Step by Step Guide to the Bicycle Workshop Application

Welcome

As **Learning by Doing** is the most effective way to get started with a new framework, I will guide you through the key features of the CUBA Platform framework and show how you can accelerate development of enterprise applications, taking bicycle workshop system as an example.

Today we will create an application for a small bike workshop. At Haulmont, we like bikes and often discuss spare parts and upcoming races in the office, and our bike parking is never empty. We will develop the application based on the CUBA platform and it will be so good that you can start using it immediately as intended. So, just two hours and it's in production!

So, a few words about the platform. CUBA Platform is a high level Java framework for rapid enterprise software development. It is based on Vaadin, Spring and EclipseLink and provides a rich set of features and development tools on top. Applications are developed in Java using popular IDEs, complemented by CUBA Studio – a development tool that offers a quick way to configure a project and

describe data model, manages DB scripts, enables scaffolding and visual design of the user interface screens.

We will use CUBA Studio both to save time and get acquainted with the platform. For the implementation of the application business logic we will use Intellij Idea IDE, which is also installed on my PC.

To sum it up, we assume, that you all have installed CUBA Studio, Intellij Idea IDE and Java 8, as CUBA platform will work with Java 8 only. If so, let's begin.

About CUBA

CUBA Platform is designed for the development of enterprise applications characterized by a complex data model, dozens or hundreds of screens, complex business logic, as well as the requirements to control access rights, scalability and fault tolerance.

A wide range of built-in functionality, advanced code generation, visual interface designer, and support for hot deployment enables you to reduce the time and cost of development dramatically.

The platform is built on a complete stack of open Java technologies, which enables you to use the know-hows of the largest free software ecosystem in the world, as well as to control the source code of the entire stack.

An open architecture enables you to override the behavior of most of the platform mechanisms, ensuring high flexibility. Developers can use popular Java IDEs and have full access to the source code.

Platform-based applications can be easily integrated into your IT infrastructure with support for major databases and application servers, as well as the ability to work in the cloud. The platform ensures fault tolerance and scalability, and provides integration with external systems.

Base platform code is distributed under open source Apache License 2.0 license.

Task

Imagine that your friend who is a bike mechanic decided to expand his business, hire a couple of smart guys and to open a proper workshop. He could not find a suitable solution so he asked you to develop a simple system to record orders, notify clients and keep track of spare parts.

So what does a bike workshop need? First of all record customers with name, mobile phone, and email to notify about order status. Next record information about orders, a price for repair and time spent by a mechanic. And in addition to this, keep track of spare parts that are in stock. Finally, there is a need for some reports to control the business.

These simple requirements of a small repair shop actually result in a full scale business application. After the session, our application will have a Rich Web UI, be Ajax driven, perform basic CRUD operations, contain the business logic for calculating prices, manage user access rights (our users are mechanics and a shop manager), present data in the form of reports and charts, send notifications and allow customers to check the order status on a separate webpage.

1 FUNCTIONAL SPECIFICATION

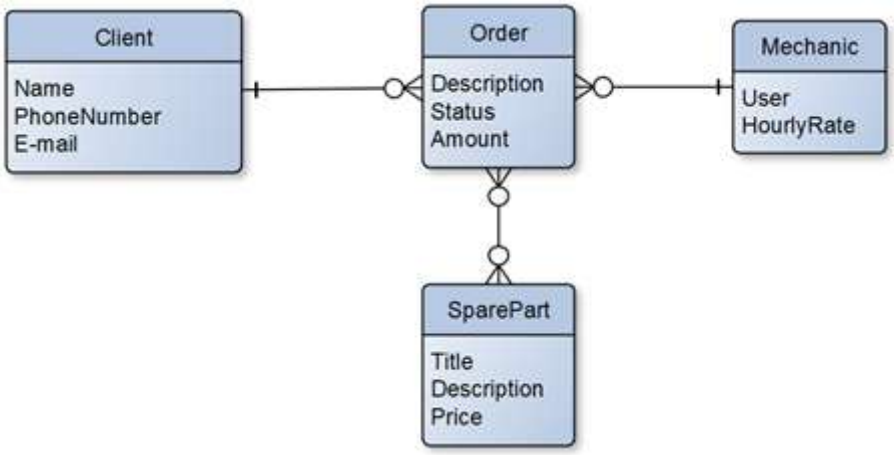
In terms of **functional specification**, the application should meet the following requirements:

1. Store customers with their name, mobile phone and email
2. Record information about orders: price for repair and time spent by mechanic
3. Keep track of spare parts in stock
4. Automatically calculate price based on spare parts used and time elapsed
5. Control security permissions for screens, CRUD operations and attributes of records
6. Perform Audit of critical data changes
7. Provide API for a mobile client to place an order

Model

Let’s have a look at the data model of our application. The main entities of the model are Client, Order, Mechanic, and SparePart. The Order entity has status presented by the OrderStatus enumeration, description, number of hours spent by the mechanic,

required spare parts and final amount. This is a pretty simple model which we will create using CUBA Studio.



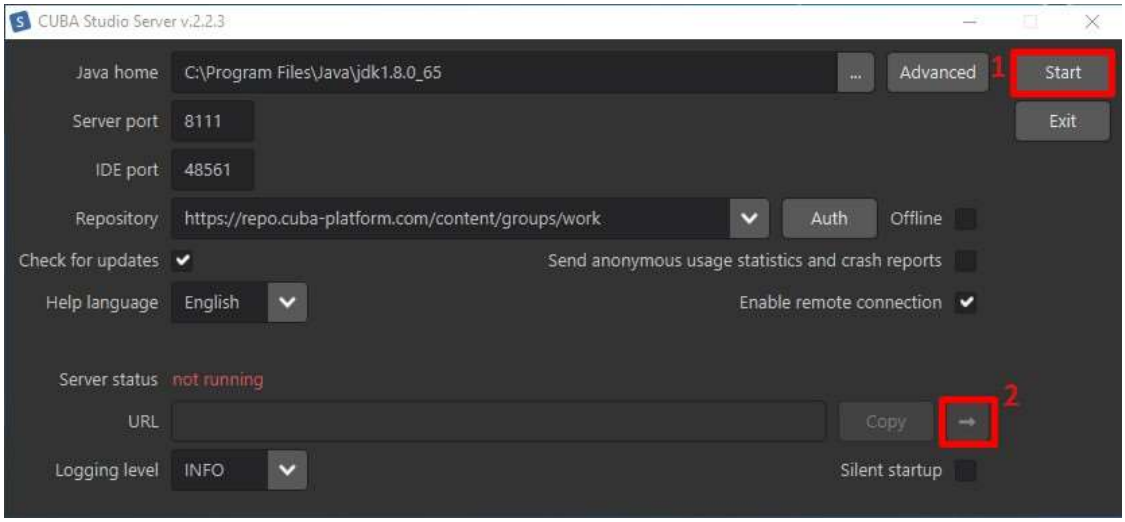
Development Environment

CUBA Studio is a standalone application, which can be used by the developer in parallel to the usual Java IDE. Studio provides a graphic interface for the mechanisms of the platform, enabling the creation of the data model, DDL scripts generation, drawing screens in the WYSIWYG-editor, and creating stubs of middleware services.

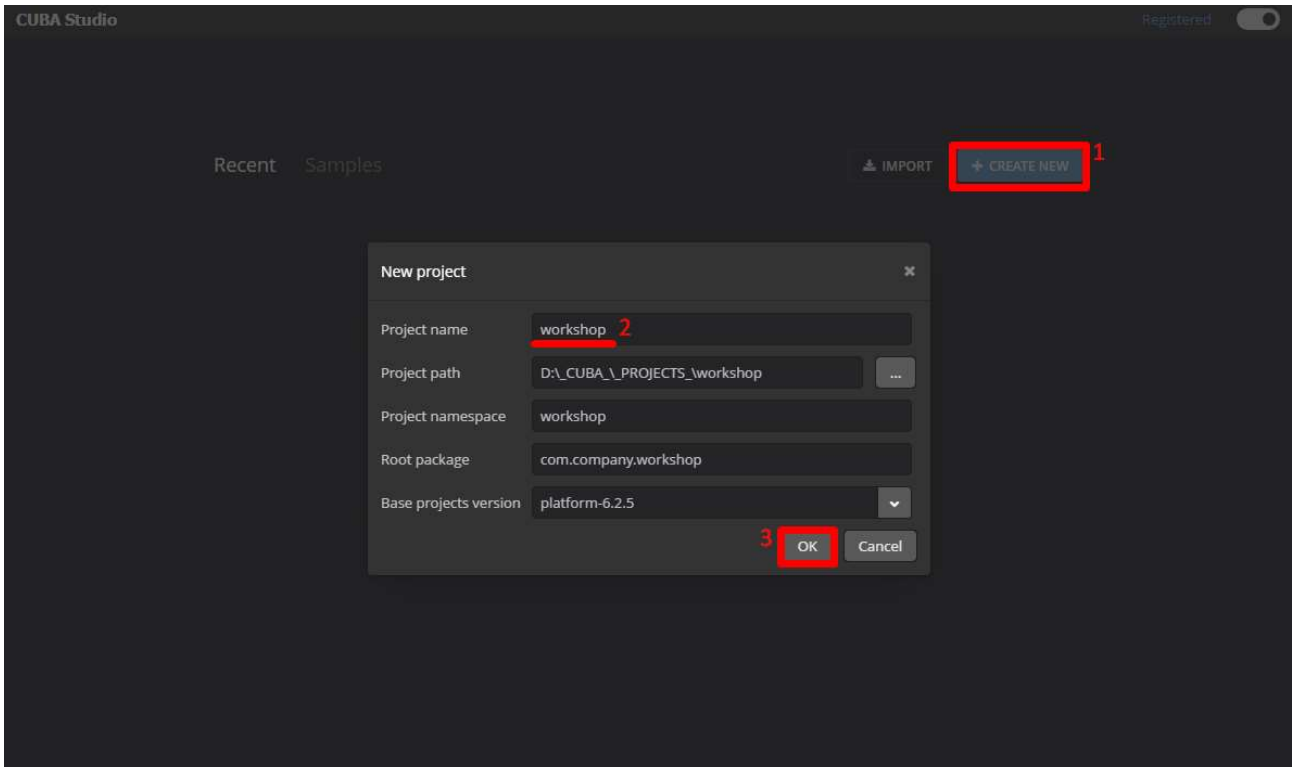
We will use CUBA Studio both to save time and get acquainted with the platform. For the implementation of the application business logic we will use Intellij Idea IDE, which is also installed on my PC. Using Studio is not required but it speeds up a lot of operations, especially at the project initial stage.

2 CREATING A NEW PROJECT

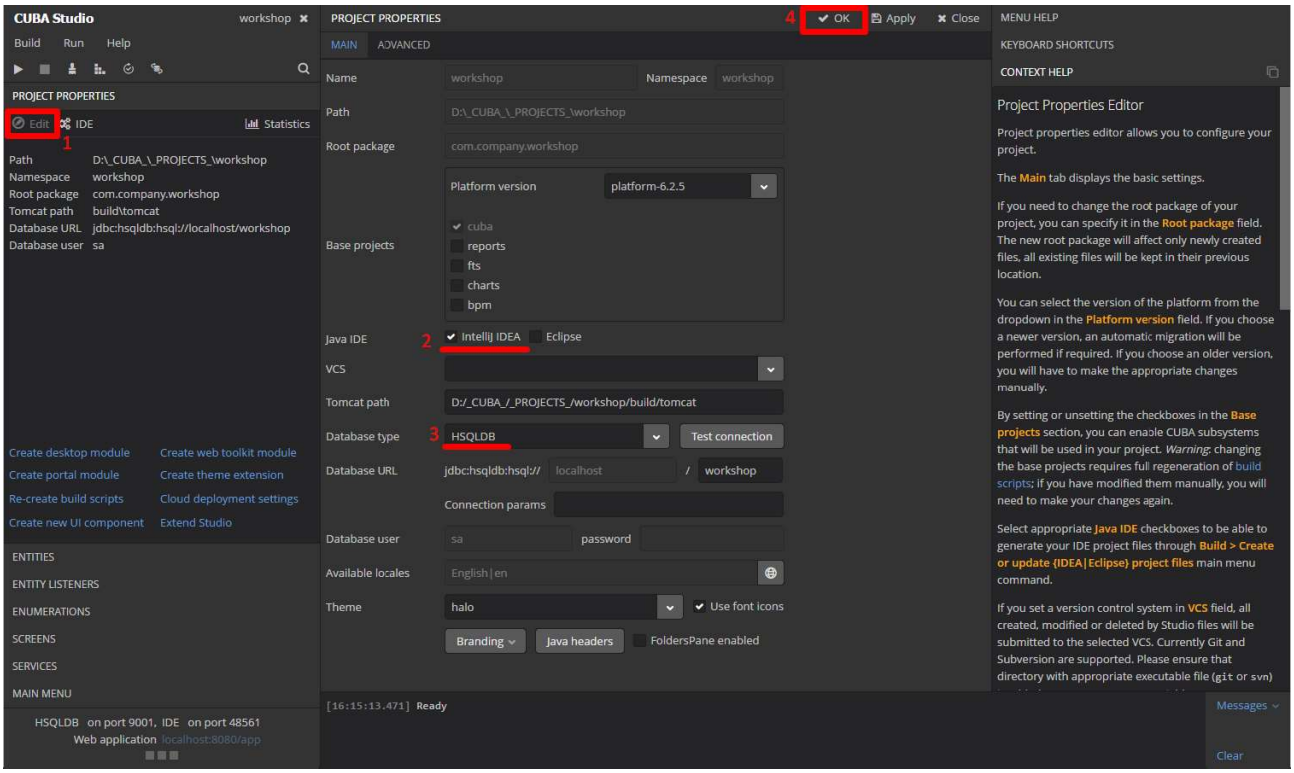
1. Start the CUBA Studio server and open the Studio in your browser by clicking the **arrow** button as it is shown in the picture below:



2. After completing the previous step you can find the CUBA Studio up and running in your browser on the *http://localhost:8111/studio/* URL. Let's create a new project. To do this,
    2.1. Click **Create New** on welcome screen
    2.2. Let's specify our project parameters, fill up the **Project name** field: workshop
    2.3. Complete the step by clicking **OK**



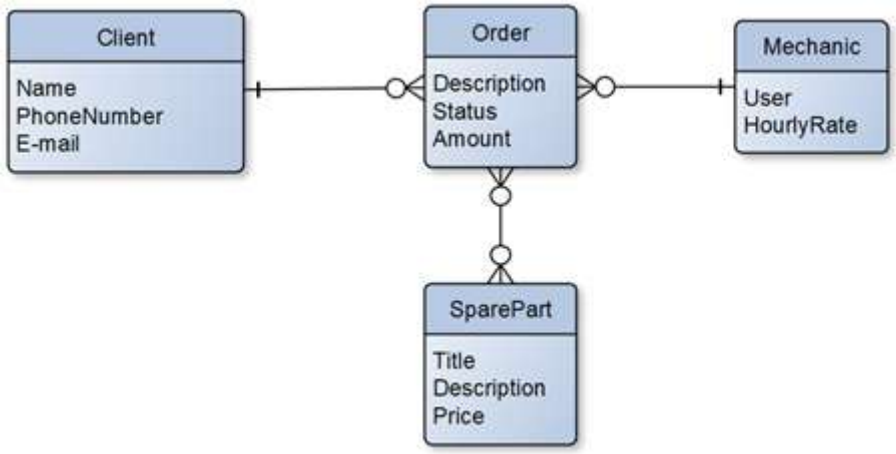
3. Now we will set up global properties of the project. Press **Edit** on the **Project Properties** panel and check the following parameters:
    3.1. Java IDE: **IntelliJ IDEA** to be checked
    3.2. Database type: CUBA platform supports Postgres, MS SQL, Oracle and HSQL databases. In this project we will use HSQL database. **HSQL** to be selected
    3.3. Press OK



Now our project is properly configured and everything is ready to start working on the data model.

3 DEFINING DATA MODEL AND CREATING THE DATABASE

Now let's move on to creating the data model of our application. According to the ER diagram we will create 4 entities: **Client, Mechanic, SparePart** and **Order** and one enumeration **OrderStatus**.

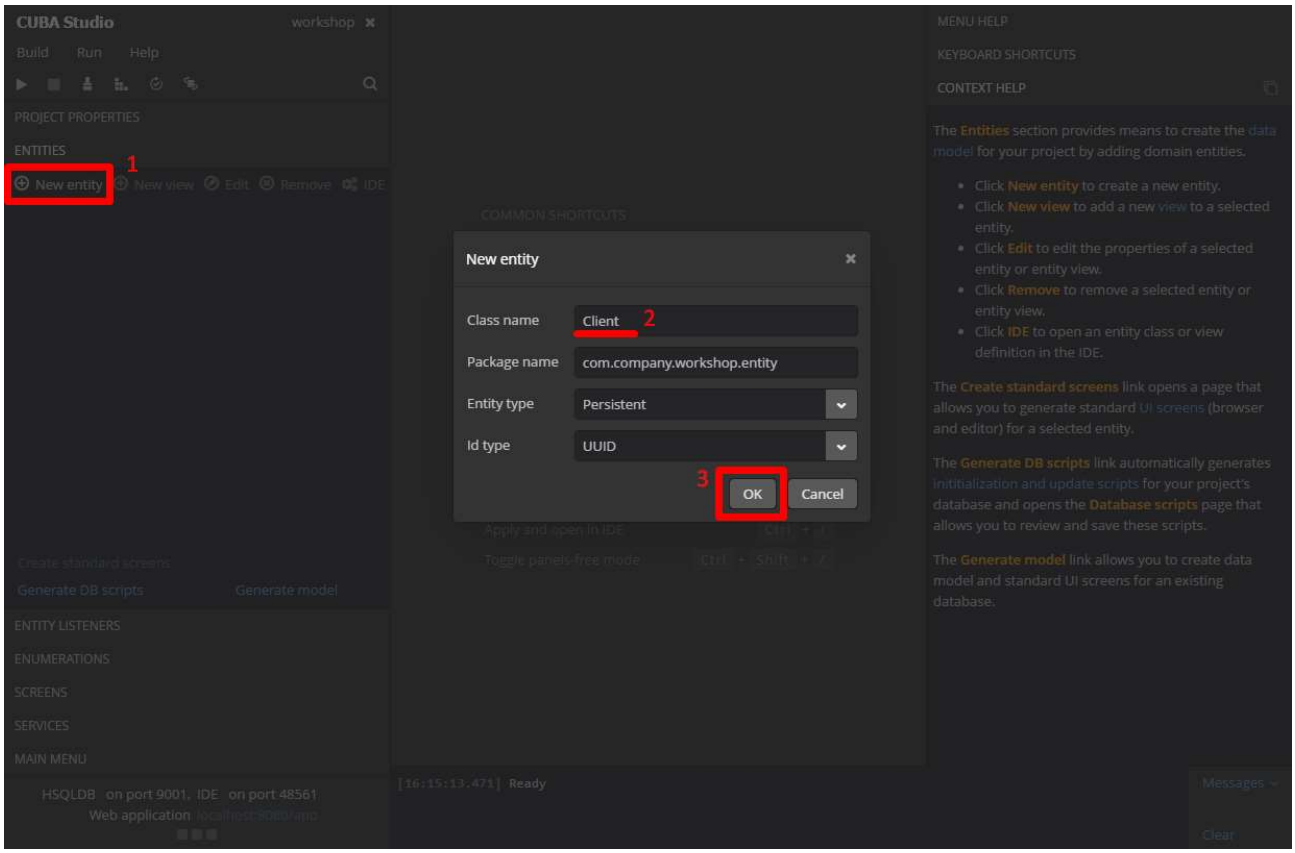


3.1 Client Entity

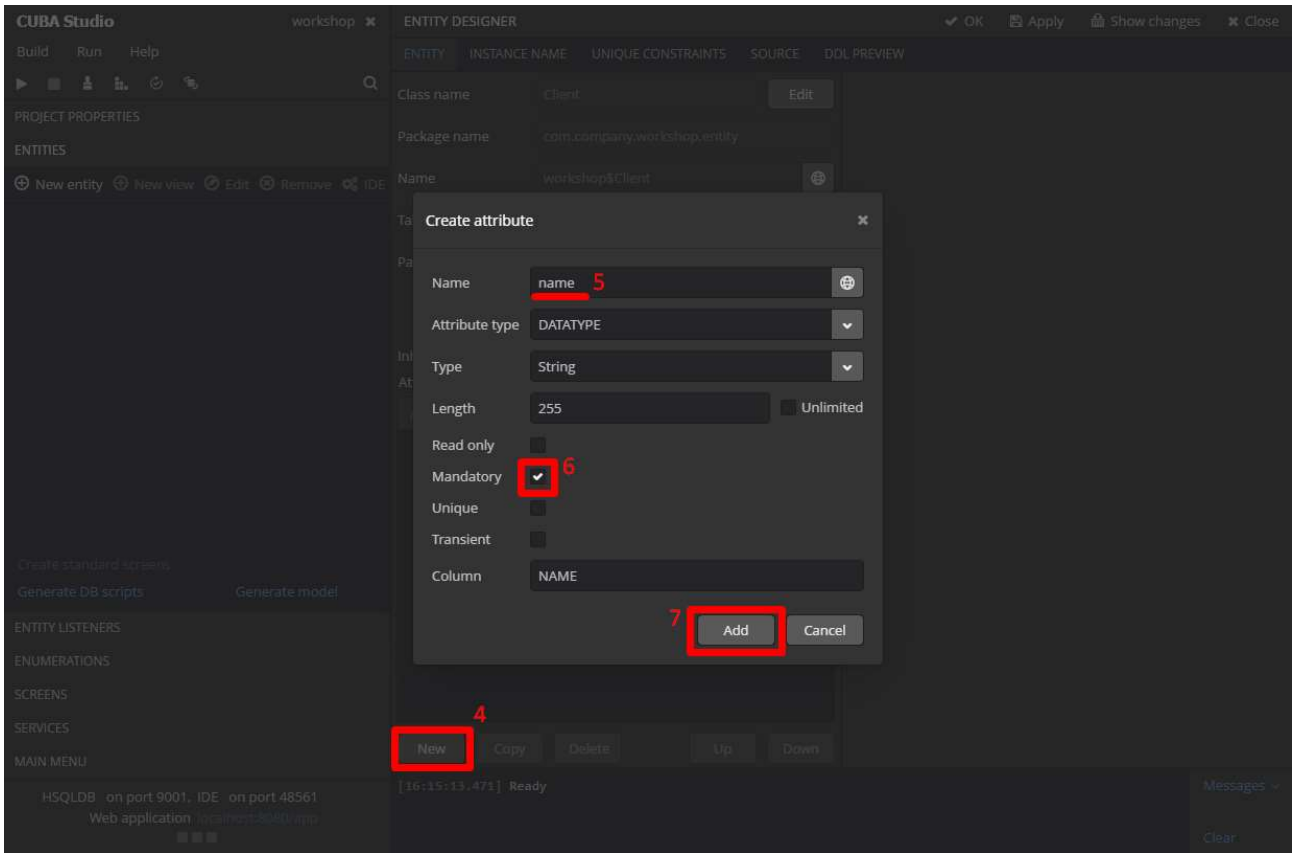
Let’s create the Client entity first.

1. Open the **Entities** section of the left hand side navigation panel and click **New entity**
2. Let’s create the Client entity first. Set **Class name**: ***Client***
3. Click **OK**

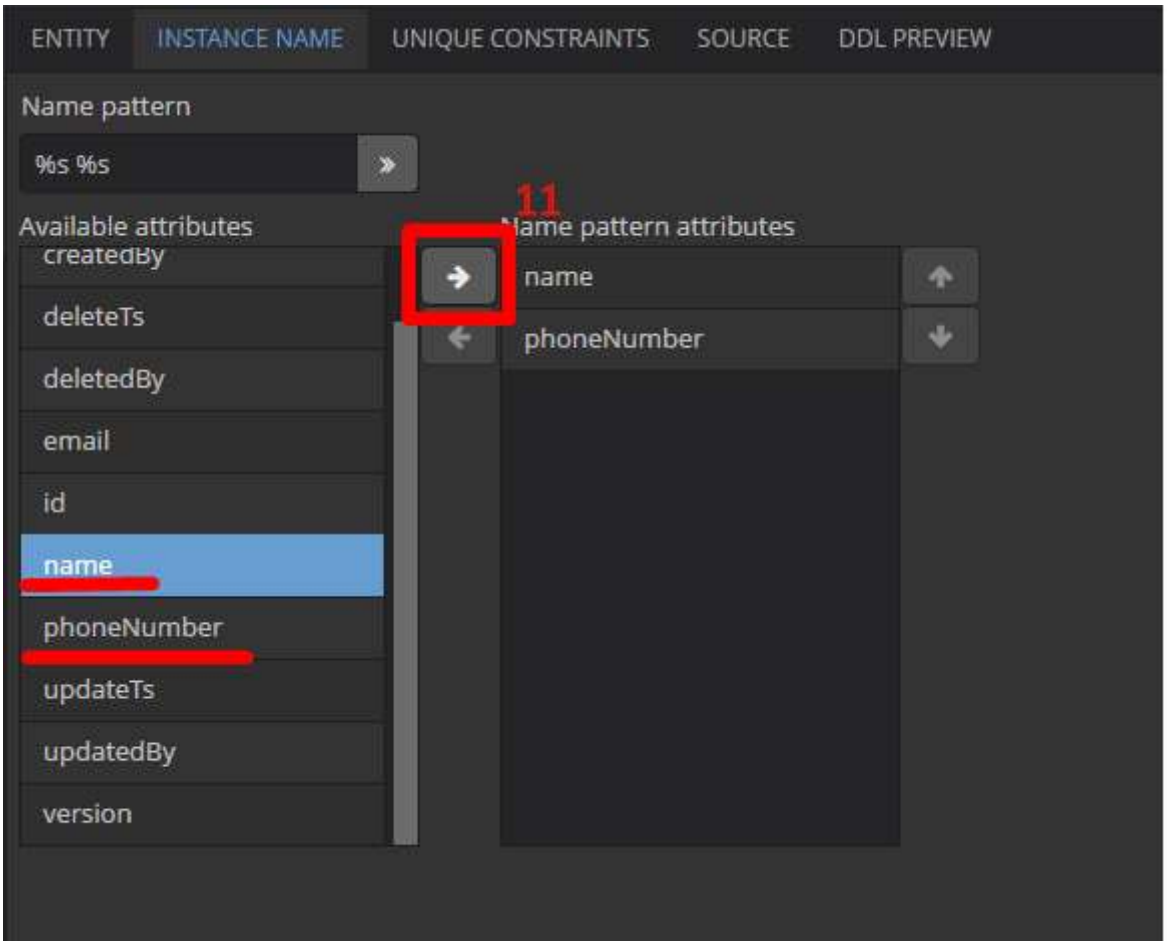
The entity editor has appeared, here we can add attributes, choose a parent class, specify corresponding table in the database and so on. Our class will be the inheritor of StandardEntity, the special base class, which already supports SoftDeletion and contains some platform internal attributes.



4. Click the **New** button under the **Attributes** table
5. Enter **Name:** ***name***
6. Name is a required string. Select the **Mandatory** checkbox.
7. Click **Add**



8. Similarly, let's add **phoneNumber** and **email**. Repeat steps 4-7 to create ***phoneNumber*** as a **mandatory string** attribute with the **length of 20** and flagged as **unique**
9. Repeat steps 4-7 to create ***email*** as a **mandatory string** attribute with the **length of 50** and flagged as **unique**
10. Now let's specify the display name of Client entity in dropdown lists and selection components. Go to the **Instance name** tab of the entity designer screen. On this tab you can specify the default string representation of an entity to be shown in tables, dropdowns, etc.
11. Select ***name*** and ***phoneNumber*** fields one by one



12. Let's see the source code files generated by Studio for our entity. Clicking the **Source** and **DDL Preview** tabs you can find your Entity bean implementation annotated with the *javax.persistence* annotations and SQL script for creating the corresponding table
13. We're done with Client, click **OK** in the right-top corner of the entity designer screen to save the entity and go to the Mechanic entity.

