the DB **4pts**
- Authentication for admin **4pts**
- Add functionality for editing the database by admins **4pts**

2.2 As an administrator, I want to be able to view any offers for a given property.
—(High priority)—
—8 points—

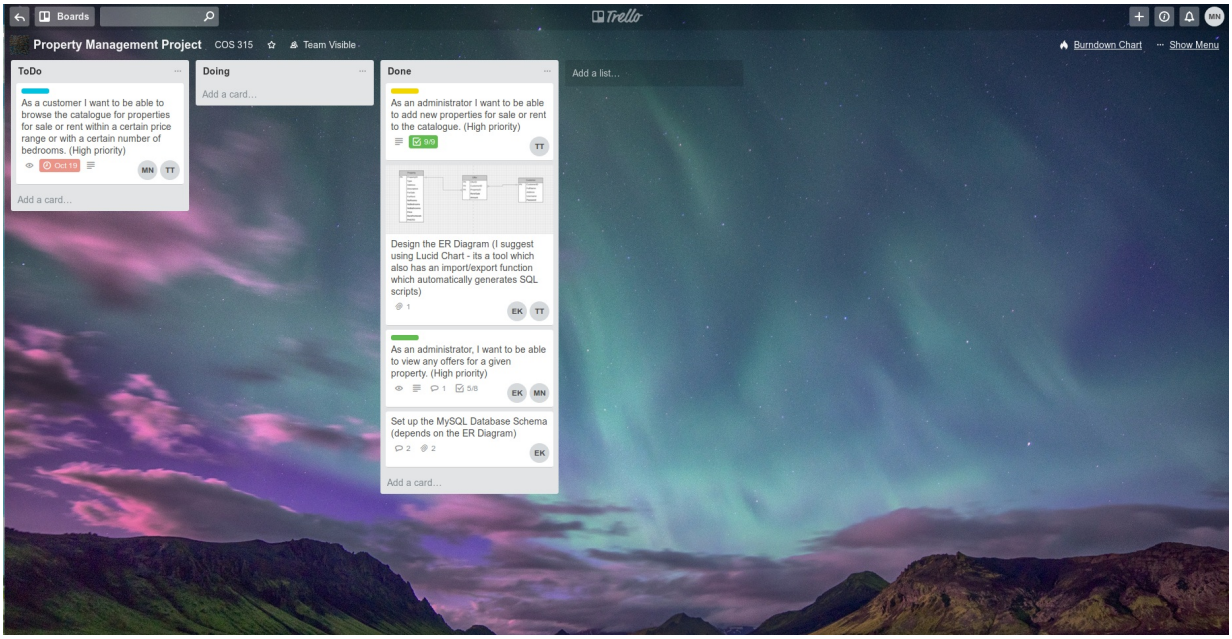
- Create user and offers tables **3pts**
- Fill it with temporary data to test it **2pts**
- Connect java web app to the DB **2pts**
- Grant the authority to read the database by the admin **1pt**

