Logiweb

Technical Solution Description

**Table of contents**

[1. Overview 3](#_Toc64906831)

[2. Frameworks and technologies used 5](#_Toc64906832)

[3. Project model 6](#_Toc64906833)

[3.1. Basic entities 6](#_Toc64906834)

[3.2. Functionality 7](#_Toc64906835)

[4. Implementation 8](#_Toc64906836)

[4.1 First part 8](#_Toc64906837)

[4.1.1 Database 8](#_Toc64906838)

[4.1.2. Entities 10](#_Toc64906839)

[4.1.3. DAO 10](#_Toc64906840)

[4.1.4 Business logic 10](#_Toc64906841)

[4.1.5 UI 13](#_Toc64906842)

[4.2 Second part 14](#_Toc64906843)

[4.2.1 Business logic 14](#_Toc64906844)

[4.2.2 UI 14](#_Toc64906845)

[4.3 Interaction between application parts. 14](#_Toc64906846)

[5. Testing 15](#_Toc64906847)

[6. Features 16](#_Toc64906848)

[7. Security 17](#_Toc64906849)

[8. Logging 18](#_Toc64906850)

[9. Deployment 19](#_Toc64906851)

# 1. Overview

Logiweb is a two-part web application that simulates the operation of the system of a certain company carrying out the transportation of goods.

The first part is a website that allows authorized users perform some actions based on their roles. User can be either a manager or a driver. Managers are responsible for creating orders and are capable of editing drivers’ profiles and cars’ records. Drivers are responsible for delivering the cargo to its destination. There is also an administrator role, but it’s more of a utility role and hardly differs from manager role in terms of business-tasks.

The second part of the application is a table of statistics. It is deployed separately from the first part. This table shows overall statistics for cars/drivers and displays full information about the latest orders.

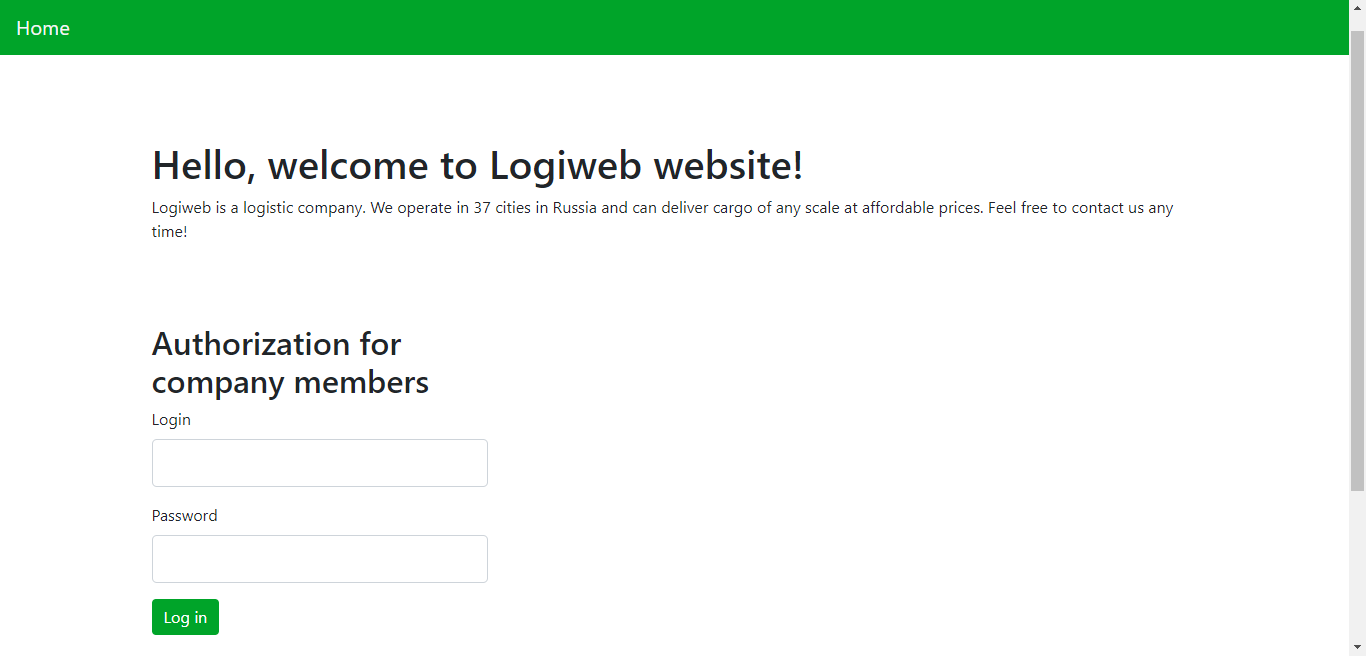


Figure 1. Main page of the first part of the application.

