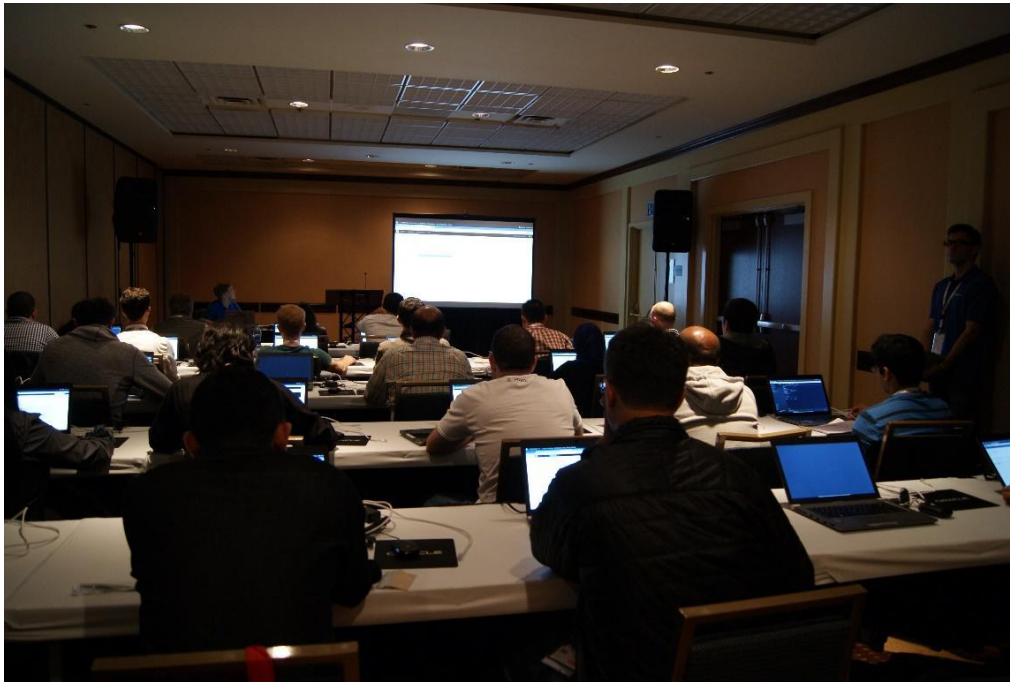
Develop a Fully Functional Business Application in Hours with CUBA Platform

Objectives

This document will guide you through the key features of the CUBA Platform framework and show how you can accelerate the development of enterprise applications in the format of Hands-on-Labs.

Estimated time to complete this lab is 3 hours.

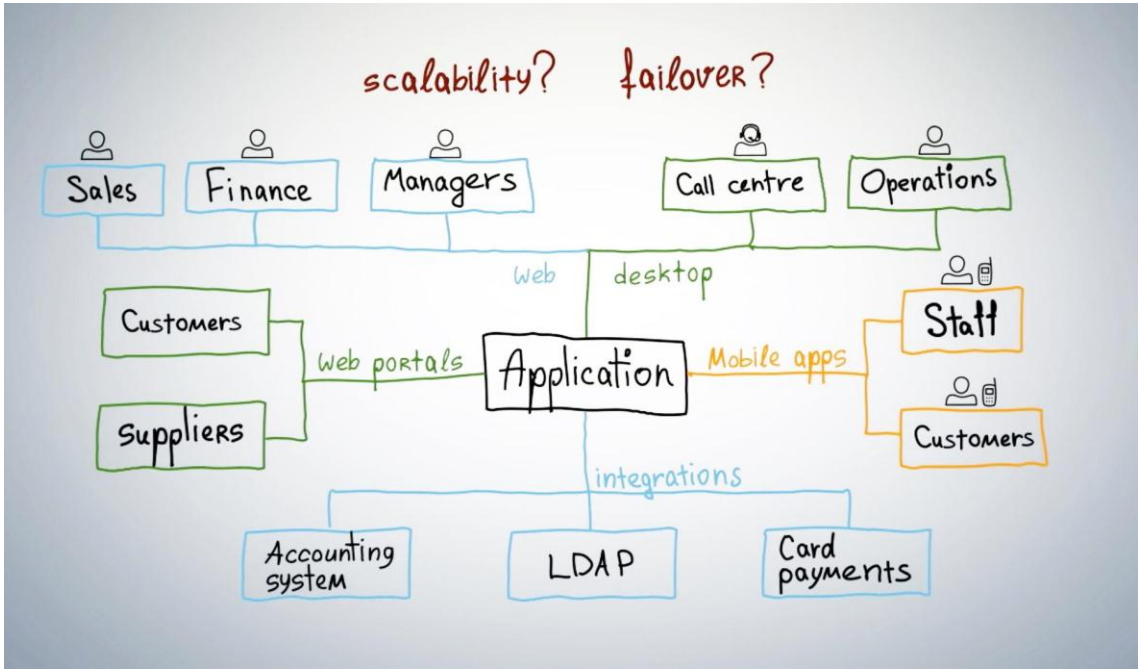
The estimation is given for developers, who have general (basic) knowledge of Java SE.



What is CUBA Platform?

CUBA Platform is a high level Java framework for rapid enterprise software development.
The platform provides a rich set of features:

- Rich web/desktop UI
- CRUD
- Role-based and row-level security
- Reporting
- Charts
- Full-text search
- REST API
- Scalable deployment



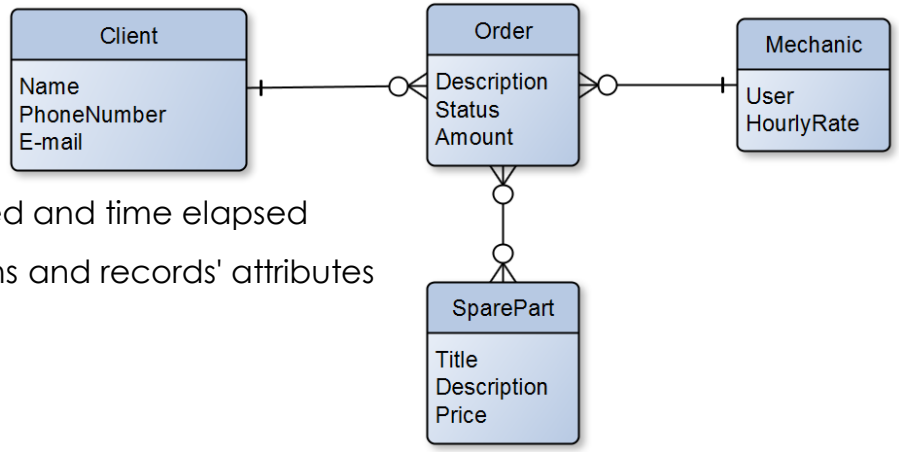
What are we going to automate

Application for a small bicycle workshop

Short functional specification from the application:

- Store customers with their name, mobile phone and email
- Record information about orders: price for repair and time spent by mechanic
- Keep track of spare parts in stock and enable search for parts
- Automatically calculate price based on spare parts used and time elapsed
- Control security permissions for screens, CRUD operations and records' attributes
- Audit of critical data changes
- Charts and reports

The data model



Application features

Our application will:

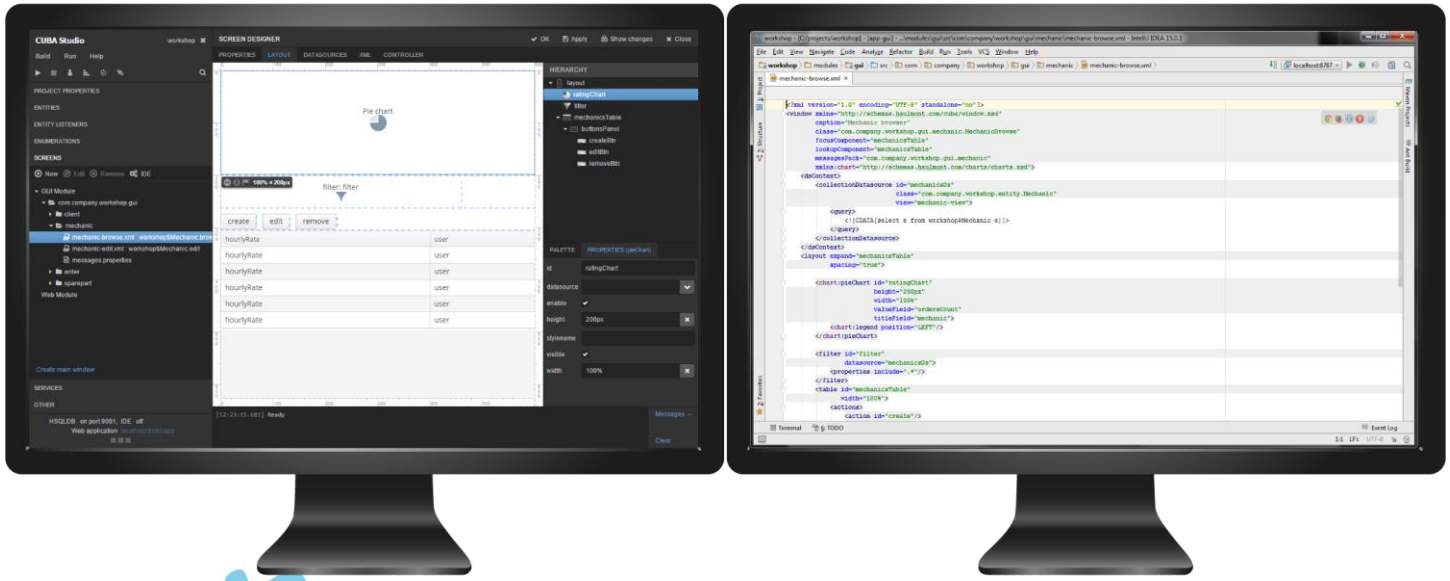
- Have Rich Web UI, with Ajax communication
- Perform basic CRUD operations
- Contain the business logic for calculating prices
- Manage user access rights
- Present data in the form of reports and charts
- Have audit capabilities
- Allow us to create mobile applications or website using REST API

Just three hours,
and we are ready for production!

Environment and tools

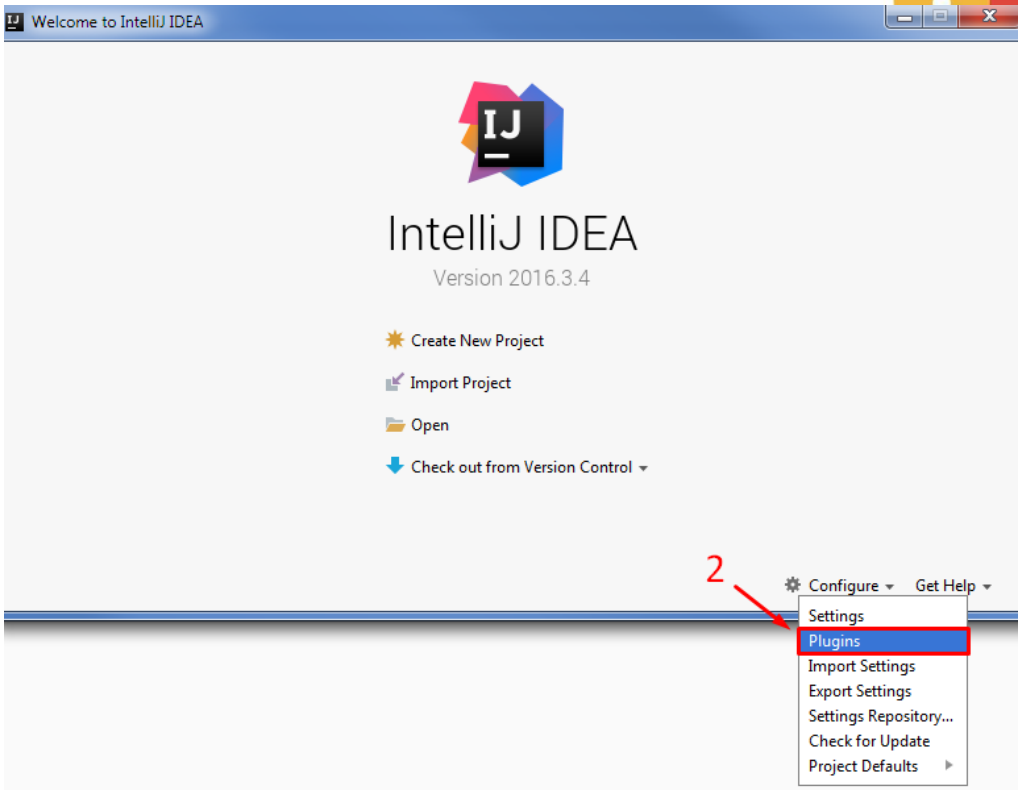
Development environment

1. Download CUBA Studio https://www.cuba-platform.com/download
2. Install IntelliJ IDEA
3. Install CUBA Plugin for IntelliJ IDEA



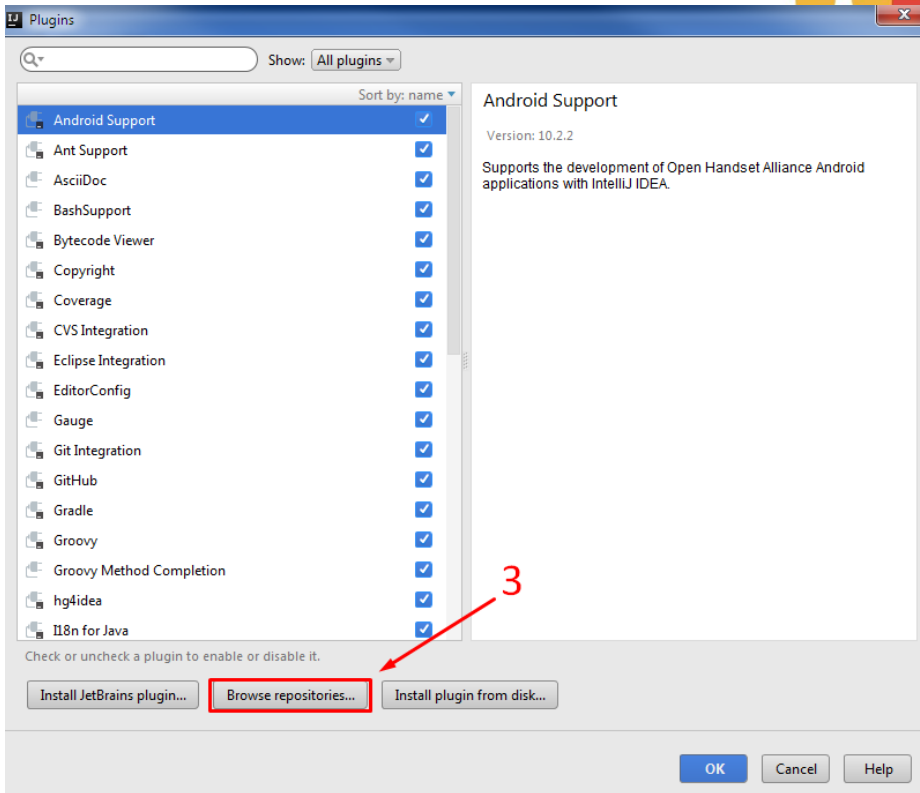
How to install CUBA Plugin for IntelliJ IDEA

1. Run IntelliJ IDEA
2. Open menu Configure - Plugins



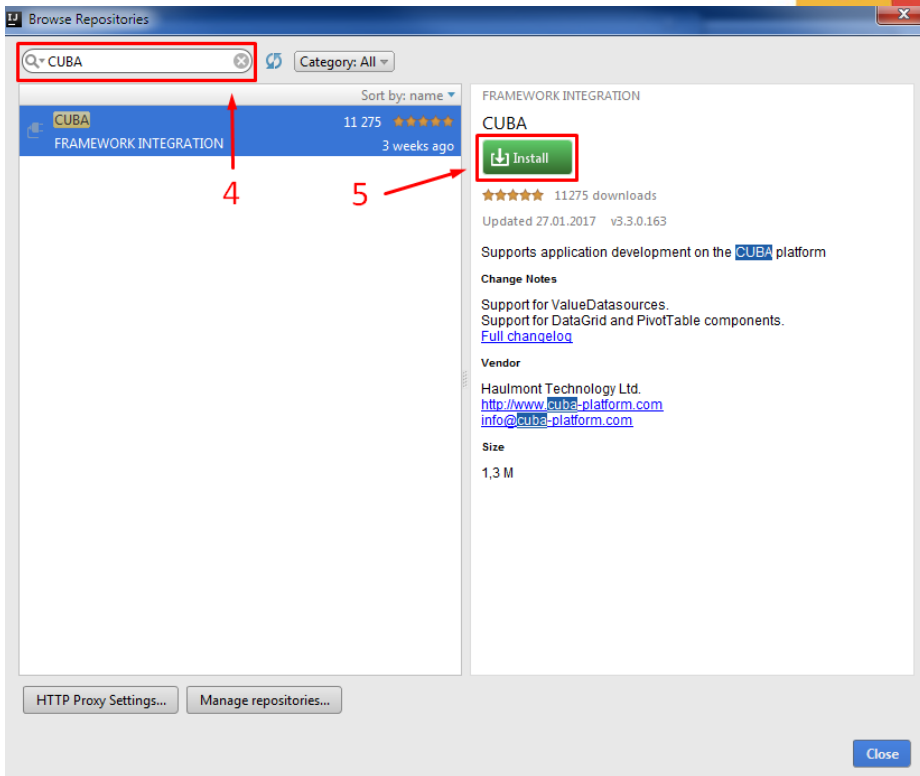
How to install CUBA Plugin for IntelliJ IDEA

3. Click on Browse repositories



How to install CUBA Plugin for IntelliJ IDEA

4. Find CUBA plugin
5. Click Install

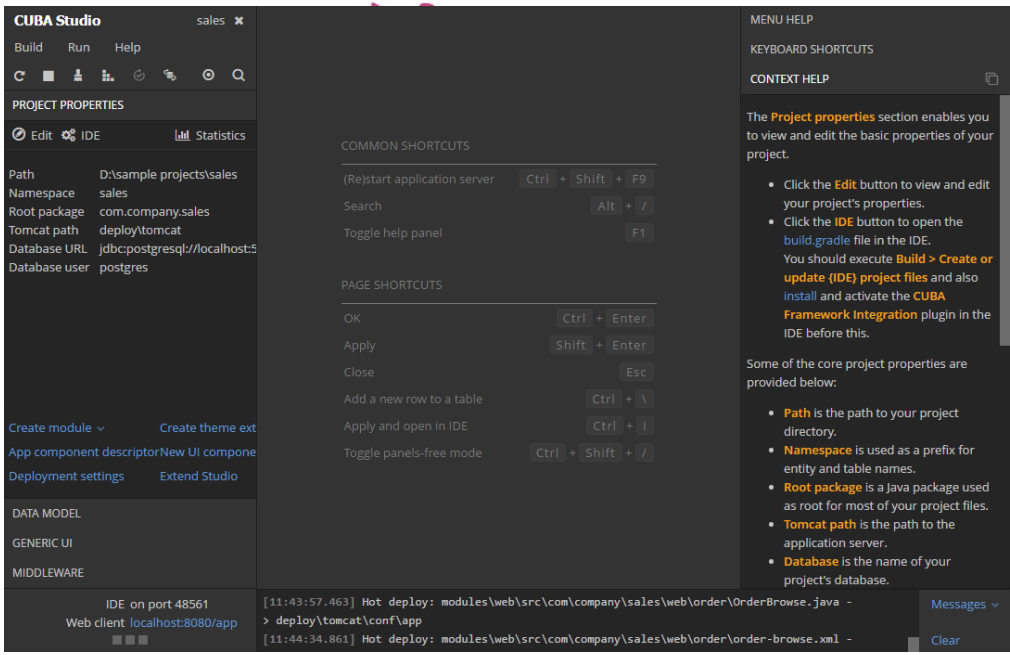


Getting started

What is CUBA Studio?

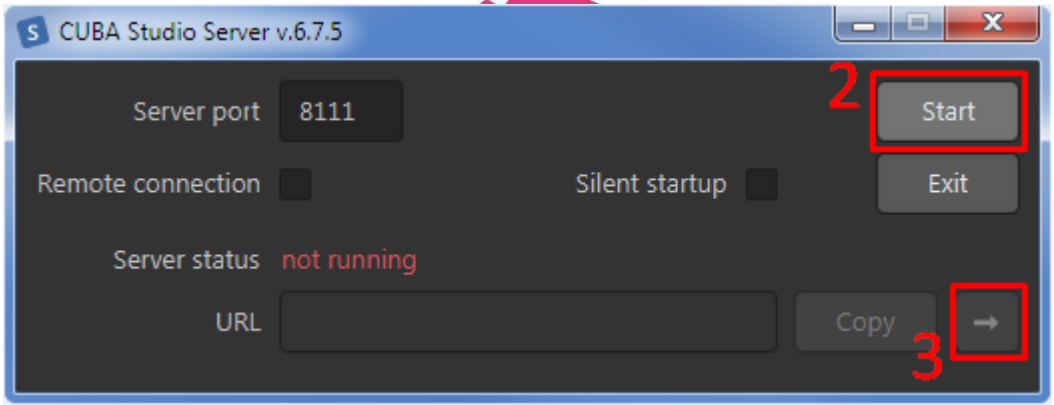
CUBA Studio – a web based development tool that

- Offers a quick way to configure a project and describe data model
- Manages DB scripts
- Enables scaffolding and visual design for the user interface
- Works in parallel with your favorite IDE: IntelliJ IDEA or Eclipse



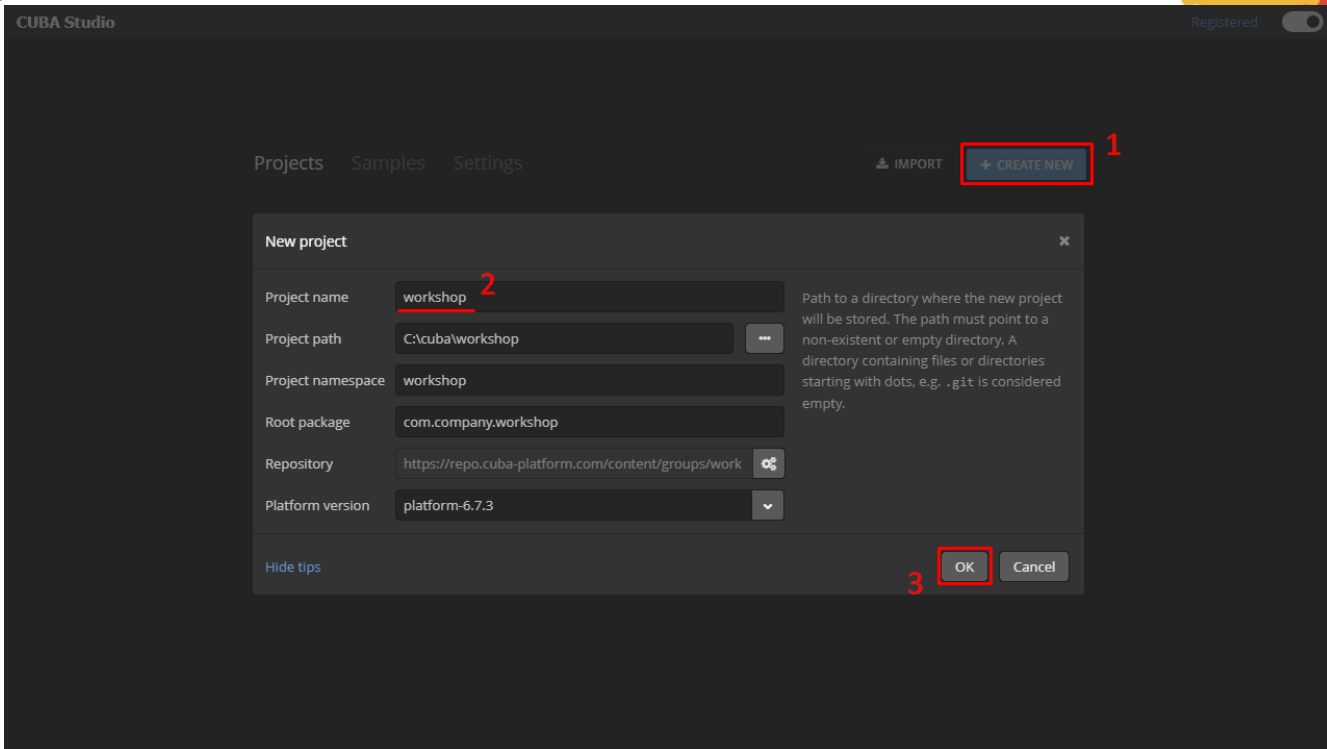
Start CUBA Studio

1. Run CUBA Studio
2. Click **Start** in the launcher window
3. Go to the browser by clicking the Arrow button



Create a new project

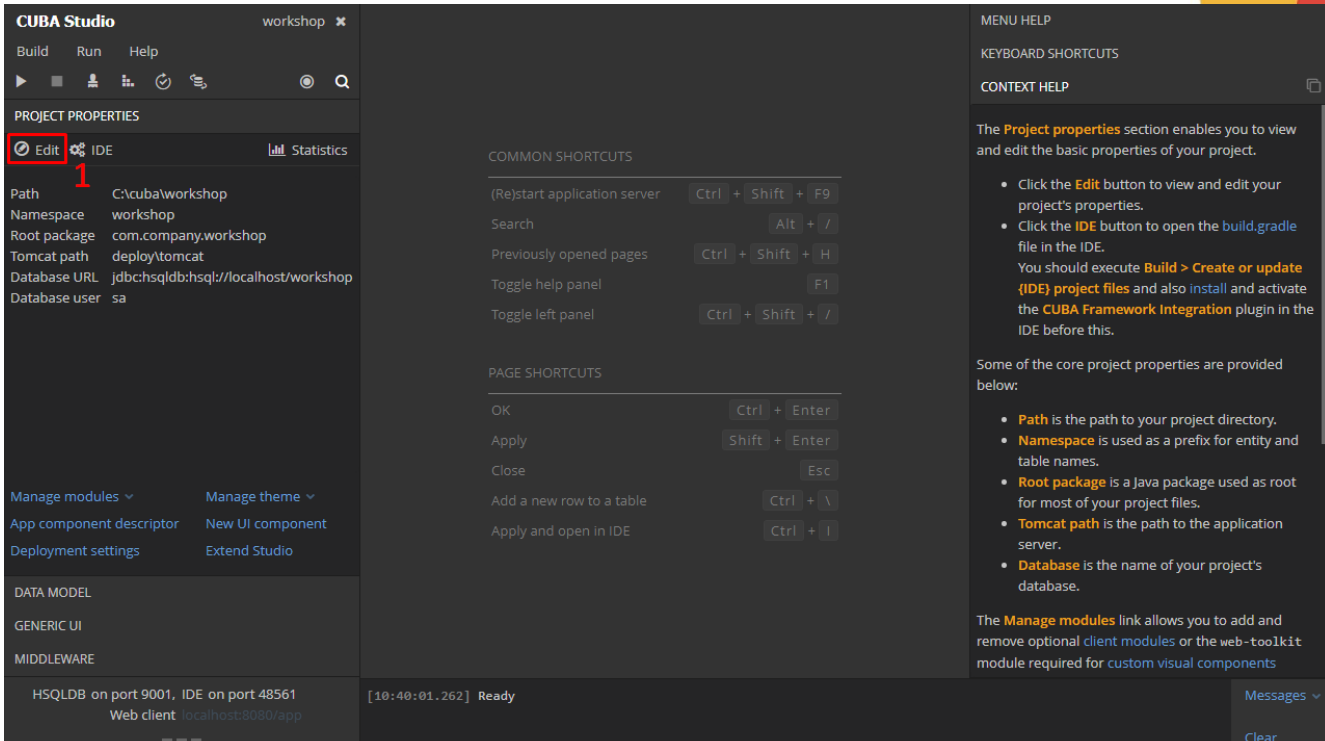
1. Click **Create New** on welcome screen
2. Fill up project name: ***workshop***
3. Click **OK** and you'll get into the CUBA Studio workspace



CUBA Studio workspace

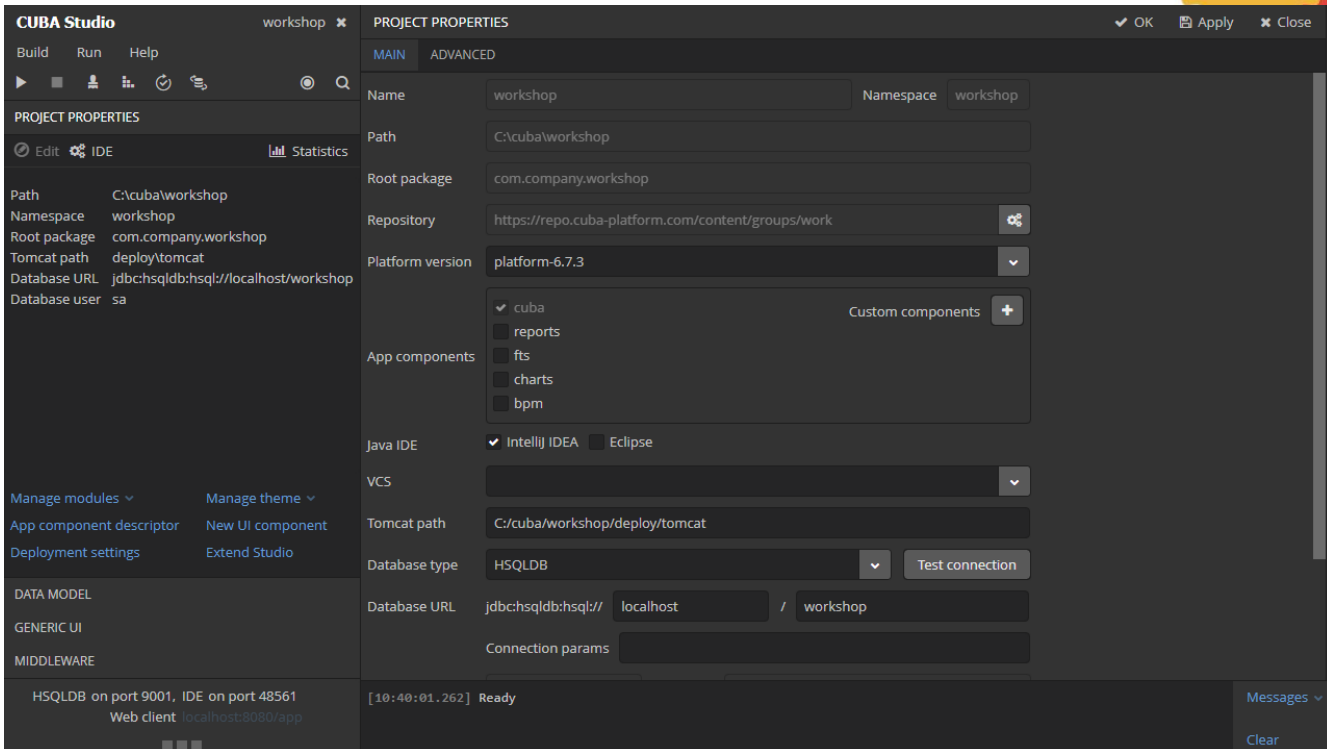
Using CUBA Studio you can easily create **Entities**, **Screens** and stubs for Services.
You can hide/show the **Help** panel using menu **Help - Show help panel**

1. Click **Edit** in the **Project Properties** section



Project properties screen

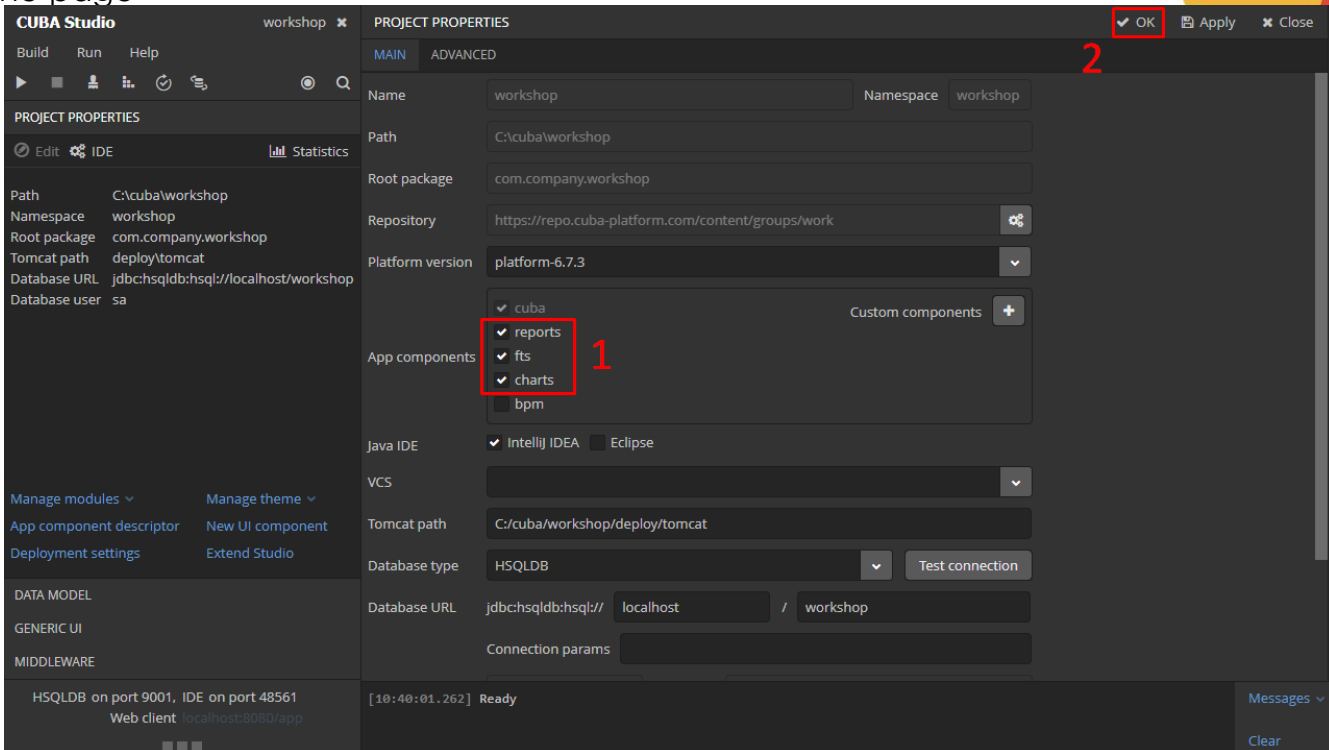
This is a page where we configure our project.
The CUBA Platform supports PostgreSQL, MS SQL, Oracle and HSQL databases.



Use required modules

1. Select the checkboxes for **reports**, **fts** (full-text search) and **charts** in the **App components** section
2. Click **OK** in the upper part of the page
3. Studio will warn us about changing the project build file, just click **OK**.

Studio will automatically add necessary dependencies and regenerate project files for IDE.

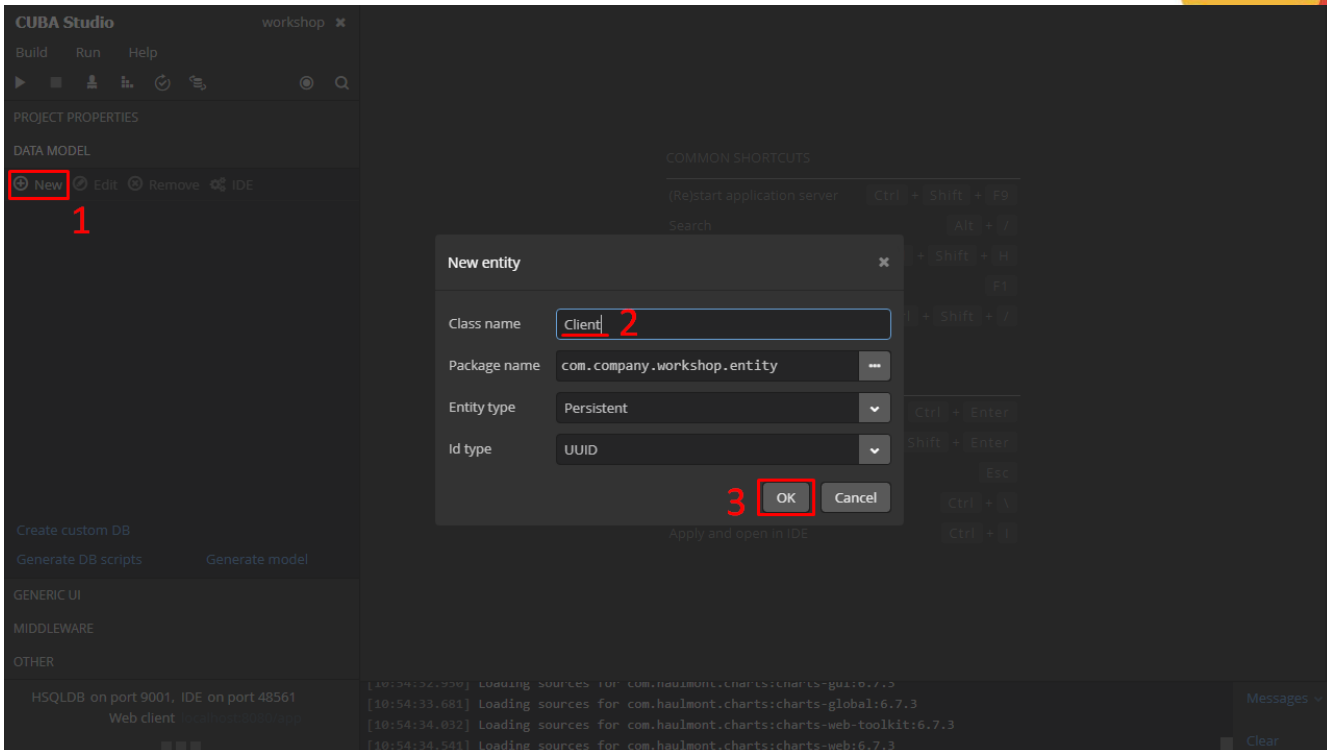


Data model

Create the data model

Open the **DATA MODEL** section of the navigation panel

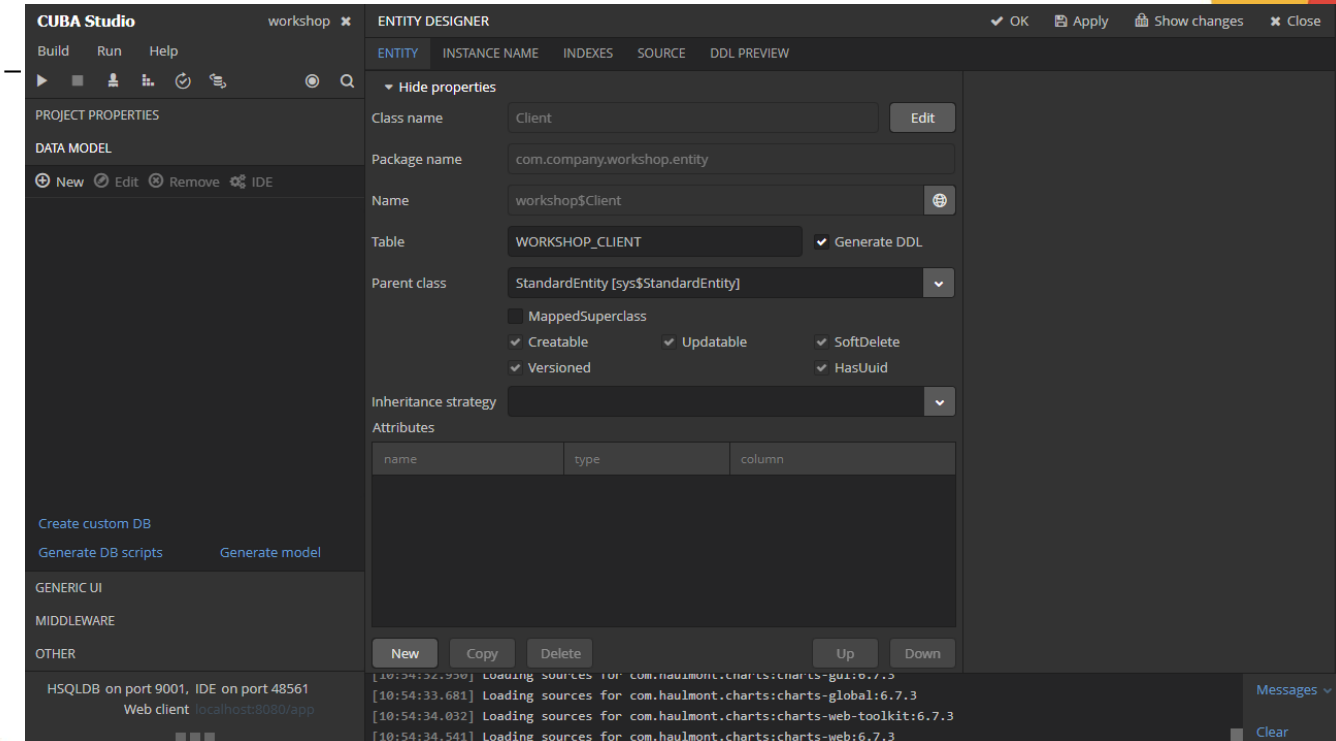
1. Click **New -> Entity**
2. Input **Class name**: ***Client***
3. Click **OK**



Entity designer

Here we can specify a parent class and corresponding table in the database, define attributes for an entity and manage other options.