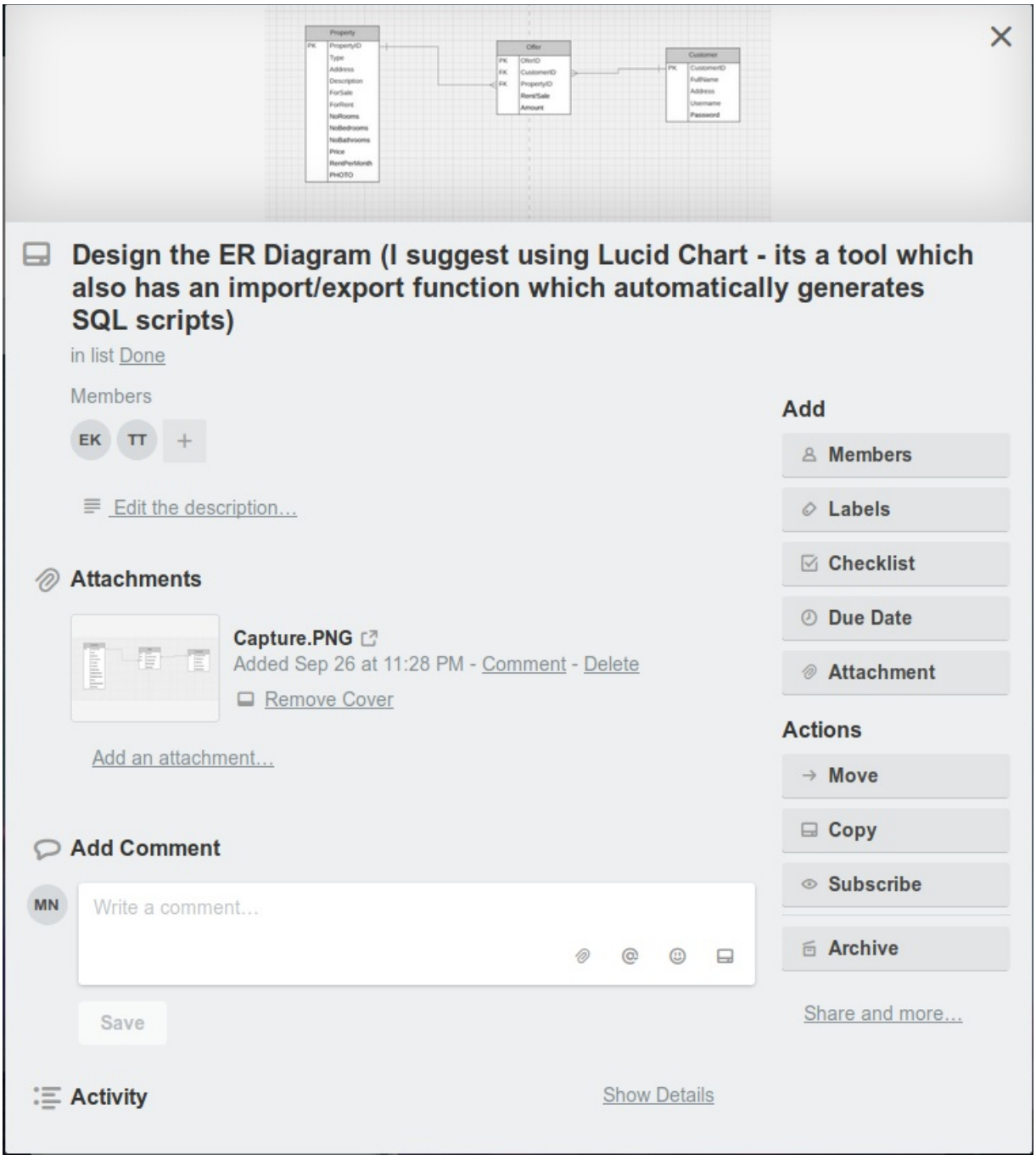
2.3 As a customer I want to be able to browse the catalogue for properties for sale or rent within a certain price range or with a certain number of bedrooms.
—(High priority)—
—21 points—

- Add filters for a price range **8pts**
- Add filter for the number of bedrooms **8pts**
- Restrict access to the DB for normal users **3pts**

