



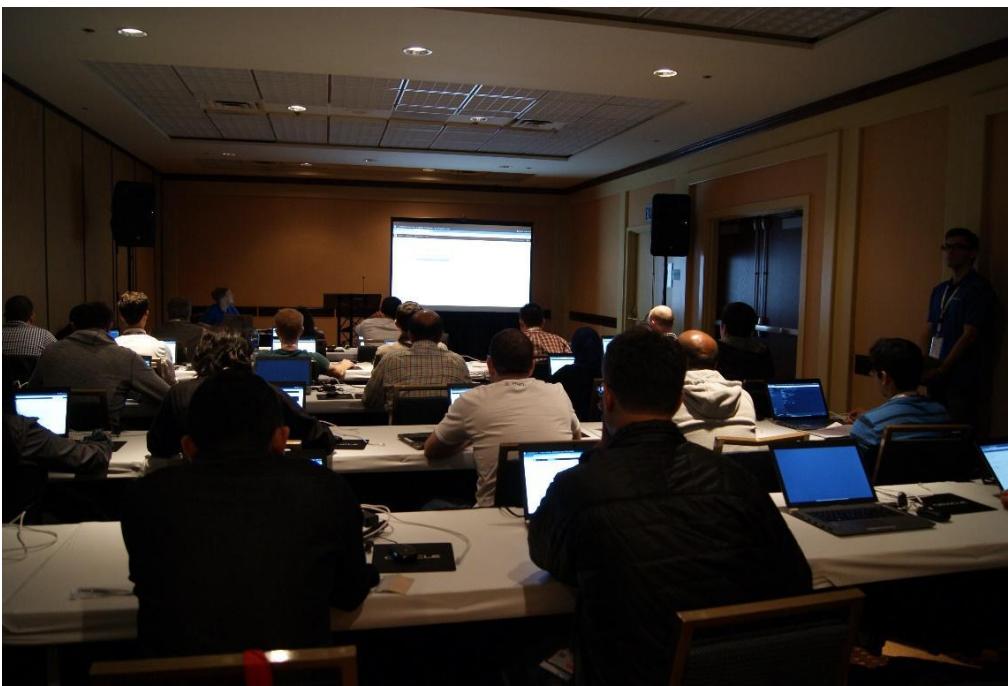
**Develop a Fully Functional
Business Application in Hours with
CUBA Platform**

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.

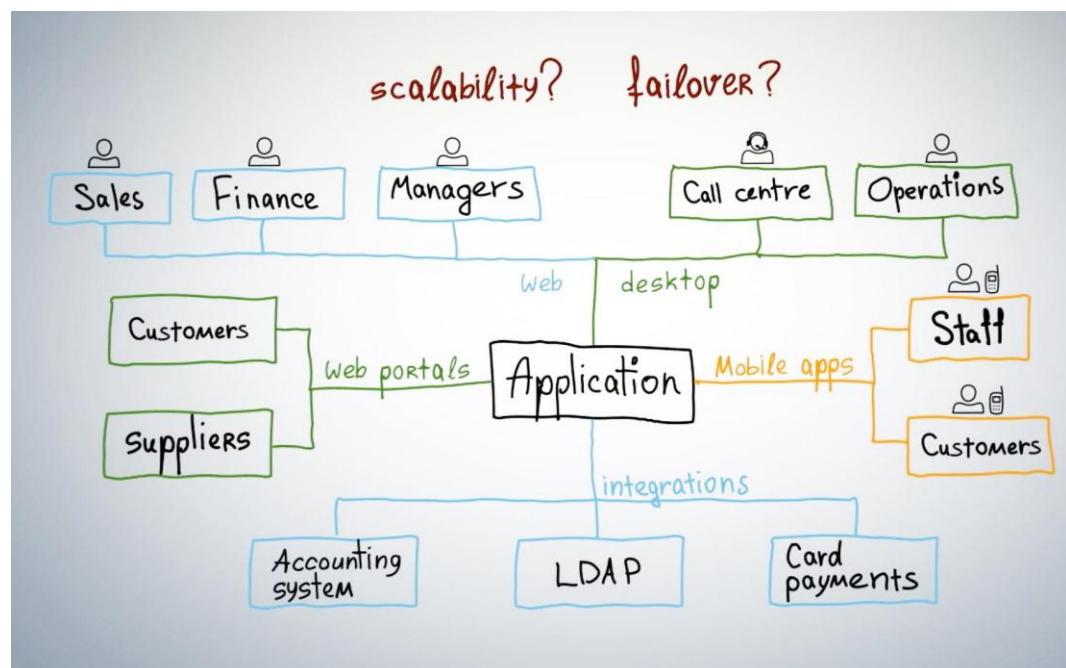


What is CUBA Platform?

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



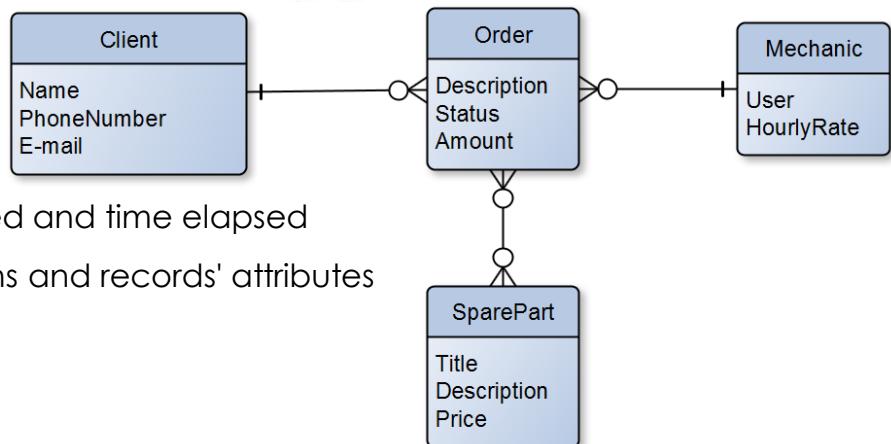
What are we going to automate

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!**

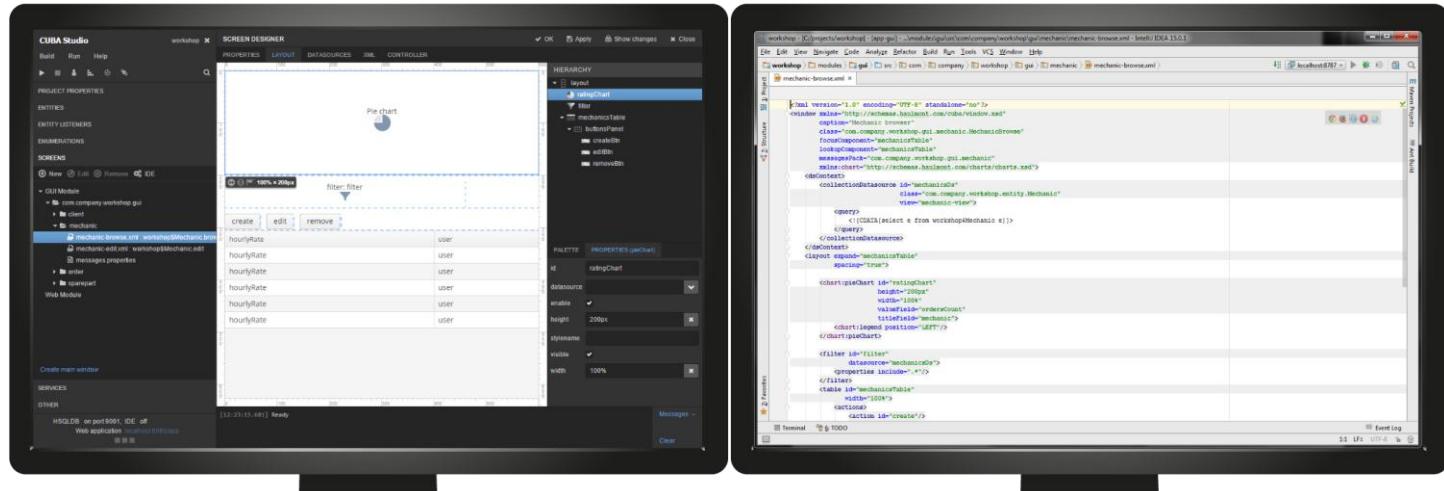


Environment and tools



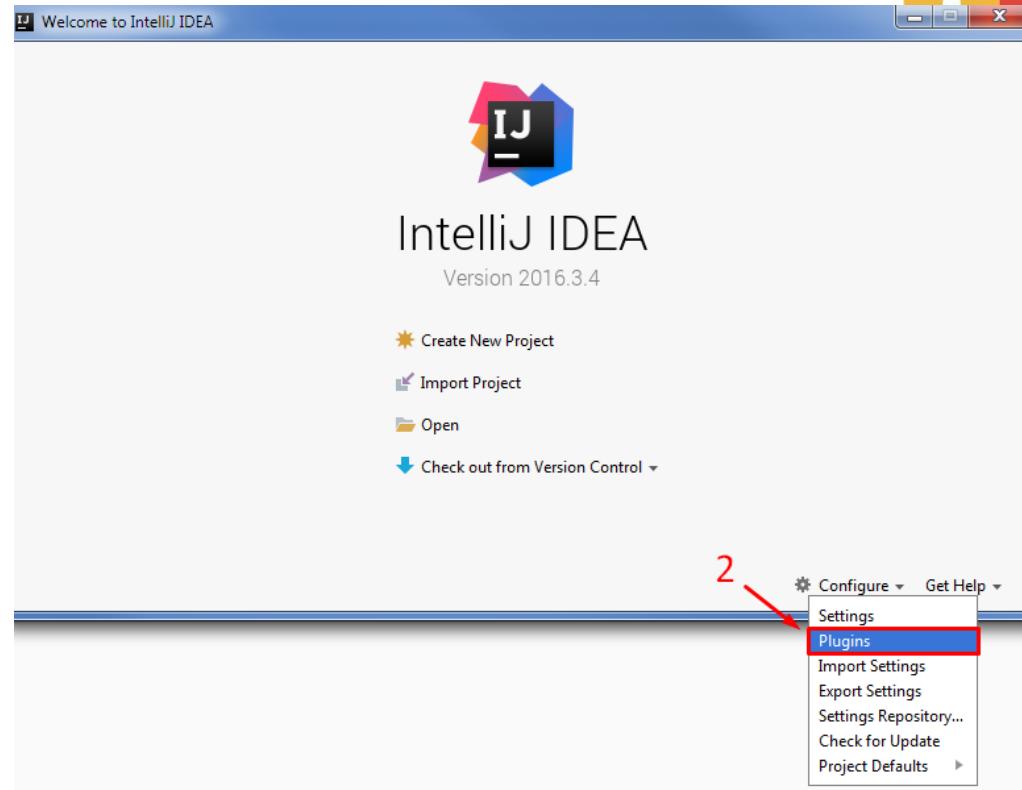
Development environment

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



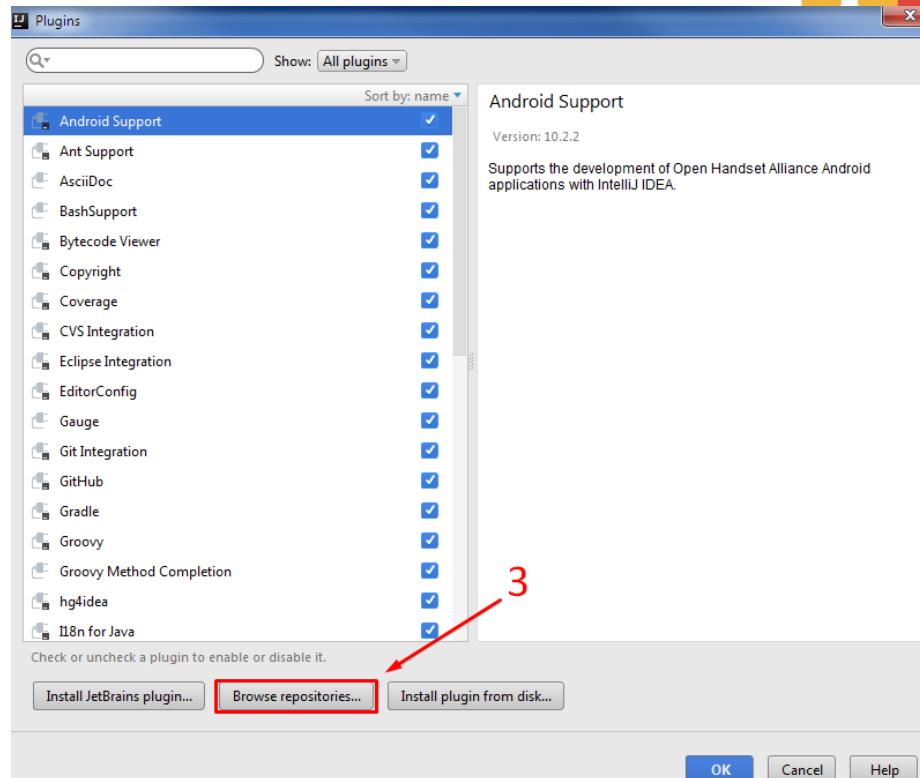
How to install CUBA Plugin for IntelliJ IDEA

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



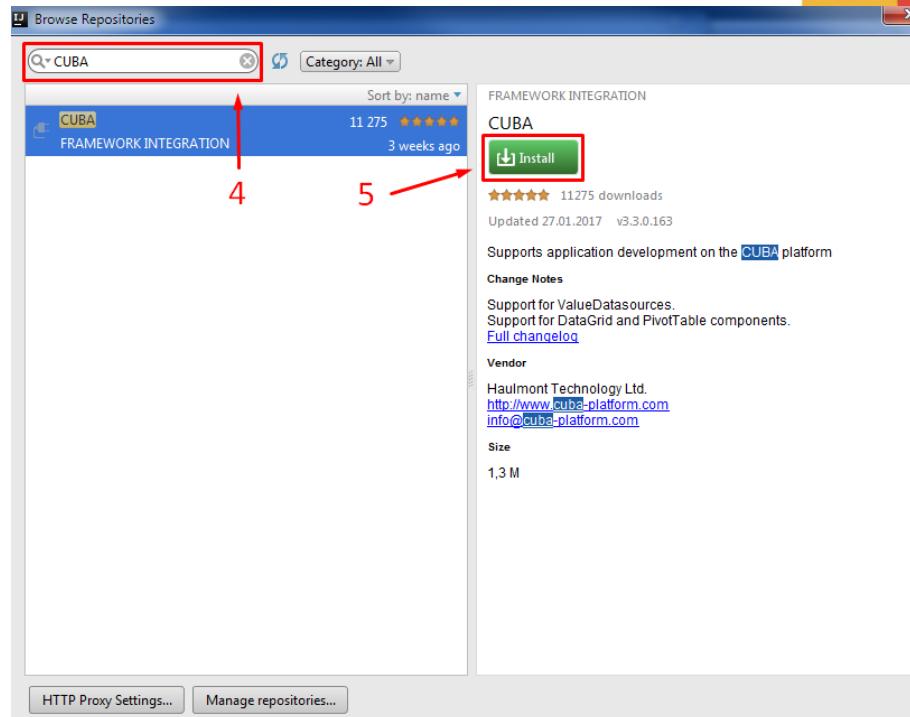
How to install CUBA Plugin for IntelliJ IDEA

3. Click on Browse repositories



How to install CUBA Plugin for IntelliJ IDEA

4. Find CUBA plugin
5. Click Install

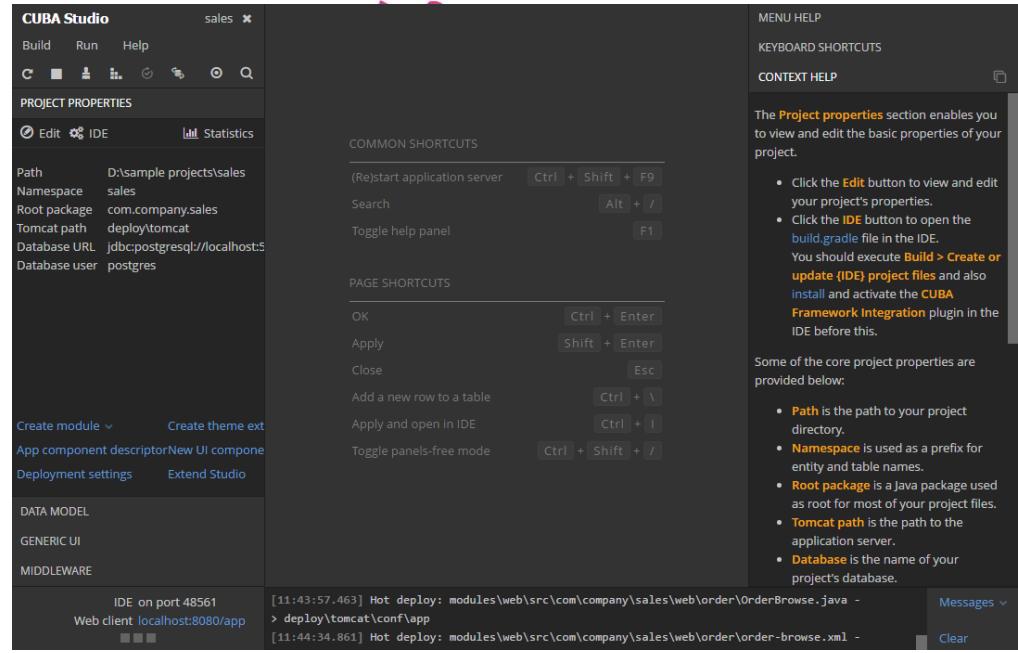


Getting started

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



The screenshot shows the CUBA Studio interface. On the left, the "PROJECT PROPERTIES" panel displays basic project settings:

Path	D:\sample projects\sales
Namespace	sales
Root package	com.company.sales
Tomcat path	deploy\tomcat
Database URL	jdbc:postgresql://localhost:5432/postgres
Database user	postgres

Below this are buttons for "Create module", "Create theme ext", "App component descriptor", "New UI compone", "Deployment settings", and "Extend Studio".

The right side of the interface features a sidebar with help documentation:

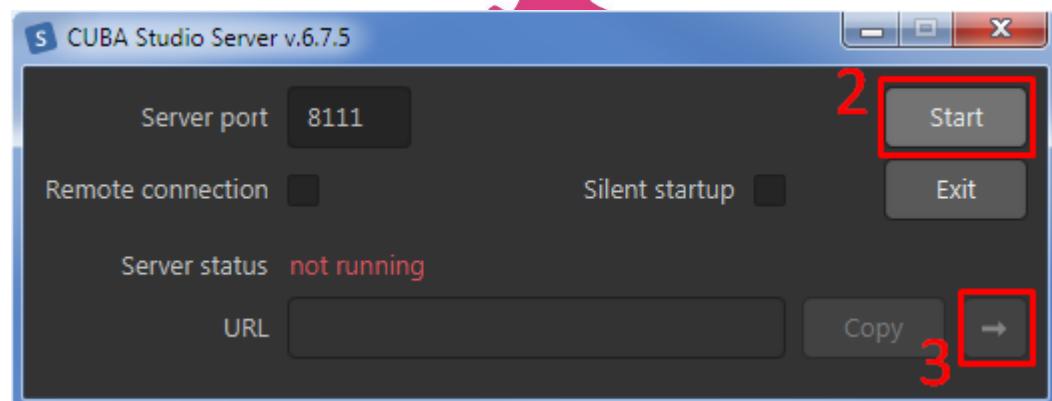
- MENU HELP**: Describes the "Project properties" section.
- KEYBOARD SHORTCUTS**: Lists common keyboard shortcuts like "Restart application server" (Ctrl + Shift + F9) and "Search" (Alt + /).
- CONTEXT HELP**: Provides detailed descriptions for various project properties and IDE integration.

At the bottom, a terminal window shows deployment logs:

```
[11:43:57.463] Hot deploy: modules\web\src\com\company\sales\web\order\OrderBrowse.java -> deploy\tomcat\conf\app  
[11:44:34.861] Hot deploy: modules\web\src\com\company\sales\web\order\order-browse.xml -
```

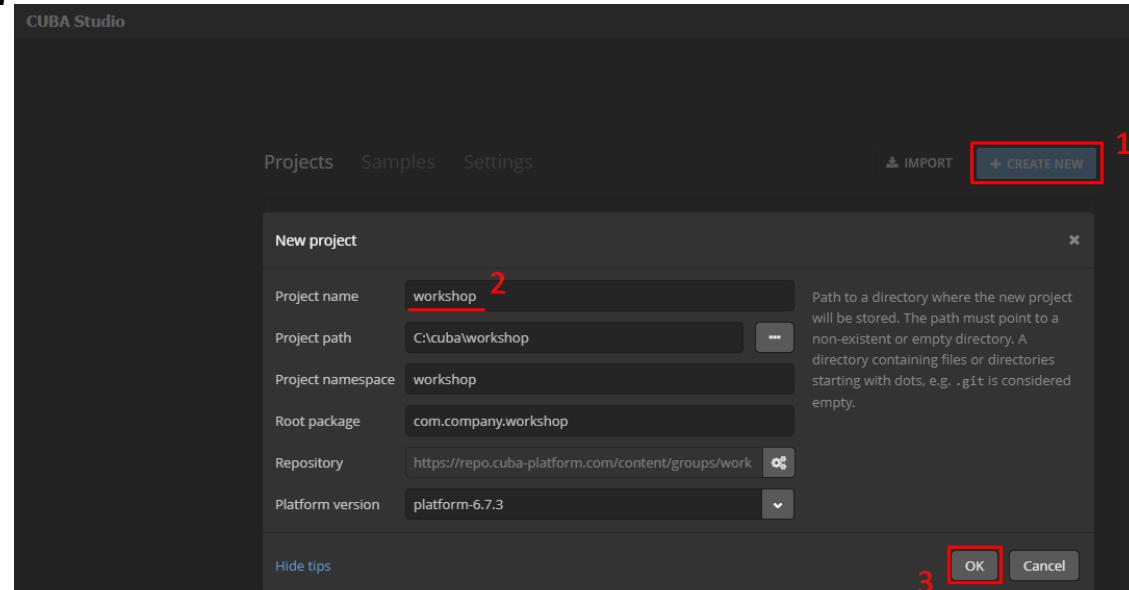
Start CUBA Studio

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

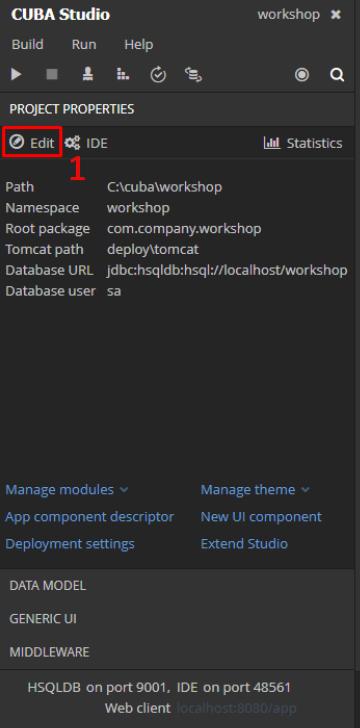


CUBA Studio workspace

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



The screenshot shows the CUBA Studio interface. On the left, the 'PROJECT PROPERTIES' dialog is open, displaying basic project settings like Path, Namespace, Root package, Tomcat path, Database URL, and Database user. The 'Edit' button is highlighted with a red box and a number '1'. On the right, the 'Help' panel is displayed, containing sections for 'COMMON SHORTCUTS' and 'PAGE SHORTCUTS', along with detailed descriptions of project properties and the Manage modules link.

MENU HELP

KEYBOARD SHORTCUTS

CONTEXT HELP

The **Project properties** section enables you to view and edit the basic properties of your project.

- Click the **Edit** button to view and edit your project's properties.
- Click the **IDE** button to open the build.gradle file in the IDE.

You should execute **Build > Create or update (IDE) project files** and also **install** and activate the **CUBA Framework Integration** plugin in the IDE before this.

Some of the core project properties are provided below:

- Path** is the path to your project directory.
- Namespace** is used as a prefix for entity and table names.
- Root package** is a Java package used as root for most of your project files.
- Tomcat path** is the path to the application server.
- Database** is the name of your project's database.

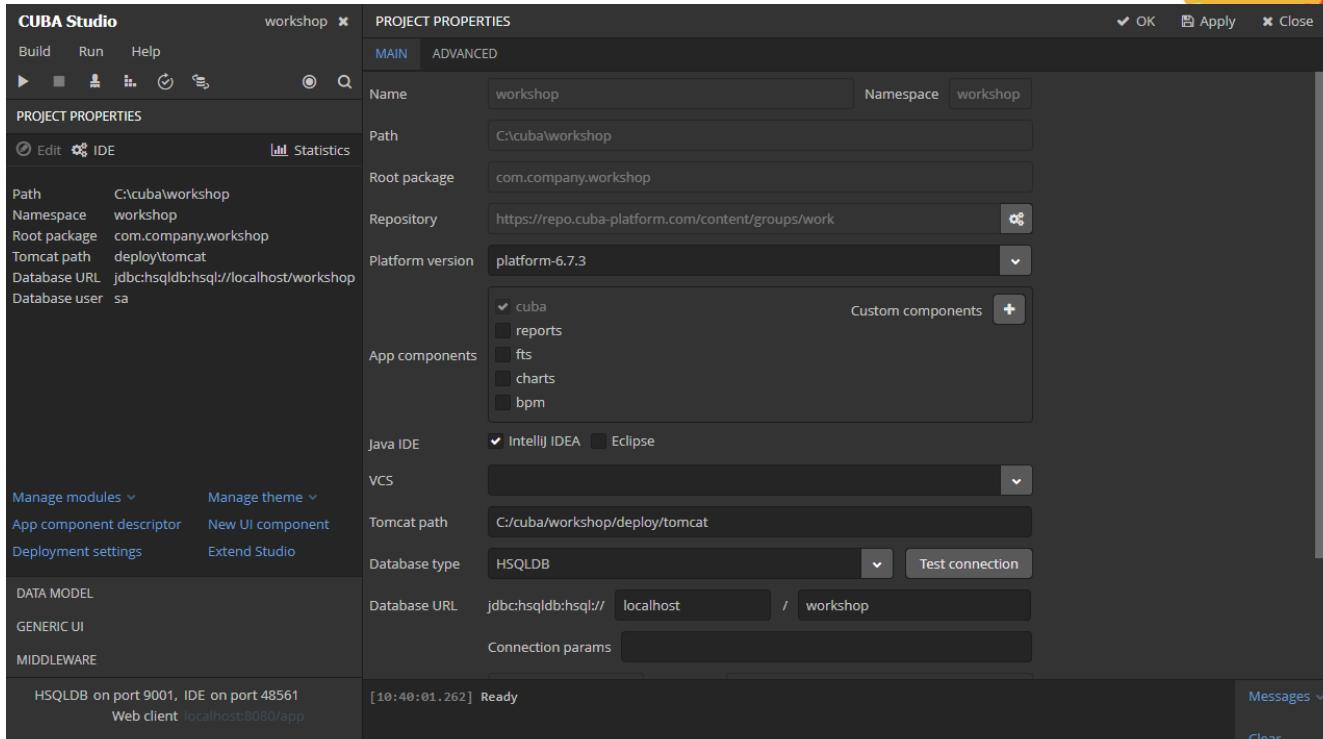
The **Manage modules** link allows you to add and remove optional client modules or the **web-toolkit** module required for **custom visual components**.

Messages 

Project properties screen

This is a page where we configure our project.

The CUBA Platform supports PostgreSQL, MS SQL, Oracle and HSQL databases.



The screenshot shows the CUBA Studio interface with the "workshop" project selected. The main window displays the "PROJECT PROPERTIES" screen, which includes tabs for "MAIN" and "ADVANCED".

MAIN Tab (Visible):

- Name:** workshop
- Namespace:** workshop
- Path:** C:\cuba\workshop
- Root package:** com.company.workshop
- Repository:** https://repo.cuba-platform.com/content/groups/work
- Platform version:** platform-6.7.3
- App components:** cuba, reports, fits, charts, bpm (checkboxes)
- Java IDE:** IntelliJ IDEA (checked), Eclipse
- VCS:** (dropdown menu)
- Tomcat path:** C:/cuba/workshop/deploy/tomcat
- Database type:** HSQLDB
- Database URL:** jdbc:hsqldb:hsqldb://localhost/localhost / workshop
- Connection params:** (text input field)

ADVANCED Tab (Hidden):

- Manage modules:** (button)
- Manage theme:** (button)
- App component descriptor:** New UI component
- Deployment settings:** Extend Studio
- DATA MODEL:** (button)
- GENERIC UI:** (button)
- MIDDLEWARE:** (button)

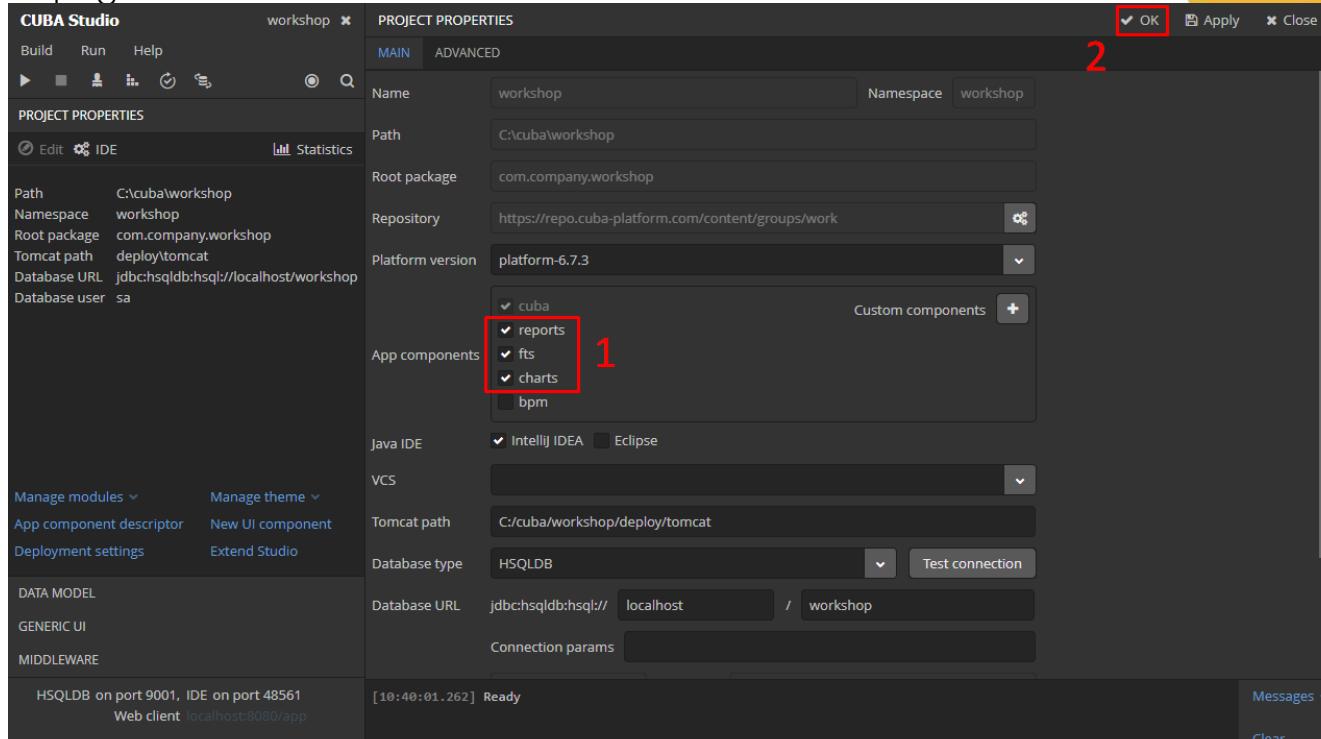
Bottom Status Bar:

- [10:40:01.262] Ready
- Messages (dropdown menu)
- Clear (button)

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.

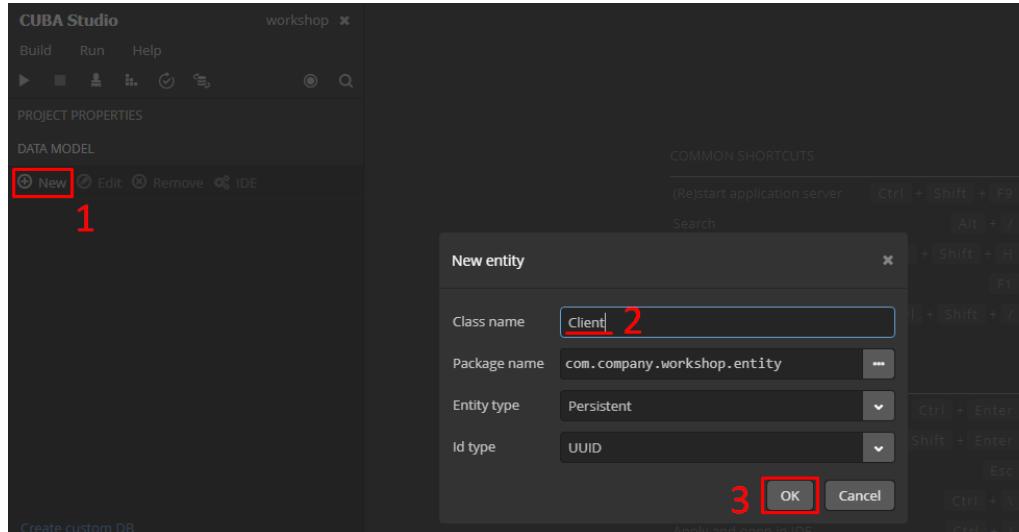


Data model

Create the data model

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

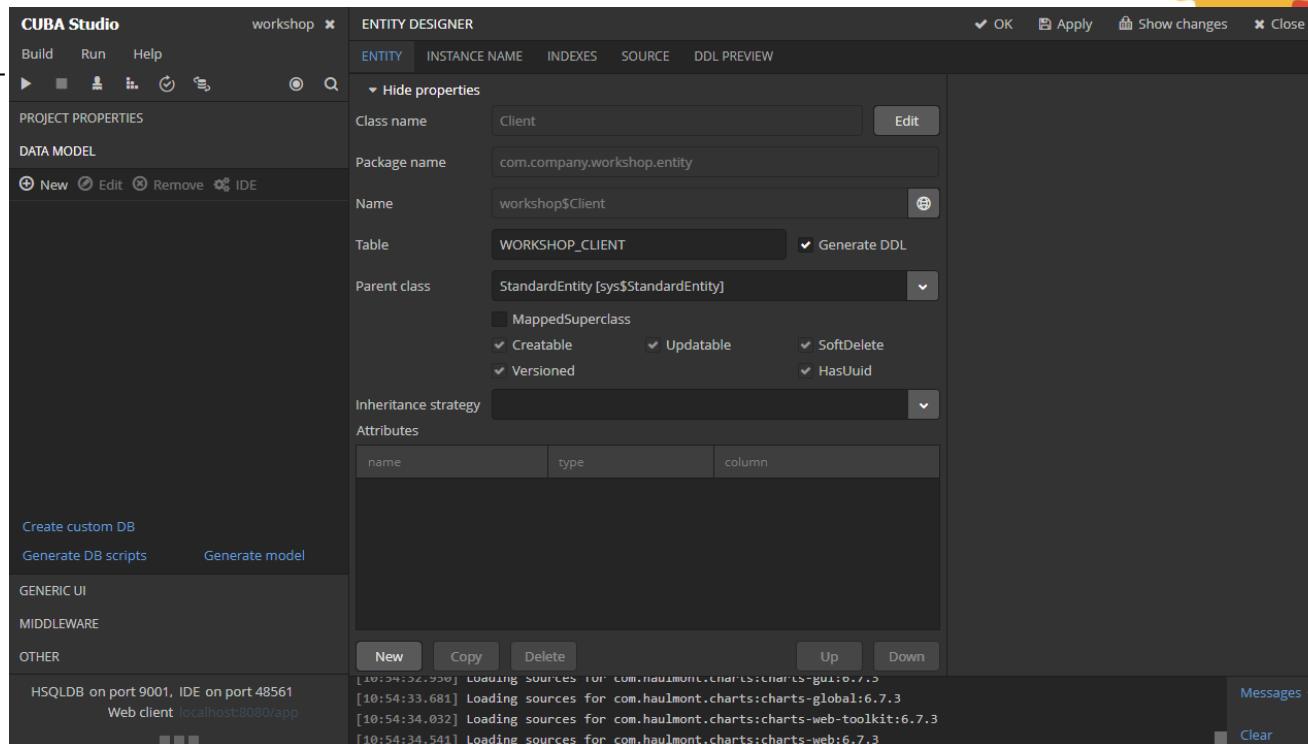
1. Click **New -> Entity**
2. Input **Class name: Client**
3. Click **OK**



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).