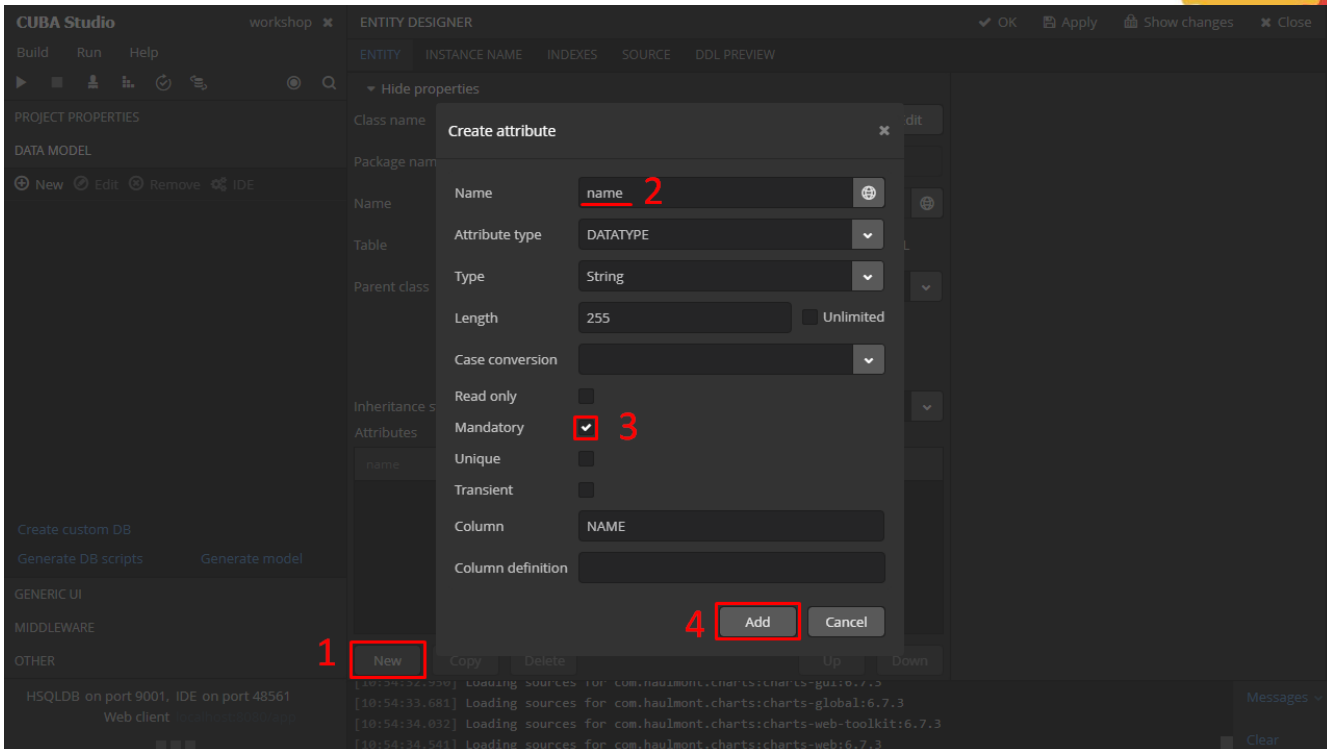
Our class inherits ***StandardEntity*** – the service class which supports **Soft Deletion** and contains a number of platform internal attributes (***createTs***, ***createdBy*** and others).



Attribute editor

1. Add a new attribute by clicking **New**
2. Enter Name: ***name***
3. Select the **Mandatory** checkbox
4. Click on **Add**

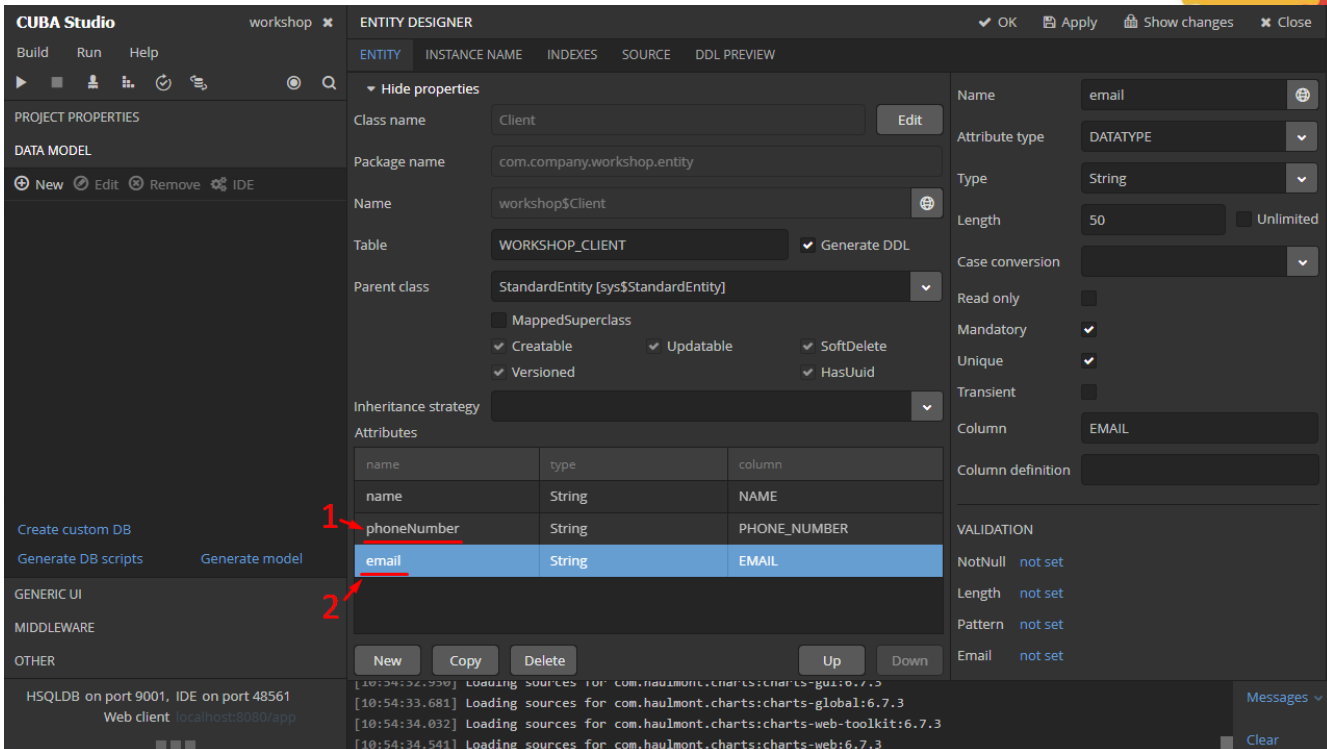
Attribute editor enables us to create or edit attribute and its parameters, such as Attribute type, Java Type, Read only, Mandatory, Unique, etc.



Client entity and its attributes

Similarly, we add ***phoneNumber*** and ***email***.

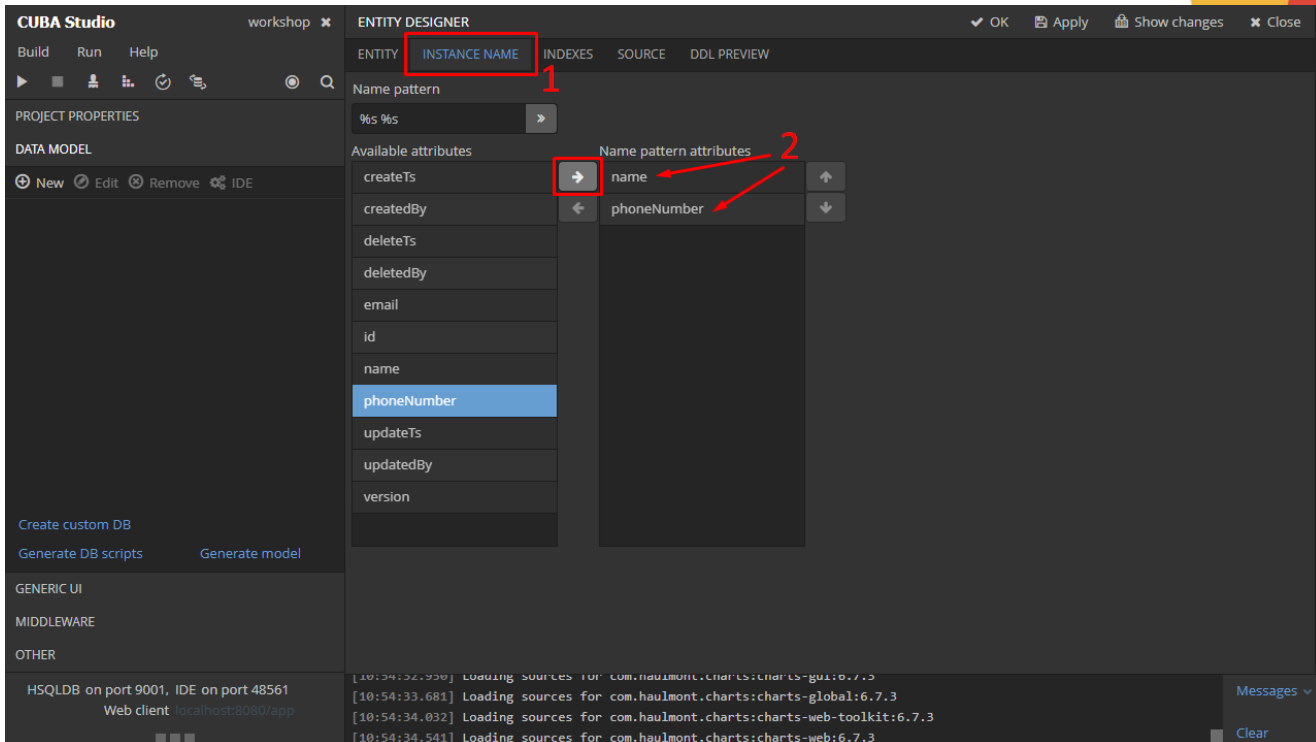
1. Add ***phoneNumber*** as a mandatory attribute with the length of 20 and unique flag
2. Add ***email*** as a mandatory attribute with the length of 50 and flagged as unique



Instance name

Instance name is a default string representation of **Entity** for user interface (tables, dropdown lists, etc).

1. Go to the **Instance name** tab
2. Select ***name*** and ***phoneNumber***

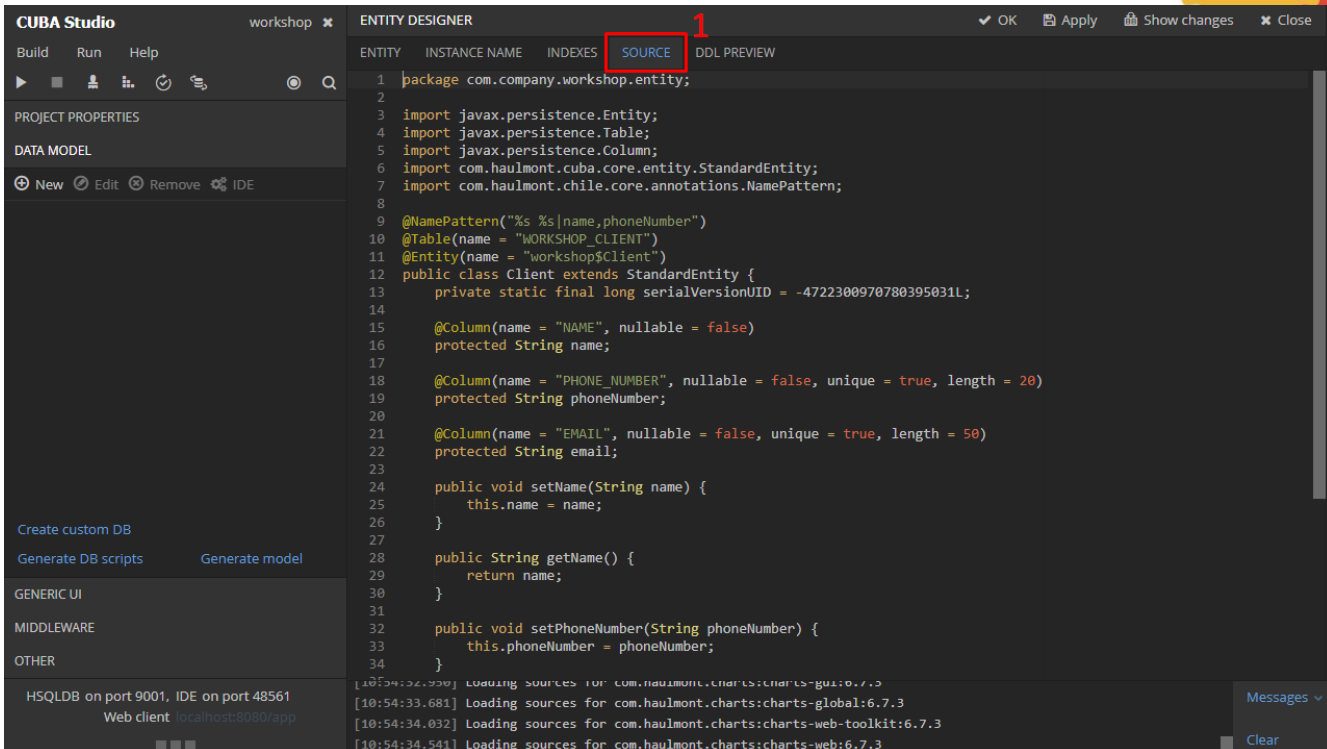


Generated source code for the Client entity

1. Click on the **Source** tab of the **Entity designer**

This is a regular Java class, annotated with the ***javax.persistence*** annotations and supplemented by CUBA annotations.

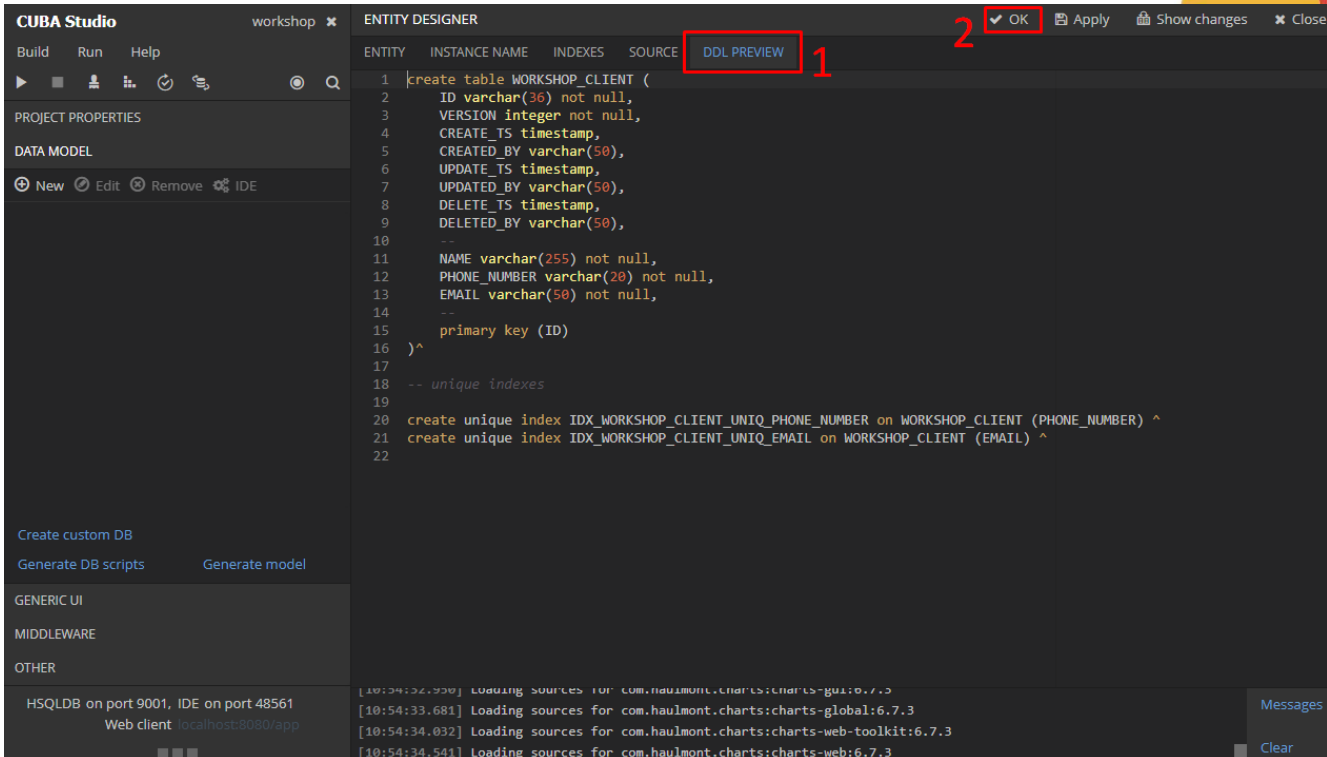
You can change source code of an entity manually, and Studio will read your changes and apply those back to model.



DDL Scripts

1. Click on **DDL Preview** tab of the **Entity designer**
2. Click **OK** to save the ***Client*** entity

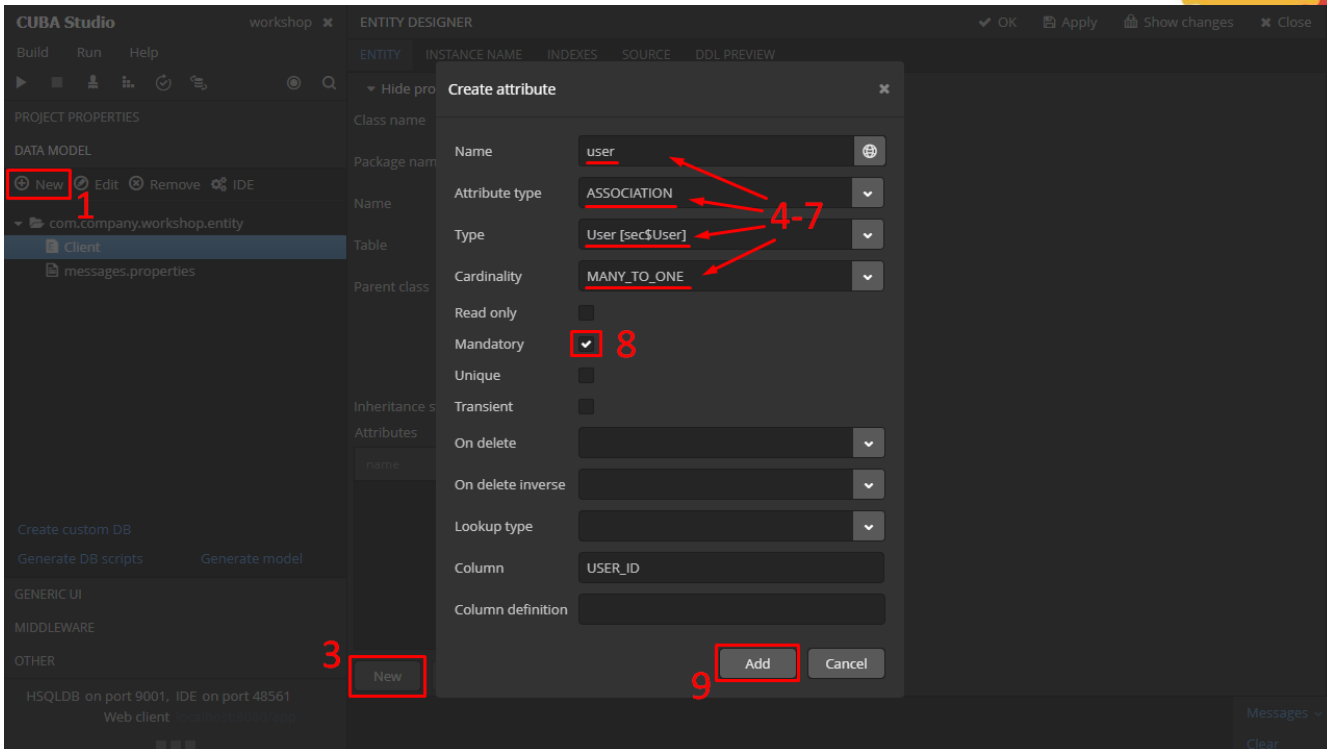
This tab illustrates preview of SQL script for corresponding table creation.



Mechanic entity

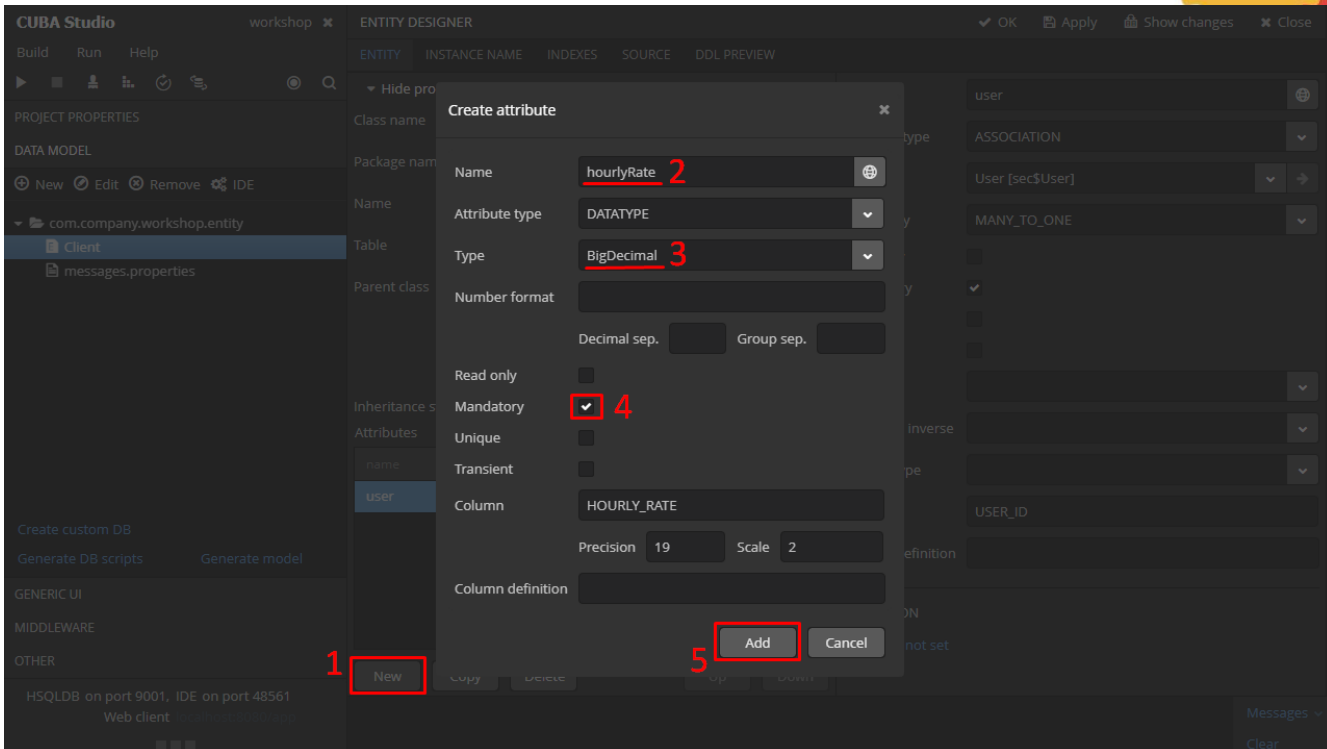
1. Click **New -> Entity**
2. Input ***Mechanic*** as entity name and click **OK**
3. Create new attribute
4. Set attribute name to ***user***
5. Set Attribute type: ***ASSOCIATION***
6. Set Type: ***User [sec$User]***
7. Set Cardinality: ***MANY_TO_ONE***
8. Select Mandatory checkbox
9. Click Add

The ***User*** entity is a standard entity used to operate with system users in the CUBA Platform.



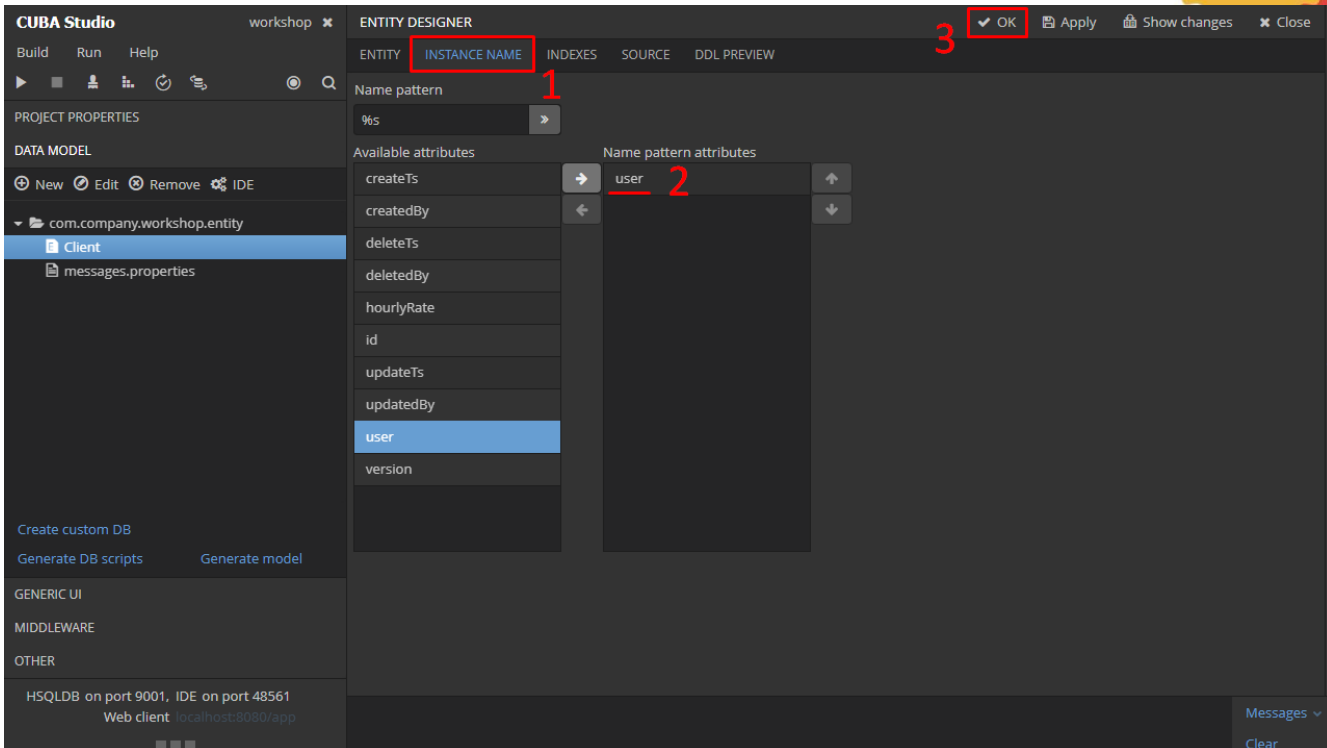
Mechanic entity — hourlyRate attribute

1. Click New to create attribute
2. Set Name: ***hourlyRate***
3. Set Type: ***BigDecimal***
4. Select Mandatory checkbox
5. Click the Add button



Mechanic entity — instance name

1. Go to the **Instance name** tab
2. Select ***user*** for the Mechanic's instance name
3. Save the **Mechanic** entity by clicking **OK**



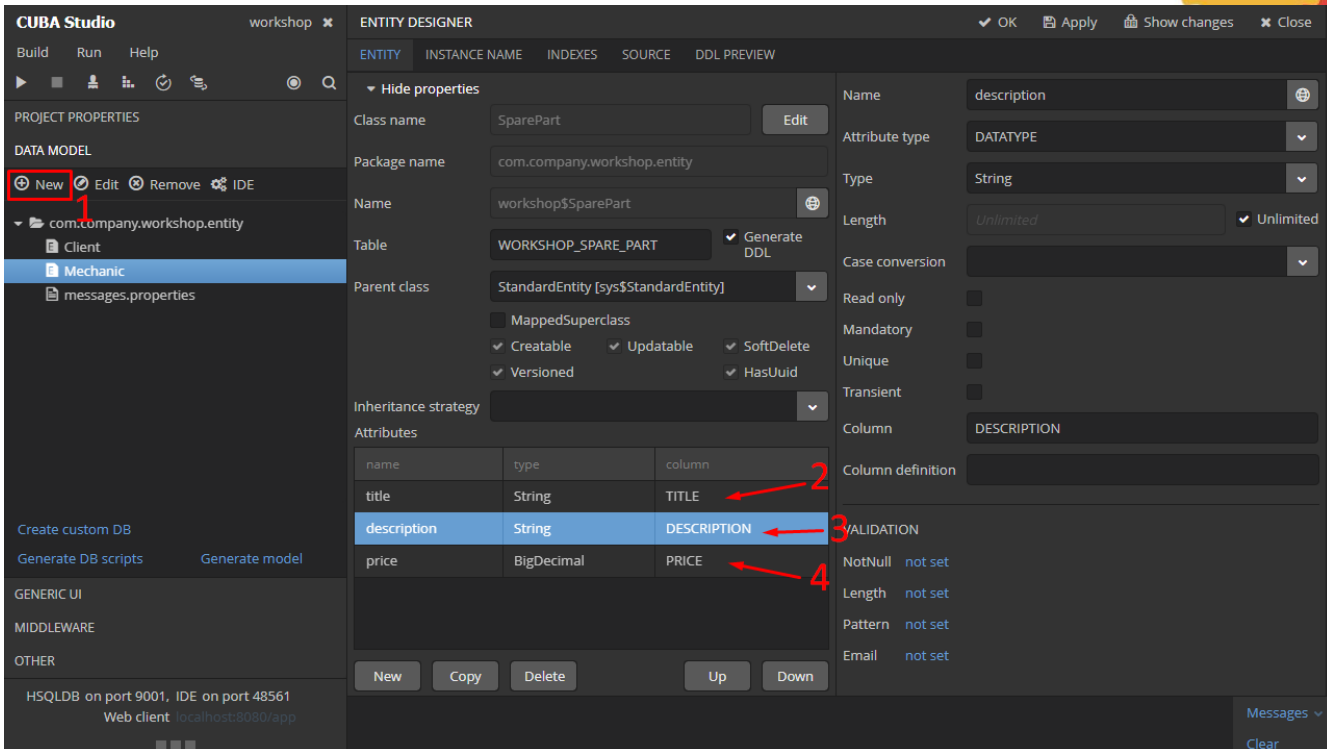
SparePart entity

1. Create new entity with Class name: ***SparePart***

2. Add the ***title*** attribute as a mandatory and unique String

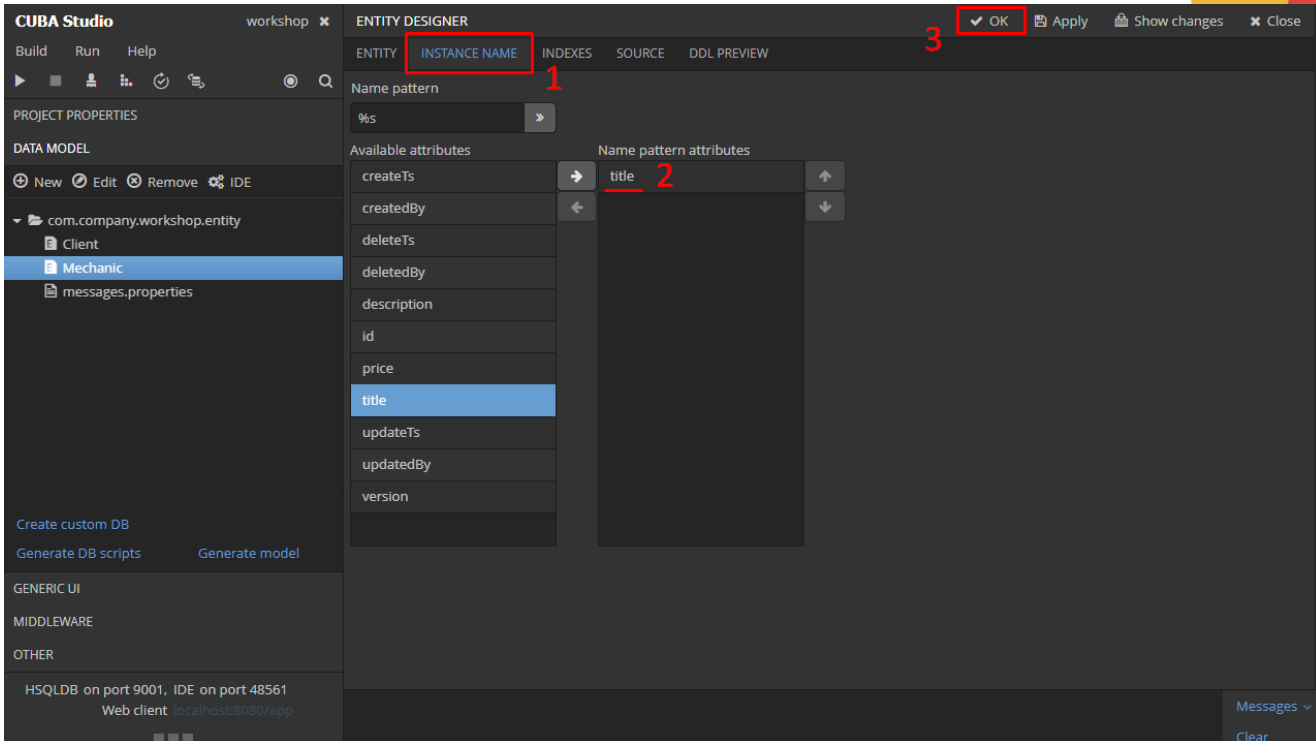
3. Add the ***description*** attribute: String; check the Unlimited checkbox, so ***description*** will have unlimited length

4. Add the ***price*** attribute: mandatory, BigDecimal



SparePart entity — instance name

1. Go to the **Instance name** tab
2. Select the ***title*** attribute for the ***SparePart*** instance name
3. Click **OK** to save the entity



OrderStatus enum

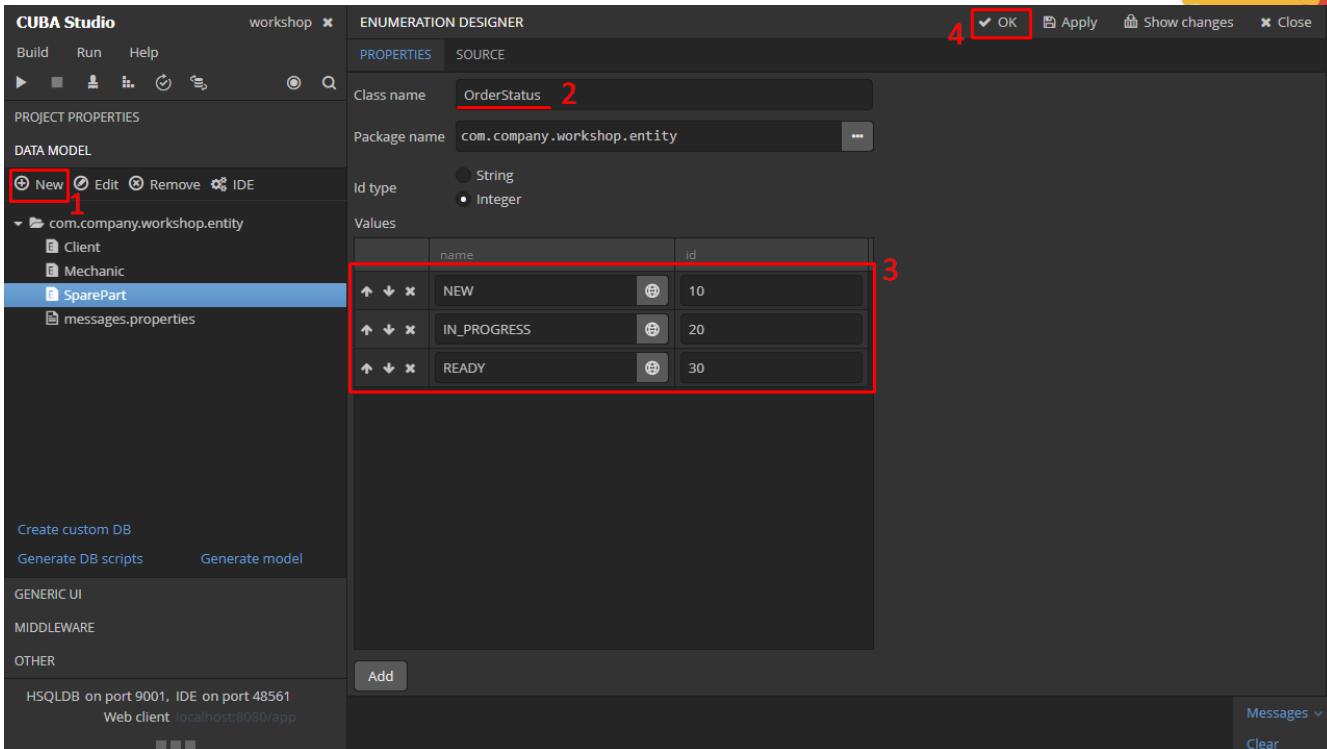
To create the ***Order*** entity we'll need to create the ***OrderStatus*** enum first.