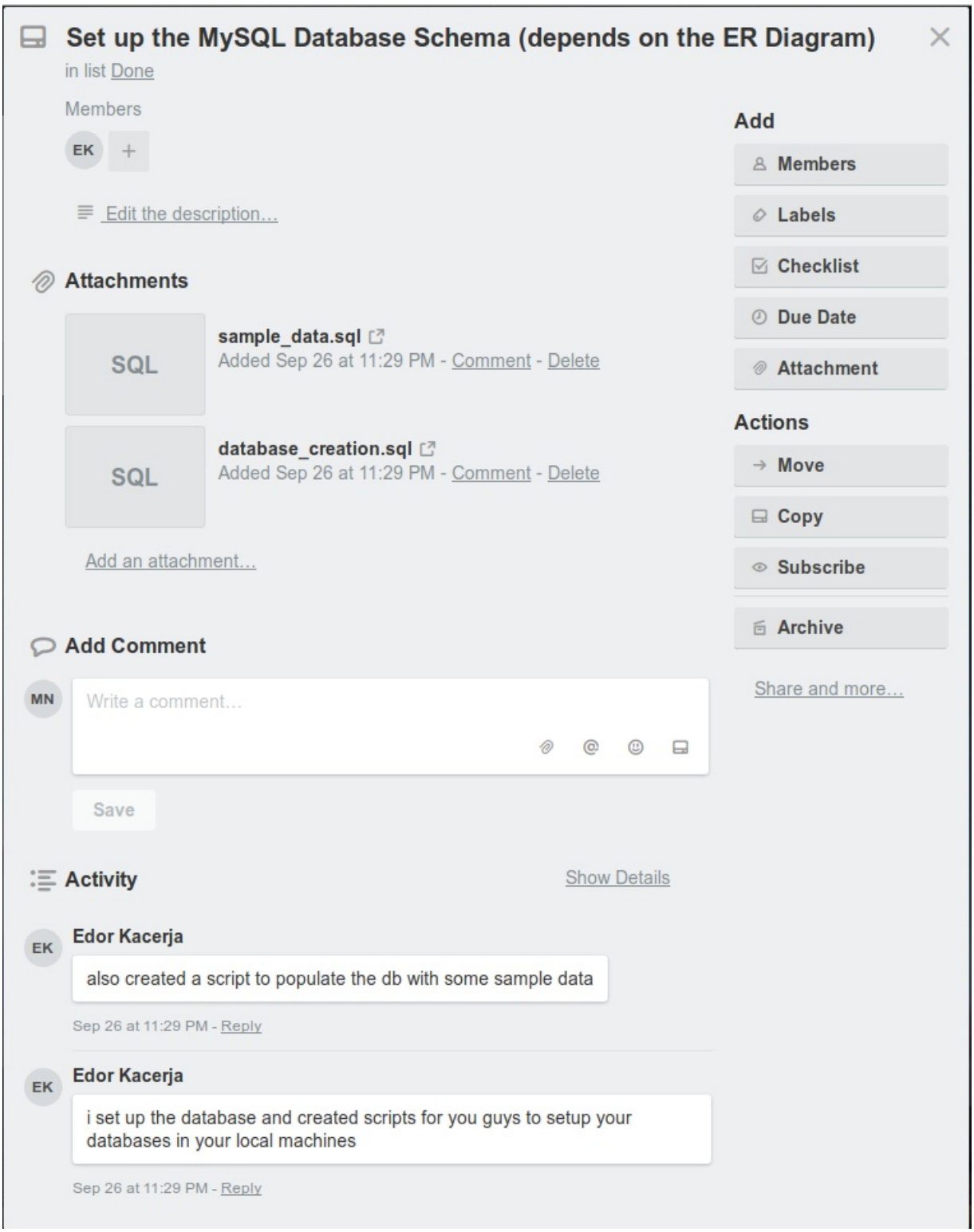
3. Scrum Task Board

Our team uses **Trello** to manage it's Task Board, where we keep track of the current development of the user stories.