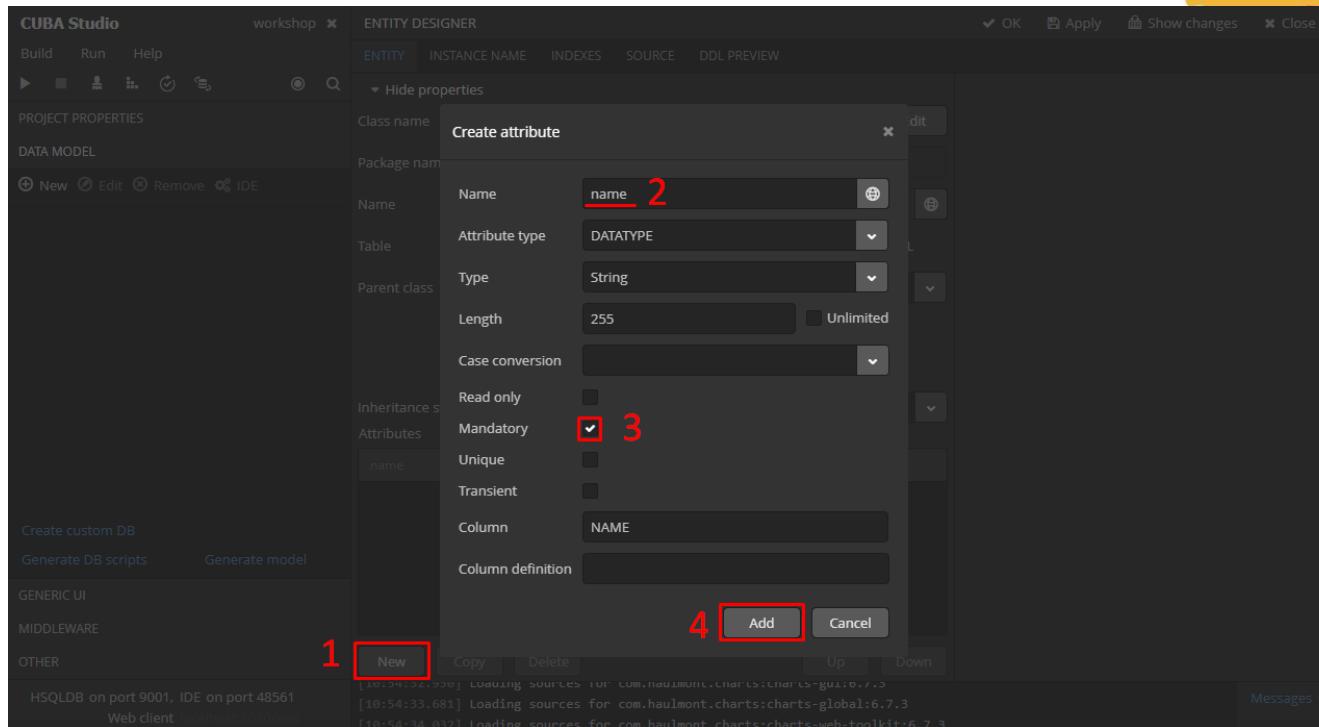


The screenshot shows the CUBA Studio Entity Designer dialog. The left sidebar has tabs for 'PROJECT PROPERTIES' and 'DATA MODEL'. Under 'DATA MODEL', there are buttons for 'New', 'Edit', 'Remove', and 'IDE'. Below these are links for 'Create custom DB', 'Generate DB scripts', and 'Generate model'. The main area is titled 'ENTITY DESIGNER' with tabs for 'ENTITY', 'INSTANCE NAME', 'INDEXES', 'SOURCE', and 'DDL PREVIEW'. The 'ENTITY' tab is selected. It shows fields for 'Class name' (Client), 'Package name' (com.company.workshop.entity), 'Name' (workshop\$Client), 'Table' (WORKSHOP_CLIENT), 'Parent class' (StandardEntity [sys\$StandardEntity]), and checkboxes for 'Generate DDL', 'MappedSuperclass', 'Creatable', 'Updatable', 'SoftDelete', 'Versioned', and 'HasUuid'. Below this is an 'Inheritance strategy' dropdown and an 'Attributes' section with a table. The table has columns for 'name', 'type', and 'column'. At the bottom are buttons for 'New', 'Copy', 'Delete', 'Up', 'Down', and 'OK', 'Apply', 'Show changes', and 'Close'.

Attribute editor

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

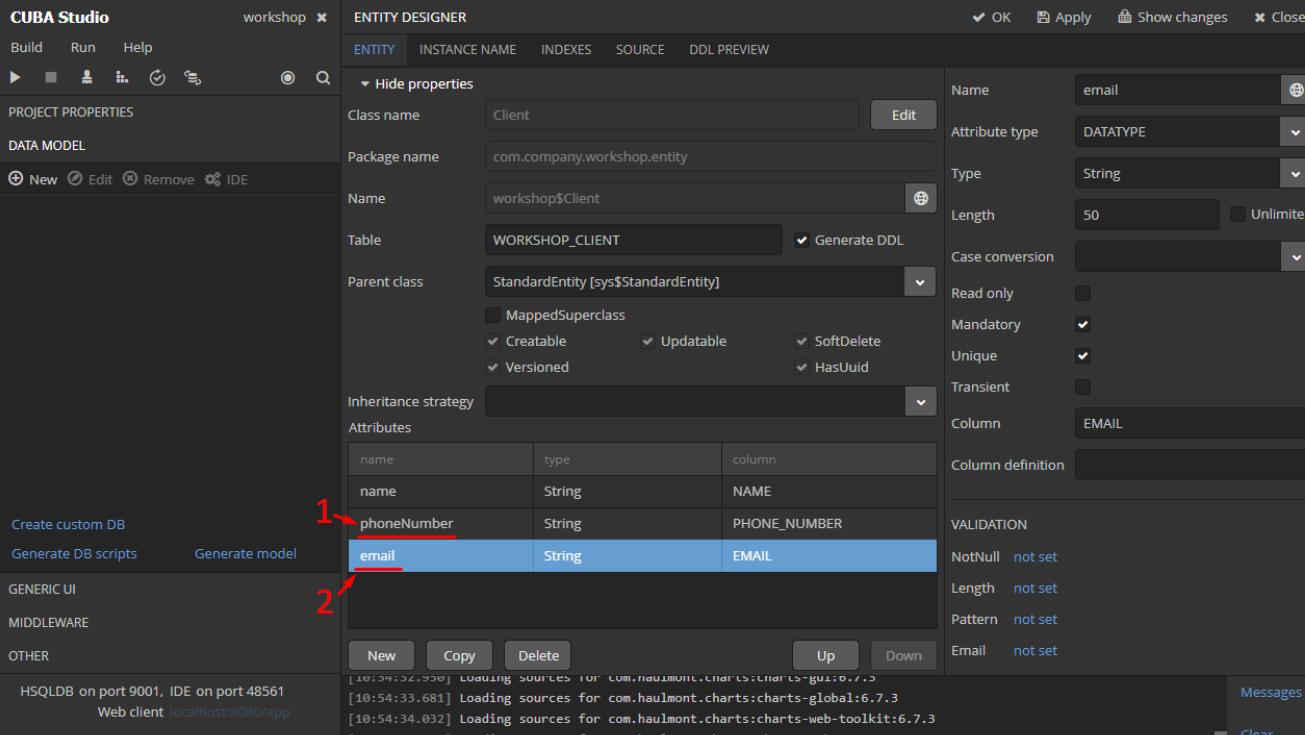
Attribute editor enables us to create or edit attribute and its parameters, such as Attribute type, Java Type, Read only, Mandatory, Unique, etc.



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



The screenshot shows the CUBA Studio Entity Designer interface. The Entity tab is selected, displaying the following configuration for the 'Client' entity:

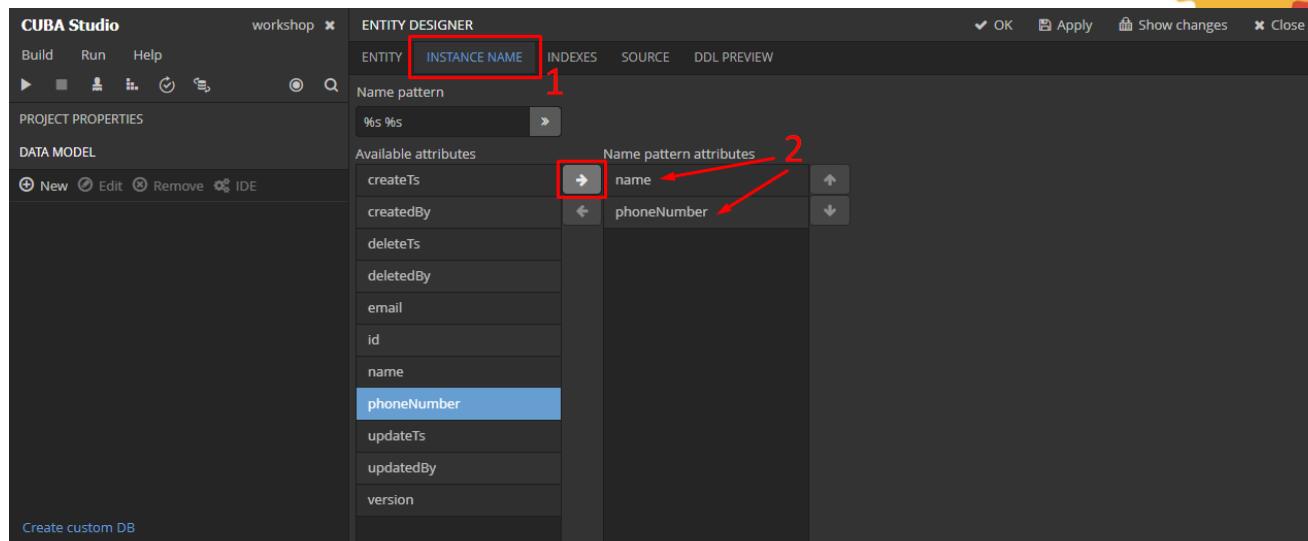
- Class name:** Client
- Package name:** com.company.workshop.entity
- Name:** workshop\$Client
- Table:** WORKSHOP_CLIENT
- Parent class:** StandardEntity [sys\$StandardEntity]
- Inheritance strategy:** (dropdown menu)
- Attributes:** A table showing two attributes:
 - 1. **phoneNumber**: String type, column NAME, length 20, Unique checked.
 - 2. **email**: String type, column EMAIL, length 50, Unique checked.

Red numbers 1 and 2 are overlaid on the 'phoneNumber' and 'email' rows respectively, pointing to the attribute names.

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**

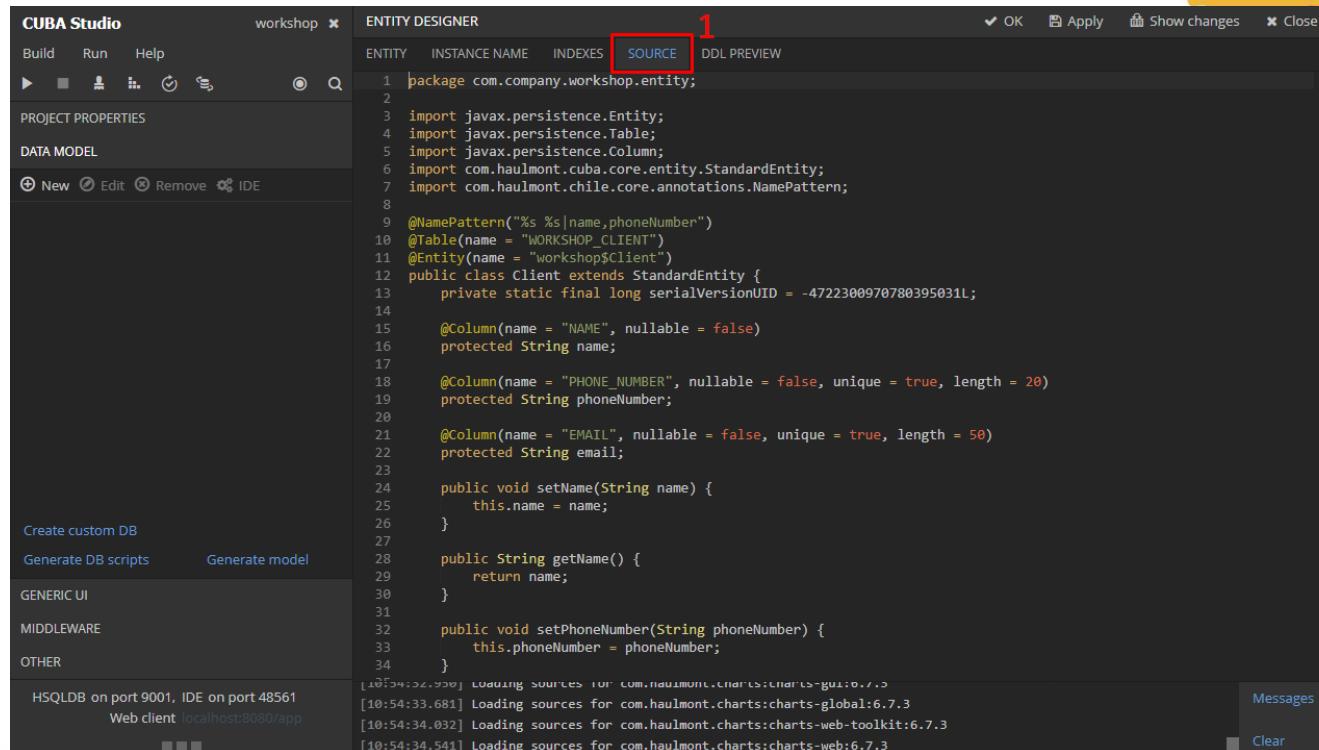


Generated source code for the Client entity

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.



The screenshot shows the CUBA Studio interface with the 'workshop' project selected. The 'PROJECT PROPERTIES' and 'DATA MODEL' tabs are visible. In the 'DATA MODEL' section, there are buttons for 'New', 'Edit', 'Remove', and 'IDE'. The 'SOURCE' tab of the 'ENTITY DESIGNER' panel is selected, indicated by a red box and a red number '1'. The code editor displays the following Java code:

```
1 package com.company.workshop.entity;
2
3 import javax.persistence.Entity;
4 import javax.persistence.Table;
5 import javax.persistence.Column;
6 import com.haulmont.cuba.core.entity.StandardEntity;
7 import com.haulmont.chile.core.annotations.NamePattern;
8
9 @NamePattern("%s %s|name,phoneNumber")
10 @Table(name = "WORKSHOP_CLIENT")
11 @Entity(name = "workshop$Client")
12 public class Client extends StandardEntity {
13     private static final long serialVersionUID = -4722300970780395031L;
14
15     @Column(name = "NAME", nullable = false)
16     protected String name;
17
18     @Column(name = "PHONE_NUMBER", nullable = false, unique = true, length = 20)
19     protected String phoneNumber;
20
21     @Column(name = "EMAIL", nullable = false, unique = true, length = 50)
22     protected String email;
23
24     public void setName(String name) {
25         this.name = name;
26     }
27
28     public String getName() {
29         return name;
30     }
31
32     public void setPhoneNumber(String phoneNumber) {
33         this.phoneNumber = phoneNumber;
34     }
35 }
```

At the bottom of the interface, there are several status messages indicating the loading of sources for various modules:

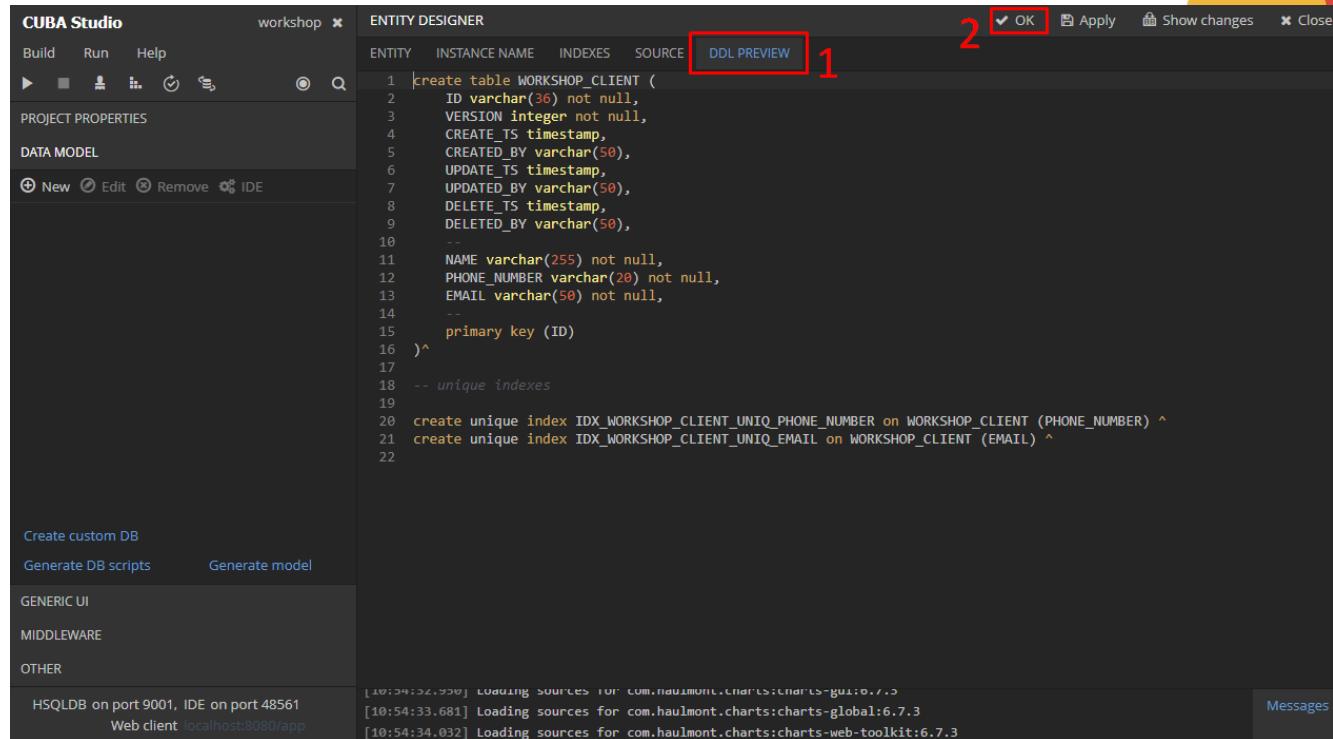
- [10:54:33.681] Loading sources for com.haulmont.charts:charts-gui:6.7.3
- [10:54:34.032] Loading sources for com.haulmont.charts:charts-global:6.7.3
- [10:54:34.732] Loading sources for com.haulmont.charts:charts-web-toolkit:6.7.3
- [10:54:34.541] Loading sources for com.haulmont.charts:charts-web:6.7.3

On the right side, there are 'Messages' and 'Clear' buttons.

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.



The screenshot shows the CUBA Studio interface with the Entity Designer open. The DDL Preview tab is highlighted with a red box and labeled '1'. The OK button is also highlighted with a red box and labeled '2'. The preview window displays the generated SQL script for creating the WORKSHOP_CLIENT table:

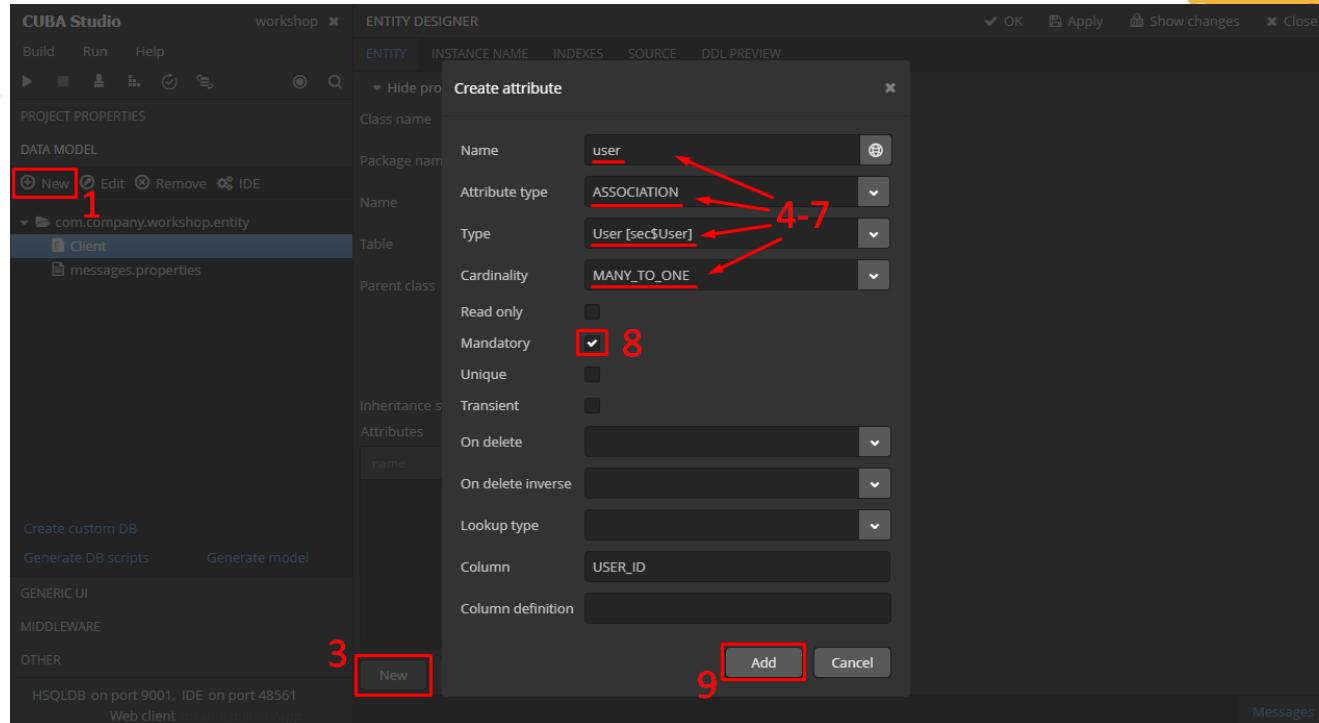
```
1 create table WORKSHOP_CLIENT (
2     ID varchar(36) not null,
3     VERSION integer not null,
4     CREATE_TS timestamp,
5     CREATED_BY varchar(50),
6     UPDATE_TS timestamp,
7     UPDATED_BY varchar(50),
8     DELETE_TS timestamp,
9     DELETED_BY varchar(50),
10    --
11    NAME varchar(255) not null,
12    PHONE_NUMBER varchar(20) not null,
13    EMAIL varchar(50) not null,
14    --
15    primary key (ID)
16 )^
17    -- unique indexes
18
19    create unique index IDX_WORKSHOP_CLIENT_UNIQ_PHONE_NUMBER on WORKSHOP_CLIENT (PHONE_NUMBER) ^
20    create unique index IDX_WORKSHOP_CLIENT_UNIQ_EMAIL on WORKSHOP_CLIENT (EMAIL) ^
21
22
```

The bottom right corner of the preview window shows a 'Messages' section with several log entries:

- [10:54:32.959] Loading sources for com.haulmont.charts:charts-gui:6.7.3
- [10:54:33.681] Loading sources for com.haulmont.charts:charts-global:6.7.3
- [10:54:34.032] Loading sources for com.haulmont.charts:charts-web-toolkit:6.7.3
- [10:54:34.541] Loading sources for com.haulmont.charts:charts-web:6.7.3

Mechanic entity

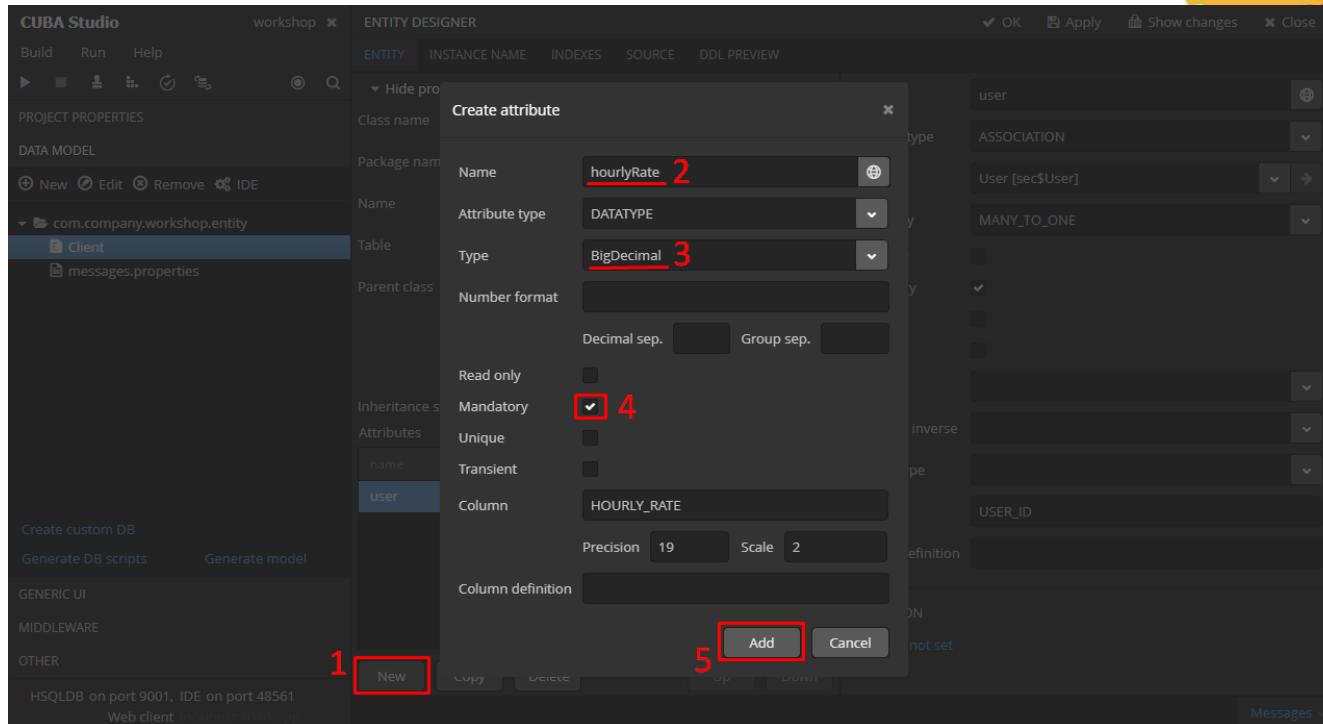
1. Click **New -> Entity**
2. Input **Mechanic** as entity name and click **OK**
3. Create new attribute
4. Set attribute name to **user**
5. Set Attribute type: **ASSOCIATION**
6. Set Type: **User [sec\$User]**
7. Set Cardinality: **MANY_TO_ONE**
8. Select Mandatory checkbox
9. Click Add



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

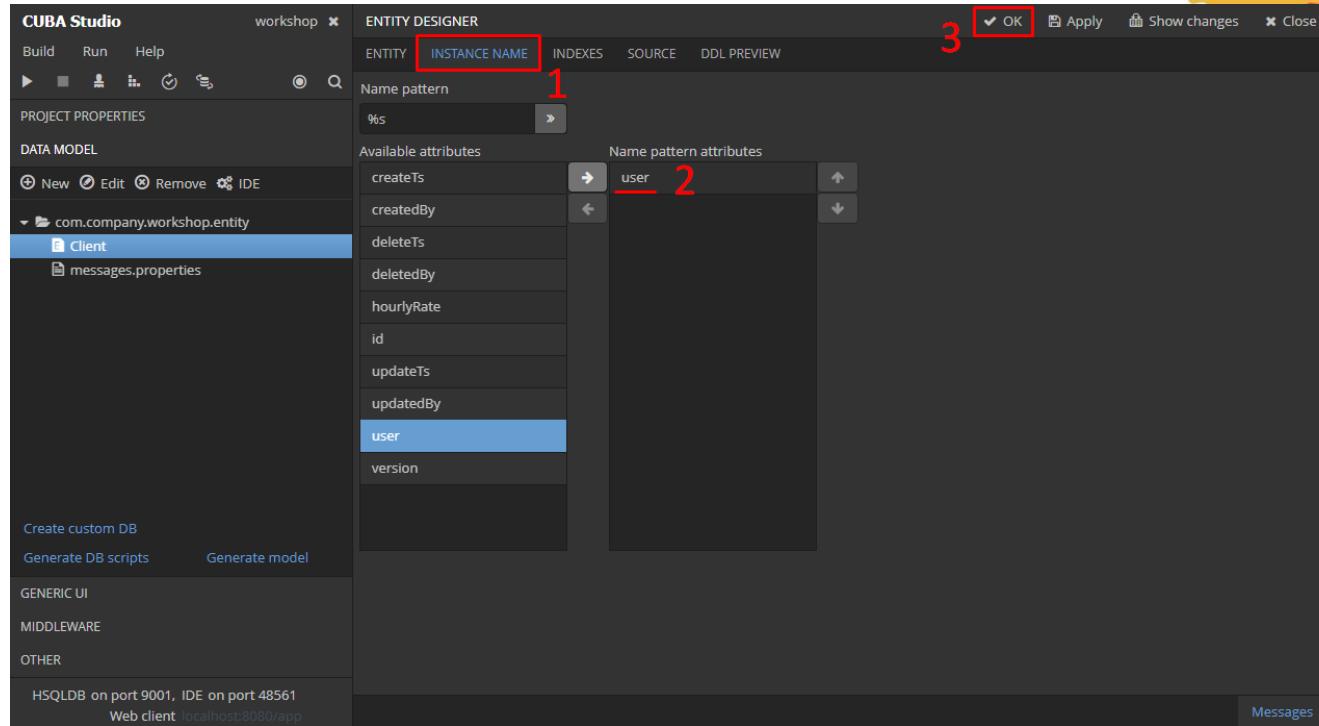
Mechanic entity — hourlyRate attribute

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**



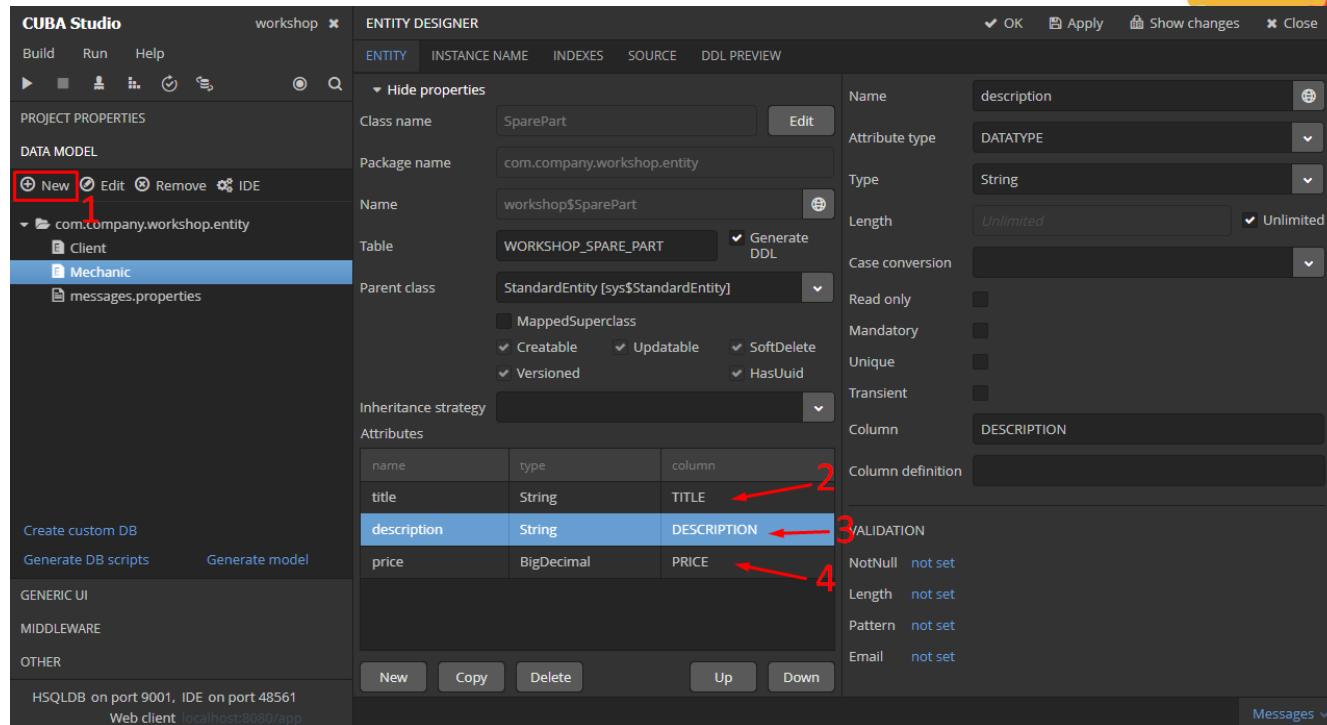
SparePart entity

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



The screenshot shows the CUBA Studio Entity Designer interface. In the left sidebar, under the 'DATA MODEL' section, there is a tree view with a 'New' button highlighted by a red box and the number '1'. The 'Mechanic' node is selected and highlighted in blue. The main panel displays the 'ENTITY DESIGNER' tab with the following configuration:

Name	Attribute type	Type	Length	Case conversion
description	DATATYPE	String	Unlimited	Unlimited

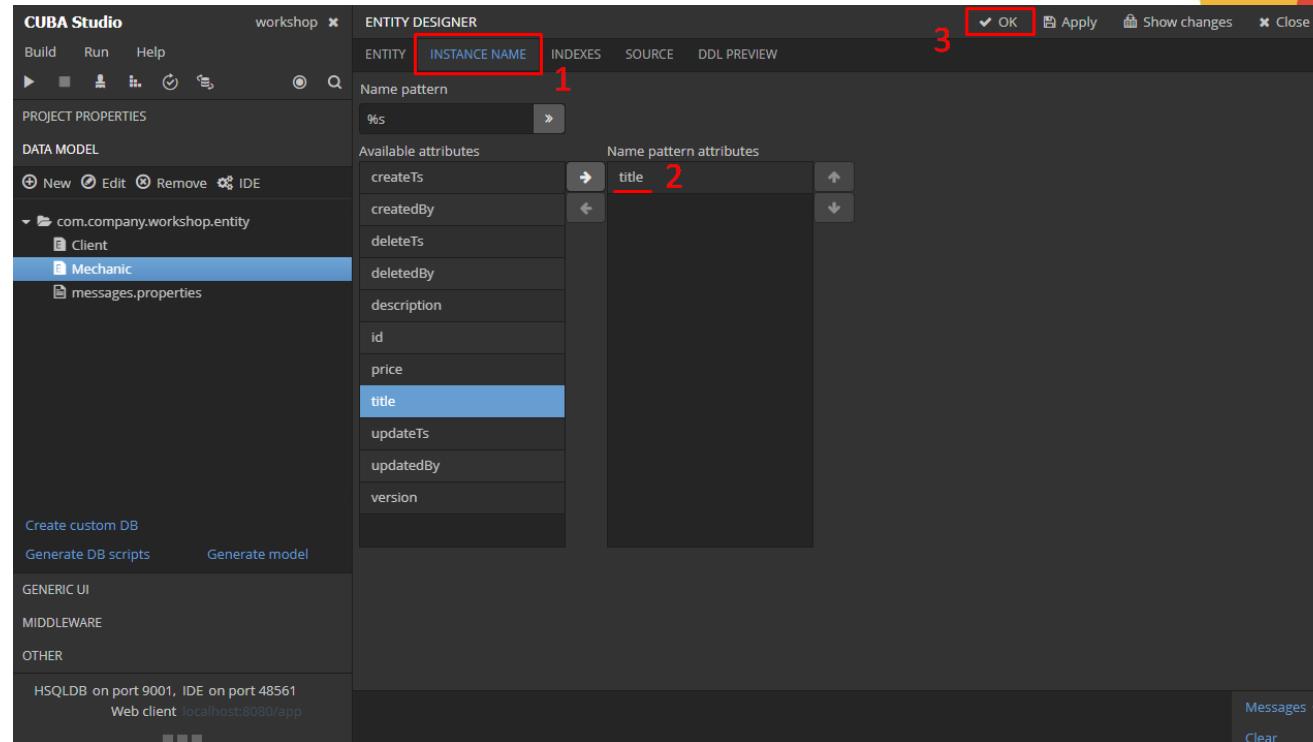
Under the 'Attributes' section, three attributes are defined:

name	type	column
title	String	TITLE
description	String	DESCRIPTION
price	BigDecimal	PRICE

Red arrows numbered 2, 3, and 4 point to the 'TITLE', 'DESCRIPTION', and 'PRICE' columns respectively, indicating the steps for configuration.

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

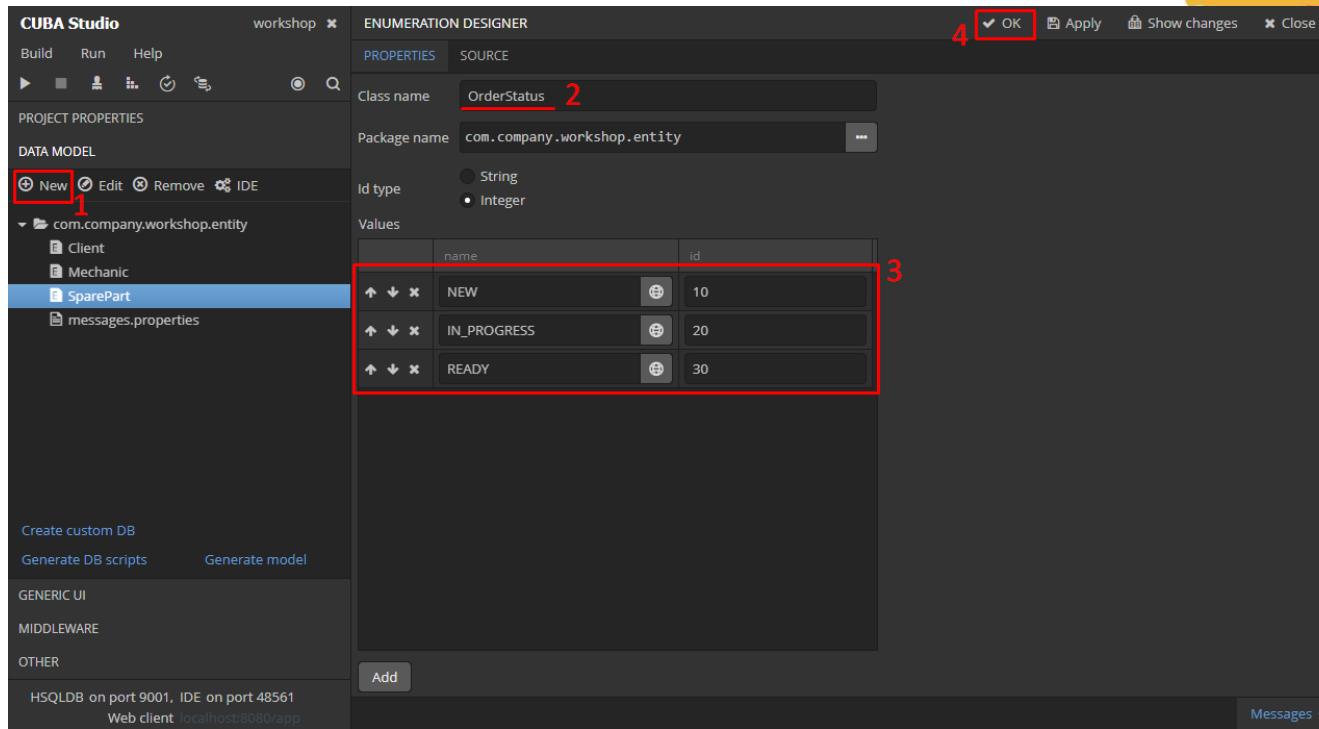
To create the **Order** entity we'll need to create the **OrderStatus** enum first.