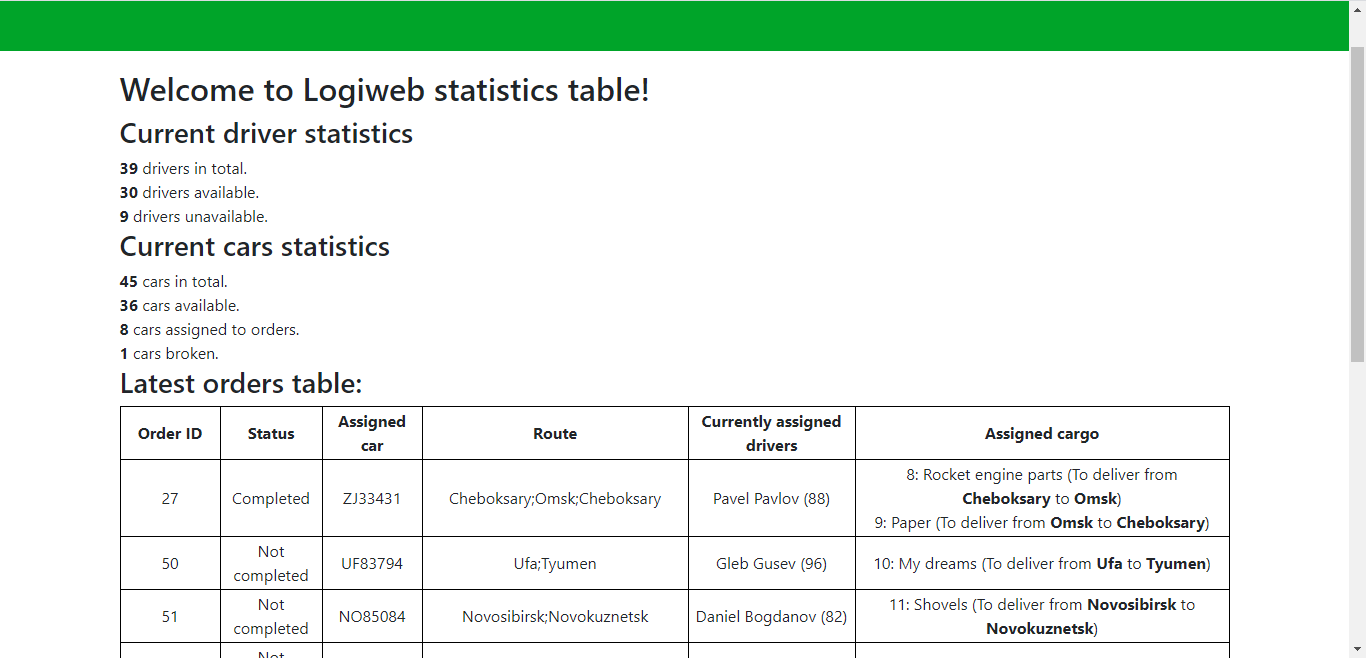


Figure 2. Main page of the second part of the application.

# 2. Frameworks and technologies used

* Java 8
* Spring 5.3.2
* Junit 4.13.1 (for testing only)
* Log4J 1.2.17
* Hibernate 5.4.26 Final
* H2Database 1.4.200 (for testing only)
* MySQL 8
* JSP 2.3
* Hamcrest 1.3 (for testing only)
* JSTL 1.2
* Mockito 3.7.7 (for testing only)
* Java EE 8
* EJB 3.2
* JSF 3.0
* Bootstrap 4
* Maven 3.6.3
* Tomcat 9.0.41 (application server for 1st part)
* WildFly 22.0.0 Final (application server for 2nd part)
* ActiveMQ 5.16.1
* SOAP 1.1

# 3. Project model

## 3.1. Basic entities

These entities are essential to application’s work:

* Car
  + Registration ID (2 Latin letters + 5 numerals)
  + Driver shift length
  + Lifting capacity (tons)
  + Condition (broken/normal)
  + Current city
* Driver
  + Name
  + Surname
  + Personal number
  + Hours worked this month
  + Status (resting, on shift, driving, loading/unloading cargo)
  + Current city
  + Current car
* Order
  + Unique number
  + Is completed (yes/no)
  + List of route points
    - City
    - Cargo
    - Action (load/unload)
  + Car assigned to the order
  + List of drivers assigned to the order
* Cargo
  + Cargo number
  + Name
  + Mass (kg)
  + Status (prepared, loaded, unloaded)
* Country map
  + Cities
  + Distances

## 3.2. Functionality

* For managers (using UI-interface):
  + viewing the list, adding, editing and deleting trucks, drivers;
  + viewing the list and adding new orders with a check that:
    - all loaded cargo will be dropped somewhere;
    - all dropped cargo will be loaded somewhere;
  + viewing the status of orders and cargo;
  + displaying the list of cars assignable to orders, if:
    - car is not broken;
    - car has enough capacity (taking into account the loading / unloading of cargo in cities along the route);
    - car is not assigned to any other order at the moment;
  + selection and appointment of drivers based on the size of the change of the used truck and the approximate travel time (calculated on the map of cities and waypoints):
    - the time limit per month (176 hours) for each of the drivers in the shift will not be exceeded during the execution of this order (also take into account the change of months during the order);
    - the driver is not currently fulfilling other orders;
    - the driver is in the same city as the truck upon assignment.
* For drivers (using UI-interface):
  + to display the next information:
    - driver’s personal number
    - personal numbers of co-drivers
    - car’s registration ID
    - order’s number
    - list of route points
  + to change the actual working hours and order status:
    - the driver has started / finished the shift
    - driver has changed status:
      * driving
      * loading and unloading cargo
      * resting
    - driver received / unloaded cargo (change order status)
      * loaded
      * unloaded

# 4. Implementation

## 4.1 First part

### 4.1.1 Database

The database schema is shown in Figure 3 and the description of tables is listed in Table 1.