3.2 Mechanic entity

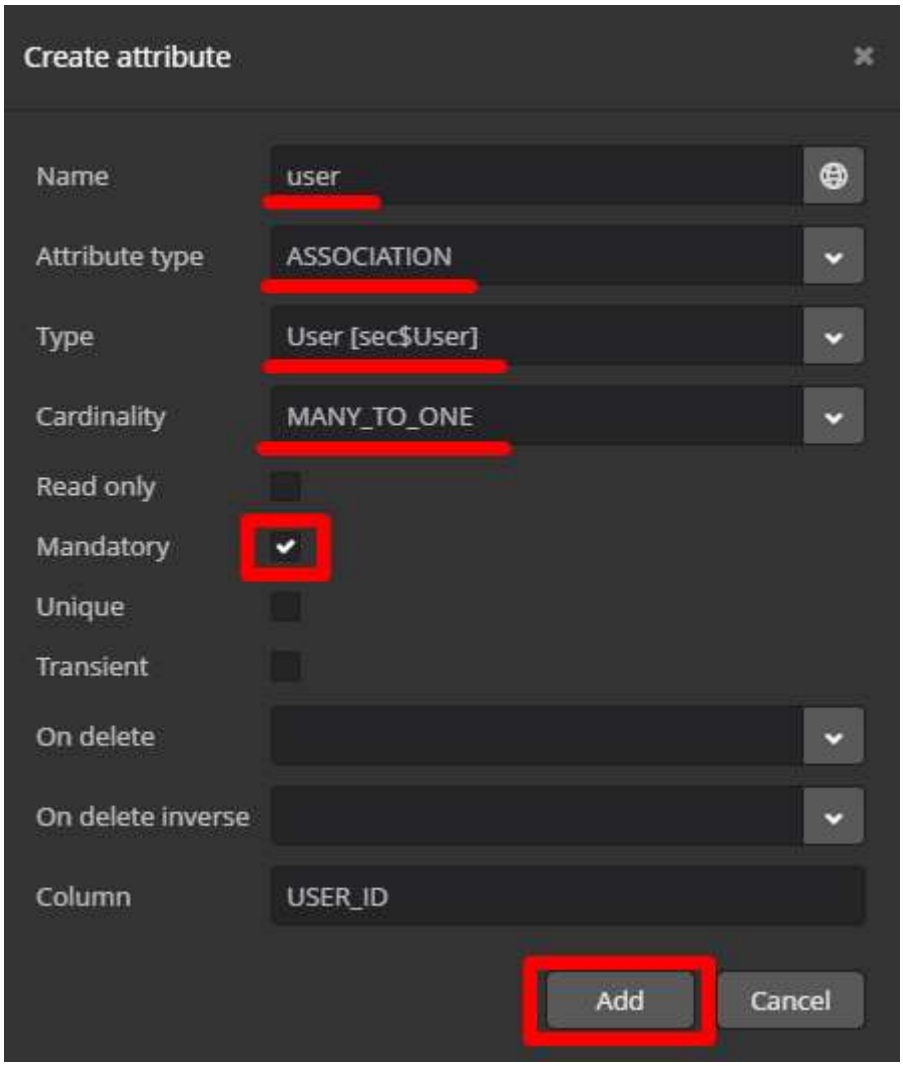
1. Create a **New entity** with **Class name**: ***Mechanic*** and click **OK**
2. Click the **New** button under the **Attributes** table.

Mechanic will be a system user, let's add the user attribute with a link to the User entity. The User entity is a standard entity storing system users in the CUBA Platform.

3. Fill up the **Create attribute** form, set the **Name: *user***.
4. To create an entity link, set the **Attribute type** to ***Association*** and choose the entity ***User [sec$User]*** in the **Type** field. Specify cardinality

MANY_TO_ONE for the association, tick the **mandatory** checkbox and click **Add.**

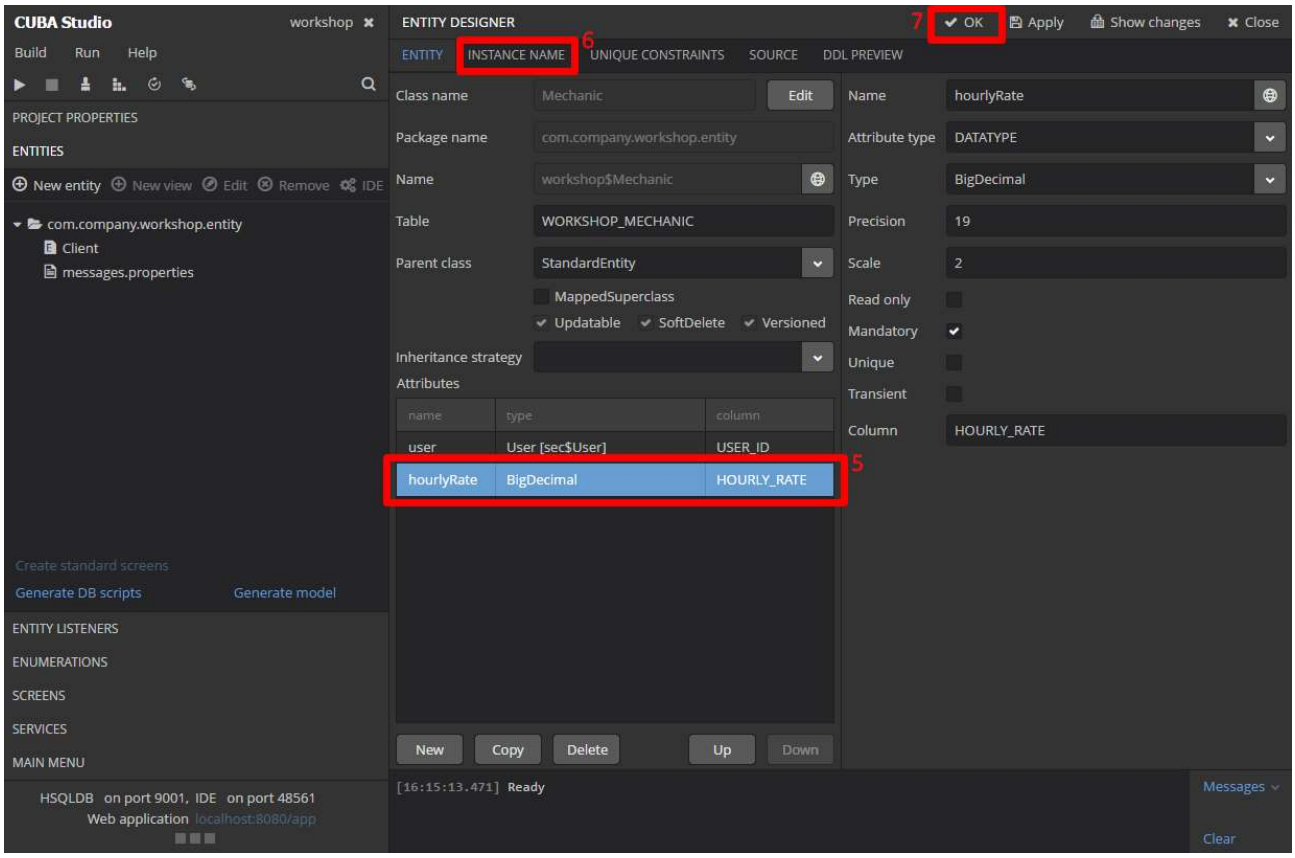
Name: ***user***
Attribute type: ***Association***
Type: ***User [sec$User]***
Cardinality: ***MANY_TO_ONE***
Mandatory: Yes



Studio has generated the name of the association column in the DB automatically, and we can change it if needed.

5. Create one more attribute. For mechanics we'll need one more mandatory attribute **Hourly Rate**.
6. Apply the following settings and click **Add**:
   Name: ***hourlyRate***
   Type: ***BigDecimal***
   Mandatory: Yes

7. Move to the **Instance name** tab and select ***user*** as an instance name for Mechanic
8. Check that your *Mechanic* entity corresponds to the the picture below and click **OK**

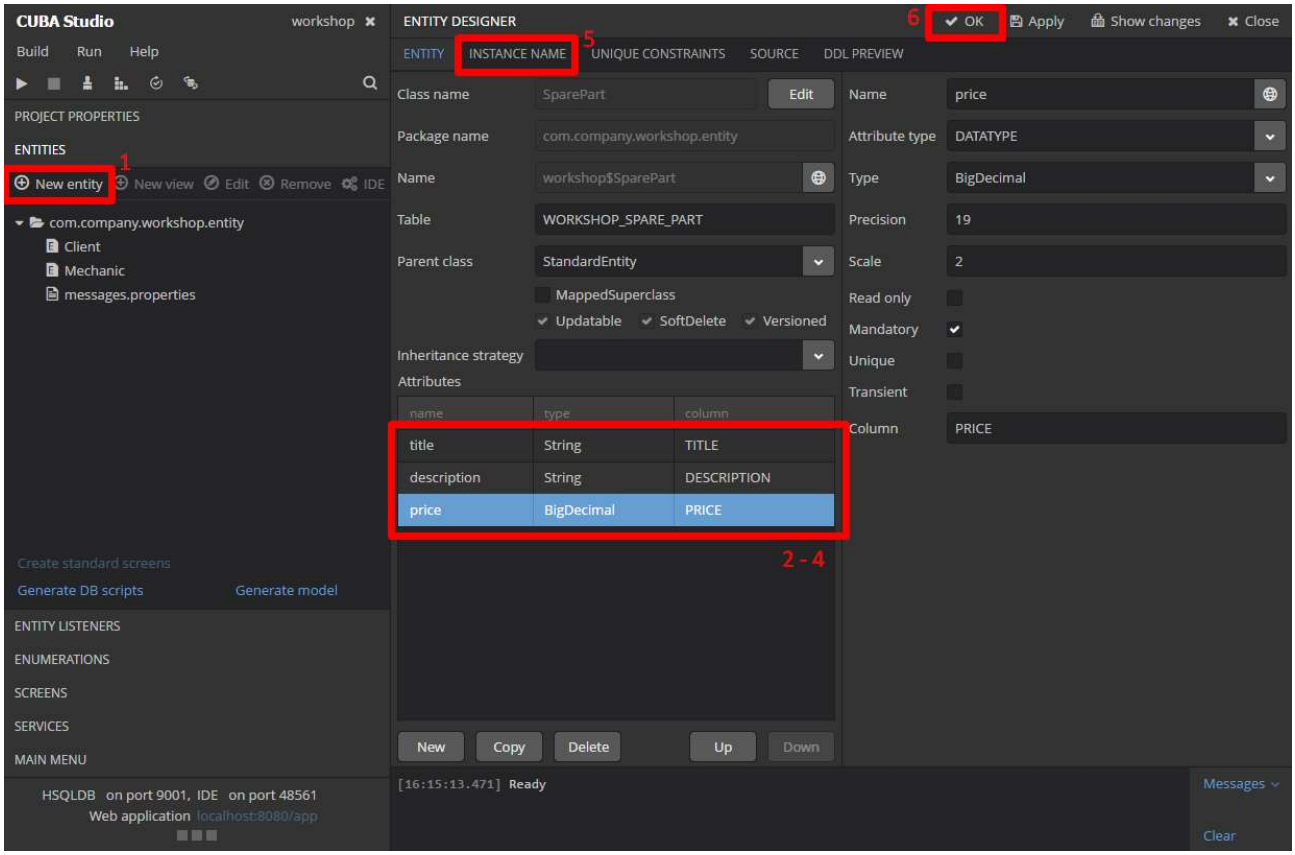


3.3 SparePart Entity

Let's create the SparePart entity. It will have the following attributes: Title, Description, and Price.

1. Create a **New entity** with **Class name: *SparePart***
2. Add an attribute Title which is a simple string, required, unique attribute.
   Name: *title*
   Type: *String*
   Mandatory: Yes
   Unique: Yes
3. Add an attribute Description that is an attribute with the unlimited length text. In order to specify the unlimited length column, tick **Unlimited** checkbox.
   Name: *description*
   Type: *String*
   Length: Unlimited

4. Add an attribute Price that has BigDecimal type and is mandatory.
   Name: ***price***
   Type: ***BigDecimal***
   Mandatory: Yes
5. Go to the **Instance name** tab and select the **title** attribute as the entity instance name
6. Check that that your *SparePart* entity corresponds to the the picture below and click **OK**

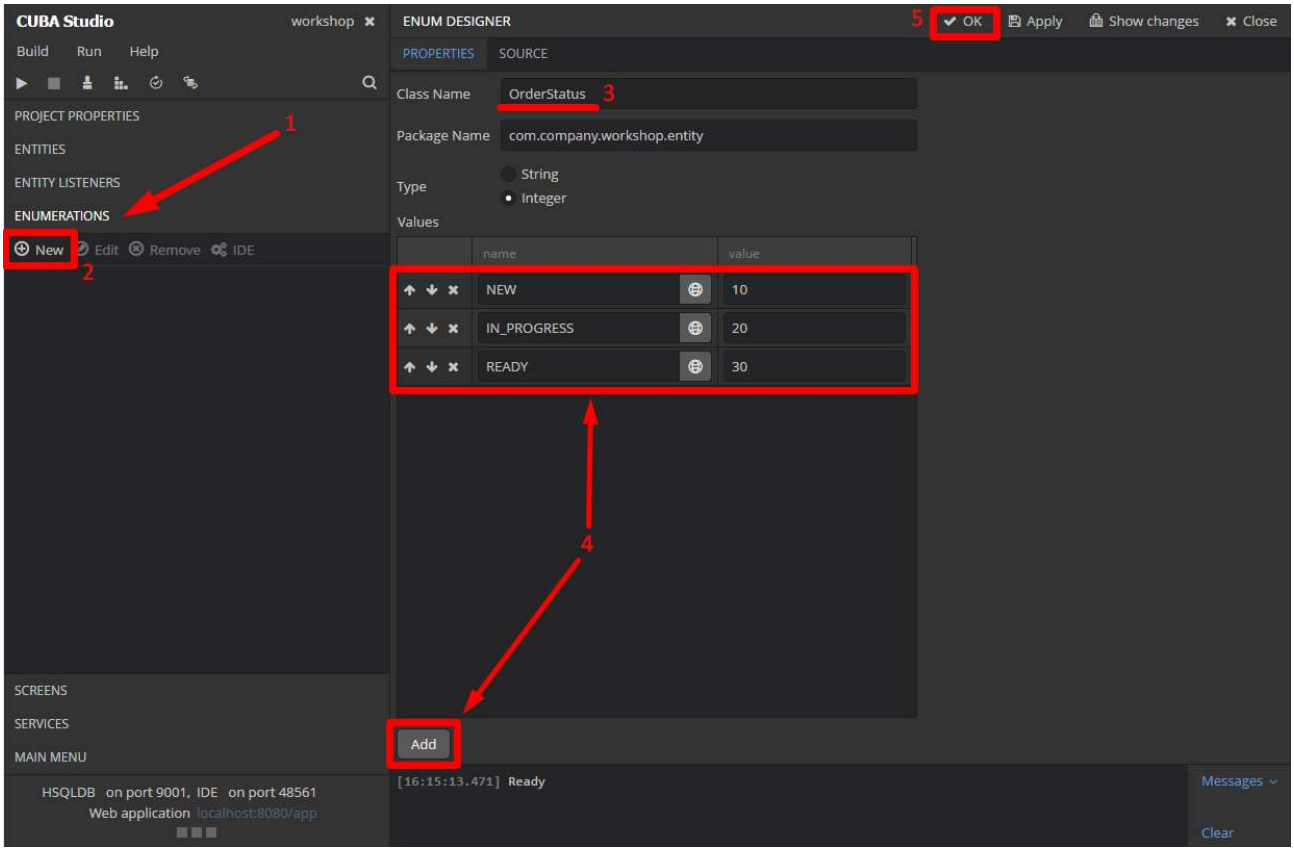


3.4 OrderStatus Enum

For the Order entity, we'll need to create the OrderStatus enum first. The status will be stored in the DB as numeric values.

1. Go to the **Enumerations** section in the left hand side navigation panel
2. Click **New**
3. Enter **Class name**: ***OrderStatus***
4. Add values in the uppercase according to the Java naming convention:
   NEW 10
   IN_PROGRESS 20
   READY 30

5. Check that your *OrderStatus* enum corresponds to the the picture below and click **OK**. Similar to entities, we can check the generated Java code in the Source tab.



3.5 Order Entity

The last missing entity of our data model is Order.

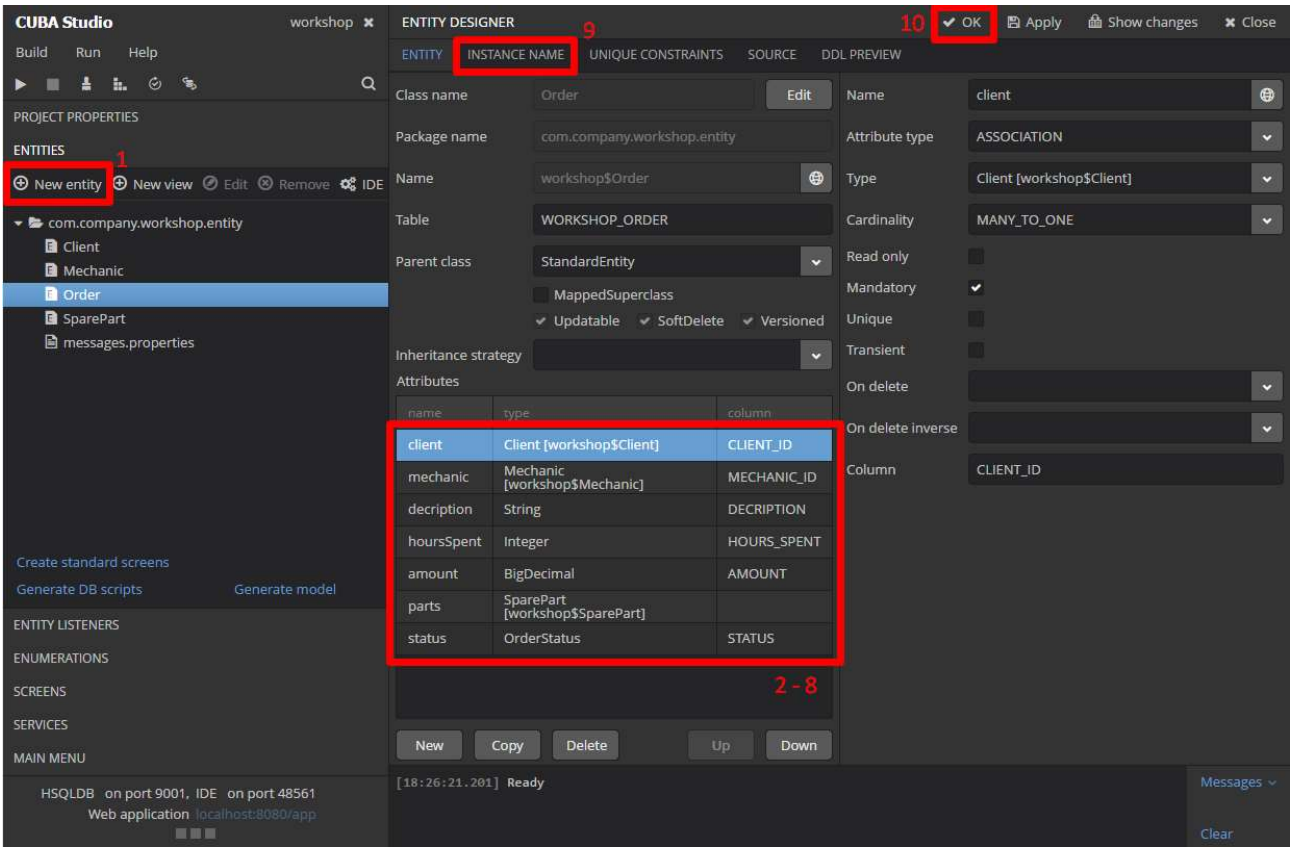
1. Create a **New entity** with **Class name: *Order***
2. First let's add a link to the client. Add an attribute which is a relation to the Client entity of the Association type, with many-to-one cardinality.
   Name: *client*
   Attribute type: *ASSOCIATION*
   Type: *Client*
   Cardinality: MANY_TO_ONE
   Mandatory: Yes
3. Then add a link to the mechanic that performs the order. Add an attribute:
   Name: *mechanic*
   Attribute type: *ASSOCIATION*
   Type: *Mechanic*

Cardinality: MANY_TO_ONE
Mandatory: Yes

4. Now let's add the order description, this will be an unlimited length string. Add an attribute:
   Name: *description*
   Type: *String*
   Length: Unlimited
5. Similarly, let's add the hoursSpent attribute
   Name: *hoursSpent*
   Type: *Integer*
6. And the amount attribute
   Name: *amount*
   Type: *BigDecimal*
7. Now we need to link our order with the SparePart list. Add an attribute
   Name: *parts*
   Attribute type: *ASSOCIATION*
   Type:SparePart
   Cardinality: MANY_TO_MANY
   The Studio will offer to create a inverse attribute - click **No**
8. The last attribute will be status. Add the attribute, in the creation dialog, specify the ENUM type and set mandatory.

   Name: *status*
   Attribute type: *Enum*
   Type: *OrderStatus*

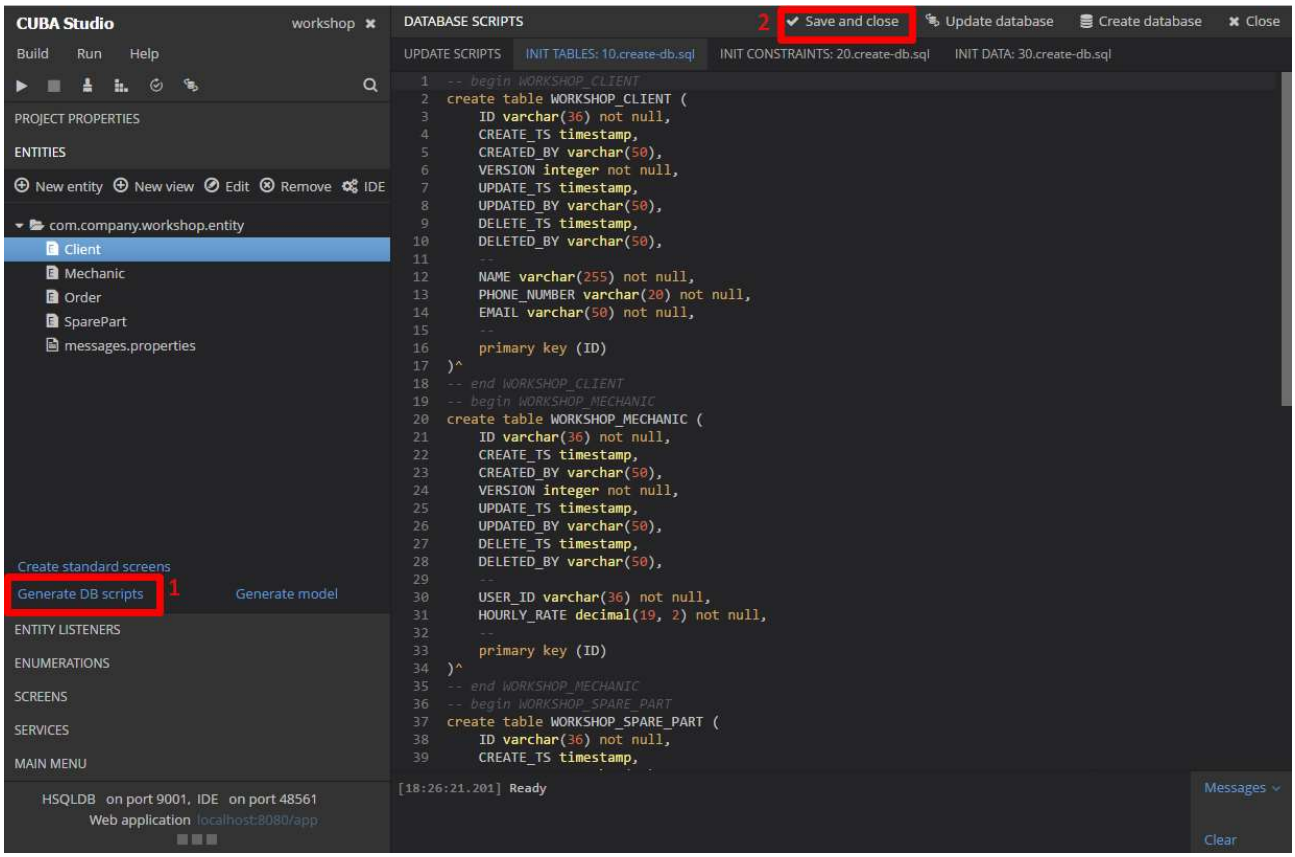
9. Set the **Instance name** for the *Order* entity to its **description** attribute
10. Check that that your *Order* entity corresponds to the the picture below and click **OK**



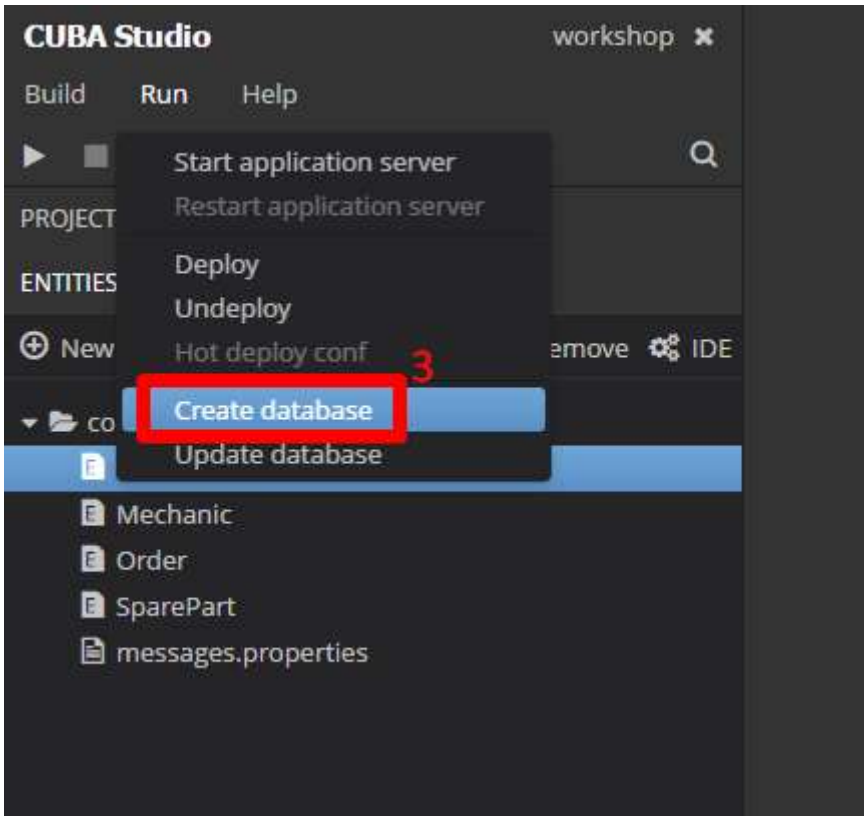
3.6 Generate DB Scripts and Create the Database

As we're done with the data model, let's generate DB update scripts and create a database.