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

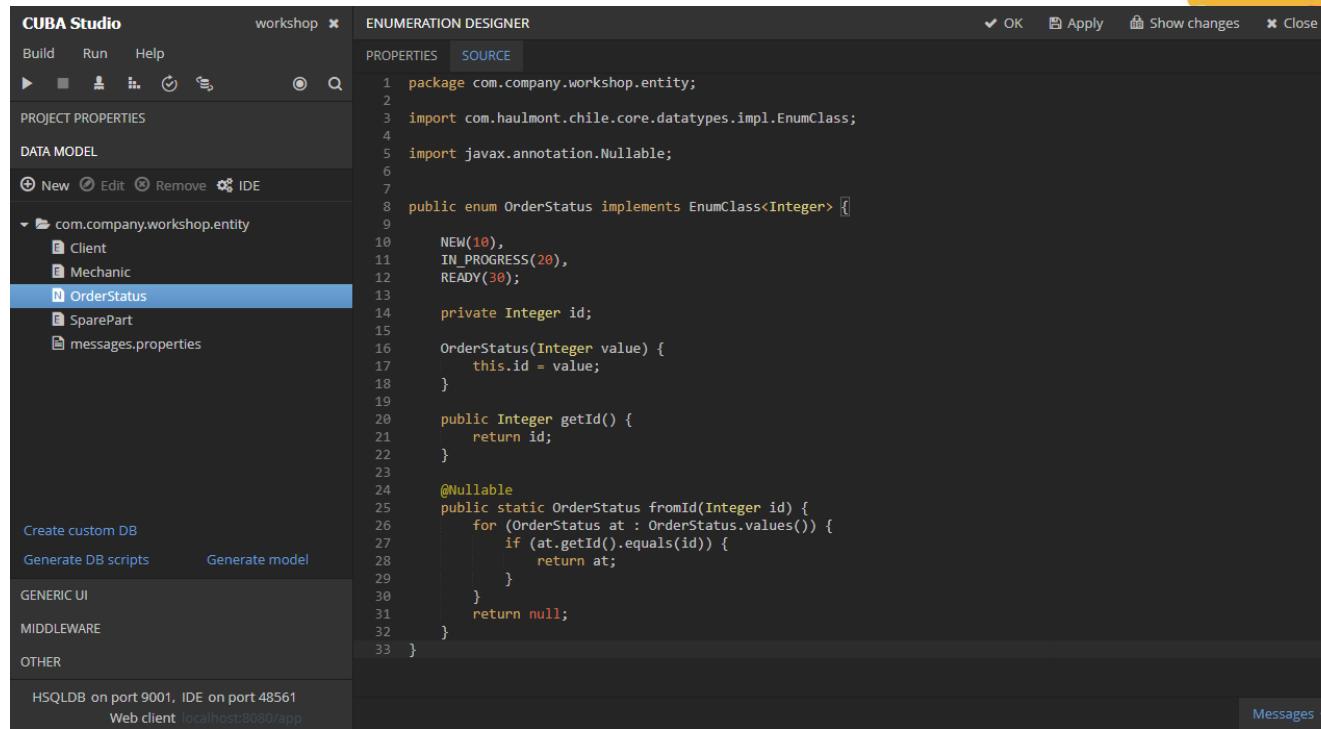
4. Click **OK**



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.



The screenshot shows the CUBA Studio interface with the 'workshop' project selected. In the left sidebar, under 'PROJECT PROPERTIES' and 'DATA MODEL', there is a tree view. The 'OrderStatus' enum is selected, highlighted with a blue background. The main workspace is titled 'ENUMERATION DESIGNER' and contains tabs for 'PROPERTIES' and 'SOURCE'. The 'SOURCE' tab is active, displaying the generated Java code for the OrderStatus enum:

```
1 package com.company.workshop.entity;
2
3 import com.haulmont.chile.core.datatypes.impl.EnumClass;
4
5 import javax.annotation.Nullable;
6
7
8 public enum OrderStatus implements EnumClass<Integer> {
9     NEW(10),
10    IN_PROGRESS(20),
11    READY(30);
12
13    private Integer id;
14
15    OrderStatus(Integer value) {
16        this.id = value;
17    }
18
19    public Integer getId() {
20        return id;
21    }
22
23    @Nullable
24    public static OrderStatus fromId(Integer id) {
25        for (OrderStatus at : OrderStatus.values()) {
26            if (at.getId().equals(id)) {
27                return at;
28            }
29        }
30        return null;
31    }
32}
33}
```

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**

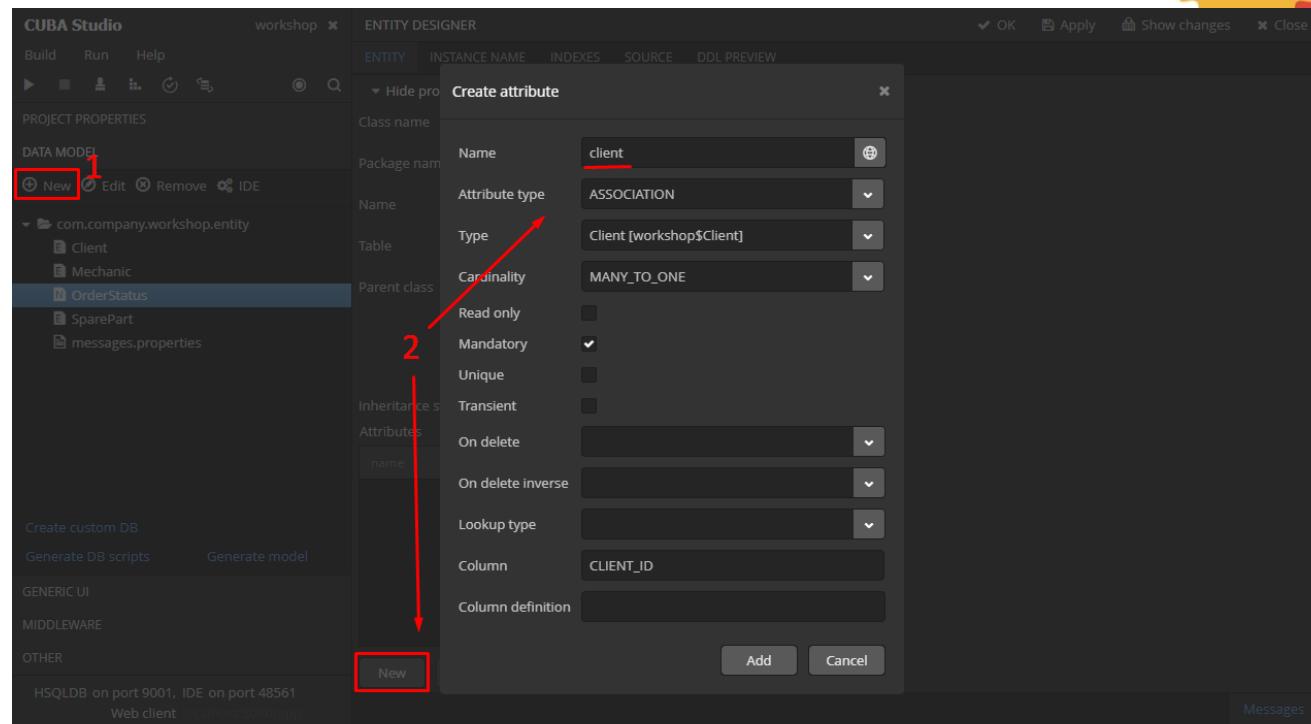
Attribute type: **ASSOCIATION**

Type: **Client**

Cardinality: **MANY_TO_ONE**

Mandatory: **true**

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



Order entity — description, hoursSpent, amount

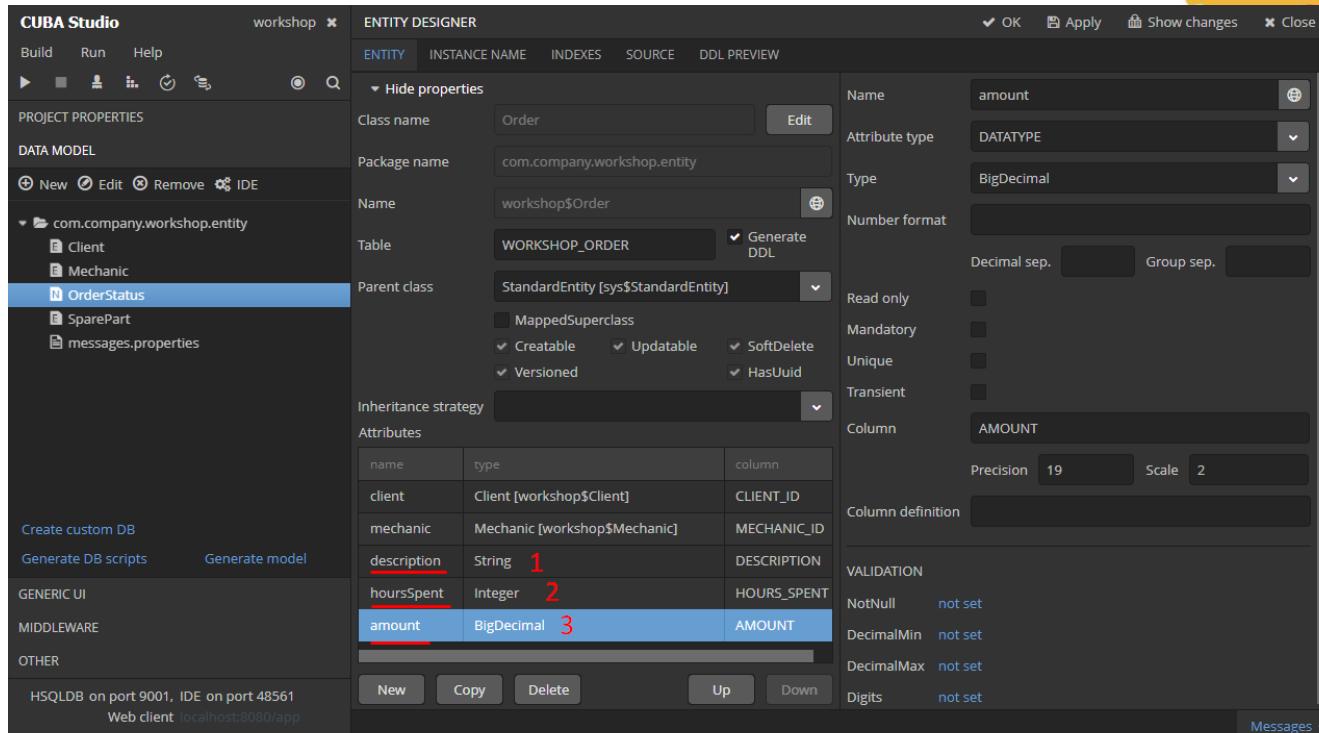
1. Add **description** attribute: **String**, check the **Unlimited** length checkbox, so **description** will have unlimited length

2. Add **hoursSpent** attribute:

Integer

3. Add **amount** attribute:

BigDecimal



The screenshot shows the CUBA Studio Entity Designer interface for the 'Order' entity. The 'DATA MODEL' tab is selected. In the 'Attributes' section, three attributes are defined:

Name	Type	Column
client	Client [workshop\$Client]	CLIENT_ID
mechanic	Mechanic [workshop\$Mechanic]	MECHANIC_ID
<u>description</u>	String 1	DESCRIPTION
hoursSpent	Integer 2	HOURS_SPENT
<u>amount</u>	BigDecimal 3	AMOUNT

Below the attributes, validation rules are listed:

- NotNull not set
- DecimalMin not set
- DecimalMax not set
- Digits not set

Order entity — parts attribute

1. Create a **New** attribute: **parts**

Attribute type: **ASSOCIATION**

Type: **SparePart**

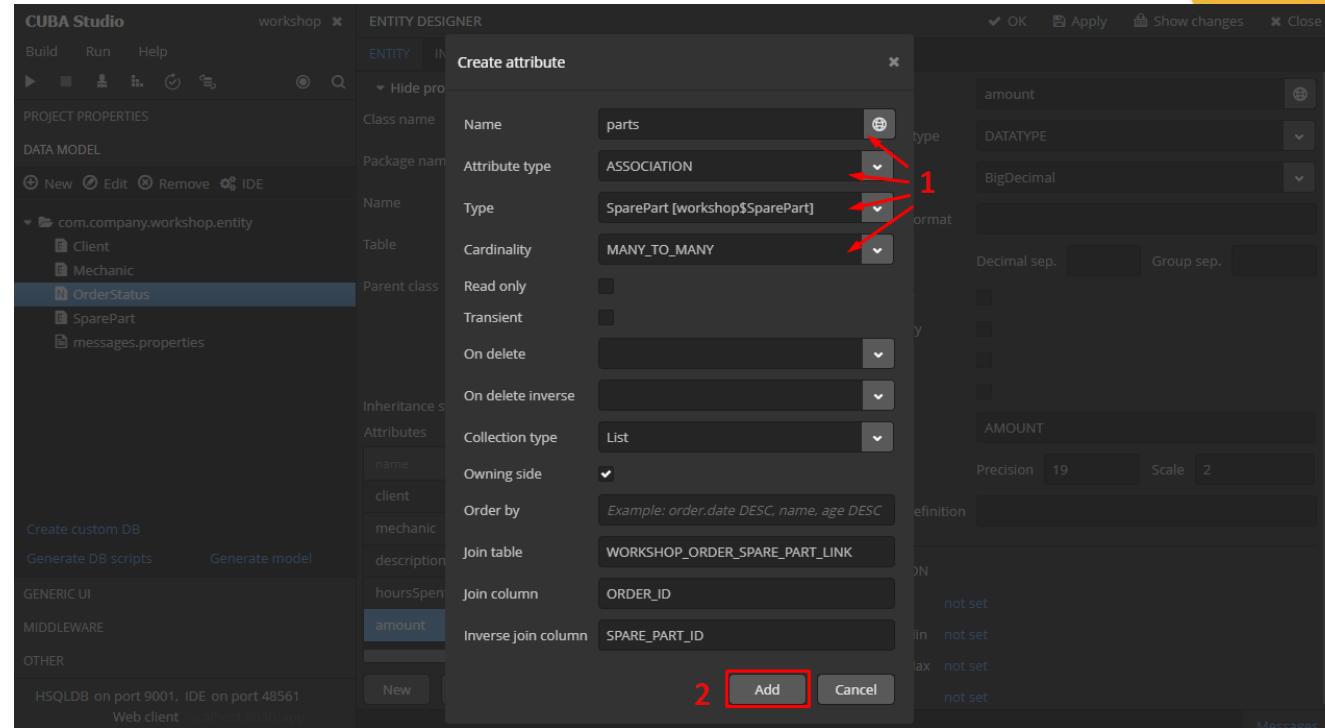
Cardinality: **MANY_TO_MANY**

2. Click on the **Add** button

3. Studio will offer to create a reverse attribute from the

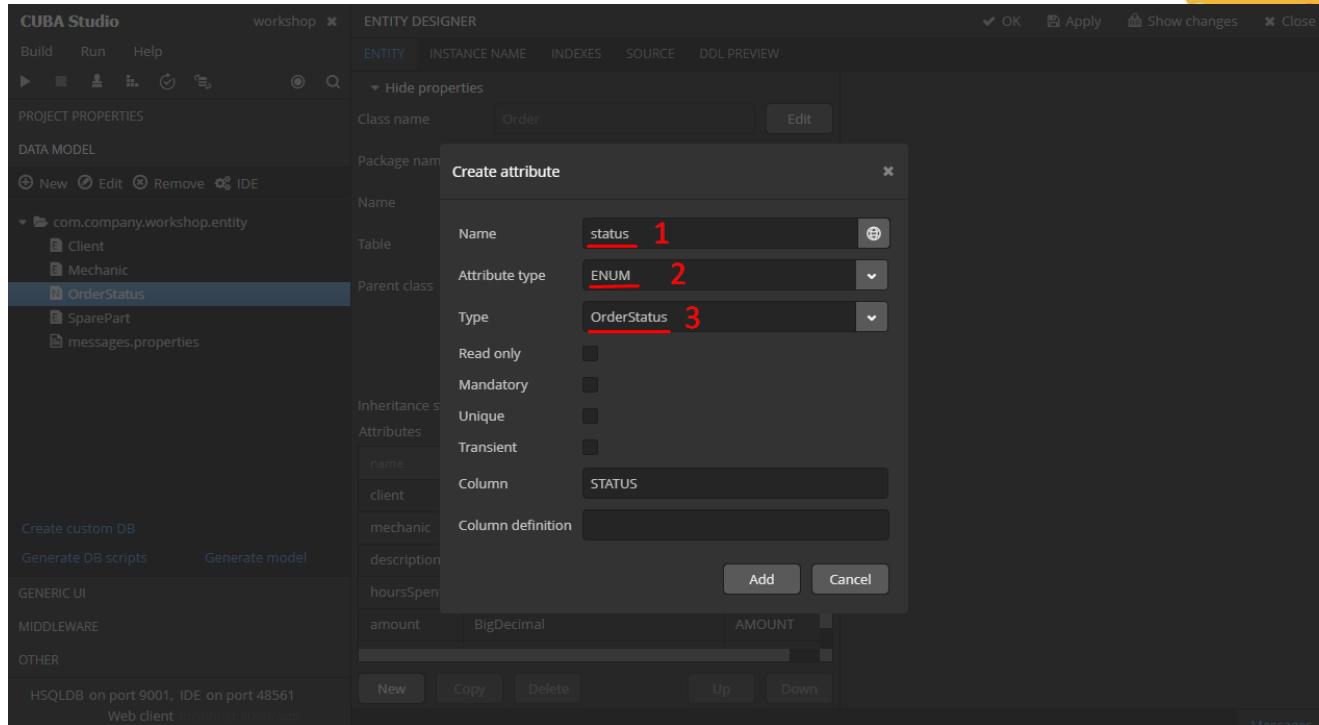
SparePart entity to link it

to **Order**, just click **No**



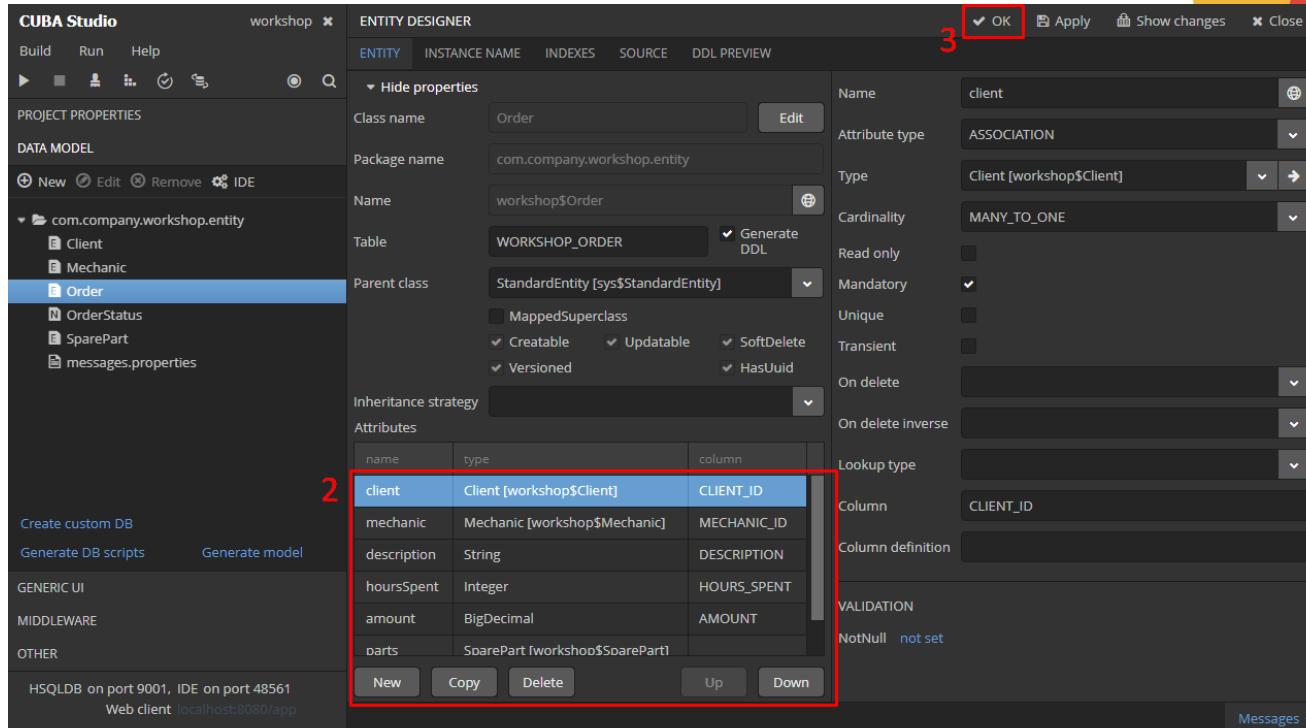
Order entity — status attribute

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



The screenshot shows the CUBA Studio Entity Designer interface for the 'workshop' project. The 'Order' entity is selected in the tree view on the left. The main panel displays the entity's properties and attributes.

Entity Designer Properties:

- Class name: Order
- Package name: com.company.workshop.entity
- Name: workshop\$Order
- Table: WORKSHOP_ORDER
- Parent class: StandardEntity [sys\$StandardEntity]
- Inheritance strategy: (dropdown menu)
- Attributes (table):

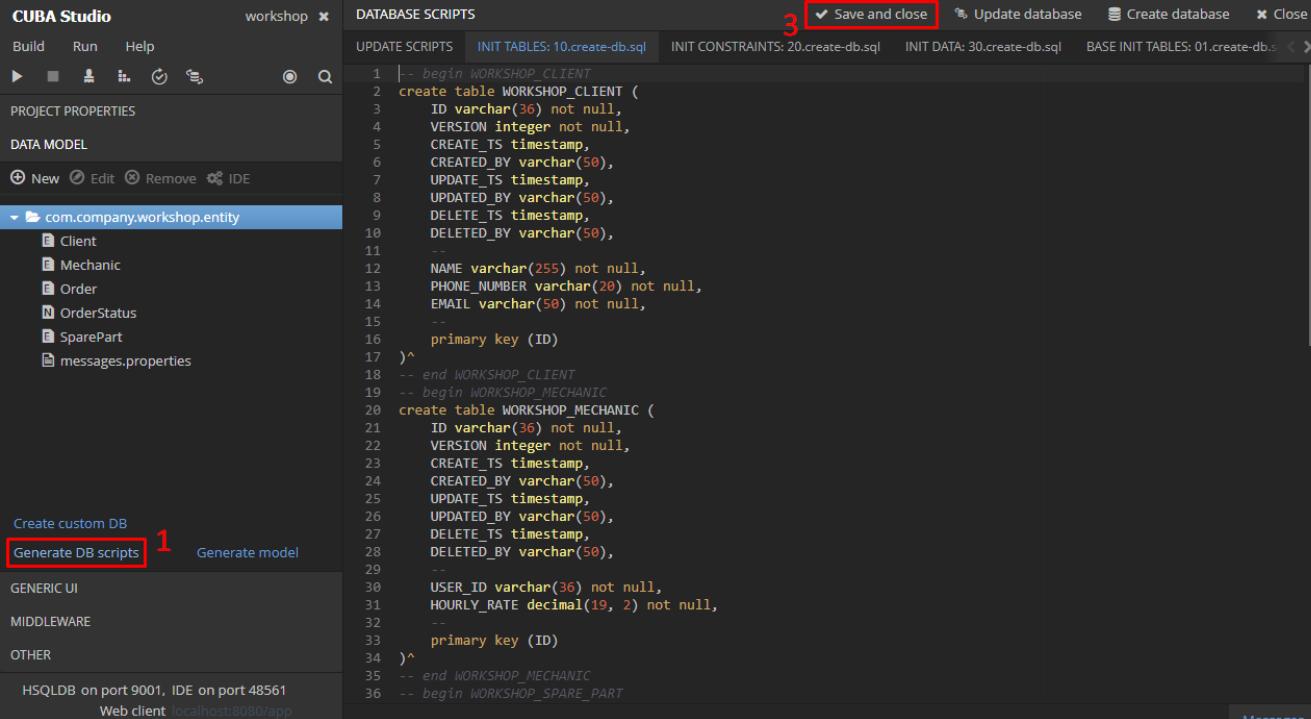
name	type	column
client	Client [workshop\$Client]	CLIENT_ID
mechanic	Mechanic [workshop\$Mechanic]	MECHANIC_ID
description	String	DESCRIPTION
hoursSpent	Integer	HOURS_SPENT
amount	BigDecimal	AMOUNT
parts	SparePart [workshop\$SparePart]	

Buttons: OK (highlighted with a red box), Apply, Show changes, Close.

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



The screenshot shows the CUBA Studio interface with the 'workshop' project open. The left sidebar displays 'PROJECT PROPERTIES' and 'DATA MODEL'. Under 'DATA MODEL', there is a folder named 'com.company.workshop.entity' containing entities like Client, Mechanic, Order, OrderStatus, SparePart, and messages.properties. At the bottom of the sidebar, there are three links: 'Generate custom DB', 'Generate DB scripts' (which is highlighted with a red box and the number 1), and 'Generate model'. The main area is titled 'DATABASE SCRIPTS' and contains a large block of SQL code for creating tables and constraints. At the top of this area, there are several tabs: 'UPDATE SCRIPTS', 'INIT TABLES: 10.create-db.sql' (which is selected and highlighted with a red box and the number 3), 'INIT CONSTRAINTS: 20.create-db.sql', 'INIT DATA: 30.create-db.sql', and 'BASE INIT TABLES: 01.create-db.s'. To the right of these tabs are buttons for 'Save and close', 'Update database', 'Create database', and 'Close'.

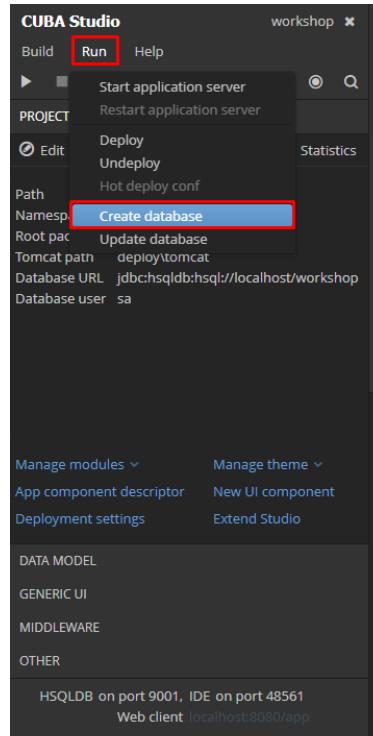
```
1 1-- begin WORKSHOP_CLIENT
2 2create table WORKSHOP_CLIENT (
3 3    ID varchar(36) not null,
4 4    VERSION integer not null,
5 5    CREATE_TS timestamp,
6 6    CREATED_BY varchar(50),
7 7    UPDATE_TS timestamp,
8 8    UPDATED_BY varchar(50),
9 9    DELETE_TS timestamp,
10 10   DELETED_BY varchar(50),
11 11   --
12 12   NAME varchar(255) not null,
13 13   PHONE_NUMBER varchar(20) not null,
14 14   EMAIL varchar(50) not null,
15 15   --
16 16   primary key (ID)
17 17 )^
18 18 -- end WORKSHOP_CLIENT
19 19 -- begin WORKSHOP_MECHANIC
20 20 create table WORKSHOP_MECHANIC (
21 21    ID varchar(36) not null,
22 22    VERSION integer not null,
23 23    CREATE_TS timestamp,
24 24    CREATED_BY varchar(50),
25 25    UPDATE_TS timestamp,
26 26    UPDATED_BY varchar(50),
27 27    DELETE_TS timestamp,
28 28    DELETED_BY varchar(50),
29 29   --
30 30    USER_ID varchar(36) not null,
31 31    HOURLY_RATE decimal(19, 2) not null,
32 32   --
33 33   primary key (ID)
34 34 )^
35 35 -- end WORKSHOP_MECHANIC
36 36 -- begin WORKSHOP_SPARE_PART
```



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.



The screenshot shows the CUBA Studio interface with the 'Run' menu open. The 'Create database' option is highlighted with a red box. The studio window title is 'workshop'. On the right, there are two sections of keyboard shortcuts:

- COMMON SHORTCUTS**
 - (Re)start application server: **Ctrl + Shift + F9**
 - Search: **Alt + /**
 - Previously opened pages: **Ctrl + Shift + H**
 - Toggle help panel: **F1**
 - Toggle left panel: **Ctrl + Shift + /**
- PAGE SHORTCUTS**
 - OK: **Ctrl + Enter**
 - Apply: **Shift + Enter**
 - Close: **Esc**
 - Add a new row to a table: **Ctrl + **

The log output area shows the following process:

```
[14:09:11.391] Creating database jdbc:hsqldb:hsq://localhost/workshop
[14:09:11.391] Executing Gradle task: createDb
:app-core:assembleDbScripts
:app-core:createDb
Using database URL: jdbc:hsqldb:hsq://localhost/workshop, user: sa
Executing SQL: drop schema public cascade;
Using database URL: jdbc:sqlDB:hsq://localhost/workshop, user: sa
Executing SQL script: C:\cuba\workshop\modules\core\build\db\10-cuba\init\hsq\create-db.sql
Executing SQL script: C:\cuba\workshop\modules\core\build\db\20-reports\init\hsq\create-db.sql
Executing SQL script: C:\cuba\workshop\modules\core\build\db\50-workshop\init\hsq\10.create-db.sql
Executing SQL script: C:\cuba\workshop\modules\core\build\db\50-workshop\init\hsq\20.create-db.sql
Executing SQL script: C:\cuba\workshop\modules\core\build\db\50-workshop\init\hsq\30.create-db.sql
```

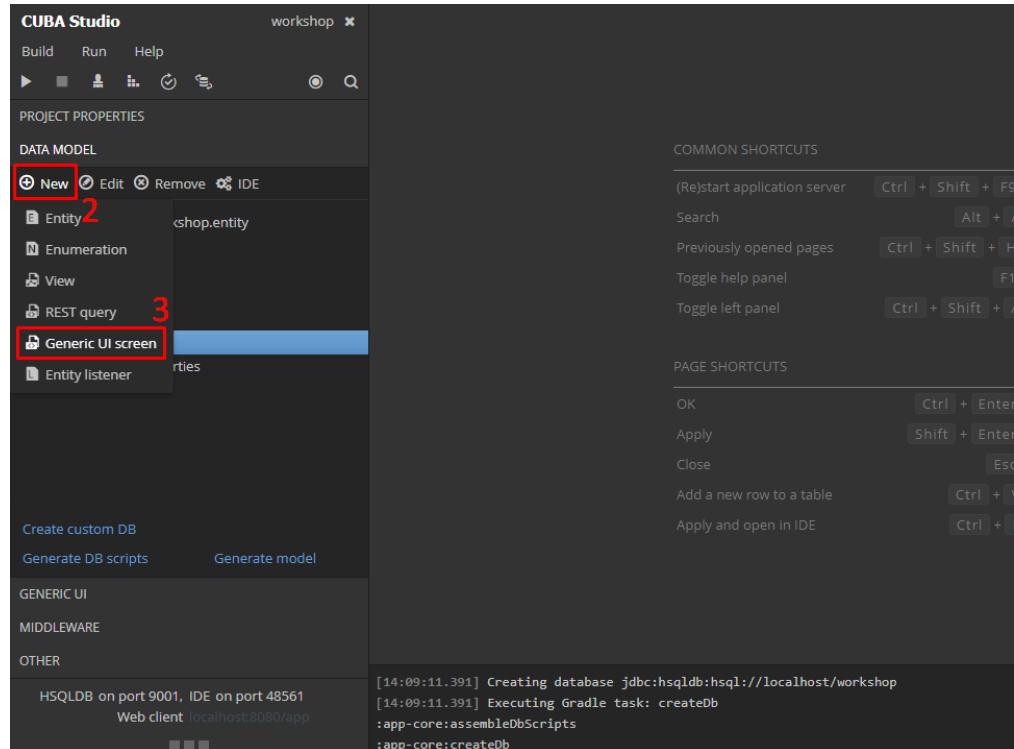
At the bottom, the message **BUILD SUCCESSFUL** is displayed in green, along with the total time taken: **Total time: 1.581 secs**.