1. Go to the **DATA MODEL** section in the navigation panel and click **New -> Enumeration**

2. Enter **Class Name**: ***OrderStatus***

3. Add values:
NEW 10
IN_PROGRESS 20
READY 30

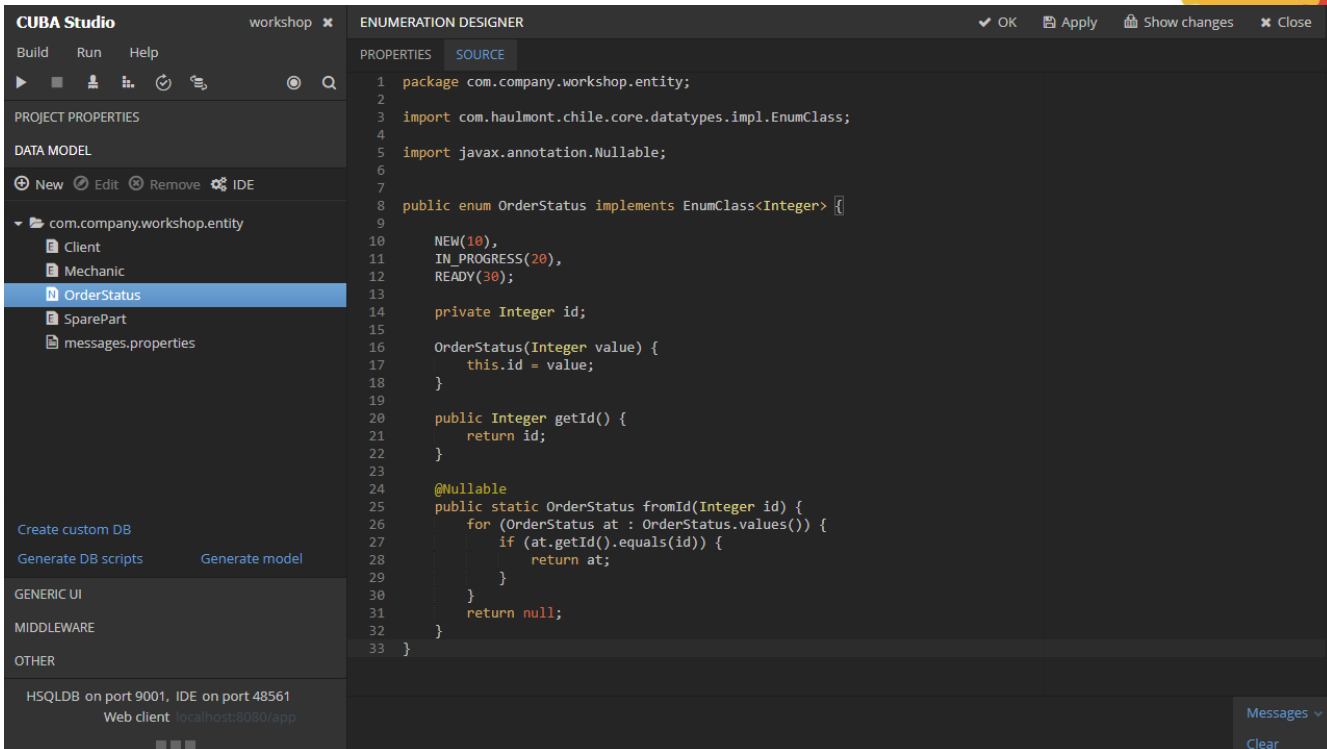
4. Click **OK**



OrderStatus enum — source code

Similar to entities, we can check the generated Java code in the **Source** tab.

You can change source code of enum manually here, and Studio can read it back
From the source to its enum model.

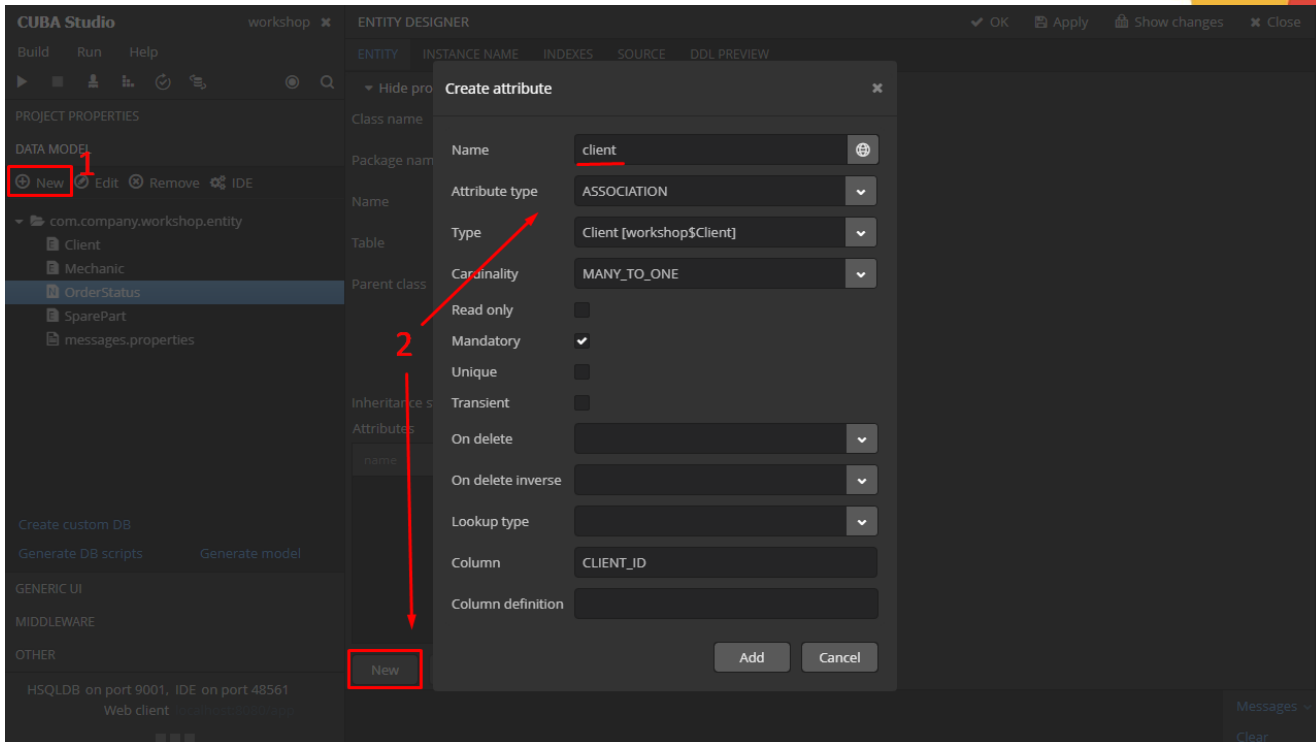


Order entity

1. Go to the **DATA MODEL** section of the navigation panel and click **New -> Entity.**
Set ***Order*** as the **Class name**

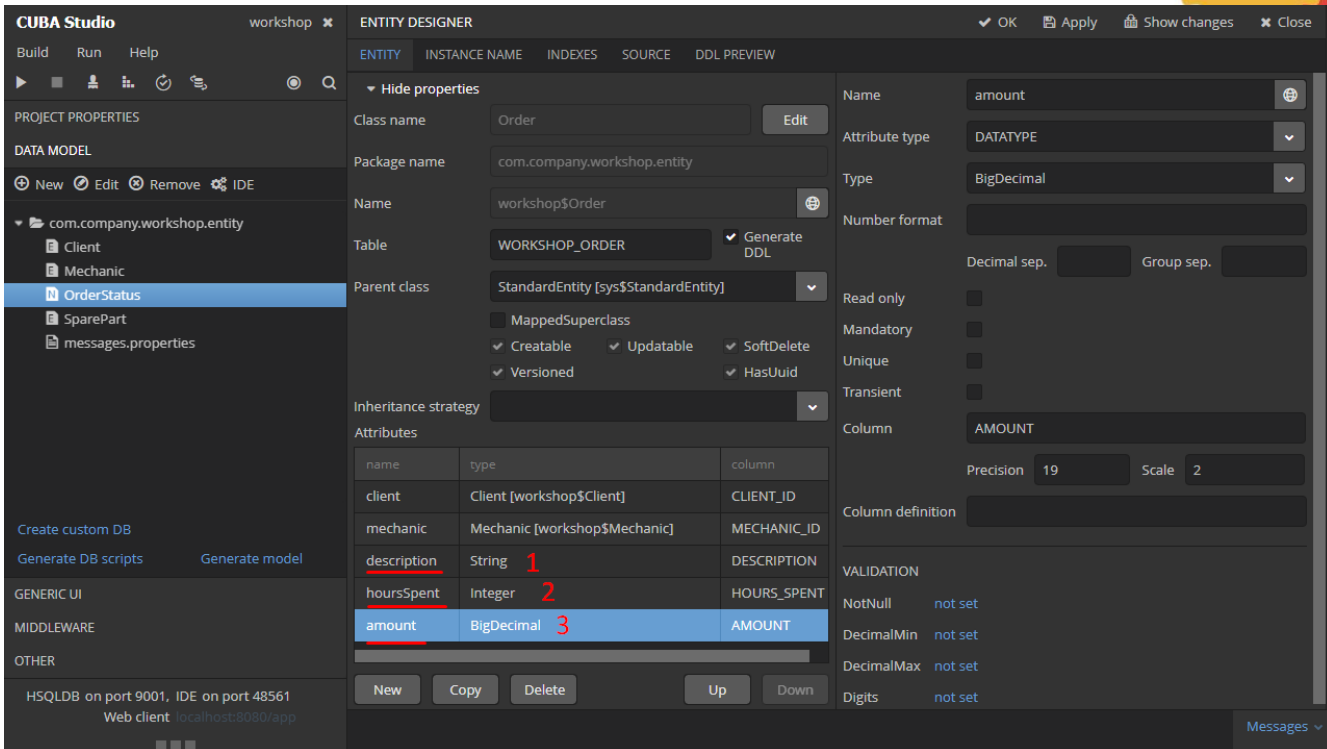
2. Add new attribute named ***client***
Attribute type: ***ASSOCIATION***
Type: ***Client***
Cardinality: ***MANY_TO_ONE***
Mandatory: ***true***

Similarly add the **mechanic** attribute with **Type: *Mechanic***



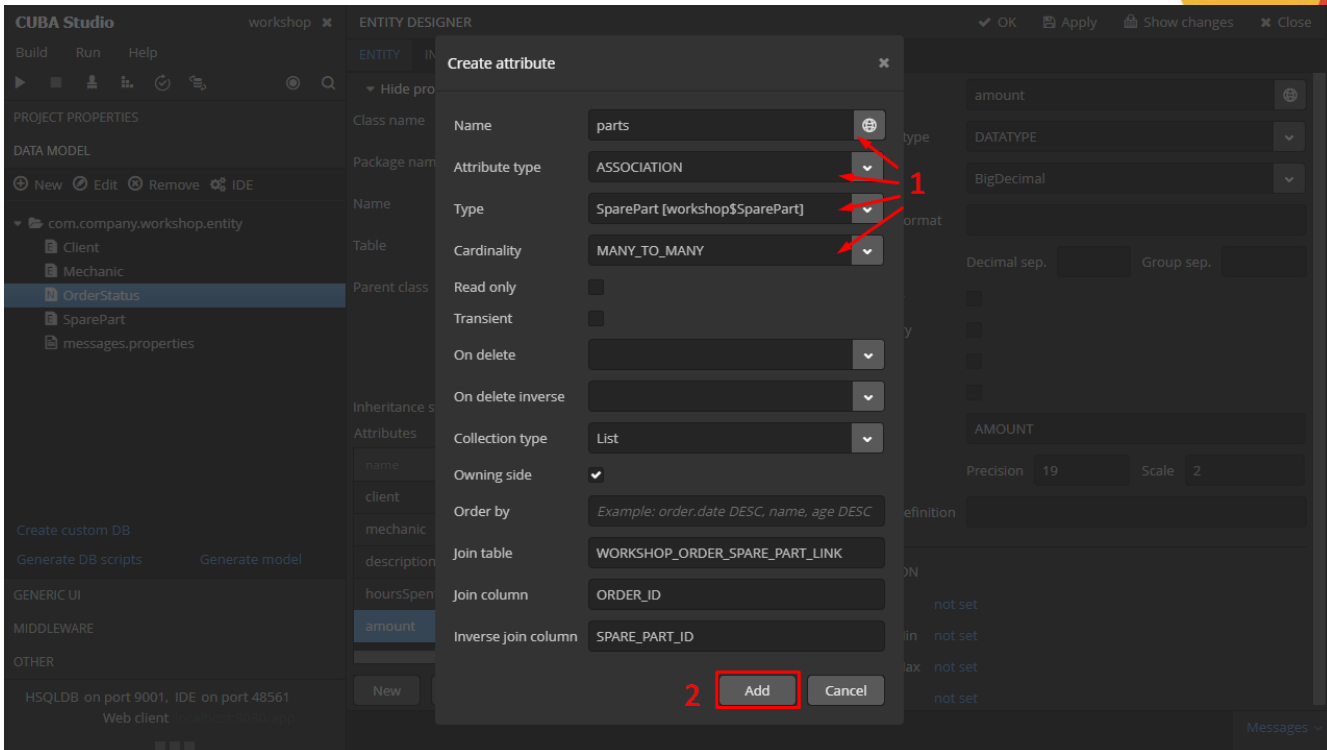
Order entity — description, hoursSpent, amount

1. Add ***description*** attribute: **String**, check the **Unlimited** length checkbox, so ***description*** will have unlimited length
2. Add ***hoursSpent*** attribute: Integer
3. Add ***amount*** attribute: BigDecimal



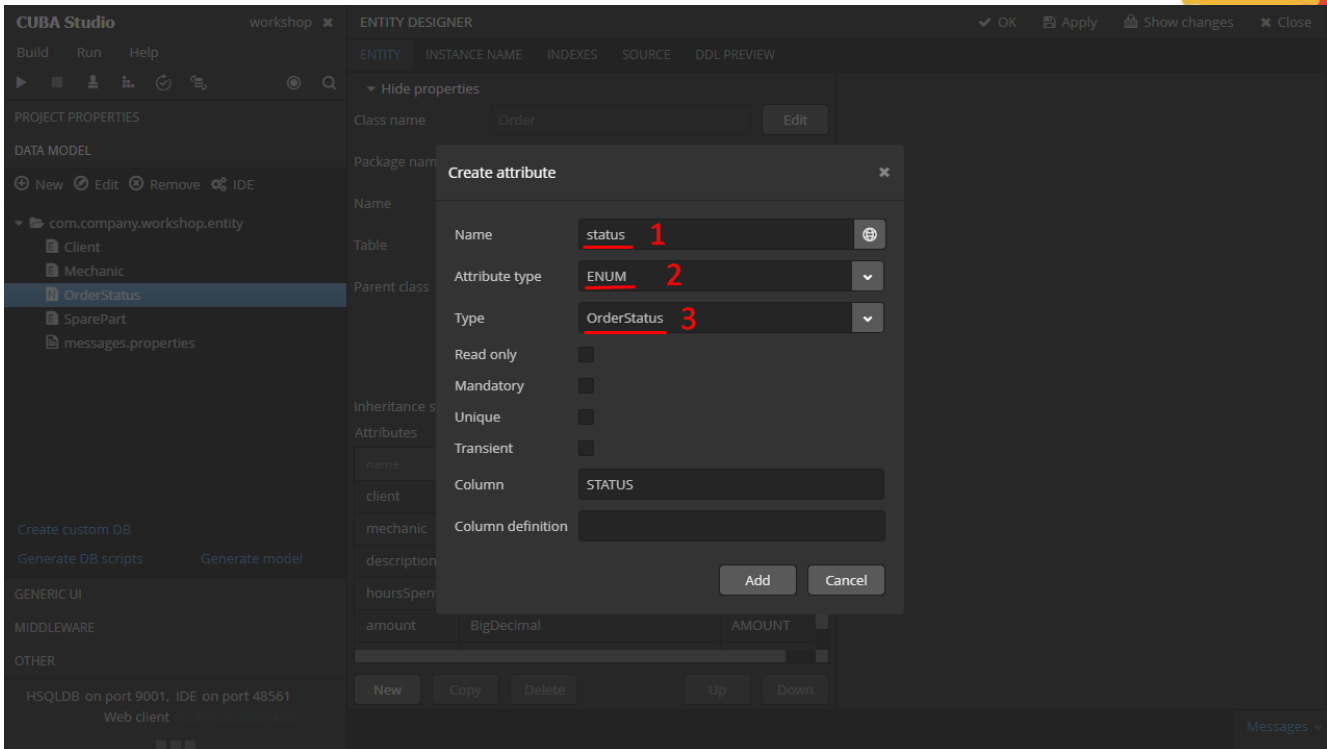
Order entity — parts attribute

1. Create a **New** attribute: ***parts***
Attribute type: ***ASSOCIATION***
Type: ***SparePart***
Cardinality: ***MANY_TO_MANY***
2. Click on the **Add** button
3. Studio will offer to create a reverse attribute from the ***SparePart*** entity to link it to ***Order***, just click **No**



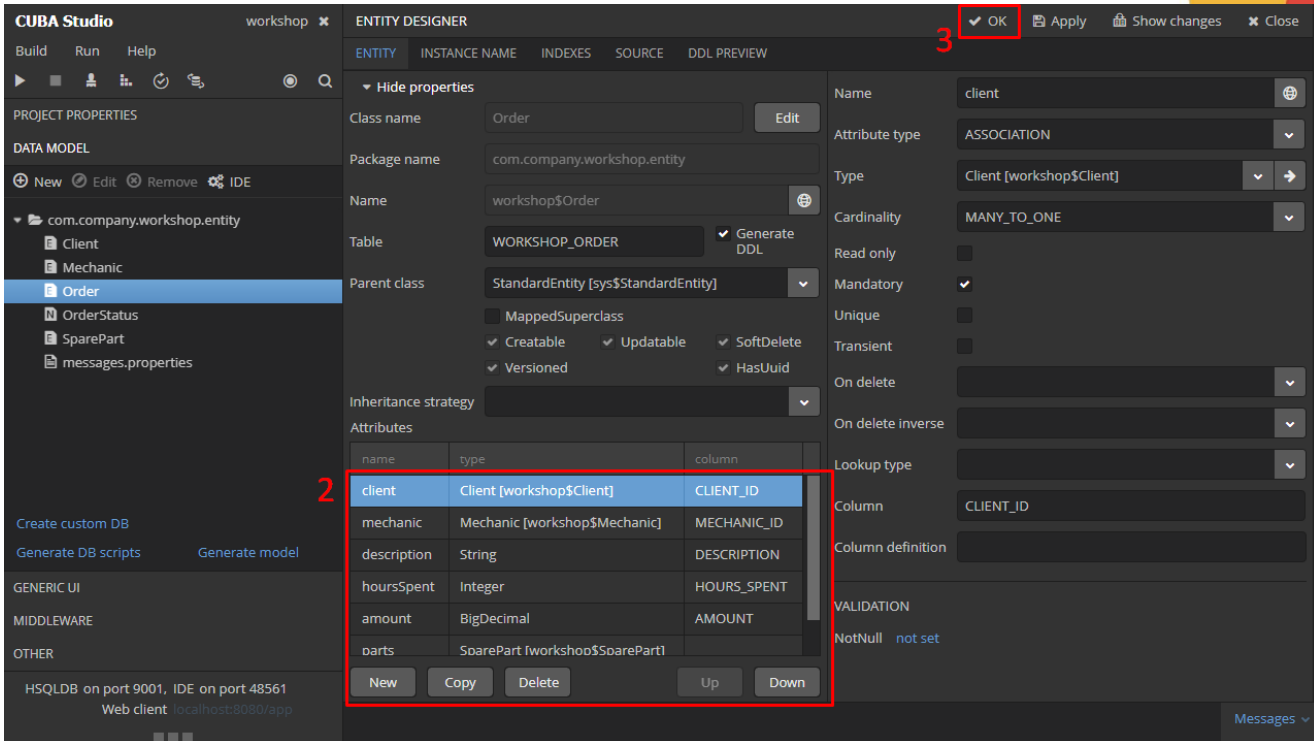
Order entity — status attribute

1. Create **New** attribute: ***status***
2. Set **Attribute type**: ***ENUM***
3. Set **Type**: ***OrderStatus***
4. Click **Add**



Order entity — done

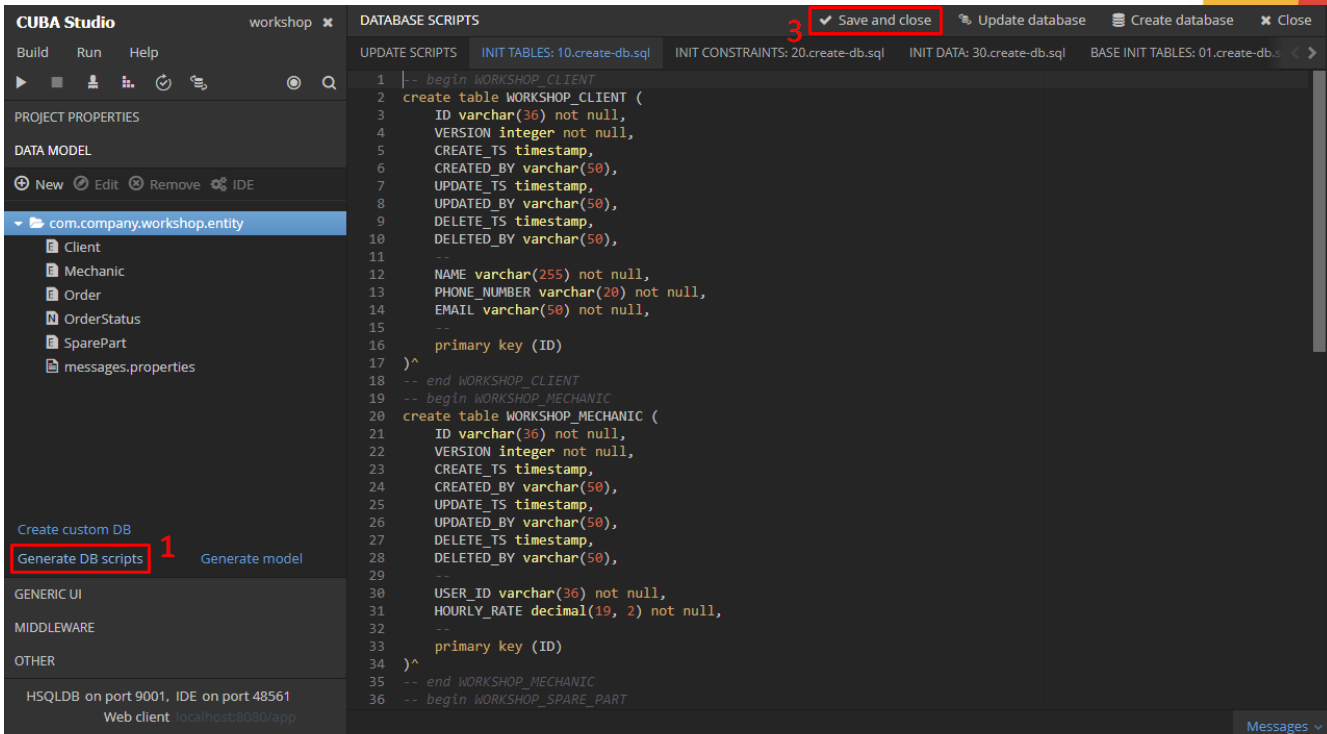
1. Set **Instance name** for the ***Order*** entity to its ***description*** *attribute*
2. Check the attributes list of the ***Order*** entity: ***client***, ***mechanic***, ***description***, ***hoursSpent***, ***amount***, ***parts***, ***status***
3. Click **OK** to save the entity



Database

Generate DB scripts

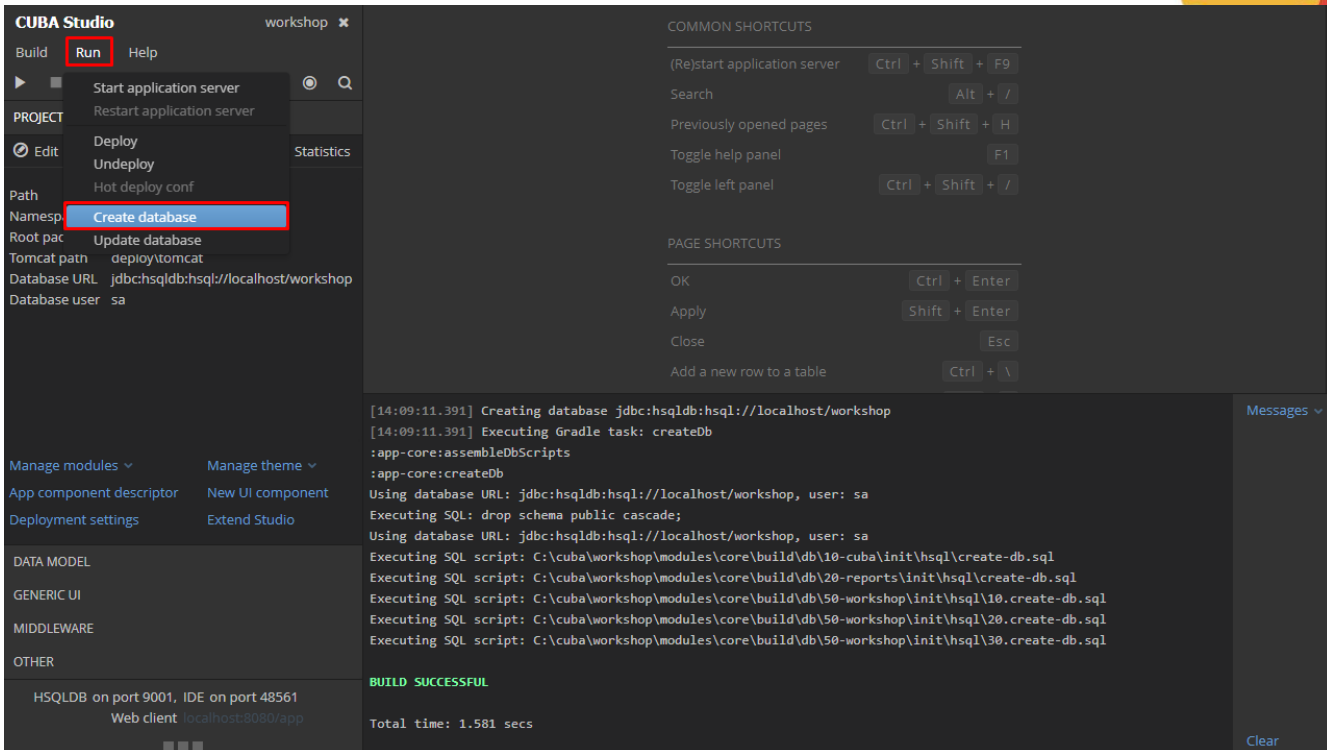
1. Click the **Generate DB scripts** link at the bottom of the **DATA MODEL** section
2. CUBA Studio has generated a script to create tables and constraints
3. Click **Save and close**
4. Studio has saved the scripts into a special directory of our project, so we will be able to access them if needed



Create database

1. Invoke the **Run — Create database** action from the menu to create a database
2. CUBA Studio warns us that the old DB will be deleted, click **OK**

Studio outputs process stages to the log. When **Build Successful** message is shown, our DB is created.

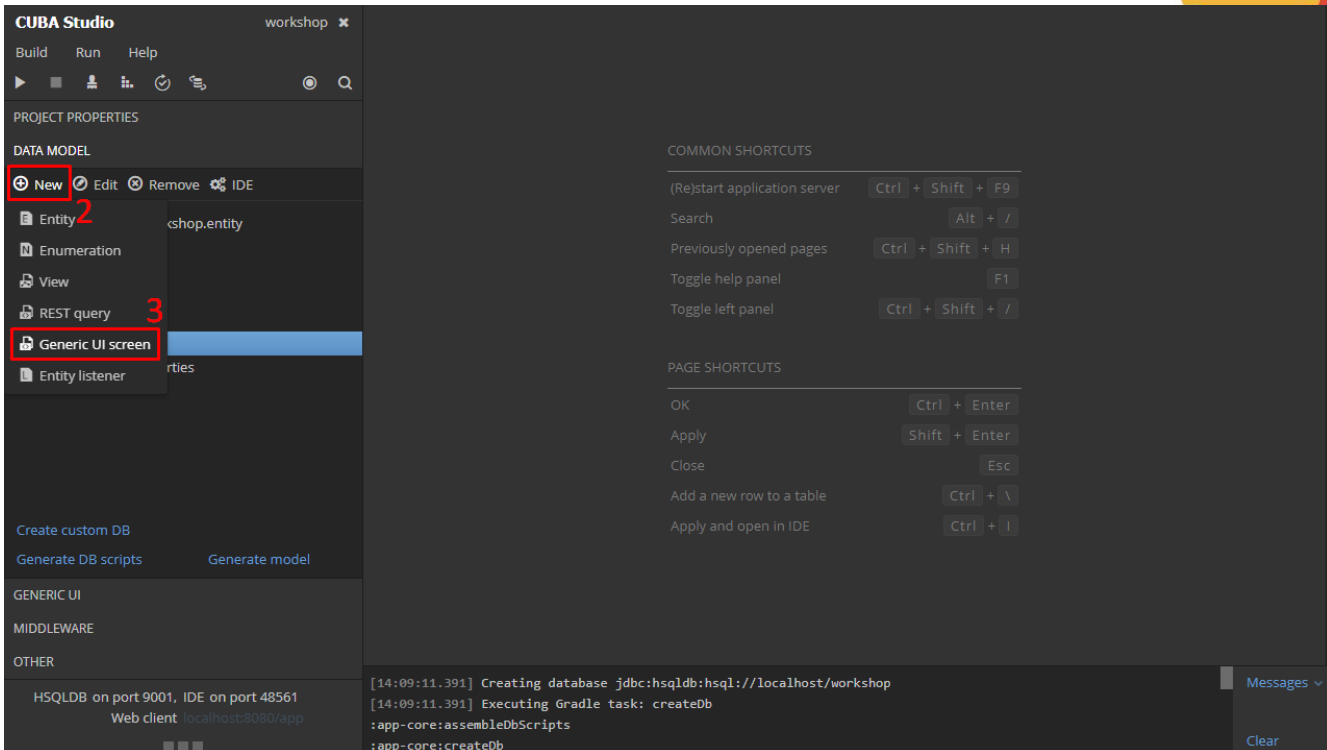


User Interface

Screens scaffolding

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

1. Select ***SparePart*** on the DATA MODEL section
2-3. Click **New -> Generic UI screen**



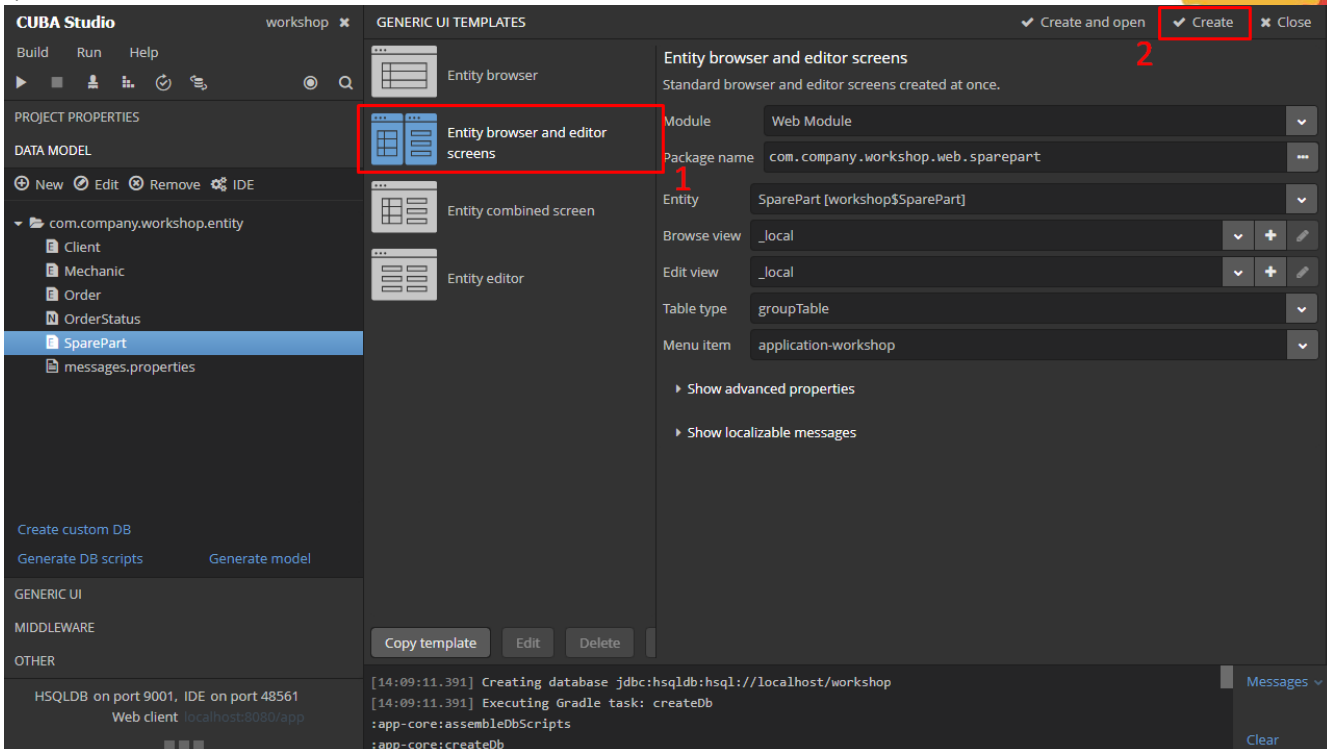
Screens scaffolding

On this screen we can specify the type of the screen, where to place the screens, which menu item will be used to open the browser screen.

The following terminology is used:

- Browser screen — screen with list of records
- Editor screen — simple edit form for record

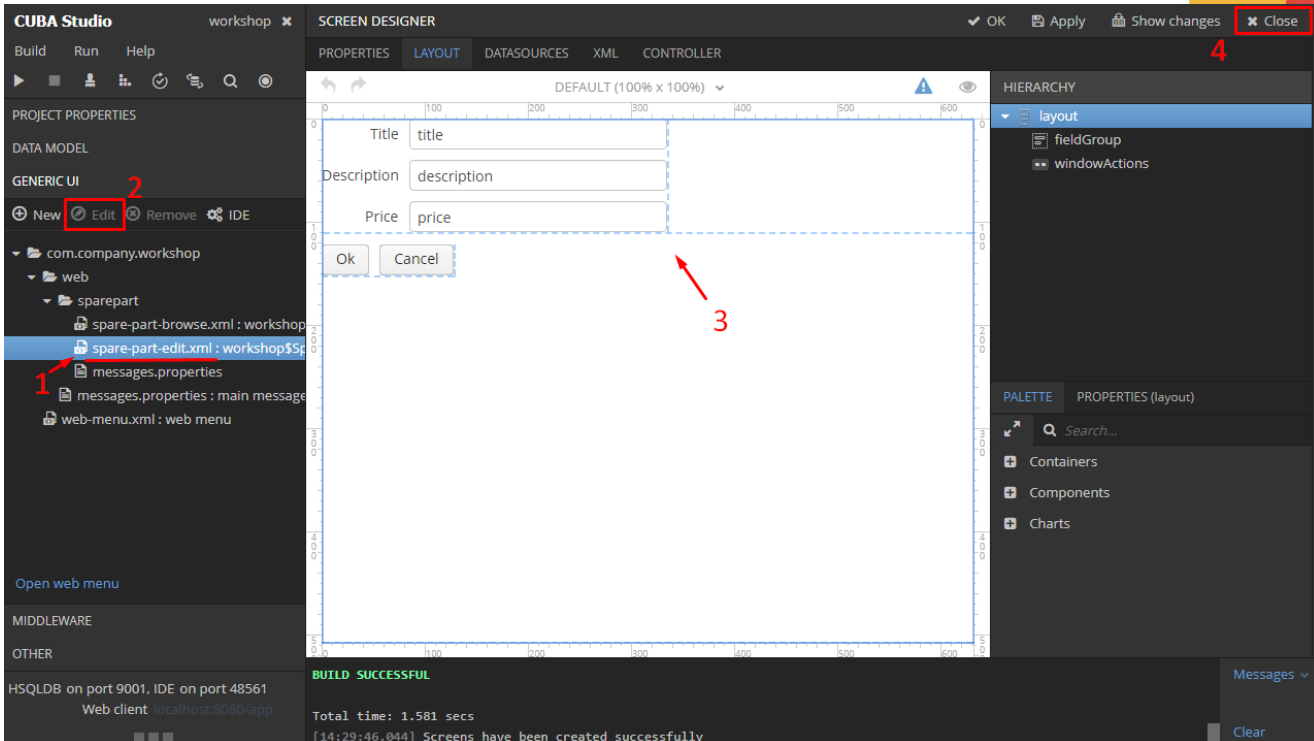
1. Select the **Entity browser and editor screens** template
2. Click **Create**



Screen designer

Studio has generated 2 screens. Let's have a look at **sparepart-edit.xml**.

1. Select **sparepart-edit.xml** in the GENERIC UI section
2. Click **Edit**
3. CUBA Studio features a built-in WYSIWYG screens editor to speed up UI development
4. Click **Close**

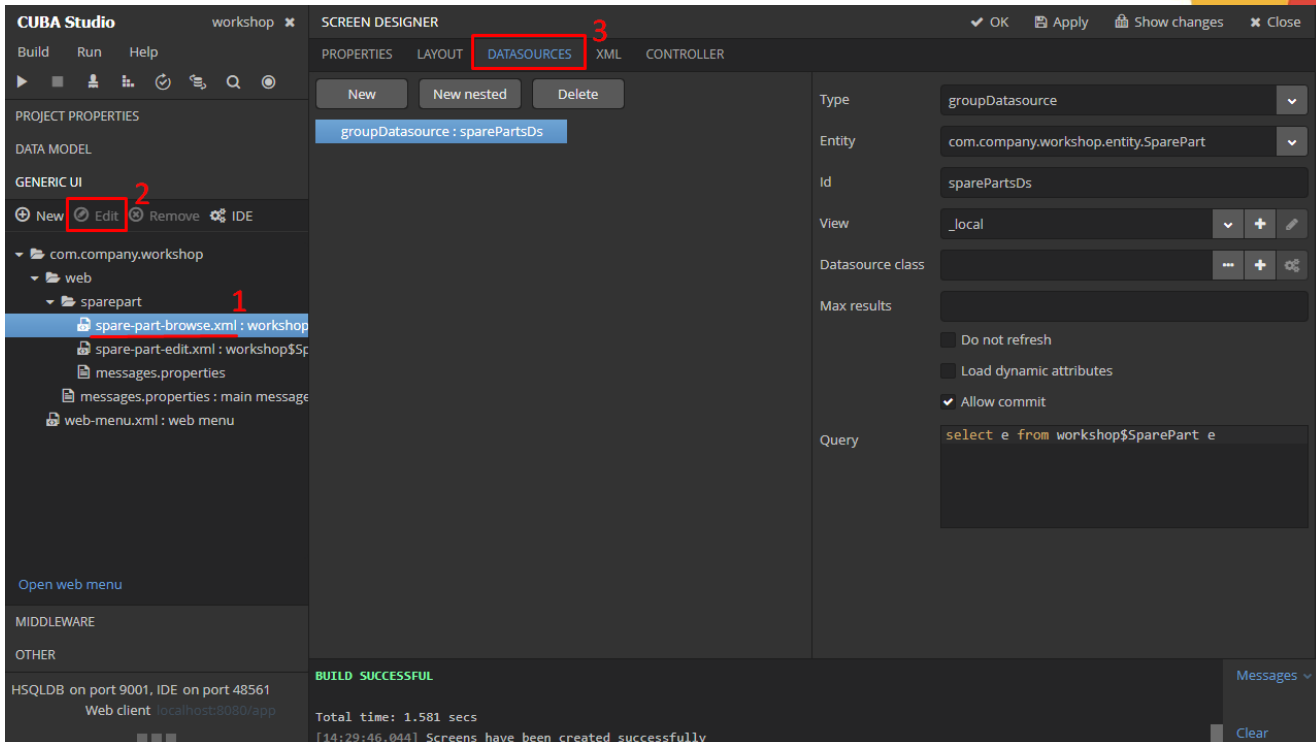


Data binding

Components are connected with datasources, which are configurable from the **Datasources** tab.