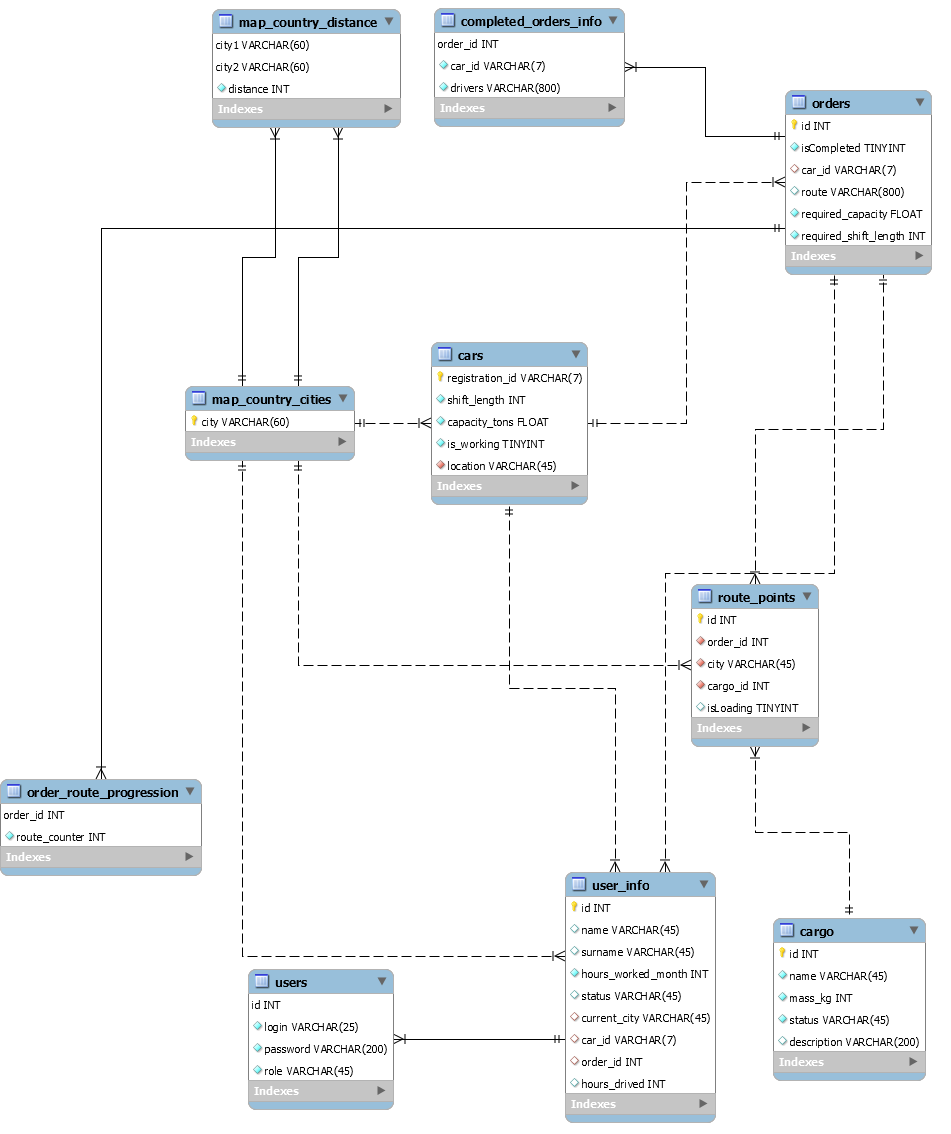


Figure 3. Database schema.

|  |  |  |
| --- | --- | --- |
| Table name | Description | Relations with other tables |
| users | Contains only user credentials and role – used for authorization | user\_info – 1:1 |
| user\_info | Table with more detailed information about users (name, surname, assigned cars/orders etc.) | users – 1:1  map\_country\_cities – M:1  cars – M:1  orders – M:1 |
| cars | Information about cars | user\_info – 1:M  map\_country\_cities – M:1  orders – 1:M |
| orders | Information about orders | user\_info – 1:M  order\_route\_progression – 1:M  route\_points – 1:M  cars – M:1  completed\_orders\_info – 1:M |
| completed\_orders\_info | Additional information to store for completed orders. The reason for this table to exist is that such fields as car\_id are often acting as foreign keys and they should be nullified when order is complete (in order to unassign car from that order and allow it to be selected for the new one) | orders – M:1 |
| route\_points | Information about route points for existing orders | map\_country\_cities – M:1  orders – M:1  cargo – 1:M |
| cargo | Information about cargo for delivering | route\_points – M:1 |
| order\_route\_progression | A table with additional info about order’s progress (what part of route is already visited). Provides more friendly UI for drivers | orders – M:1 |
| map\_country\_cities | List of cities where the company operates | user\_info – 1:M  cars – 1:M  route\_points – 1:M |
| map\_country\_distance | List of distances between pairs of cities | map\_country\_cities – M:1 |

Table 1. Description of tables.

The relations between tables are based only on their configuration (foreign keys etc.).

### 4.1.2. Entities

Since Hibernate is used, entities are defined by Hibernate annotations. Hibernate is configured with combination of both Java-config and XML-config.

It should be noted that entities for Hibernate are mapped differently from MySQL relations. For example, cars are related to orders as 1:M in MySQL but in Hibernate they are mapped as 1:1 because there is a given restriction that car selected for order must not be assigned to any other order.

Please refer to documentation for additional info on the entities.

### 4.1.3. DAO

DAO is defined by implementing a generic interface for entity class. 4 basic operations are had to be implemented – add, delete, read (single row), update. There is a 5th operation – read all but it is optional to define (default method will throw exception).

For most entities the implementation of operations above will do “create session, create transaction, perform operation, commit transaction, close session” sequence. The only exception here is DAO for orders and route points. They will require an active Session object before they can perform DAO operations. The reason is that new order has to be added with route points within the same session due to complex foreign key relationship.

### 4.1.4 Business logic

Algorithm for adding/editing/deleting user is quite simple:

1. Check authorities.
2. Validate input.
3. Proceed if everything is OK.

However, creating order is a lot more complex:

1. Validate input for submitted list of route points.
2. If input is valid, application will try to calculate an optimal path for delivering all cargo. Optimal path calculation is basically solving the "Traveling Salesman Problem" with additional restrictions, such as:
   1. City of unloading for cargo X cannot be visited before city of loading for cargo X;
   2. Only loading cities can be starting points;
   3. Avoid starting from city cycles (that means that city X is city both for loading and unloading come cargo) at all costs;

In common case we have open version of TSP (no need to return back to starting city), if not specified directly. All restrictions above are based on three reasons: 1) to avoid infinite loops during calculation 2) to make found routes applicable to the cause 3) to save extra time (the guesses are hypothetical and not proved mathematically).

[Simulated annealing algorithm](https://en.wikipedia.org/wiki/Simulated_annealing) is used for finding an optimal path. Basically, it is a probabilistic algorithm (he is trying to guess the best solution) but it provides an optimal solution (thought it’s not guaranteed to be the best).