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**



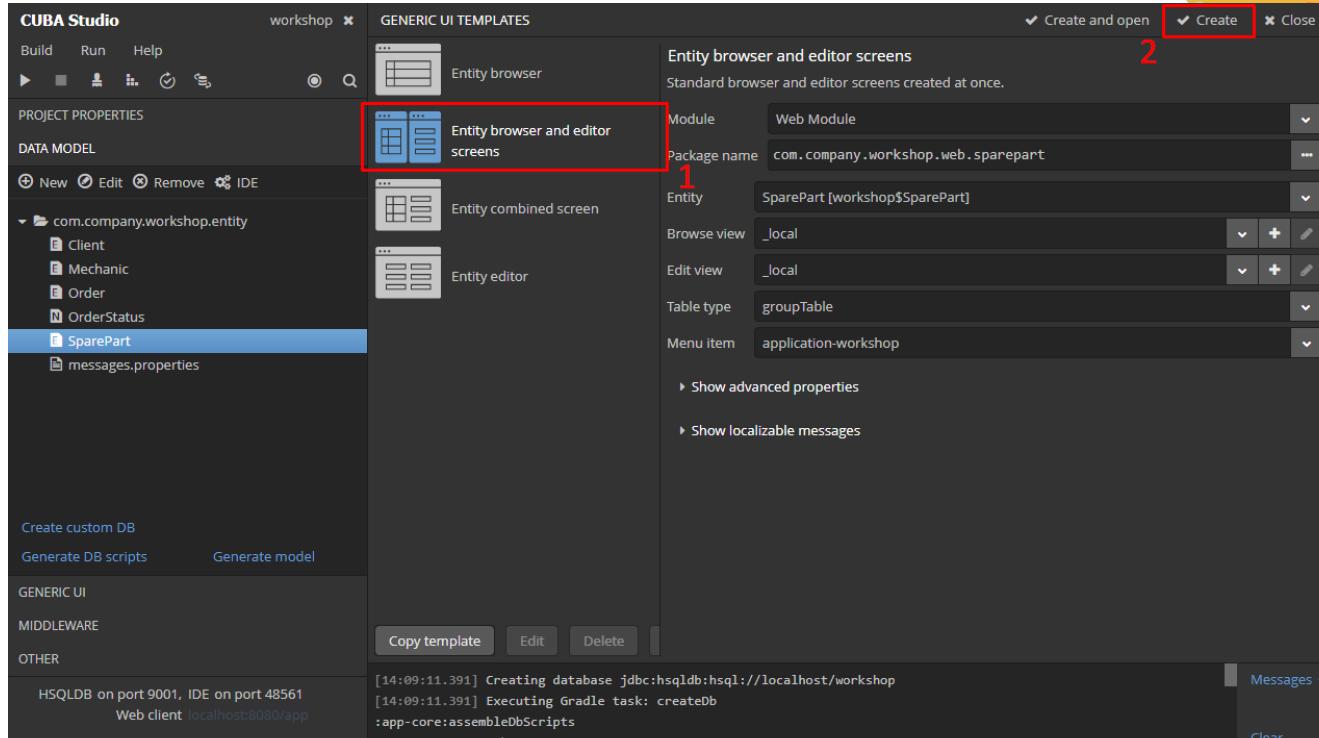
Screens scaffolding

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

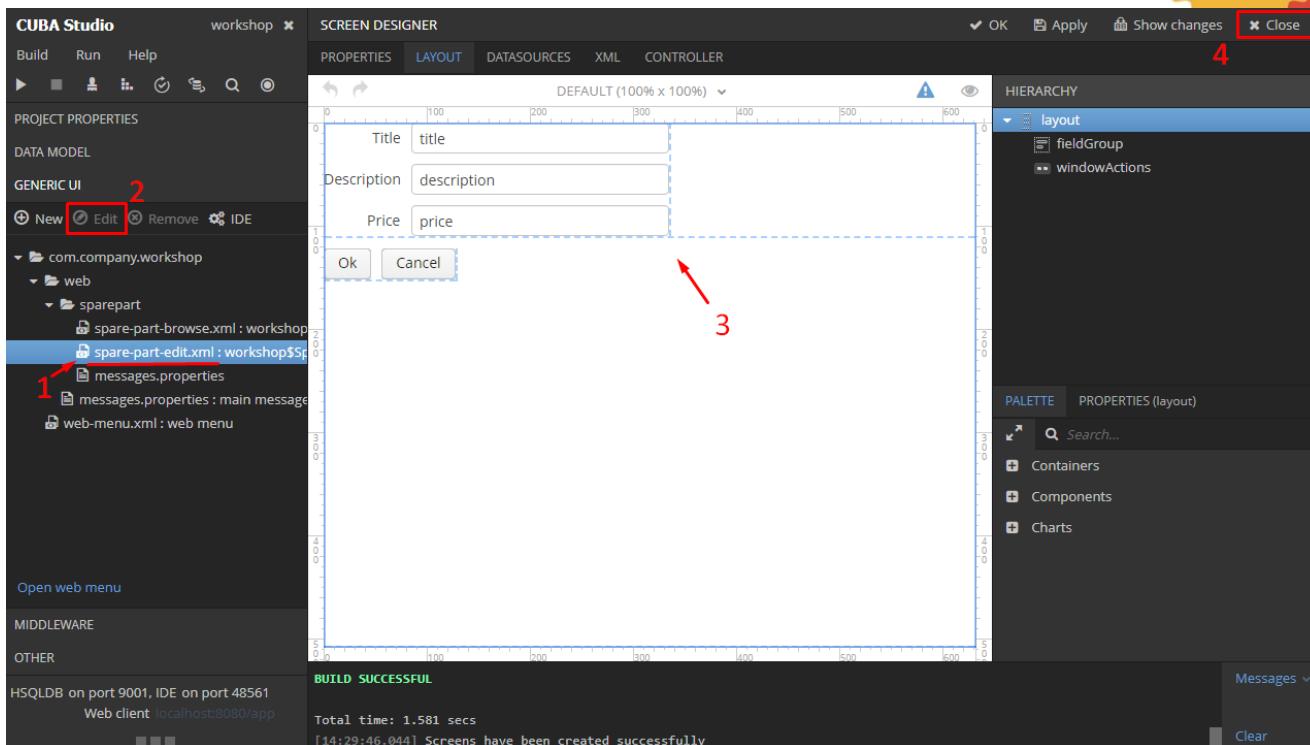
1. Select the **Entity browser and editor screens** template
2. Click **Create**



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**

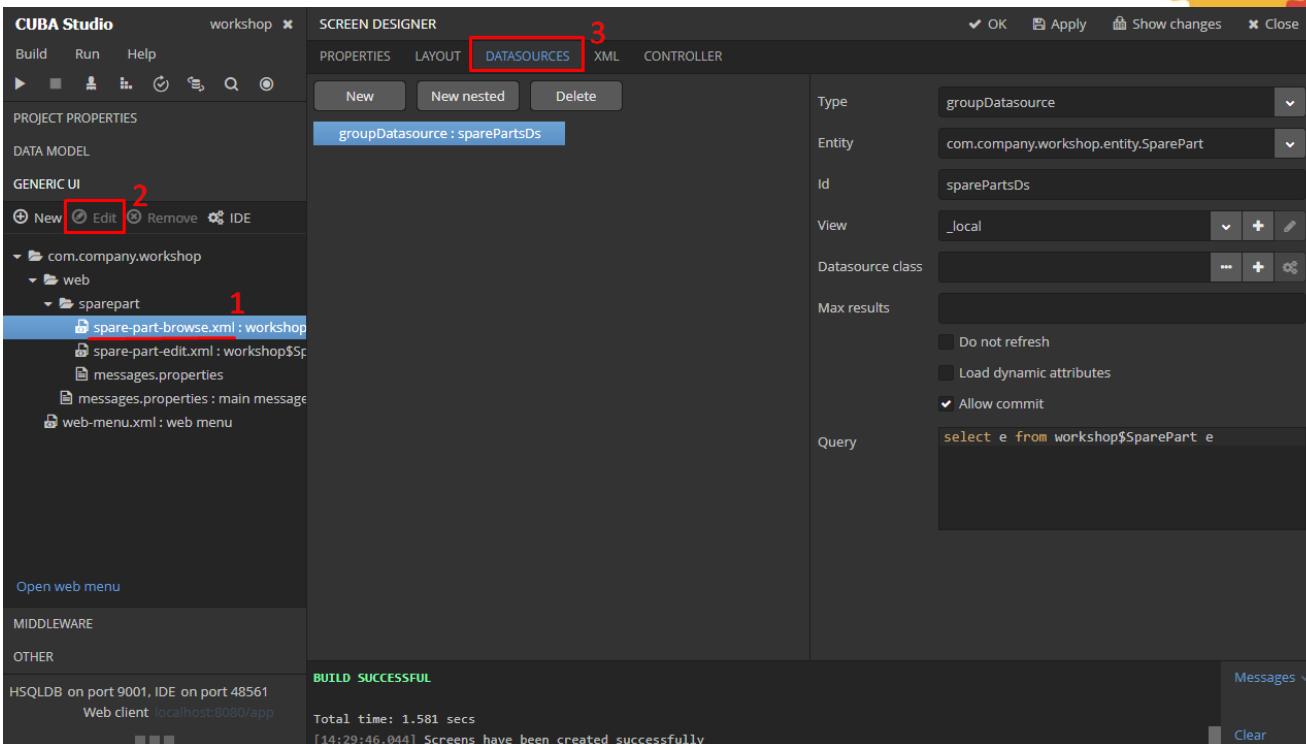


Data binding

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

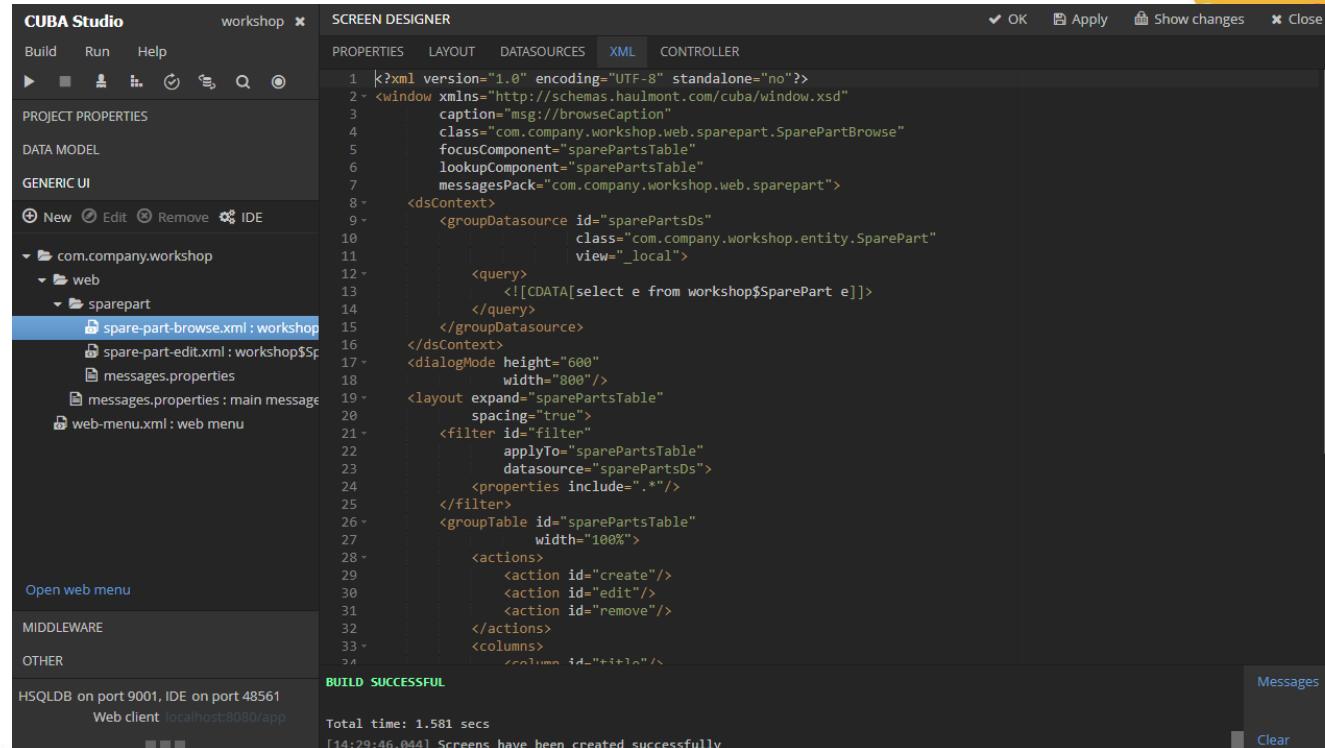
Datasources use JPQL queries to load data.



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.



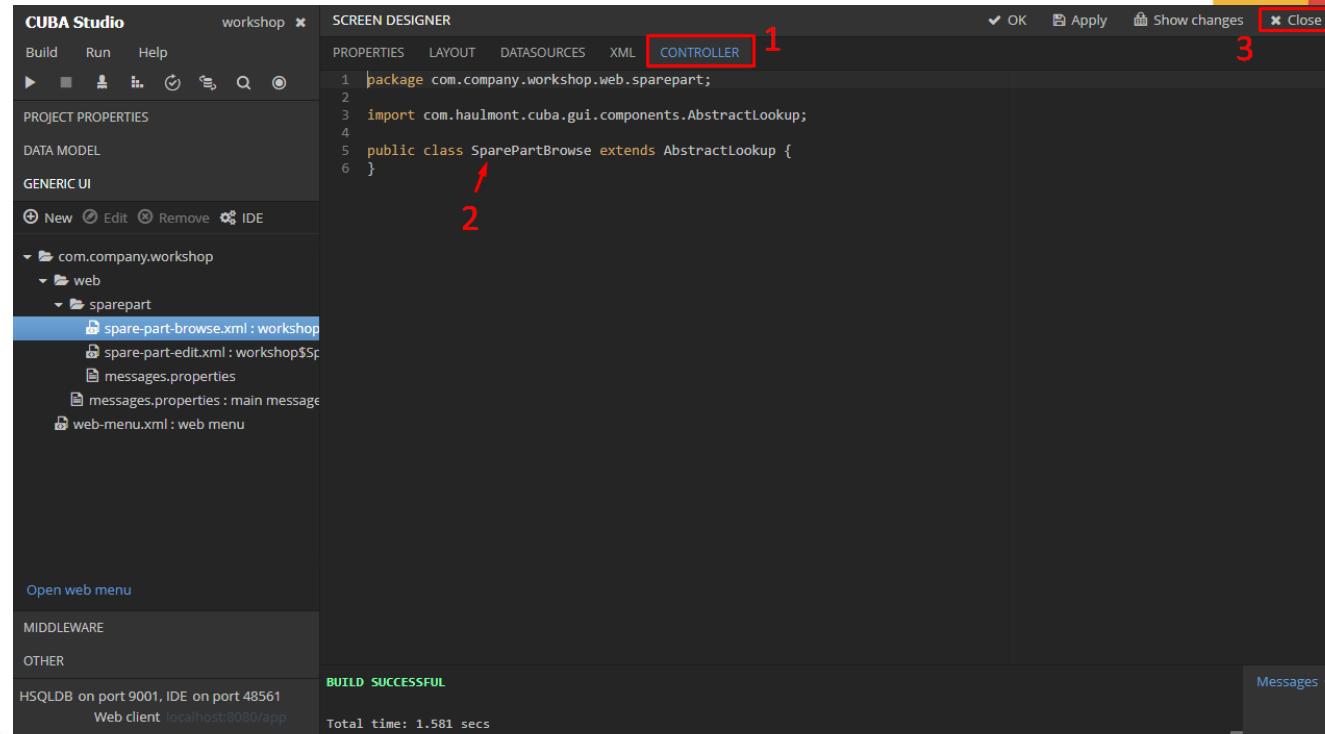
The screenshot shows the CUBA Studio interface with the "workshop" project selected. In the left sidebar, under "PROJECT PROPERTIES", "DATA MODEL", and "GENERIC UI", there are buttons for "New", "Edit", "Remove", and "IDE". Below these, the "com.company.workshop" package is expanded, showing "web" and "sparepart" subfolders. The "sparepart" folder contains three files: "spare-part-browse.xml" (selected), "spare-part-edit.xml", and "messages.properties". The "messages.properties" file is also listed under the main "messages.properties" entry. At the bottom of the sidebar, there are buttons for "Open web menu", "MIDDLEWARE", and "OTHER". The "MIDDLEWARE" section shows "HSQLDB on port 9001, IDE on port 48561" and "Web client localhost:8080/app". The right side of the screen is the "SCREEN DESIGNER" tab, which has tabs for "PROPERTIES", "LAYOUT", "DATASOURCES", "XML" (which is selected), and "CONTROLLER". The XML code is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<window xmlns="http://schemas.haulmont.com/cuba/window.xsd"
    caption="msg://browseCaption"
    class="com.company.workshop.web.sparepart.SparePartBrowse"
    focusComponent="sparePartsTable"
    lookupComponent="sparePartsTable"
    messagesPack="com.company.workshop.web.sparepart">
    <dsContext>
        <groupDataSource id="sparePartsDs"
            class="com.company.workshop.entity.SparePart"
            view="_local">
            <query>
                <![CDATA[select e from workshop$SparePart e]]>
            </query>
        </groupDataSource>
    </dsContext>
    <dialogMode height="600"
        width="800"/>
    <layout expand="sparePartsTable"
        spacing="true">
        <filter id="filter"
            applyTo="sparePartsTable"
            datasource="sparePartsDs">
            <properties include=".*"/>
        </filter>
        <groupTable id="sparePartsTable"
            width="100%">
            <actions>
                <action id="create"/>
                <action id="edit"/>
                <action id="remove"/>
            </actions>
            <columns>
                <column id="+id+>" />
            </columns>
        </groupTable>
    </layout>
</window>
```

At the bottom of the XML tab, there are buttons for "OK", "Apply", "Show changes", and "Close". The status bar at the bottom of the screen shows "BUILD SUCCESSFUL" and "Total time: 1.581 secs [14:29:46.044] Screens have been created successfully".

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

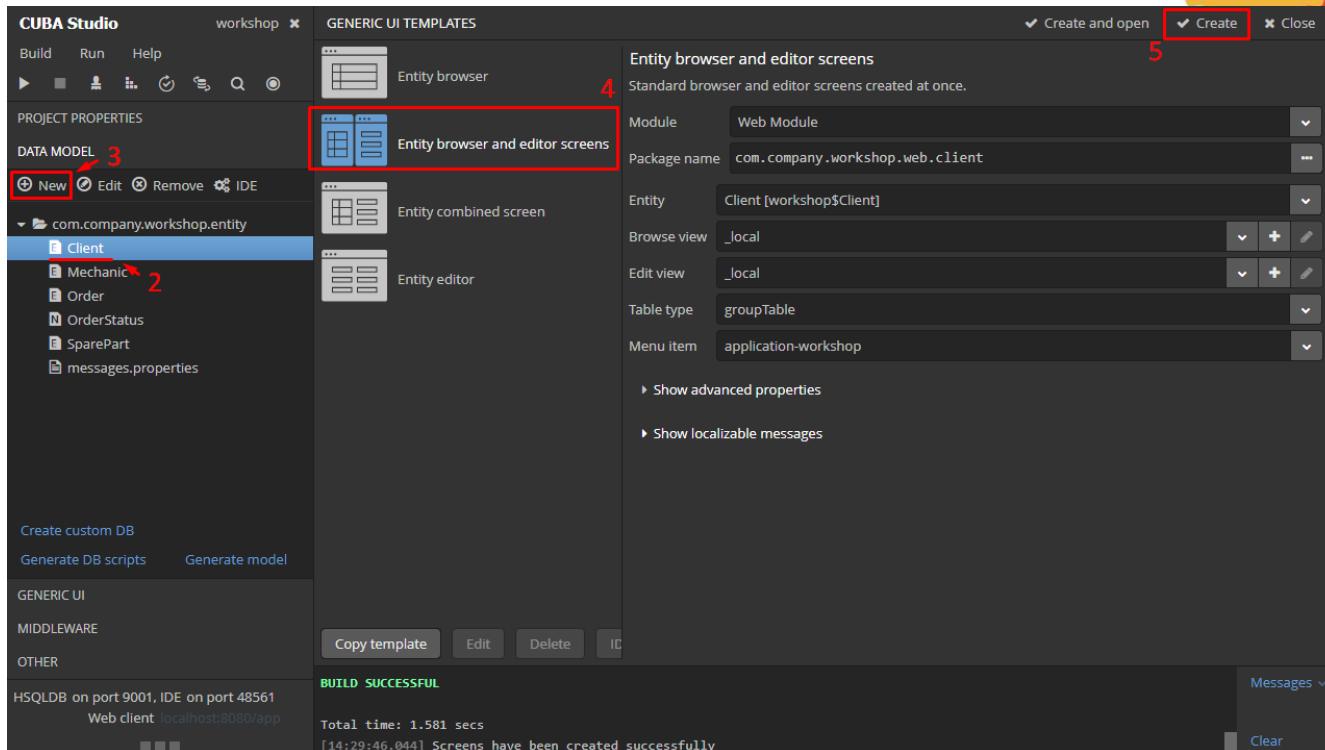
2. Select the **Client** entity

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.

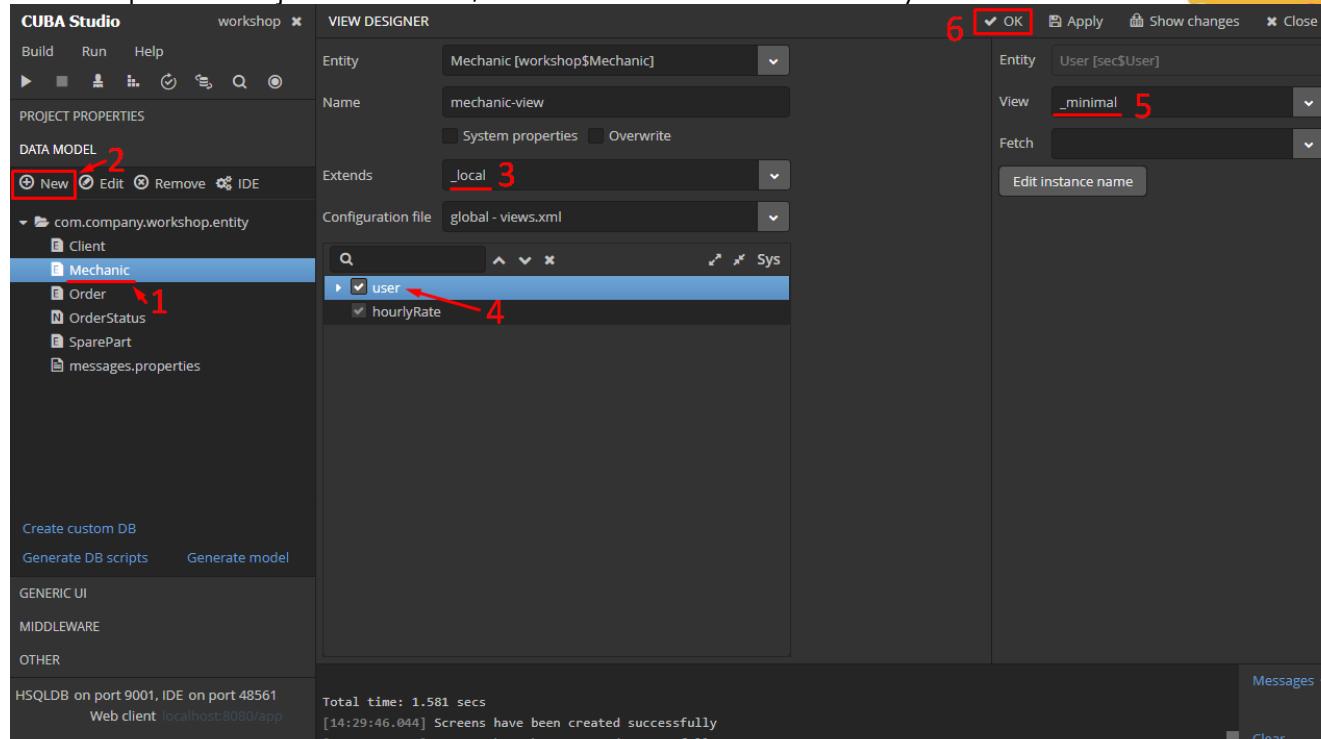
Let's create a view for the **Mechanic** entity, which will include **User**.

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

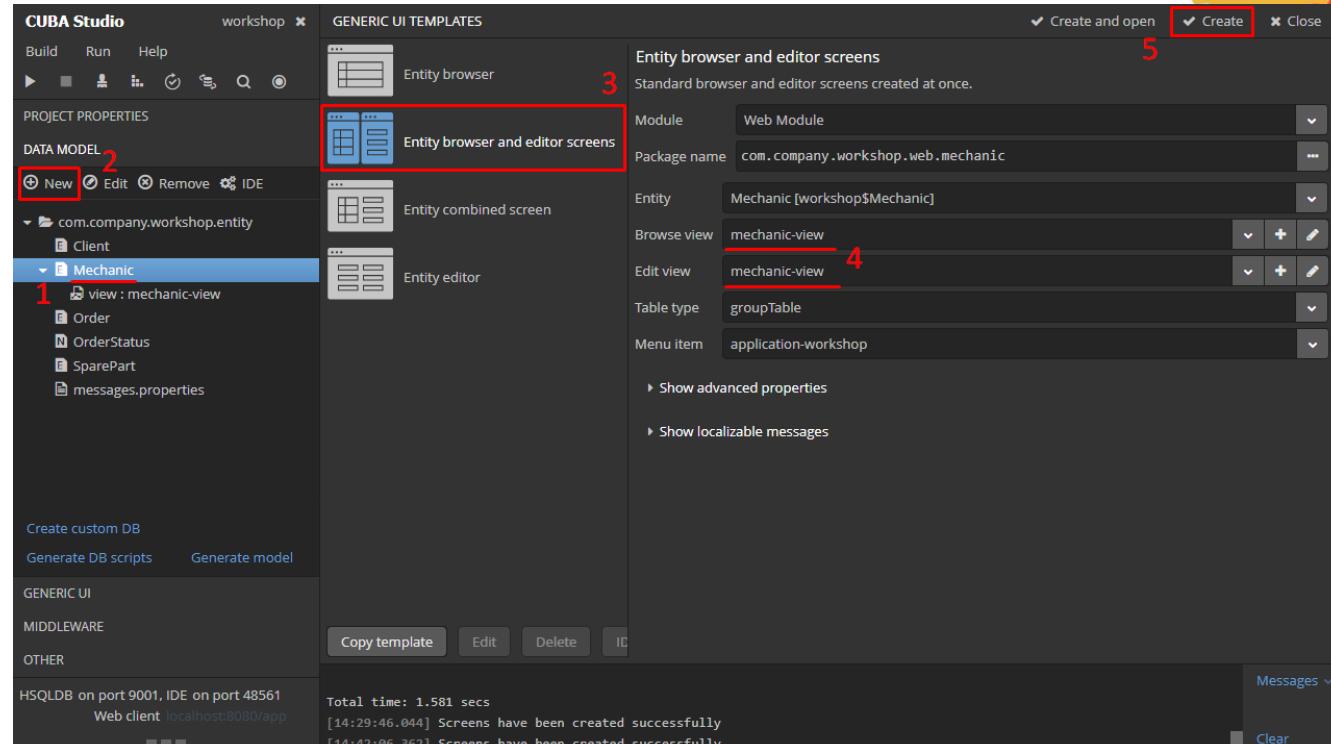
Instance Name of an entity

6. Click **OK** to save the view



Generate screens for Mechanic

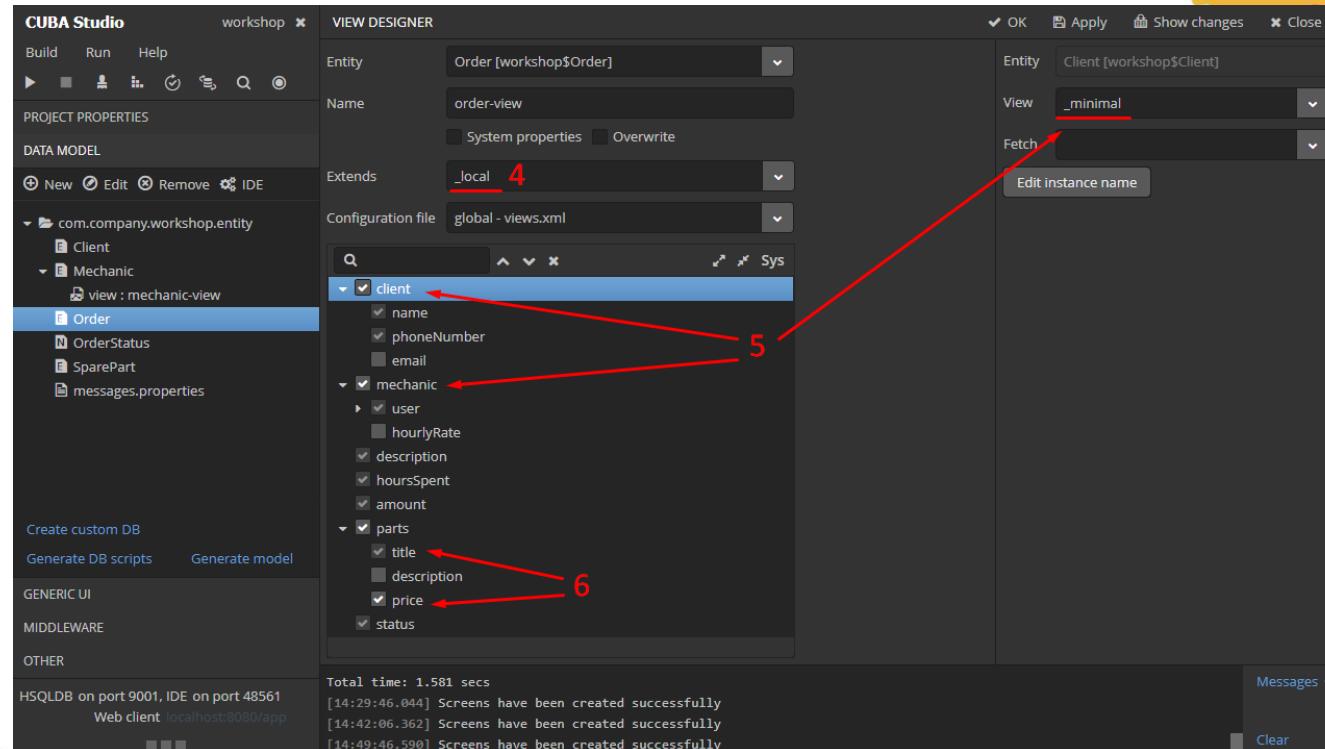
1. Select the **Mechanic** entity
2. Click **New -> Generic UI screen**
3. Select **Entity browser and editor screens** template
4. Choose **mechanic-view** for both browser and editor screen
5. Click **Create**



View for Order browser and editor

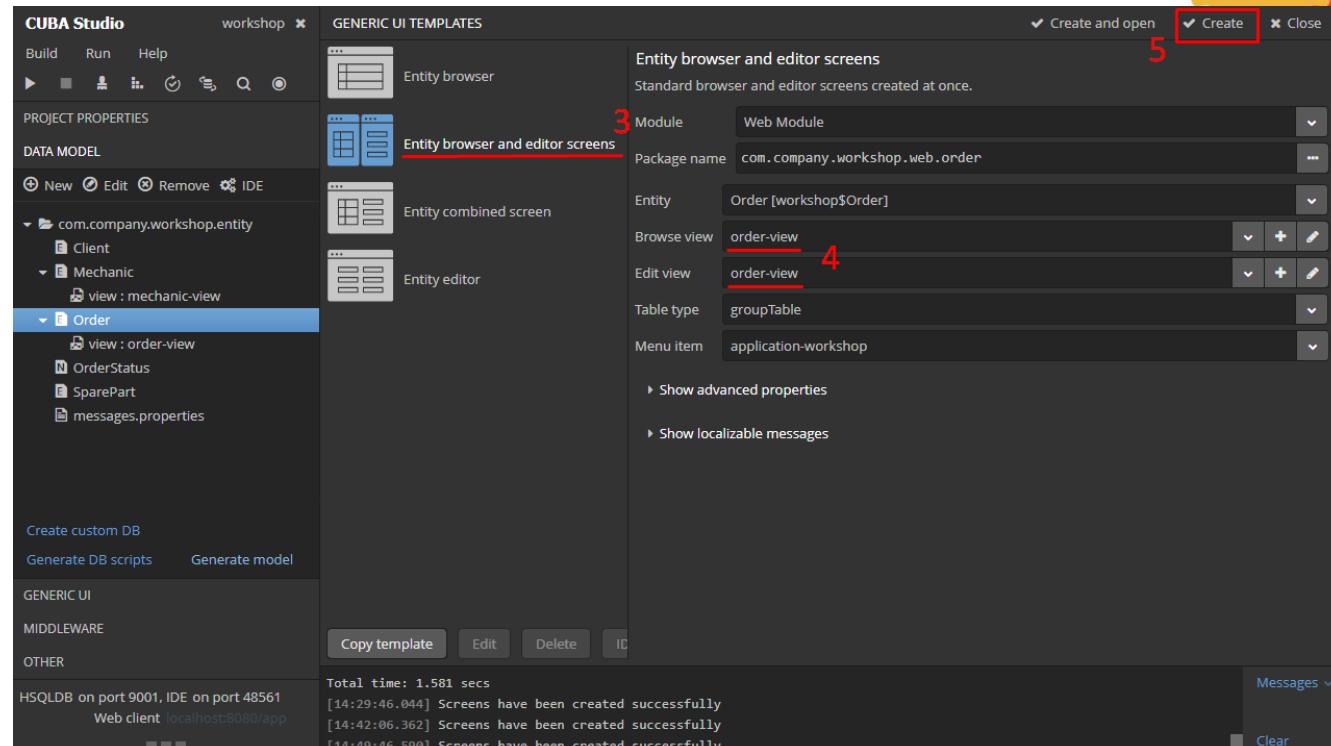
Now we need to create screens for the **Order** entity. We'll also need to create a special view.

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

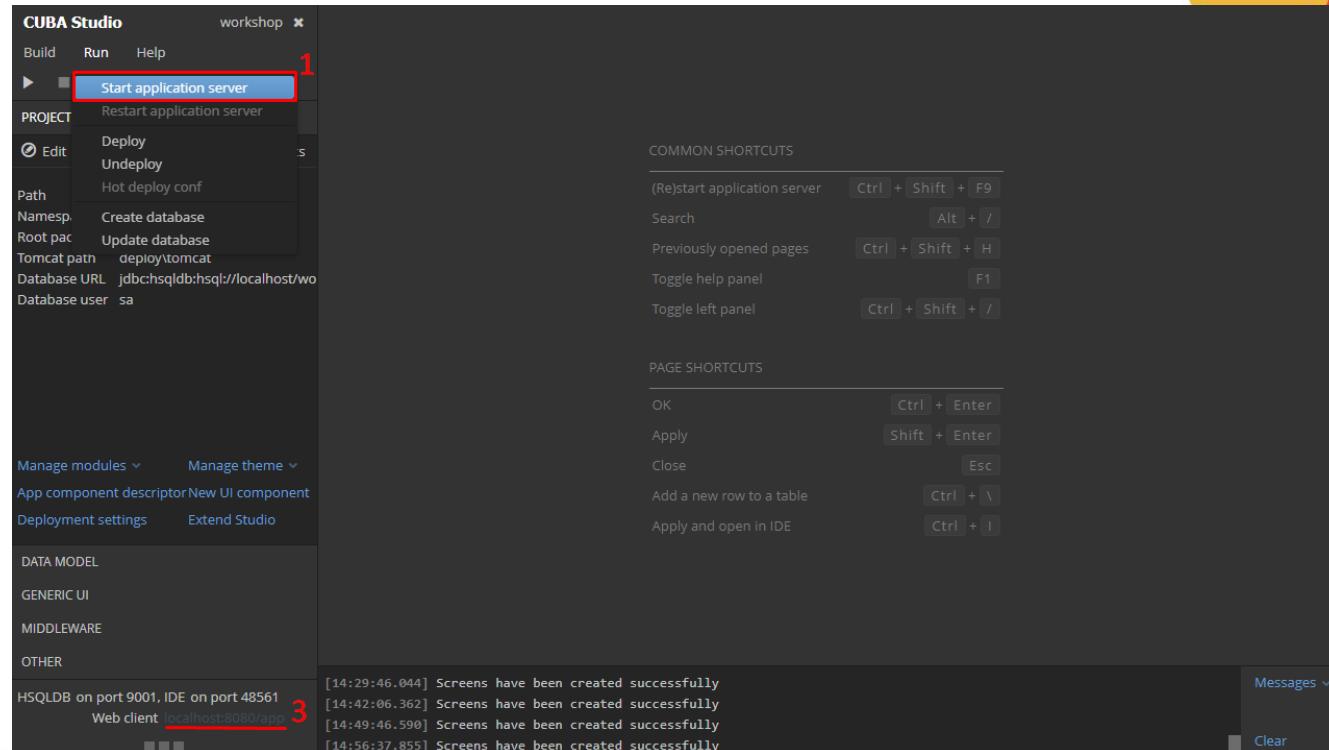
Our application **is done**, of course, to a first approximation.

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.

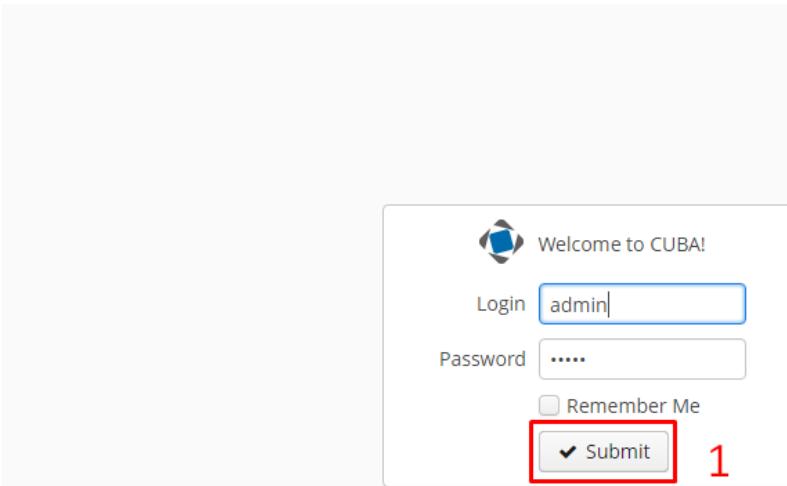


First launch and CRUD

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



Welcome to CUBA!

Login

Password

Remember Me

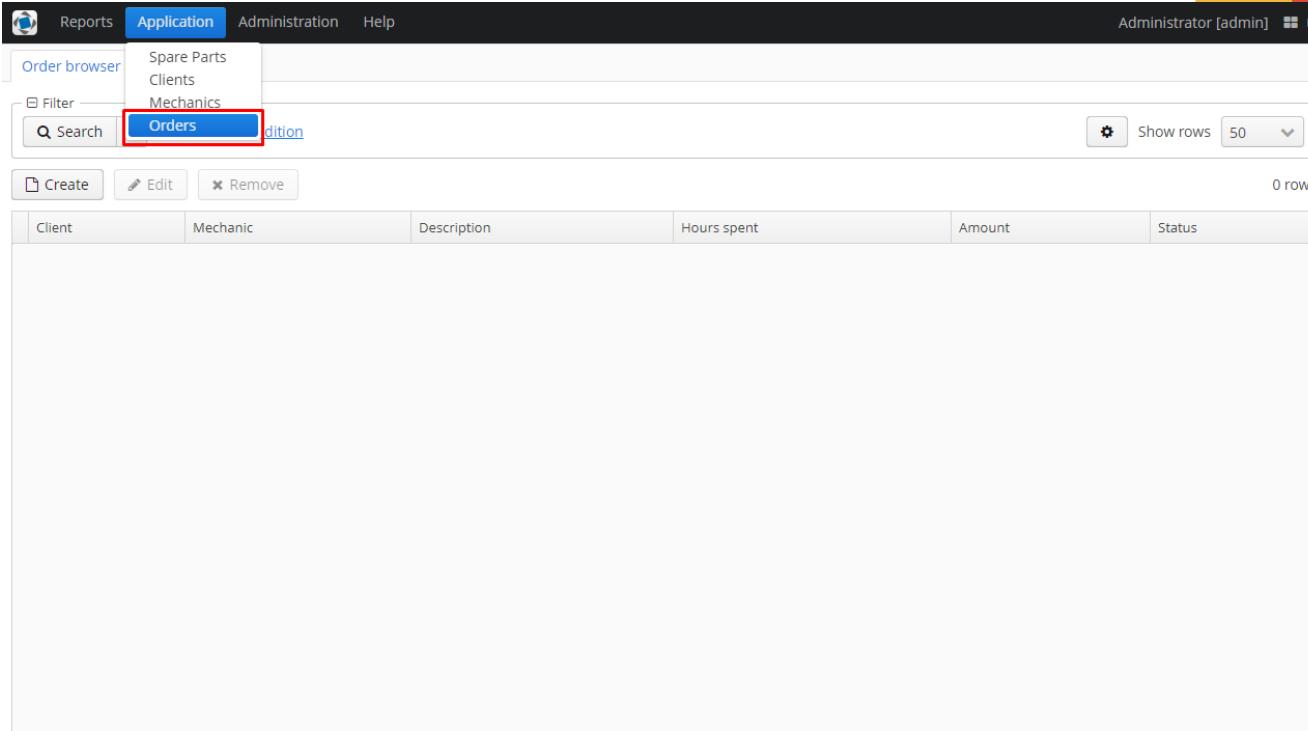
Submit 1

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.



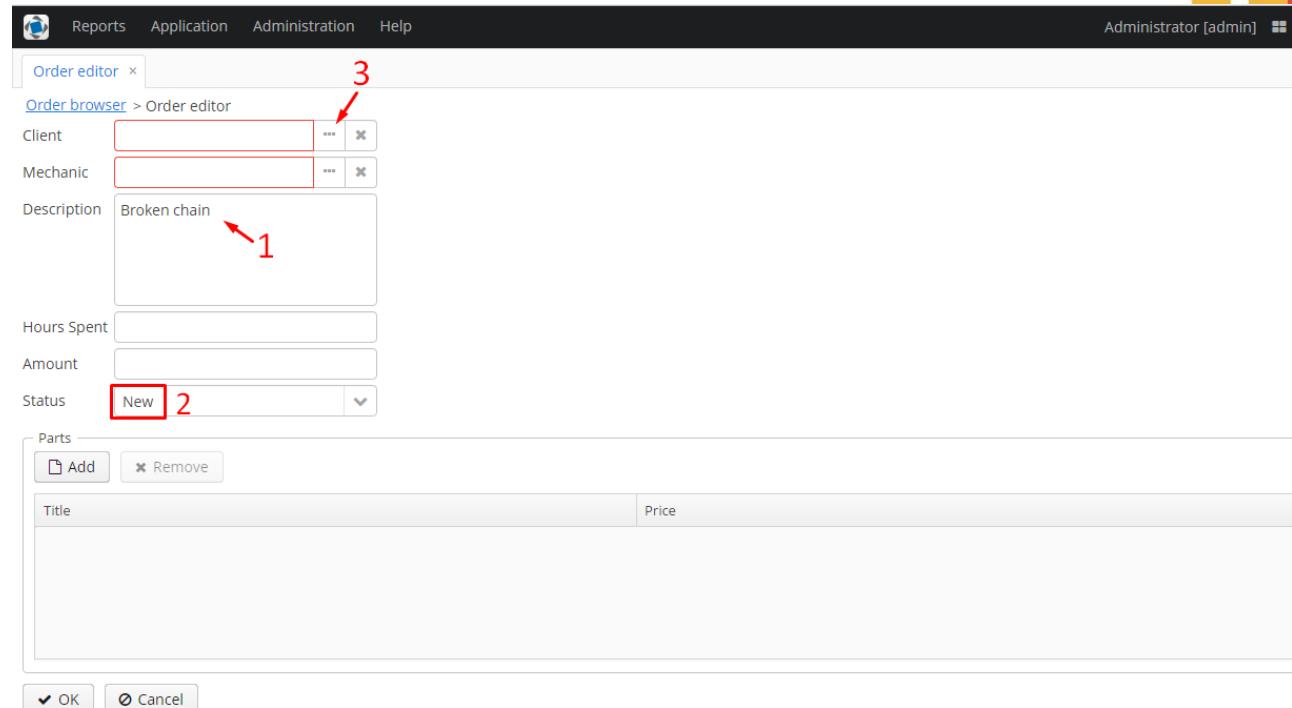
The screenshot shows a standard web-based application interface. At the top, there is a navigation bar with tabs: 'Reports' (disabled), 'Application' (selected), 'Administration', and 'Help'. On the right side of the navigation bar, it says 'Administrator [admin]' and has a user icon. Below the navigation bar is a sidebar with a tree view containing 'Order browser' (selected), 'Spare Parts', 'Clients', and 'Mechanics'. Under 'Order browser', there is a 'Filter' button and a 'Search' input field. A red box highlights the 'Orders' button in the sidebar. Below the sidebar is a toolbar with 'Create', 'Edit', and 'Remove' buttons. To the right of the toolbar is a 'Show rows' dropdown set to '50'. The main area is a table with the following columns: Client, Mechanic, Description, Hours spent, Amount, and Status. The table currently displays 0 rows. The table header row has thin borders, while the data rows have thick borders.

Order edit screen

1. Click **Create** and enter the **description**

2. Select Status: **New**

3. Click button [...] to select a **client** for the order

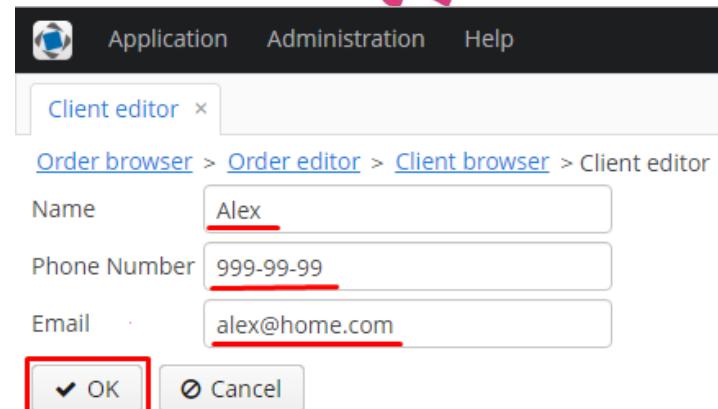


The screenshot shows the 'Order editor' screen. At the top, there's a navigation bar with 'Reports', 'Application', 'Administration', and 'Help'. On the right, it says 'Administrator [admin]'. Below the navigation, the title 'Order editor' is displayed along with the breadcrumb 'Order browser > Order editor'. The main form contains several fields: 'Client' (with a red box and arrow 3), 'Mechanic' (with a red box and arrow 3), 'Description' (with a red box and arrow 1), 'Hours Spent' (empty), 'Amount' (empty), and 'Status' (set to 'New', with a red box and arrow 2). Below the form is a section for 'Parts' with 'Add' and 'Remove' buttons, and a table for listing parts with columns 'Title' and 'Price'. At the bottom are 'OK' and 'Cancel' buttons.

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**](mailto:alex@test.com)
3. Click **OK**
4. Click **Select** to set client to the order

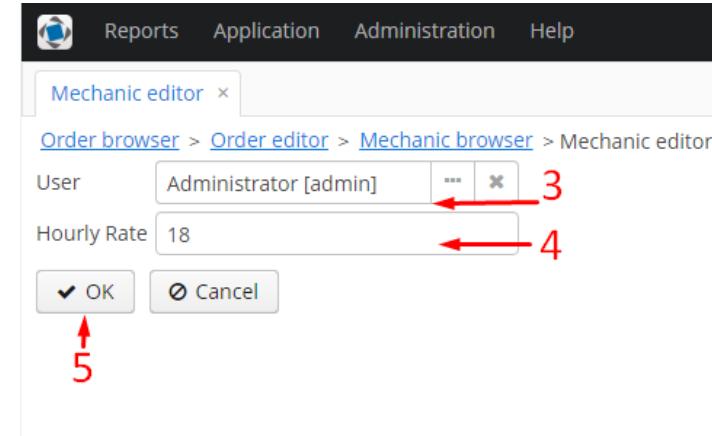


Assign mechanic for the order

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.



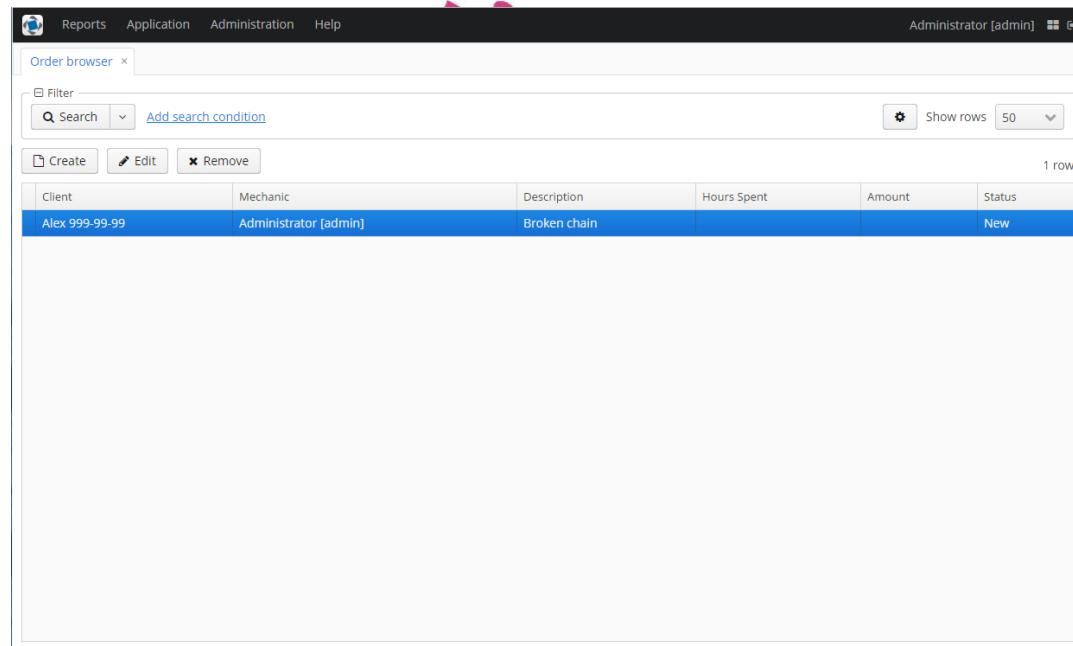
CRUD application

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.



The screenshot shows a desktop application window titled "Order browser". The window has a dark header bar with the title and standard window controls. Below the header is a toolbar with buttons for "Filter", "Search", "Add search condition", "Create", "Edit", and "Remove". To the right of the toolbar are buttons for "Show rows" (set to 50) and a refresh icon. The main area is a table with the following data:

Client	Mechanic	Description	Hours Spent	Amount	Status
Alex 999-99-99	Administrator [admin]	Broken chain			New

Integration with IDE and project structure

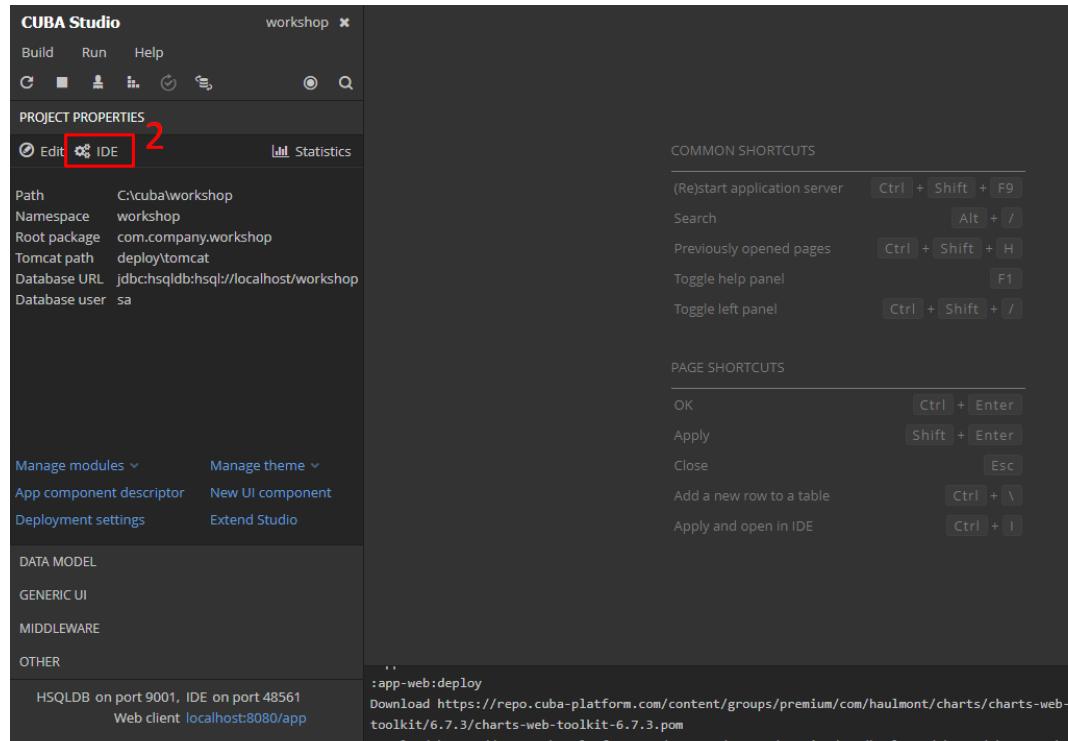
Go to the IDE

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

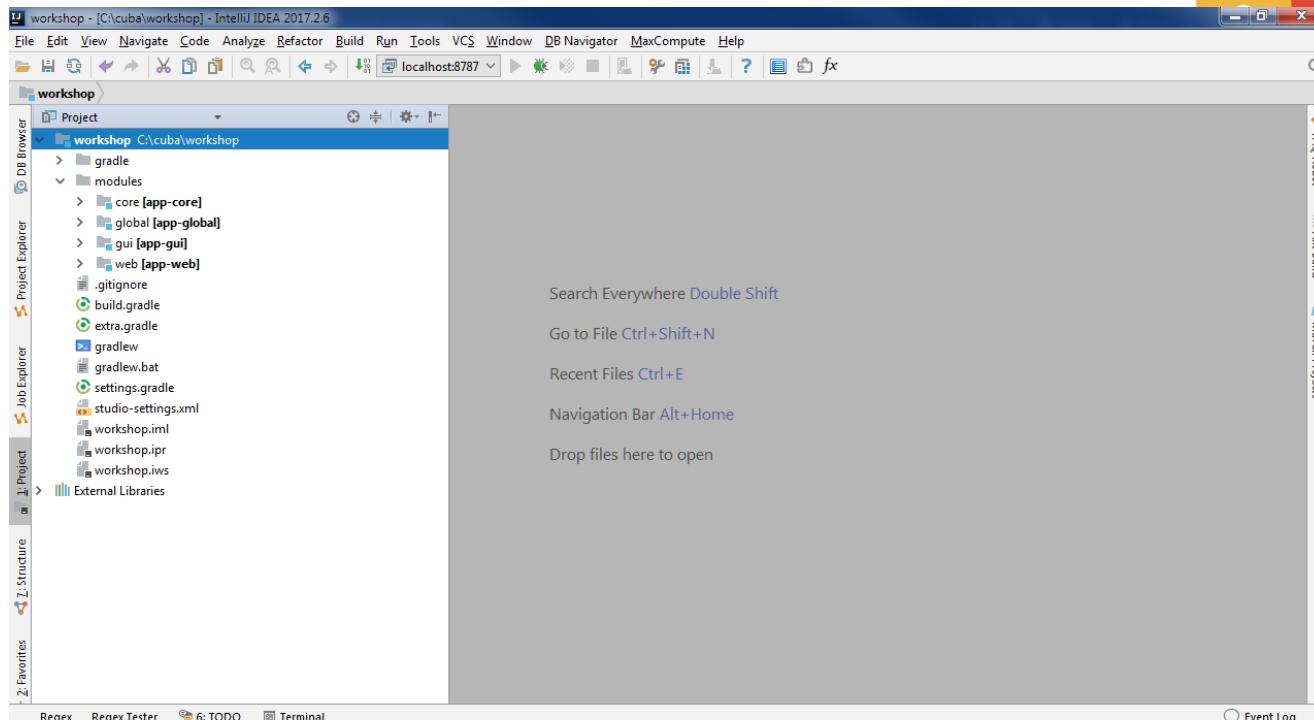
By default, any project consists of 4 modules: **global**, **core**, **web**, **gui**.

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.

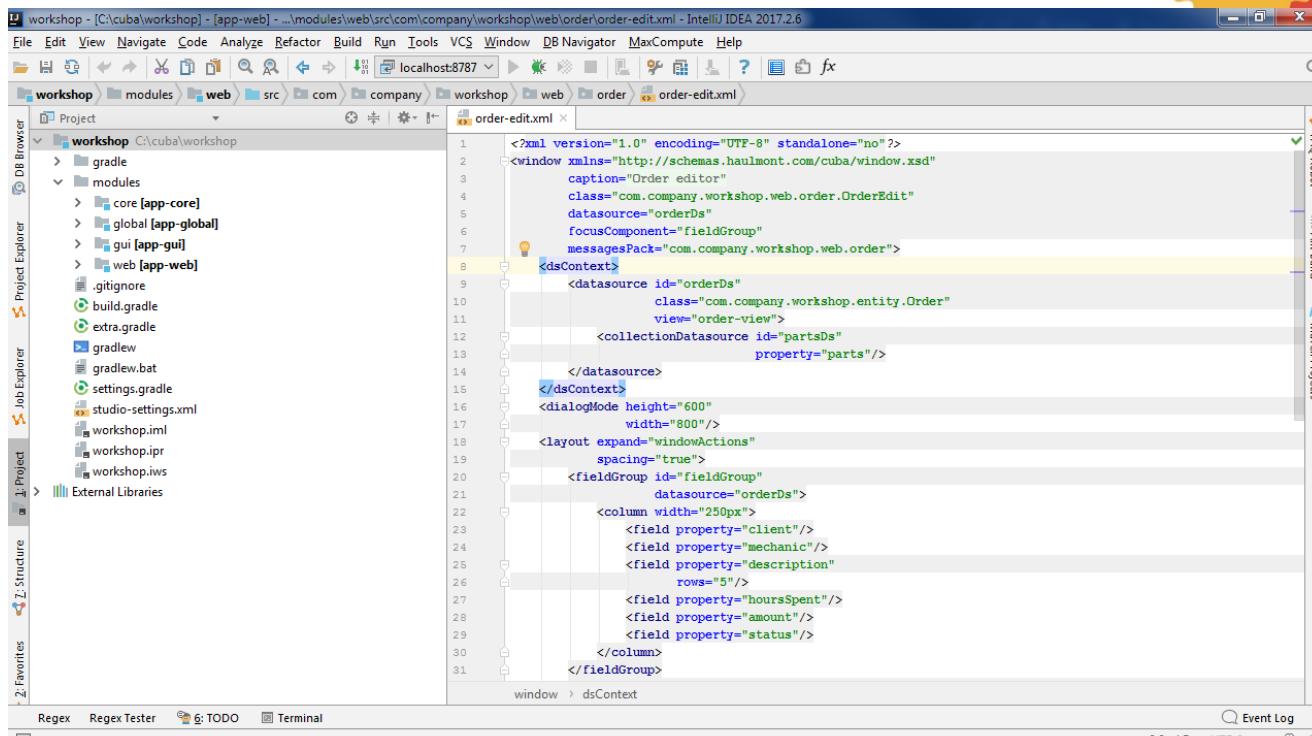


CUBA Studio IDE integration

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).



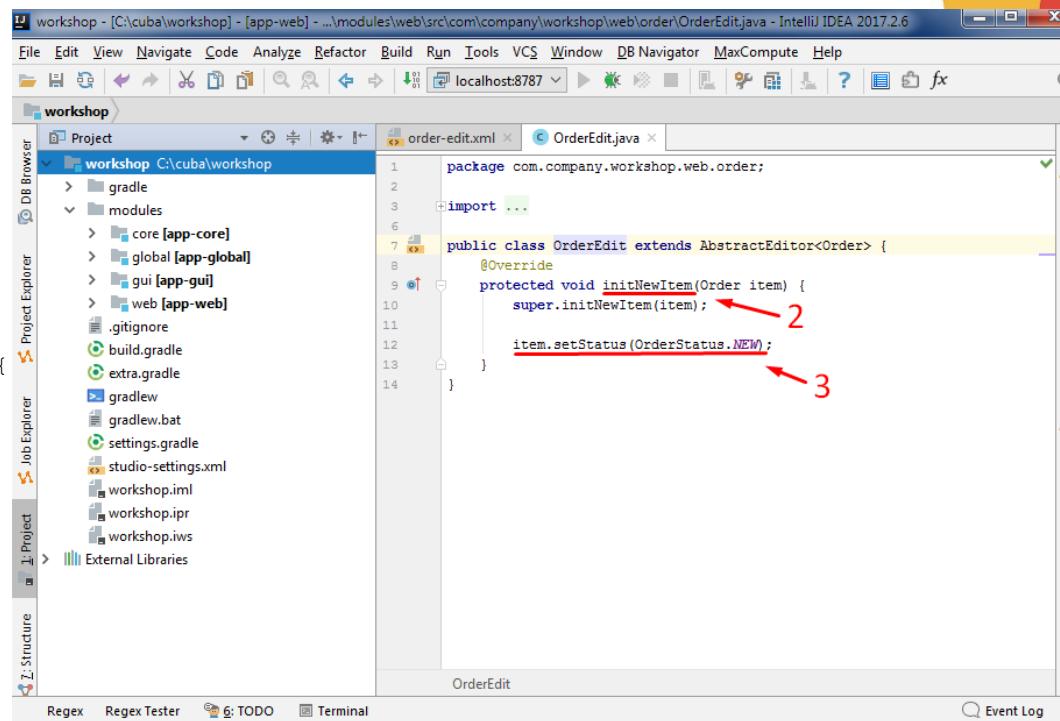
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<window xmlns="http://schemas.haulmont.com/cuba/window.xsd"
         caption="Order editor"
         class="com.company.workshop.web.order.OrderEdit"
         datasource="orderDs"
         focusComponent="fieldGroup"
         messagesPack="com.company.workshop.web.order">
    <dsContext>
        <datasource id="orderDs"
                    class="com.company.workshop.entity.Order"
                    view="order-view">
            <collectionDatasource id="partsDs"
                property="parts"/>
        </datasource>
    </dsContext>
    <dialogMode height="600"
               width="800"/>
    <layout expand="windowActions"
           spacing="true">
        <fieldGroup id="fieldGroup"
                   datasource="orderDs">
            <column width="250px">
                <field property="client"/>
                <field property="mechanic"/>
                <field property="description"
                      rows="5"/>
                <field property="hoursSpent"/>
                <field property="amount"/>
                <field property="status"/>
            </column>
        </fieldGroup>
    </layout>
</window> > dsContext
```