1. Select **sparepart-browse.xml**
2. Click **Edit**
3. Go to the **Datasources** tab

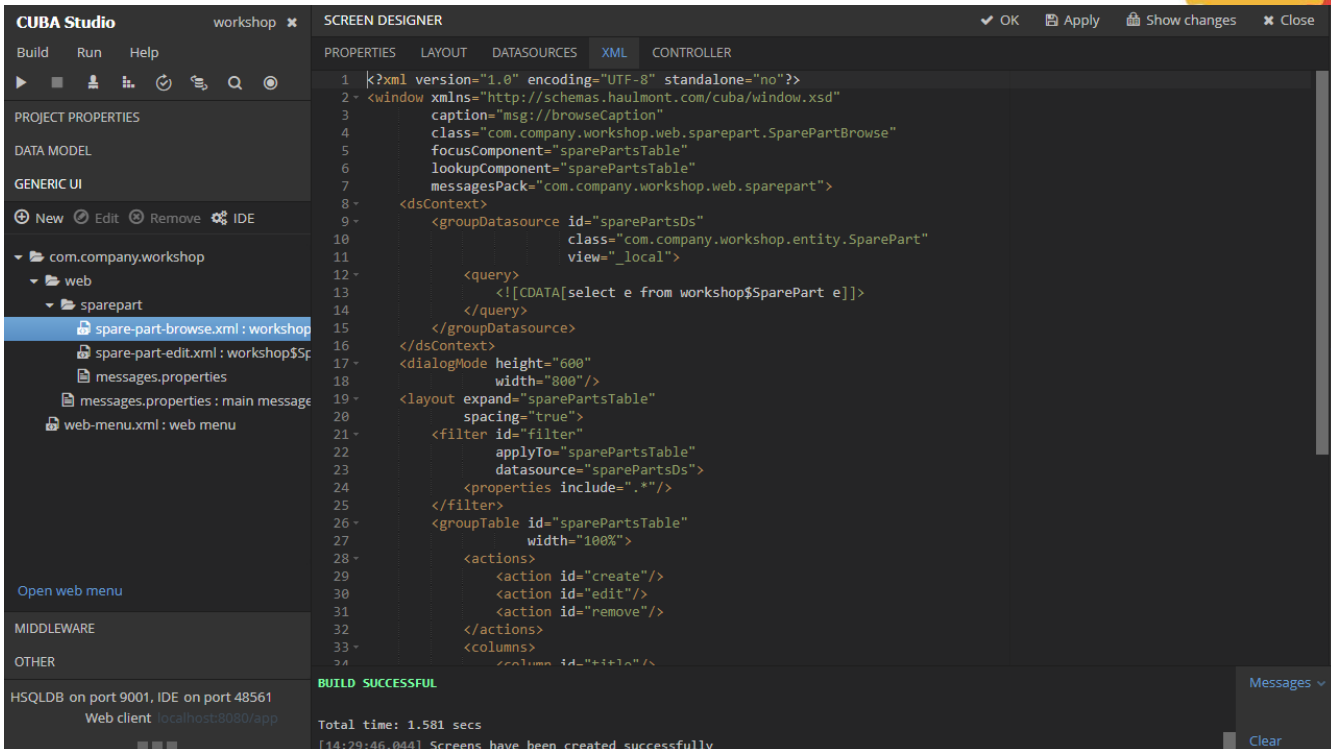
Datasources use JPQL queries to load data.



Declarative UI definition

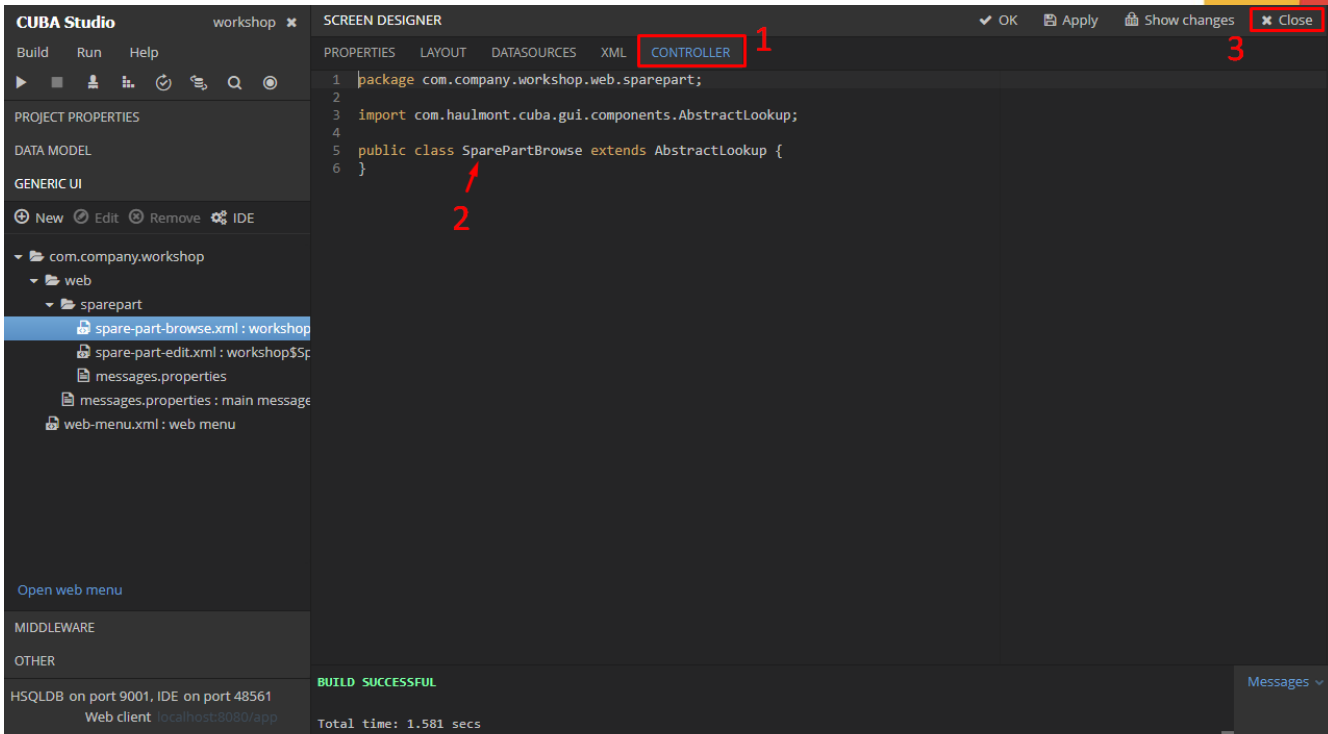
UI is described declaratively using XML, we can see an example of the descriptor in the **XML** tab

The XML view is synchronized with the graphical design, and if we make changes in XML, then the graphical view will be updated, and vice versa.



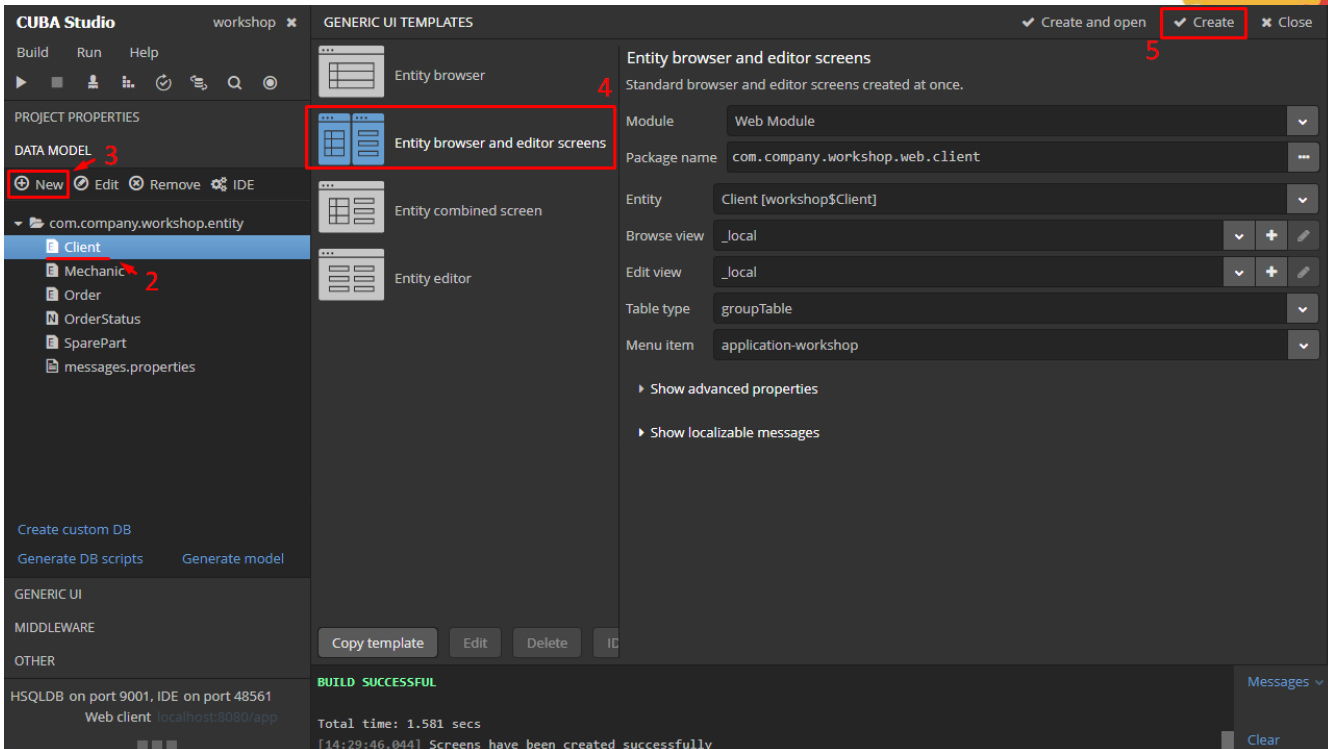
Screen controller

1. Go to the **Controller** tab
2. Apart from XML, Studio creates a controller for each screen, which is a Java class that implements the logic and handling component events
3. Click **Close**



Generate screens for Client entity

1. Open the DATA MODEL section of the navigation panel
2. Select the ***Client*** entity
3. Click **New -> Generic UI screen**
4. Select **Entity browser and editor screens**
5. Click **Create**

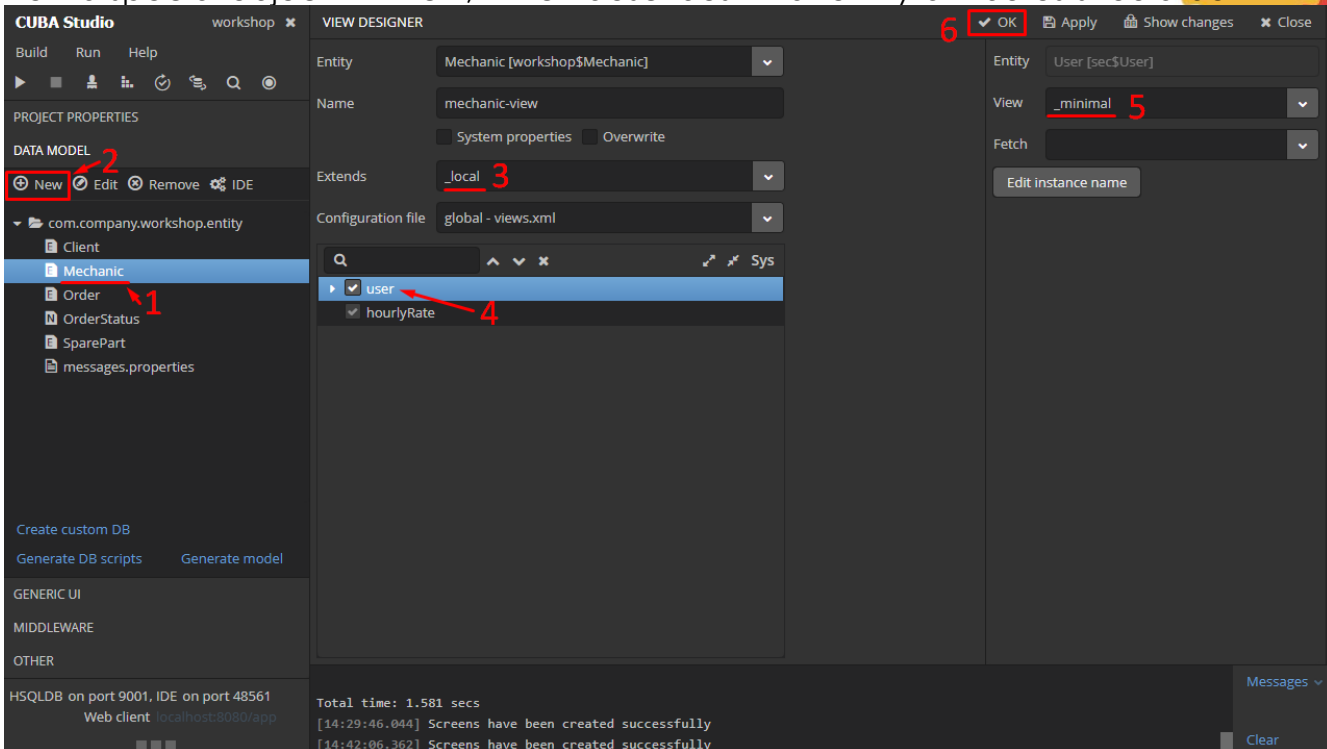


View. Loading of entity graphs from DB

The ***Mechanic*** entity is linked to ***User***. So, we need to load related ***User*** entity to display it in the browser and editor screens. In CUBA, this is done via special object — **View**, which describes what entity attributes should be loaded from the database.

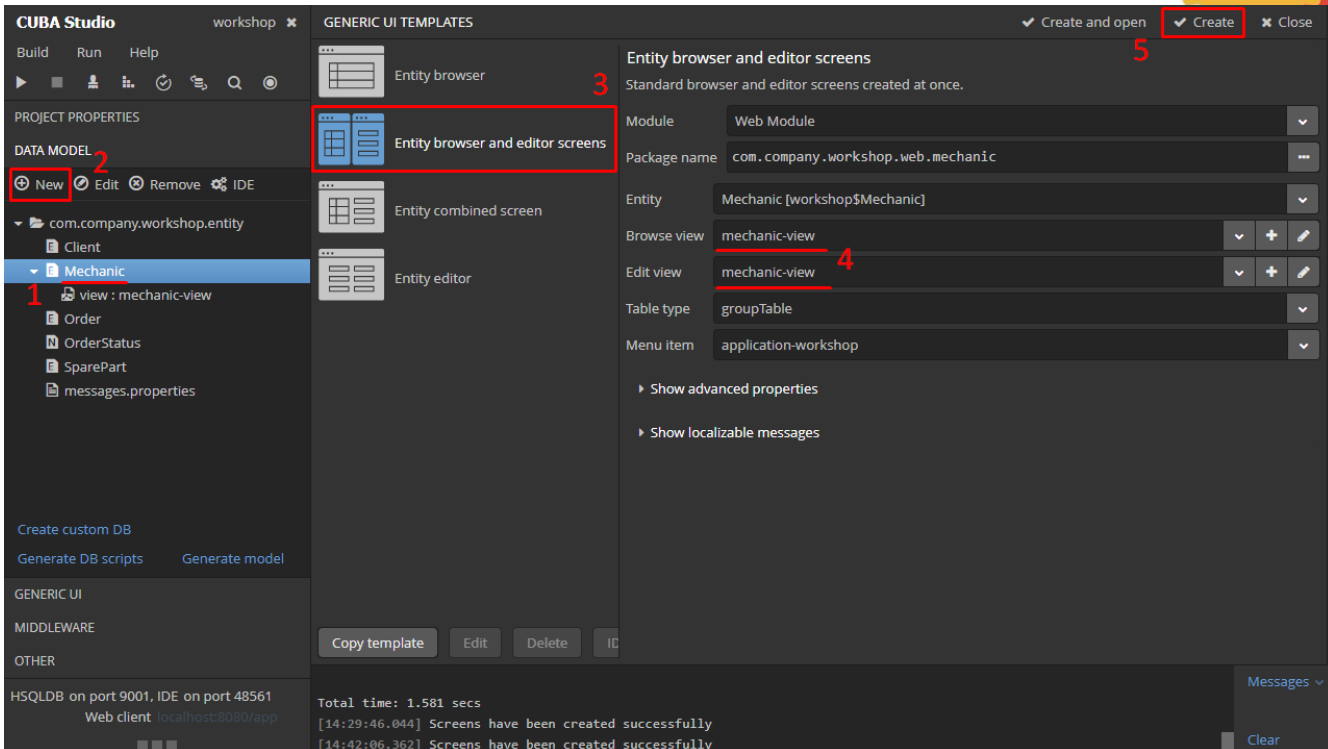
Let's create a view for the ***Mechanic*** entity, which will include ***User***.

1. Select the ***Mechanic*** entity
2. Click **New -> View**
3. Choose **Extends view**: ***_local***, as we want to include all local attributes
4. Select the ***user*** attribute,
5. Specify ***_minimal*** view for this attribute.
   _minimal view includes only attributes that are specified in the **Instance Name** of an entity
6. Click **OK** to save the view



Generate screens for Mechanic

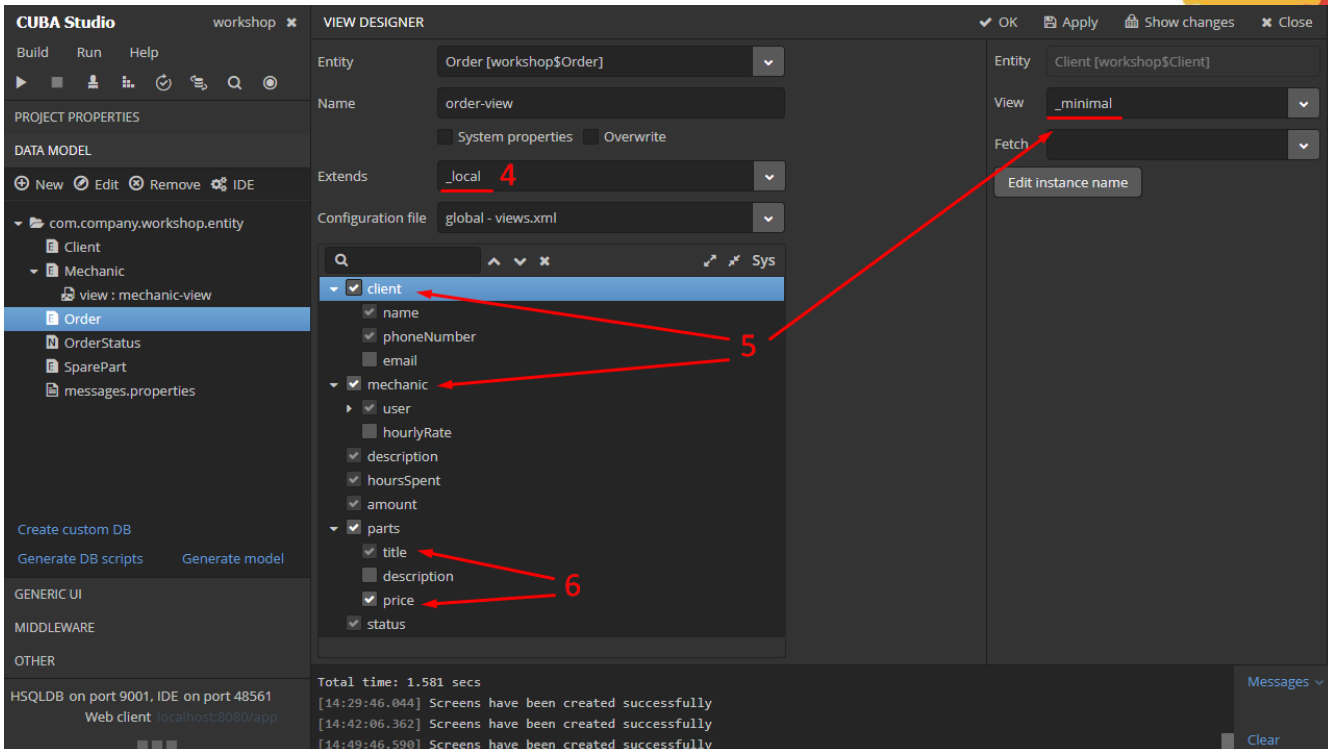
1. Select the ***Mechanic*** entity
2. Click **New -> Generic UI screen**
3. Select **Entity browser and editor screens** template
4. Choose ***mechanic-view*** for both browser and editor screen
5. Click **Create**



View for Order browser and editor

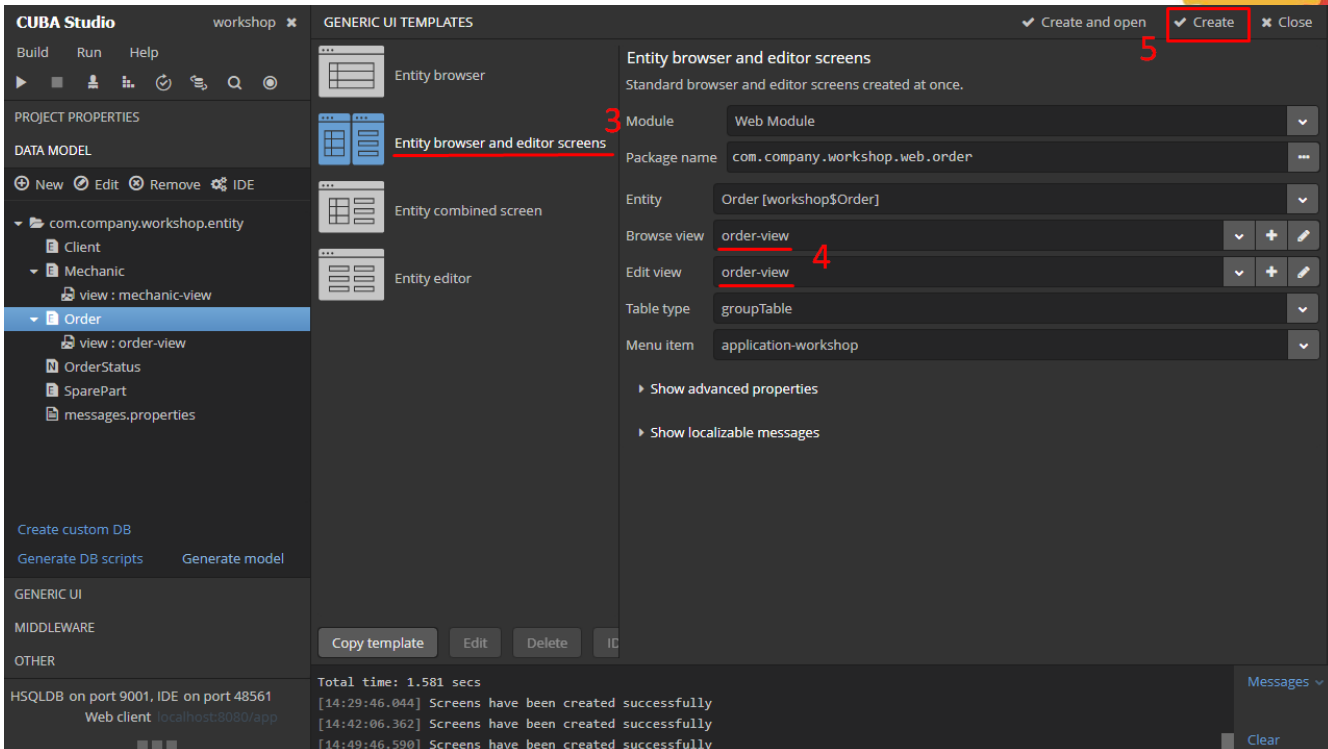
Now we need to create screens for the ***Order*** entity. We'll also need to create a special view.

1. Open the DATA MODEL section of the navigation panel
2. Select the ***Order*** entity
3. Click **New -> View**
4. Set **Extends** to ***_local*** to include all local properties
5. Tick ***client***, ***mechanic*** and select the ***_minimal*** view for them
6. Tick ***title*** and ***price*** for ***parts***
7. Click **OK** to save the view



Generate screens for the Order entity

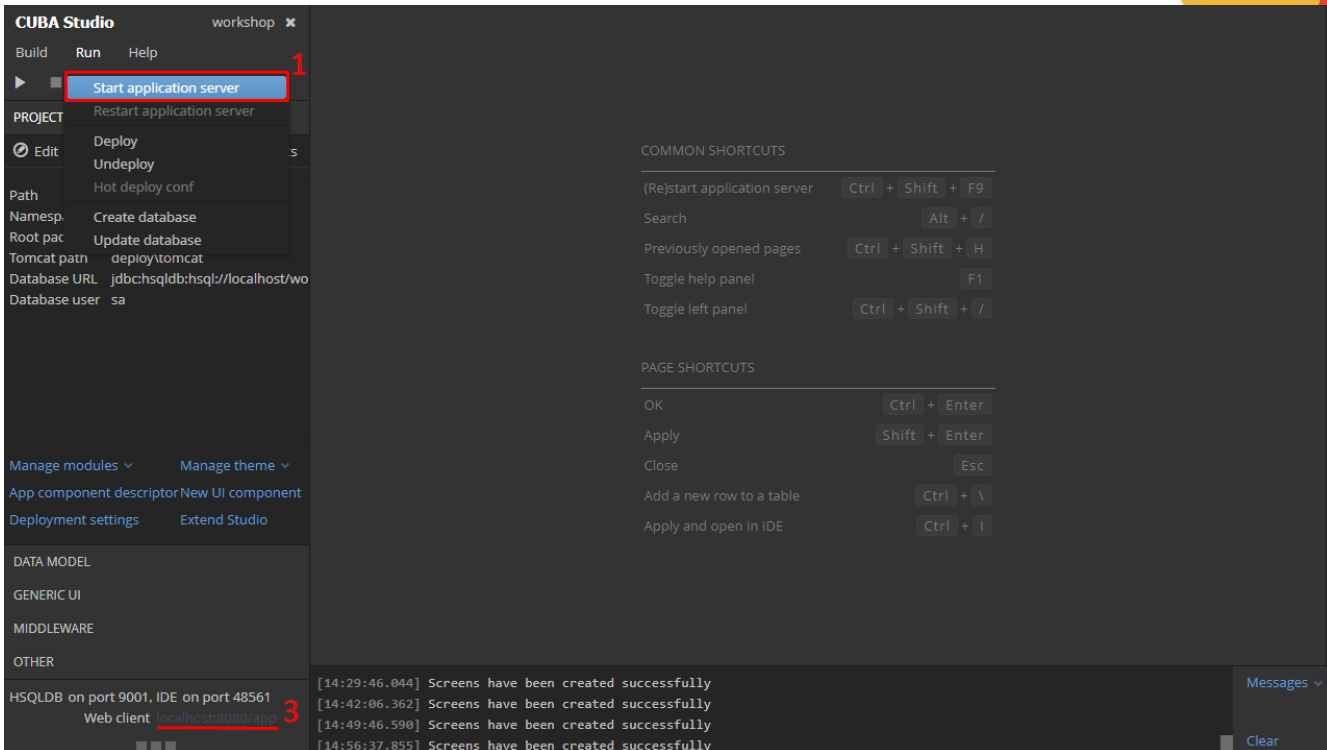
1. Select the ***Order*** entity
2. Click **New -> Generic UI screen**
3. Select **Entity browser and editor screens**
4. Choose **order-view** for both screens
5. Click **Create**



Let's test it

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.
2. Studio will deploy a local Tomcat instance in the project subdirectory, deploy the compiled application there and launch it.
3. Open the application by clicking a link in the bottom part of Studio.

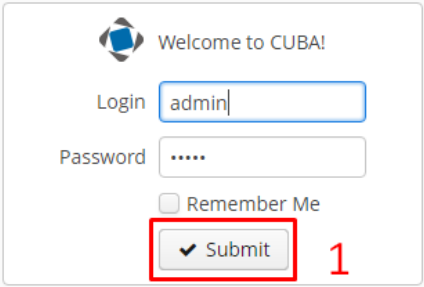


First launch and CRUD

Login screen

The system login screen has appeared. This is a standard CUBA screen, which can be customized, as everything in CUBA, to meet specific requirements.

1.Click **Submit** to login

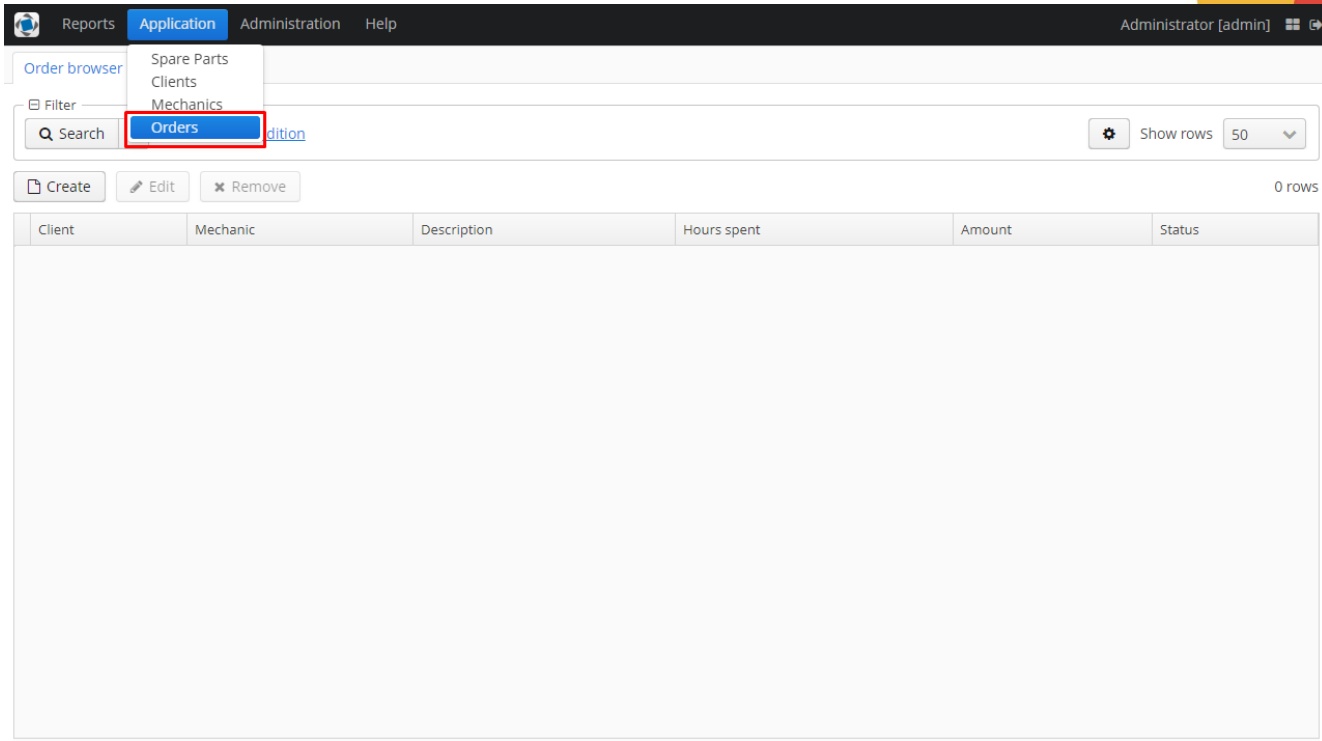


Order browser

Since we have not changed the application menu, our items are displayed by default under the **Application** menu.

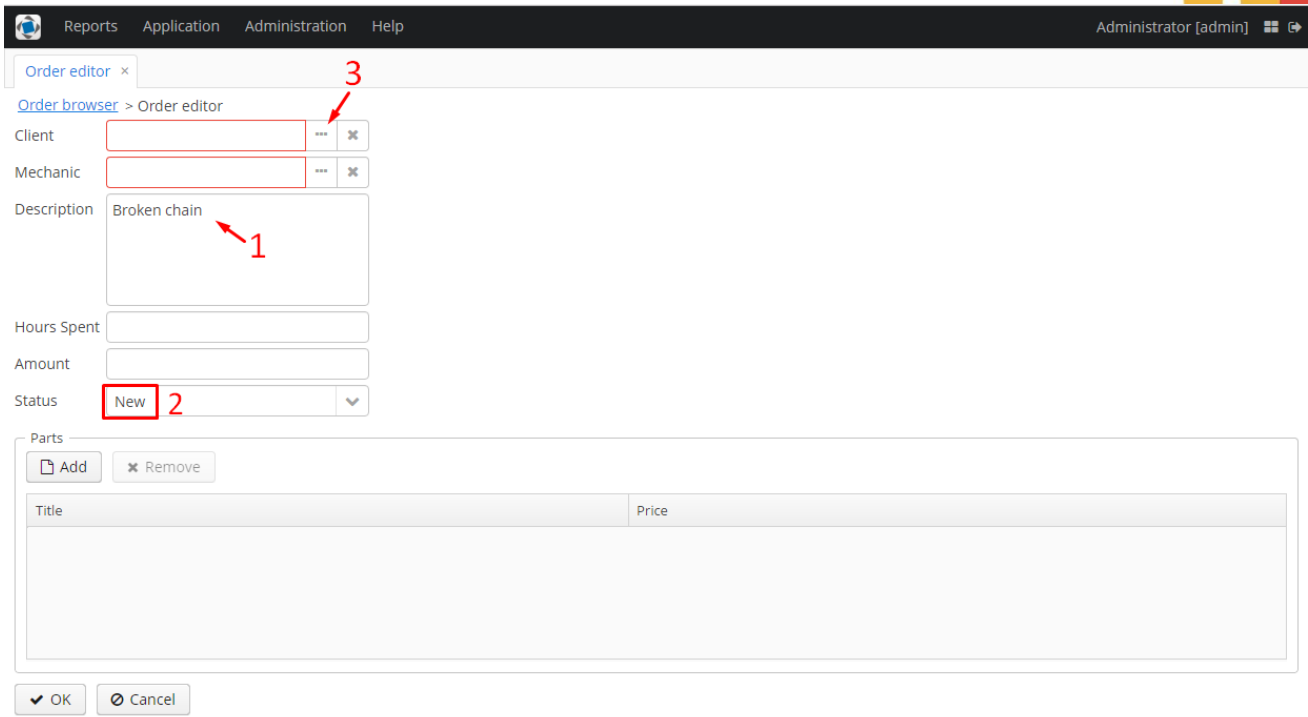
1.Open **Application — Orders** from the menu

This is a standard browser screen with a filter on top and a table below.



Order edit screen

1. Click **Create** and enter the ***description***
2. Select Status: **New**
3. Click button **[...]** to select a ***client*** for the order



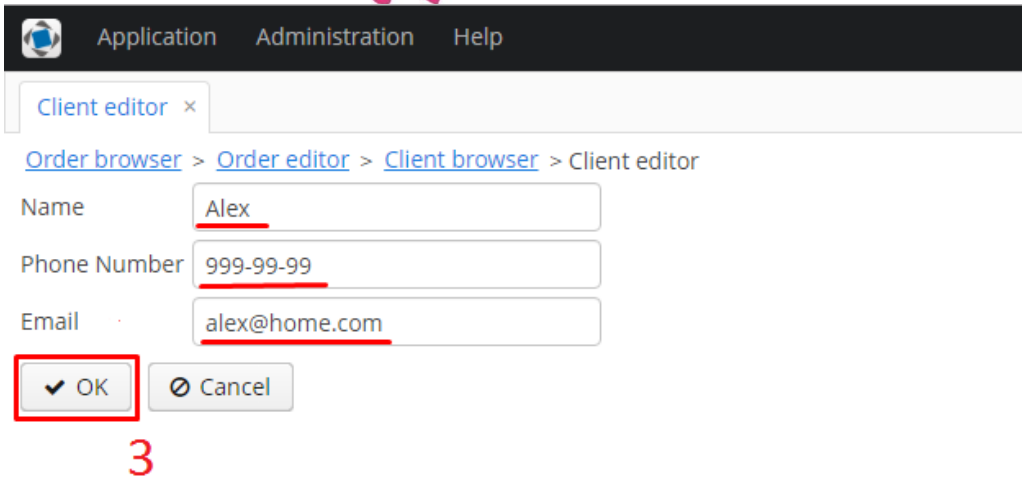
Client browser

So far we don't have any clients. Let's create one.

1. Click **Create**
2. Fill attributes of the new client

Name: ***Alex***
Phone number: **999-99-99**
Email: ***alex@test.com***

3. Click **OK**
4. Click **Select** to set client to the order

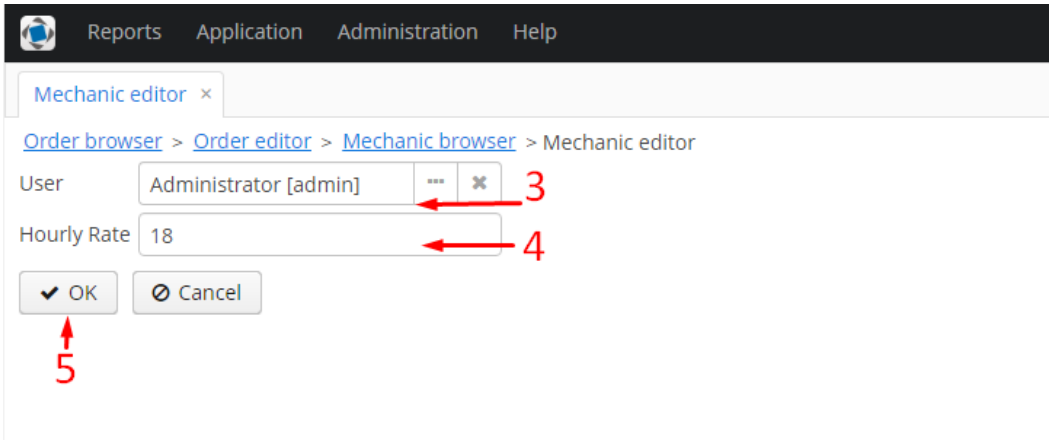


Assign mechanic for the order

You are now back to the ***Order* editor** screen

1. Click button **[...]** at the right of the **mechanic field** in the **Order editor**
2. Click **Create** to add a new mechanic
3. Select ***admin*** user for this mechanic
4. Enter **hourly rate**
5. Click **OK**
6. Select mechanic for the order

You can go back to any of opened screens using the navigation at the top of screen.



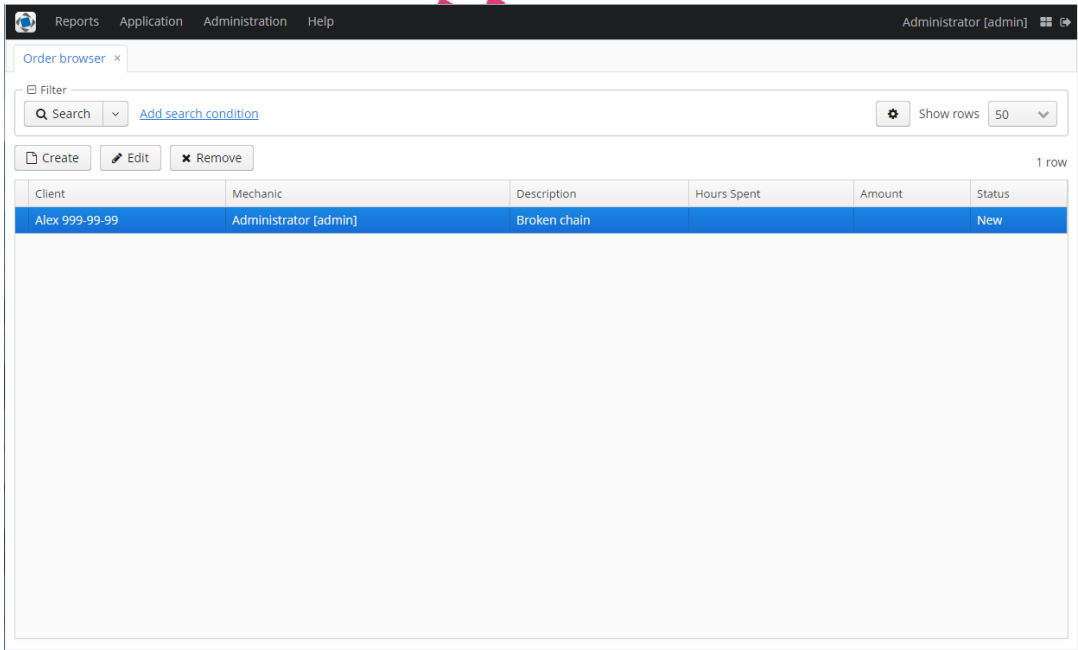
CRUD application

1. Fulfil some description and set the **New** status
2. Click **OK** to save the order

This is a small working CRUD application that writes data to the database and allows you to simply keep track of orders.