If required, Dijkstra algorithm is used to calculate distance between two cities (because in DB only real “roads” is stored).

The path itself is stored as String (cities names with special delimeter).

1. Application is calculating the required lifting capacity for the car to deliver cargo. This is done by “visiting” all cities along the route with adding/subtracting masses of cargo at each point.
2. All cars are filtered to match 2 conditions – 1) car must be located in the first city of the route. 2) car capacity must be more or equal to capacity calculated in step 3.
3. If no cars are found – error is shown to user. Otherwise, for each car application is trying to simulate the movement of the car with assigning drivers (considering month change and car shift limit). If there is a solution, car and list of assigned drivers is accepted as one of possible answers. Otherwise, car is excluded.
4. From all remaining cars the car with minimal capacity is chosen. This car with the drivers selected from step 5 will be assigned to the new order.
5. Application adds new order, route points, route progression, updates the selected car and drivers.

Drivers and delivering cargo.

The driver can move between cities with special method (or button in UI) (because this is a simulation after all). Driver can only move towards the next city on the order route. Driver can’t move unless he loaded/unloaded all cargo for the city he’s currently in. Driver can change his status manually but after some actions his status will be updated automatically (for example, driver loaded cargo – status has been changed to “loading/unloading cargo”, driver changed city – status has been changed to “driving” etc.). At some point driver might reach shift length capacity (for car assigned). That means he can drive no longer and another driver will take the car from now on.

Order is counted completed when all cargo assigned to it has status “unloaded”.

Table 2 is the table with description of each business-logic service in the application.

|  |  |
| --- | --- |
| Service name | Description |
| CargoService | Responsible for adding, reading and updating cargo. |
| CarService | Responsible for adding, reading, deleting and updating cars. |
| CityService | Responsible for reading cities. |
| DriverService | Responsible for updating driver status/location/etc. |
| OrderService | Responsible for creating, reading and updating orders |
| UserService | Responsible for adding, reading, deleting and updating users. |

### 4.1.5 UI

UI is implemented with usage of JSP, JS and CSS. Navigation bar at the top is represented as separate .jsp file and included in all other page via tag <jsp:include>. Links displayed in navbar are depending on user’s authorities (for example, driver won’t even see links to orders page). CSS styles are mostly Bootstrap 4. JS is just manually-written scripts.

## 4.2 Second part

### 4.2.1 Business logic

The statistics data is loaded on the start of application (if possible) and then only when a message is received from MQ. Depending on the message, one of three updates is requested – for drivers, orders or cars. Data is stored in application-scoped beans.

### 4.2.2 UI

Websockets are used in UI to refresh information on the page without reloading the page itself.

## 4.3 Interaction between application parts.

The interaction between the website and a statistics table is shown in Figure 4. When information update occurs, website sends message to MQ server. When the second part receives it, a SOAP request is sent back to the first part and a SOAP response with new information is received.

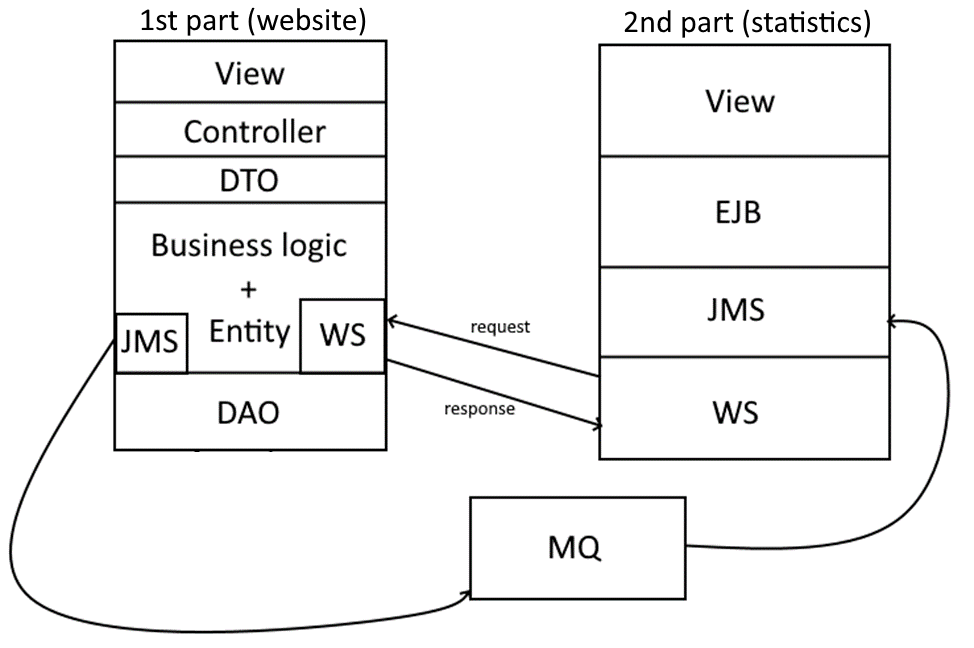


Figure 4. Interaction between the first and the second part of application.