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

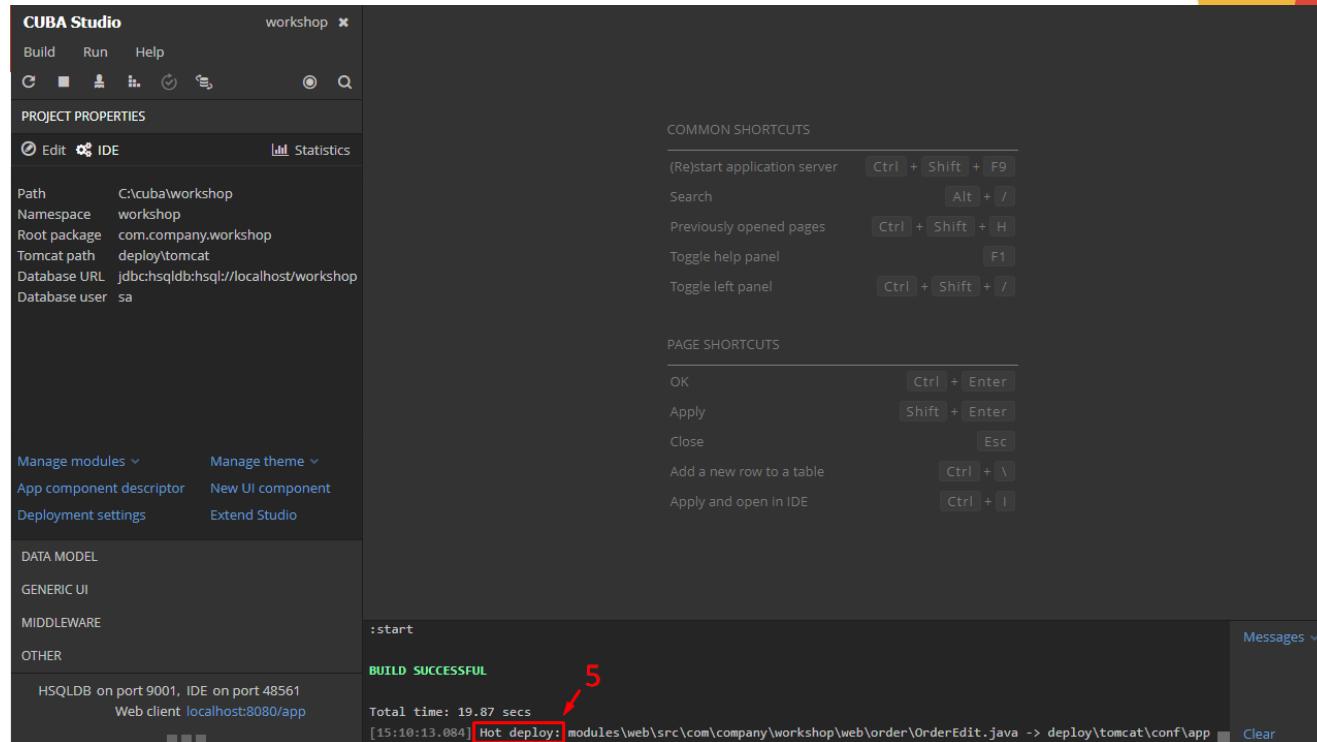
```
public class OrderEdit extends AbstractEditor<Order> {  
    @Override  
    protected void initNewItem(Order item) {  
        super.initNewItem(item);  
  
        item.setStatus(OrderStatus.NEW);  
    }  
}
```



```
package com.company.workshop.web.order;  
import ...  
  
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

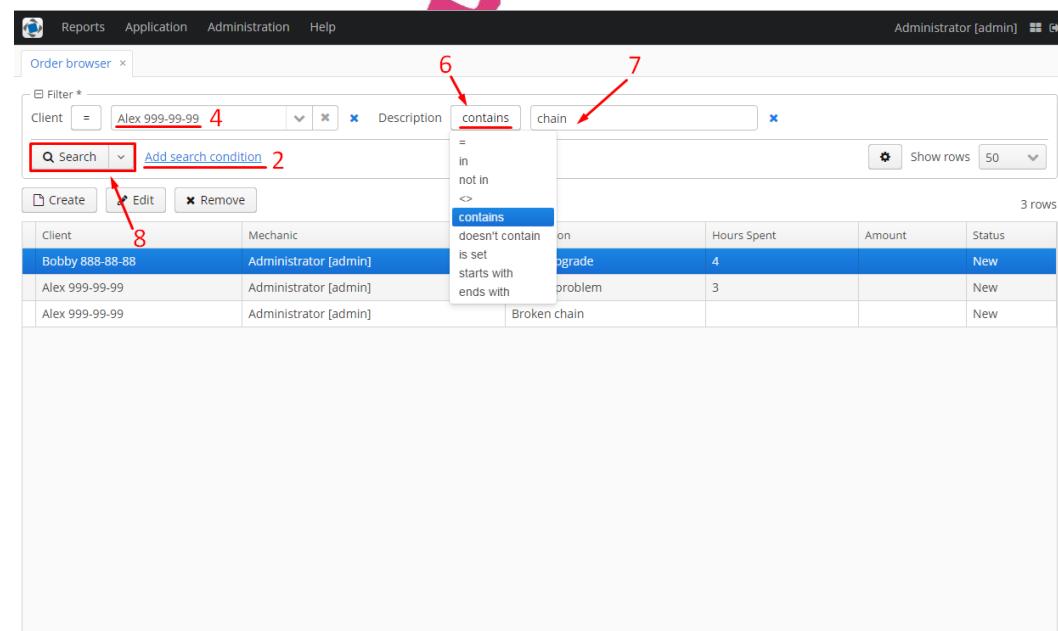


Filter component

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.



The screenshot shows a 'Order browser' window with a filter dialog open. The filter criteria are:

- Client = Alex 999-99-99 (highlighted with red box 4)
- Description contains chain (highlighted with red box 6)

Below the filter, there are buttons for Create, Edit, and Remove. A dropdown menu is open over the 'contains' button, showing various operators: =, in, not in, <>, doesn't contain, is set, starts with, ends with. The 'contains' option is highlighted with a red box 7. A red box 8 highlights the 'Search' button. The results table shows three rows of data:| Client | Mechanic | On | Hours Spent | Amount | Status |
| --- | --- | --- | --- | --- | --- |
| Bobby 888-88-88 | Administrator [admin] | Upgrade | 4 | | New |
| Alex 999-99-99 | Administrator [admin] | problem | 3 | | New |
| Alex 999-99-99 | Administrator [admin] | Broken chain | | | New |

Actions

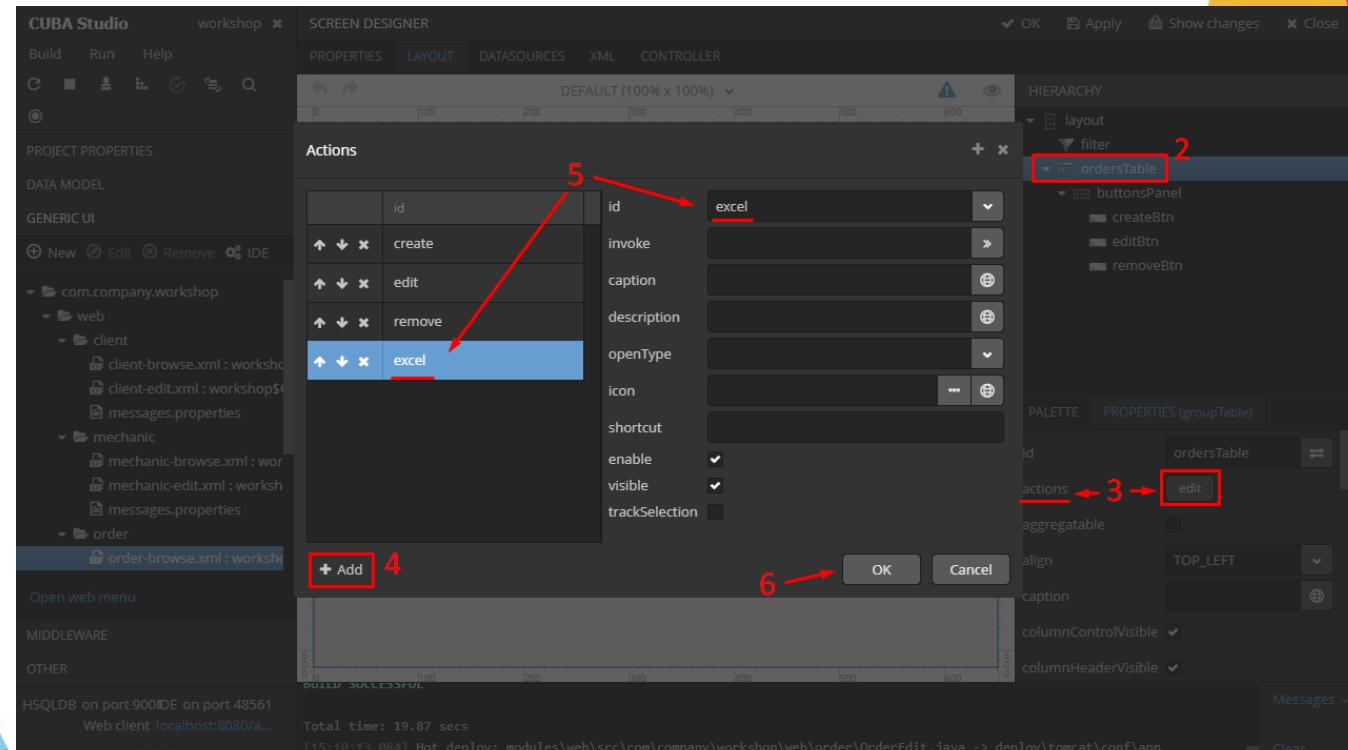


Standard actions

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

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

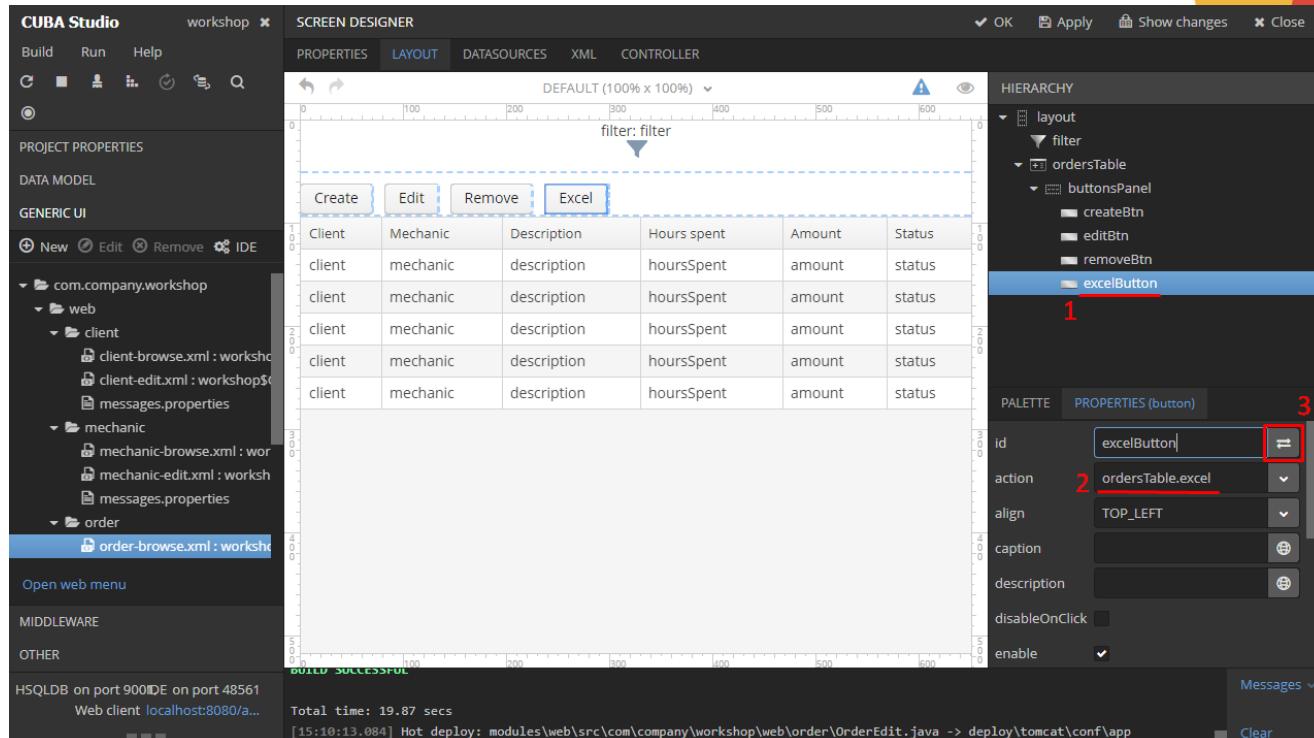
1. Open **order-browse.xml** screen in Studio.
2. Select table component, go to properties panel
3. Click the **edit** button in the **actions** property
4. Add a new action row to the list
5. Specify id as **excel** for this action
6. Click **OK**



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.



The screenshot shows the CUBA Studio interface with the following details:

- Screen Designer:** Displays a table with columns: Client, Mechanic, Description, Hours spent, Amount, Status. The "Excel" button in the toolbar is highlighted.
- Hierarchy Panel:** Shows the component tree:
 - layout
 - filter
 - ordersTable
 - buttonsPanel
 - createBtn
 - editBtn
 - removeBtn
 - excelButton** (highlighted with a red border)
- Properties Panel:** For the **excelButton**:
 - id:** excelButton (highlighted with a red border)
 - action:** **ordersTable.excel** (highlighted with a red border)
 - align:** TOP_LEFT
 - caption:** (empty)
 - description:** (empty)
 - disableOnClick:** (checkbox)
 - enable:** (checkbox)

Security



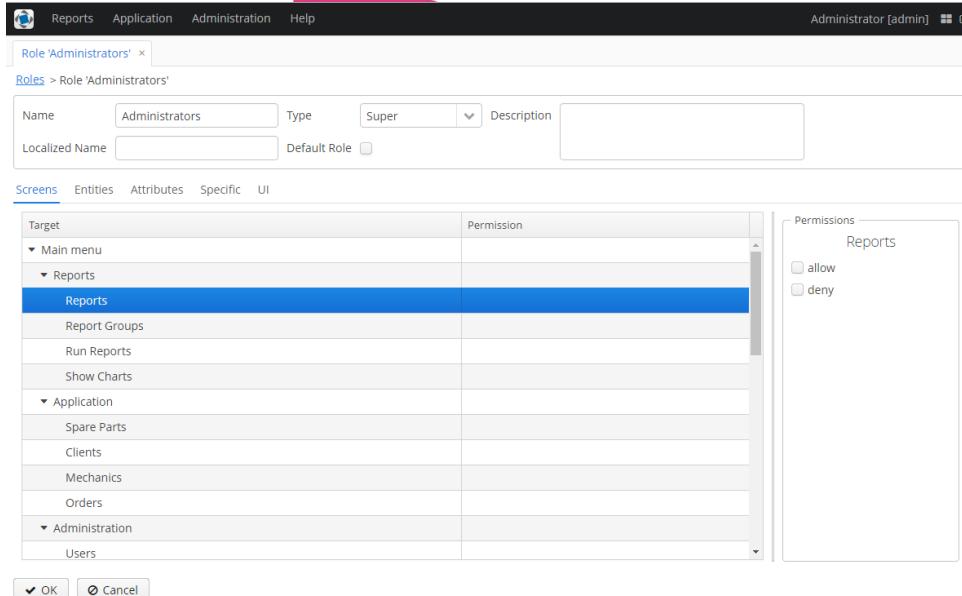


Security subsystem

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.



The screenshot shows the CUBA platform's administration interface for managing roles. The top navigation bar includes 'Reports', 'Application', 'Administration', and 'Help'. The user is logged in as 'Administrator [admin]'. The main content area is titled 'Role 'Administrators'' and shows the details for this role. The 'Name' is 'Administrators', 'Type' is 'Super', and there is a 'Description' field. Below this, there are tabs for 'Screens', 'Entities', 'Attributes', 'Specific', and 'UI'. The 'Screens' tab is selected, displaying a list of targets and their associated permissions. The 'Reports' target is currently selected, showing its specific permissions. On the right side, there is a sidebar titled 'Permissions' with sections for 'Reports', 'allow', and 'deny'. At the bottom, there are 'OK' and 'Cancel' buttons.

Target	Permission
▼ Main menu	
▼ Reports	
Reports	
Report Groups	
Run Reports	
Show Charts	
▼ Application	
Spare Parts	
Clients	
Mechanics	
Orders	
▼ Administration	
Users	

Permissions

Reports

allow

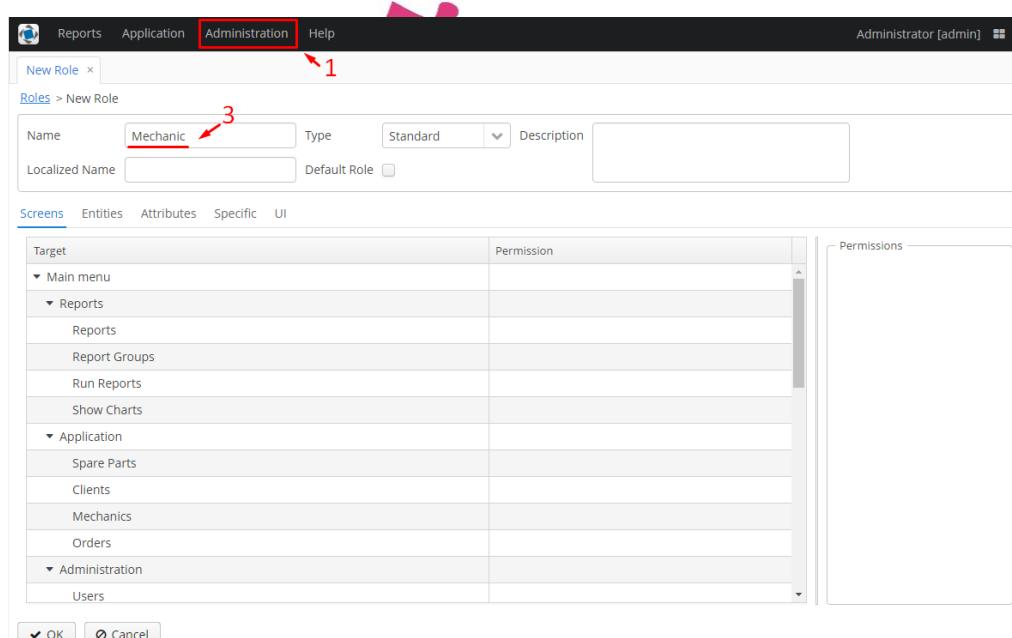
deny

OK Cancel

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.

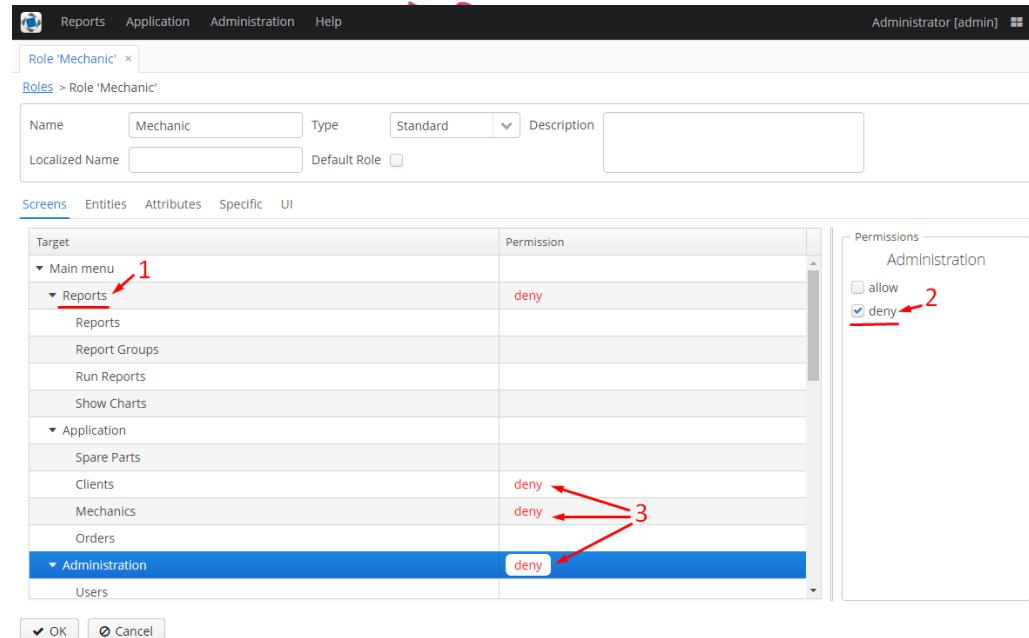
1. Open **Administration — Roles** from the menu
2. Click **Create**
3. Set **Name: Mechanic**



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**



The screenshot shows the 'Role' configuration screen for the 'Mechanic' role. The 'Screens' tab is selected. The table lists various targets and their permissions:

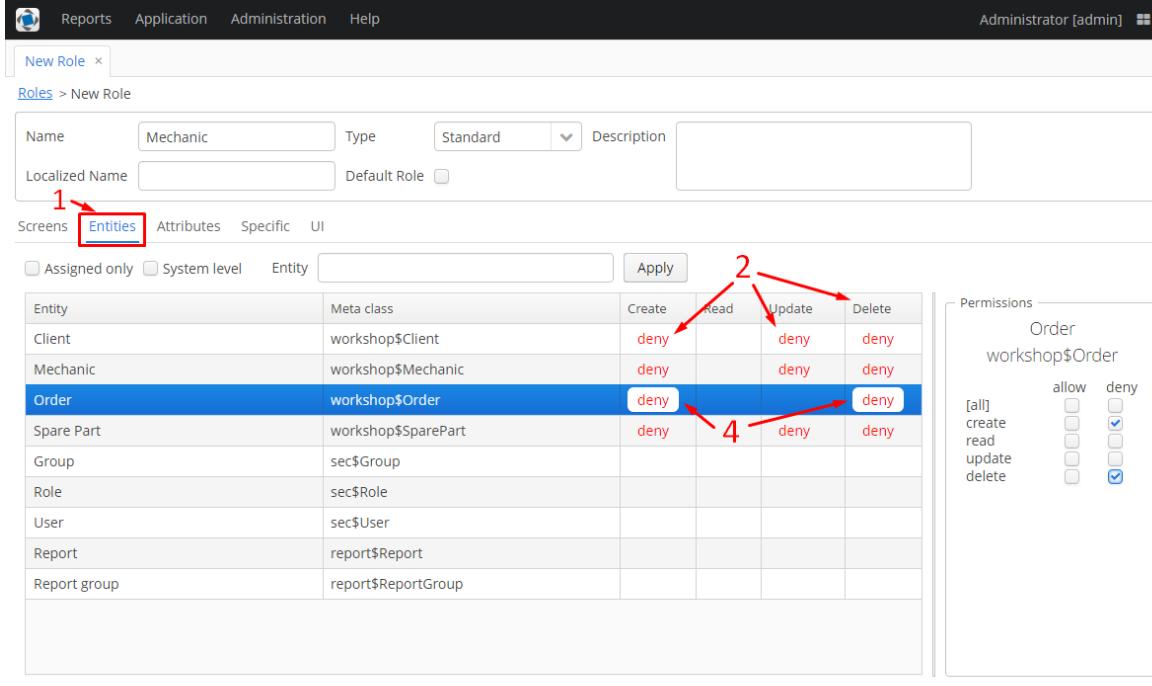
Target	Permission
Main menu	
Reports	deny
Report Groups	
Run Reports	
Show Charts	
Application	
Spare Parts	
Clients	deny
Mechanics	deny
Orders	
Administration	deny
Users	

Annotations with red arrows and numbers indicate specific actions:

- Annotation 1 points to the 'Reports' row in the tree view under 'Main menu'.
- Annotation 2 points to the checked 'deny' checkbox in the 'Permissions' column for the 'Reports' target.
- Annotation 3 points to the 'deny' checkboxes for the 'Clients', 'Mechanics', and 'Administration' targets.

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**



The screenshot shows the 'New Role' configuration screen in the CUBA platform. The 'Entities' tab is selected, indicated by a red box and the number 1. The 'Order' entity is selected in the list, indicated by a blue highlight and the number 4. Red arrows point from the numbers 2 and 3 to the 'Create', 'Update', and 'Delete' columns for the Client, Mechanic, and Order entities respectively, all of which have the value 'deny'. A legend on the right side of the table shows 'allow' (unchecked) and 'deny' (checked) for each operation.

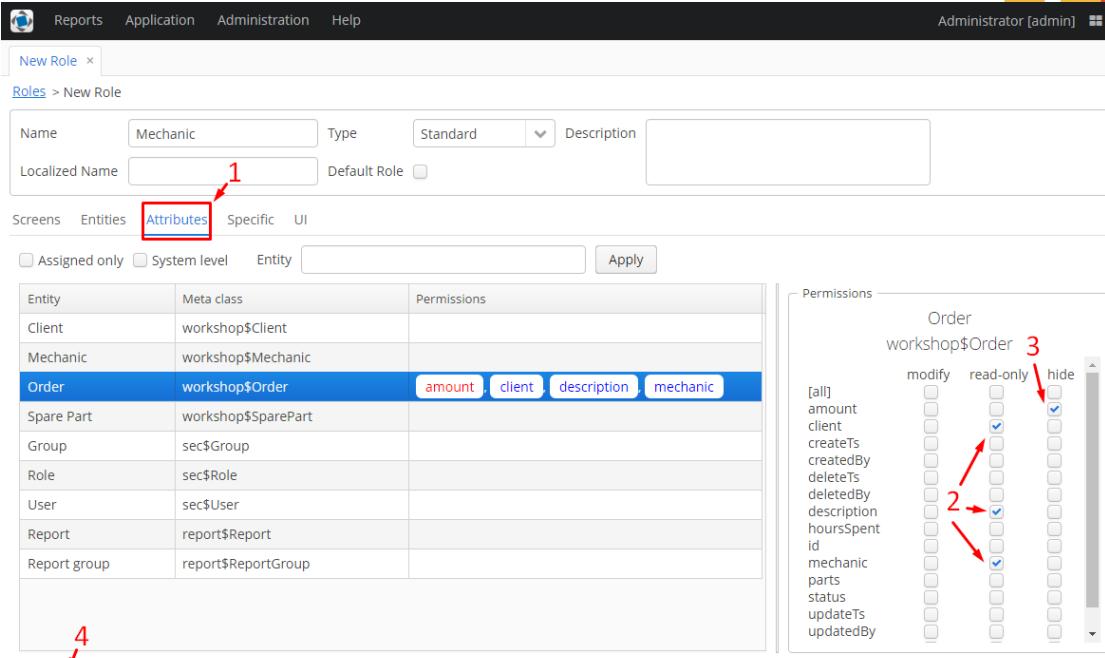
Entity	Meta class	Create	Read	Update	Delete
Client	workshop\$Client	deny	deny	deny	deny
Mechanic	workshop\$Mechanic	deny	deny	deny	deny
Order	workshop\$Order	deny	deny	deny	deny
Spare Part	workshop\$SparePart	deny	deny	deny	deny
Group	sec\$Group				
Role	sec\$Role				
User	sec\$User				
Report	report\$Report				
Report group	report\$ReportGroup				

Permissions for Order workshop\$Order

	allow	deny
[all]	<input type="checkbox"/>	<input checked="" type="checkbox"/>
create	<input type="checkbox"/>	<input checked="" type="checkbox"/>
read	<input type="checkbox"/>	<input checked="" type="checkbox"/>
update	<input type="checkbox"/>	<input checked="" type="checkbox"/>
delete	<input type="checkbox"/>	<input checked="" type="checkbox"/>

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



The screenshot shows the 'New Role' configuration screen. The 'Attributes' tab is selected, indicated by a red box and the number 1. In the 'Permissions' section for the 'Order' entity, the 'amount' attribute has the 'read-only' permission checked (indicated by a red box and the number 2). The 'client', 'mechanic', and 'description' attributes also have their 'read-only' checkboxes checked (indicated by red boxes and the number 3). The 'hide' checkbox for the 'amount' attribute is checked (indicated by a red box and the number 4). At the bottom, the 'OK' button is highlighted with a red box.

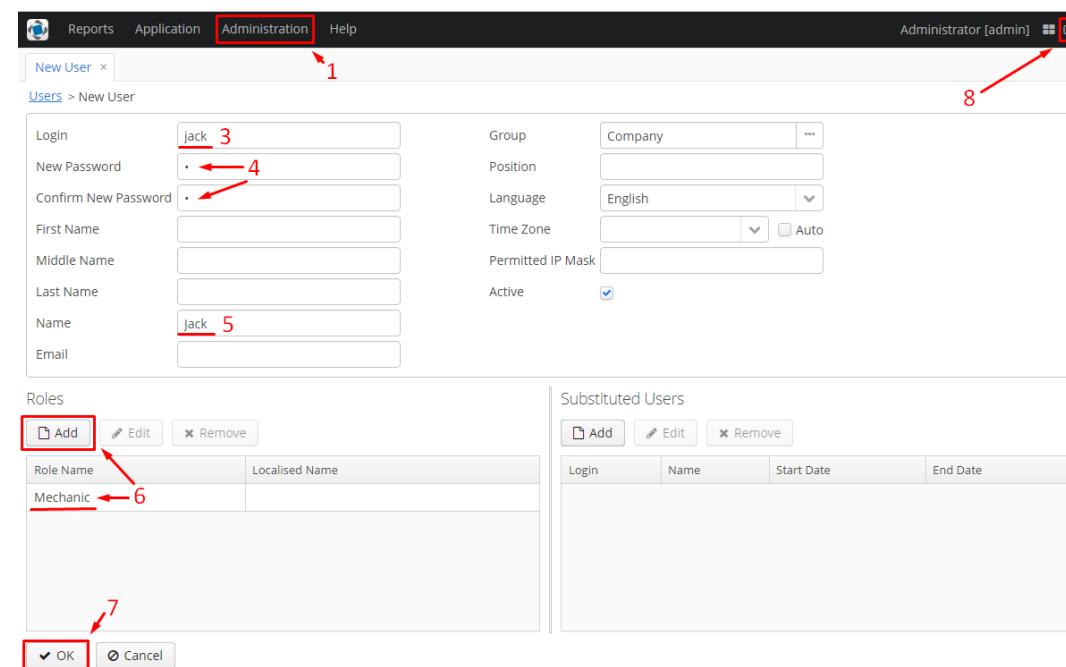
Entity	Meta class	Permissions
Client	workshop\$Client	