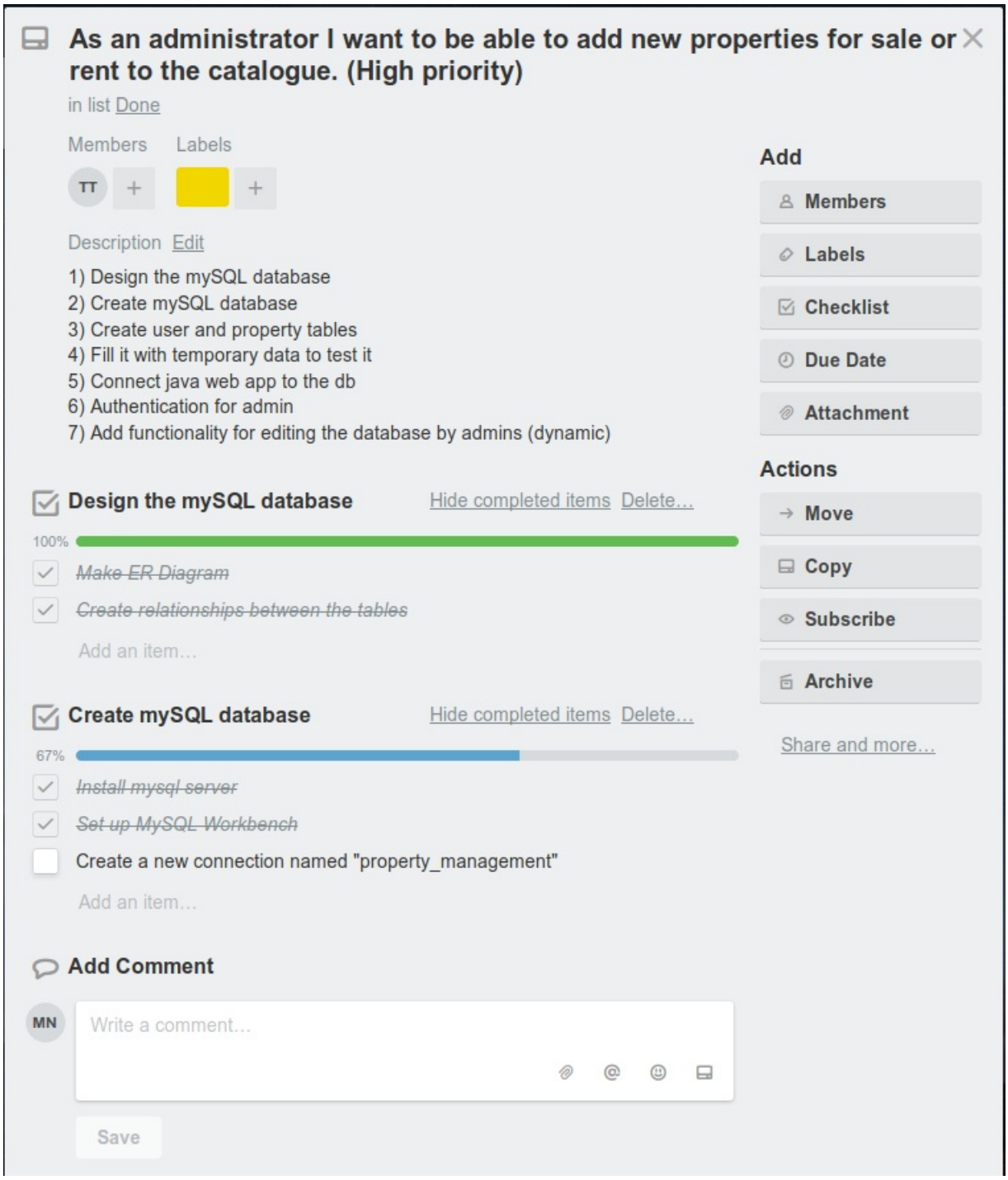




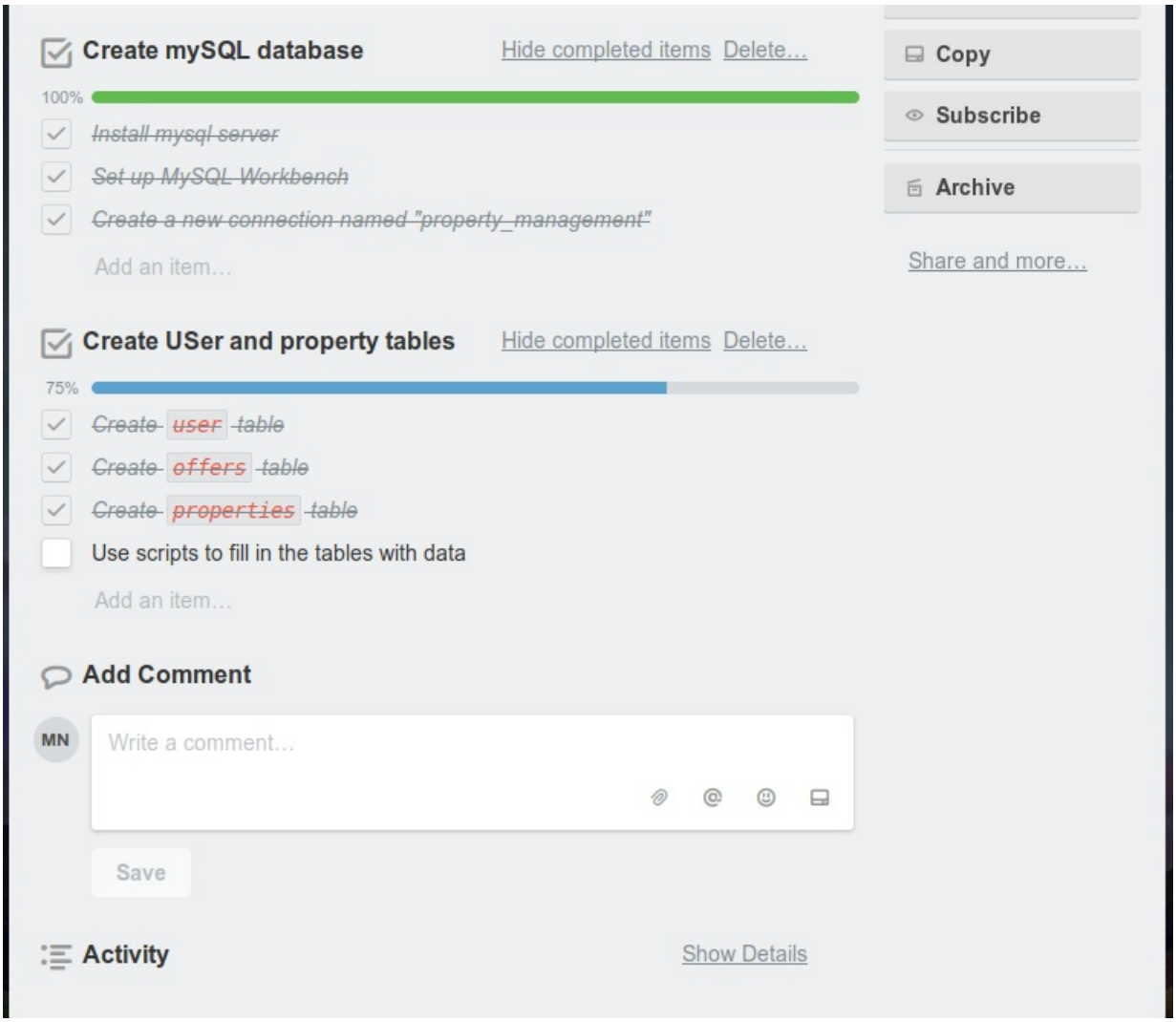
We started thinking about our DataBase early on, where we managed to make some *ER* diagrams to get a better idea of what they should be.



On this part we have provided several scripts that will allows us to automatically import the test data into our DataBase.