# 5. Testing

There are 171 JUnit tests with a total coverage of 95.6% for all application (DTO, entity and configs are excluded from the statistics).

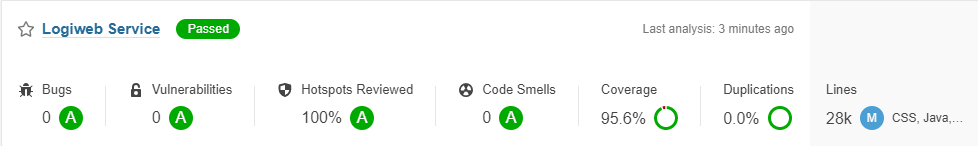


Figure 5. SonarQube scan result.

Tests are split into packages and their names (both classes and test method) are self-explaining.

Controller tests are not unit-tests, these are more of an integration tests (they cover both controllers and service packages).

# 6. Features

* Automatic calculation of path, assigning car and drivers when creating order.
* Custom error pages (500, 403, 404)
* Websockets
* Application is fully deployed to Google Cloud

# 7. Security

Security is configured with Spring Security. All pages (except home page and WS) have restricted access. Additional security checks are done manually (for example, when editing user, in case someone will change page code to edit admin).

# 8. Logging

Logging is configured to display log information both to console and log file.

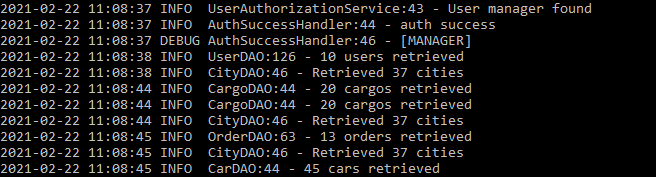


Figure 6. Console log display.

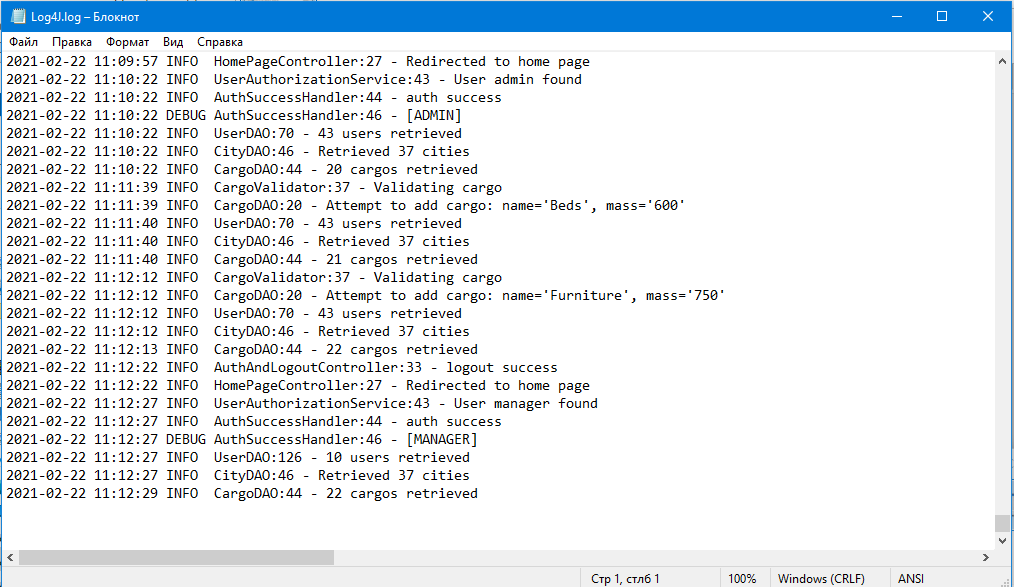


Figure 7. Log file.

# 9. Deployment

Testing, building and deployment is done by Maven. Projects are configured to be deployed separately (first part is deployed to Tomcat Server and the second part is deployed to WildFly server).