1. Click the **Generate DB scripts** link In the bottom of the **Entities** section; the CUBA Studio will generate a script to create tables and constraints
2. Click **Save and close** and the Studio will save the scripts into a special directory of our project, so we will be able to access them if needed



3. Invoke the **Run — Create database** action from the menu to create a database



4. The CUBA Studio warns us that the old DB will be deleted, click **OK**

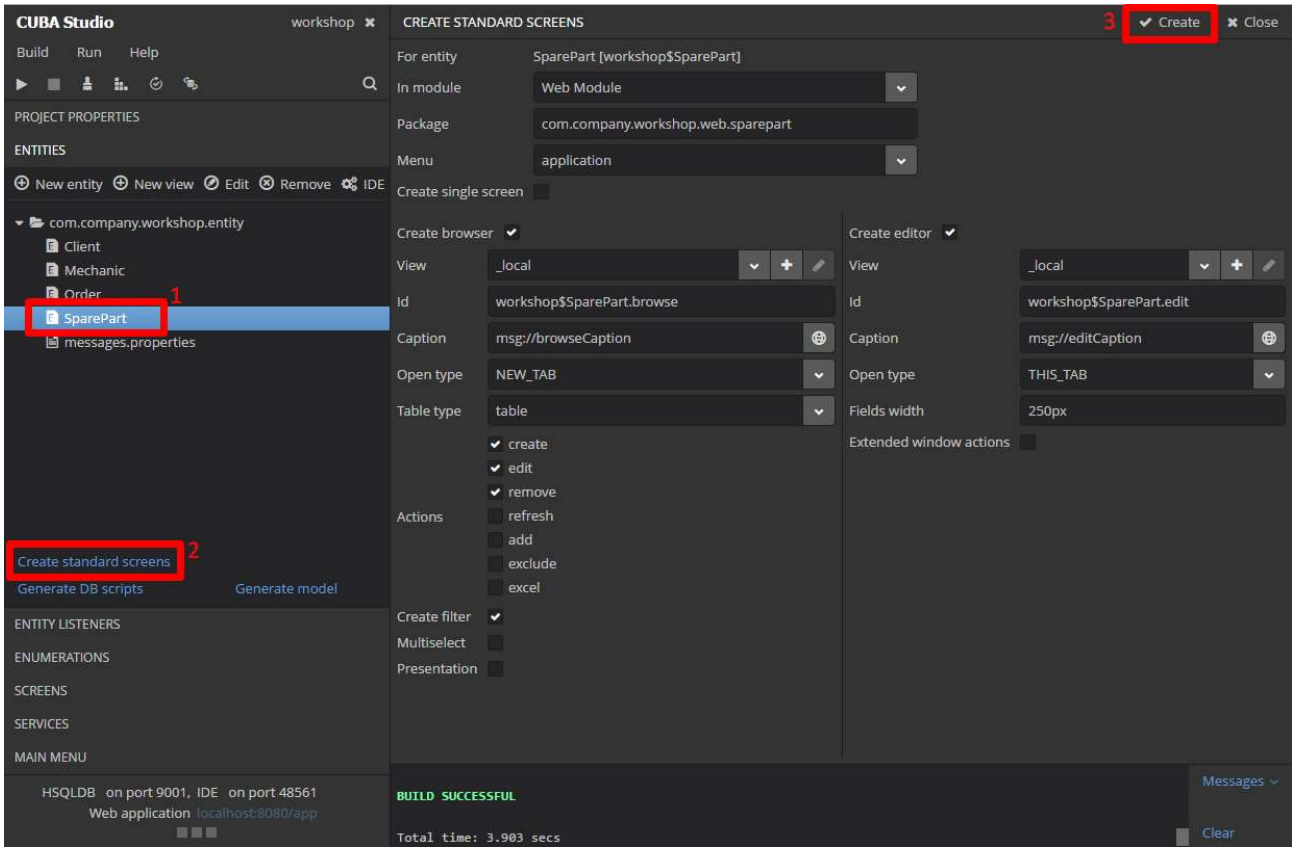
4 AUTO-GENERATED CRUD UI

Under CRUD or standard UI we understand screens that allow you to browse the list of records (browser) and create/edit one record (editor).

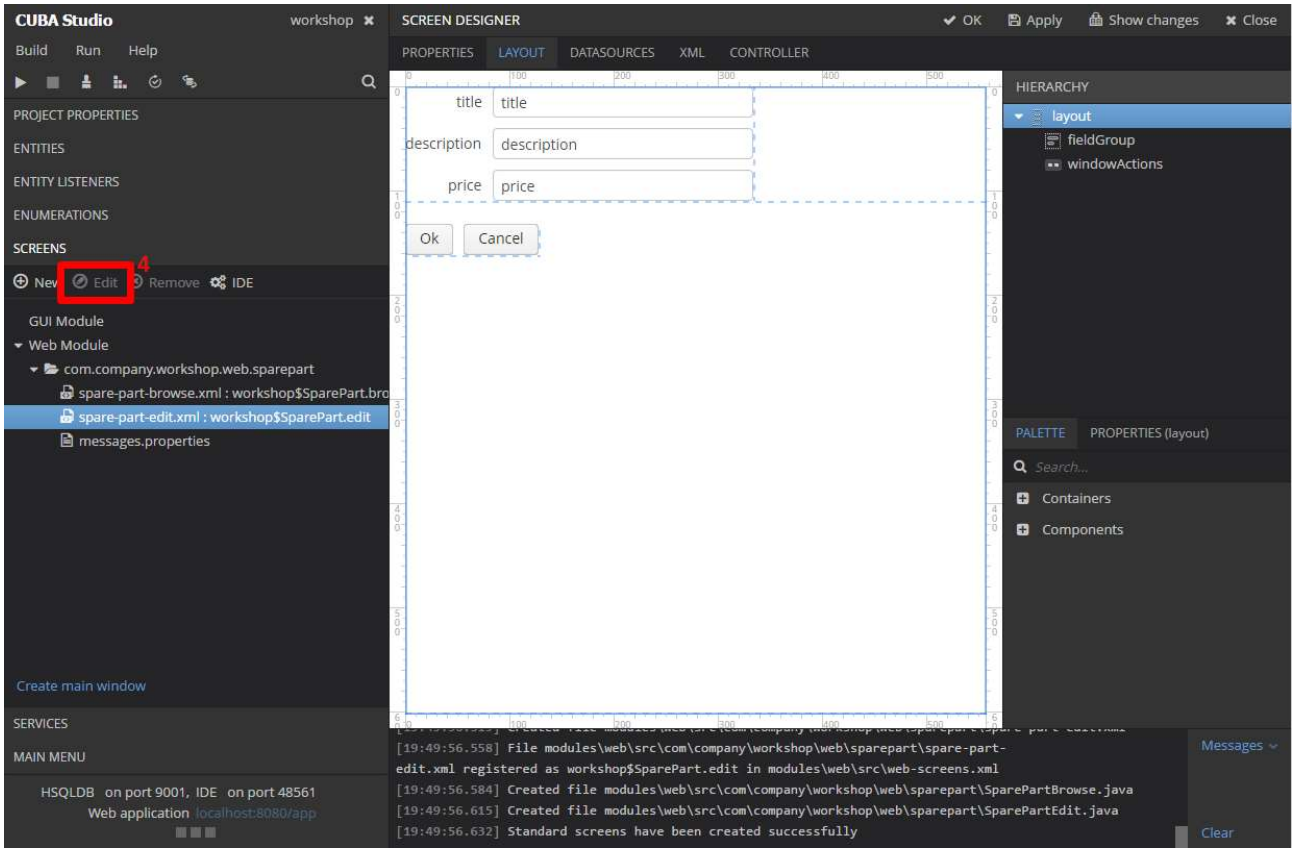
4.1 SparePart CRUD UI

Now let's create standard browser and editor screens for the SparePart entity.

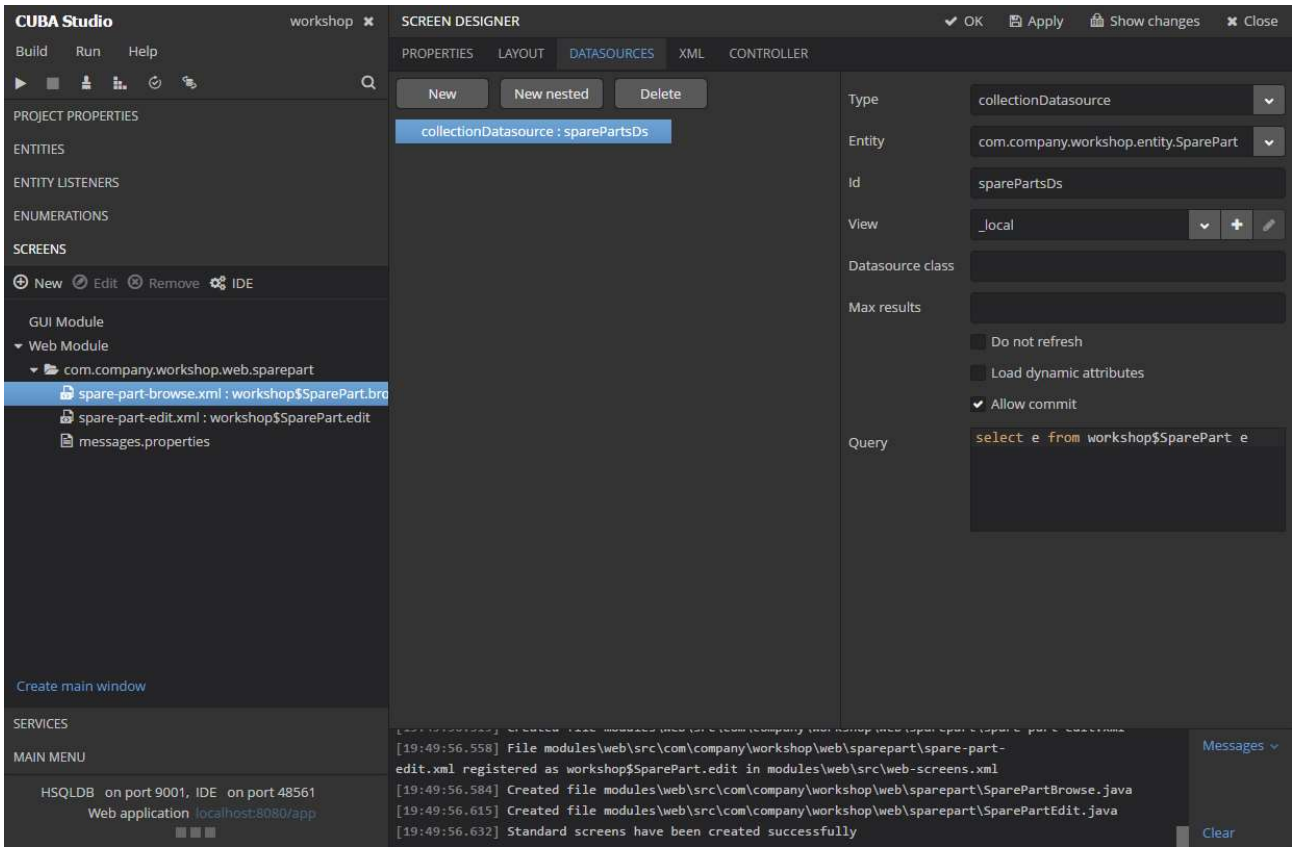
1. Open the **Entities** section of the left hand side navigation panel and select the **SparePart** entity
2. Click on the **Create standard screens** link.
   Create standard screens dialog appears. It allows you to specify where to place the screens in the project, what menu item to be used to access the browser screen, should it be a single combined screen for all operations or two separate screens to read the list and create, update and delete an instance
3. Keep the default values and click **Create**



4. The studio has generated separate screens: browser and editor. You can pick one of them and click **Edit** in the **Screens** section of the navigation panel. Screen designer has WYSIWYG editor, XML and Controller tab to be able to see screen declaration and its Java controller.



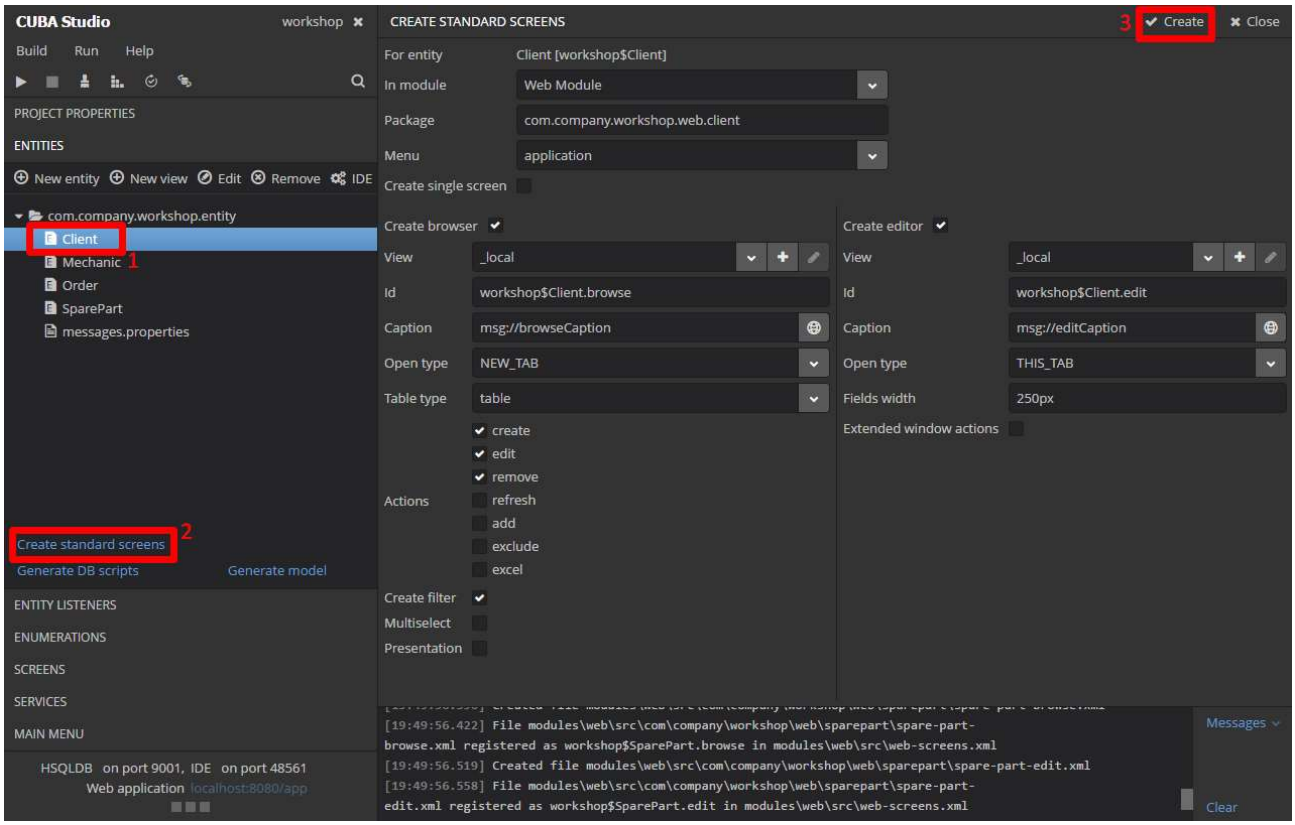
5. Visual components that tightly work with data (tables, dropdowns, etc) are data-aware and connect to the database through datasources. Taking SparePart browser as an example, let's have a look at data binding. Select the **sparepart-browse.xml** screen, click **Edit** and open the **Datasources** tab. Datasources use standard JPQL queries to load data. In our example the result list of this query will be automatically shown in the main table of the screen, as ***sparePartsDs*** is set to the datasoure property of the table.



4.2 Client CRUD UI

Then let’s generate the screens for the Client entity.

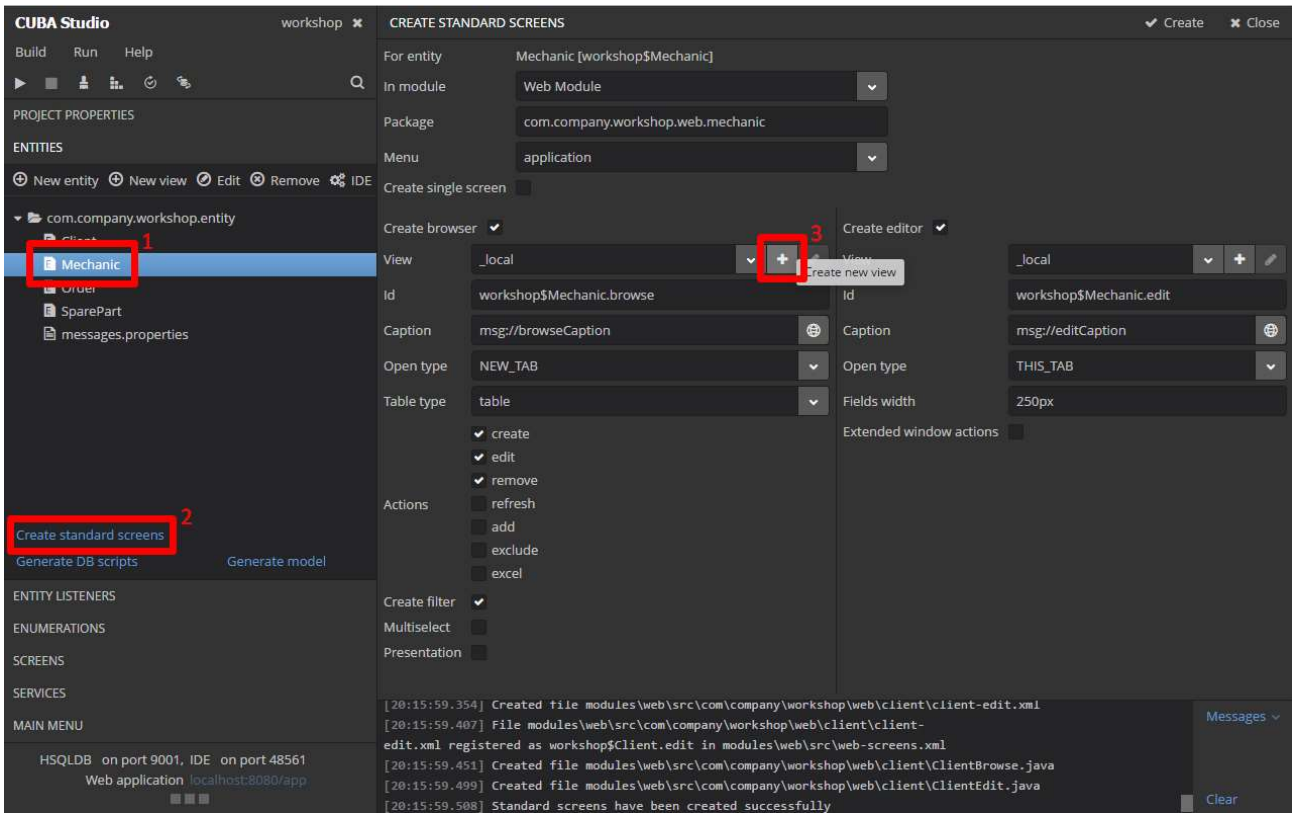
1. Open the **Entities** section of the navigation panel and select the **Client** entity
2. Click **Create standard screens**
3. Click **Create**



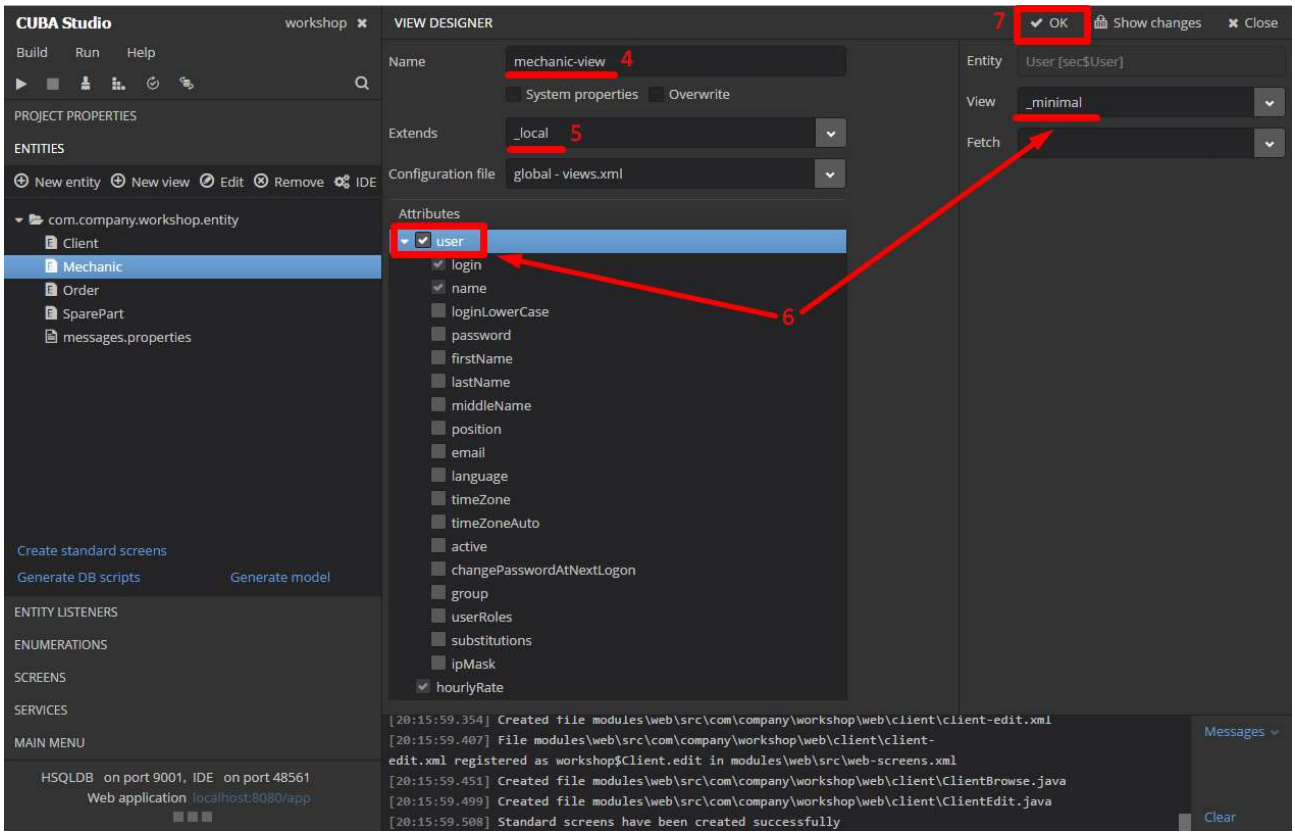
4.3 Mechanic CRUD UI

Similarly, let's generate the screens for Mechanic.

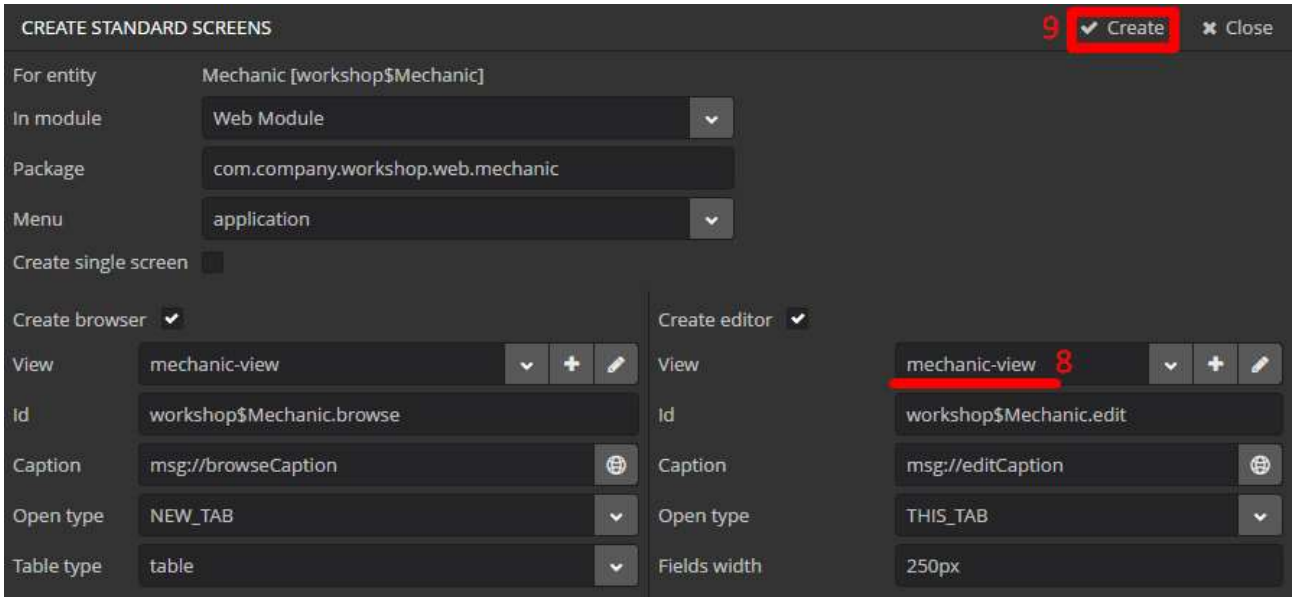
1. Open the **Entities** section of the navigation panel and select the **Mechanic** entity
2. Click **Create standard screens**
3. Mechanic differs from the entities we have just created UI for. It has relation to the system **User** entity, which is used in CUBA for security reasons. So, our datasource should load the **user** field along with the simple fields of the mechanic entity. For that purpose CUBA uses views - an XML description of what entity attributes should be loaded from the database.
   Click the **plus** button to create a new view and the **View designer screen** will appear



4. Specify the **Name**: ***mechanic-view***, as we will use it for both browser and editor
5. Our view extends the ***_local*** one, which is the system view that includes all the simple (local) attributes.
   By default all screens use *_local* view for standard screen generation. It means that references to other entities will not be loaded and shown by default.
6. Select **user** attribute and specify the ***_minimal*** system view for it; _minimal view includes only attributes, that are specified in the **Instance Name** of an entity. This view will give us enough information to show an entity in the browser's table and editor's field group
7. Click **OK** to save the view



8. Specify the same view for the editor
9. Click **Create**

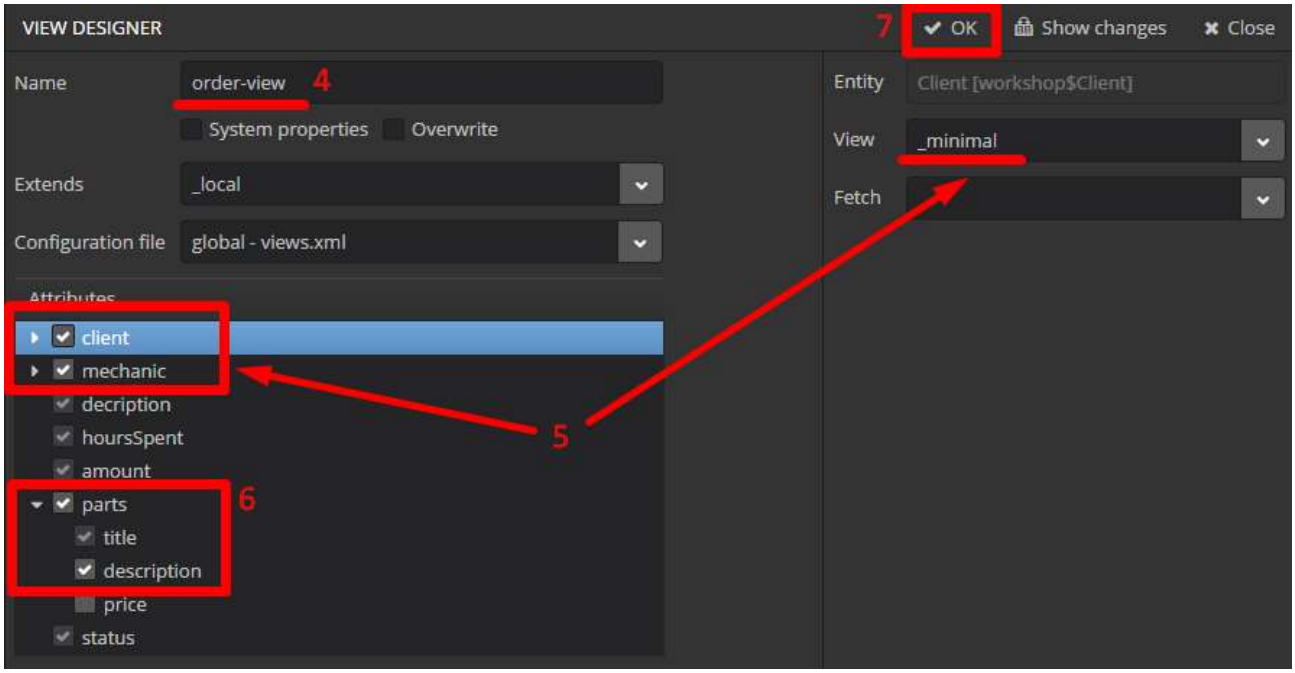


4.4 Order CRUD UI

Now we need to create screens for the Order entity

1. Open the **Entities** section of the navigation panel and select the **Order** entity
2. Click **Create standard screens**
3. Similarly to what we did for the Mechanic entity, click the plus button to create an extended view for the **Order** entity