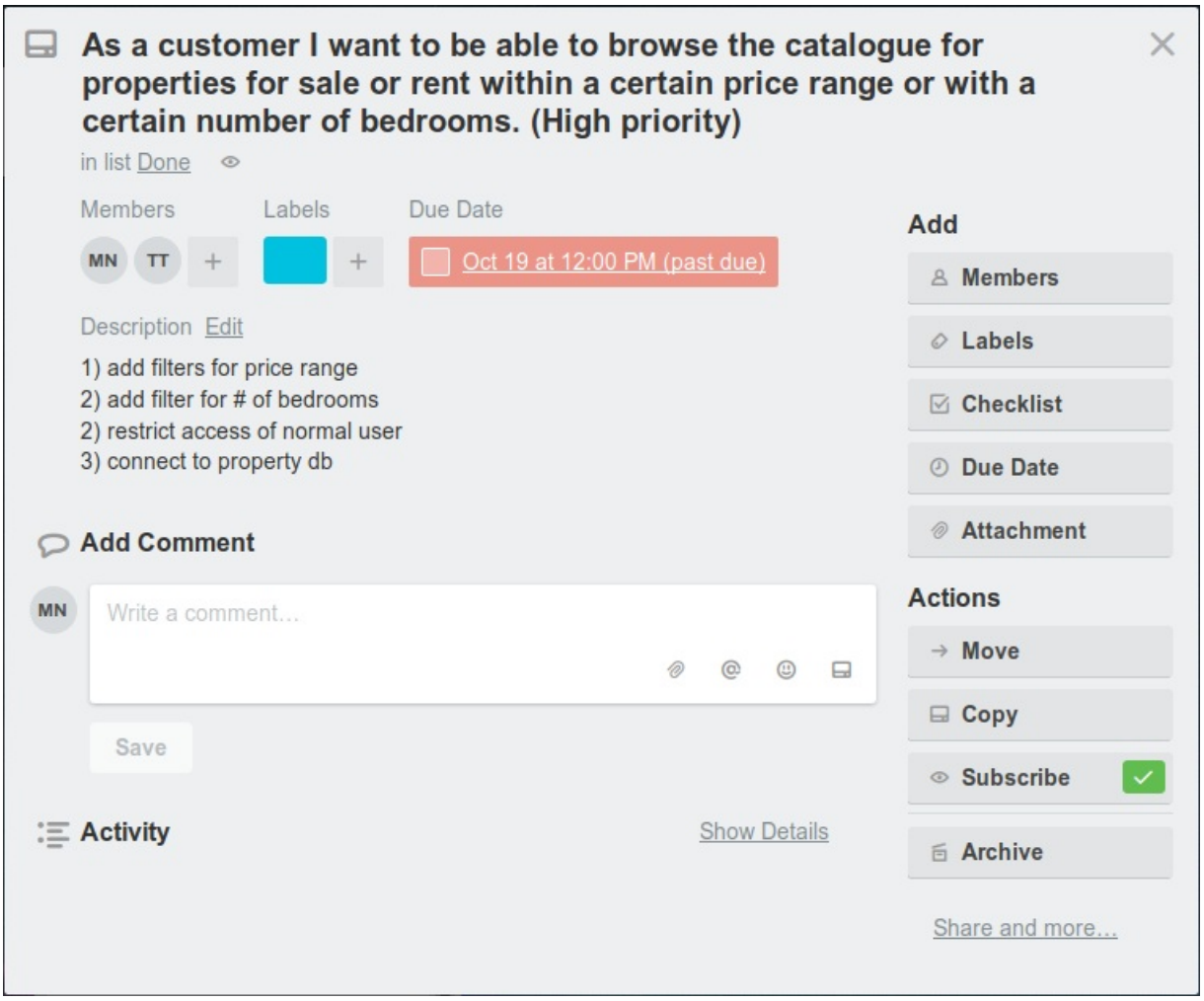


This is how we discretisized our tasks into even smaller pieces for us to work on. We were really concentrated on the DataBase at this point, because it played a big role of the whole set up. Without it working, we could not write any other code.



Finally we managed to get to the creation of the tables and the realtionships between them.

At this point we wrote a lot of code that was connected with the ednpoints of the logging in system and the authentication. Because we knew how *Spring Security* works we were able to quickly blaze through this part.



Here we can see, that we did not manage to reach this user storie and the due date has passed. We did not expect to take so long, that we won't be able to create a full user interface.

4. Burn-down Chart

The Burn-down Chart shows the progress over the course of *Sprint #1*. The Chart represents our team velocity, measured in relative points.

$$1\ relative\ point = 1\ productive\ day\ of\ coding$$

Our team velocity for *Sprint #1* was **50** points. This was calculated by playing Planning Poker and estimating the relative size of the three highest user stories. By adding them up, we estimated that we will manage to do the 50 relative points in our sprint.

*The Burn-down chart was created using the open source website burndowngenerator.com

From the Chart we can extract the following information:
We started the project on time, where we were able to quickly establish a MySQL DataBase. But over the course of the sprint, we started falling behind. This was due to several reasons.

One is that we took too long to connect the Application Backend with the DataBase, which was a difficult part of the sprint. After we managed to have a stable connection, we implemented the backend logic in order to edit, delete, add, etc. tables from the DataBase. This was done on days **4**, **5**, **6**.