We can search for orders using our Filter.

Table component enables us to hide and change the width of columns. Also, our table is sortable.



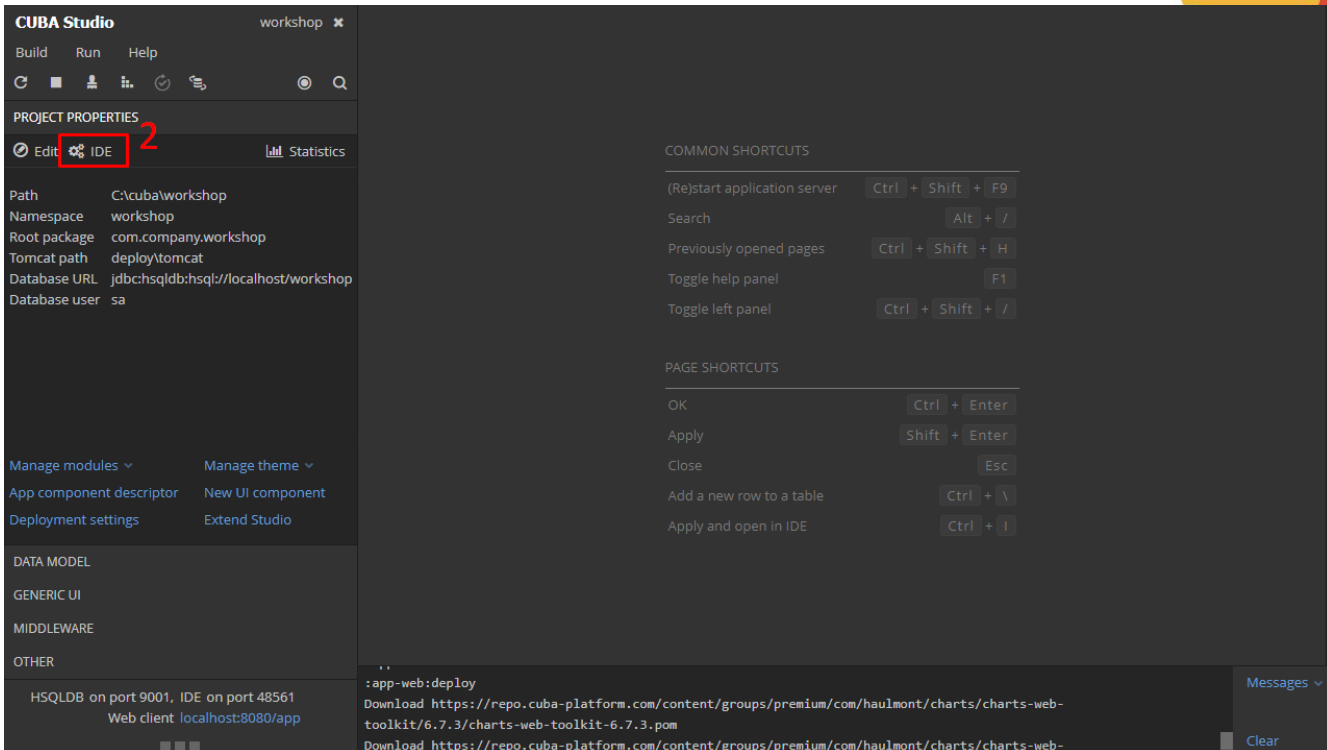
Integration with IDE and project structure

Go to the IDE

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 CUBA Studio
2. Go to Studio and click the **IDE** button in the **Project properties** section

The project will come up in the IDE.



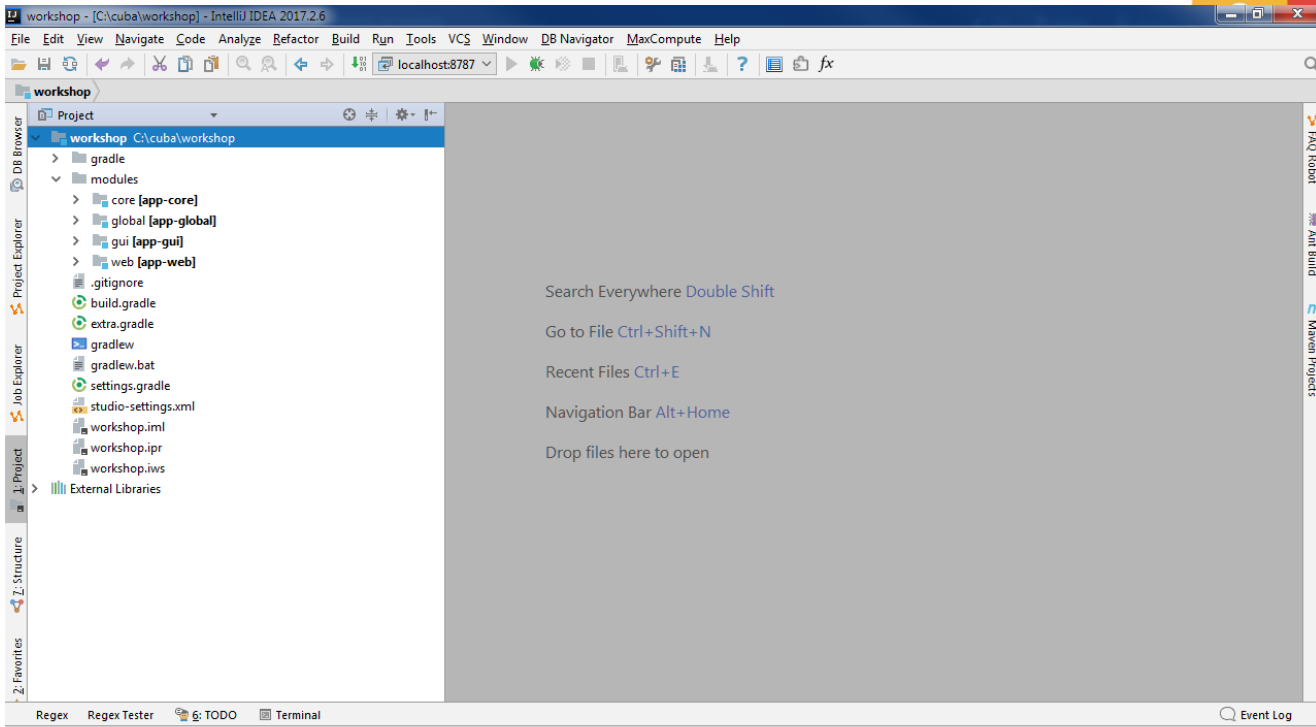
Project structure

By default, any project consists of 4 modules: **global**, **core**, **web**, **gui**.
The **global** module contains data model classes, **core** - middle tier services, **gui** - screens and components, **web** - web client-specific code.

You can have other clients in your project, such as a desktop application or a web portal, which will be placed in separate modules.

The project root directory contains the application build scripts.

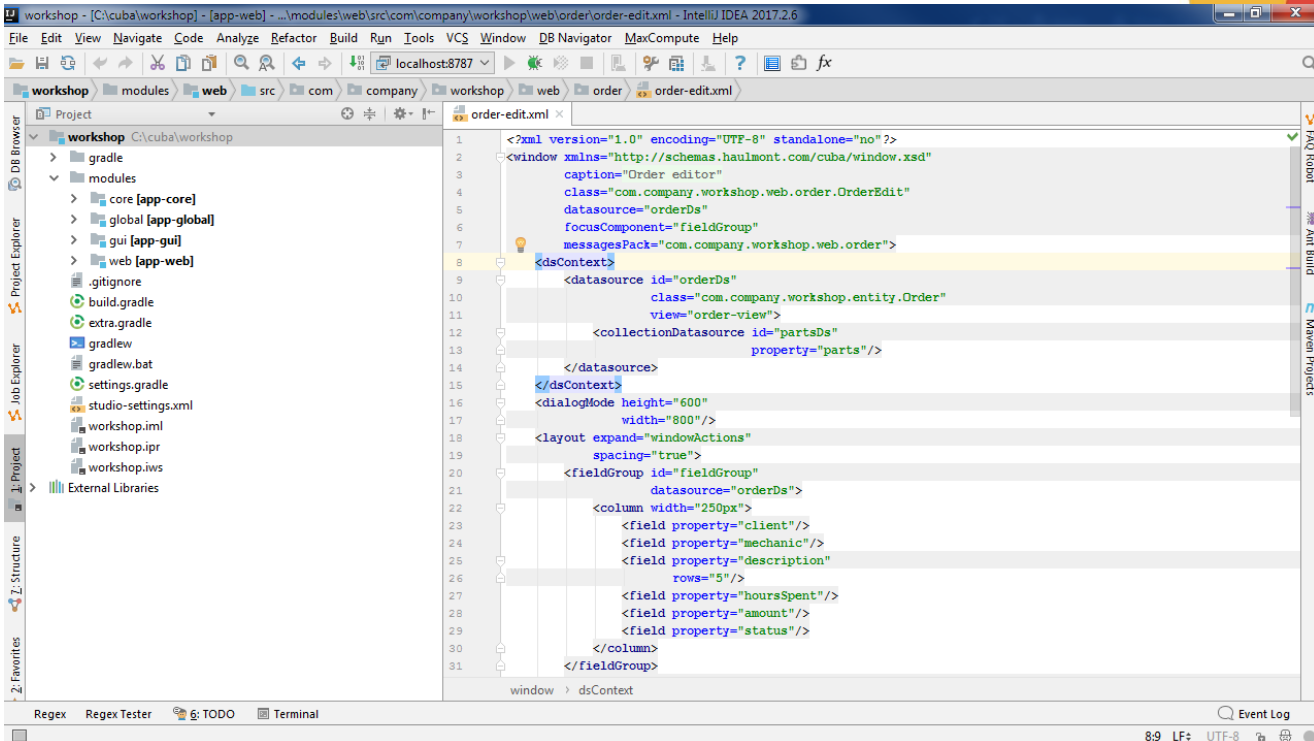
Applications are built using Gradle.



CUBA Studio IDE integration

1. Go to the GENERIC UI section of the navigation panel in CUBA Studio
2. Select the **order-edit.xml** screen
3. Click the **IDE** button on top of the section

IntelliJ IDEA will open the **order-edit.xml** file.
We can edit any file of the project manually using IntelliJ IDEA (or your favorite IDE).

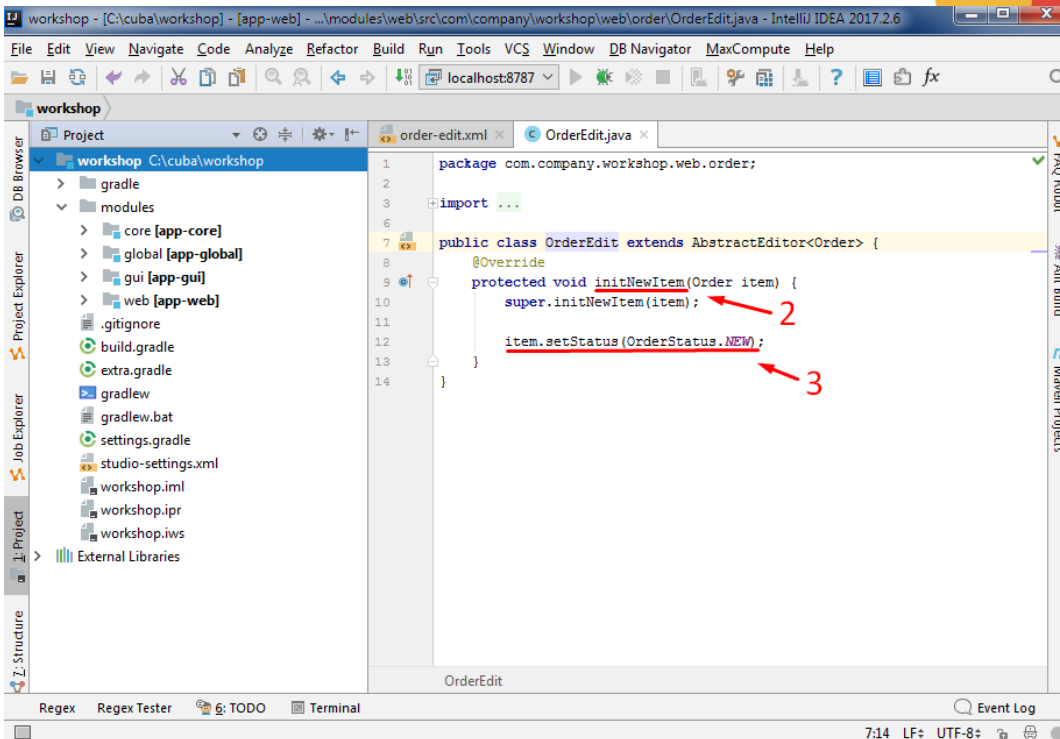


Set default Status for an order

Stay in the IDE and follow the steps:

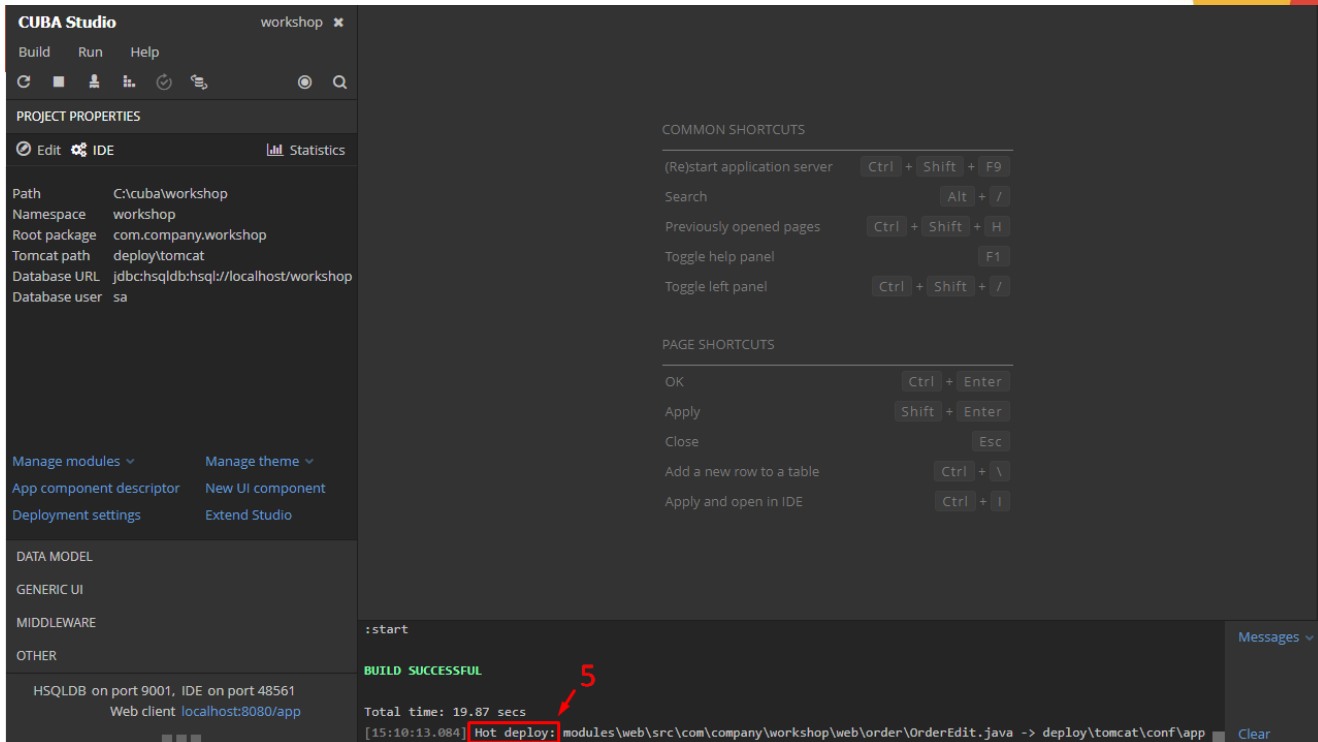
1. Hold **Ctrl** button and click on ***OrderEdit*** in the class attribute of the XML descriptor to navigate to its implementation
2. Override method ***initNewItem***
3. Set status ***OrderStatus.NEW*** to the passed order