4. Rename the view to ***order-view***
5. Select **client** and **mechanic** attributes and specify the ***_minimal*** view for both of them
6. Select the **parts** attribute, which contains a collection of the SpareParts instances. Adding parts to an order we would like to see only **title** and **description** attributes, so tick them
7. Click **OK**



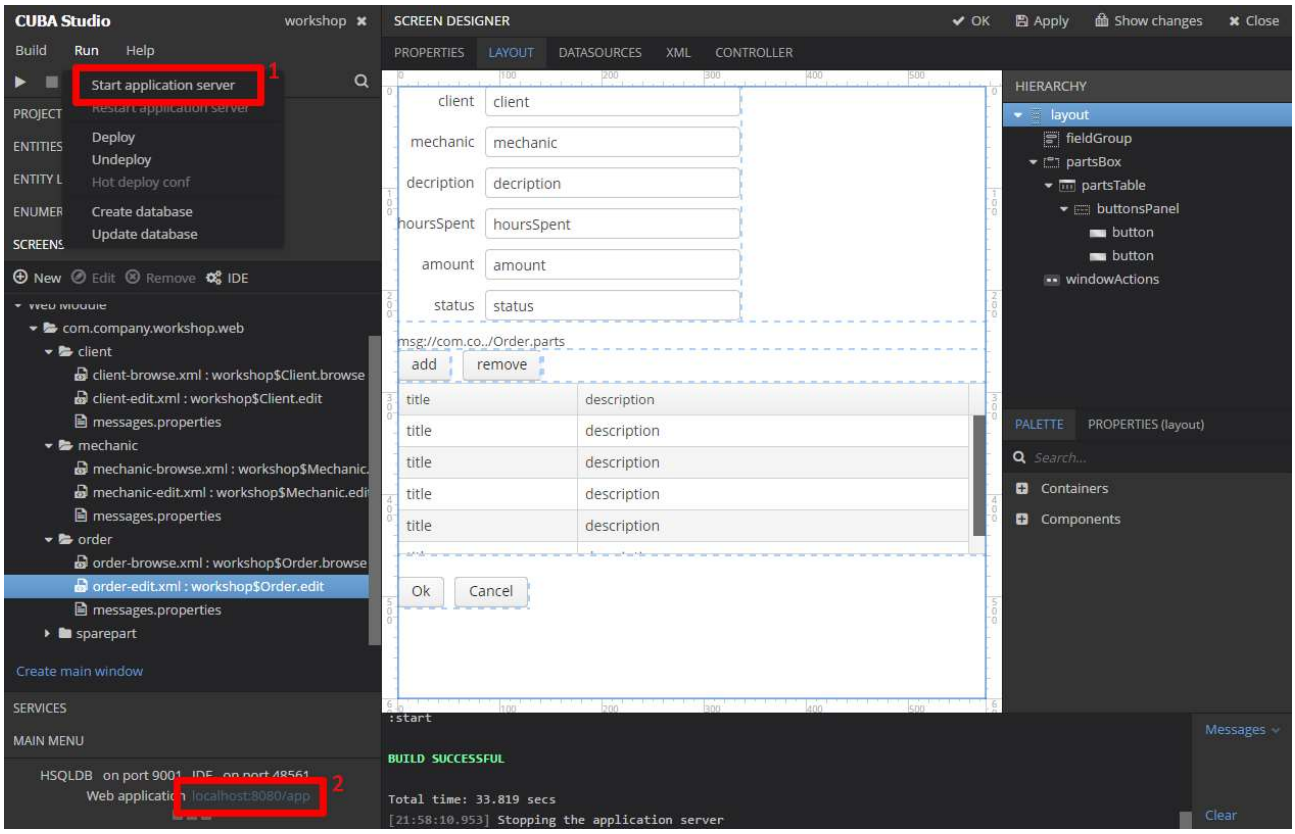
8. Select ***order-view*** also for the editor screen
9. Click **Create**

5 FIRST LAUNCH

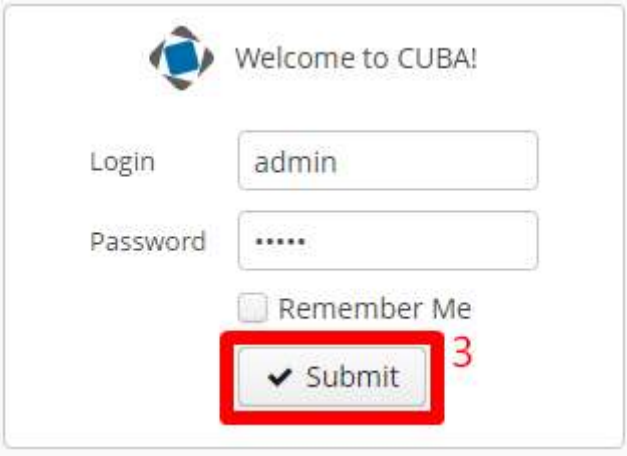
Our application is done, of course, to a first approximation.

Let’s compile and launch it!

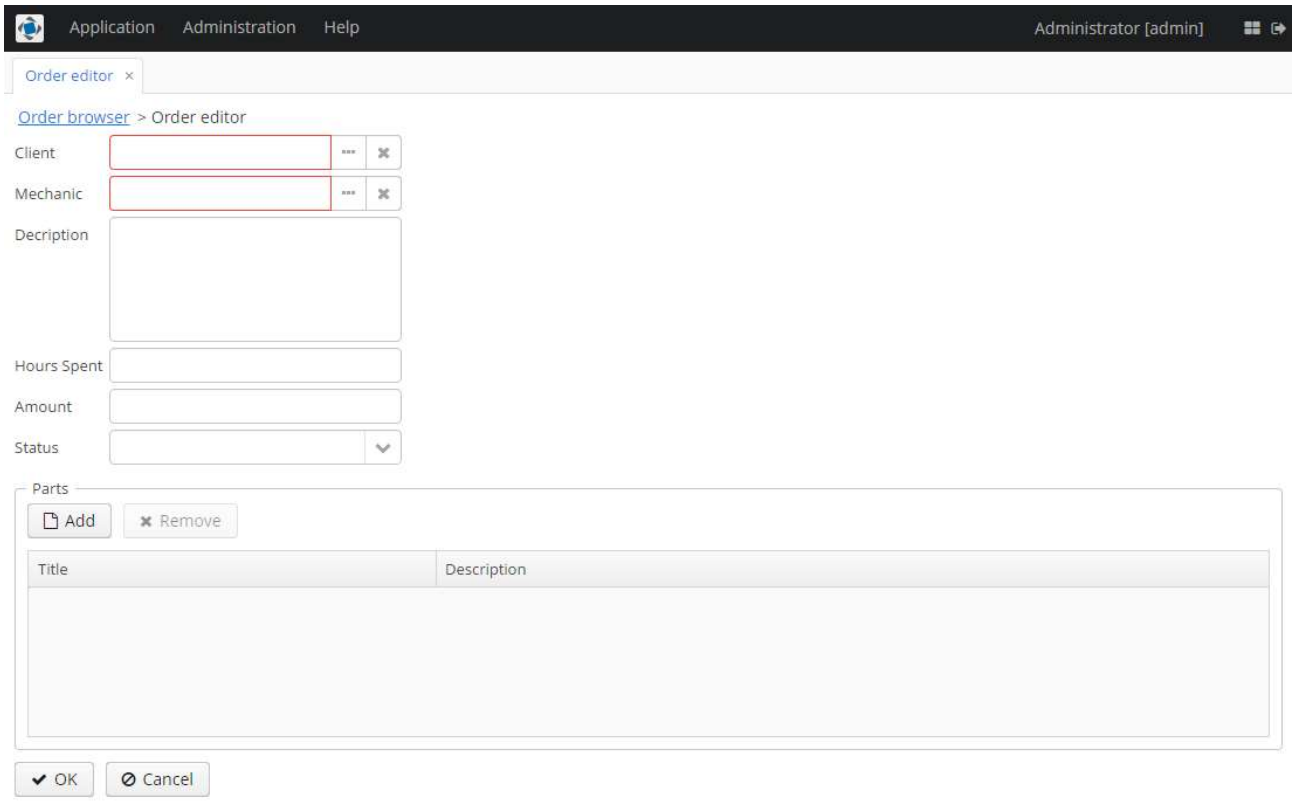
1. Invoke the **Run — Start application** action from the menu. Studio will deploy a local Tomcat instance in the project subdirectory, deploy the compiled application there and launch it
2. Open the application by clicking a link in the bottom part of the Studio



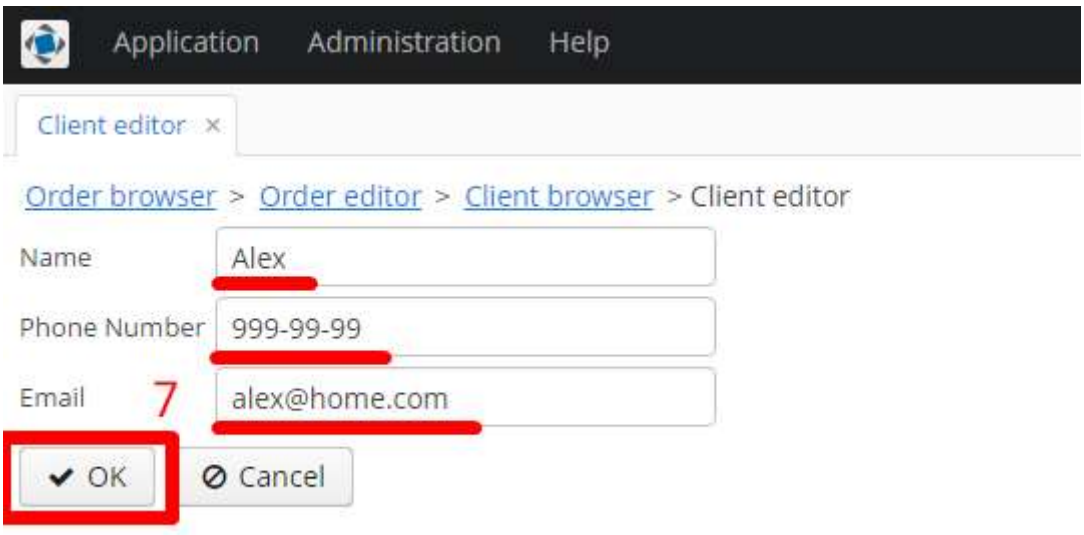
3. CUBA application welcomes us with the login screen. It's a part of the security module, integrated in the platform. Default login and password are already set in the screen, so just click **Submit**.



4. After successful login you can see the main window of your application, which can be customized from the Studio, similarly to any other screen.
5. Open **Application - Orders** from the menu. The standard browser screen appears. It contains a generic data filter on top, a table of records with buttons on top, performing standard CRUD actions. There are more standard actions available in the platform, for example export to excel. Click **Create**.
6. This is the autogenerated editor screen for the Order entity. Let's fill up the form. Click **[…]** to select a client.

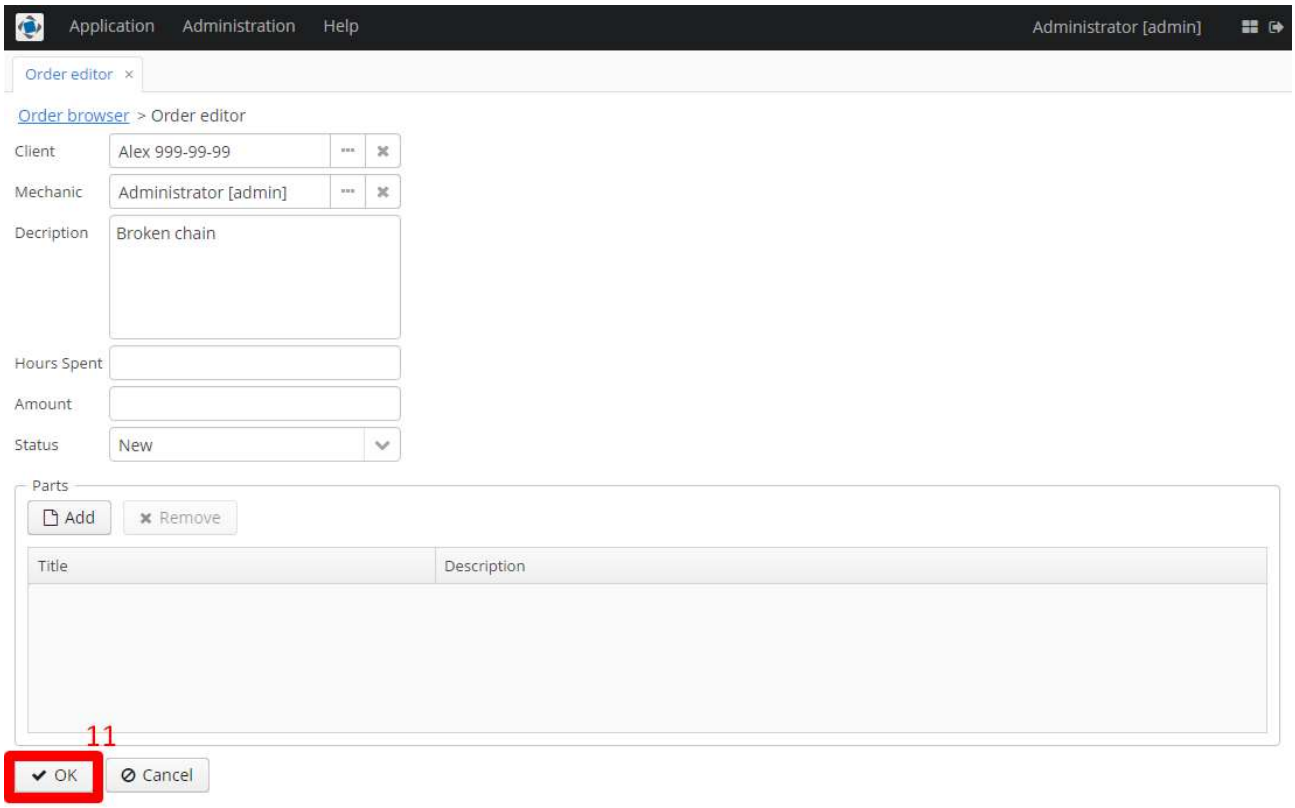


7. There are no clients in the system yet, so the client browser is empty. Click **Create** to add a new client to the system and fill up client editor with the following values and click **OK**:
   Name: ***Alex***
   Phone Number: 999-99-99
   Email: *alex@home.com*



8. Double click the client record we have just created or simply click **Select** on the bottom part of the client browser

9. We have specified a client for our order and it is shown in the client field accordingly to the **Instance name**, we specified for the Client entity (Name Phone number)
10. Create set the Mechanic field in the same way
    User: ***admin***
    Hourly Rate: ***10***
11. Complete entering data for the order using the following values and click **OK**
    Description: ***Broken chain***
    Status: ***New***



Our CRUD application is ready, so we have successfully completed items 1-3 of the functional specification!

5.1 Generic Filter

Now let's examine how data filter works. The CUBA generic filter enables you to filter data by all the fields of the screen main entity, all the entities it refers to as well as their fields. It also gives you ability to create custom filtering rules based on JPQL queries and to save them for repeated use.

Create one more order with the following parameters:

Client

Name: ***Freeman***

Phone Number: ***111-11-11***
Email: ***freeman@world.com***

Mechanic
User: ***admin***
Hourly Rate: ***8***

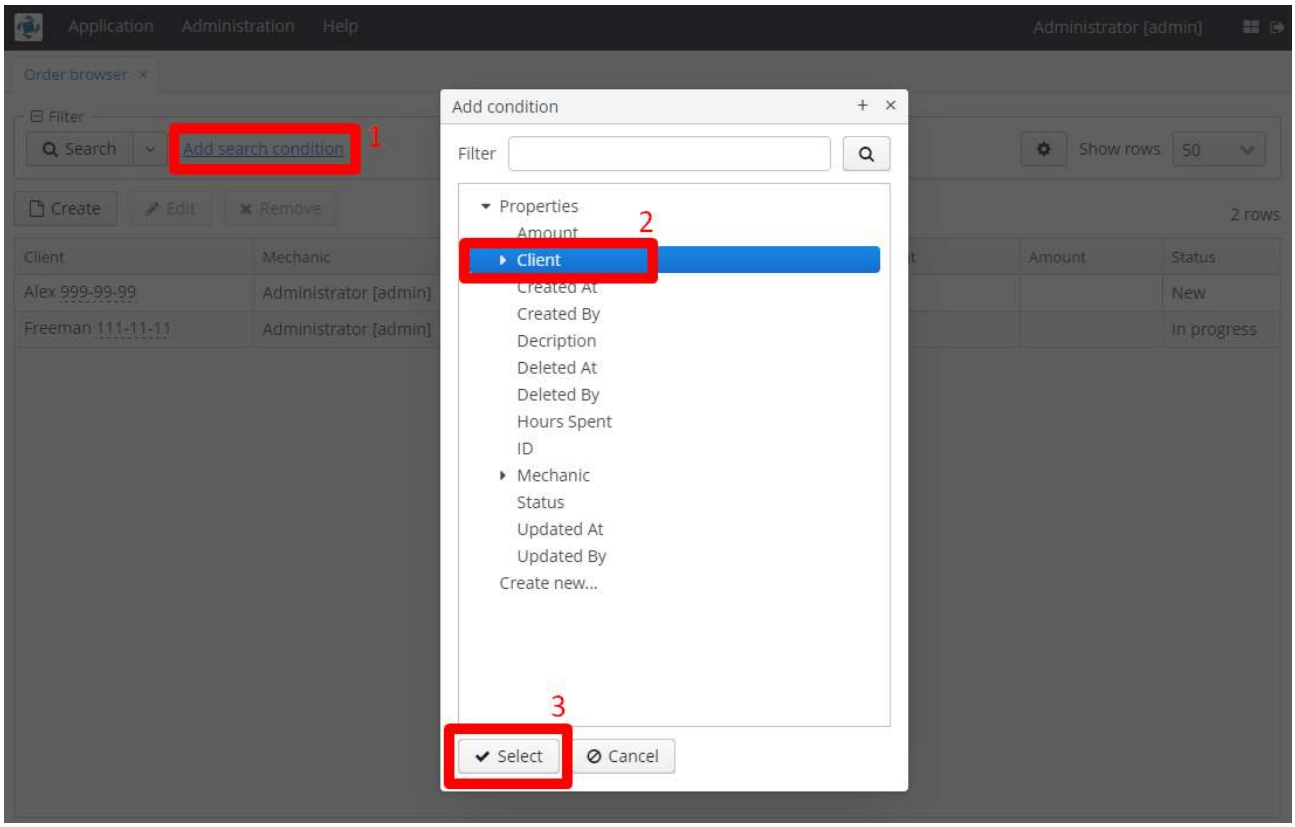
Order
Description: ***Wheels problem***
Hours Spent: ***2***
Status: ***In progress***

Parts (add one part)
Title: Wheel 29”
Description: Standard wheel
Price: 50

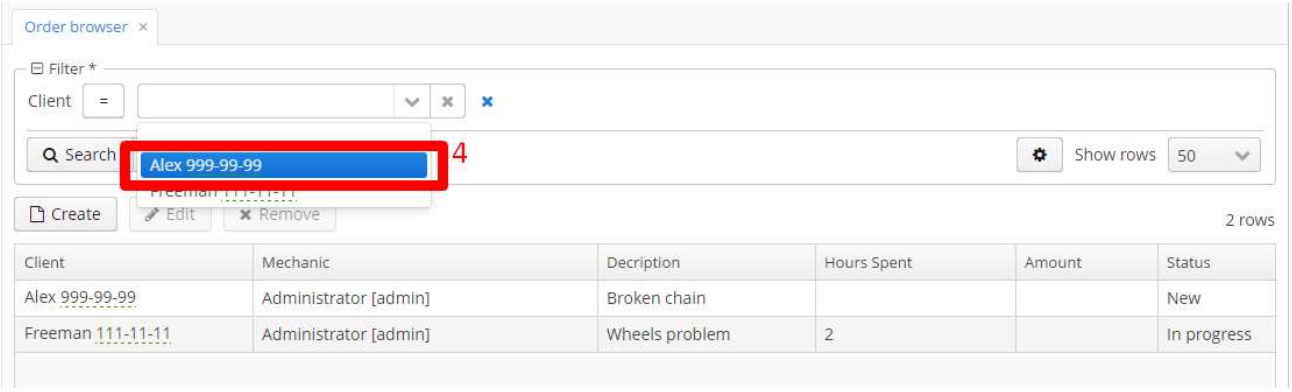
5.1.1 Client’s Orders Filter

Let’s create a simple filter to find all the orders of a particular client.

1. Go to the Order Browser screen (**Application — Orders**) and click the **Add search condition** link
2. Select the **Client** property
3. Click **Select**



4. Your filter automatically shows you options that can be applied for the **Client** field. Select ***Alex*** and click **Search**.



5. Only one record left after filtering.
6. Click on the **[=]** button to see what comparison options CUBA filters offer for users

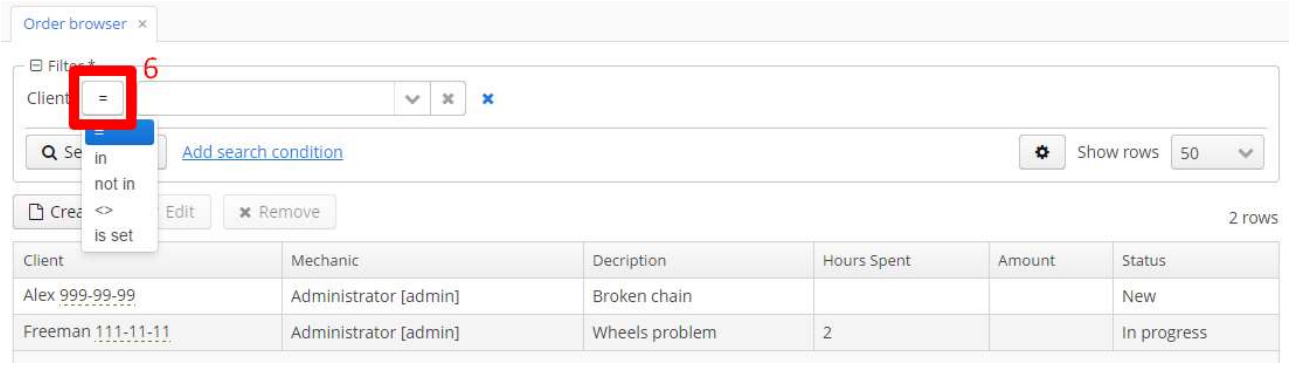
   [=] - exact equality

   [in] - the field is contained in a specified set of values

   [not in] - the field is NOT contained in a specified set of values

   [<>] - the field is not equal to the specified value

   [is set] - the field is null/not null

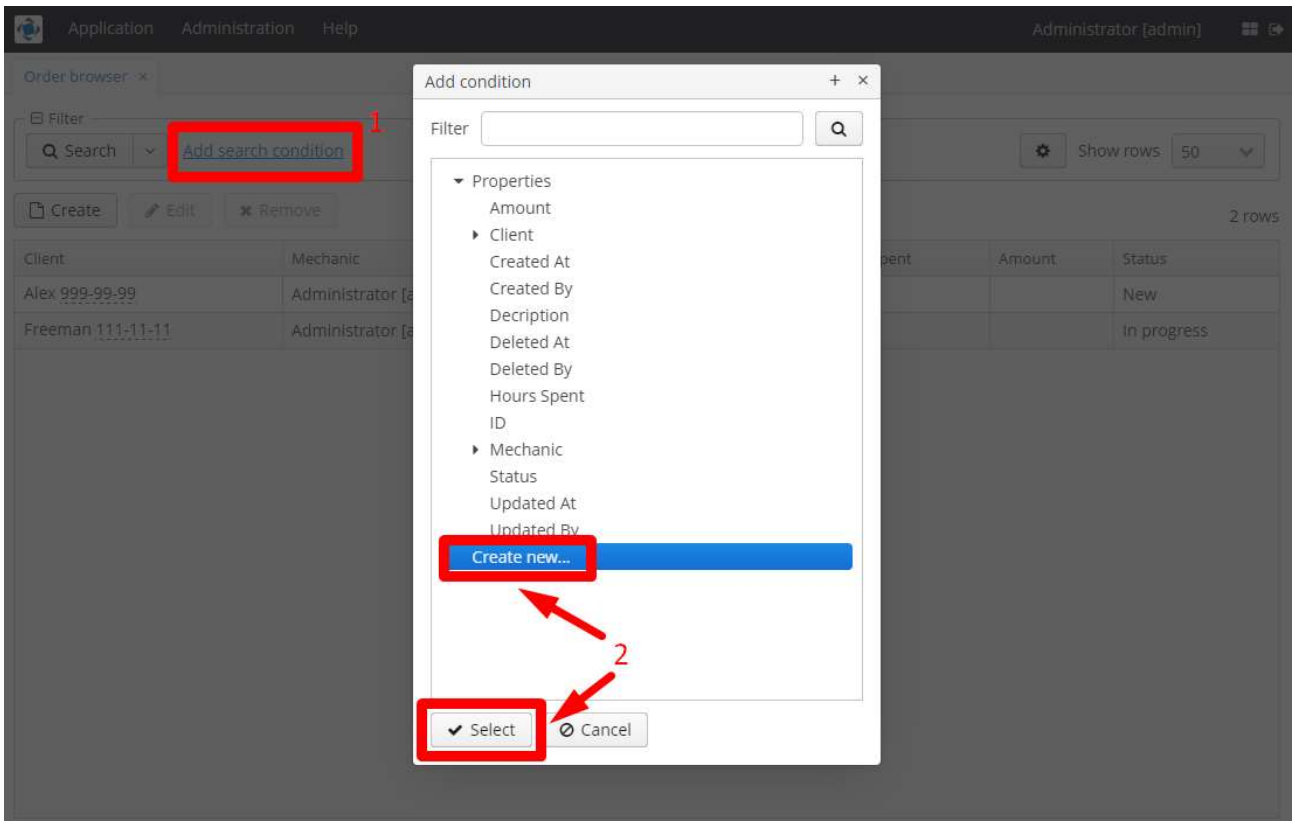


In a similar way you can filter by all types of attributes (string, numeric, boolean, etc).

5.1.2 Part Specified in Order Filter

Sometimes it is not possible to filter records without join operation. Let's take the following example: we want to filter all the orders that contain some particular part in it.