On day **8** we were able to compensate some of the time with the successful implementation of the Authentication module on the Backend.

At the end we took too long with the User Interface and could not manage to implement an Interface where we can edit the DataBase in a user friendly manner. We only managed to built a Login System UI.

Product Tasks

1. Testing

For every implemented user story, there is a test plan with a corresponding acceptance test to it and an indication of a pass or fail.

Our tests aim to cover *Functionality*, *Utility*, and *Performance*.

Test 1 — *Editing of DataBase with Admin Privileges*

This test is only applicable to the *Admins* of the DataBase. By using their designated name for full access, they are able to manipulate the DataBase, being able to *add*, *delete*, or *edit* the different property tables in the DB.

Results: Through the stage of testing, we first started with granted admin rights to us, thus we were able to test the different methods that manipulated the DataBase. An admin is capable of having such privileges. The test is successful.

Test 2 — *Accessibility*

This test covers how accessible the application is, how it allows for easy navigation throught the different menus, how easy it is to register or log in, and how fast information is retrieved from the DataBase.

Results: Because the application is still in its early stages, some of the features, such as menu navigation and full booking or buying capabilities are not yet implememted, thus for them, results of the tests cannot be given. There is a working log-in/registration system, where the user can *log-in* or *register*. This system is working and fully integrated into the application.

Test 3 — *Buying/Bidding on a Property*

This test covers the main functionality of our application, where users are able to either buy or bid on specific properties they have selected. By doing so they are able to either imediately access their purchase or be notified if they bid has either been successful or not.

Results: It is too early for us to evaluate the results of this test, due to the early stages of the product. Future tests on this case will be performed evalutaing if this part of the system is integrated and working.

Acceptance Tests

Feature Test 1: As an administrator I want to be able to add new properties for sale or rent to the catalogue.
—(High priority)—

a. *The administrator can login with valid userid and password. Error if either is invalid.*
PASS

b. *The administrator is then able to navigate to a page that allows him/her to add details of new properties for sale or rent. Error if property already exists.*
PASS

c. *Confirmation message displayed that an update to database has been made.*
FAIL

d. *System displays details of newly added property.*
PASS

Feature Test 2: As a customer I want to be able to browse the catalogue for properties for sale or rent within a certain price range or with a certain number of

bedrooms.
—(High priority)—

a. *Customer is able to navigate from home page to a page which prompts the user.*

1. For type of property - sale or rent
2. For a price range
3. For a certain number of bedrooms

And then lists the properties with these features. No login required.
Validation of data entered is required, with appropriate error message if invalid.
FAIL

2. Implementation

Martin Nestorov — For the current Sprint #1 I have worked on mainly the development of the `Authentication` system and `Logging in and Registration` system. I also helped with connection to the DataBase and the manipulation of the `user` table in the DB. With this I worked on the user stories connected with *administration work and access of the database by users*.

The Authentication system utilizes the *Spring Security* Framework. It works by receiving input from the *log in/register* web page. A user is re-directed to the log-in page and by entering his credentials, he is allowed access to the properties. His credentials are send to the `Authentication` module, where his name and password are compared with the DataBase. For the `password` I used a hashing algorithm on `10 000` iterrations, in order to save the hashed password in a secure manner. Nowhere do we work with the *"naked"* password of the user. Several checks are made - if the user actually exists in the DataBase, if the password matches after it has been hashed. After the successful authentication, I pass a `GrantedAuthority` that allows the current user runnig the App Instance to navigate through the different `endpoints` and view the properties.

The `PasswordHashing` class has been created in order to manually control this part of the authentication. This way we can increase the number of iterrations, making the hashing algorithm stronger, and also we are not relying on what *Spring* has implemented underneath its corpus. It turns out that *Spring* uses internal authentication, which is much harder to manipulate and change.

```
/**
 * Authenticate the user that logs in using the credentials from the html page and reading from the DB
 *
 * @param authentication
 *      Spring Security authentication class
 * @return A valid authentication if the user is found with the correct credentials.
 * @throws AuthenticationException
 */
@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {

    String loginName = authentication.getName();

    String password = authentication.getCredentials().toString();
```

```
    User currUser = userService.getUserByUsername(loginName);

    if (currUser.getId() != 0) {

        try {
            String generatedSecuredPasswordHash = PasswordHash.get().generateStorngPasswordHash(passwo

            boolean matched = PasswordHash.get().validatePassword(password, currUser.getPassword());

            if (currUser != null) {

                if (loginName.equals(currUser.getUsername()) && matched) {

                    // It is very important ot pass the List of Granted Authority to the
                    // UsernamePasswordAuthenticationTokens so it can properly send the
                    // WebSecurity authenticator the credentials and let the user in the app.
                    List<GrantedAuthority> grantedAuths = new ArrayList<>();

                    grantedAuths.add(new SimpleGrantedAuthority(currUser.getRole()));

                    LOGGER.info("Successful authentication of " + loginName);

                    // It is necessary to pass in only the username, not the whole user object, becaus
                    // the user is updated in couchDB, the objects revision number increments, so in o
                    // the user object consistently, the endpoints have to read the directly from the
                    // this authenticator.
                    return new UsernamePasswordAuthenticationToken(currUser.getUsername(), password, g
                }
            }

        } catch (NoSuchAlgorithmException | InvalidKeySpecException she) {

            LOGGER.error("Could not find algorithm", she);
        }
    }
    LOGGER.info("Unsuccessful authentication.");

    return null;
}
```

This is the basic Authentication process, where any exceptional cases are handled.