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

        item.setStatus(OrderStatus.NEW);
    }
}
```



Hot deploy

1. Open our application in the browser
2. Open/Reopen **Application — Orders** screen
3. Click **Create**
4. We see our changes, although we haven't restarted the server
5. CUBA Studio automatically detects and hot-deploys changes, except for the data model, which saves a lot of time while UI development



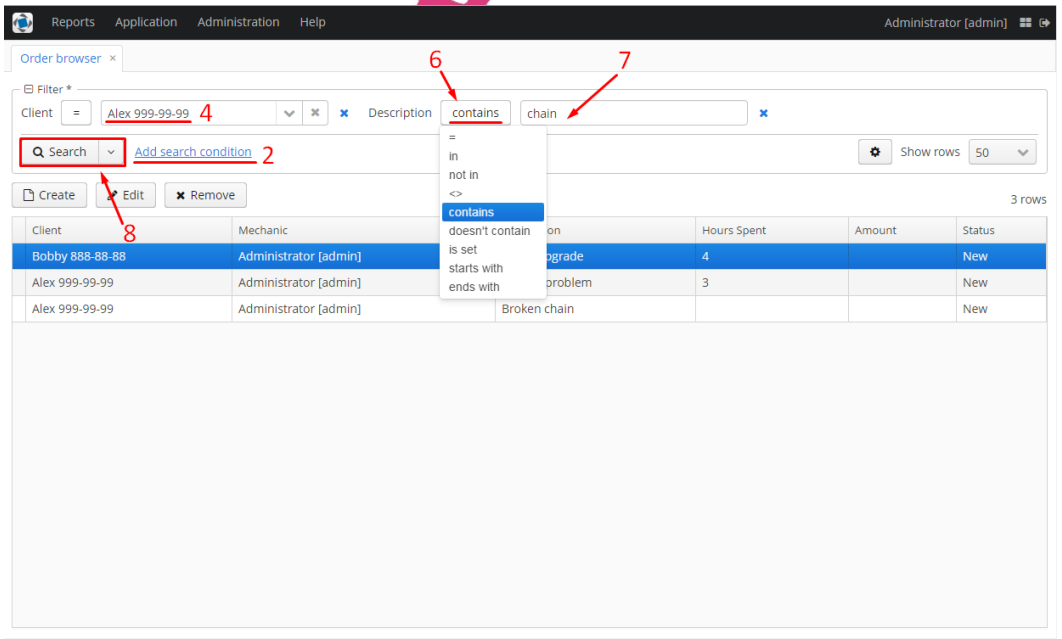
Generic filter

Filter component

1. Add a few orders to the system
2. Click **Add search condition**
3. Select **Client**
4. Set ***Alex*** as value for condition for the **Client** attribute
5. Click **Add new condition**, select **Description**
6. Change **[=]** operation to **[contains]**
7. Enter a word to **Description** field
8. Click **Search**

The filter is a versatile generic tool for filtering lists of entities, typically used on browser screens.

It enables quick data filtering by arbitrary conditions and saving them for repeated use.





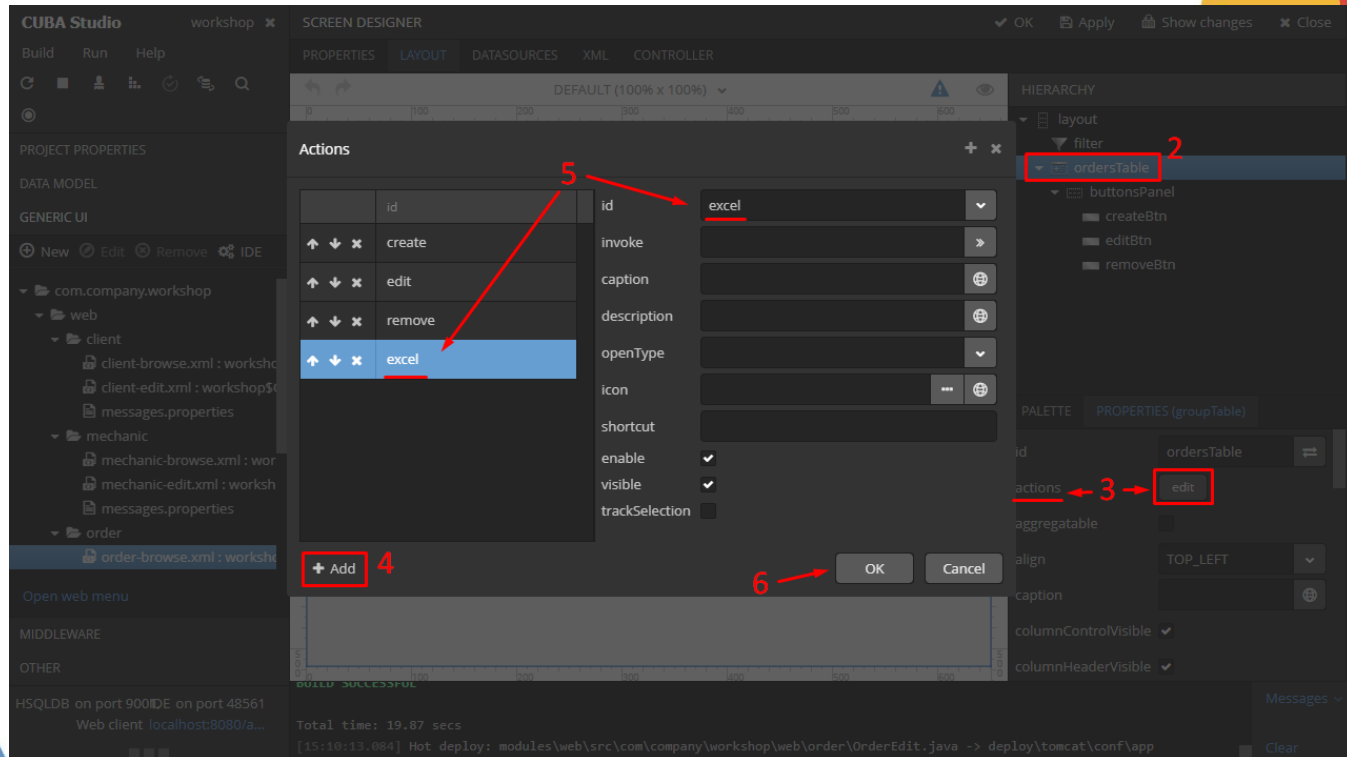

Actions

Standard actions

The standard screens contain **Create**, **Edit**, and **Remove** actions by default.

Let's add an action to **export** the order list to **Excel**.

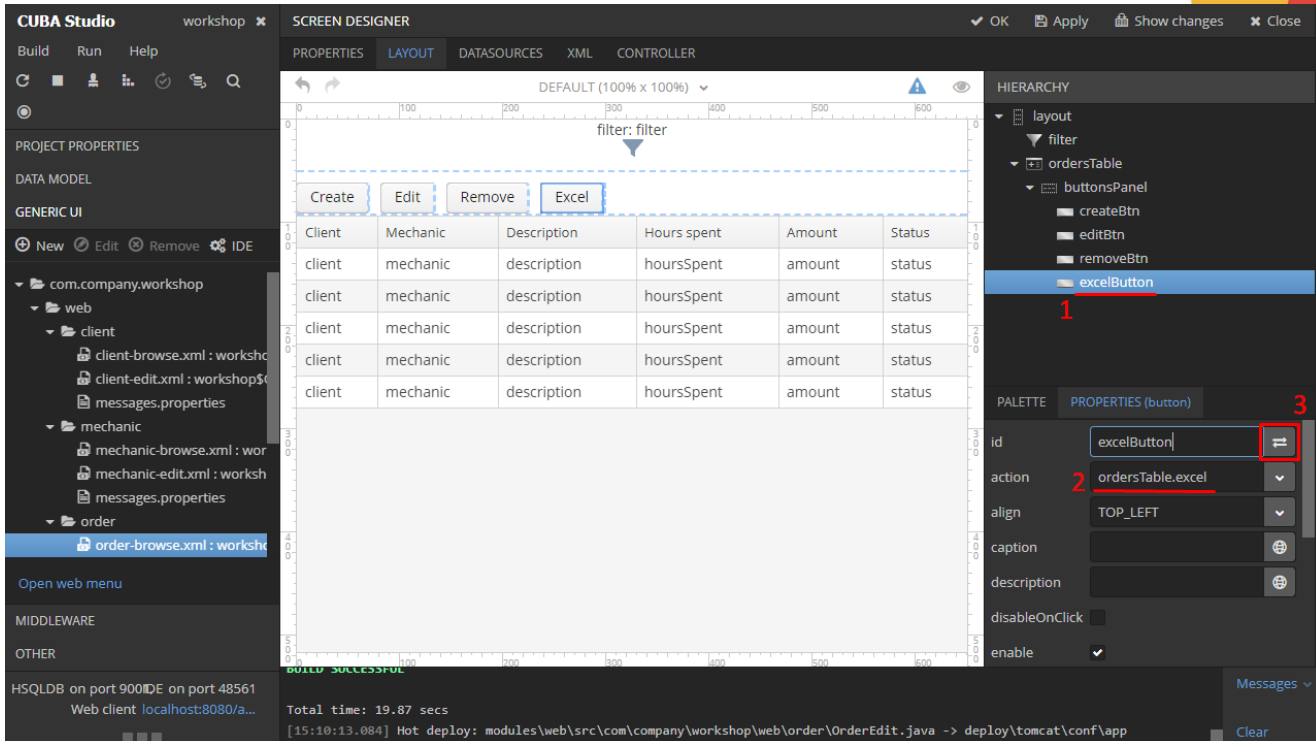
1. Open **order-browse.xml** screen in Studio.
2. Select table component, go to properties panel
3. Click the **edit** button in the **actions** property
4. **Add** a new action row to the list
5. Specify id as ***excel*** for this action
6. Click **OK**



Excel action

1. Add a new button to the button panel (**drag and drop** it into the hierarchy of components)
2. Select ***ordersTable.excel*** action for button using properties panel
3. Generate the button ID
4. Save the screen
5. Open/Reopen the **Orders** screen
6. Click **Excel** to export your orders to an XLSX file

The platform has standard actions for common operations: **Create**, **Edit**, **Remove**, **Include**, **Exclude** (for sets), **Refresh**, **Excel**, and you can create your own actions.





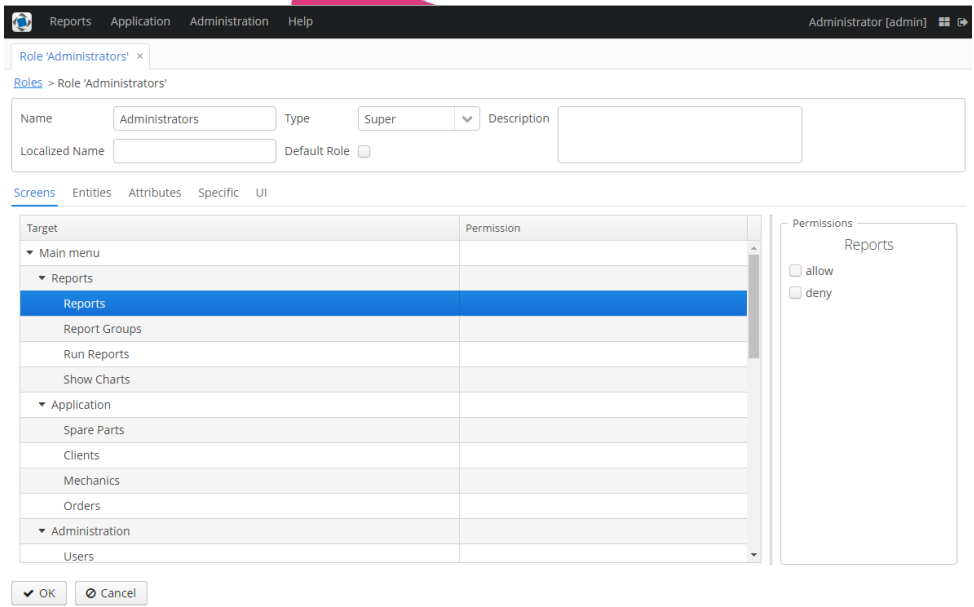

Security

Security subsystem

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.

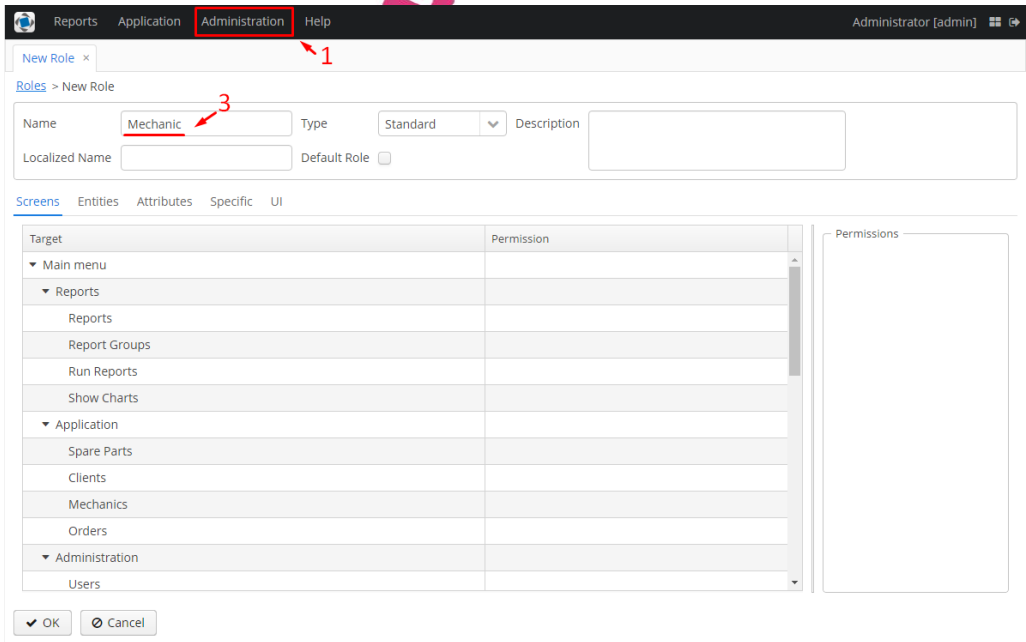
All security settings can be configured at runtime. There is also an additional facility to control row level access.



Mechanic role

We need the **Mechanic role** for 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.

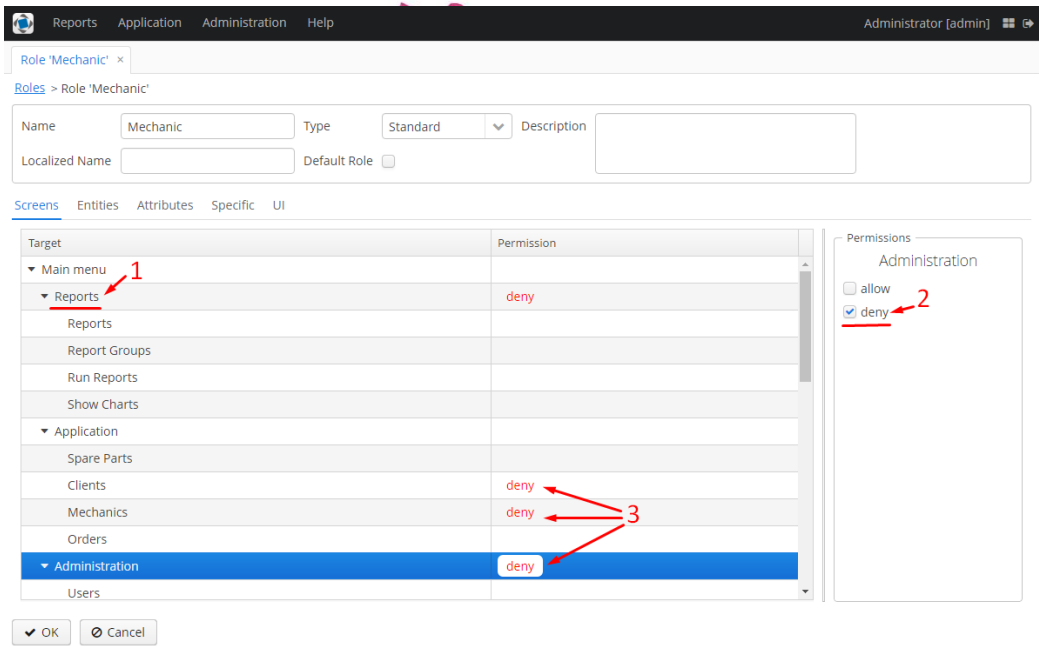
1. Open **Administration — Roles** from the menu
2. Click **Create**
3. Set **Name**: ***Mechanic***



Screen permissions

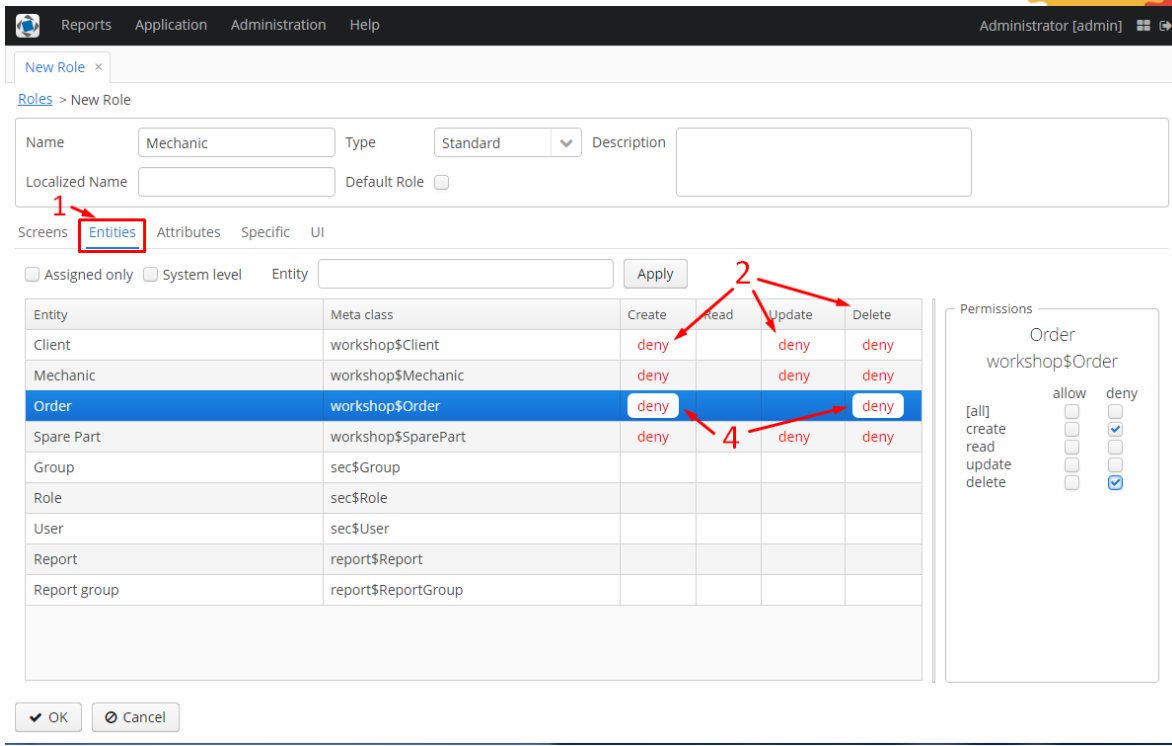
We want to **restrict** access to **Administration screens** for all **Mechanic users**, so let's forbid the **Administration** menu and **Reports** menu items. Also, mechanics don't need access to the mechanics and clients browsers, let's forbid the corresponding screens.

1. Select **Reports** row in the table with **Screens**
2. Select **deny** checkbox at the right
3. Similarly deny access for **Administration**, **Clients** and **Mechanics**



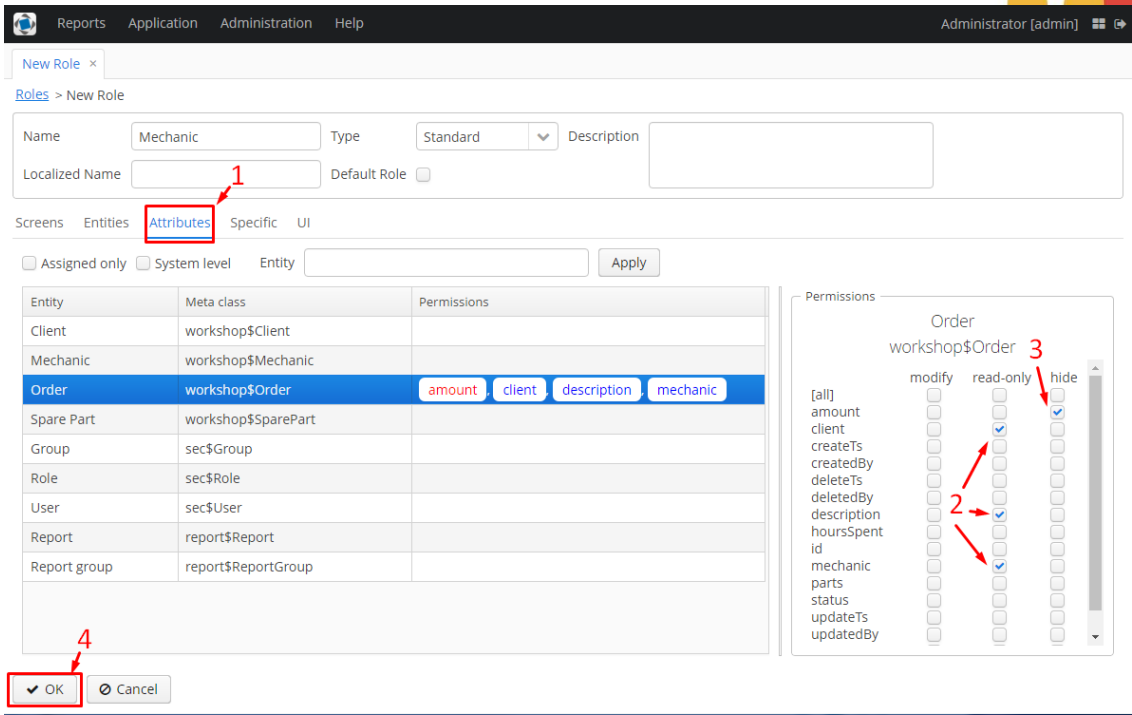
CRUD permissions

1. Open the **Entities** tab
2. Select the ***Client*** entity and forbid **create**, **update** and **delete** operations
3. Same for the ***Mechanic*** and ***SparePart*** entities
4. For **Order**, we'll restrict only **create** and **delete**



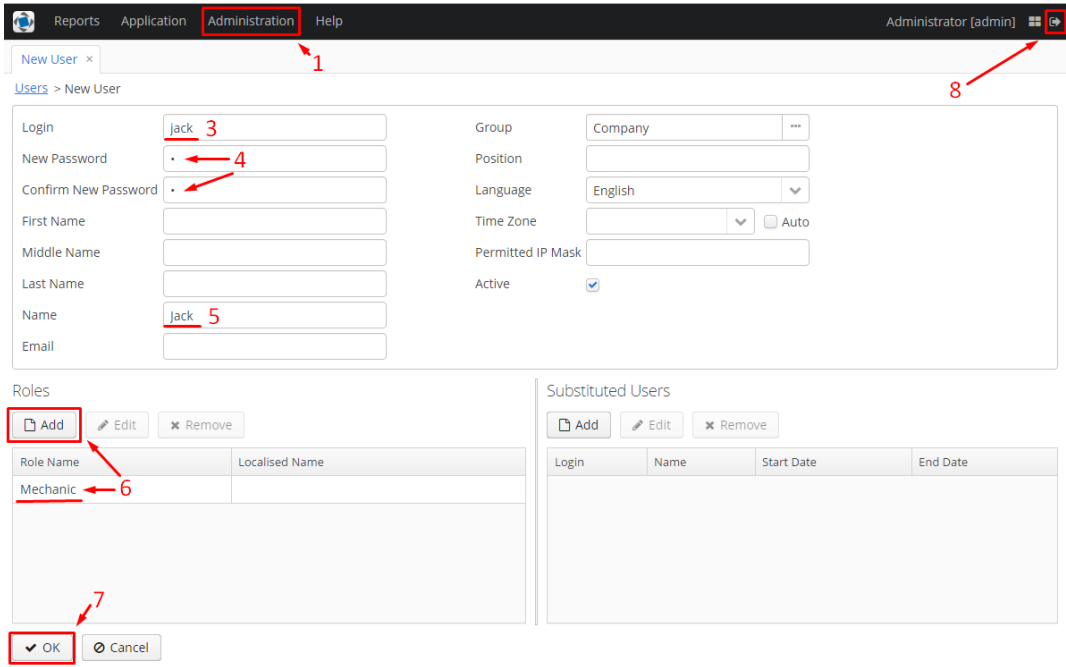
Attribute permissions

1. Open the **Attributes** tab
2. Select **Order** row and tick **read only** for ***client***, ***mechanic*** and ***description***
3. Set **hide** for amount attribute
4. Click **OK** to save the role



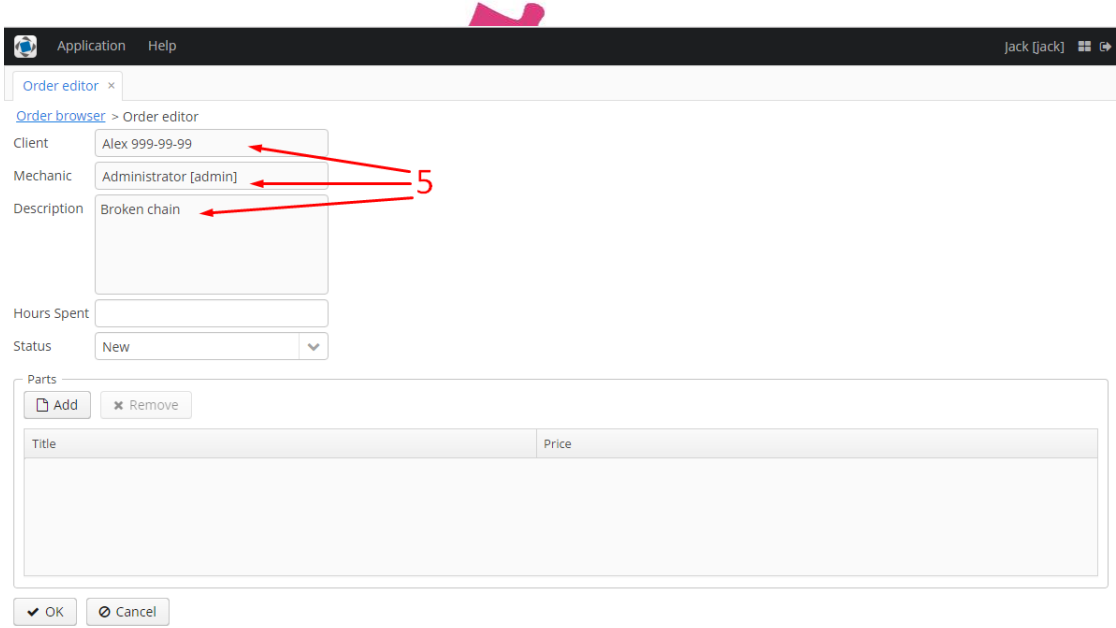
New user

1. Open **Administration — Users** from the menu
2. Click **Create**
3. Set **Login**: ***jack***
4. Specify password and password confirmation
5. Set **Name**: **Jack**
6. Add the ***Mechanic*** role to user ***Roles***
7. Click **OK** to save the user
8. Click on **exit** icon at the top right corner of application window



Role-based security in action

1. Login to the system as ***jack***
2. **Reports** and **Administration** menus are now hidden
3. Open **Application — Orders** from the menu
4. **Edit** existing order
5. The ***description***, ***client*** and ***mechanic*** fields are read-only
6. The ***amount*** field is hidden

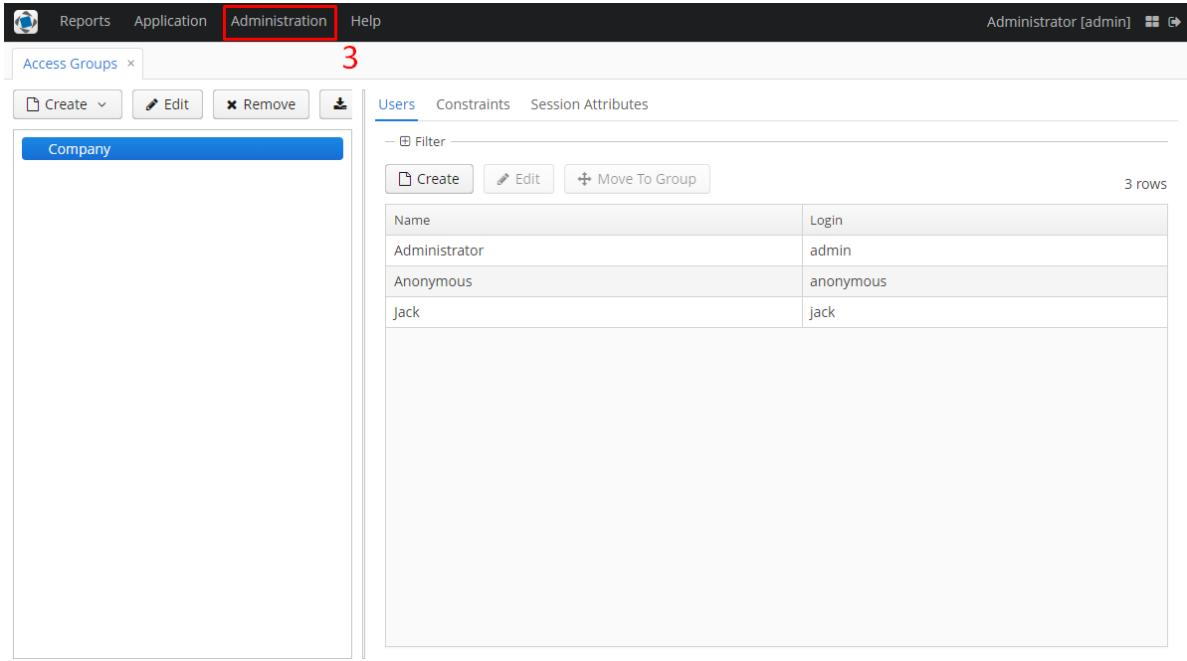


Row level security

What about the visibility of orders for the mechanic?

Let's limit the list of displayed orders to the logged in mechanic's orders only. We will use the **access group** mechanism for this.

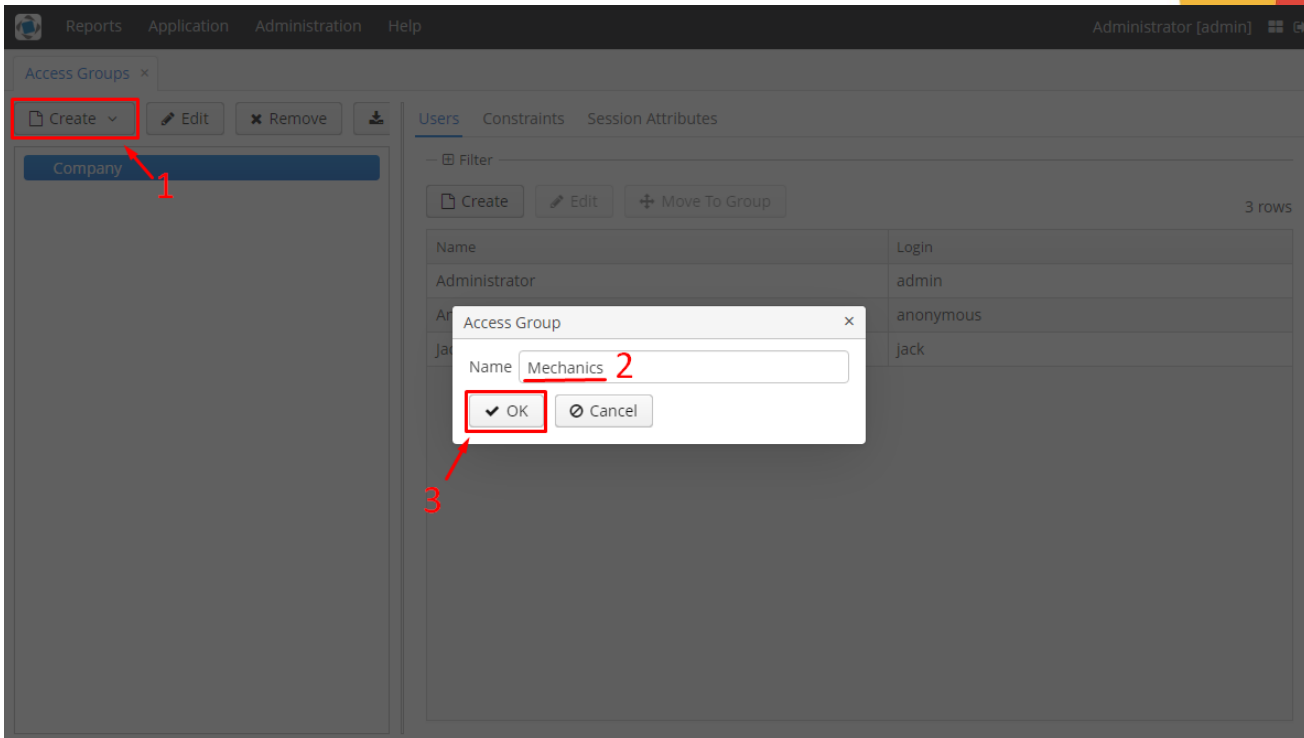
1. **Log out** of the system
2. Log in as ***admin***
3. Open **Administration — Access Groups** from the menu



The groups have hierarchical structure, where
each element defines a set of constraints,
allowing controlling access to individual entity
instances (at row level).

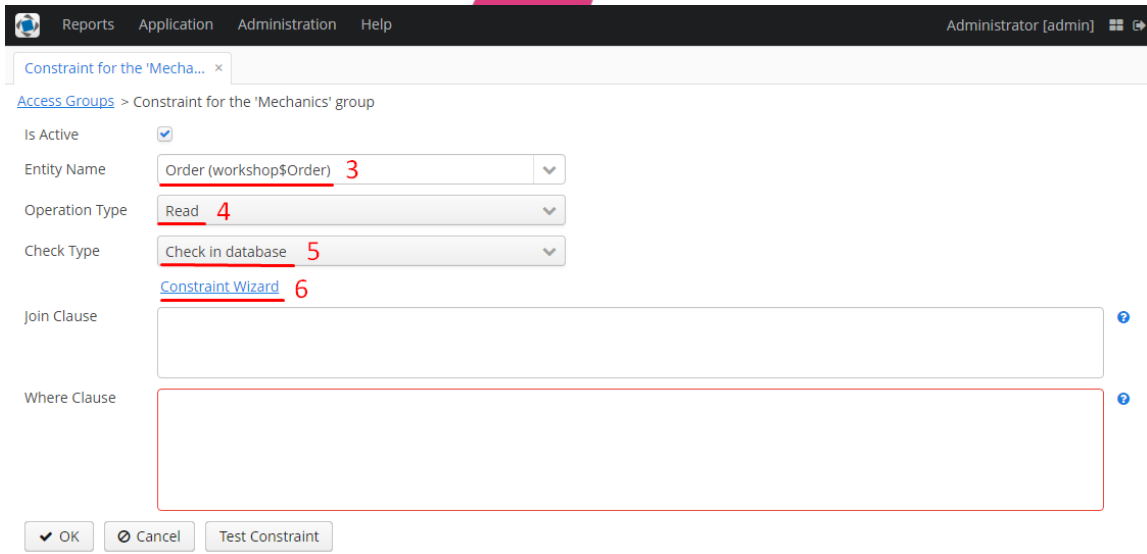
Create an access group

1. Click **Create — New**
2. Set **Name**: ***Mechanics***
3. Click **OK**



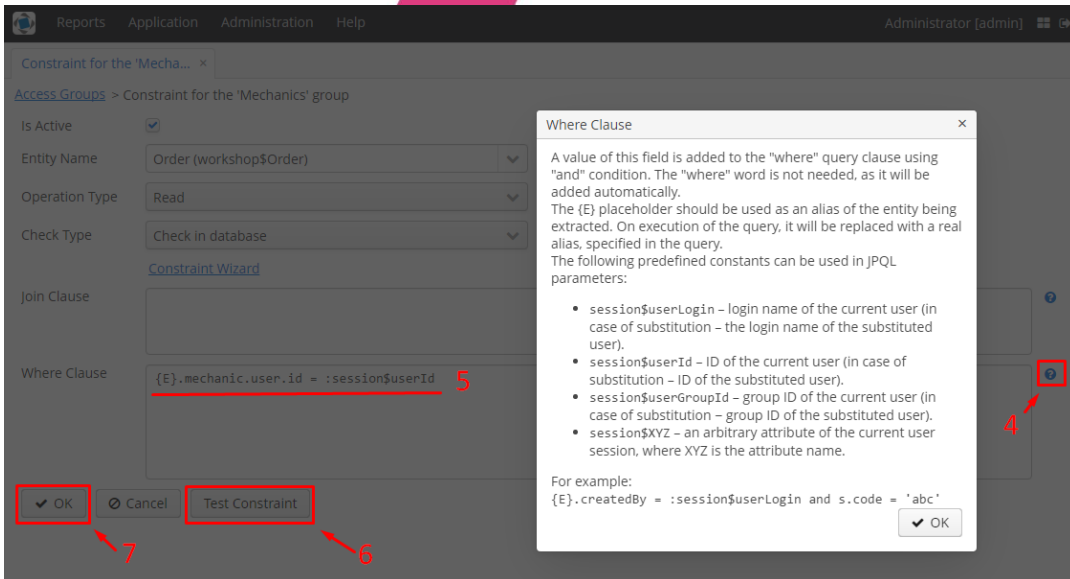
Add constraint for the access group

1. Open the **Constraints** tab for the newly created group
2. Click **Create** in the **Constraints** tab
3. Select Entity Name: ***workshop$Order***
4. We can define what operation type we want to assign restriction to. Keep **Operation Type: *Read***
5. Keep the ***Check in database*** check type to perform check on the database level, e.g. delete and update operations
6. Click the **Constraint Wizard** link



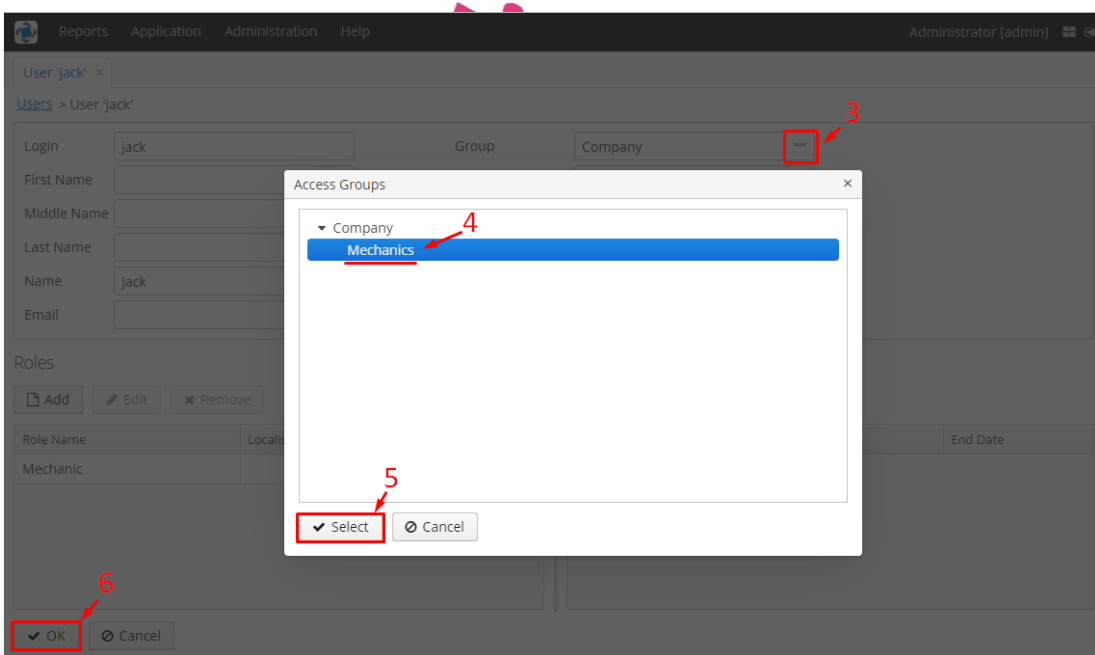
Add constraint for the access group

1. Click **Add**
2. Expand the **Mechanic** field and select the child **User** attribute
3. Click **OK** in the Define query window.
4. The platform has generated the where clause as *{E}.mechanic.user.id = 'NULL'*, where ***{E}*** is a generic alias for the entity
5. Click the question mark located in the top-right corner of the **Where Clause** field to see what predefined constants can be used here. Change *'NULL'* to *:session$userId*
6. Click **Test Constraint**
7. If test shows that constraint is valid, then click **OK**



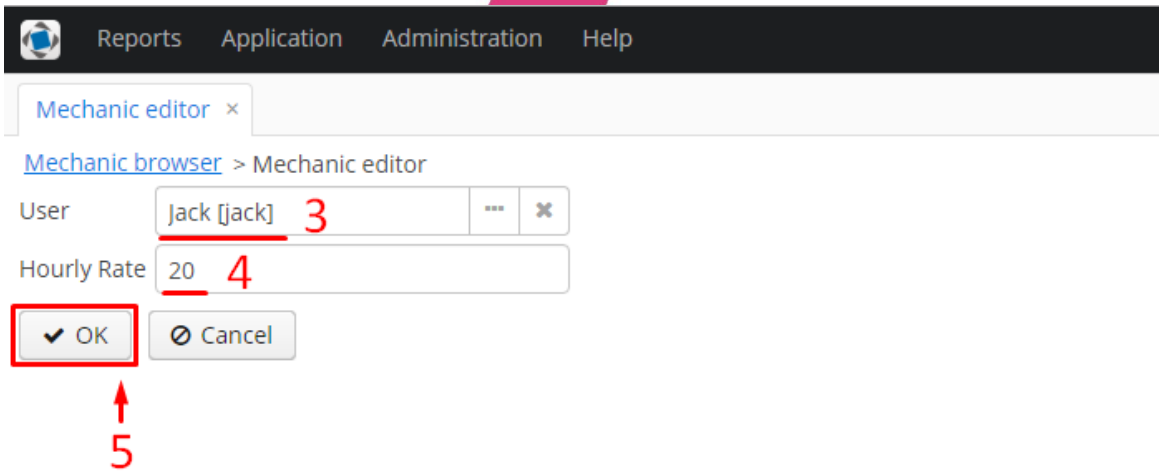
Assign group to the user

1. Open **Administration — Users** from the menu
2. Edit the user with login: ***jack***
3. Click on button **[...]** at the right of the **Group** field
4. Select the **Mechanics** group
5. Click **Select**
6. Click **OK** to save the user



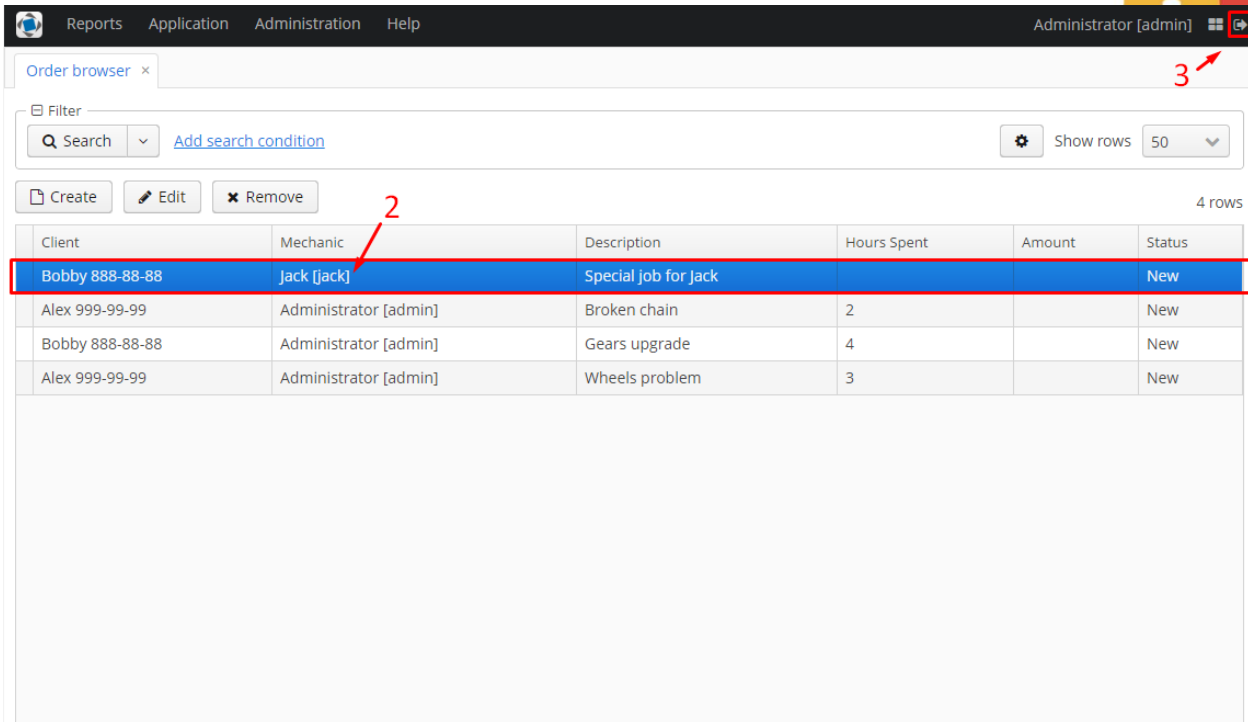
Create a mechanic for the user

1. Open **Application — Mechanics** from the menu
2. Click **Create**
3. Select user: ***jack***
4. Set **Hourly Rate**
5. Click **OK** to save the mechanic



Create an order for the mechanic

1. Open **Application — Orders** from the menu
2. Create an order for **Jack**
3. **Log out** of the system

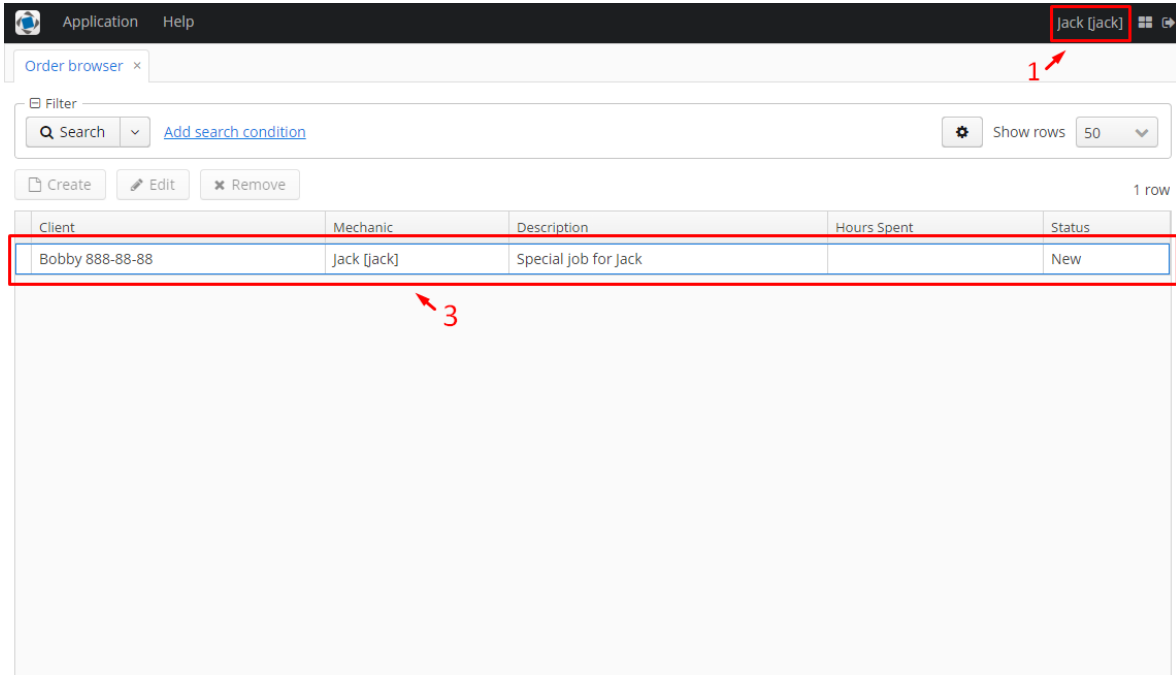


Row level security in action

1. Log in to the system as ***jack***
2. Open **Application — Orders** from the menu
3. We see only one order for Jack!

We have restricted access for particular orders only to the mechanics who perform them.

The access groups functionality allows you to configure the Row-level security in your application completely transparent for your application code without interfering with a screen code.





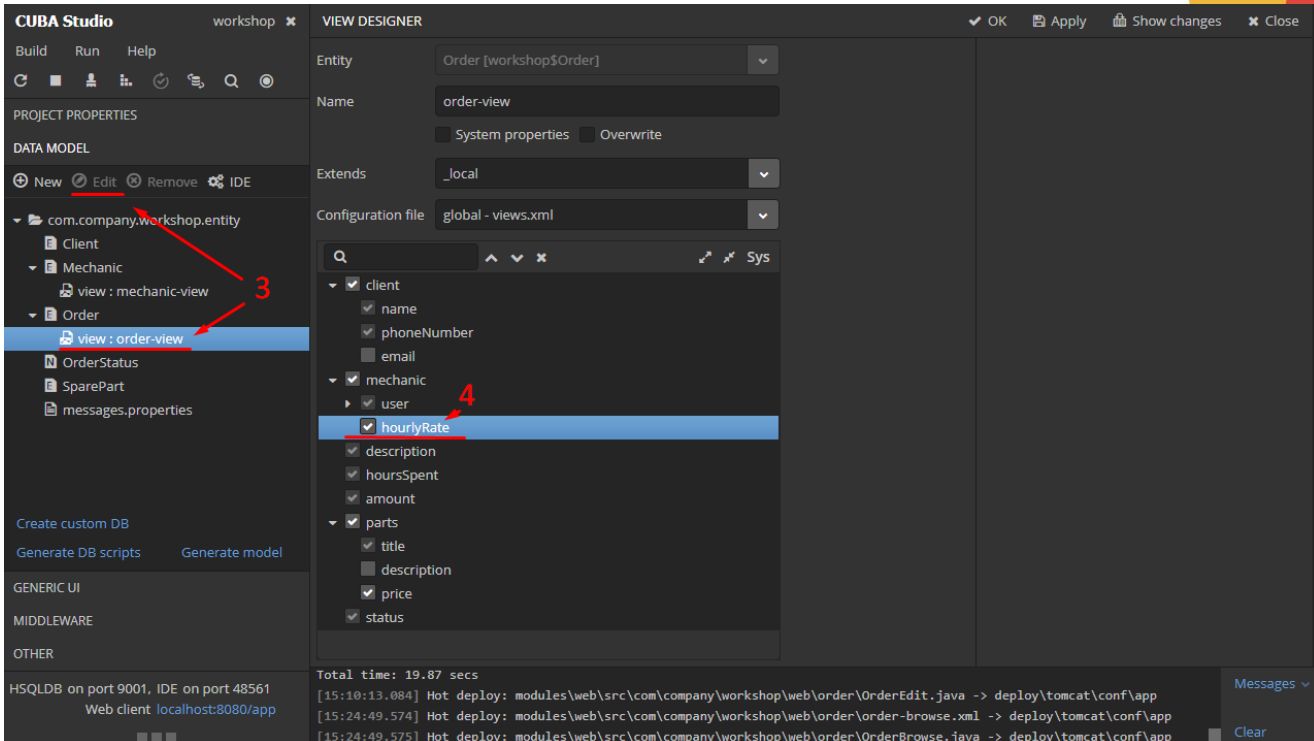

Services

Services

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.

To use mechanic hourly rate, we'll need to load this attribute, so we need to add it to **order-view**.

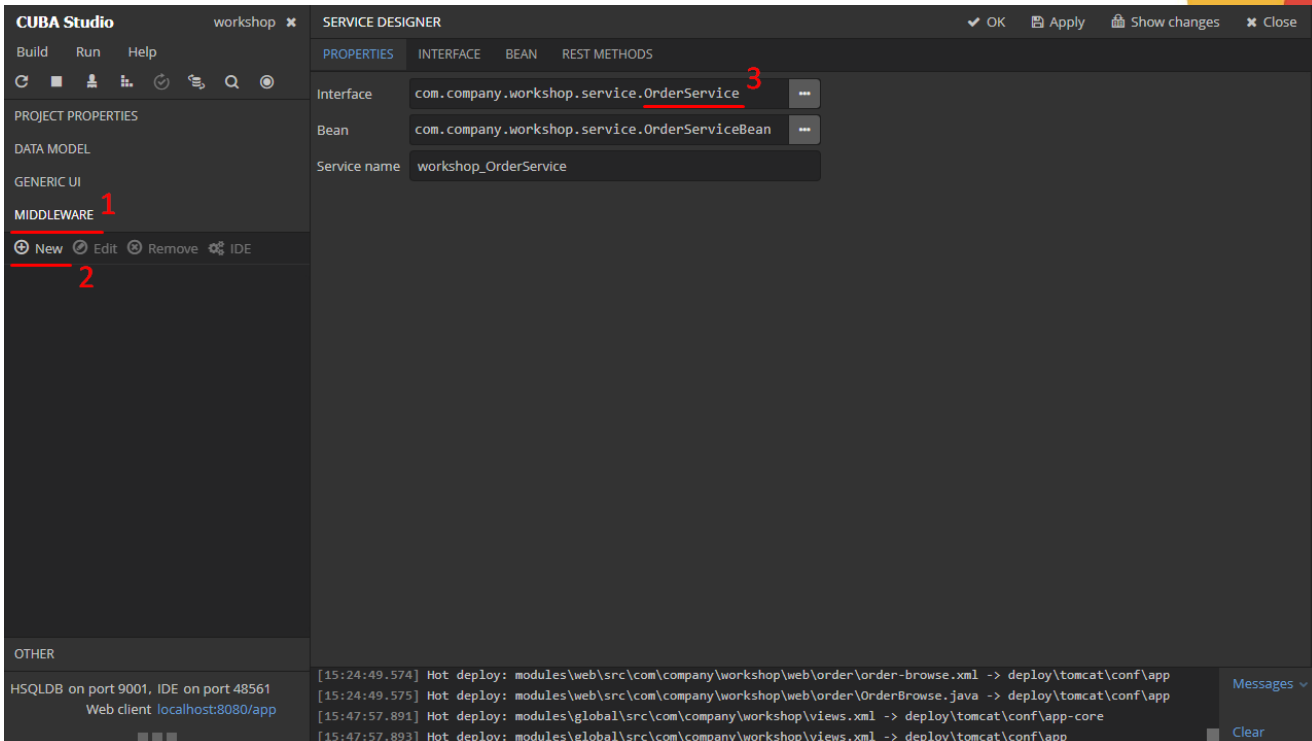
1. Switch to Studio
2. Open the DATA MODEL section of Studio navigation panel
3. Edit ***order-view***
4. Include the **hourlyRate** attribute to the view
5. Click **OK** to save the view



Generate Service stub

Business logic changes can happen very often, so it would be better to put it in a separate class - a service that different system parts will be able to invoke to calculate the price for repair. Let's create a stub for such service from Studio and implement the price calculation logic there. And on our screen, we'll create the method to invoke this service.

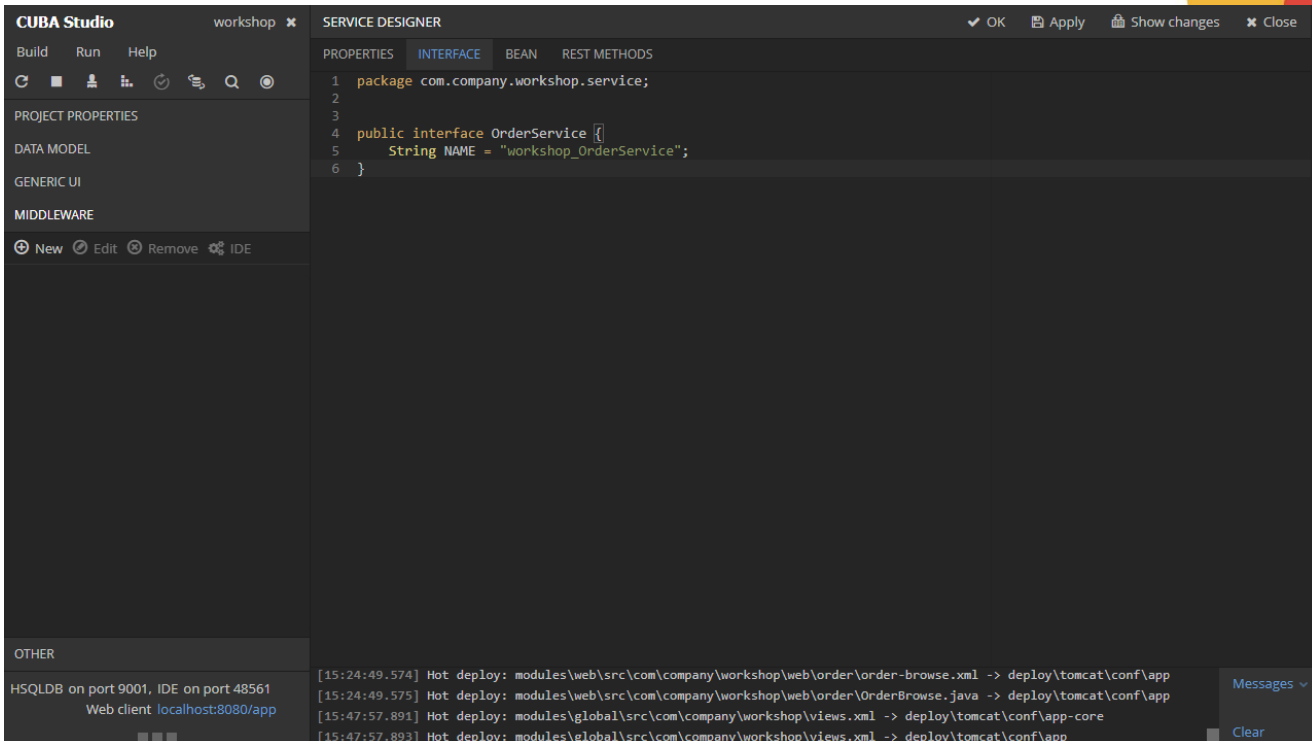
1. Go to the MIDDLEWARE section in Studio
2. Click **New -> Service**
3. Change the last part of Interface name to ***OrderService***



Service interface and bean generation

In the **Interface** tab we can see the source code of the service interface, the **Bean** tab shows its implementation. 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).

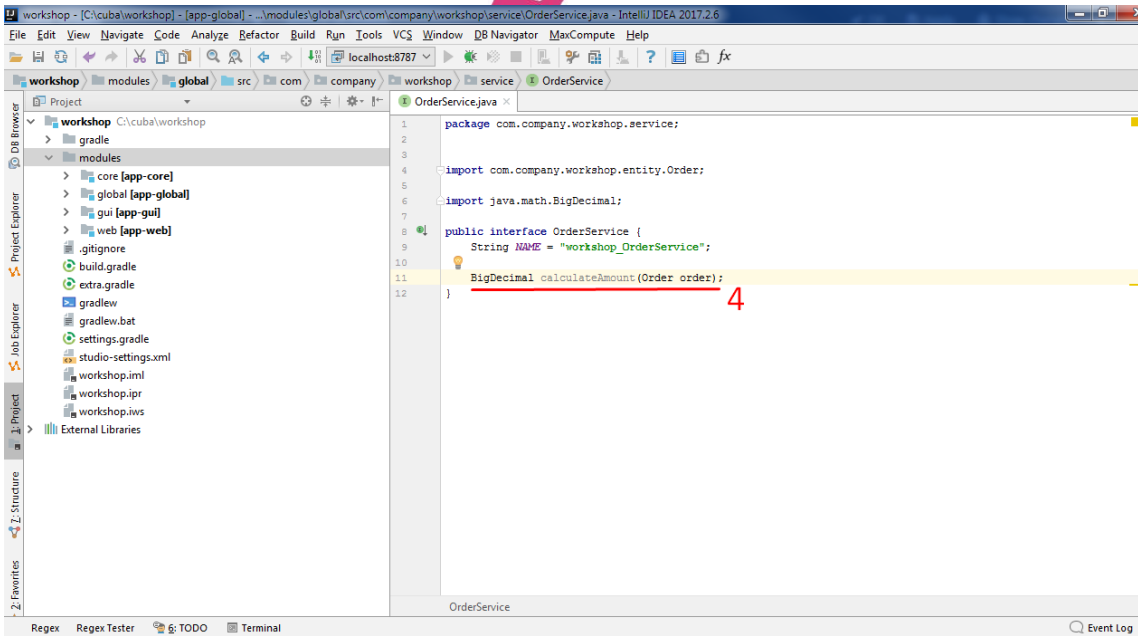


Add method to a service

1. Click **Apply** to save interface stub
2. Select the ***OrderService*** item in the **Interface** field
3. Click **IDE**
4. In the Intellij IDEA, we'll see the service interface, let's add the amount calculation method to it

BigDecimal calculateAmount(Order order)

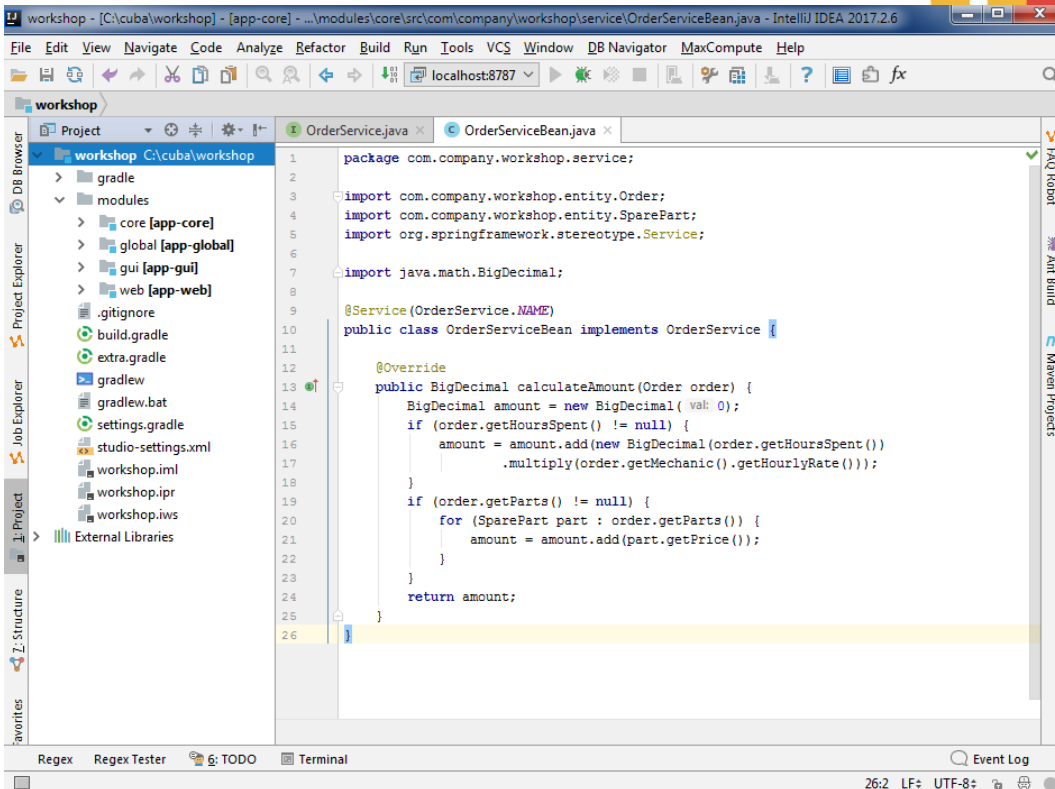
```
package com.company.workshop.service;

import com.company.workshop.entity.Order;
import java.math.BigDecimal;

public interface OrderService {
    String NAME = "workshop_OrderService";