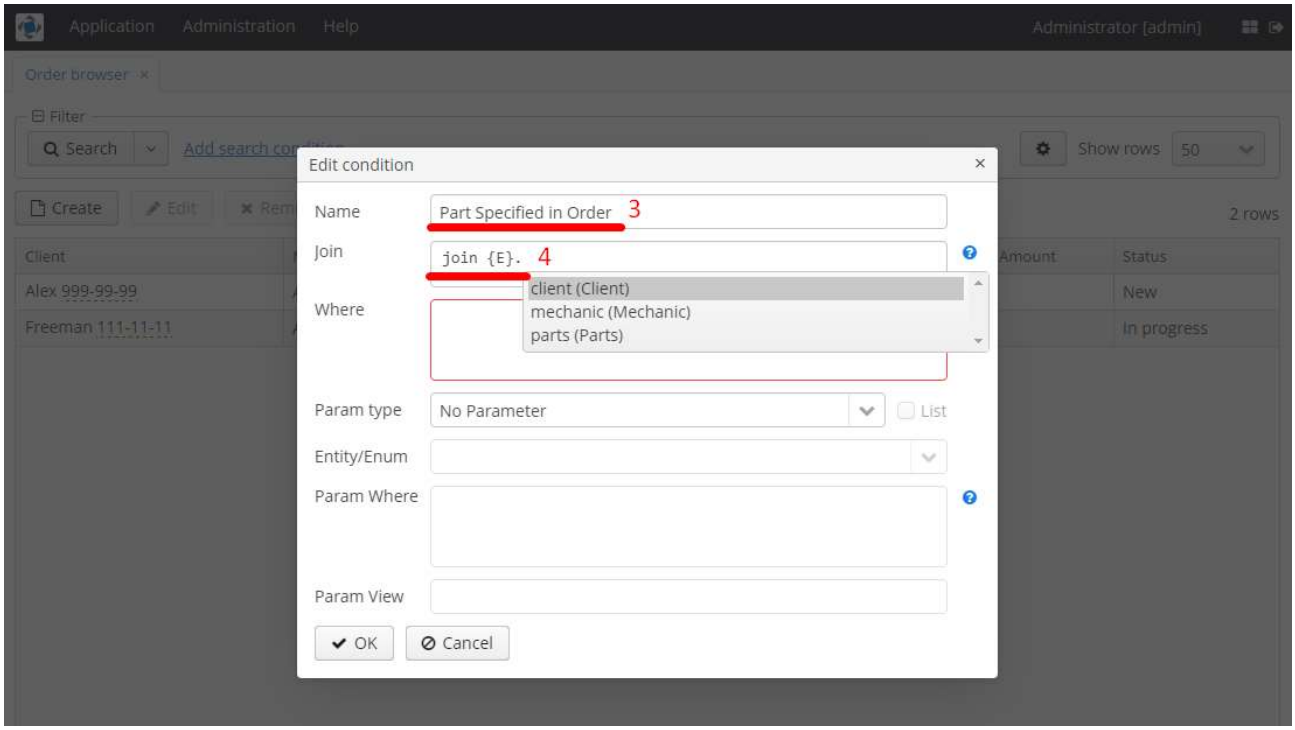
1. Click the **Add search condition** link
2. Select the **Create new…** option



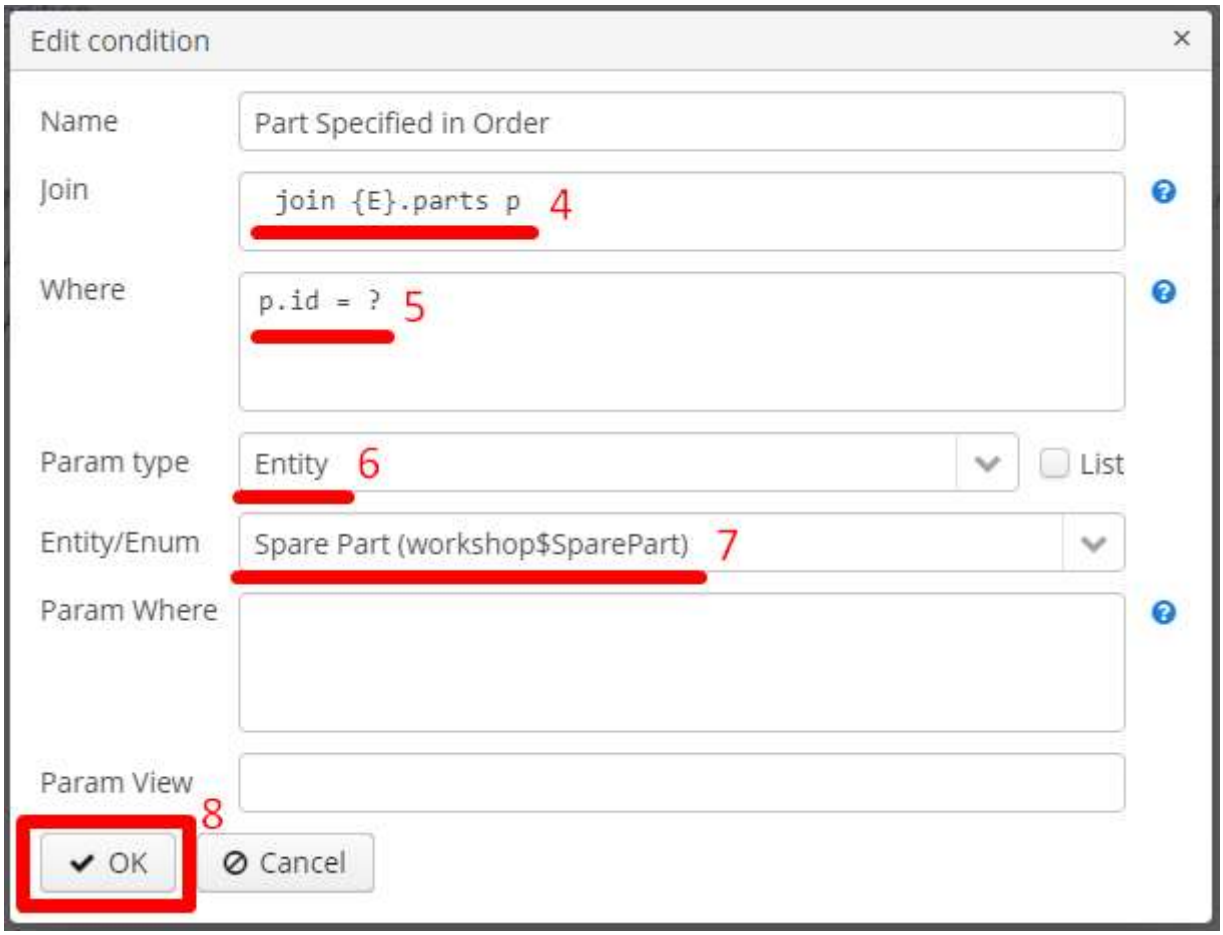
3. Set condition **Name**: ***Part Specified in Order***
4. Our main entity in the screen is *Order*, which has *Set<SparePart> parts* field, containing all the used parts. Let's join parts to the orders: type ***join {E}.*** and the system will show you suggestions. Select the ***parts*** suggestion or just type

it manually and assign alias ***p*** to the joining table.

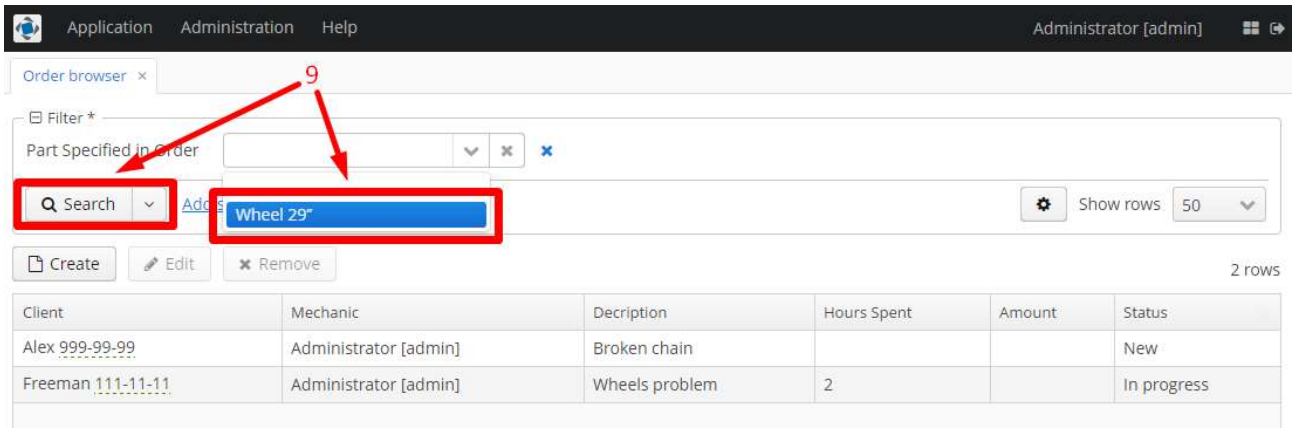
You should get the following **Join** clause: ***join {E}.parts p***



5. Input ***p.id = ?*** in the **Where** field. Question here means parameter, that we will send to the condition from the generic filter UI.
6. Select **Param type**: ***Entity***
7. Set **Entity/Enum**: ***SparePart***
8. Click **OK**



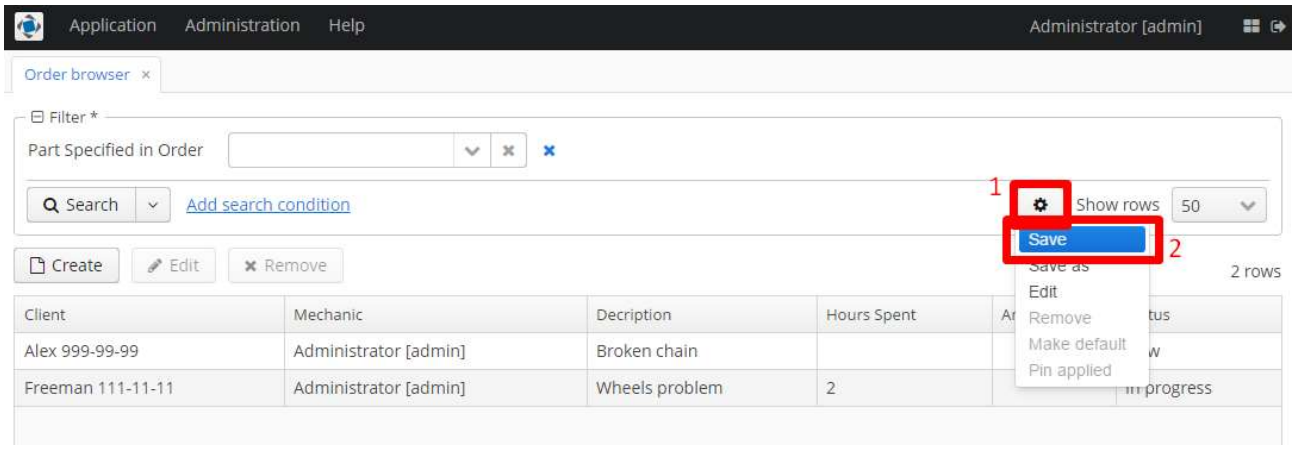
9. Specify the ***Wheel 29"*** value in your custom filter and click **Search** to see the result list



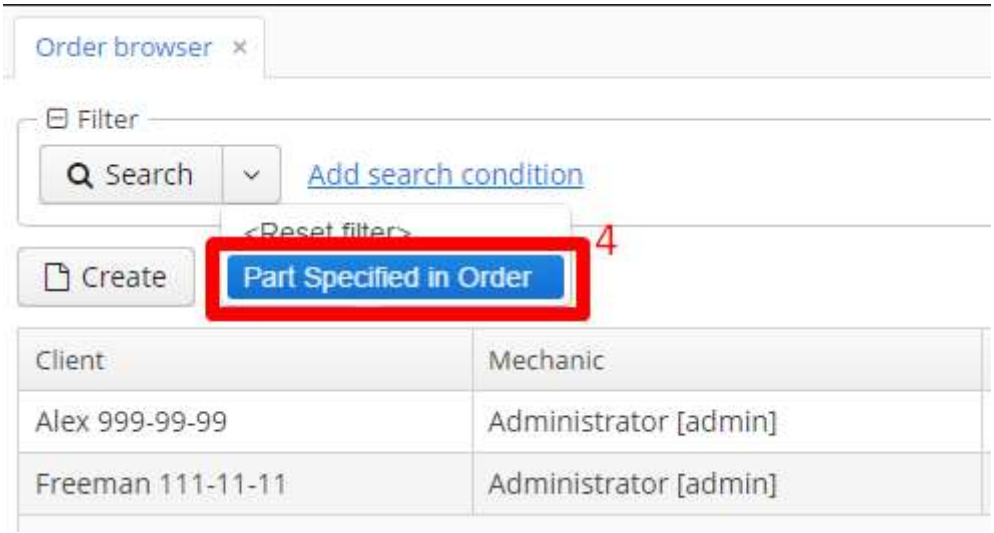
5.1.3 Reusing Filters

Obviously, there are a number of filters, that end-users reuse frequently. For that purpose you can save filters you create. Let's save the "part specified in order" filter we have just created.

1. Click the **cogwheel button** on the right hand side of the filter component
2. Click **Save**



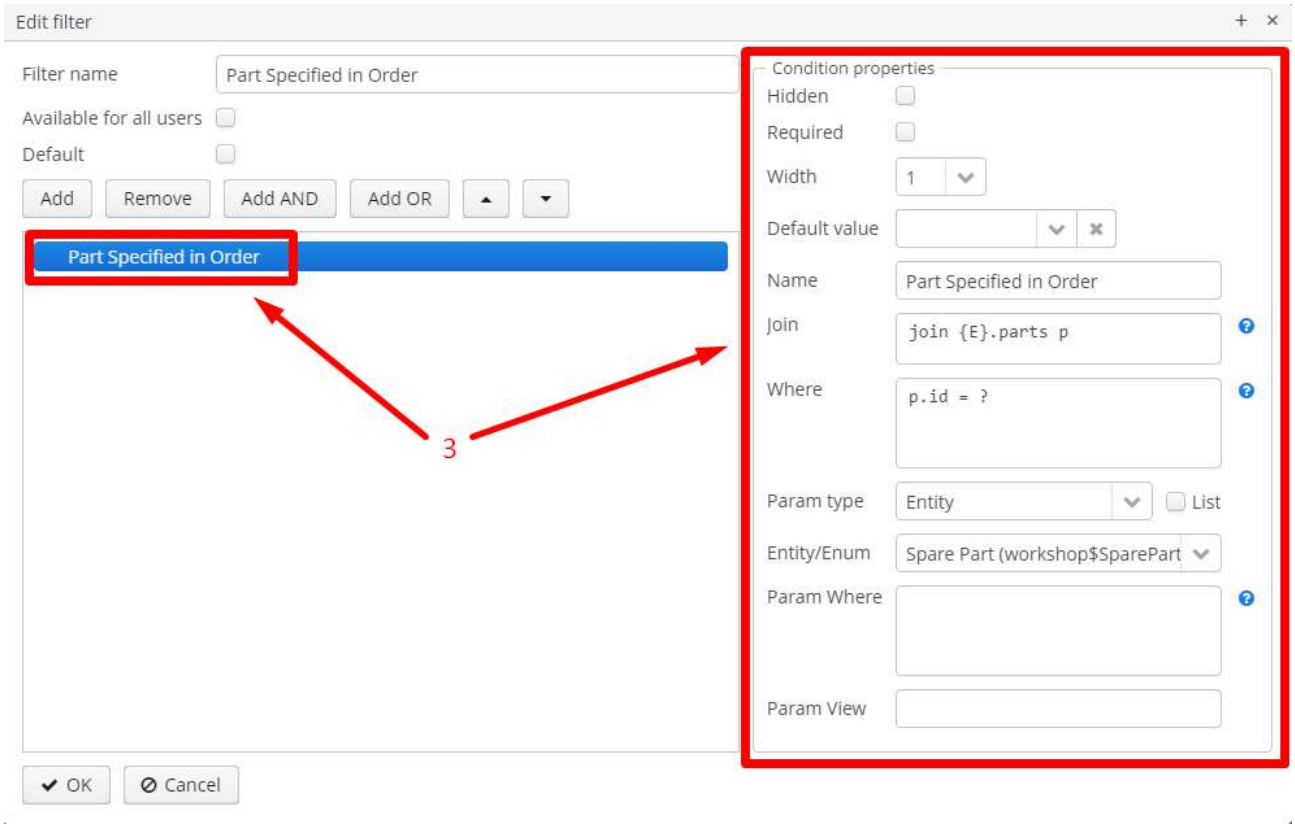
3. Input **Filter name**: ***Part specified in order***
4. Now you can access your filter by clicking the **down arrow button**, placed next to the Search one. Apply the saved filter.



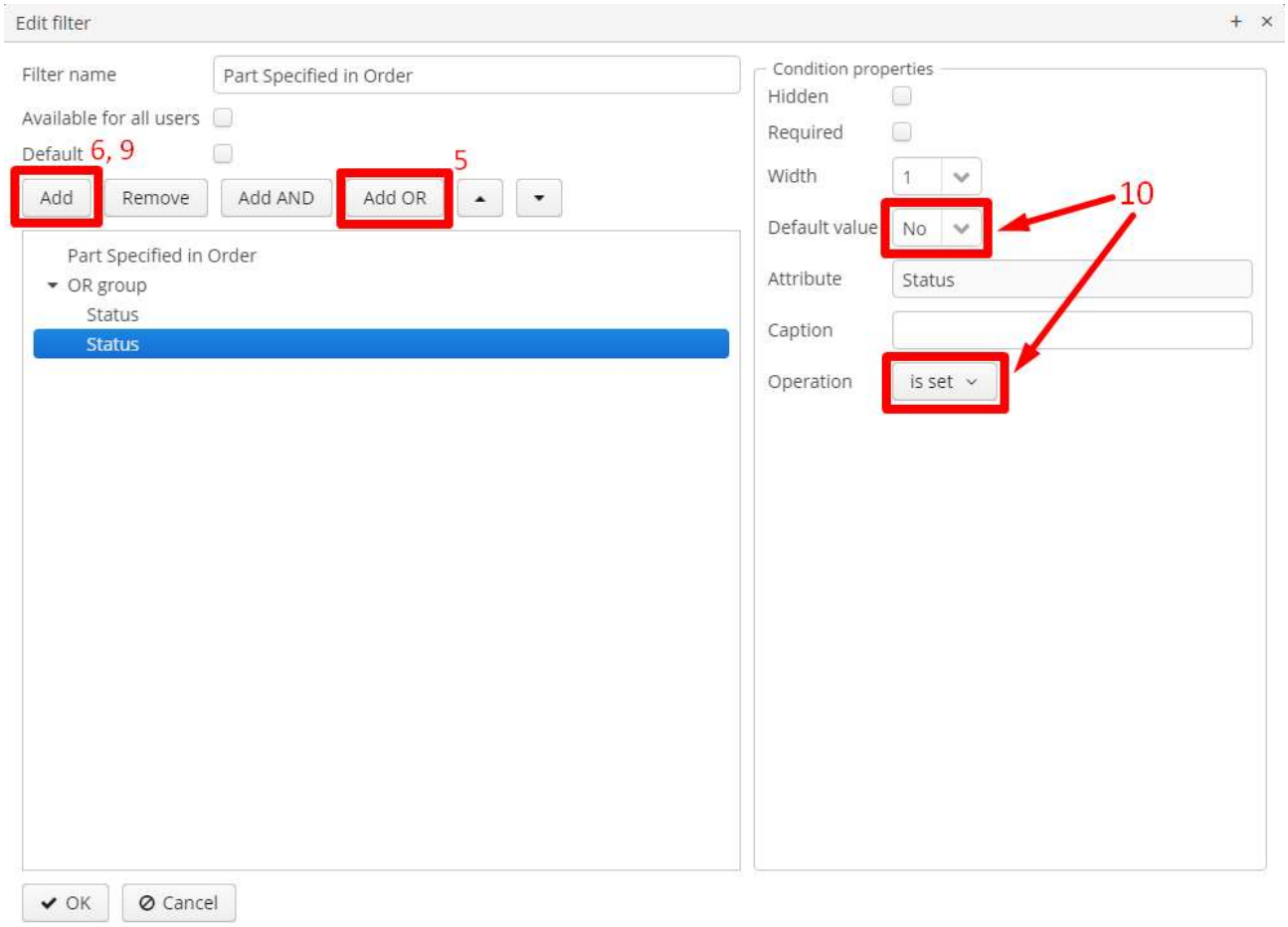
5. Applying a saved filter you can edit, remove it or make selected filter as default one by clicking the **cogwheel button**. Default filters appear in the screen automatically after opening.

5.1.4 Editing Saved Filters and Grouping Conditions

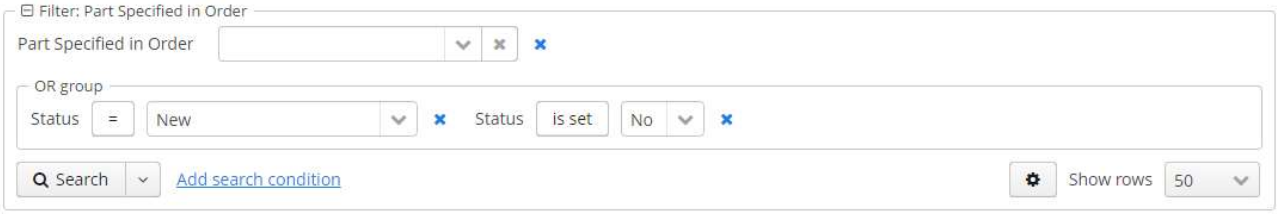
1. Open the saved filter
2. Click the **cogwheel button — Edit**
3. Selecting conditions users are able to edit them



4. Also there is an ability to additional conditions and group them using OR and AND logical operators. Let's add one more condition. For example, we want to control our stock of spare parts, so, we would like to know how many parts of some particular type we need for the new or unstatused orders. Logical expression we are going to build is the following:
   Part Specified in Order = ? AND (*Status* = NEW OR *Status* is not set).
5. Click the **Add OR** button and check that the **OR group** item is selected in the conditions list
6. Click **Add**
7. Select the **Status** attribute
8. Set **Default value:** ***New***
9. Select the **OR group** item in the conditions list and add one more condition for the Status attribute
10. Change Operation to **[is set]** one and specify **No** in the **Default value** field



11. Click **OK**
12. Now we can see our complex filter



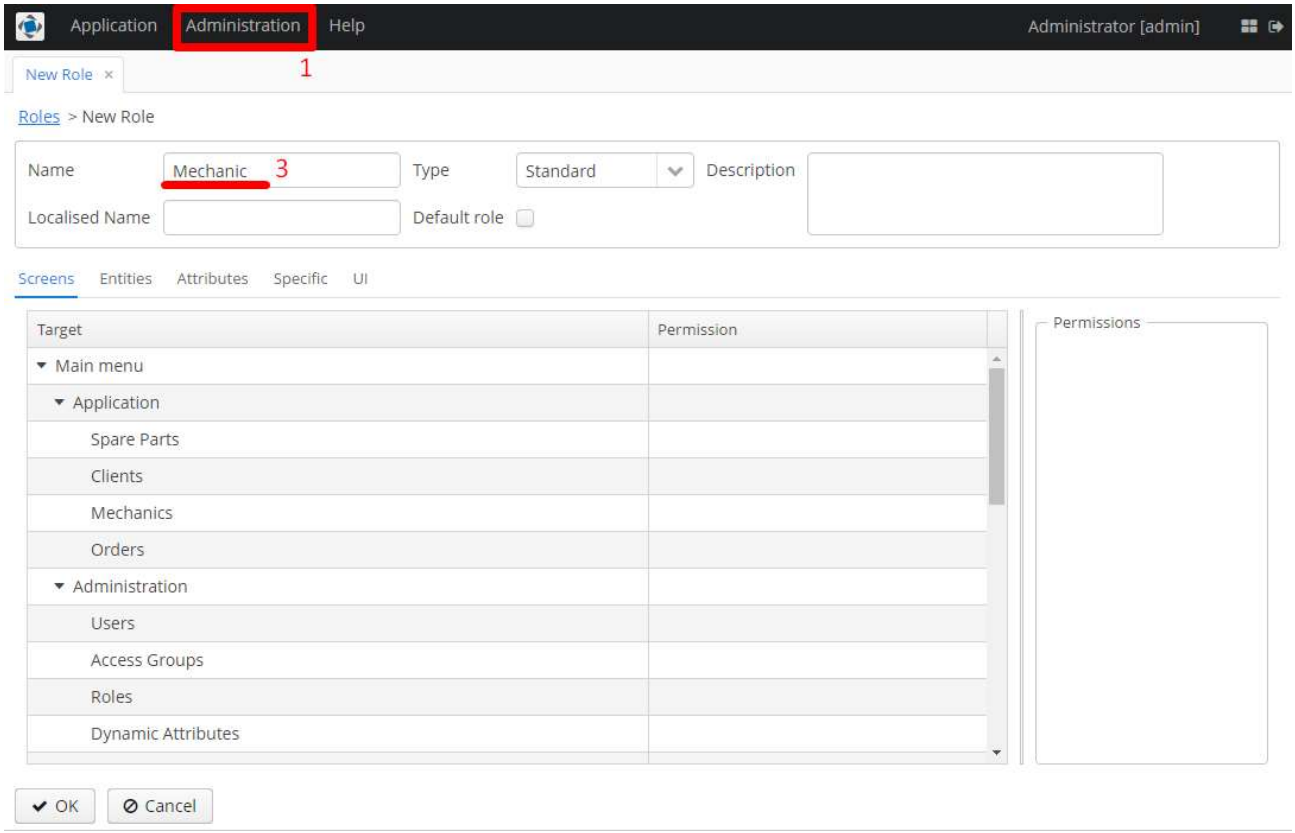
13. Editing filters you can share them between users, set them as default screen filter, hide some conditions or groups of conditions. More information on generic filter component can be found here.

5.2 Security Setup

The platform has built-in functionality to manage users and access rights. This functionality is available from the **Administration** menu. The CUBA platform security model is role-based and controls CRUD permissions for entities, attributes, menu items and screen components and supports custom access restrictions. All security settings can be configured at runtime. There is also an additional facility to control row level access.

We need a role for mechanics in our application. A Mechanic will be able to modify an order and specify the number of hours they spent, and add or remove spare parts. The Mechanic role will have limited administrative functions. Only admin will be allowed to create orders, clients and spare parts.