I also helped out with the `User` class, which is able to connect to the DataBase to holds the information for the *normal* users and the *admins*.

The `UserController` allows for the admin to view all of the users that are in the DataBase and there is implemented logic that deletes any unwanted people from the DataBase. New users are added when they register through the registration *User Interface*.

The Controllers logic is done through several classes that utilize the implemented from *Spring* `jdbcTemplate` which allows us to integrate `SQL` scripts in the code and receive objects from the DataBase. This connection is done though *DAO* Interfaces and objects, which implement the logic behind adding, deleting, or editing users in the DataBase.

```
@Override
public User getUserByUsername(String username) {

    String getUserNameQuery = "SELECT * FROM user WHERE username = ?";

    RowMapper<User> rowMapper = new UserRowMapper();

    User user = this.jdbcTemplate.queryForObject(getUserNameQuery, rowMapper, username);

    LOGGER.info("Retrieved user by username");

    return user;
}

@Override
public void addUser(User user) {

    String addUserQuery = "INSERT INTO user (id,full_name,address,username,password,role) values (?,?,
    jdbcTemplate.update(addUserQuery, user.getId(), user.getFullName(), user.getAddress(), user.getUse
            user.getPassword(), user.getRole());

    LOGGER.info("Added user to DB.");
}
```

Tatiana Triukova — Having analyzed our tasks, we decided to use the `JDBC framework`. As it was new to me so first I had to go through a tutorial to understand how it works.

Then, having issues with remote database, I decided to set up the local database and connected it to the application.

```
private final JdbcTemplate jdbcTemplate;

@Autowired
public PropertyDAO(JdbcTemplate jdbcTemplate) {

    this.jdbcTemplate = jdbcTemplate;
}

@Override
public List<Property> getAllProperties() {

    String sql = "SELECT * FROM property";
    RowMapper<Property> rowMapper = new PropertyRowMapper();
    return this.jdbcTemplate.query(sql, rowMapper);
}
```

I first wrote the queries for inserting new property's information into the database and to check whether the database already contains this property.

Then I wrote the `addProperty` method in the `propertyService` class that

incorporates the method for testing the double entries (`propertyExists` method) and the `addProperty` method with queries called from `DAOProperty` class. It also includes the *exception handling* that returns the htttp statuses.

```
    @Override
public void addProperty(Property property) {

    String sql = "INSERT INTO property (property_id,type, address,description,for_sale,for_rent,no_roo
    jdbcTemplate.update(sql, property.getPropertyId(), property.getType(), property.getAddress(),
            property.getDescription(), property.isForSale(), property.isForRent(), property.getNumberO
            property.getNumberOfBedrooms(), property.getNumberOfBathrooms(), property.getPrice(),
            property.getRentPerMonth(), property.getPictureUrl());
}

@Override
public boolean propertyExists(String type, String address, String description, boolean forSale, boolea
    String sql = "SELECT count(*) FROM property WHERE type = ? and address=? and description=? and for
    int count = jdbcTemplate.queryForObject(sql, Integer.class, type, address, description, forSale, f
    if (count == 0) {
        return false;
    } else {
        return true;
    }
}
```

The controller class receives the data through post request and calls the adding method.
In order to make sure that the method works, I tested it with *Postman*.

```
@Override
public ResponseEntity<String> addProperty(Property property) {

    try {
        if (propertyDAO.propertyExists(property.getType(), property.getAddress(), property.getDescript

            return new ResponseEntity<>("The property already exists",HttpStatus.CONFLICT);
        }
        else{
            propertyDAO.addProperty(property);
            return new ResponseEntity<>("The property is created",HttpStatus.CREATED);
        }
    }
    catch(Exception e){

        LOGGER.error("Could not add property.", e);
        return new ResponseEntity<>("Something went wrong. Please, check your request",HttpStatus.BAD_
    }
}
```

Edor Kacerja — On this sprint I mainly worked on creating the database and the ER Diagram and the connection to the web app. We decided to use `JDBC` instead of

`Spring Data JPA`. I have previous experience with `JDBC` so it was more convenient for us to bring our experiences into this project. I created also two scripts so the other team members can create and populate their databases in their local machines quickly. Besides that I also wanted to have working connection to the database and lay out the foundations for the structure of the app.