    BigDecimal calculateAmount(Order order);
}
```



Service method implementation

1. Go to ***OrderServiceBean*** using the green navigation icon at the left
2. Implement the method

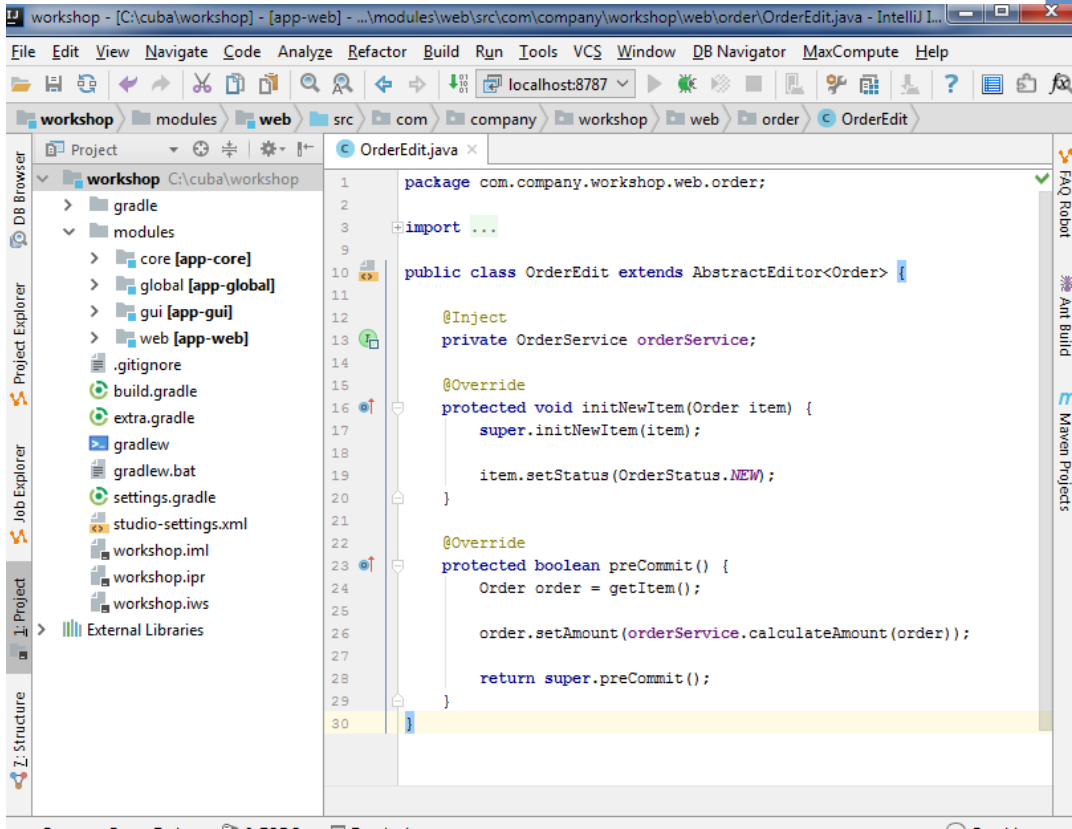
```
package com.company.workshop.service;

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;
    }
}
```



Call the service method from UI

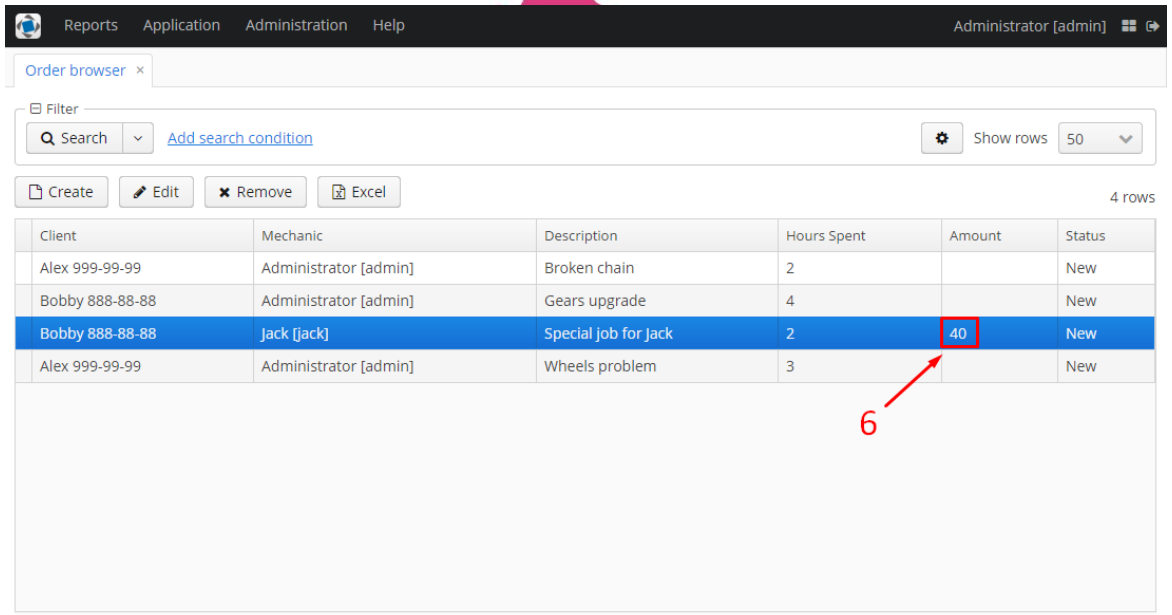
1. Go to the screen controller of the Order editor (***OrderEdit*** class)
2. Add ***OrderService*** field to class and annotate it with **@Inject** annotation
3. Override the ***preCommit()*** method and invoke the calculation method of ***OrderService***

```
public class OrderEdit extends AbstractEditor<Order> {
    @Inject
    private OrderService orderService;
    // ...
    @Override
    protected boolean preCommit() {
        Order order = getItem();

        order.setAmount(orderService.calculateAmount(order));

        return super.preCommit();
    }
}
```



Test the service call

1. Restart your application using the **Run — Restart application** action from Studio
2. Open **Application — Orders** from the menu
3. Open **editor screen** for any order
4. Set **Hours Spent**
5. Click **OK** to save order
6. We can see a newly calculated value of the amount in the table

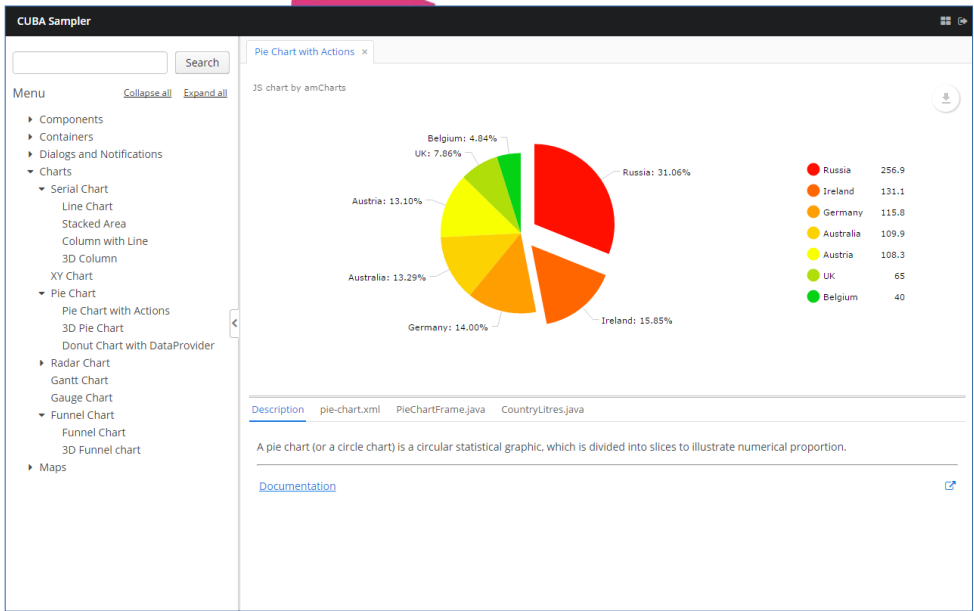


Charts

Charts

Let's assume our mechanic uses and likes the application but now he wants to add statistics. He wants a chart showing the number of orders per mechanic to reward them at the end of the month.

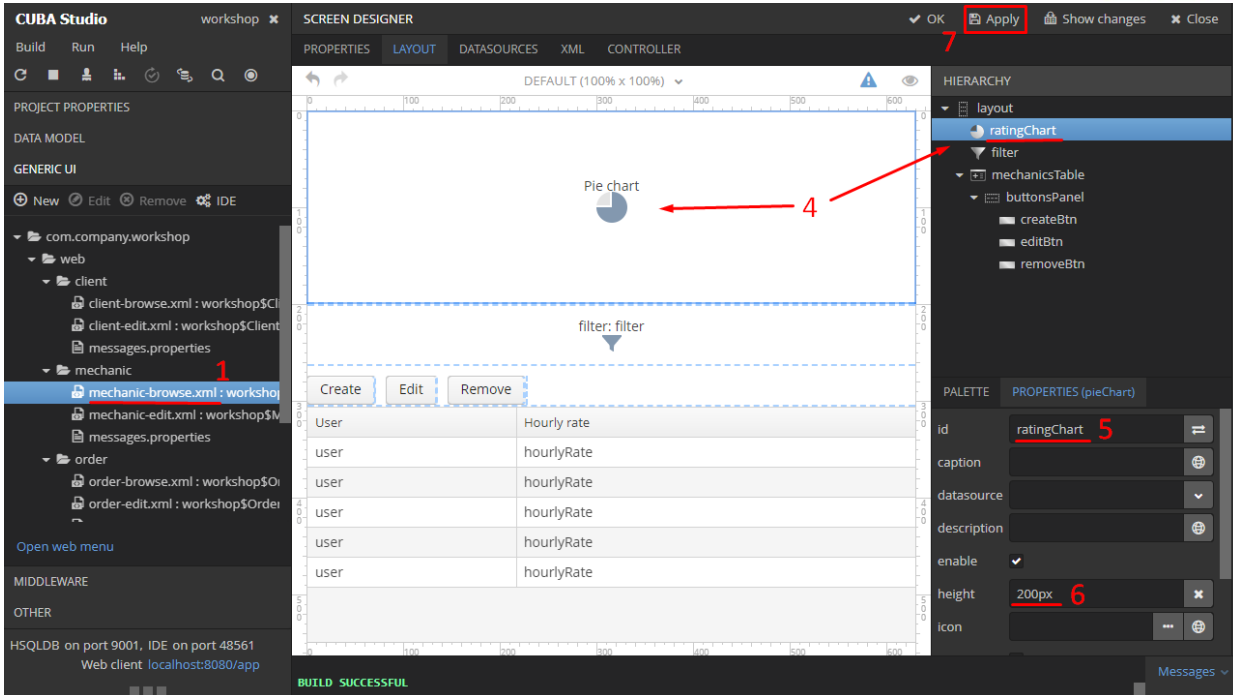
To implement this functionality we'll use the **charts** module of the CUBA platform, based on AmCharts. It allows us to display interactive charts in a web application based on system data and specify chart configuration via XML.



Add chart component to screen

Let's place the work distribution chart on the mechanics browser screen.

1. Open **mechanic-browse.xml** screen in Studio for editing
2. Place the cursor into the components palette, type ***Chart***
3. Studio will filter the component list and show us components to display charts
4. Drag **PieChart** and drop it to the UI editor area
5. Set id for chart: ***ratingChart***
6. Set width 100% and height 200px using **Properties** panel
7. Click **Apply** to save the screen



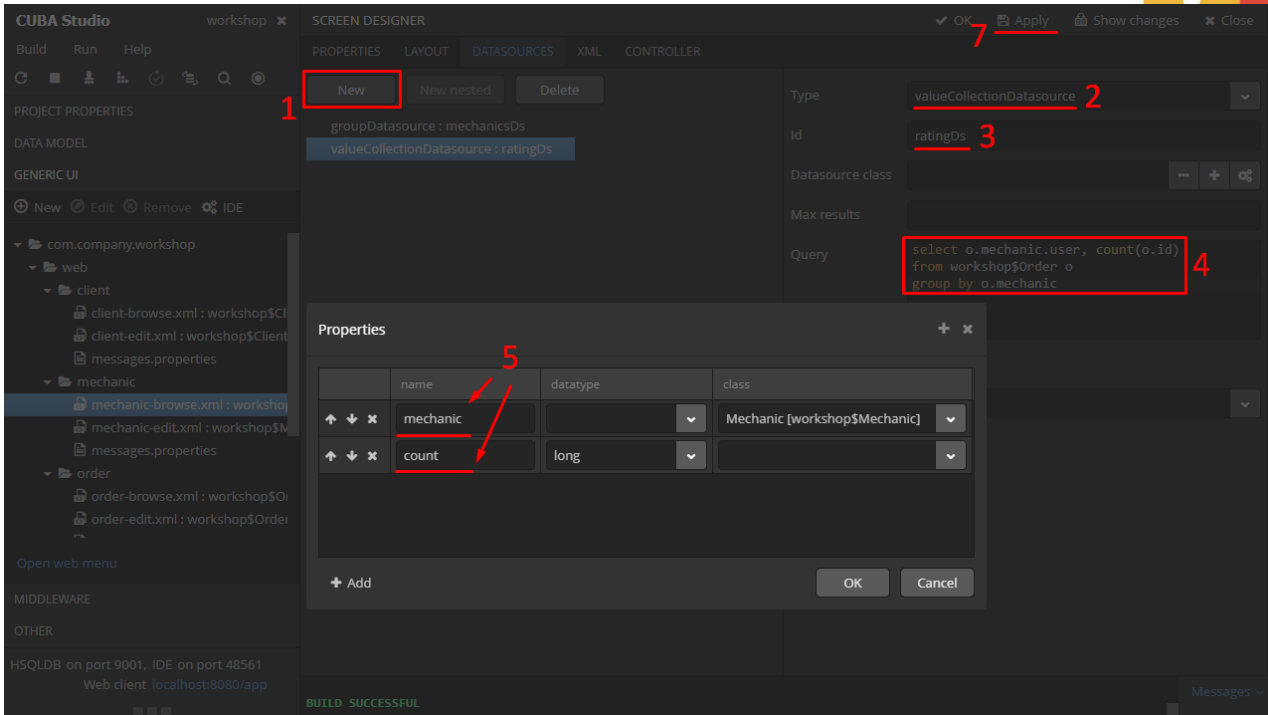
Load data for chart

To load data for our chart, let's create a new datasource of ValueCollectionDatasource type.

1. Open the DATASOURCES tab and click **New**
2. Select ***ValueCollectionDatasource*** type in the **Type** field
3. Set the data source ID
4. Enter the query to get the number of orders for each mechanic:

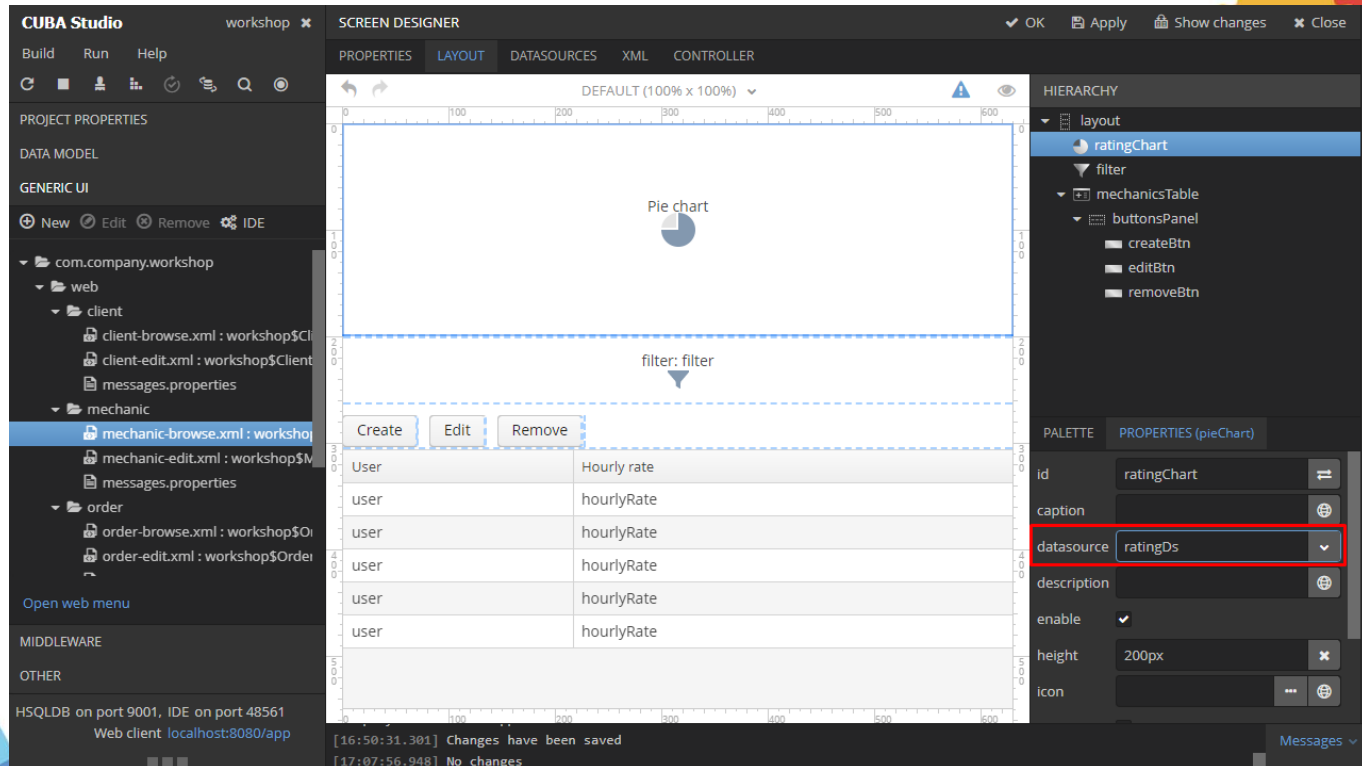
```
select o.mechanic.user, count(o.id)
from workshop$Order o
group by o.mechanic
```

5. Click **Edit** and add the following properties: **mechanic** (class = [workshop$Mechanic]) and **count** (datatype = long)
6. Click **OK**
7. **Apply** the **changes**



Load data for chart

Go back to the LAYOUT tab and set the newly created datasource for the chart.



Adding legend for chart

1. Go to the XML tab
2. Add **titleField** and **valueField** attributes to the chart element
3. Add legend to the chart: **<chart:legend position="LEFT"/>**
4. Click **OK** to save the changes

SCREEN DESIGNER

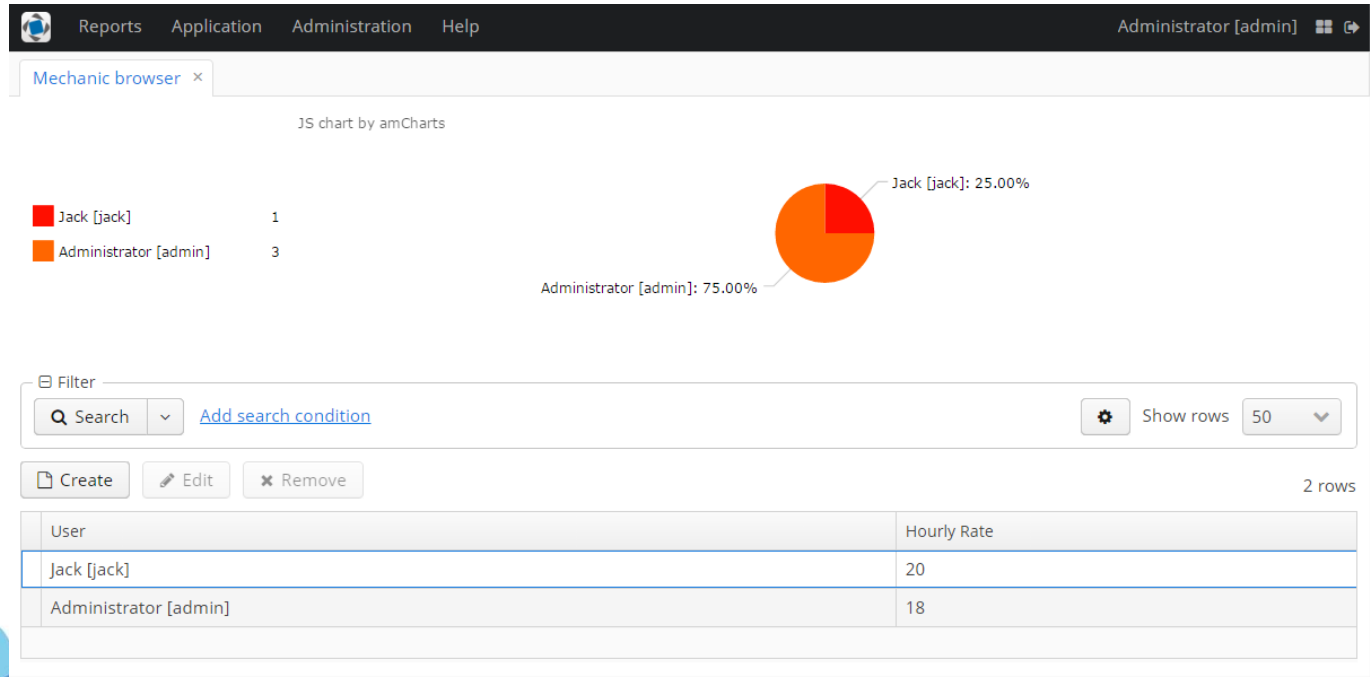
PROPERTIES | LAYOUT | DATASOURCES | XML | CONTROLLER

```
29              </properties>
30          </valueCollectionDatasource>
31      </dsContext>
32      <dialogMode height="600"
33                  width="800"/>
34      <layout expand="mechanicsTable"
35              spacing="true">
36          <chart:pieChart id="ratingChart"
37                          datasource="ratingDs"
38                          height="200px"
39                          titleField="mechanic"
40                          valueField="count"
41                          width="100%">
42              <chart:legend position="LEFT"/>
43          </chart:pieChart>
44          <filter id="filter"
45                  applyTo="mechanicsTable"
46                  datasource="mechanicsDs">
47              <properties include=".*"/>
48          </filter>
```

Open screen with chart

Reopen **Application — Mechanics** mechanic browser

Now we know exactly who should get a bonus.



Reporting

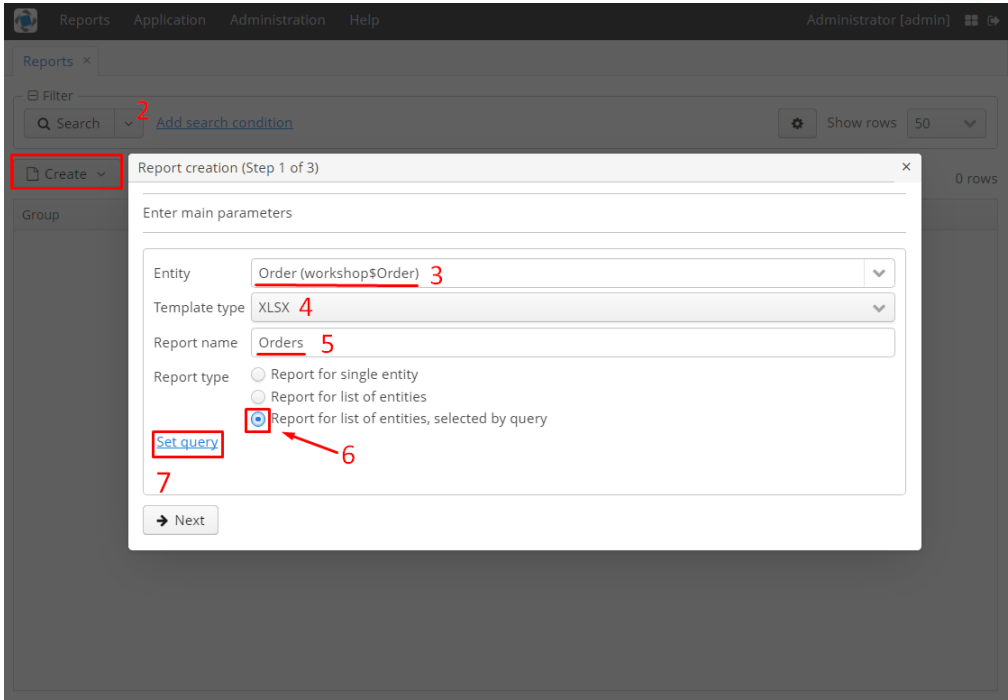
Reports

A rare business application goes without reports. That's why our mechanic has asked us to make a report, showing undertaken work for a certain period of time.

1. Open Reports — Reports from the menu
2. Click Create — Using wizard
3. Select Entity: ***Order*** (***workshop$Order***)
4. Set Template type: ***XLSX***
5. Set Report Name: ***Orders***
6. Select Report type:

Report for list of entities by query

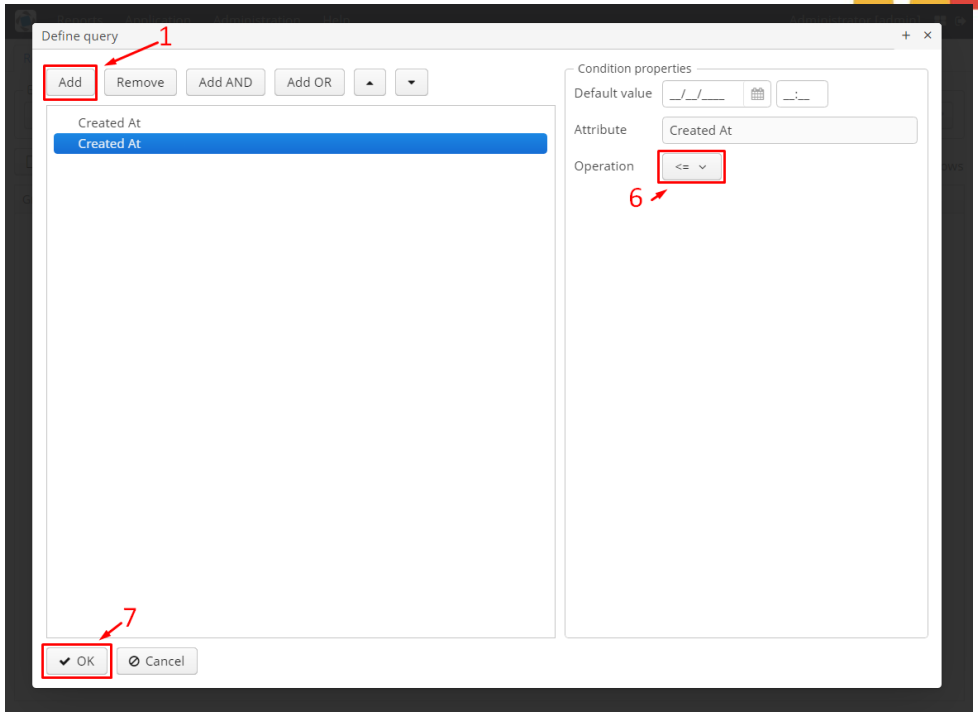
7. Click Set query



Report query builder

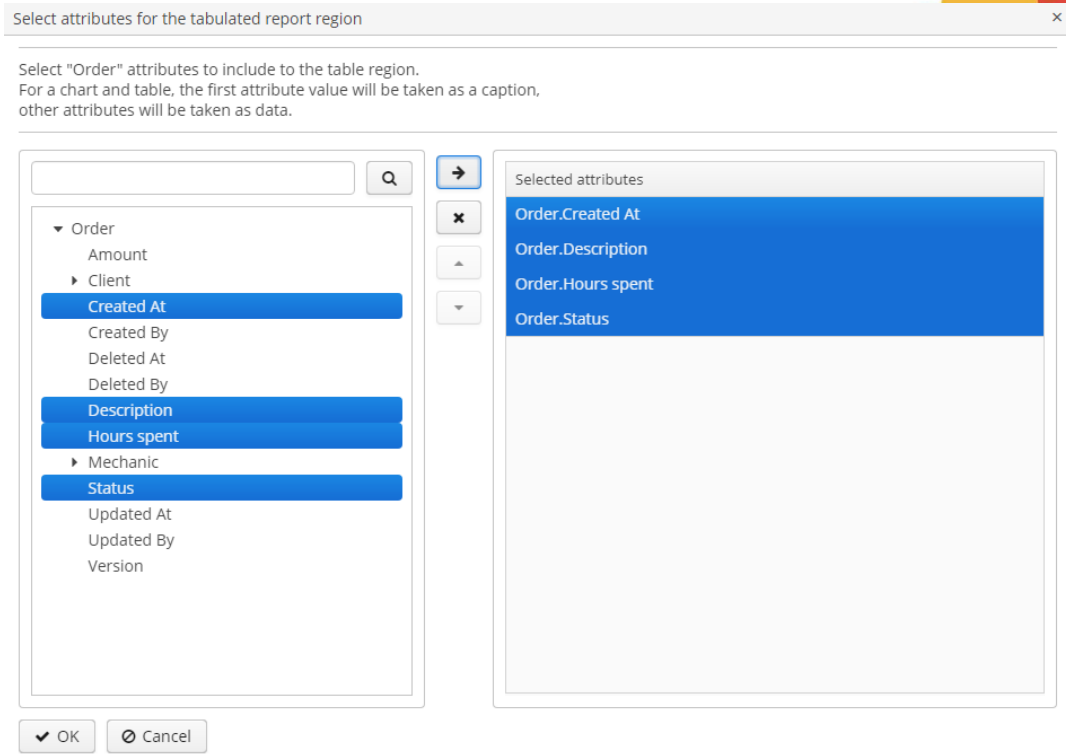
Report Wizard allows us to create a query using the graphical expressions constructor.

1. Click **Add**
2. Select the **Created at** attribute
3. Change operation for created condition to **[>=]**
4. Click **Add** once again
5. Select the **Created at** attribute
6. Change operation for created condition to **[<=]**
7. Click **OK**
8. Click **Next**



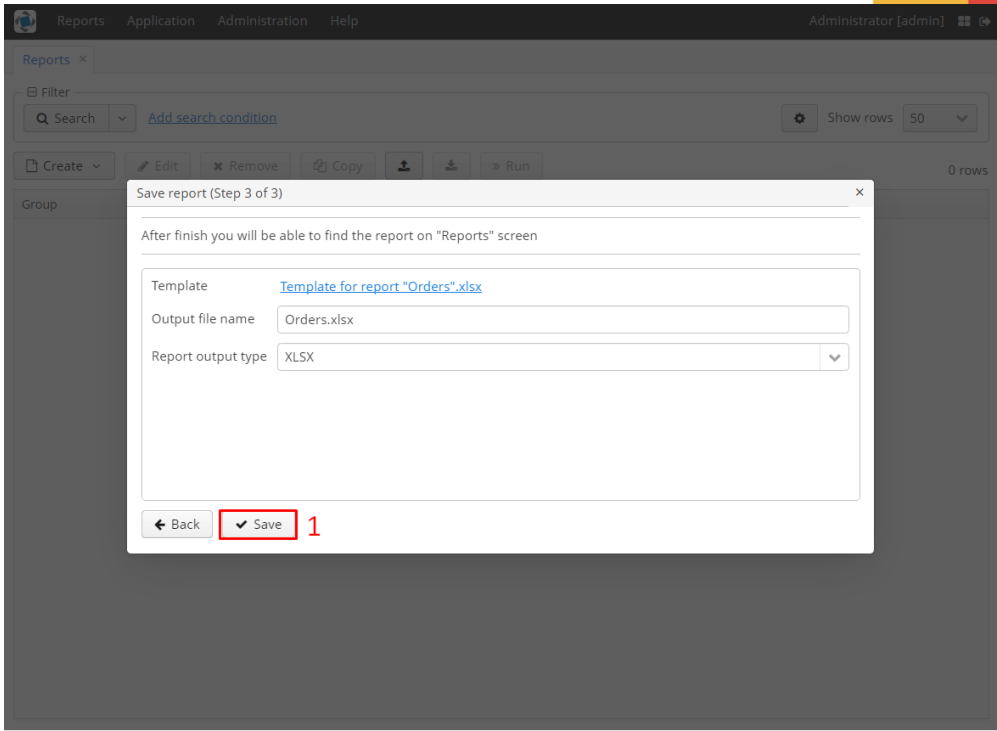
Select attributes for report

1. Select ***Order*** attributes that the report will contain: ***Created At***, ***Description***, ***Hours Spent***, ***Status***
2. Click **OK**
3. Click **Next**



Save report

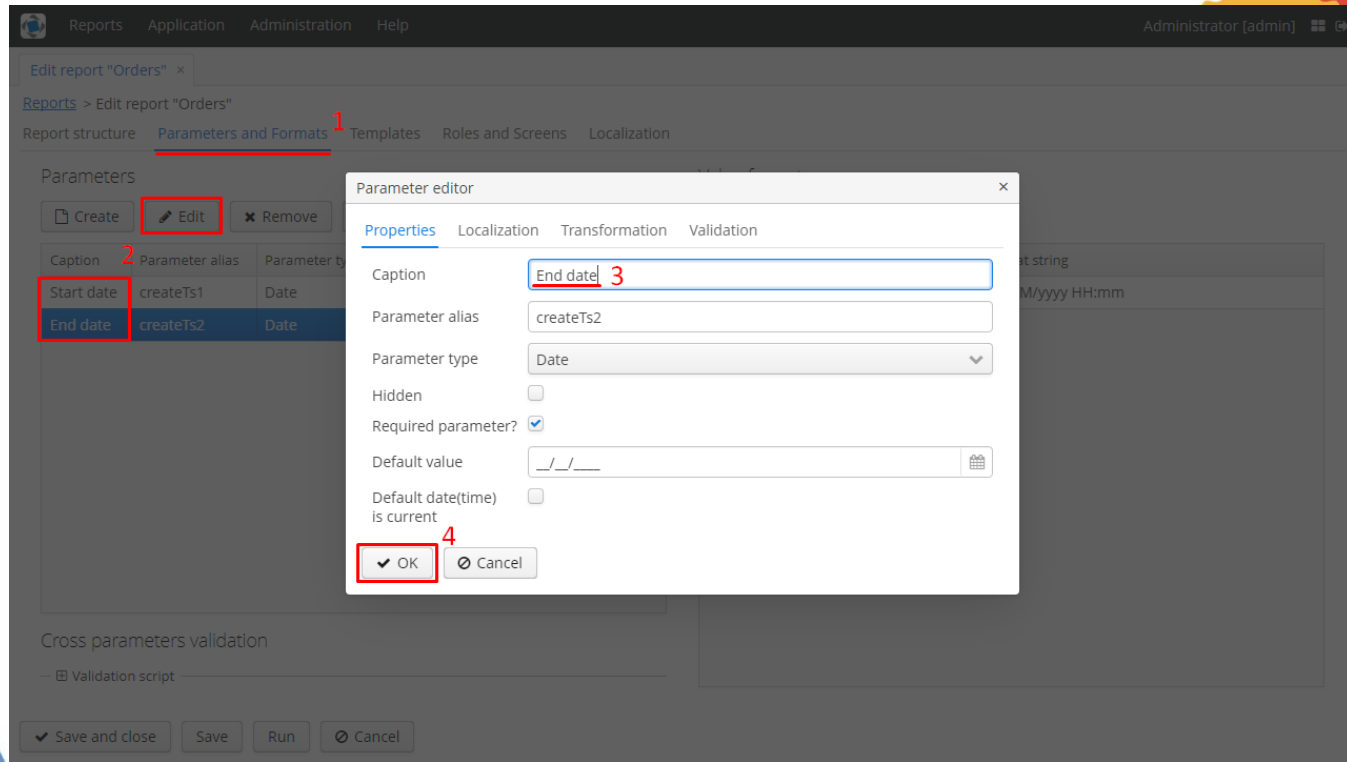
1. Click **Save** to save the report



Change parameter names

The **Wizard** will open the report editor so that we can make additional changes if needed.

1. Open **Parameters and Formats** tab
2. Edit the ***CreateTs1*** parameter
3. Set Parameter Name: ***Start date***
4. Click **OK**
5. Edit the ***CreateTs2*** parameter
6. Set Parameter Name: ***End date***
7. Click **OK**

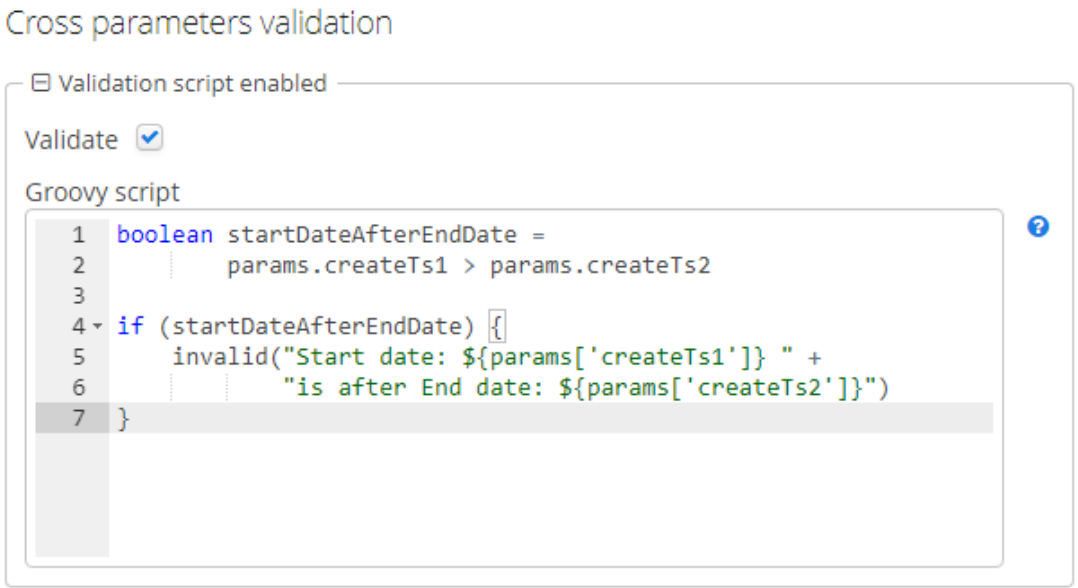


Set up cross parameter validation

It would be a good idea to anticipate the incorrect user input of report parameters. So, let's create a Groovy script that should check whether parameter values make sense in relation to each other.

1. Expand **Cross parameters validation** section on the **Parameters and Formats** tab of the report editor.
2. Enter the script
3. Check the **Validate** checkbox
4. Click **Save and close**

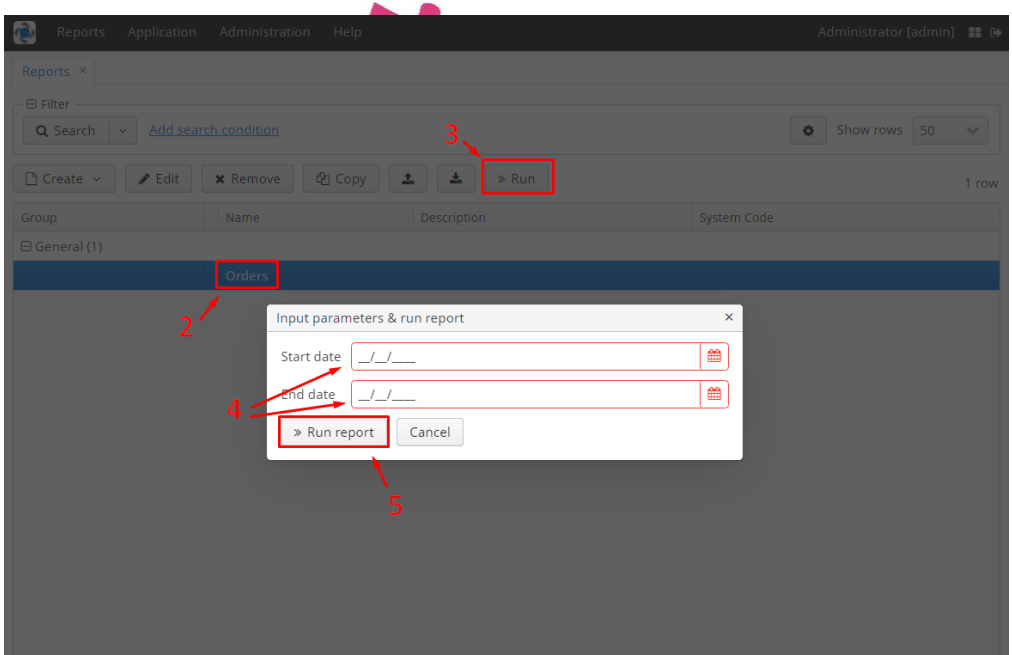
```
boolean startDateAfterEndDate =
        params.createTs1 > params.createTs2