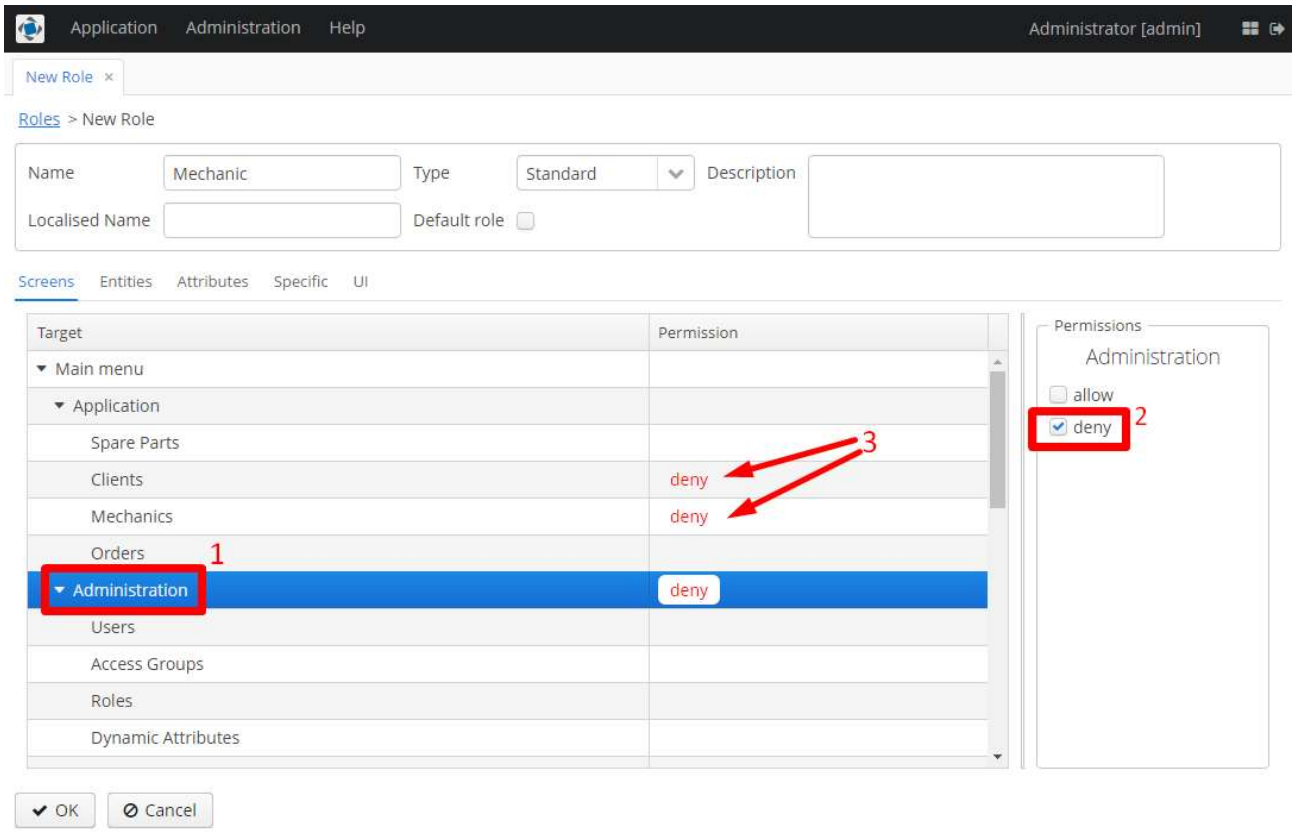
1. Open **Administration — Roles** from the menu
2. Click **Create**
3. Set **Name**: ***Mechanic***



5.2.1 Screen Permissions

We want to restrict access to the Administration screens for all Mechanic users, so let's forbid the Administration menu item. Also, mechanics don't need access to the mechanics and clients browsers, let's forbid the corresponding screens too.

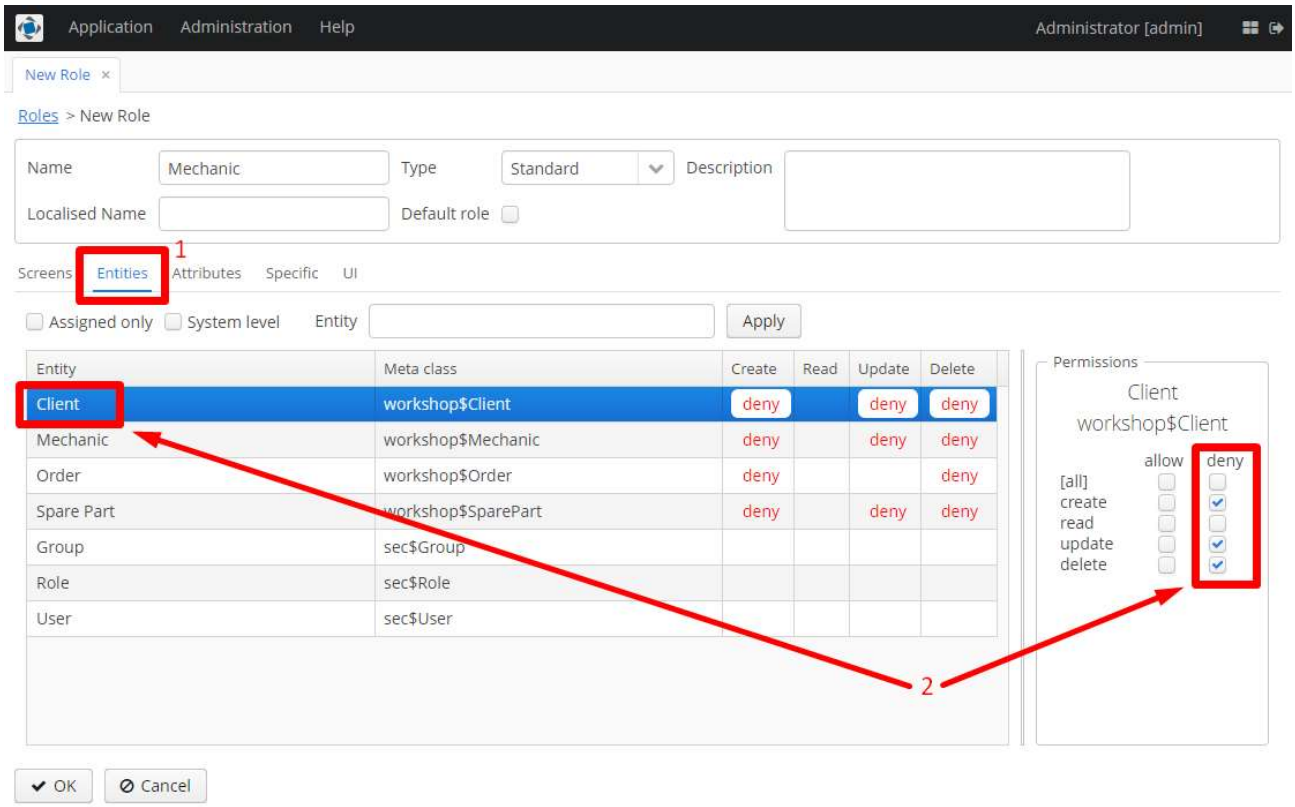
1. Select the **Administration** row in the Screens table
2. Select **deny**
3. Similarly deny access for **Clients** and **Mechanics**



5.2.2 CRUD Permissions

As it has already been mentioned we want a mechanic only allow read access to clients, mechanics and parts. Also a mechanic should not be able to create or delete an order in the system. Only update it's status, hours spent and manipulate with parts in the order.

1. Open the **Entities** tab
2. Select the **Client** entity and forbid **create**, **update** and **delete** operations

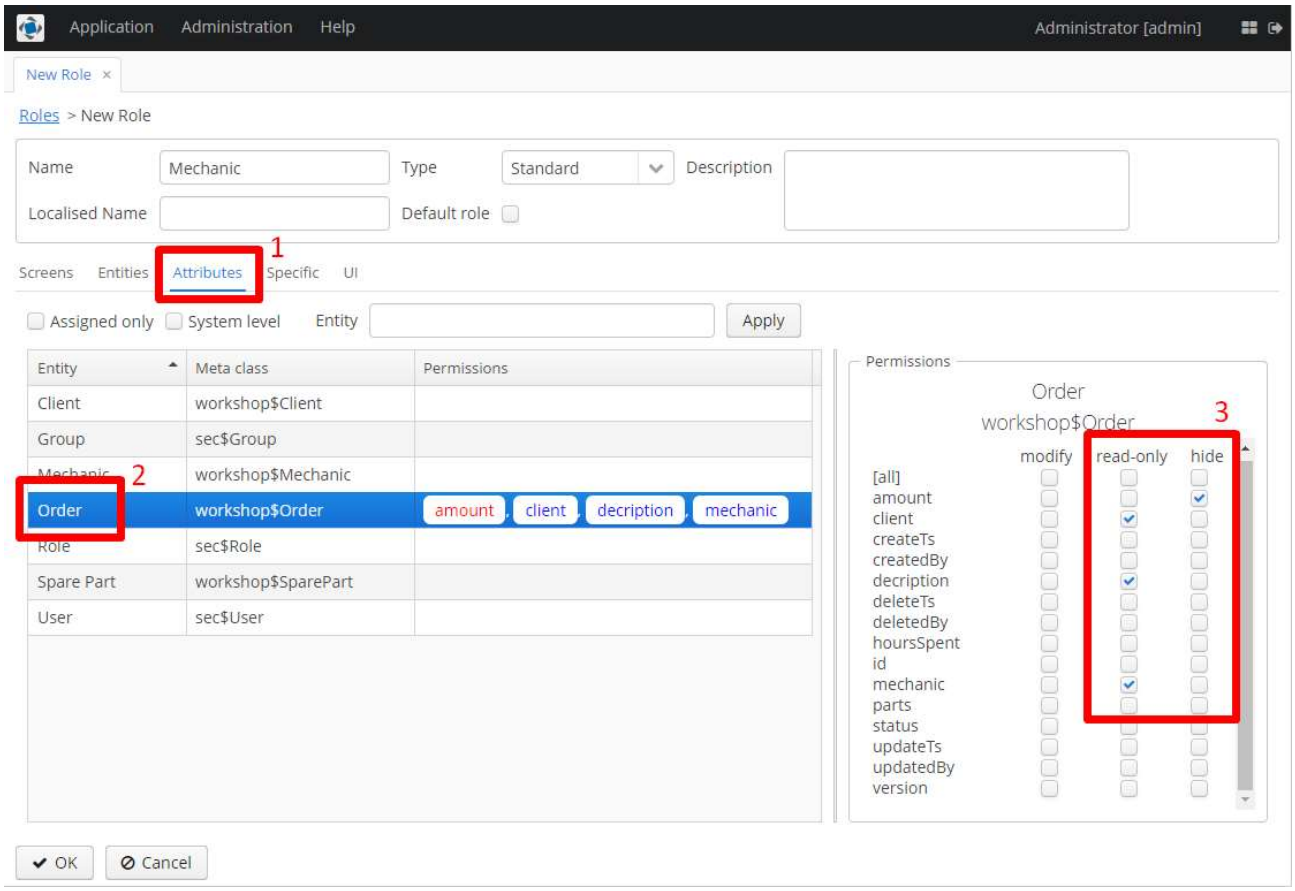


3. Repeat the second step for the **Mechanic** and **SparePart** entities
4. Forbid only **create** and **delete** operations for the **Order** entity

5.2.3 Attribute Permissions

On attribute level we don't want to allow a mechanic to set values for the client, mechanic and description attributes; also we would like to hide the amount field from them.

1. Open the **Attributes** tab
2. Select the **Order** row and
3. Tick **read-only** for **client**, **mechanic** and **description**; tick **hide** for the **amount** attribute

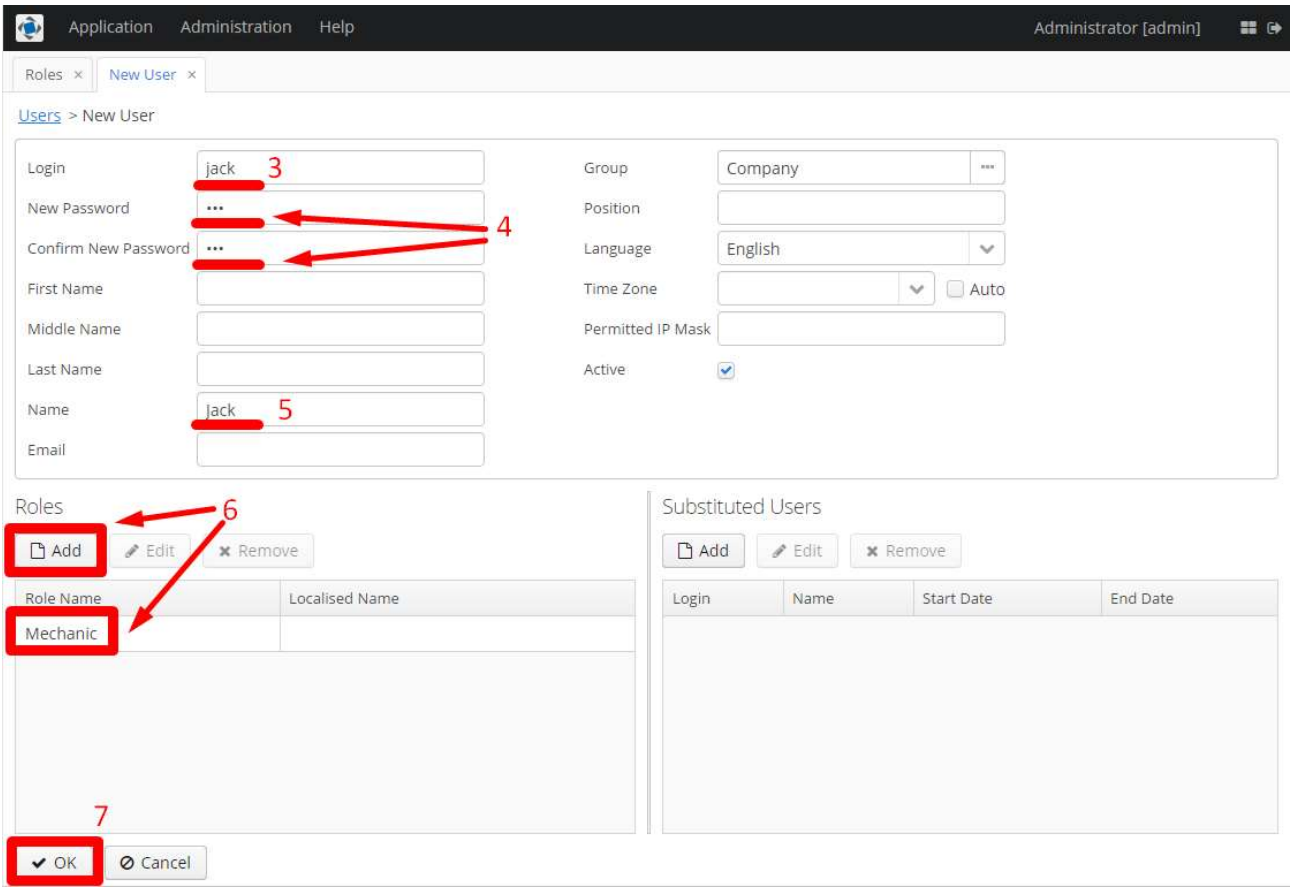


4. Click **OK** to save the role

5.2.4 Role-based Security in Action

Let's apply the mechanic role to a new user and take a look on the application from the mechanic's point of view.

1. Open **Administration — Users**
2. Click **Create**
3. Set **Login**: ***jack***
4. Set **password** and **password confirmation**: ***qwe***
5. Set **Name**: ***Jack***
6. Add the **Mechanic** role to user **Roles**
7. Click **OK** to save the user



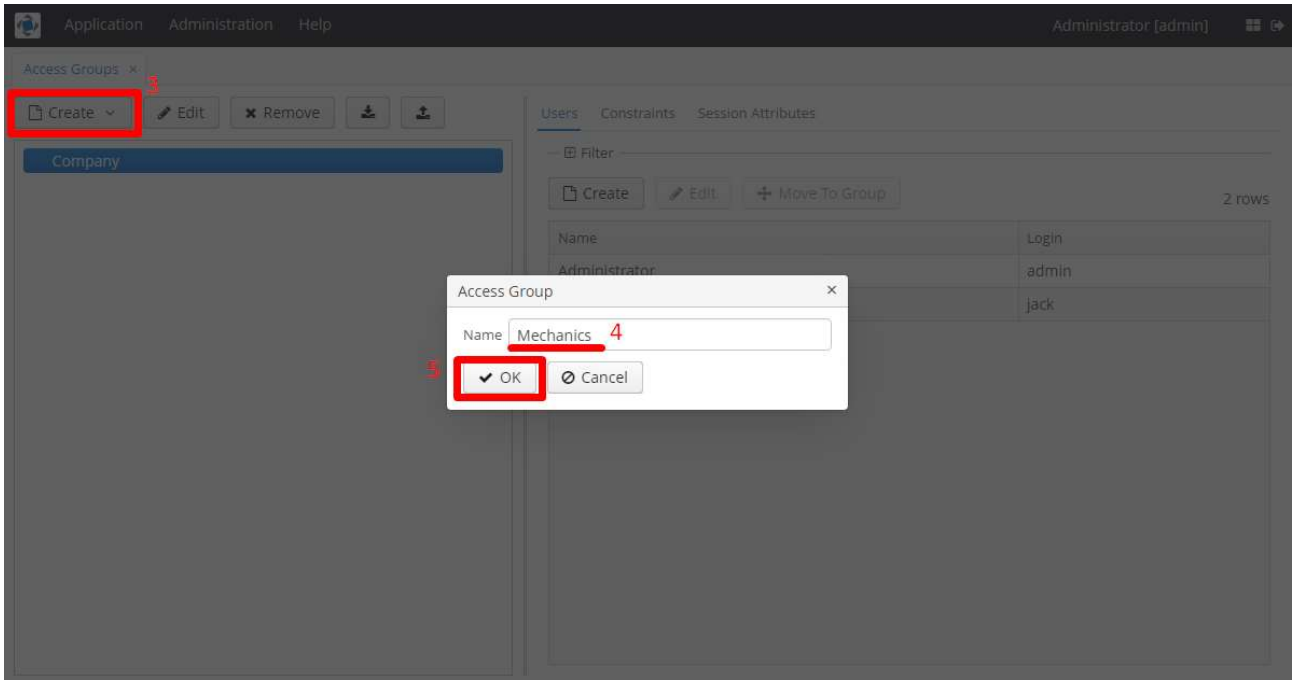
8. Let's exit the application by clicking the **top-right door button** of the screen
9. Login under the new user Login:***jack***, Password:***qwe***
10. The Administration menu item is hidden as well as Mechanics and Clients browsers;
    The Spare Parts browser is available in read-only mode;
    The Orders browser allows us only editing of existing records;
    The Order editor allows only changing the Hours spent and Status fields, as well as adding parts to an order; the amount field is hidden from our user.

5.2.5 Row-level Security

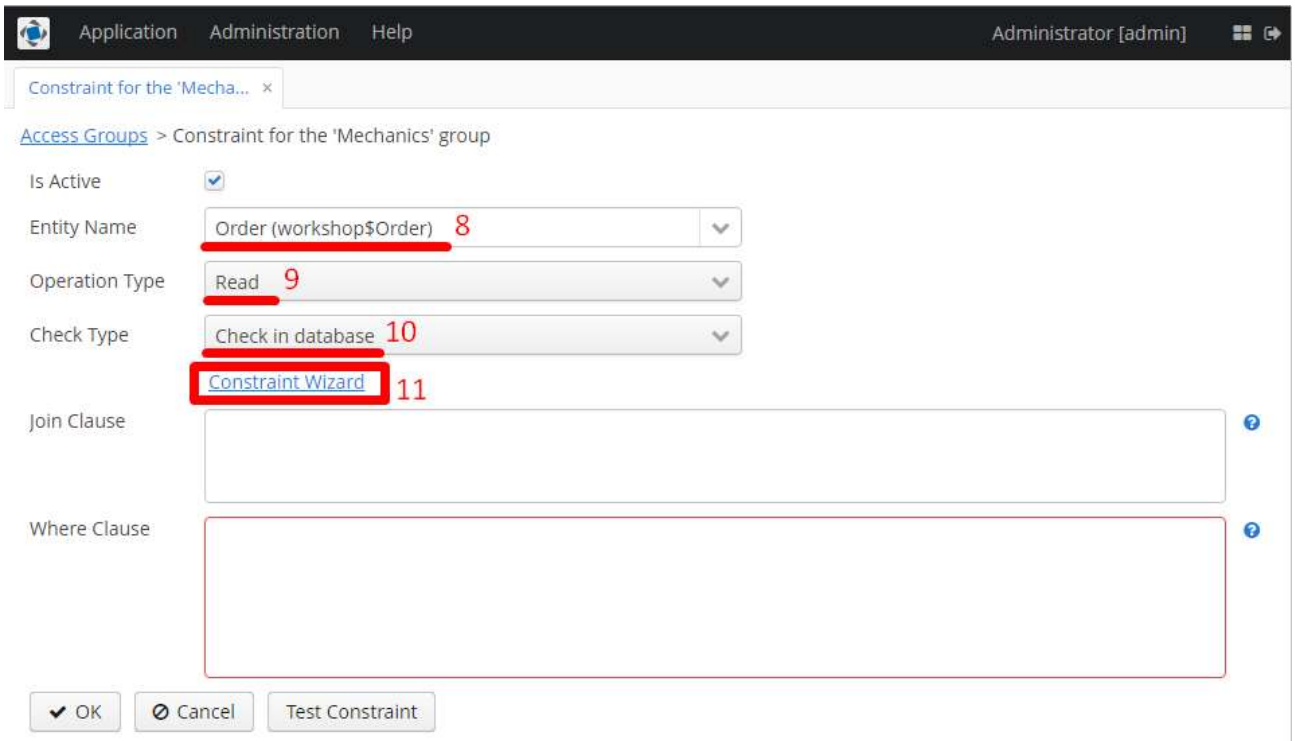
The CUBA Platform Security subsystem allows you to control and restrict access on the level of records using the **access group** mechanism. For example, we want to restrict access of our mechanics to see only orders that were assigned to them.

1. **Log out** from the system and **login** under administrator (**Login**: ***admin***; **Password**: ***admin***)
2. Open **Administration — Access Groups** from the menu. The groups have hierarchical structure, where each element defines a set of constraints, allowing controlling access to individual entity instances (at row level).
3. Click **Create — New**

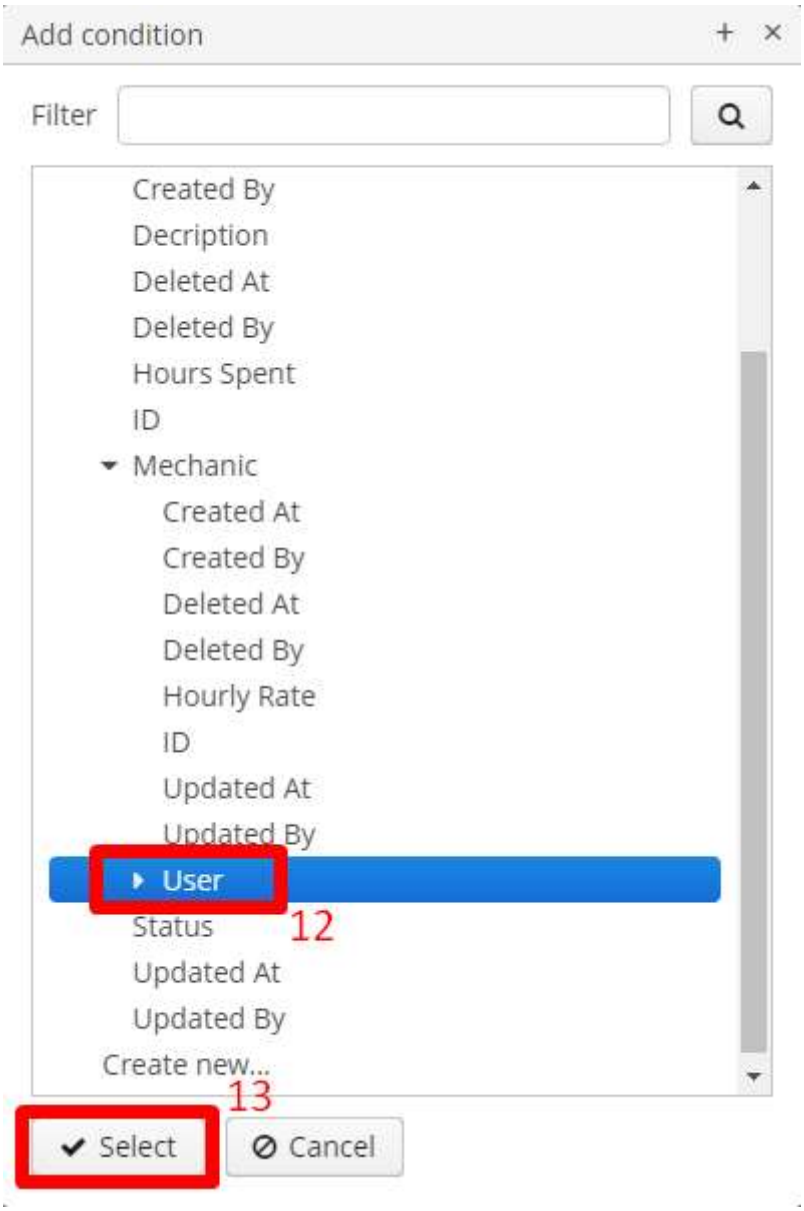
4. Set **Name:** ***Mechanics***
5. Click **OK**



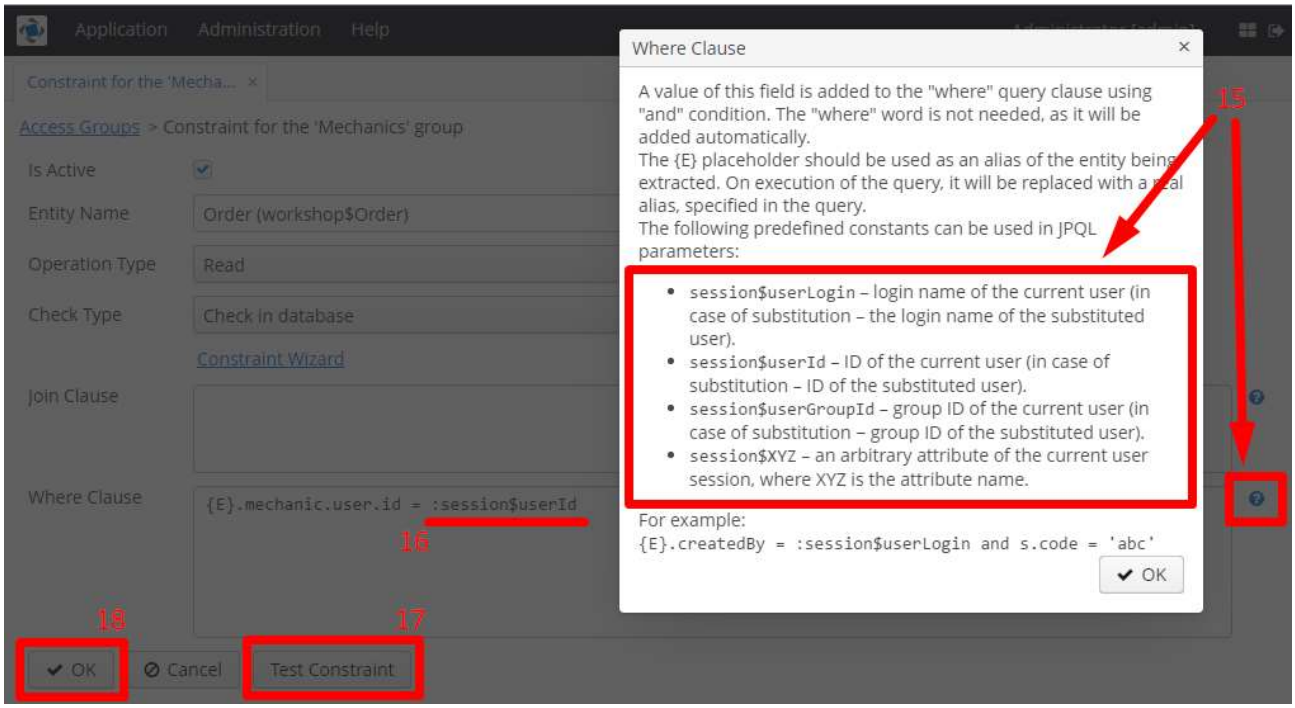
6. Open the **Constraints** tab for the newly created group
7. Click **Create** in the **Constraints** tab
8. Set **Entity Name**: ***Order***
9. We can define what operation type we want to assign restriction to. Keep **Operation Type:** ***Read***
10. For some operation types it is not possible to perform check on the database level, e.g. delete and update operations. For this reason there are three **Check Types** implemented in the platform: *Check in database* (using JPQL), *Check in memory* (using Groovy script; the only option for Create, Update, Delete, All operation types) and *Check in database and in memory* (using both JPQL and Groovy). In our example we will use the ***Check in database*** Check Type
11. For the *Check in database* Check Type you can specify a JPQL query in the same way as we did for custom conditions in the generic filter. Click the **Constraint Wizard** link