Mechanic	workshop\$Mechanic	
Order	workshop\$Order	amount, client, description, mechanic
Spare Part	workshop\$SparePart	
Group	sec\$Group	
Role	sec\$Role	
User	sec\$User	
Report	report\$Report	
Report group	report\$ReportGroup	

Permissions

Order	workshop\$Order
modify	<input type="checkbox"/>
read-only	<input checked="" type="checkbox"/>
hide	<input checked="" type="checkbox"/>
[all]	
amount	<input type="checkbox"/>
client	<input checked="" type="checkbox"/>
createTs	<input type="checkbox"/>
createdBy	<input type="checkbox"/>
deleteTs	<input type="checkbox"/>
deletedBy	<input type="checkbox"/>
description	<input type="checkbox"/>
hoursSpent	<input type="checkbox"/>
id	<input type="checkbox"/>
mechanic	<input type="checkbox"/>
parts	<input type="checkbox"/>
status	<input type="checkbox"/>
updateTs	<input type="checkbox"/>
updatedBy	<input checked="" type="checkbox"/>

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



New User x 1

Administrator [admin] 2

Users > New User

Login	jack 3	Group	Company
New Password	• 4	Position	
Confirm New Password	• 4	Language	English
First Name		Time Zone	
Middle Name		Permitted IP Mask	
Last Name		Active	<input checked="" type="checkbox"/>
Name	Jack 5		
Email			

Roles

Add 6	Edit	Remove
Role Name	Localised Name	
Mechanic		

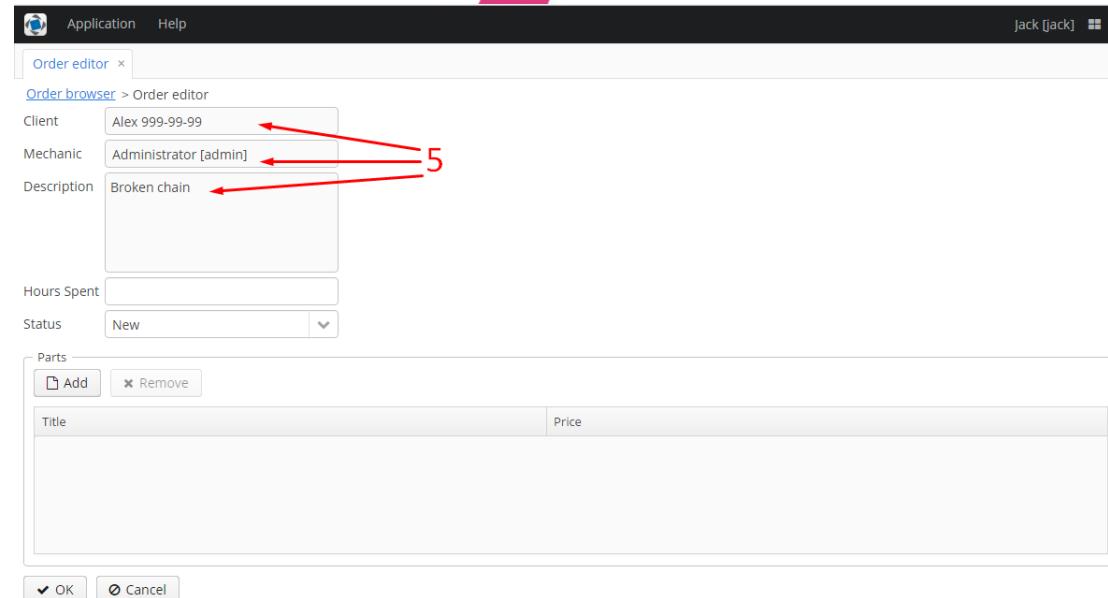
Substituted Users

Add	Edit	Remove	
Login	Name	Start Date	End Date

OK 7 Cancel

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



The screenshot shows the 'Order editor' window with several fields and annotations:

- Client:** Alex 999-99-99 (read-only)
- Mechanic:** Administrator [admin] (read-only)
- Description:** Broken chain (read-only)
- Hours Spent:** (Hidden field)
- Status:** New (dropdown menu)
- Parts:** A section with 'Add' and 'Remove' buttons, and a table with columns 'Title' and 'Price'. The table is currently empty.
- Buttons:** OK and Cancel

Red arrows point from the numbered list items to the corresponding fields: item 5 points to the Client, Mechanic, and Description fields.



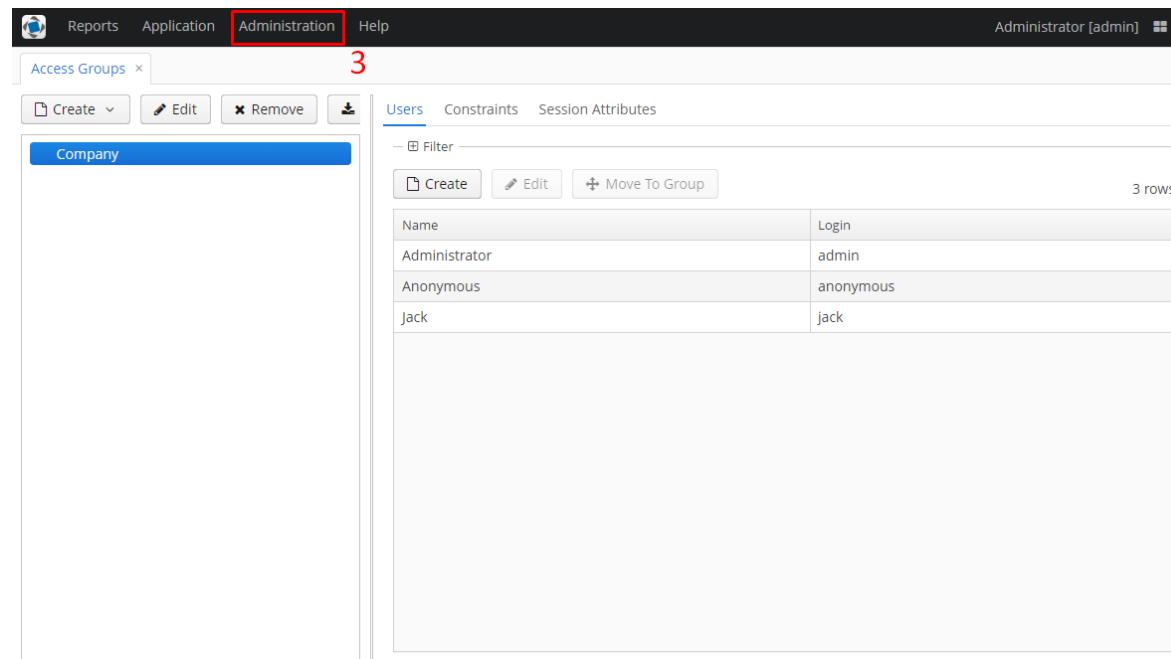
Row level security

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).

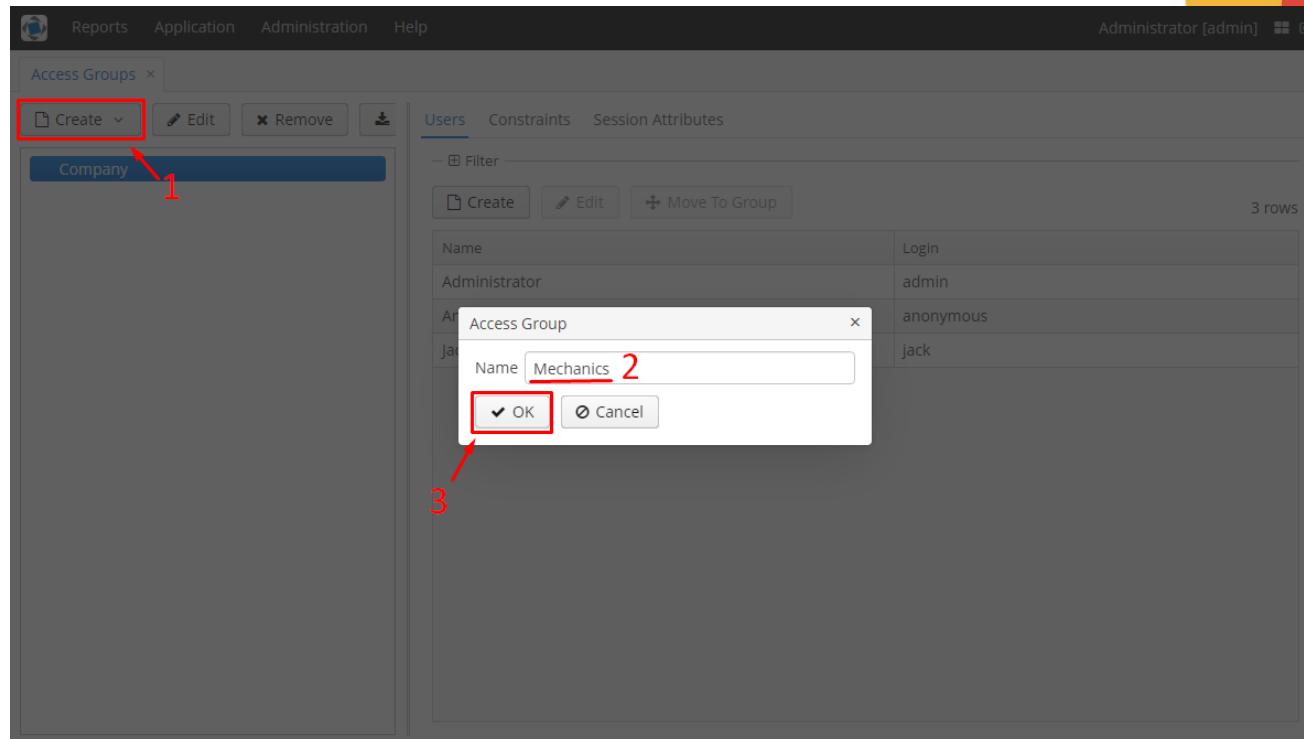


Name	Login
Administrator	admin
Anonymous	anonymous
jack	jack



Create an access group

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

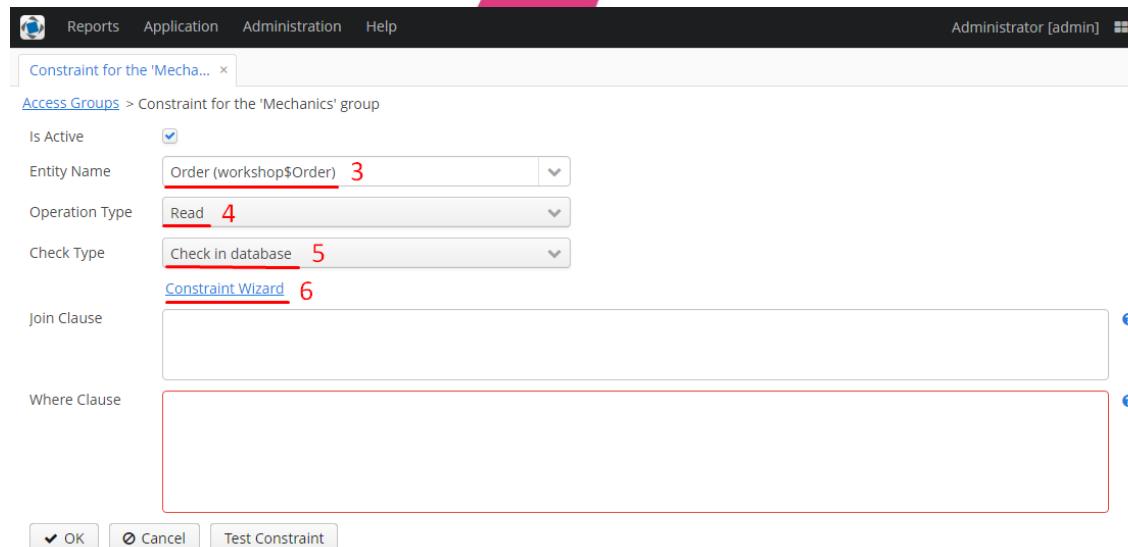


Add constraint for the access group

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



Constraint for the 'Mechanics' group

Access Groups > Constraint for the 'Mechanics' group

Is Active

Entity Name Order (workshop\$Order) 3

Operation Type Read 4

Check Type Check in database 5

Constraint Wizard 6

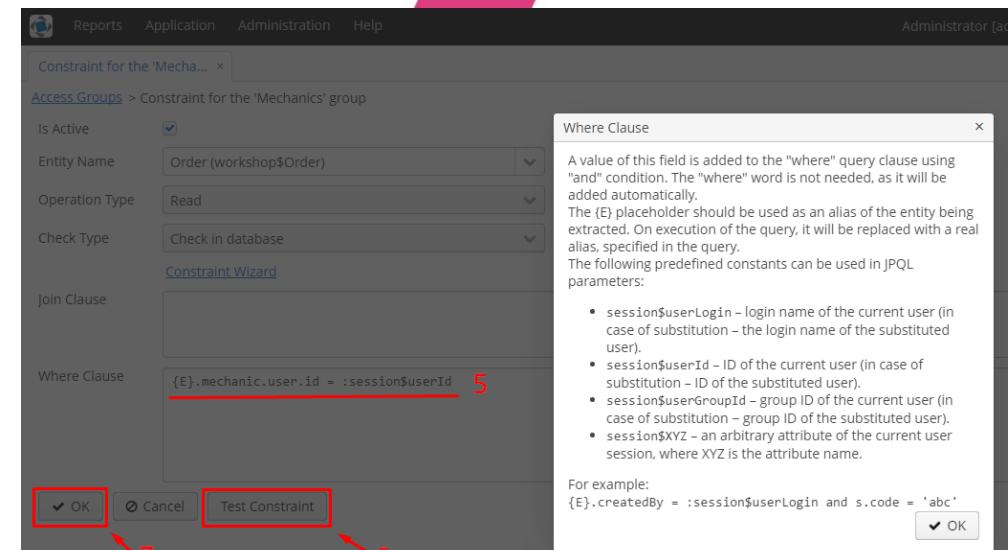
Join Clause

Where Clause

OK Cancel Test Constraint

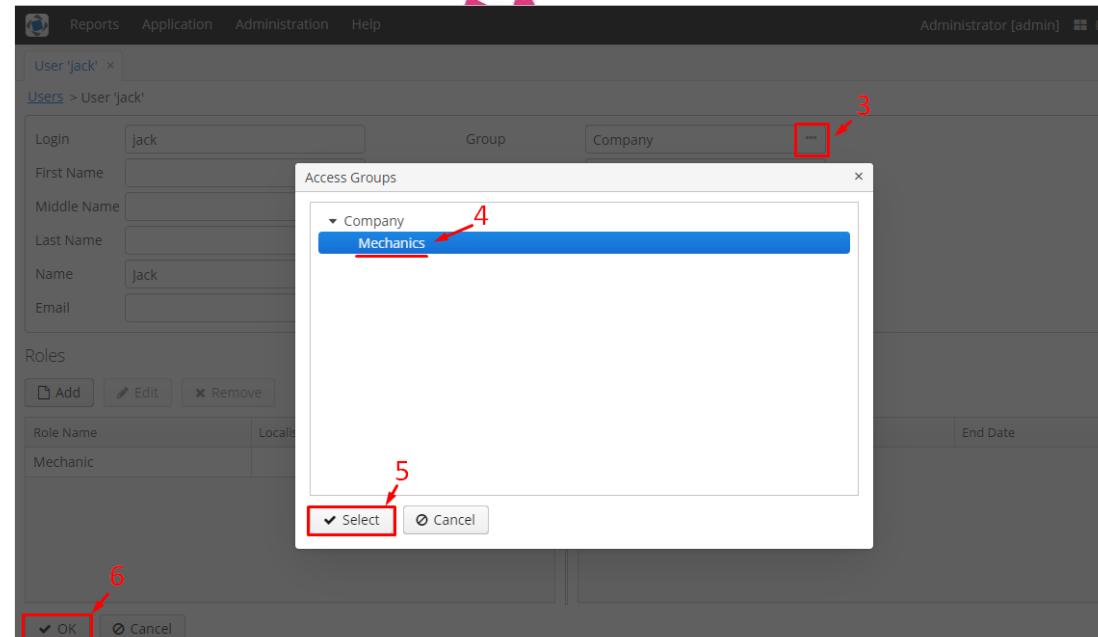
Add constraint for the access group

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**



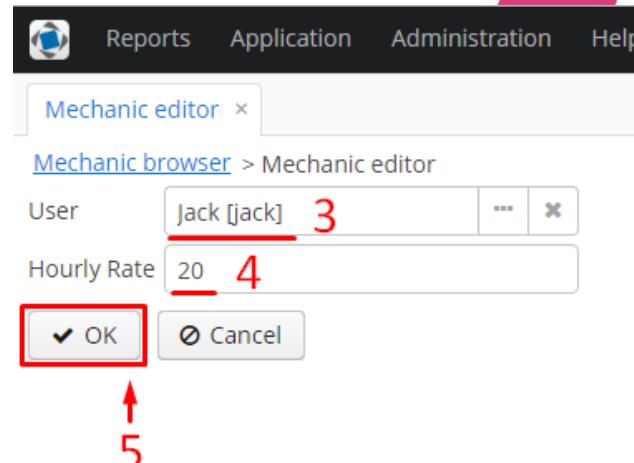
Assign group to the user

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

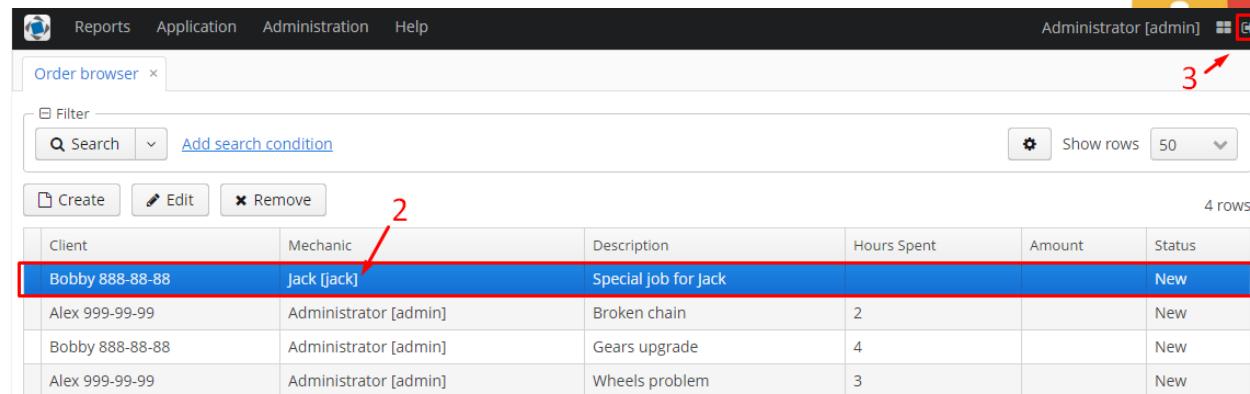
1. Open **Application — Mechanics** from the menu
2. Click **Create**
3. Select user: **jack**
4. Set **Hourly Rate**
5. Click **OK** to save the mechanic



5

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



Administrator [admin] 

Order browser  Filter  Search  Add search condition  Show rows 50  4 rows

 Create  Edit  Remove

Client	Mechanic	Description	Hours Spent	Amount	Status
Bobby 888-88-88	Jack [jack]	Special job for Jack			New
Alex 999-99-99	Administrator [admin]	Broken chain	2		New
Bobby 888-88-88	Administrator [admin]	Gears upgrade	4		New
Alex 999-99-99	Administrator [admin]	Wheels problem	3		New

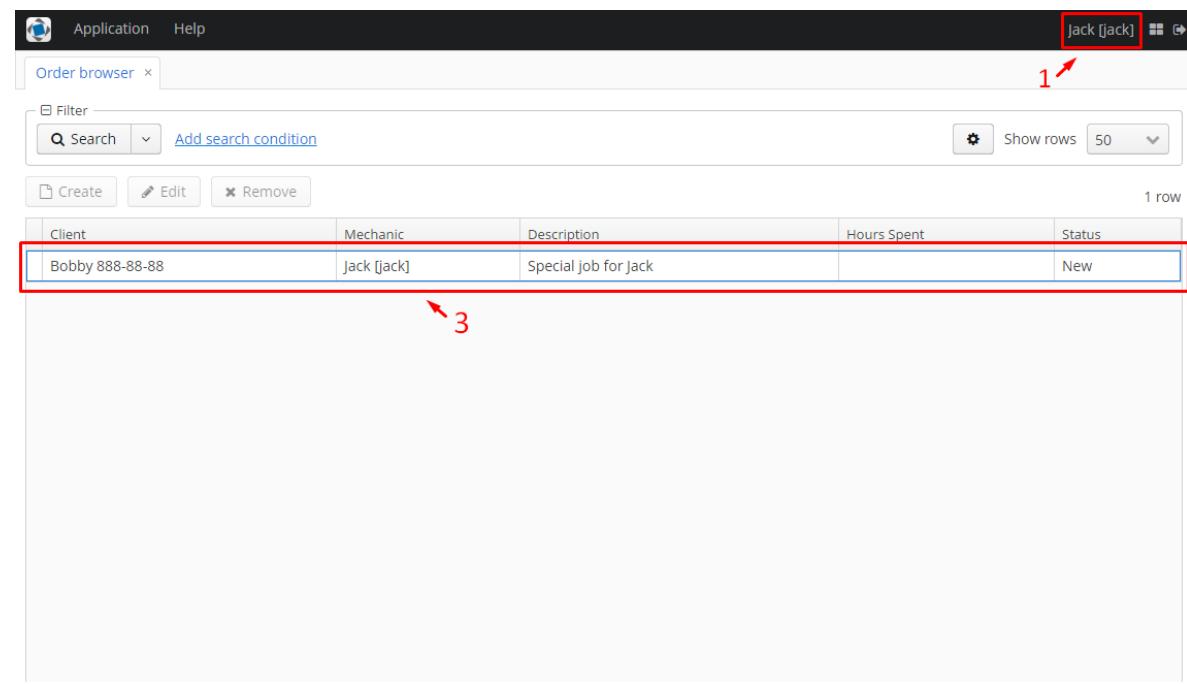


Row level security in action

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.



The screenshot shows a screenshot of a desktop application window titled "Order browser". The window has a dark header bar with "Application" and "Help" menus. In the top right corner, there is a user session indicator showing "Jack [jack]" with a red box and a red arrow pointing to it. Below the header is a toolbar with "Search", "Add search condition", "Create", "Edit", and "Remove" buttons. To the right of the toolbar are "Show rows" and "50" buttons. The main area is a table with the following data:

Client	Mechanic	Description	Hours Spent	Status
Bobby 888-88-88	Jack [jack]	Special job for Jack		New

A large red box highlights the entire row of data. A red arrow points to the number "3" located below the table, likely indicating the row count or a specific record identifier.

Services



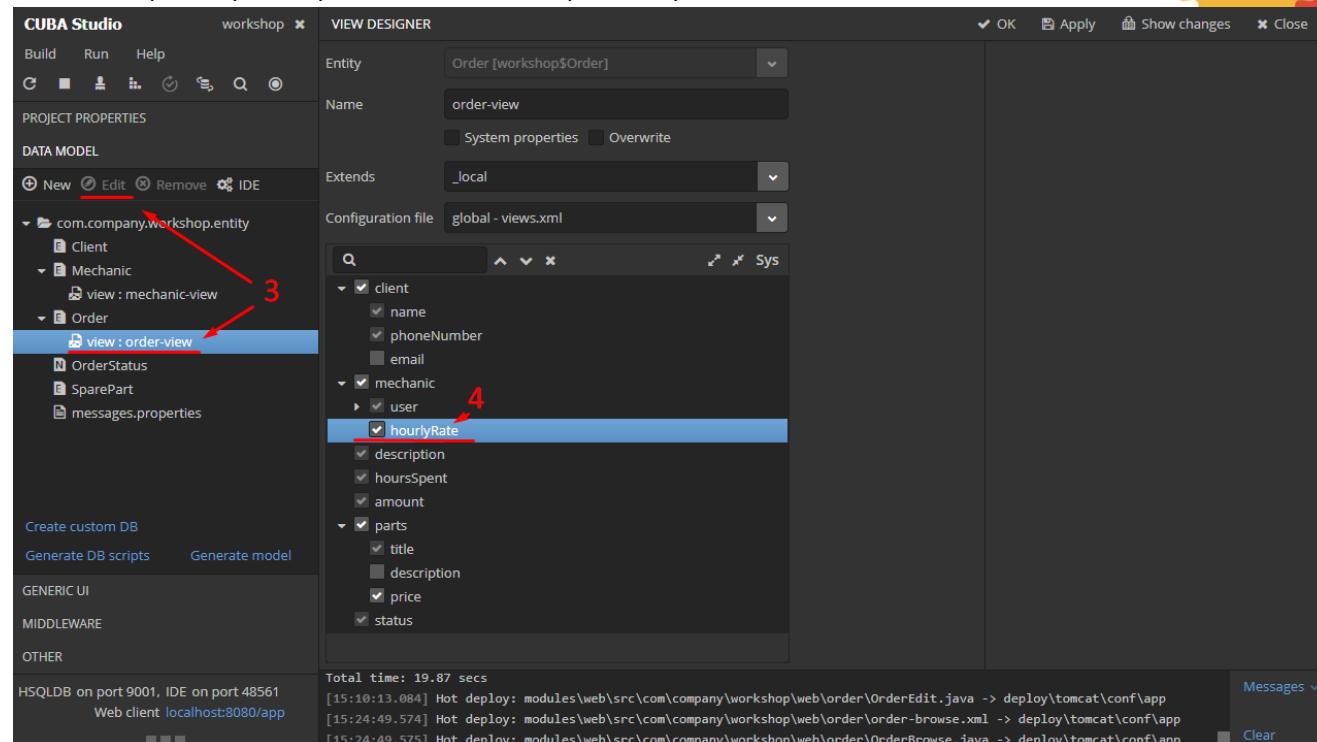


Services

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



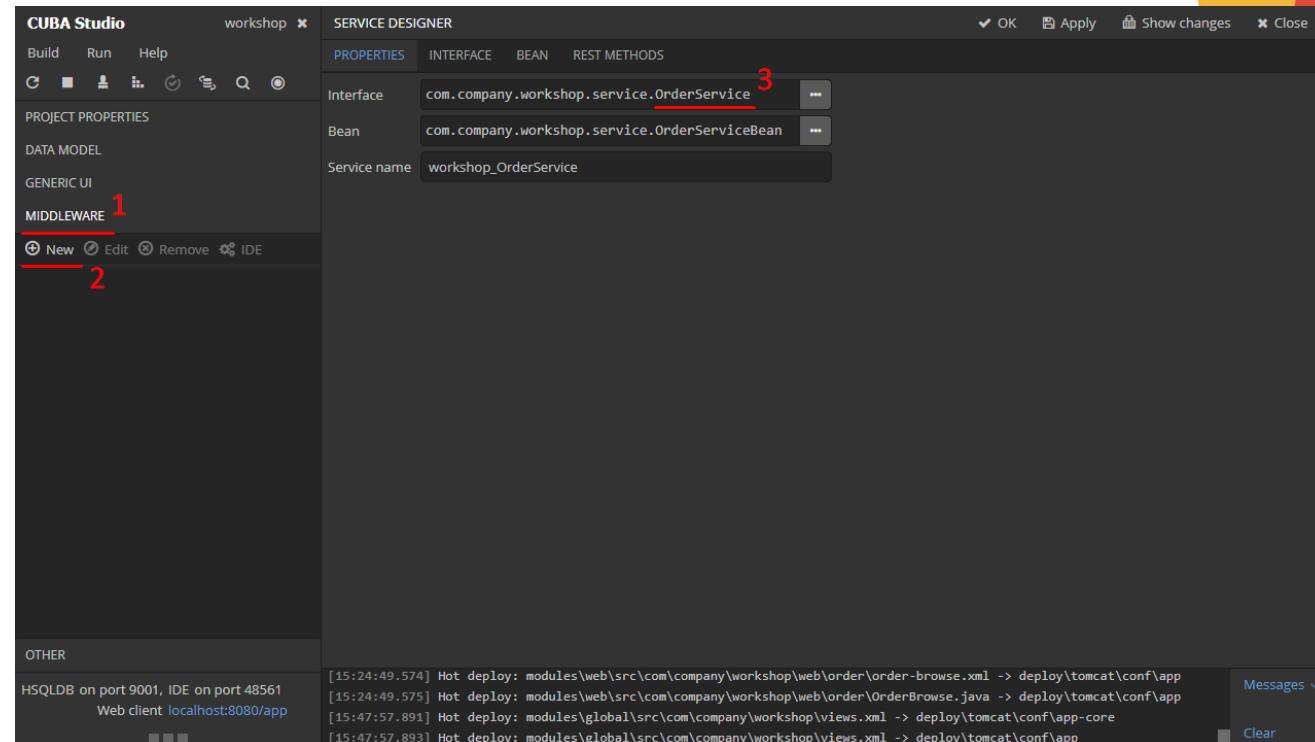
The screenshot shows the CUBA Studio View Designer interface. The title bar says "CUBA Studio workshop x". The left sidebar has "PROJECT PROPERTIES" and "DATA MODEL" sections. Under "DATA MODEL", there are buttons for "New", "Edit" (which is highlighted with a red arrow), "Remove", and "IDE". Below these are nodes for "com.company.workshop.entity": "Client", "Mechanic" (with a sub-node "view : mechanic-view" highlighted with a red arrow), and "Order" (with a sub-node "view : order-view" highlighted with a red arrow). Other nodes include "OrderStatus", "SparePart", and "messages.properties". At the bottom of the sidebar are buttons for "Create custom DB", "Generate DB scripts", and "Generate model". The main area is titled "VIEW DESIGNER" with tabs for "Entity", "Name", "Extends", and "Configuration file". The "Entity" tab shows "Order [workshop\$Order]". The "Name" tab shows "order-view" with checkboxes for "System properties" and "Overwrite". The "Extends" tab shows "_local". The "Configuration file" tab shows "global - views.xml". Below these tabs is a search bar and a list of attributes. The "mechanic" section contains "client" (with "name", "phoneNumber", "email" checked) and "user" (with "hourlyRate" checked, highlighted with a red arrow). The "parts" section contains "parts" (with "title", "description", "price" checked) and "status". At the bottom, a message says "Total time: 19.87 secs" followed by three log entries. The status bar at the bottom right says "Messages" and "Clear".



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.

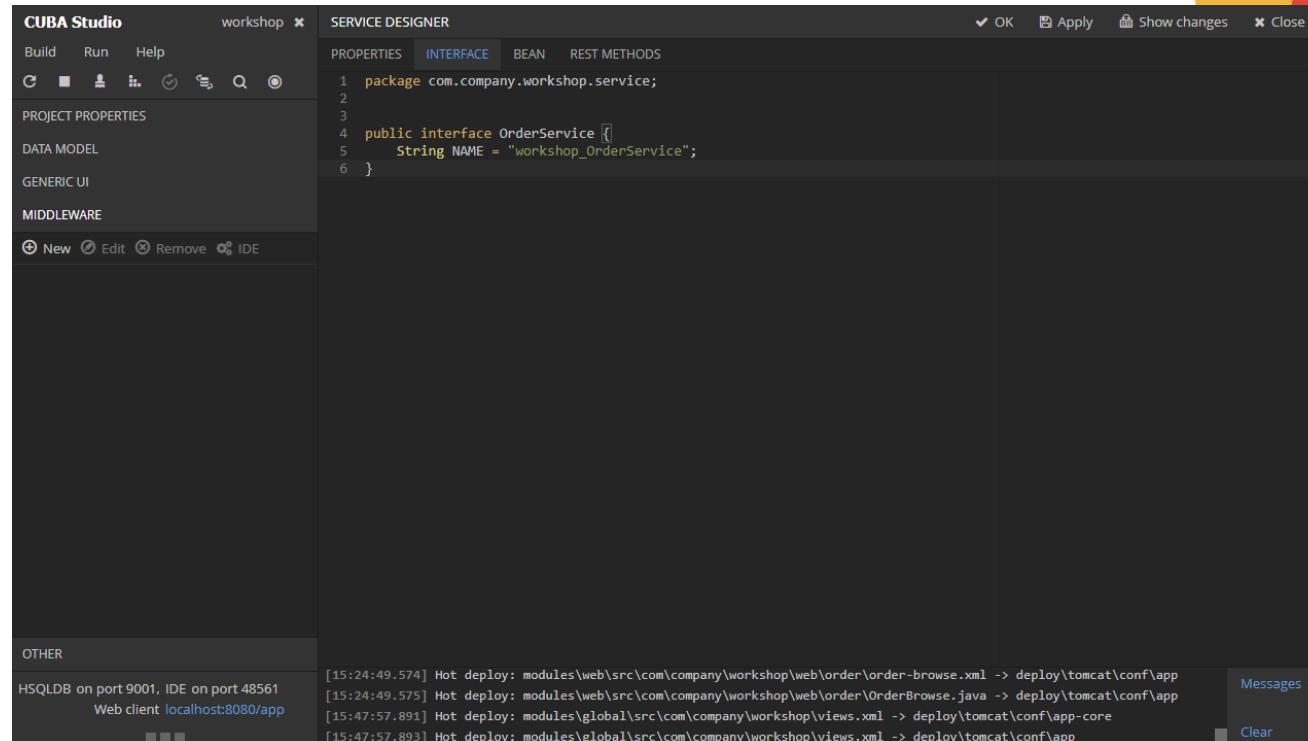
1. Go to the MIDDLEWARE section in Studio
2. Click **New -> Service**
3. Change the last part of Interface name to **OrderService**



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).



The screenshot shows the CUBA Studio interface with the 'workshop' project selected. The left sidebar includes 'PROJECT PROPERTIES', 'DATA MODEL', 'GENERIC UI', and 'MIDDLEWARE' sections, along with 'New', 'Edit', 'Remove', and 'IDE' buttons. The main area is titled 'SERVICE DESIGNER' with tabs for 'PROPERTIES', 'INTERFACE', 'BEAN', and 'REST METHODS'. The 'INTERFACE' tab is active, displaying the following Java code:

```
1 package com.company.workshop.service;
2
3
4 public interface OrderService {
5     String NAME = "workshop_OrderService";
6 }
```

Below the code editor, the 'OTHER' section shows logs for a hot deployment:

```
[15:24:49.574] Hot deploy: modules\web\src\com\company\workshop\web\order\order-browse.xml -> deploy\tomcat\conf\app
[15:24:49.575] Hot deploy: modules\web\src\com\company\workshop\web\order\OrderBrowse.java -> deploy\tomcat\conf\app
[15:47:57.891] Hot deploy: modules\global\src\com\company\workshop\views.xml -> deploy\tomcat\conf\app-core
[15:47:57.893] Hot deploy: modules\global\src\com\company\workshop\views.xml -> deploy\tomcat\conf\app
```

On the right side, there are 'OK', 'Apply', 'Show changes', and 'Close' buttons, and a 'Messages' section with a 'Clear' button.



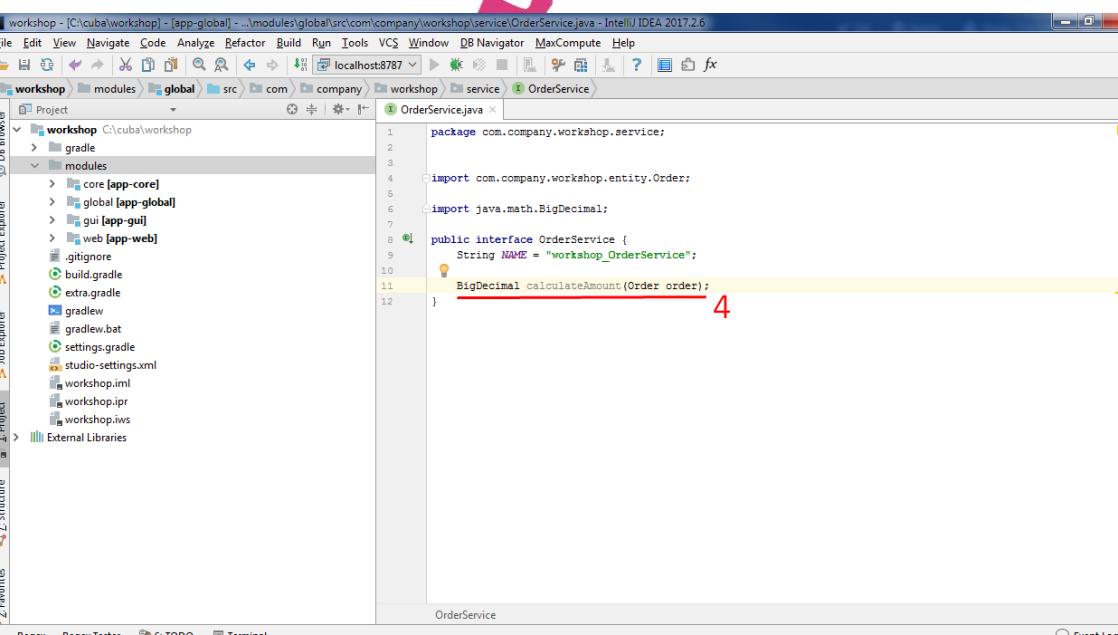
Add method to a service

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);
}
```



The screenshot shows the IntelliJ IDEA interface with the OrderService.java file open in the editor. The code is identical to the one above. A red box highlights the 'calculateAmount' method, and a red number '4' is placed to its right, likely indicating a step or note related to the method's implementation.



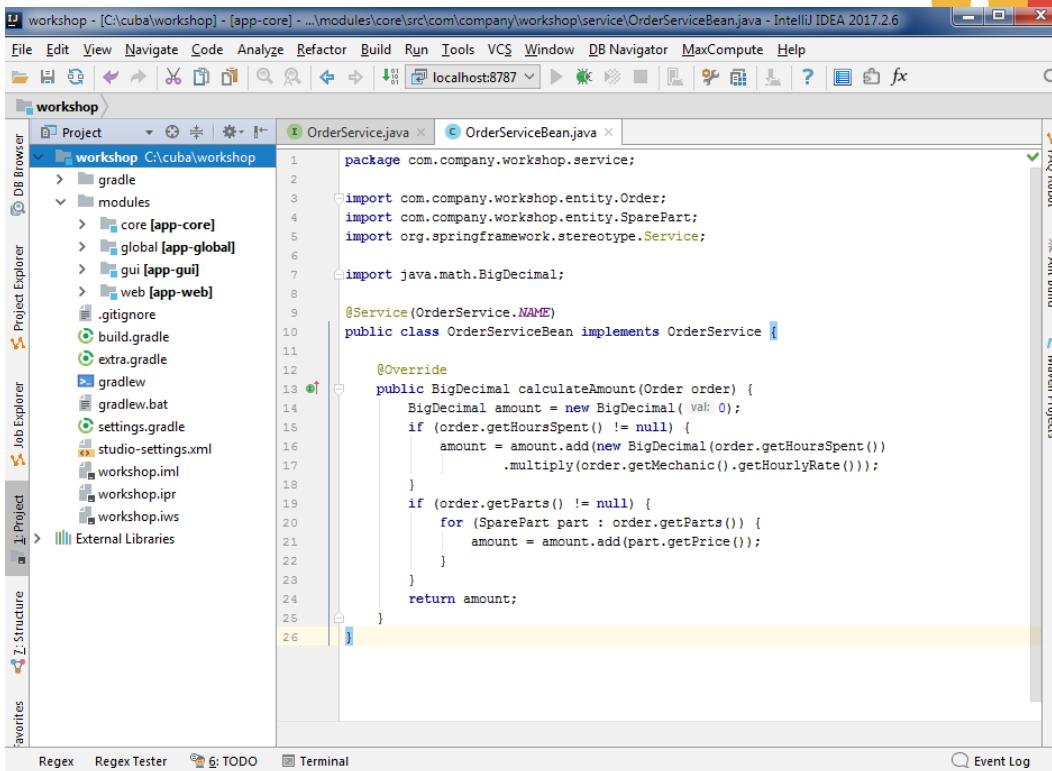
Service method implementation

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;
    }
}
```



The screenshot shows the IntelliJ IDEA interface with the OrderServiceBean.java file open in the editor. The code is identical to the one above, implementing the calculateAmount method. The IntelliJ IDEA interface includes toolbars, a menu bar, and various project navigation panels like Project, DB Browser, and External Libraries.

```
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;

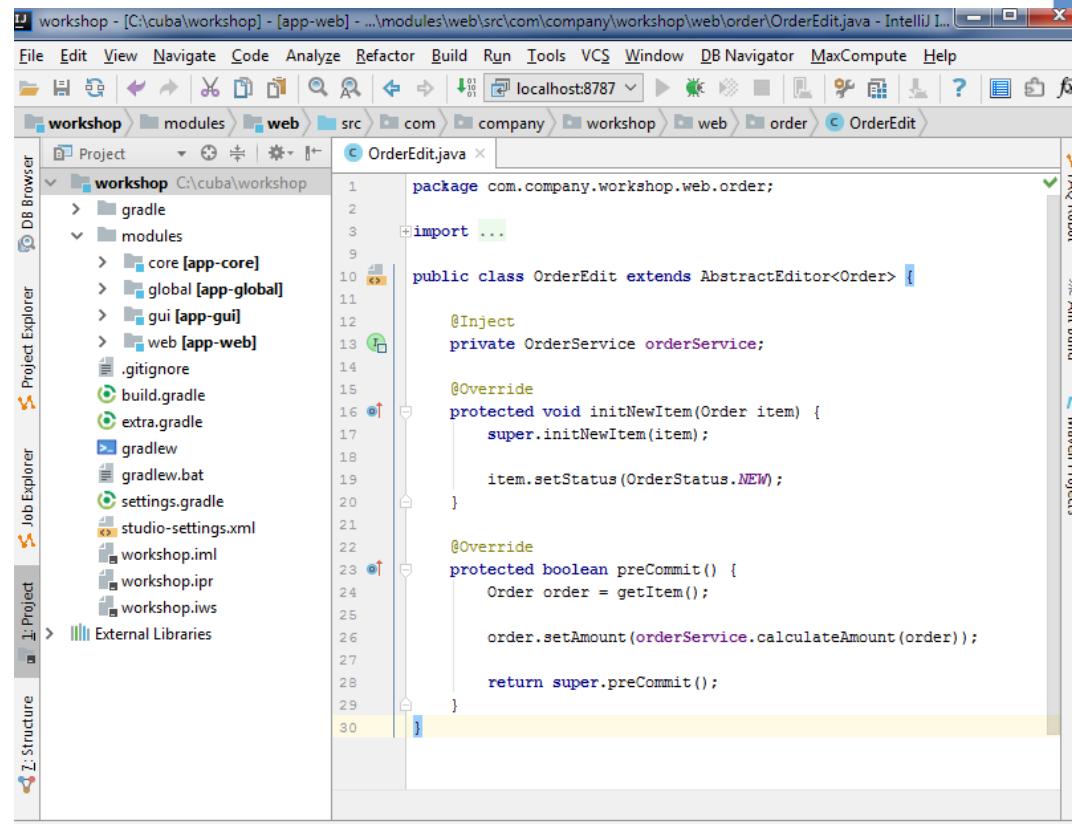
@Service(OrderService.NAME)
public class OrderServiceBean implements OrderService {
    @Override
    public BigDecimal calculateAmount(Order order) {
        BigDecimal amount = new BigDecimal(0);
        if (order.getHoursSpent() != null) {
            amount = amount.add(new BigDecimal(order.getHoursSpent())
                    .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**

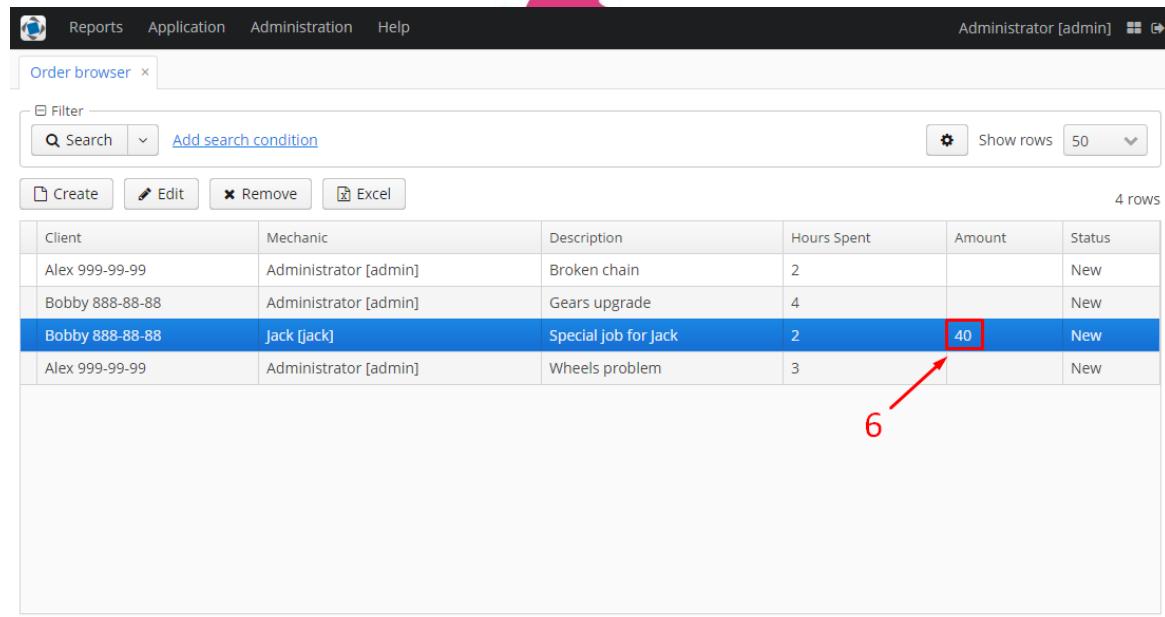
```
public class OrderEdit extends AbstractEditor<Order> {  
  
    @Inject  
    private OrderService orderService;  
    // ...  
  
    @Override  
  
    protected boolean preCommit() {  
        Order order = getItem();  
  
        order.setAmount(orderService.calculateAmount(order));  
  
   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



```
workshop - [C:\cuba\workshop] - [app-web] - ...\\modules\\web\\src\\com\\company\\workshop\\web\\order\\OrderEdit.java - IntelliJ IDEA  
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window DB Navigator MaxCompute Help  
workshop modules web src com company workshop web order OrderEdit  
Project DB Browser Job Explorer Project External Libraries Structure  
workshop C:\cuba\workshop  
modules  
core [app-core]  
global [app-global]  
gui [app-gui]  
web [app-web]  
.gitignore  
build.gradle  
extra.gradle  
gradlew  
gradlew.bat  
settings.gradle  
studio-settings.xml  
workshop.ilm  
workshop.ipr  
workshop.iws  
OrderEdit.java  
package com.company.workshop.web.order;  
import ...  
public class OrderEdit extends AbstractEditor<Order> {  
    @Inject  
    private OrderService orderService;  
  
    @Override  
    protected void initNewItem(Order item) {  
        super.initNewItem(item);  
  
        item.setStatus(OrderStatus.NEW);  
    }  
  
    @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



Client	Mechanic	Description	Hours Spent	Amount	Status
Alex 999-99-99	Administrator [admin]	Broken chain	2		New
Bobby 888-88-88	Administrator [admin]	Gears upgrade	4		New
Bobby 888-88-88	Jack [jack]	Special job for Jack	2	40	New
Alex 999-99-99	Administrator [admin]	Wheels problem	3		New

Charts

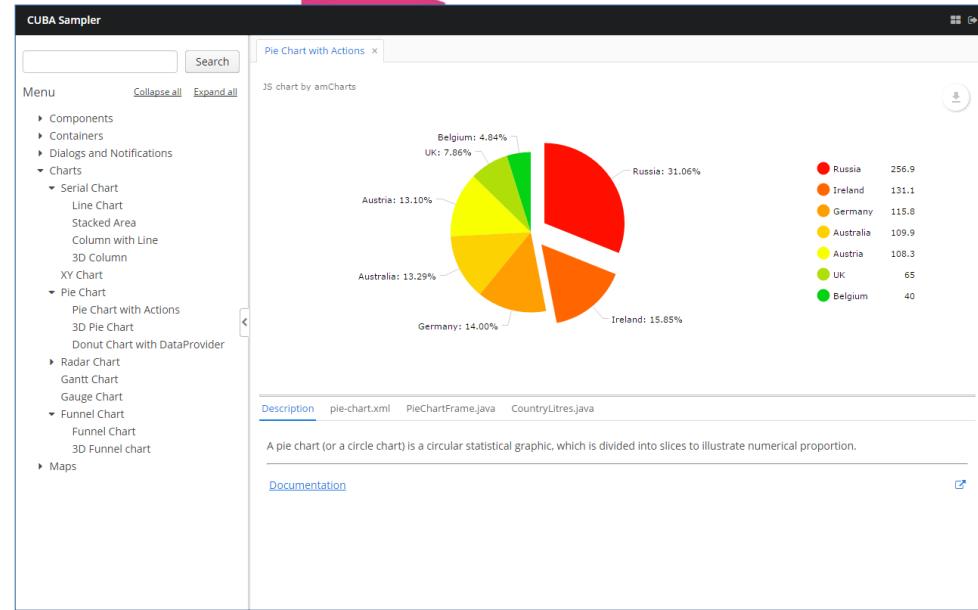




Charts

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**

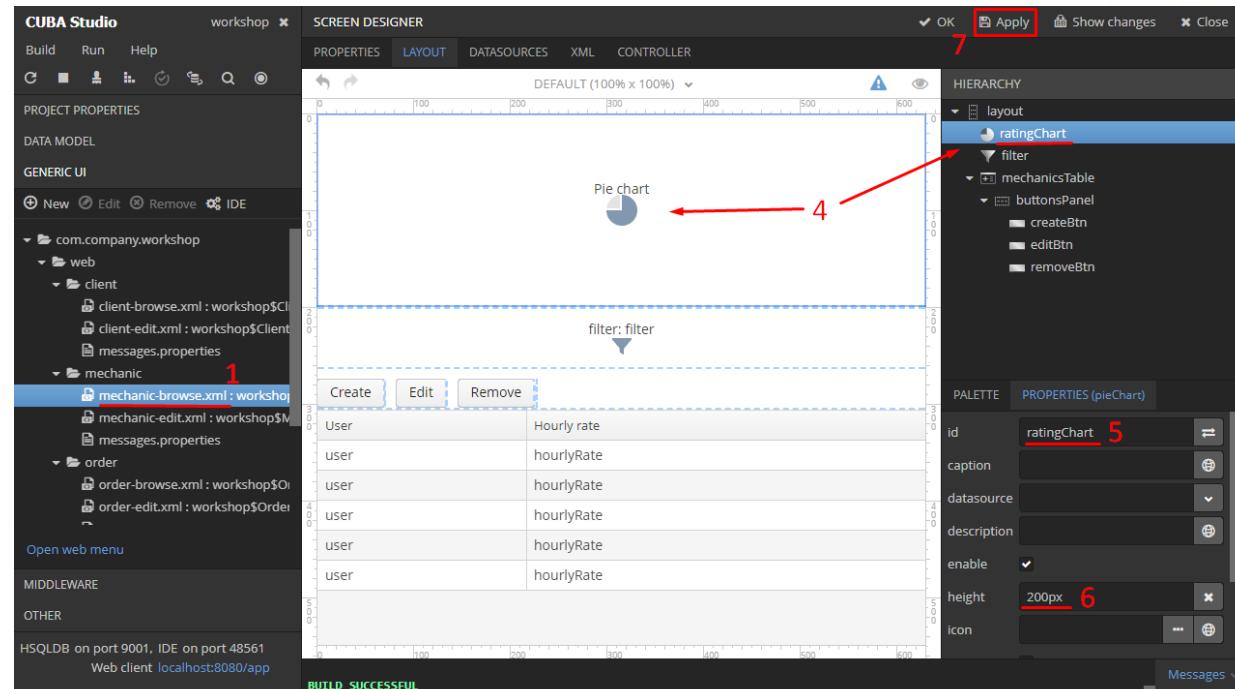
3. Studio will filter the component list and show us components to display charts