if (startDateAfterEndDate) {
    invalid("Start date: " +
            "${params['createTs1']}"+
            "is after End date:" +
            "${params['createTs2']}")
}
```



Run report

1. Expand **General** report group
2. Select the report
3. Click **Run**
4. Enter **Start date** and **End date**
5. Click **Run report**

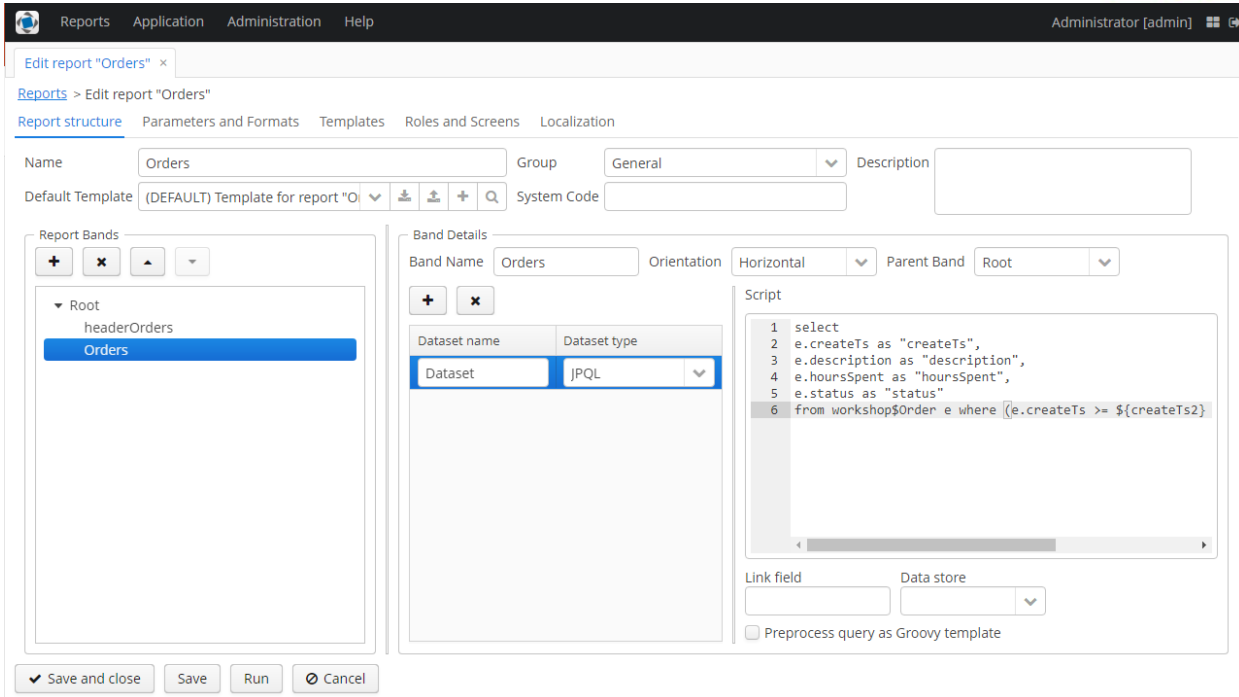
The system has generated an **XSLX file**, we can download it and view its content. Due to the fact that the report templates have the same format as the one that is required for the output, we can easily prepare templates from customer's report examples.



Report editor

You can also create reports manually using the ***Report editor***. Data can be extracted via ***SQL***, ***JPQL*** or even ***Groovy*** scripts. The template is created in ***XLS(X)***, ***DOC(X)***, ***CSV***, ***HTML***, ***JasperReports*** formats using standard tools. Report output can be converted to ***PDF***.

Also, using the **Report editor** you can specify users who will have access to the report, and system screens where it should appear.

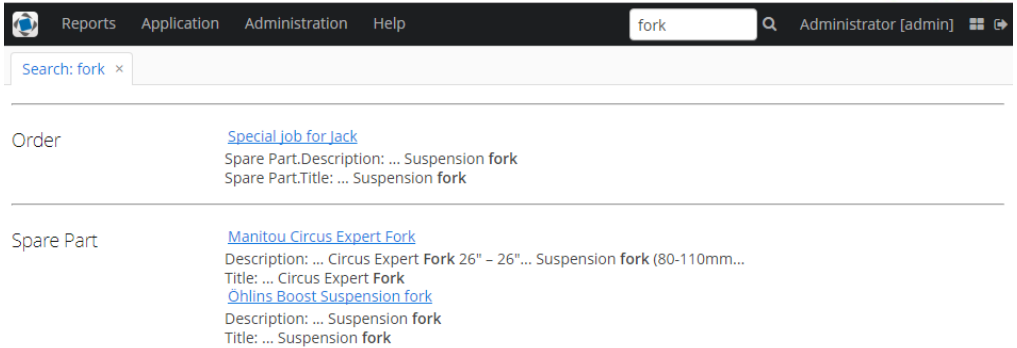


Full-Text Search

Full-Text Search

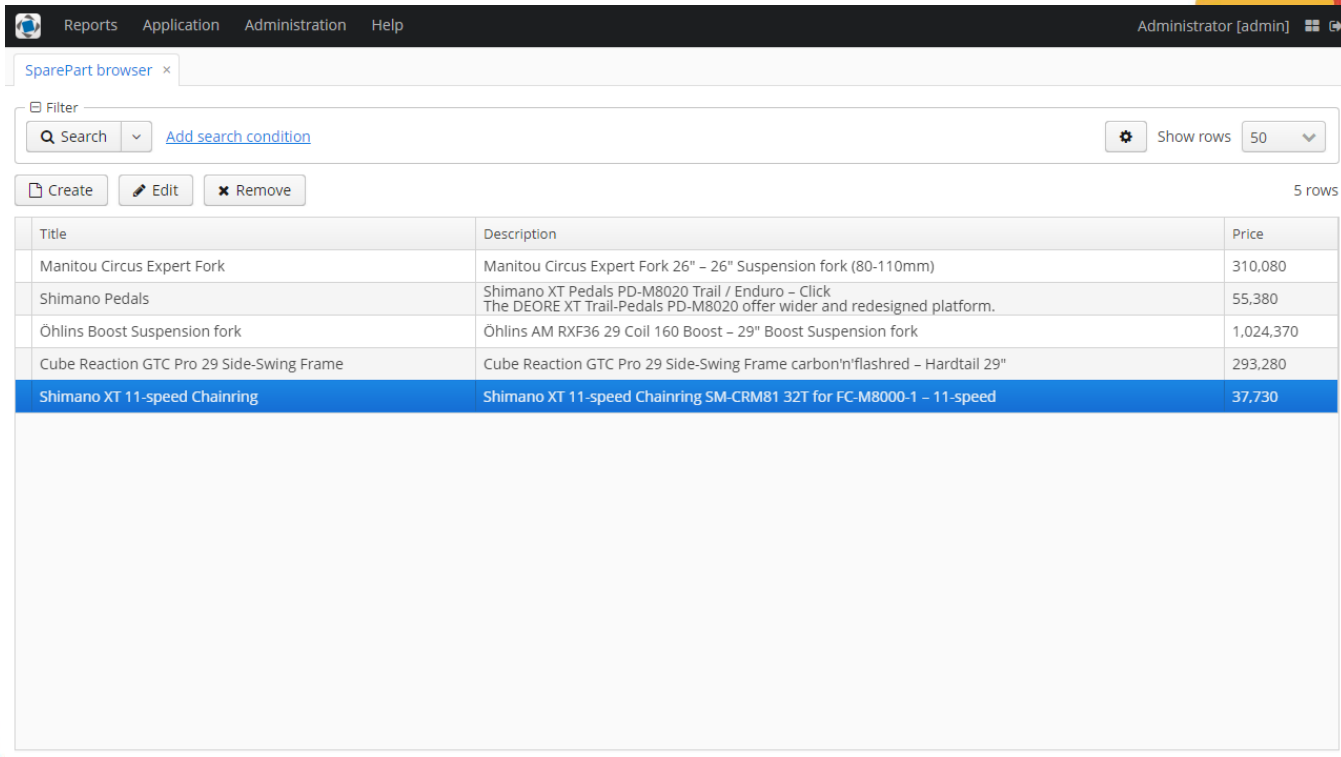
Our system stores information about spare parts, but there are quite a few of them. It would be useful to search them simply by typing a string like we google in a browser.

The CUBA Platform includes the **Full-Text Search** module based on Apache Lucene. It indexes content, including files of different formats, and enables text search using this index.
Search results are filtered according to security constraints.



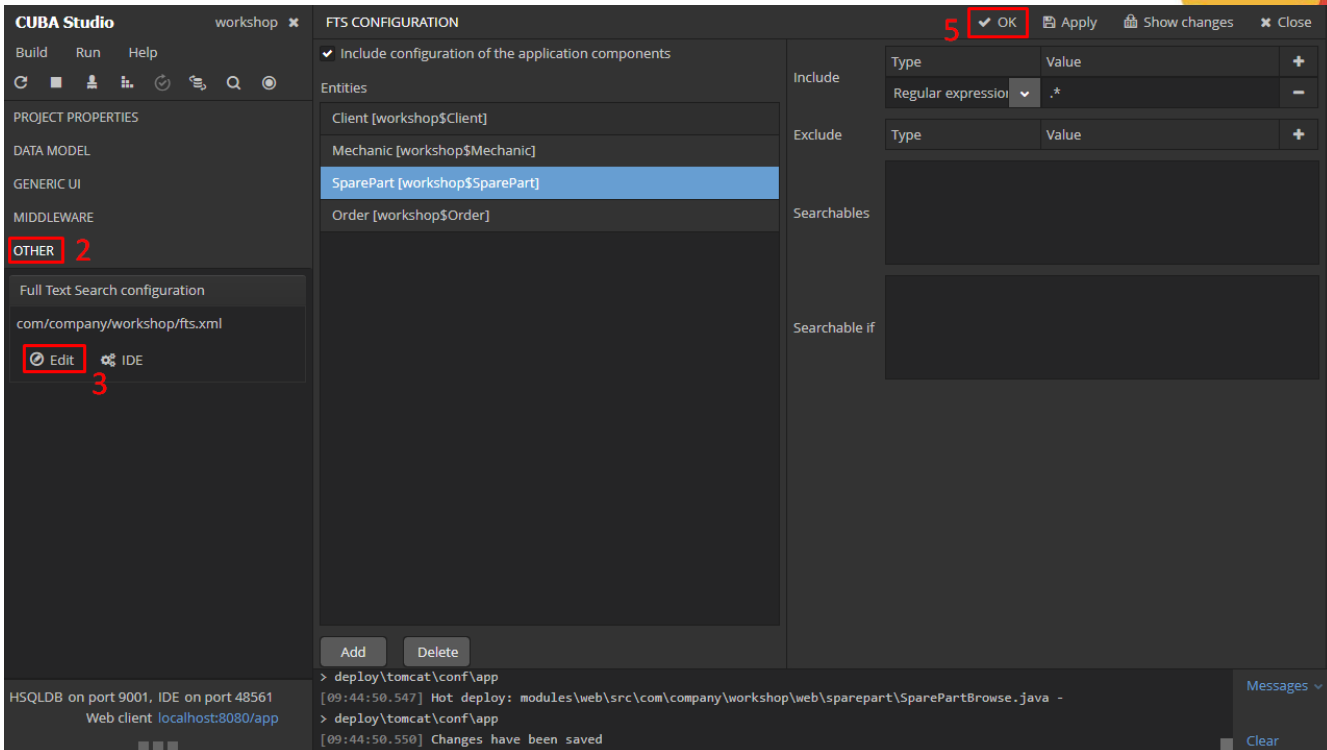
Adding spare parts

1. Open **Application — Spare Parts** from the menu
2. Add some spare parts
3. Use these spare parts in random orders



Configure Full-Text Search Index

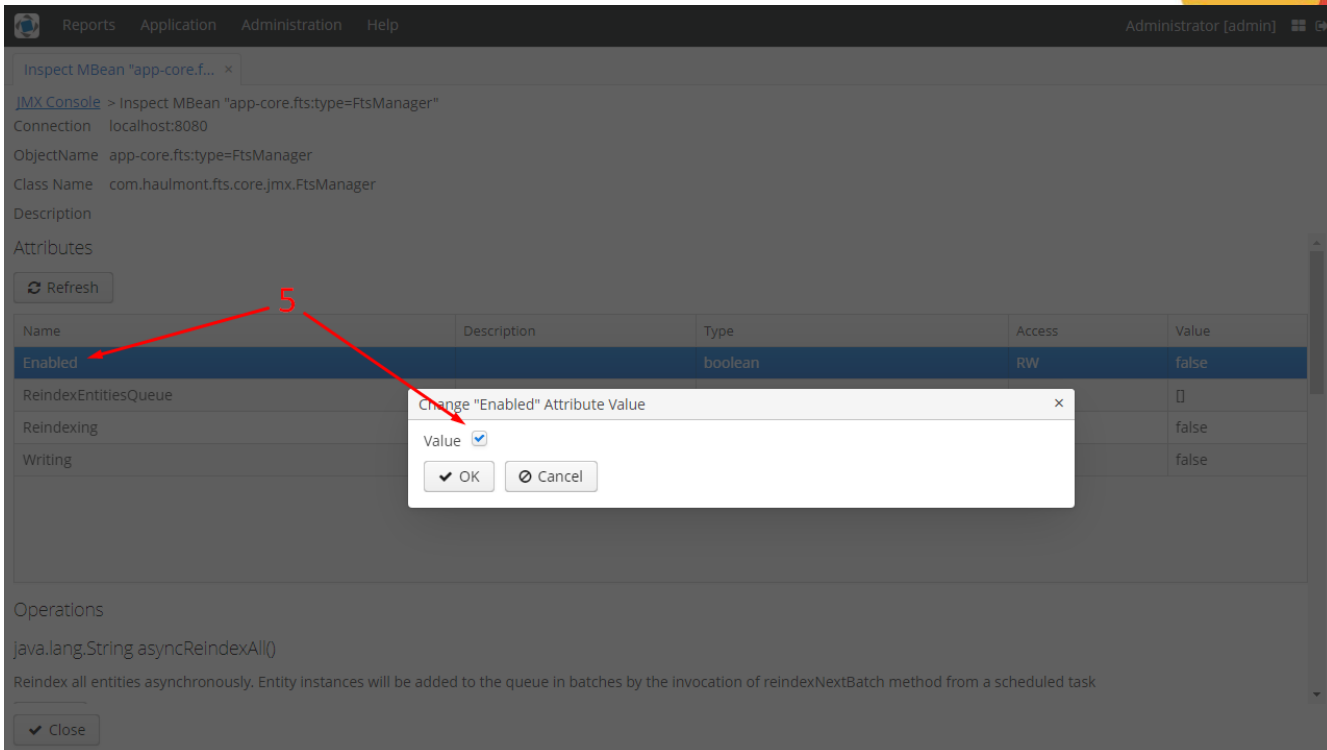
1. Switch back to Studio
2. Go to **Others** section of the navigation panel
3. Click **Edit** for **Full-Text Search configuration**
4. By default, Studio has added all our entities to the index configuration.
From this screen we can manage entities and fields that will be indexed
5. Click **OK**



Enable Full-Text Search for the application

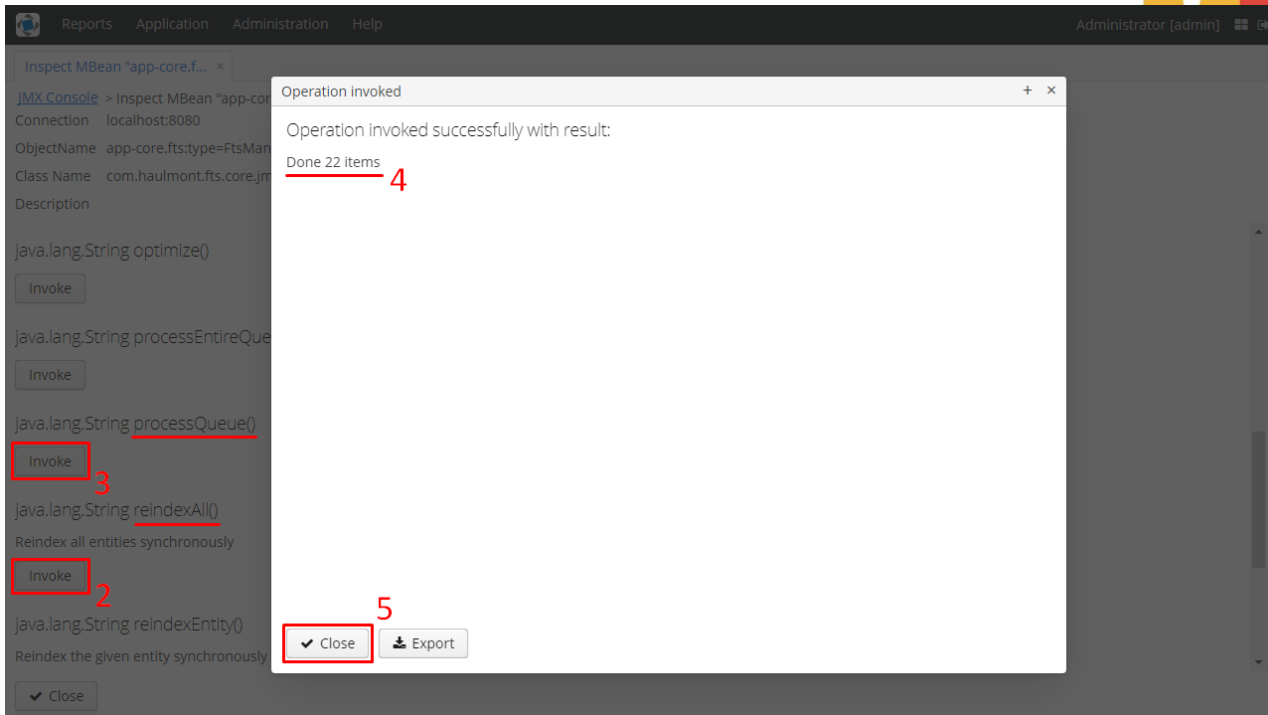
The further configuration will be done via the CUBA interface.

1. Open **Administration — JMX Console** from the menu
2. This is a web version of the console for the JMX interface; it allows us to manage internal system mechanisms
3. Find ***FtsManager*** using the **Search by ObjectName** field
4. Open ***FtsManager***
5. Change the **Enabled** property to true



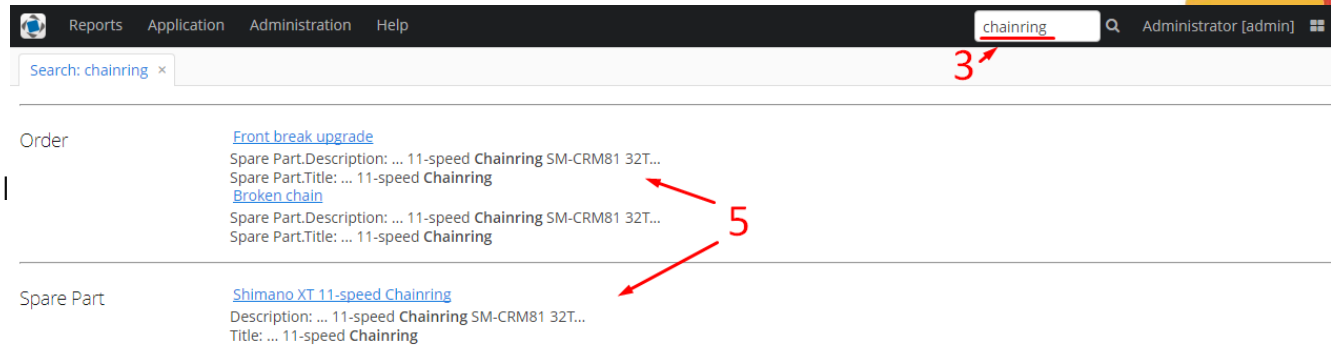
Add records to index

1. Scroll down to see ***reindexAll*** and ***processQueue*** methods of ***FtsManager***
2. Invoke the ***FtsManager reindexAll()*** method
3. Invoke the ***FtsManager processQueue()*** method
4. The system will display the current number of indexed records
5. Click **Close**



FTS in action

1. **Log out** of the system
2. **Log in** again
3. In the application top panel, the **search field** will appear, allowing you to search through all added to FTS objects.
4. Let's find something, for example: ***chainring.***
5. You will see the screen with search results, which contains not only spare parts but also orders that have spare parts with this word in its name

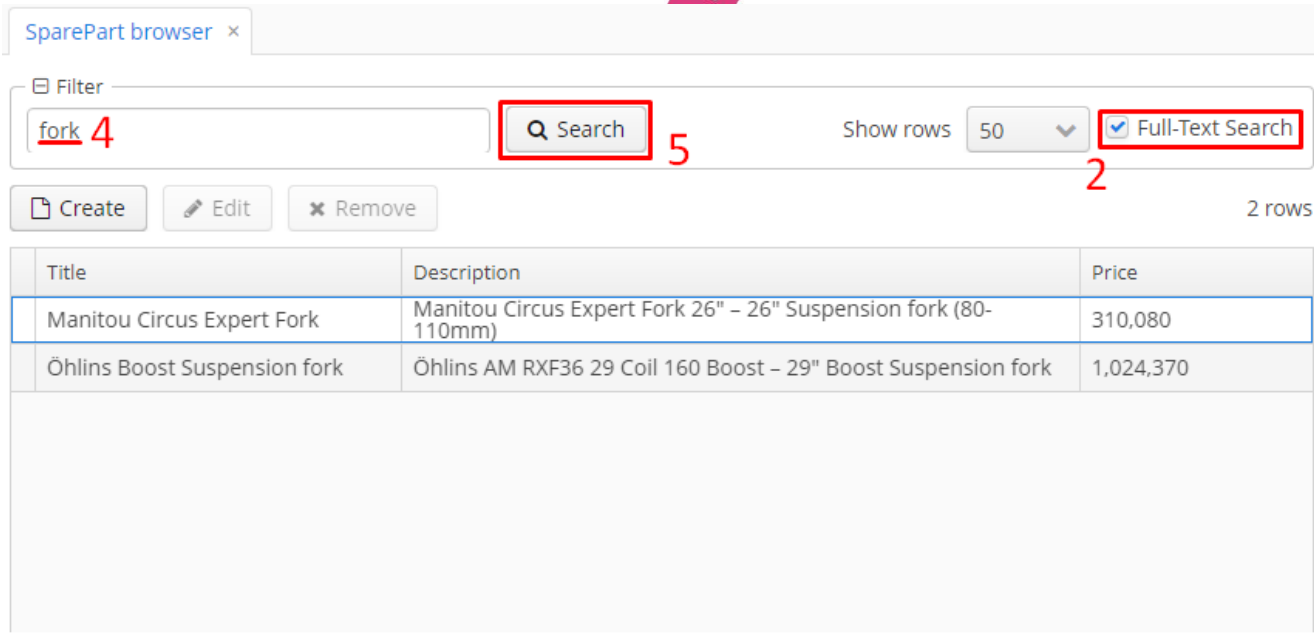


FTS integration with filters

But what if we want to search only for spare parts?

1. Open **Application — Spare Parts** from the menu
2. Select **Full-Text Search** checkbox in the filter panel
3. The text field will appear
4. Let's enter something, for example: ***fork***
5. Click **Search**
6. The table will display records that contain ***fork*** in their ***description***

So, now mechanics will be able to find spare parts by description quickly

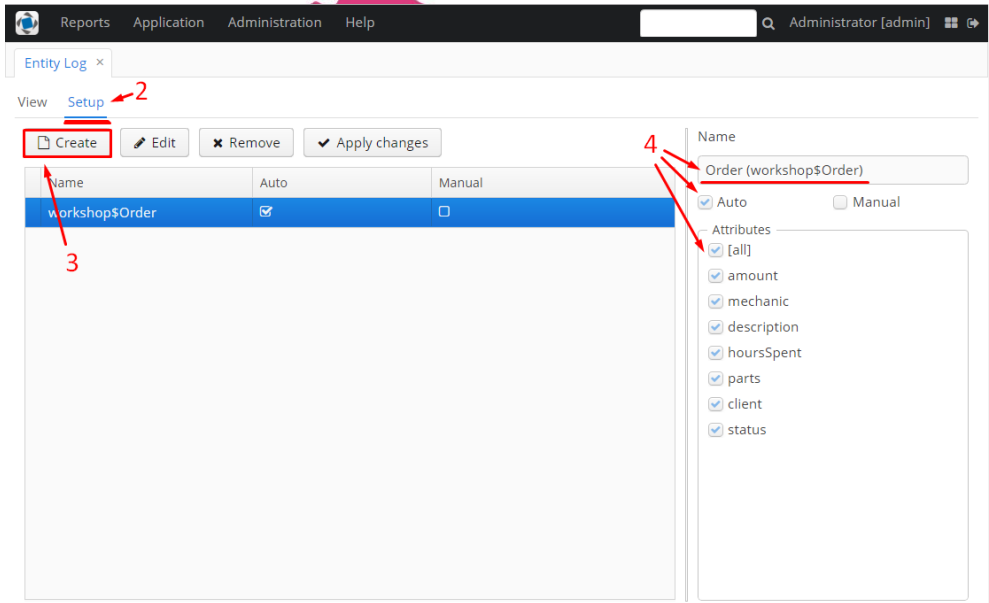


Audit

Audit

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. CUBA has a built-in mechanism to track entity changes, which you can configure to track operations with critical system data.

Let's apply it for logging order changes.



1. Open **Administration — Entity log** from the menu
2. Go to the **Setup** tab
3. Click **Create**
4. Set **Name**: ***Order*** (***workshop$Order***)

Auto: **true**

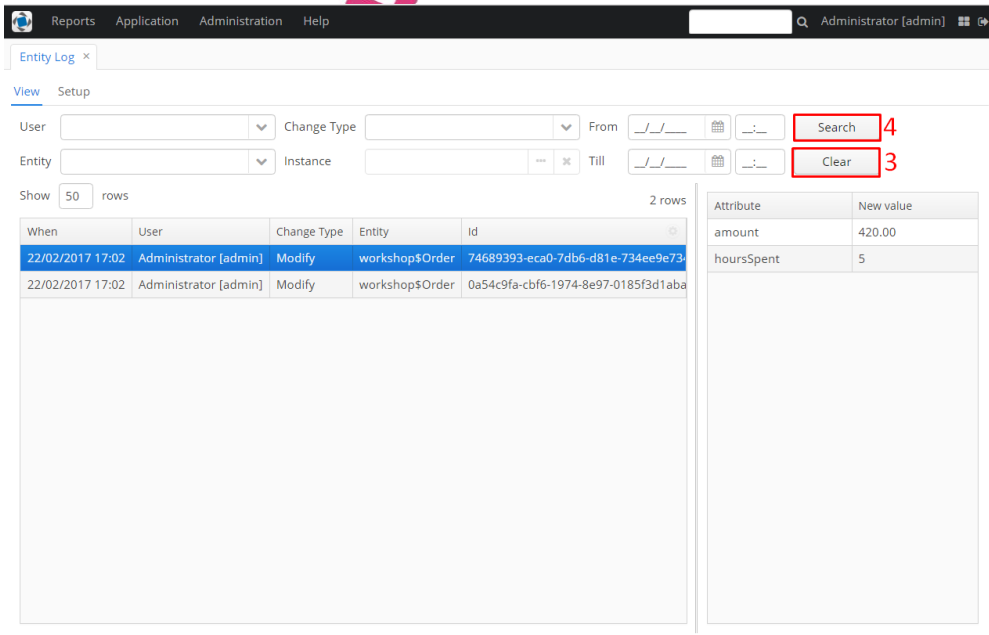
Attributes: ***all***

5. Click **Save**

Audit in action

1. Let's change an order description (or even clean it up)
2. Go back to the **View** tab of **Administration — Entity Log** page
3. Click **Clear** to reset security records filter
4. Click **Search**

The table shows changes and the user that made them, the changed fields and their new values. By sorting the changes by date and filtering them for a particular record, we'll be able to restore the entire chronology of events.



REST API

Generic 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 mobile application for our customers. For this purpose, CUBA provides developers with the generic REST API.

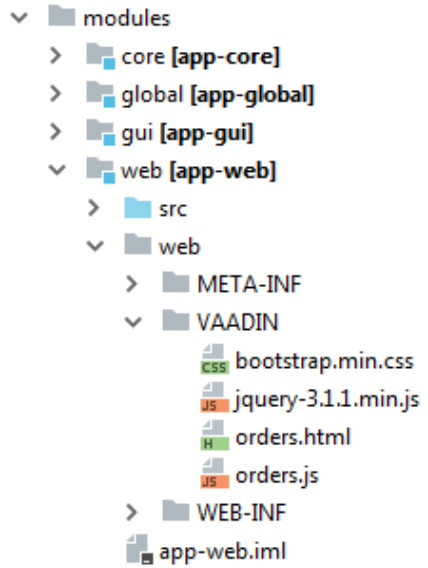
REST API is enabled by default in the web client, thus we can use it right now for a simple web page that will show the list of recent orders for a logged in user.

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

1. Create **orders.html** and **orders.js** in the web module:

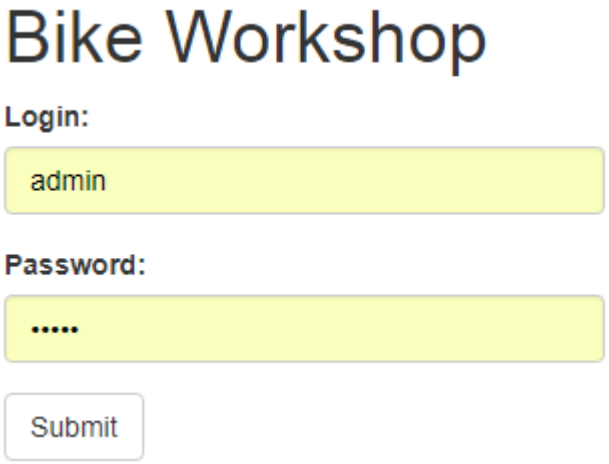
/modules/web/web/VAADIN/orders.html

2. Add *jquery-3.1.1.min.js* and *bootstrap.min.css* prerequisites.



REST API: orders.html

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:




```
<!DOCTYPE html>
<html lang="en">
<head>
    <script type="text/javascript" src="jquery-3.1.1.min.js"></script>
    <link rel="stylesheet" href="bootstrap.min.css"/>
    <script type="text/javascript" src="orders.js"></script>
</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 id="recentOrders" style="display: none">
        <h2>Orders</h2>
        <table id="ordersList" width="300">
            <tr>
                <th>Description</th>
                <th>Amount</th>
            </tr>
        </table>
    </div>
</div>
</body>
</html>
```

REST API: orders.js

We will send login request according to OAuth protocol:

```
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();
        }
    })
}
```

Add method to OrderService

We already have a *calculateAmount()* method in *OrderService,* but it requires an Order instance as a parameter.

As far as we will get only the order ID by the GET request, let's create another *calculateAmount* implementation with UUID orderId as a parameter:

BigDecimal calculateAmount(UUID orderId);

```
public interface OrderService {
    String NAME = "workshop_OrderService";

    BigDecimal calculateAmount(Order order);

    BigDecimal calculateAmount(UUID orderId);

}
```

Implementation in OrderServiceBean

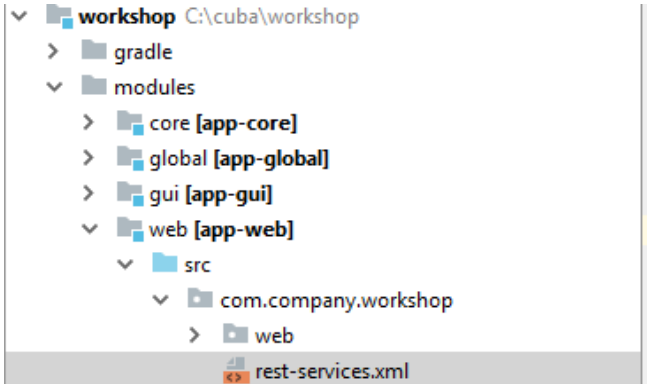
This newly created method will recover an order instance from its UUID and invoke the first *calculateAmount* method to return the BigDecimal value of the order amount.

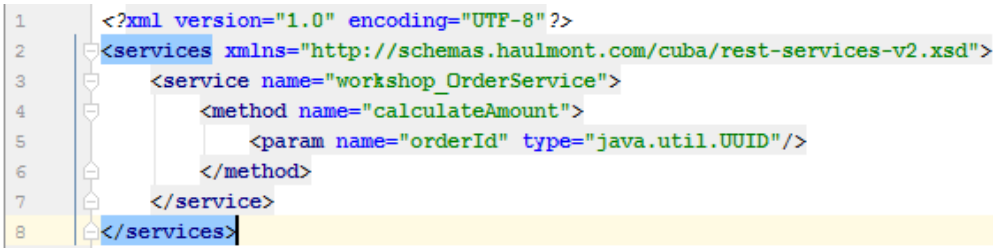
```
@Override
public BigDecimal calculateAmount(UUID orderId) {
    LoadContext<Order> context = LoadContext.create(Order.class)
            .setQuery(LoadContext.createQuery("select o from workshop$Order o where o.id = :orderId")
                    .setParameter("orderId", orderId)).setView("order-view");
    Order order = dataManager.load(context);
    return calculateAmount(order);
}
```

Service Method Invocation (GET)

Before the execution with the REST API, a service method invocation must be allowed in the configuration file. Create the **rest-services.xml file** in the main package of the web module (e.g. com.company.workshop):

```
<?xml version="1.0" encoding="UTF-8"?>
<services xmlns="http://schemas.haulmont.com/cuba/rest-services-v2.xsd">
    <service name="workshop_OrderService">
        <method name="calculateAmount">
            <param name="orderId" type="java.util.UUID"/>
        </method>
    </service>
</services>
```

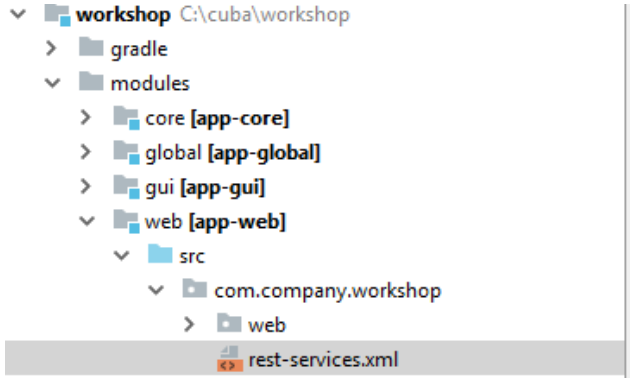




Service Method Invocation (GET)

Then define the rest-services.xml file in the application properties file of the web module (*web-app.properties*):

cuba.rest.servicesConfig = **+com/company/workshop/rest-services.xml**




```
1  ##########################################################################
2  #                              Configuration                             #
3  ##########################################################################
4
5  cuba.rest.servicesConfig = +com/company/workshop/rest-services.xml
6
7  cuba.springContextConfig = +com/company/workshop/web-spring.xml
8
9  cuba.dispatcherSpringContextConfig = +com/company/workshop/web-dispatcher-spring.xml
10
11 cuba.persistenceConfig = +com/company/workshop/persistence.xml
12
13 cuba.metadataConfig = +com/company/workshop/metadata.xml
```

REST API: orders.js

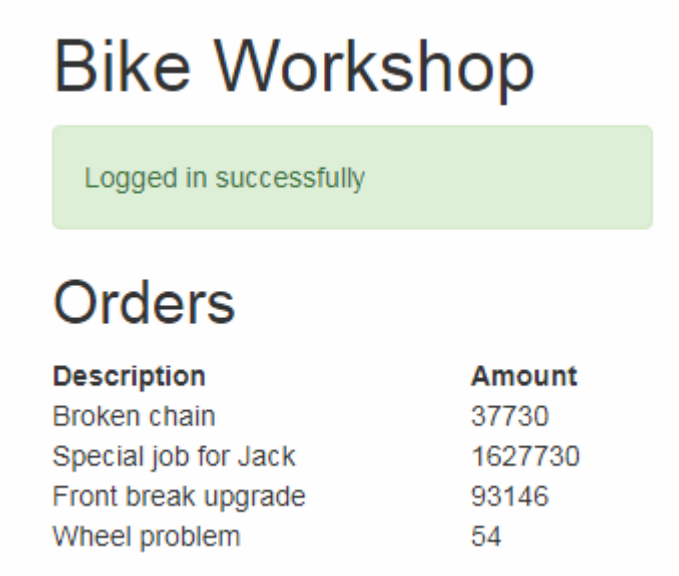
Now we can come back to the **orders.js** file. So, if we are successfully logged in, then we load the list of recent orders.

1. Firstly, we invoke the *calculateAmount()* method of the *OrderService* via the GET request v authentication parameters.
2. In case the data retrieved, we append a new table row to the **ordersList** table: the order descript and the order amount, calculatec the server side.

```
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) {
            $.get({
                url: 'http://localhost:8080/app/rest/v2/services/workshop_OrderService/' +
                       'calculateAmount?orderId='.concat(order.id),
                headers: {
                    'Authorization': 'Bearer ' + oauthToken,
                    'Content-Type': 'application/x-www-form-urlencoded'
                    },
                success: function (data) {
                    $('#ordersList').append("<tr>" + "<td>" + order.description + "</td>" +
                    "<td>" + data + "</td>" + "</tr>");
                    }
                })
            });
        }
    });
}
```

REST API: result

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



Summary

This is a very small application for bicycle workshop management. It is simple but can be applied to a real local workshop.

You can run it in the 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.

We have many more features!

In this session covers just a few features of CUBA, but the platform has many more...

If you want to learn more about additional modules and components just take a look at CUBA documentation:

https://www.cuba-platform.com/manual

Questions?

Visit our forum

https://www.cuba-platform.com/discuss/