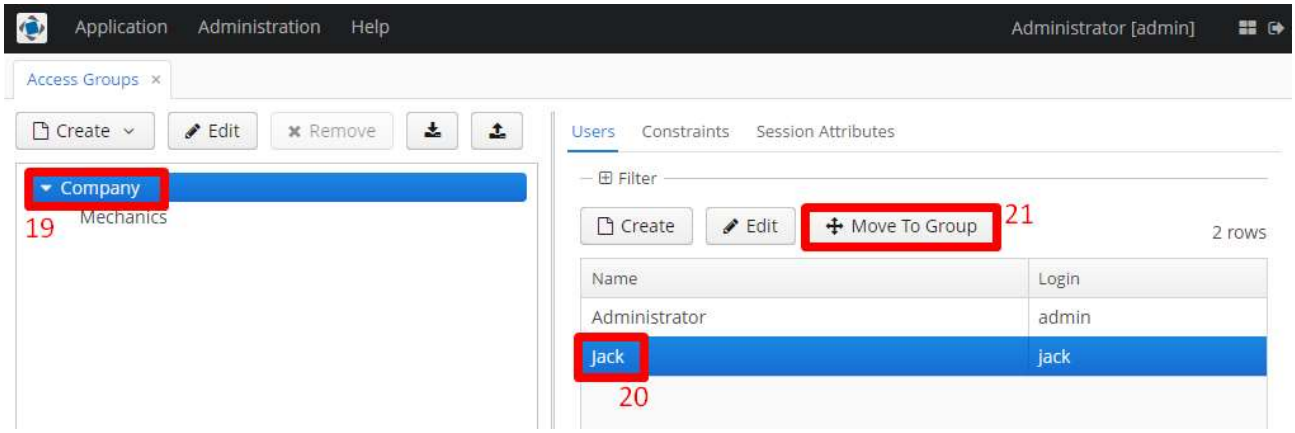
12. Expand the **Mechanic** field and select the child **User** attribute
13. Click **Select**



14. Click **OK** in the Define query window. The platform has generated the where clause as *{E}.mechanic.user.id = 'NULL'*. We will need to tune it a bit.
15. Click the question mark located in the top-right corner of the Where Clause filed to see what predefined constants can be used here
16. We need ID of the current user to compare with the mechanic's user id. So, change *'NULL'* to *:session$userId*. Finally you should have the following code: ***{E}.mechanic.user.id = :session$userId***
17. Click **Test Constraint**
18. If test shows that constraint is valid, then click **OK**



19. Select the **Company** group on the **Access Groups** screen
20. Select ***Jack*** in the **Users** tab
21. Click **Move To Group**



22. Select the **Mechanics** group and click **Select**
23. Let's now assign one of our mechanics with the new user. Go to **Application - Mechanics** in the main menu
24. Select the **mechanic with hourly rate of 10**
25. Click **Edit**

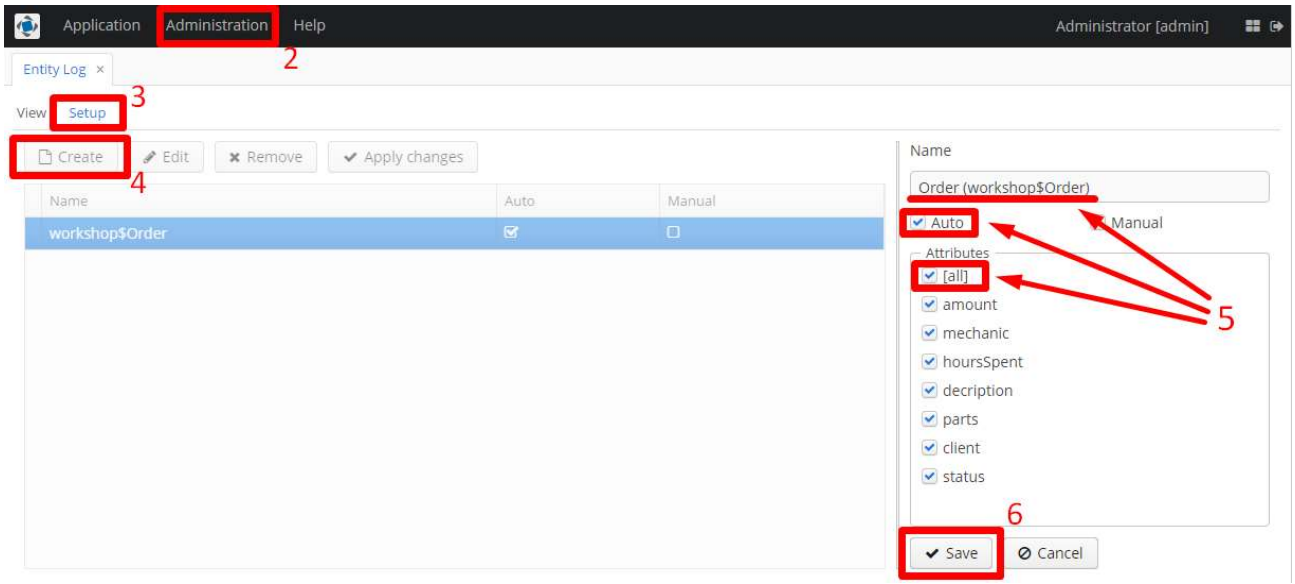
26. Specify the ***Jack*** user for this record and click **OK**
27. Now our mechanic is associated with the jack user. Re-login into the application using Jack's credentials: **Login**: ***jack***; **Password**: ***qwe***
28. Open the Orders browser (**Application — Orders**). Now we can only see the orders assigned to Jack

5.3 Audit

There is a requirement in our functional specification to track critical data changes. CUBA Platform has a built-in mechanism to track entity changes, which you can configure to track operations with critical system data.

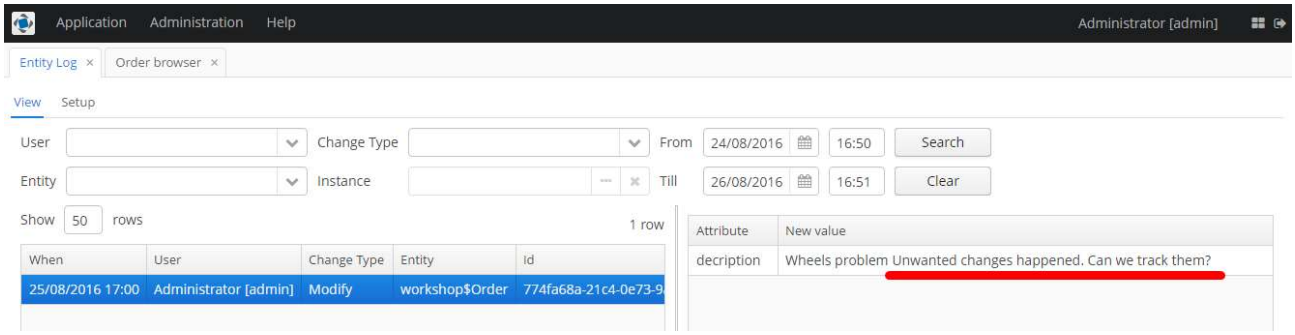
Keeping track of our application, it happens when one day someone has accidentally erased the order description. It is not appropriate to call the client on the phone, apologize and ask them to repeat the what needs to be done. Let's see how this can be avoided.

1. Re-login into the application as administrator (**Login**: ***admin***; **Password**: ***admin***)
2. Open **Administration — Entity Log** from the menu
3. Go to the **Setup** tab
4. Click **Create**
5. Set **Name**: ***Order***
   Auto: ***true***
   Attributes: *all*
6. Click **Save**



Now the system will track all CRUD changes of all the fields of the Order entity. Go to the order browser and change status or description of any record, them get back to the Entity Log screen and you will see who made what change!

I changed the description of one of the orders and see what we have in the entity log:



5.4 First Launch Conclusion

While spending short time exploring the CUBA Platform and its features we have already finished the following requirements from the functional specification:

1. ~~Store customers with their name, mobile phone and email~~
2. ~~Record information about orders: price for repair and time spent by mechanic~~
3. ~~Keep track of spare parts in stock~~
4. Automatically calculate price based on spare parts used and time elapsed
5. ~~Control security permissions for screens, CRUD operations and records' attributes~~
6. ~~Perform Audit of critical data changes~~
7. Provide API for a mobile client to place an order

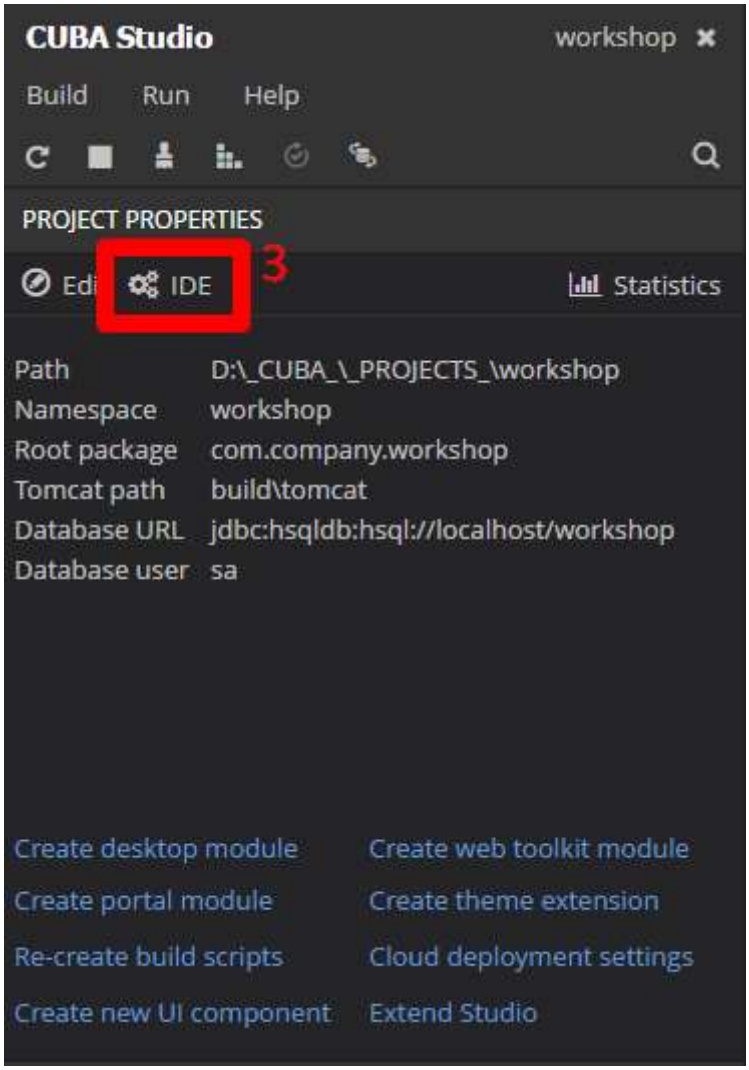
Note that we didn't even type any line of source code, everything has been provided by the platform or scaffolded by the Studio. Let's move forward towards business logic.

6 DEVELOPMENT BEYOND CRUD

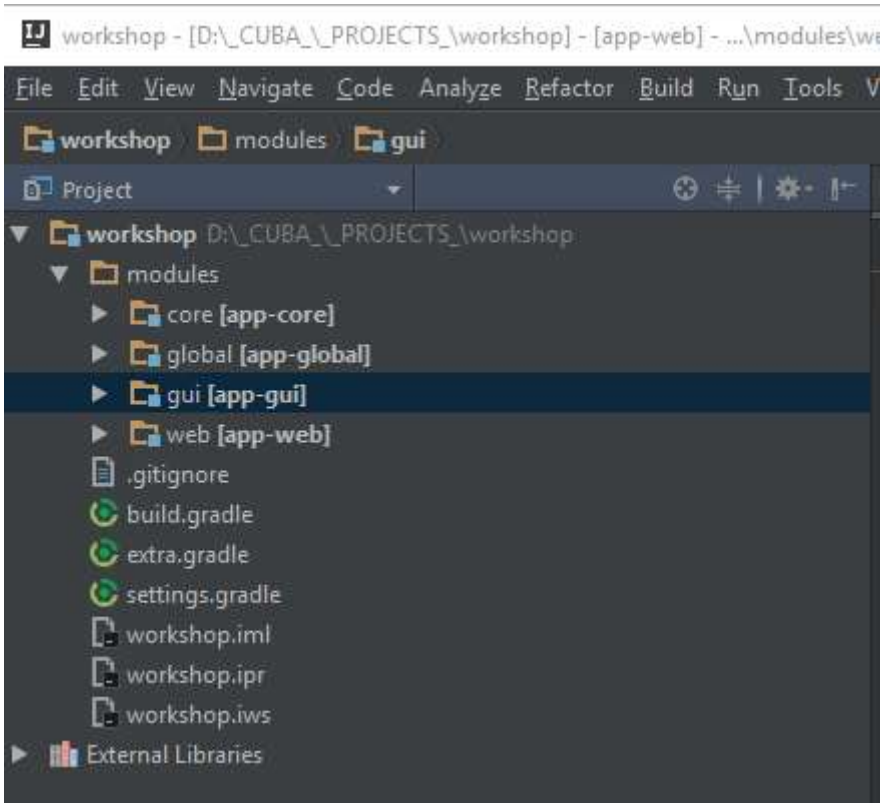
6.1 Integration with IDE and Project Structure

Let's have a look how our project looks from inside. Keep your application up and running and follow the steps:

1. Launch IntelliJ IDEA. The IDE should be up and running to enable integration with the CUBA Studio. If you don't have the CUBA Plugin installed, please install it, cause it is used as a communication bridge between the Studio and IDE
2. Go to the Studio and click the **IDE** button in the **Project properties** section. The Studio will generate project files and open the project in the IDE



3. Move to the IDE and press **Alt+1** to see the project structure. By default any project consists of 4 modules: **global**, **core**, **web**, **gui**.
   global - data model classes
   core - middle tier services
   gui - common component interfaces; screens and components for both Web and Desktop clients and
   web - screens and components for Web client



6.2 Customization of Existing Screens and Hotdeploy

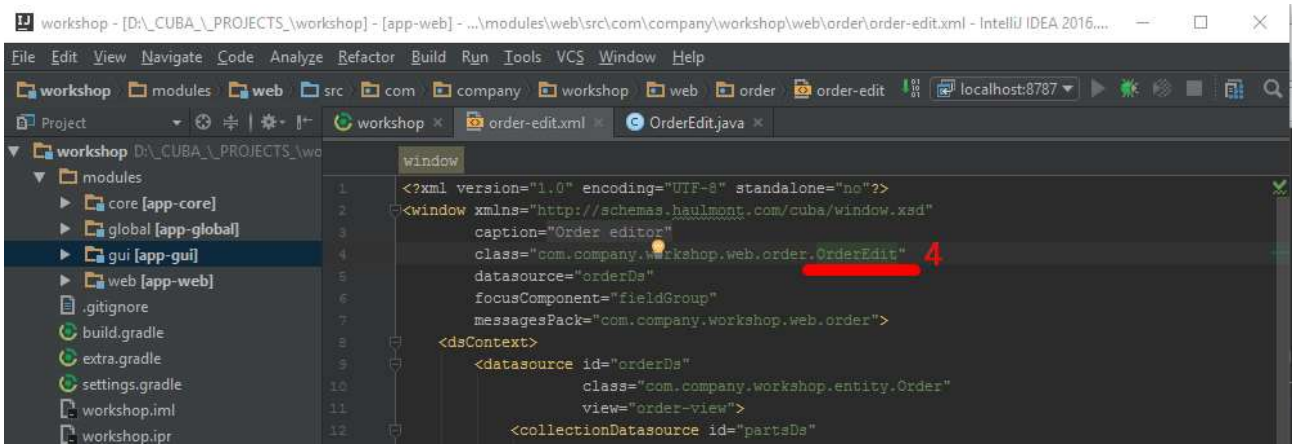
In this chapter we will polish our Orders browser and editor by adding logic into controller and changing the user interface.

6.2.1 Initialization of a New Record in Editor

Let's solve the first problem with empty status of the newly creating orders. A new order should be created with the NEW order status pre-set.

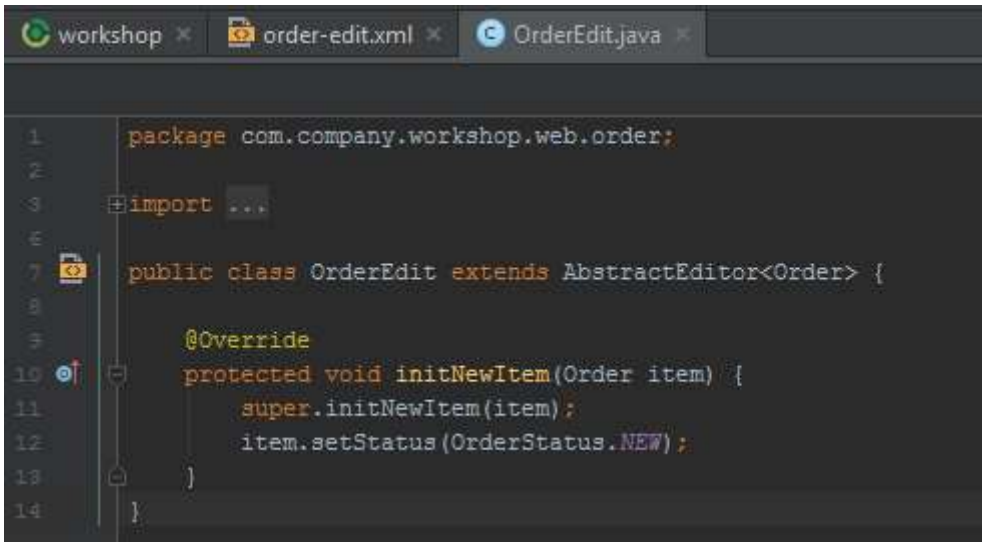
1. Go to the **Screens** section of the navigation panel in the CUBA Studio
2. Select the **order-edit.xml** screen
3. Click the **IDE** button on top of the section. Screen descriptor appears in your IDE. You can make any changes right from the source code, because The Studio and an IDE has two ways synchronization

4. Hold **Ctrl** button and click on **OrderEdit** in class attribute of the XML descriptor to navigate to its implementation



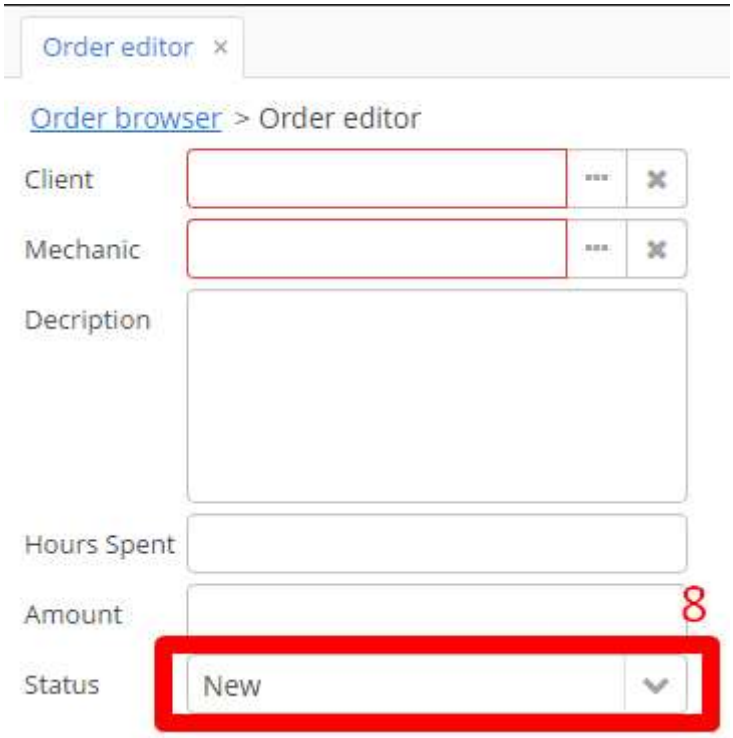
5. Override method ***initNewItem*** and set status ***OrderStatus.NEW*** to the passed order:

```
public class OrderEdit extends AbstractEditor<Order> {
        @Override
        protected void initNewItem(Order item) {
                super.initNewItem(item);
                item.setStatus(OrderStatus.NEW);
        }
}
```



6. We haven't stopped our application and it is still up and running in the browser. Open/Reopen **Application — Orders** screen
7. Click **Create**
8. We see our changes, although we haven't restarted the server. The CUBA Studio automatically detects and the hot-deploys changes, except for the data

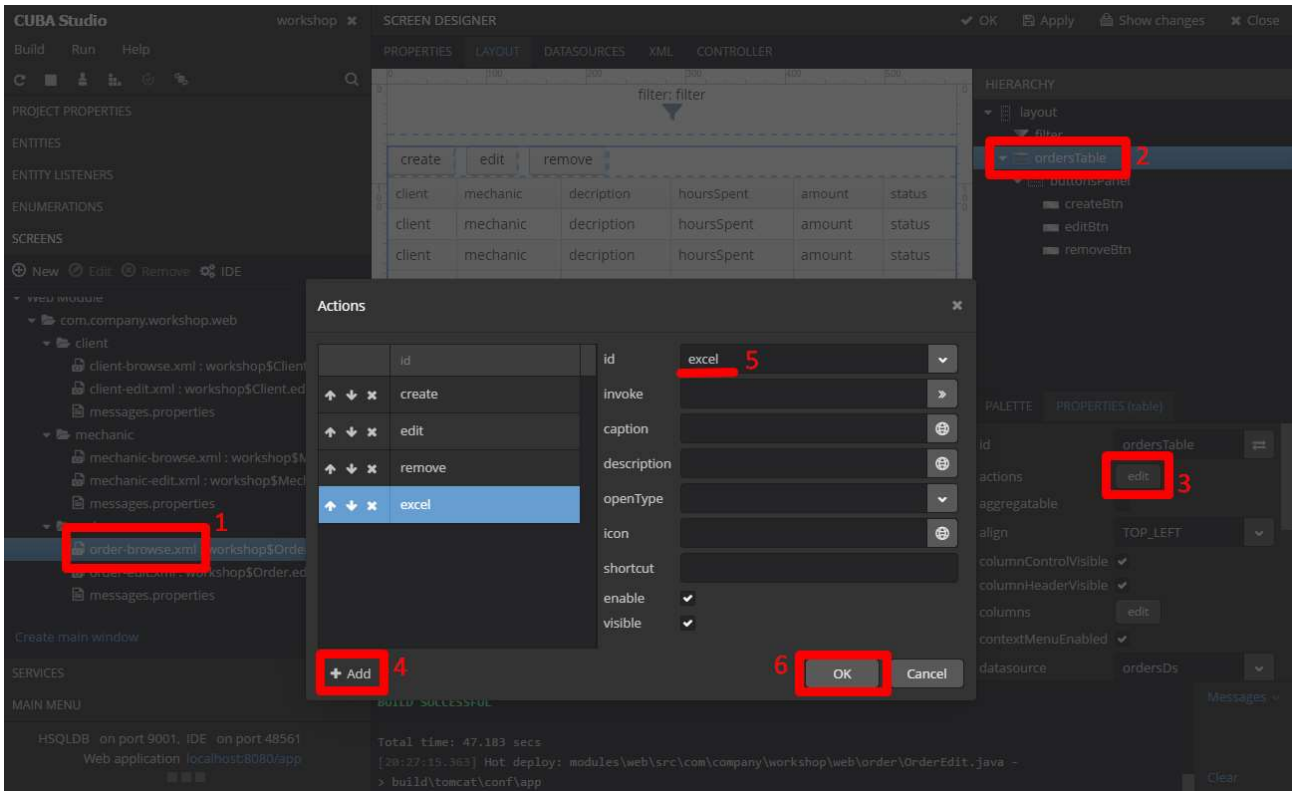
model, which saves a lot of time while UI development and business logic development



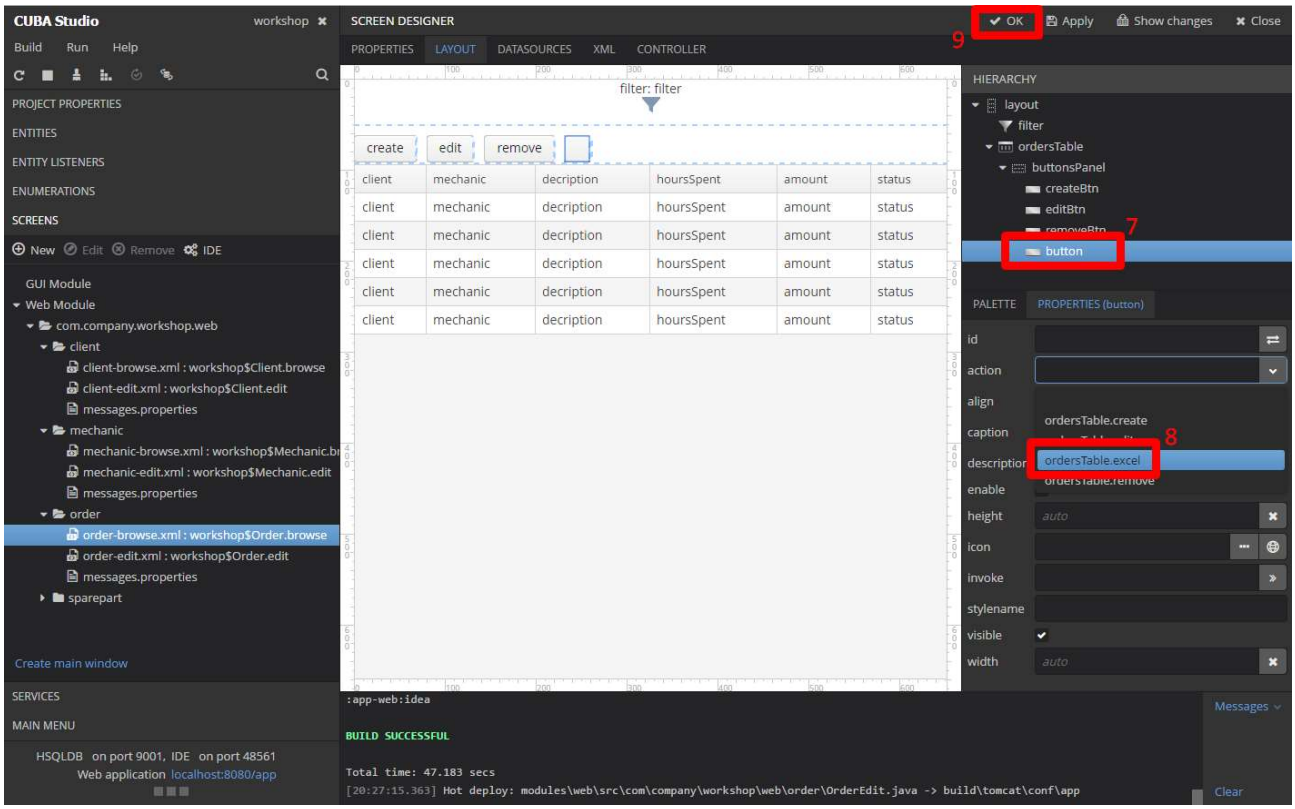
6.2.2 Adding Standard Excel Action to the Orders Browser

The standard screens contain **Create**, **Edit**, and **Remove** actions by default. Let's add an action to export the order list to **Excel**, which is also a standard action you can use out of the box. You can follow the link to learn more about standard actions.