After I made the connection work with a simple class I tried to query the database. It was successful and then I wanted to structure our project so we can all be in the same page and everybody can contribute productively.

I created 4 modules which consist of *DAO*s (Data Access Object), *Models*, *RowMappers*, and *Services*. All of these modules (except the Models), have interfaces and their respective implementation in another subfolder.

```
@RestController
public class PropertyController {

@Autowired
private IPropertyService propertyService;

@RequestMapping("/")
public ResponseEntity<List<Property>> getAllProperties() {
    System.out.println("properties get controller");
    List<Property> list = propertyService.getAllProperties();

    return new ResponseEntity<>(list, HttpStatus.OK);
    }
}
```

My goal was to make a working sample which query's the database and retrieves all the properties while following the structure that I just layed out.

The entry point is of course the controller in my case `Property Controller`.

Through a Service named `propertyService` we call its method to retrieve all properties and assign them in a list. This Service includes all the methods that we need for the business logic.

```
public interface IPropertyService {

    List<Property> getAllProperties();
    Property getPropertyById(int propertyId);
    boolean addProperty(Property property);
    void updateProperty(Property property);
    void deleteProperty(int propertyId);

}
```

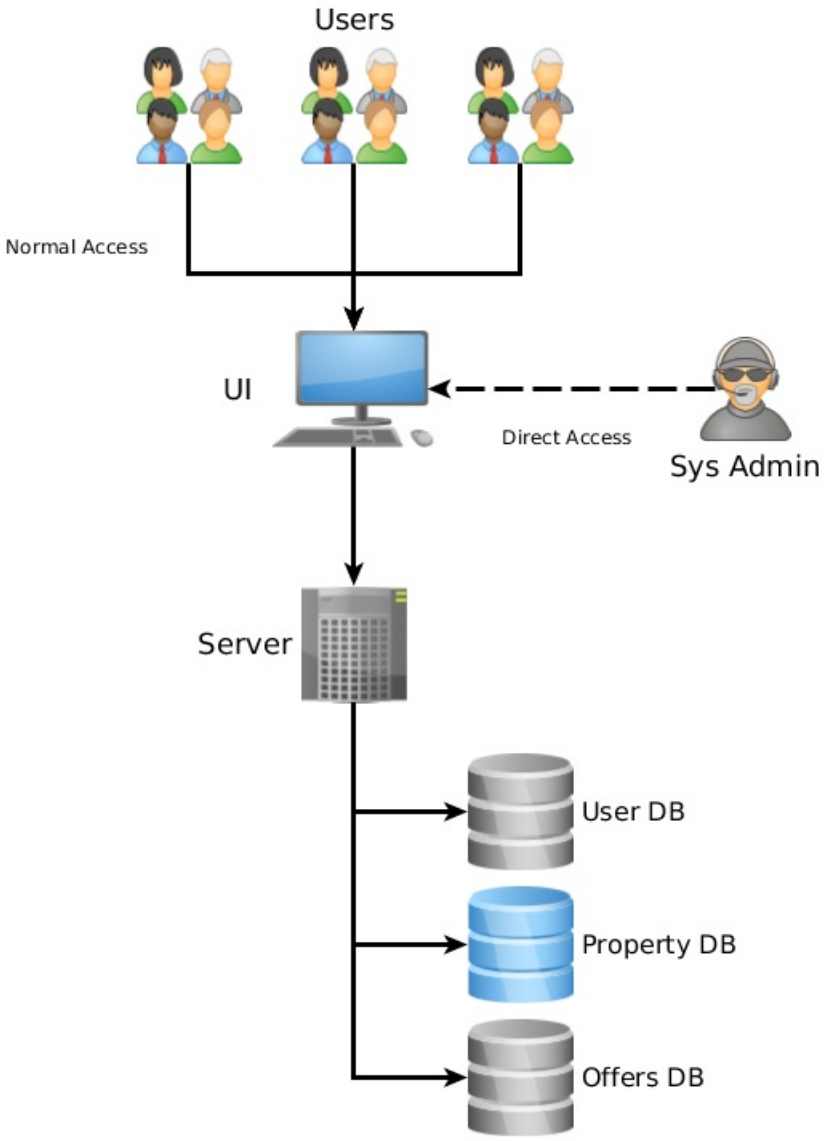
The implementation of these methods is done in PropertyDAO class where the database is queried through jdbc and data is returned.

3. Code Quality

Our team relies on a unified coding standard typical for the *Java* and *Spring* Framework. All of us are using the *Camel Case* standard, our code is supplied with JavaDoc comment blocks, and we have separated our classes in the *MVC* standard into the Model, View, and Controller sections.

4. Sequence Diagram

In the following diagram, we can see what the general idea of the operation of the application is.



The users and the admins are both going to access the same user interface that is developed, with the difference being that the admins will have several extra buttons, allowing for more actions. Through the UI, we are able to connect to the server, which is then connected to our DataBase.