1. Open **order-browse.xml** screen in the Studio
2. Select table component, go to properties panel
3. Click the **edit** button in the actions property
4. **Add** a new action row to the list
5. Specify id as ***excel*** for this action
6. Click **OK**



7. Add a new button to the button panel (**drug and drop** it from the components palette into the hierarchy of components)
8. Select ***ordersTable.excel*** action for button using properties panel
9. Save the screen by clicking **OK** in the top-right corner

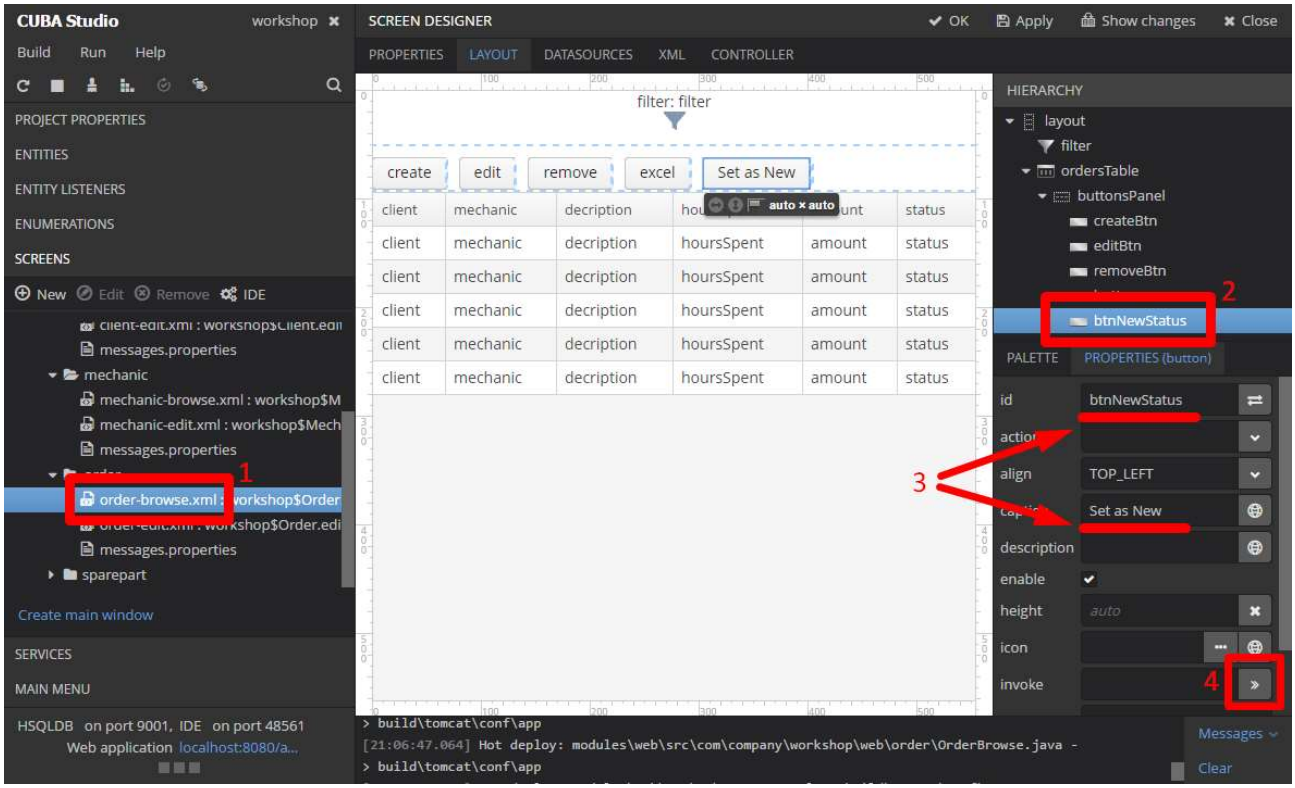


10. Let's use magic of hotdeploy once again. **Open/Reopen** the Orders screen
11. Click **Excel** to export your orders to an xls file

6.2.3 Adding Custom Behaviour

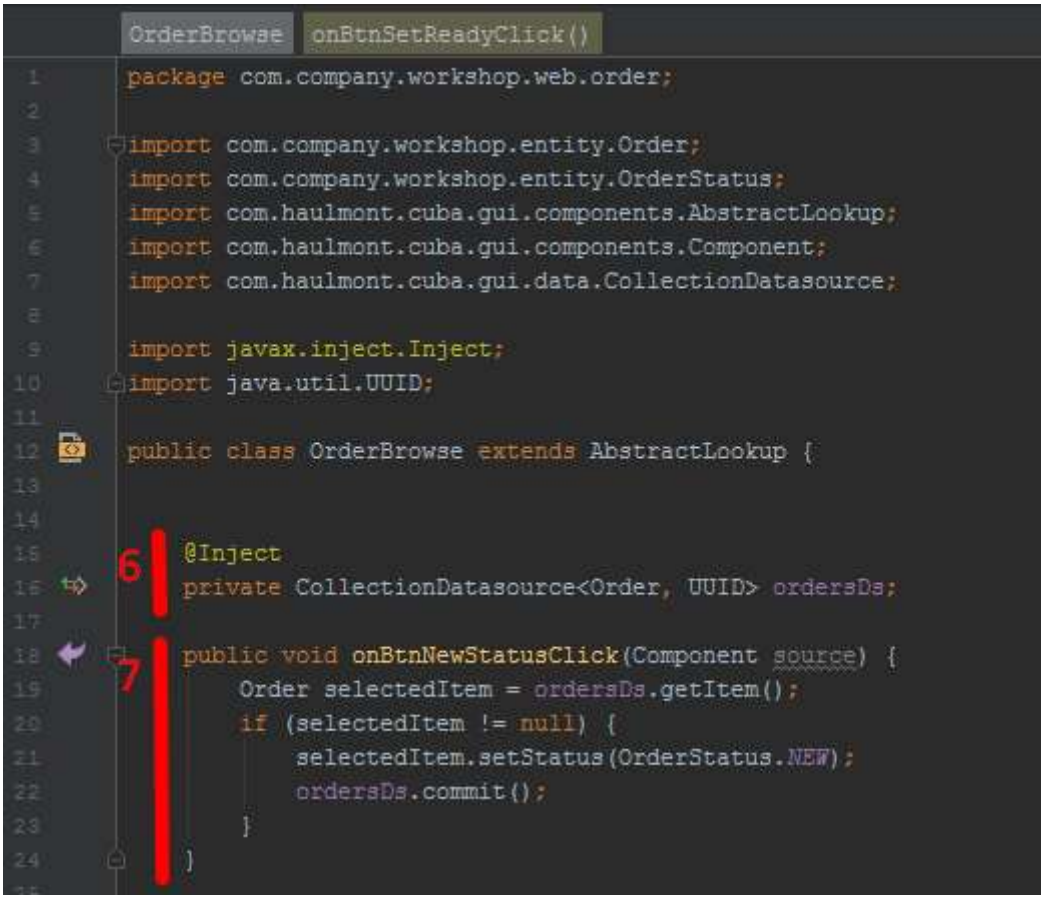
Our mechanics are yelling that it's too annoying to go to the Order editor every time they want to change the order status. They would like to click a button and set the corresponding status from the browser.

1. Open **order-browse.xml** screen in the Studio
2. Add a new button to the button panel (**drag and drop** it into the hierarchy of components)
3. Set the following properties for the button
   id: ***btnNewStatus***
   caption: ***Set as New***
4. Click the **[>>]** button on the right side of the invoke field



5. The Studio will generate a method, that will be called on button click. Press **Ctrl+I** to save the changes and open the screen controller in your IDE
6. CUBA extends the standard spring injection mechanism with ability to inject CUBA related infrastructure and UI components. We will need to inject the datasource from our screen:
   ```
   @Inject
   private CollectionDatasource<Order, UUID> ordersDs;
   ```
7. Let's implement the ***onBtnNewStatusClick*** method, that was created by the Studio:

```
public void onBtnNewStatusClick(Component source) {
  Order selectedItem = ordersDs.getItem();
  if (selectedItem != null) {
    selectedItem.setStatus(OrderStatus.NEW);
      ordersDs.commit();
  }
}
```



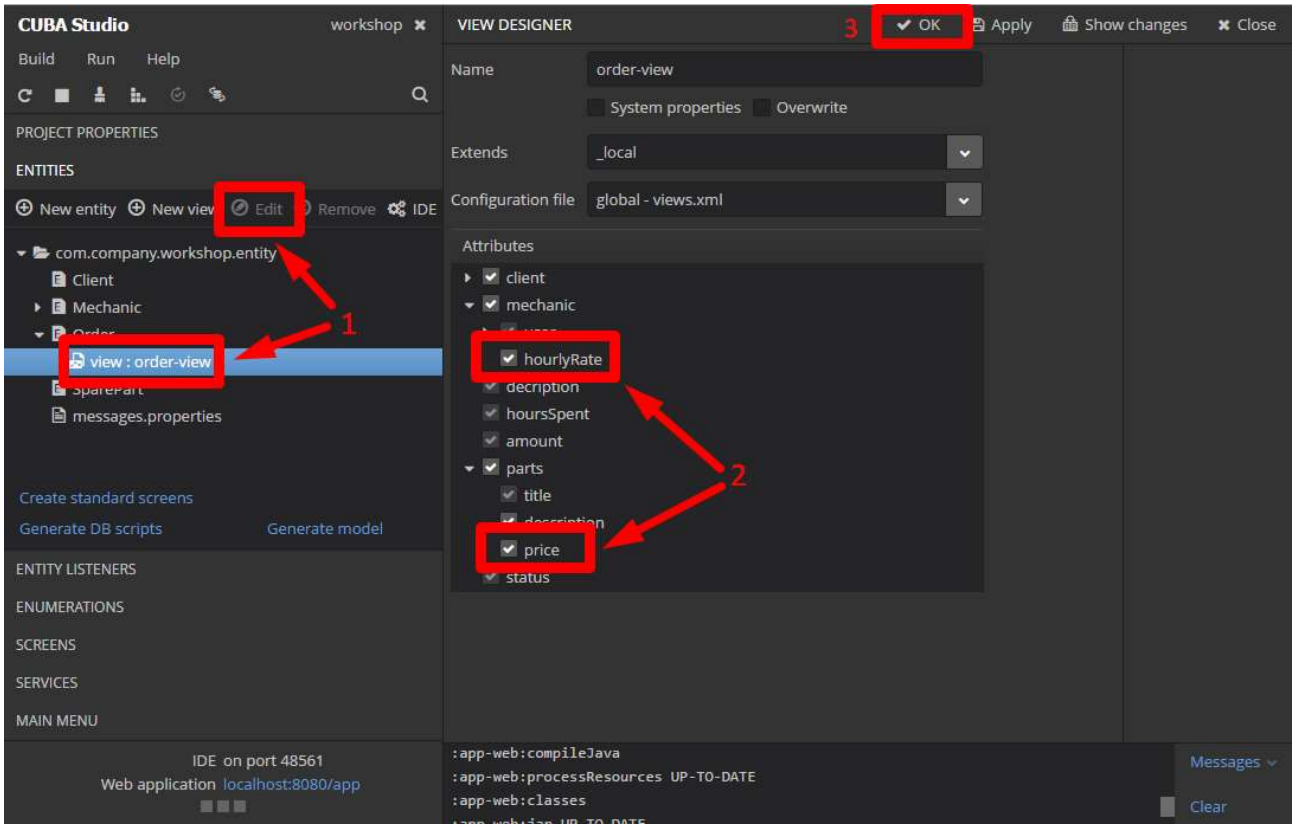
8. **Open/Reopen** the Orders screen to see the mystery of hotdeploy. Now our mechanics can work with maximum productivity

6.3 Business Logic Development

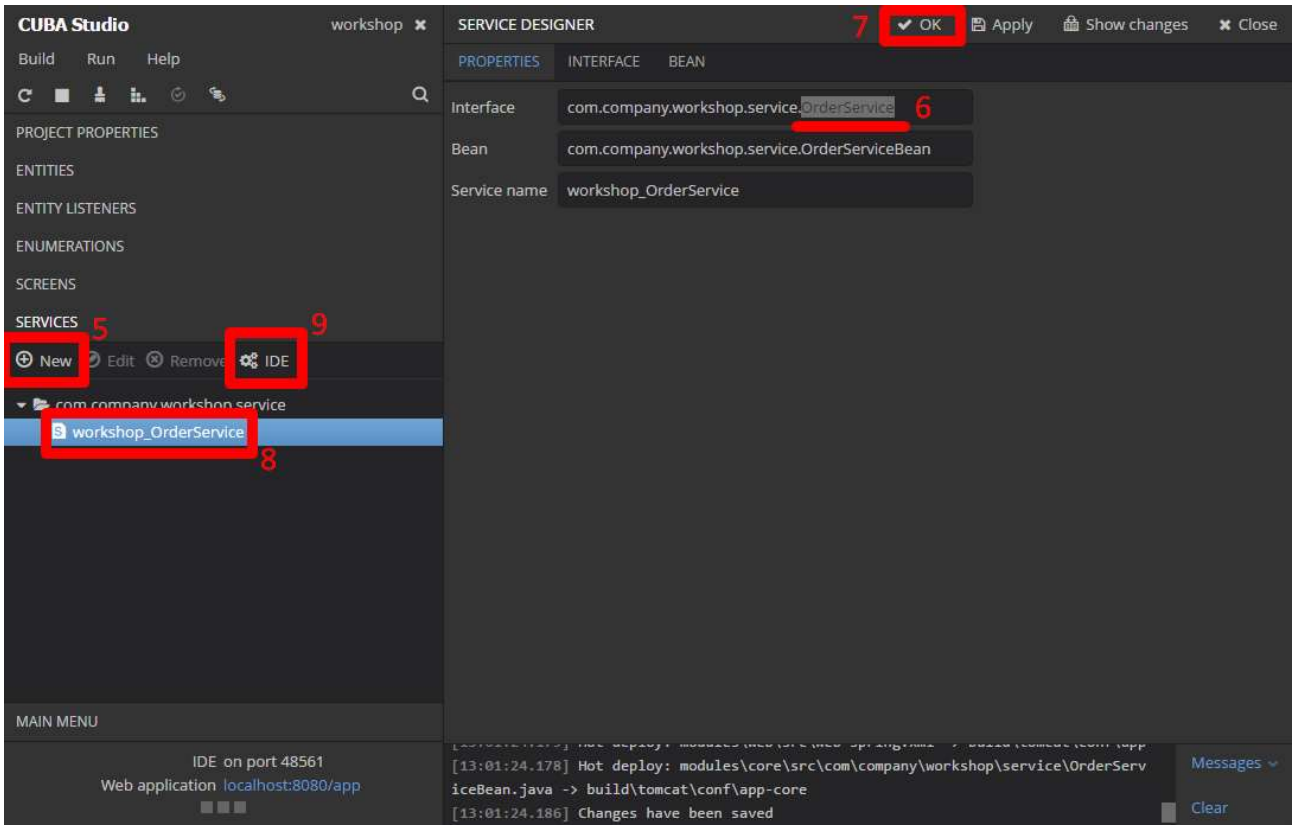
As the next step, let's add business logic to our system to calculate the order price when we save it in the edit screen. The amount will be based on the spare parts price and time spent by the mechanic.

1. To use mechanic hourly rate and prices for parts, we'll need to load this attribute, so we need to add it to **order-view**. In order to change the view switch to the Studio, open the Entities section of the Studio navigation panel, select the **order-view** item and click **Edit**
2. Include the **hourlyRate** and **price** for parts attributes to the view

3. Click **OK** to save the view. From now we will have access to the *hourlyRate* and *price* attributes from the orders screens (editor and browser)



4. Go to the **Services** section in the Studio
5. Click **New**
6. Change the last part of Interface name to ***OrderService***
7. Click **OK**. The interface will be located in the global module, its implementation - in the core module. The service will be available for invocation for all clients that are connected to the middle tier of our application (web-client, portal, mobile clients or integration with third-party applications)
8. Select the **OrderService** item in the navigation panel
9. Click **IDE**



10. In the Intellij IDEA, we'll see the service interface, let's add the amount calculation method to it:
BigDecimal calculateAmount(Order order)

```
package com.company.workshop.service;


import com.company.workshop.entity.Order;

import java.math.BigDecimal;

public interface OrderService {
    String NAME = "workshop_OrderService";

    BigDecimal calculateAmount(Order order);
}
```

11. Go to **OrderServiceBean** using the green navigation icon at the left
12. Implement the *caclulateAmount* method

```
@Service(OrderService.NAME)
public class OrderServiceBean implements OrderService {

    @Override
    public BigDecimal calculateAmount(Order order) {
        BigDecimal amount = new BigDecimal(0);
        if (order.getHoursSpent() != null) {
```

```
            amount = amount.add(new BigDecimal(order.getHoursSpent())
                    .multiply(order.getMechanic().getHourlyRate()));
        }
        if (order.getParts() != null) {
            for (SparePart part : order.getParts()) {
                amount = amount.add(part.getPrice());
            }
        }
        return amount;
    }
}
```

OrderServiceBean calculateAmount()

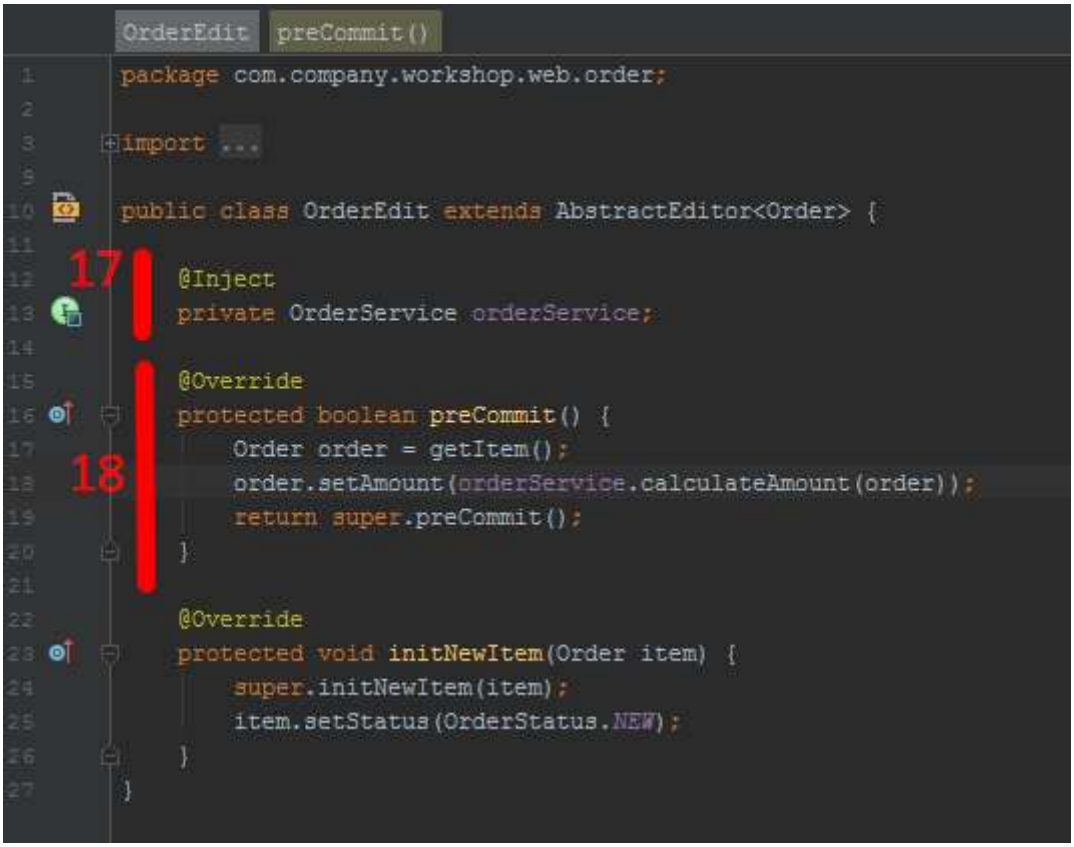
```
package com.company.workshop.service;

import com.company.workshop.entity.Order;
import com.company.workshop.entity.SparePart;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;

@Service(OrderService.NAME)
public class OrderServiceBean implements OrderService {

    @Override
    public BigDecimal calculateAmount(Order order) {
        BigDecimal amount = new BigDecimal(0);
        if (order.getHoursSpent() != null) {
            amount = amount.add(new BigDecimal(order.getHoursSpent())
                    .multiply(order.getMechanic().getHourlyRate()));
        }
        if (order.getParts() != null) {
            for (SparePart part : order.getParts()) {
                amount = amount.add(part.getPrice());
            }
        }
        return amount;
    }
}
```

13. Go back to the Studio
14. Select the **order-edit.xml** screen in the **Screens** section of the navigation panel
15. Click **IDE**
16. Go to the screen controller (**OrderEdit** class)
17. Add **OrderService** field to class and annotate it with @Inject annotation

```
@Inject
private OrderService orderService;
```

18. Override the ***preCommit()*** method and invoke the calculation method of ***OrderService***

```
@Override
protected boolean preCommit() {
    Order order = getItem();
    order.setAmount(orderService.calculateAmount(order));
    return super.preCommit();
}
```



19. Restart your application using the **Run — Restart application** action from the Studio
20. Open **Application — Orders** from the menu
21. Open **editor screen** for any order
22. Set **Hours Spent**
23. Click **OK** to save order
24. We can see a newly calculated value of the amount in the table

Client	Mechanic	Decription	Hours Spent	Amount	Statu
Alex 999-99-99	Jack [jack]	Broken chain	1	10	New
Freeman 111-11-11	Administrator [admin]	Wheels problem Unwanted changes happened. Can we track them?	3	74	New

Now we can automatically calculate price based on spare parts used and time elapsed as it was mentioned in our functional specification!

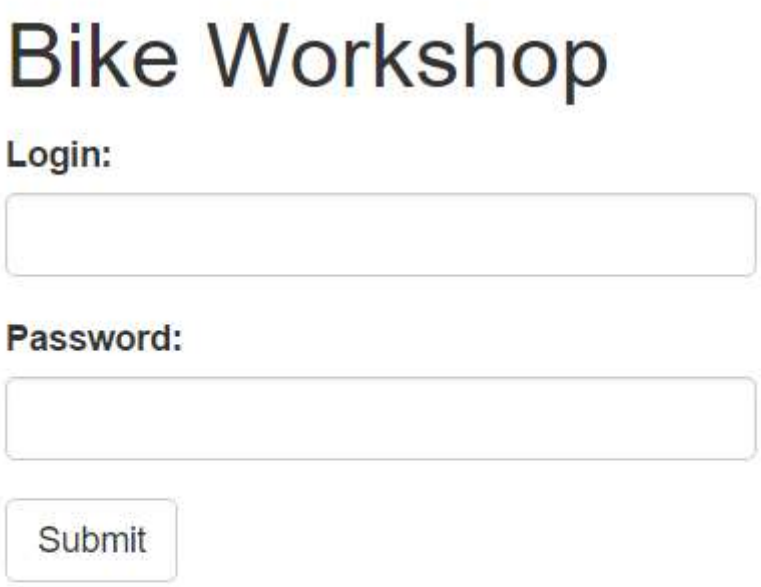
6.4 REST API

Often enterprise systems have more than one client. In our example we have web client for our back office staff. In the future we could come with an idea of having a portal and/or a mobile application for our customers. For this purpose CUBA provides developers with the generic REST API.

REST API is enabled by default in web client, thus we can use it right now for simple web page that will show list of recent orders for a logged in user.

Our page will consist of login form and list of recent orders.

1. Create orders.html in web module: /modules/web/web/VAADIN/orders.html
2. Add jquery-3.1.1.min.js and bootstrap.min.css prerequisites.
3. Let's implement our page:



Our HTML form will consist of 2 fields: login and password. We will send login request to REST API on Submit button click:

```
<html>
<head>
    <script type="text/javascript" src="jquery-3.1.1.min.js"></script>
    <link rel="stylesheet" href="bootstrap.min.css"/>
</head>
<body>
<div style="width: 300px; margin: auto;">
    <h1>Bike Workshop</h1>
```

```
    <div id="loggedInStatus" style="display: none" class="alert alert-success">
        Logged in successfully
    </div>

    <div id="loginForm">
        <div class="form-group">
            <label for="loginField">Login:</label>
            <input type="text" class="form-control" id="loginField">
        </div>
        <div class="form-group">
            <label for="passwordField">Password:</label>
            <input type="password" class="form-control" id="passwordField">
        </div>
        <button type="submit" class="btn btn-default"
onclick="login()">Submit</button>
    </div>
</div>
```

We send login request according to OAuth protocol:

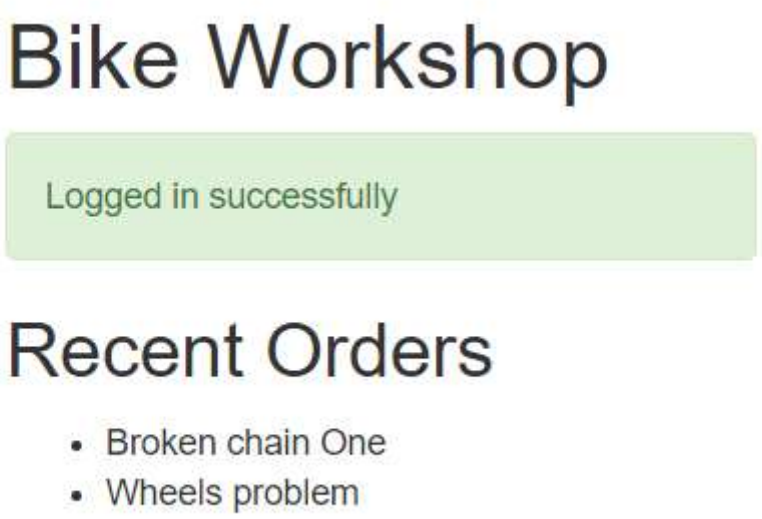
```
<script type="text/javascript">
    var oauthToken = null;

    function login() {
        var userLogin = $('#loginField').val();
        var userPassword = $('#passwordField').val();

        $.post({
            url: 'http://localhost:8080/app/rest/v2/oauth/token',
            headers: {
                'Authorization': 'Basic Y2xpZW50OnNlY3JldA==',
                'Content-Type': 'application/x-www-form-urlencoded'
            },
            dataType: 'json',
            data: {grant_type: 'password', username: userLogin, password:
userPassword},
            success: function (data) {
                oauthToken = data.access_token;

                $('#loggedInStatus').show();
                $('#loginForm').hide();

                loadRecentOrders();
            }
        })
    }
```

If we sucessfully logged in then we load list of recent orders:

```
function loadRecentOrders() {
```

```
    $.get({
        url:
'http://localhost:8080/app/rest/v2/entities/workshop$Order?view=_local',
        headers: {
            'Authorization': 'Bearer ' + oauthToken,
            'Content-Type': 'application/x-www-form-urlencoded'
        },
        success: function (data) {
            $('#recentOrders').show();

            $.each(data, function (i, order) {
                $('#ordersList').append("<li>" + order.description + "</li>");
            });
        }
    });
}
```

And if we try to log in using default login: admin and password: admin we will see our orders list:



That was the last requirement from the functional specification! Now the system is ready for production! For sure, this is very small application and simple, but can be applied for a real local workshop. You can run it in production environment (including clouds) as is and it will be suitable for its purpose.

You can add much more functionality using CUBA additional modules, and this enables you to grow your application to a big strong solution.

Questions

The main goal of our webinar is reached, the simple app is ready for production. If have time now for a debriefing, please feel free to ask any questions about CUBA platform in the chat. Who's going to lead off?

Conclusion

Those who already feel quite familiar with CUBA platform and would like to deepen the knowledge, I invite to take part in our training. It is advanced-level training that covers profound platform issues and development tools. You will learn how to design user interfaces in Studio and XML, how to use CUBA REST API, create platform components, add dynamic attributes and so on. You can find information on the training, as well as on upcoming webinars, on our website https://www.cuba-platform.com.