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

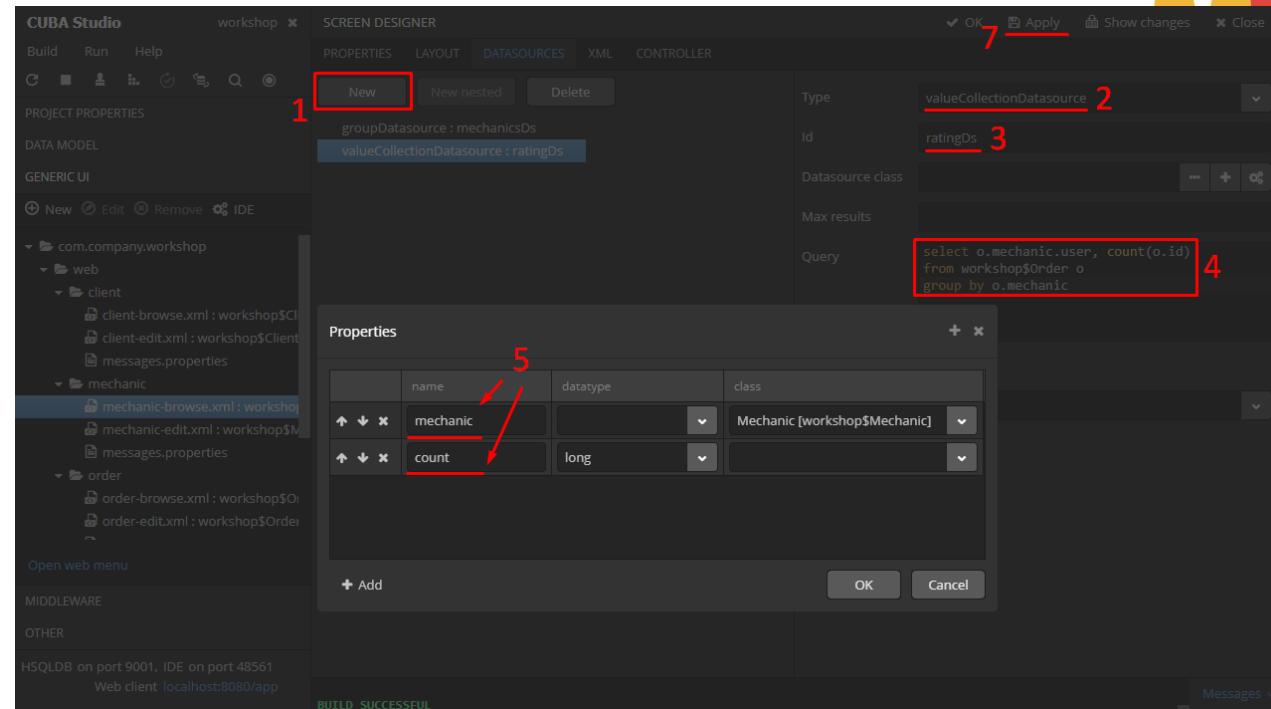
4. Enter the query to get the number of orders for each mechanic:

```
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**



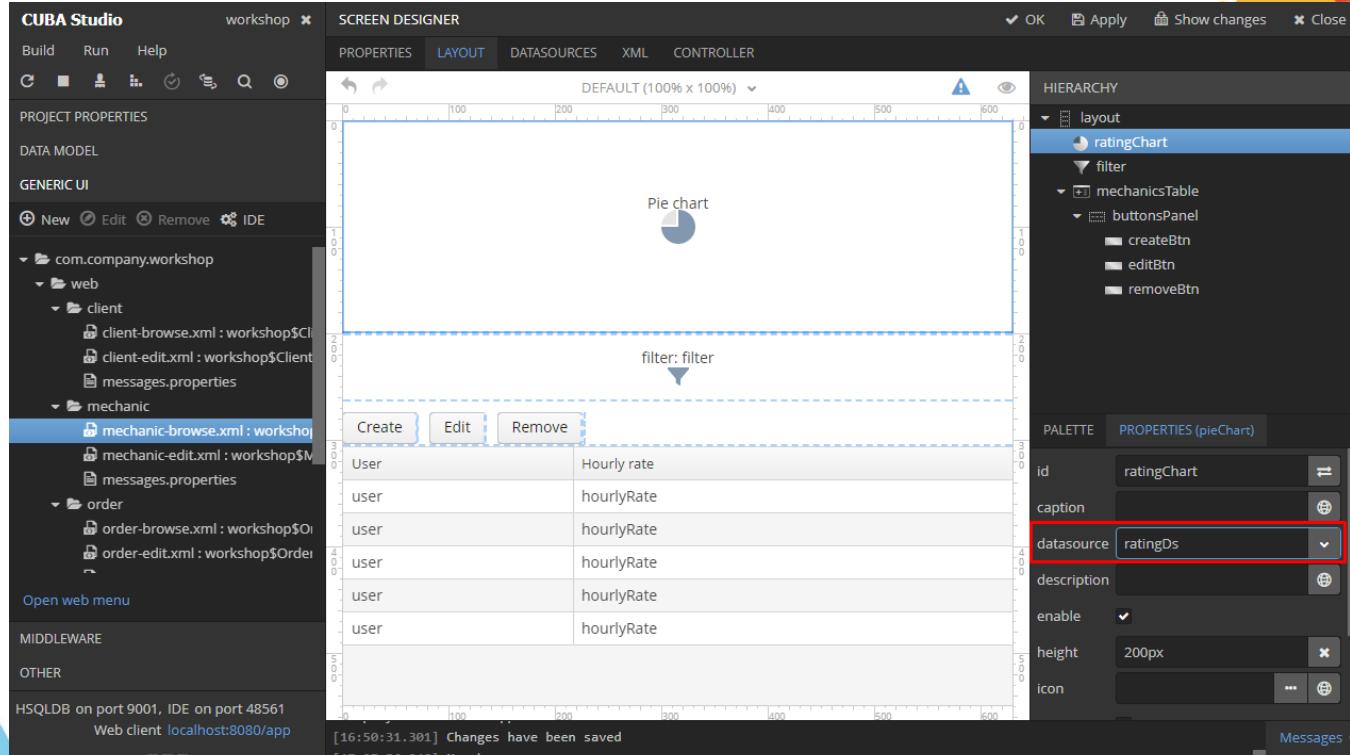
The screenshot shows the CUBA Studio interface with the 'DATASOURCES' tab selected. A 'New' button is highlighted with a red box (1). The 'Type' dropdown shows 'valueCollectionDatasource' (2), which is also highlighted with a red box. The 'Id' field contains 'ratingDs' (3). The 'Query' field contains the SQL query (4):

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

A 'Properties' dialog is open, showing two properties: 'mechanic' (class: Mechanic [workshop\$Mechanic], datatype: object) and 'count' (datatype: long) (5). Red arrows point from the 'name' column to the respective columns in the properties table. The bottom right of the dialog shows 'OK' and 'Cancel' buttons, with 'OK' highlighted with a red box (7).

Load data for chart

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



The screenshot shows the CUBA Studio interface with the "workshop" project selected. The "SCREEN DESIGNER" tab is active, displaying the "LAYOUT" tab. In the center, there is a "Pie chart" component. The "HIERARCHY" panel on the right shows the structure of the screen, with "ratingChart" as the root node under "layout". The "PALETTE" and "PROPERTIES (pieChart)" panels are visible on the right, showing the component's properties. The "datasource" field in the properties panel is highlighted with a red border, indicating it is being edited. The value "ratingDs" is entered in this field.

Properties of pieChart:

- id**: ratingChart
- caption**: (empty)
- datasource**: ratingDs (highlighted)
- description**: (empty)
- enable**: checked
- height**: 200px
- icon**: (empty)

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

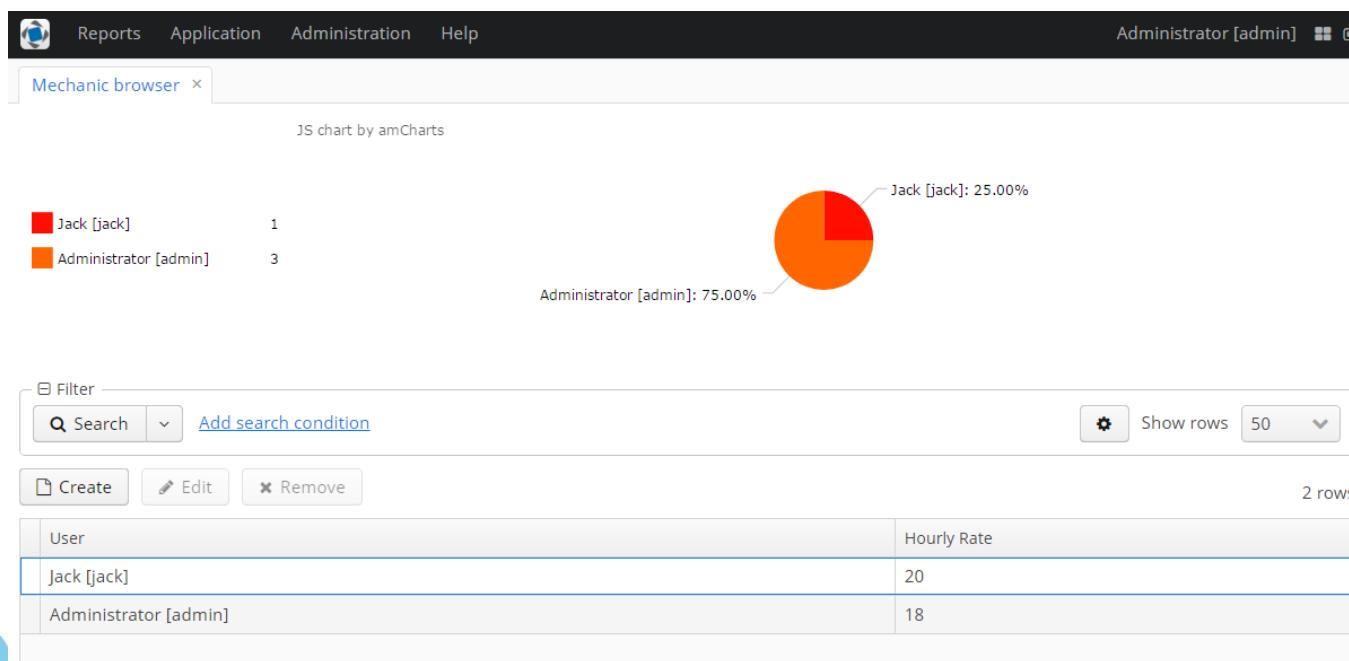
SCREEN DESIGNER

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

Open screen with chart

Reopen **Application — Mechanics** mechanic browser

Now we know exactly who should get a bonus.

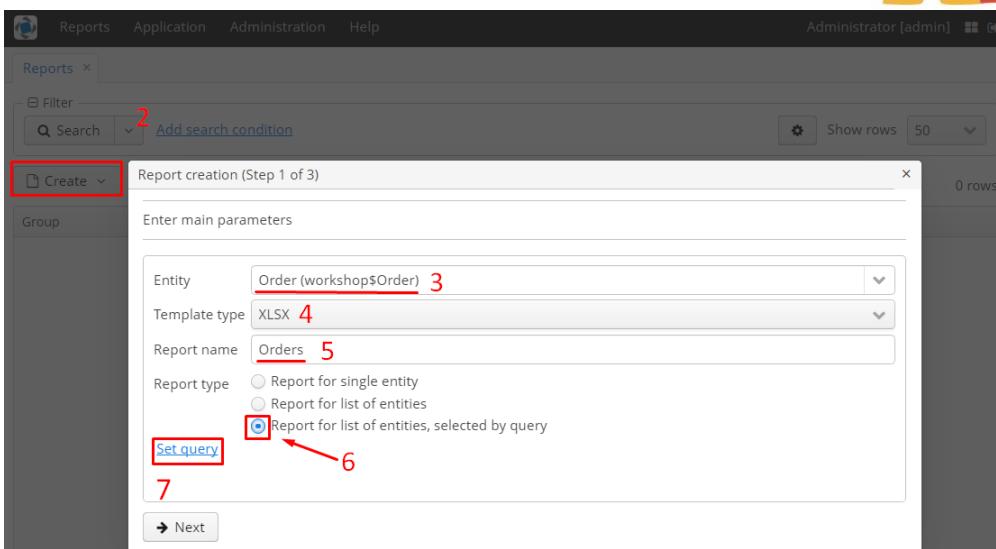


Reporting

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.

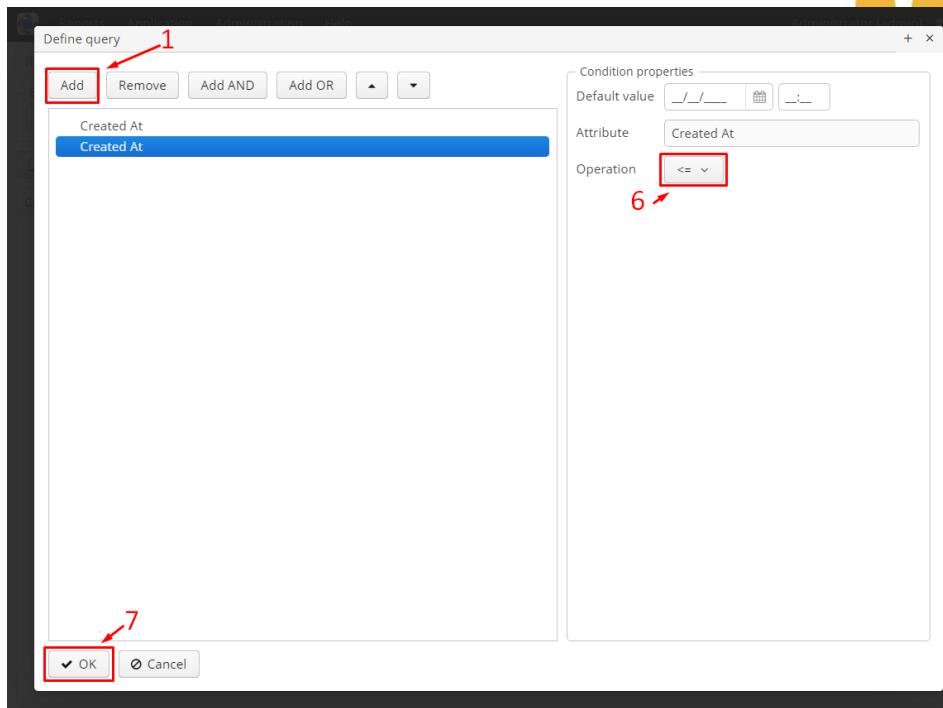
1. Open Reports — Reports from the menu
2. Click Create — Using wizard
3. Select Entity: **Order (workshop\$Order)**
4. Set Template type: **XLSX**
5. Set Report Name: **Orders**
6. Select Report type:
Report for list of entities by query
7. Click Set query



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**

Select attributes for the tabulated report region

Select "Order" attributes to include to the table region.
For a chart and table, the first attribute value will be taken as a caption,
other attributes will be taken as data.

Order

- Amount
- Client
 - ▶ Created At
 - Created By
 - Deleted At
 - Deleted By
- Description
- Hours spent
- Mechanic
 - ▶ Status
- Status
 - Updated At
 - Updated By
 - Version

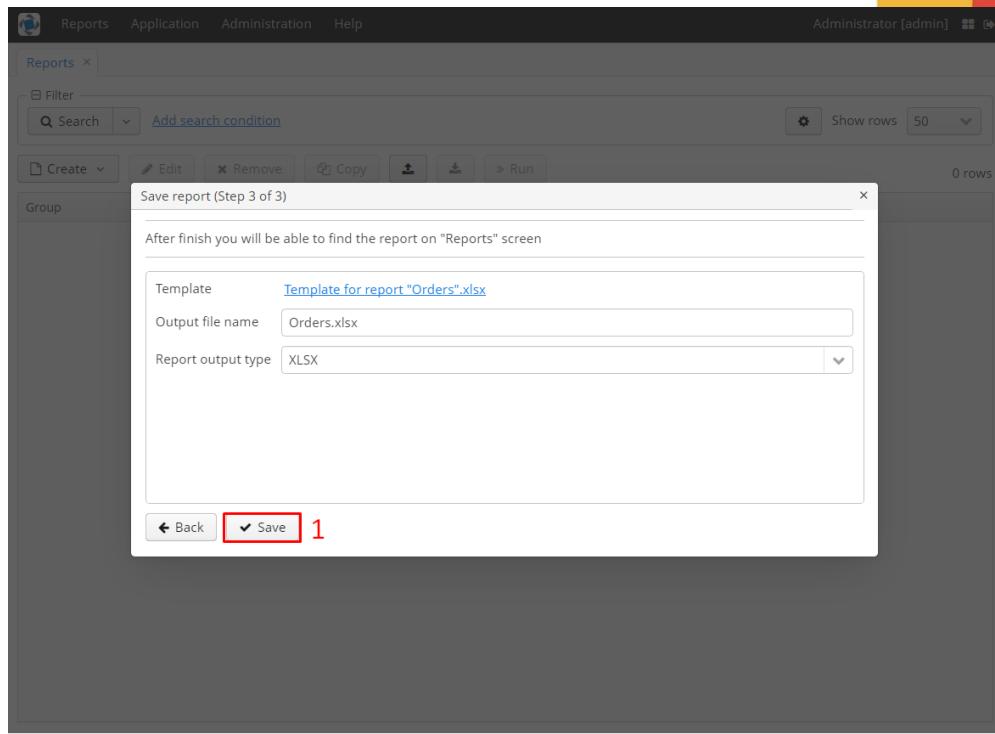
Selected attributes

Order.Created At
Order.Description
Order.Hours spent
Order.Status

◀ OK □ Cancel

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

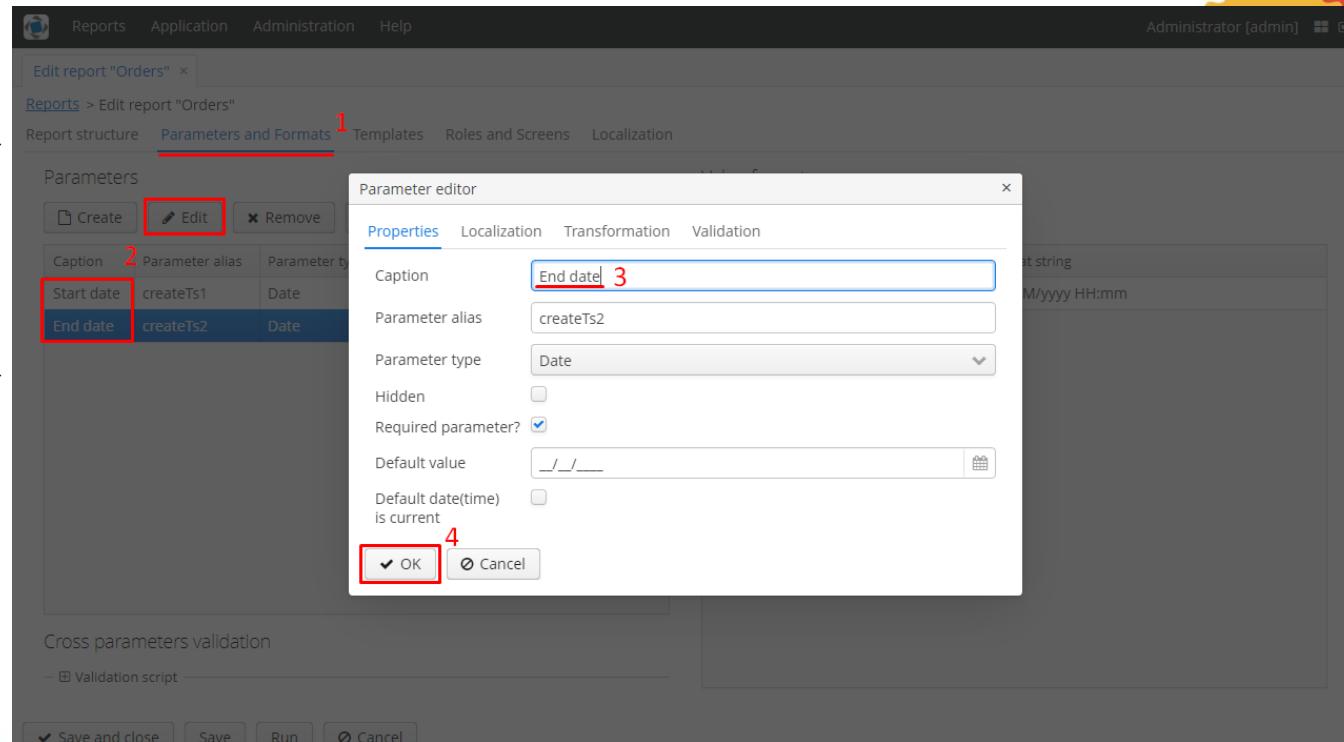
3. Set Parameter Name: **Start date**

4. Click **OK**

5. Edit the **CreateTs2** parameter

6. Set Parameter Name: **End date**

7. Click **OK**



Administrator [admin] Reports > Edit report "Orders"

Report structure Parameters and Formats 1 Templates Roles and Screens Localization

Parameters

Caption	Parameter alias	Parameter type
Start date	createTs1	Date
End date	createTs2	Date

Parameter editor

Properties Localization Transformation Validation

Caption	End date 3
Parameter alias	createTs2
Parameter type	Date
Hidden	<input type="checkbox"/>
Required parameter?	<input checked="" type="checkbox"/>
Default value	/ / /
Default date(time)	<input type="checkbox"/>
is current	<input type="checkbox"/>

OK Cancel 4

Cross parameters validation

Validation script

Save and close Save Run Cancel

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']}")
}
```

Cross parameters validation

Validation script enabled

Validate

Groovy script

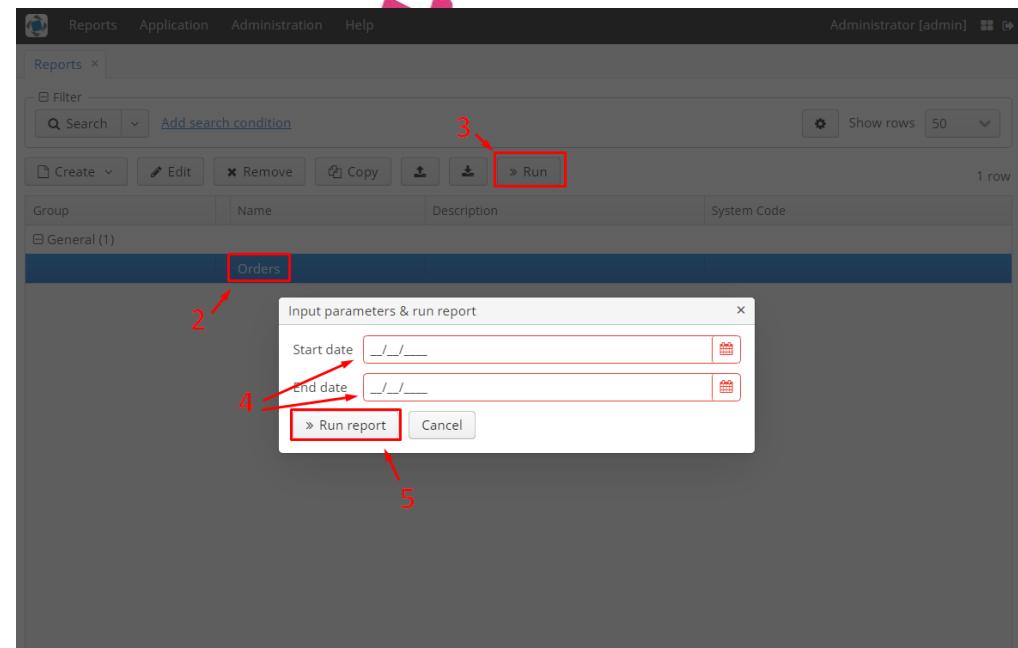
```
1 boolean startDateAfterEndDate =
2     params.createTs1 > params.createTs2
3
4 if (startDateAfterEndDate) [
5     invalid("Start date: ${params['createTs1']} " +
6         "is after End date: ${params['createTs2']}")
7 ]
```



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**

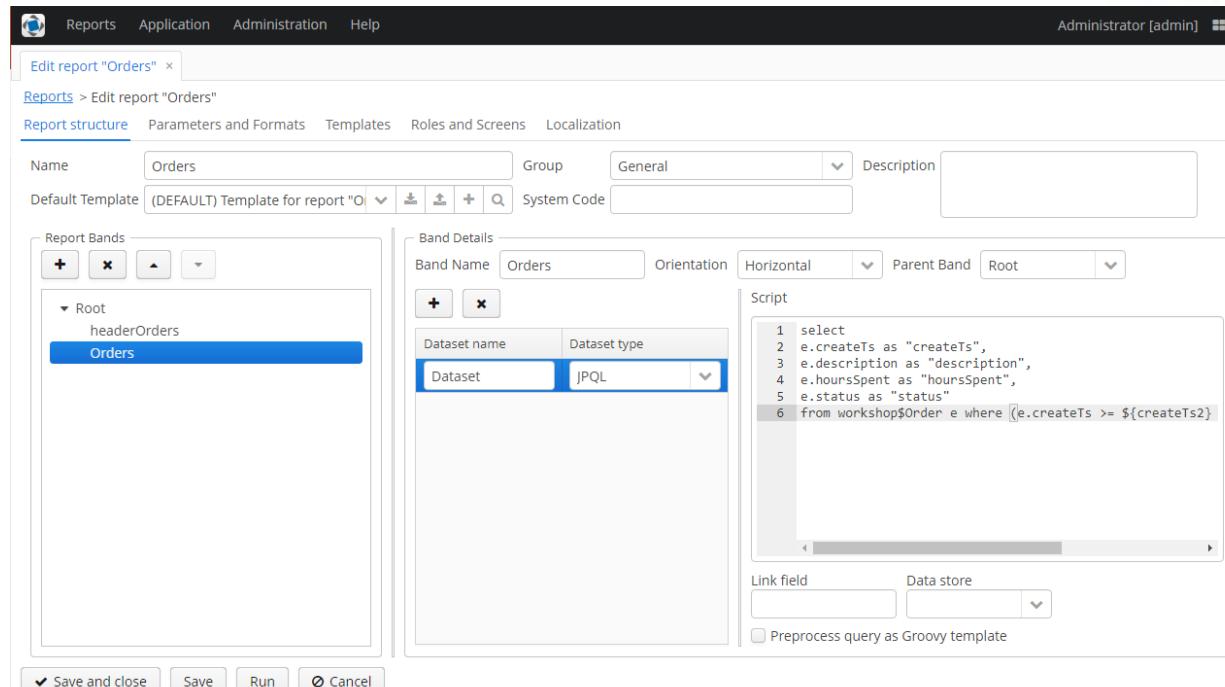
The system has generated an **XSLX file**, we can download it and view its content. Due to the fact that the report templates have the same format as the one that is required for the output, we can easily prepare templates from customer's report examples.



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.



The screenshot shows the 'Edit report "Orders"' screen in the CUBA Report editor. The 'Report structure' tab is selected. In the 'Report Bands' section, there is a tree view with 'Root' expanded, showing 'headerOrders' and 'Orders' under it. The 'Orders' node is highlighted. In the 'Band Details' section, the 'Band Name' is set to 'Orders', 'Orientation' is 'Horizontal', and 'Parent Band' is 'Root'. The 'Script' section displays the following JPQL code:

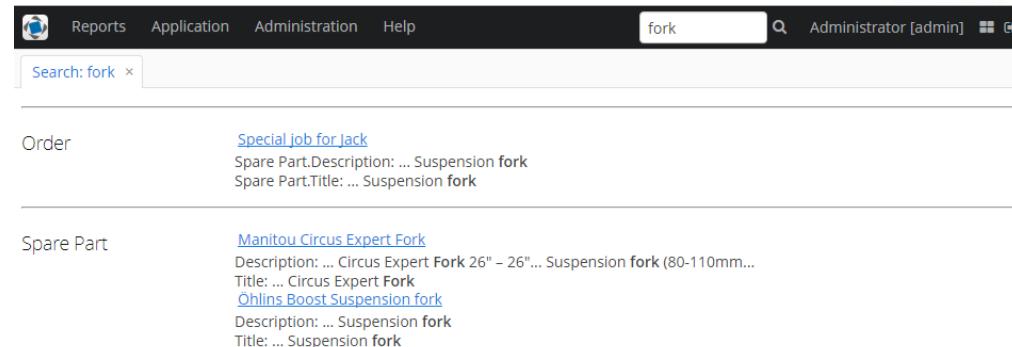
```
1 select
2 e.createTs as "createTs",
3 e.description as "description",
4 e.hoursSpent as "hoursSpent",
5 e.status as "status"
6 from workshop$Order e where (e.createTs >= ${createTs2}
```

At the bottom, there are buttons for 'Save and close', 'Save', 'Run', and 'Cancel'.

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 screenshot shows a web-based application interface for the CUBA Platform. At the top, there is a navigation bar with links for Reports, Application, Administration, and Help. On the right side of the navigation bar, there is a search bar containing the text "fork", a magnifying glass icon, and the text "Administrator [admin]". Below the navigation bar, there is a search results table with two rows. The first row is for an "Order" and the second row is for a "Spare Part". Both rows contain a link to a detailed view ("Special job for Jack" and "Manitou Circus Expert Fork" respectively) and some descriptive text. The background of the slide features large, semi-transparent puzzle pieces in various colors (blue, yellow, green, red) scattered across the screen.

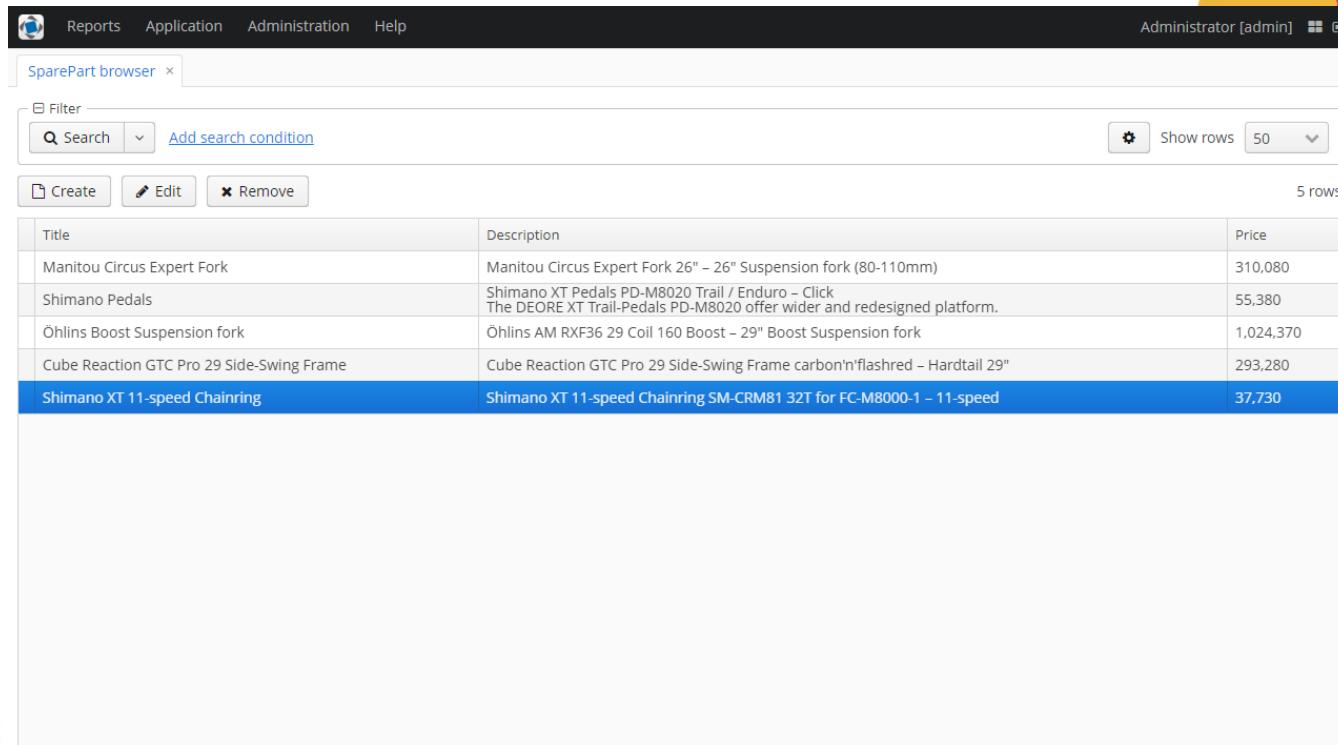
Order	Special job for Jack Spare Part.Description: ... Suspension fork Spare Part.Title: ... Suspension fork
Spare Part	Manitou Circus Expert Fork Description: ... Circus Expert Fork 26" - 26" ... Suspension fork (80-110mm... Title: ... Circus Expert Fork Ohlins Boost Suspension fork Description: ... Suspension fork Title: ... Suspension fork

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



The screenshot shows the CUBA platform's SparePart browser interface. The top navigation bar includes Reports, Application, Administration, Help, and a user account for Administrator [admin]. The main window title is "SparePart browser". Below the title is a search bar with a "Search" button and a "Add search condition" link. There are also "Create", "Edit", and "Remove" buttons. To the right of the search bar are settings for "Show rows" (set to 50) and a "5 rows" indicator. The main content area displays a table of spare parts:

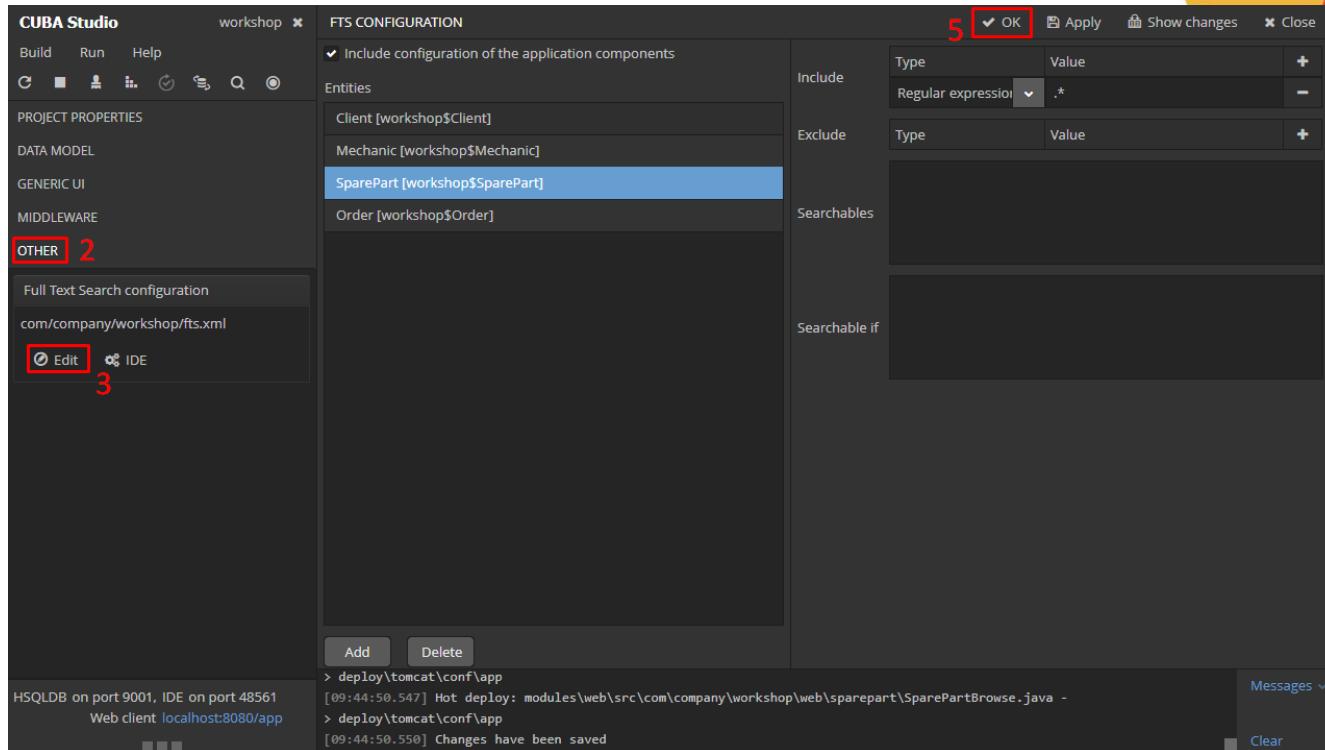
Title	Description	Price
Manitou Circus Expert Fork	Manitou Circus Expert Fork 26" – 26" Suspension fork (80-110mm)	310,080
Shimano Pedals	Shimano XT Pedals PD-M8020 Trail / Enduro – Click The DEORE XT Trail-Pedals PD-M8020 offer wider and redesigned platform.	55,380
Öhlins Boost Suspension fork	Öhlins AM RXF36 29 Coil 160 Boost – 29" Boost Suspension fork	1,024,370
Cube Reaction GTC Pro 29 Side-Swing Frame	Cube Reaction GTC Pro 29 Side-Swing Frame carbon'n'flashred – Hardtail 29"	293,280
Shimano XT 11-speed Chainring	Shimano XT 11-speed Chainring SM-CRM81 32T for FC-M8000-1 – 11-speed	37,730



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**



The screenshot shows the CUBA Studio interface with the 'workshop' project selected. On the left, the navigation panel has 'OTHER' selected (marked with a red box). In the center, a dialog box titled 'FTS CONFIGURATION' is open. The 'Include' tab is active, showing a list of entities: Client [workshop\$Client], Mechanic [workshop\$Mechanic], SparePart [workshop\$SparePart] (which is highlighted with a blue selection bar), and Order [workshop\$Order]. The 'Exclude' tab is empty. Below these tabs are 'Searchables' and 'Searchable if' sections, both also empty. At the top right of the dialog, there is an 'OK' button with a red box around it, and other buttons for 'Apply', 'Show changes', and 'Close'. At the bottom of the dialog, there are 'Add' and 'Delete' buttons, and a message log showing deployment logs for 'tomcat' and 'Changes have been saved'. The status bar at the bottom of the studio window indicates 'HSQldb on port 9001, IDE on port 48561' and 'Web client localhost:8080/app'.



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

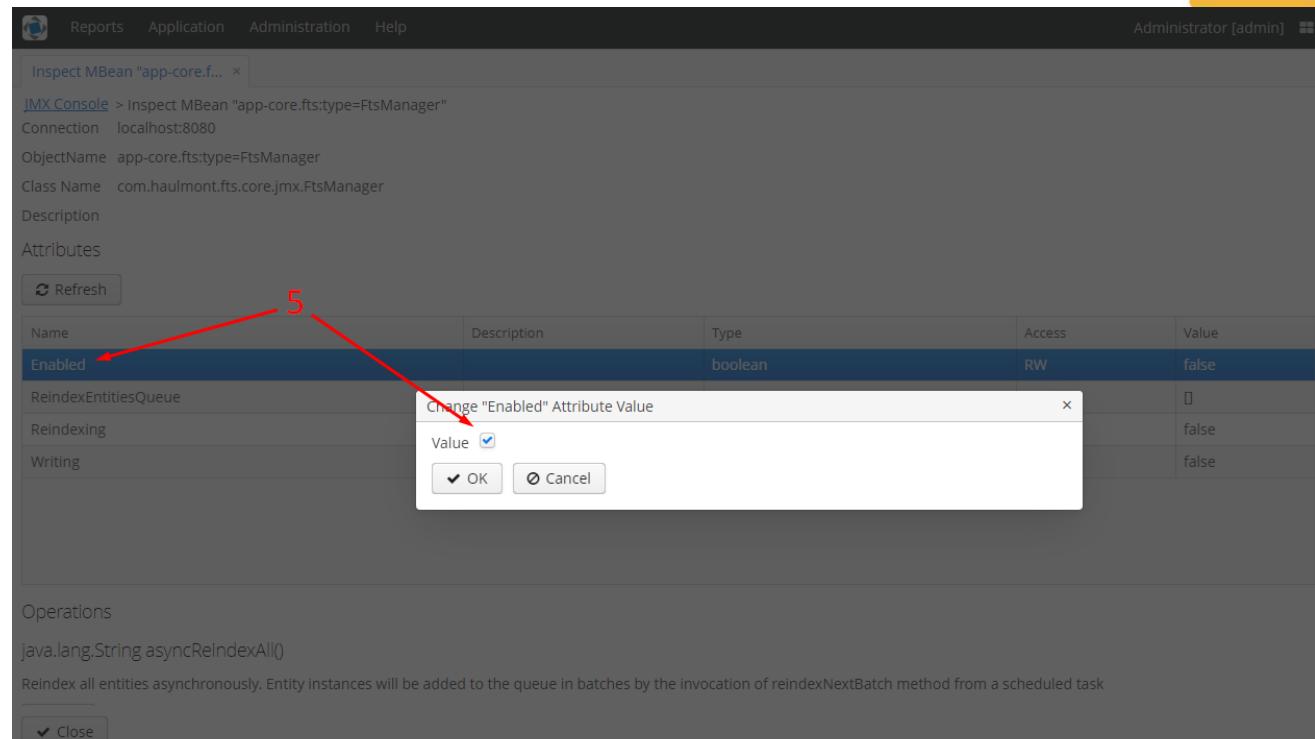
2. This is a web version of the console for the JMX interface; it allows us to manage internal system mechanisms

3. Find **FtsManager** using the

Search by ObjectName field

4. Open **FtsManager**

5. Change the **Enabled** property to true



Inspect MBean "app-core.fts:type=FtsManager"

IMX Console > Inspect MBean "app-core.fts:type=FtsManager"

Connection localhost:8080

ObjectName app-core.fts:type=FtsManager

Class Name com.haulmont.fts.core.jmx.FtsManager

Description

Attributes

Name	Description	Type	Access	Value
Enabled		boolean	RW	false
ReindexEntitiesQueue				
Reindexing				false
Writing				

Operations

java.lang.String asyncReindexAll()

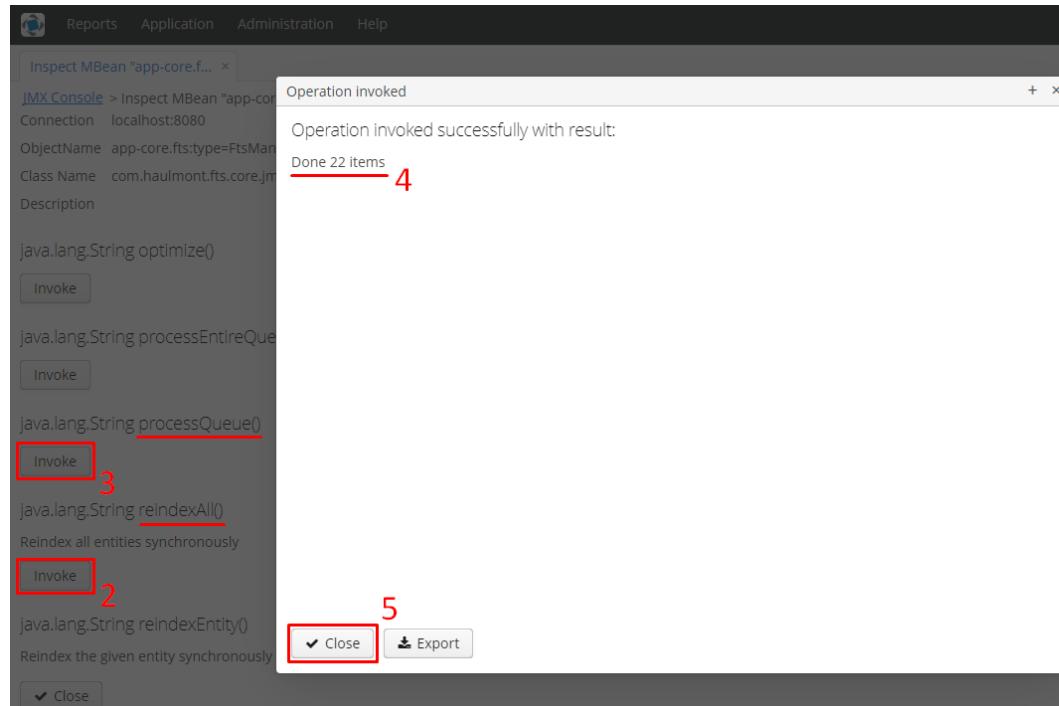
Reindex all entities asynchronously. Entity instances will be added to the queue in batches by the invocation of reindexNextBatch method from a scheduled task

Value

OK Cancel

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

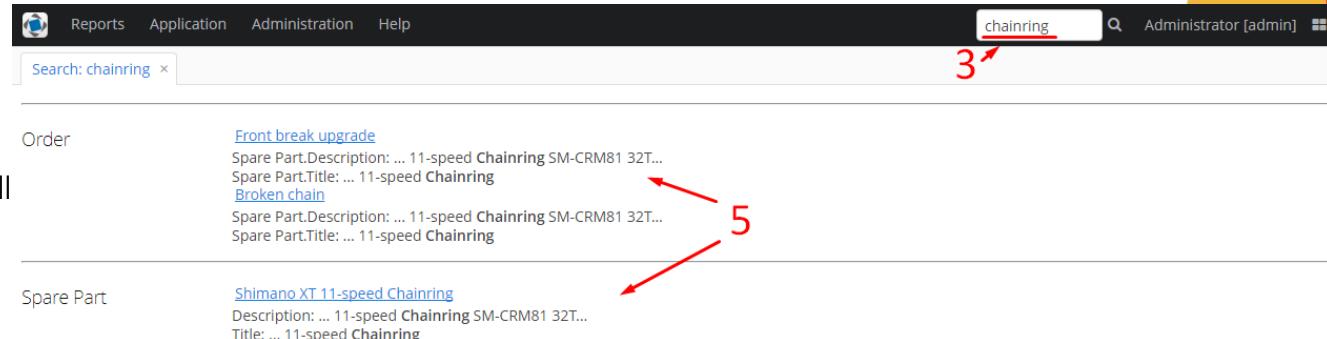
1. **Log out** of the system

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



The screenshot shows the CUBA application interface with a search bar at the top containing the text "chainring". A red arrow labeled "3" points to the search bar. Below the search bar, there are two sections of search results:

- Order**:
 - [Front break upgrade](#)
Spare Part.Description: ... 11-speed Chainring SM-CRM81 32T...
Spare Part.Title: ... 11-speed Chainring
[Broken chain](#)
Spare Part.Description: ... 11-speed Chainring SM-CRM81 32T...
Spare Part.Title: ... 11-speed Chainring
- Spare Part**:
 - [Shimano XT 11-speed Chainring](#)
Description: ... 11-speed Chainring SM-CRM81 32T...
Title: ... 11-speed Chainring

A red arrow labeled "5" points from the text "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" to the search results section.

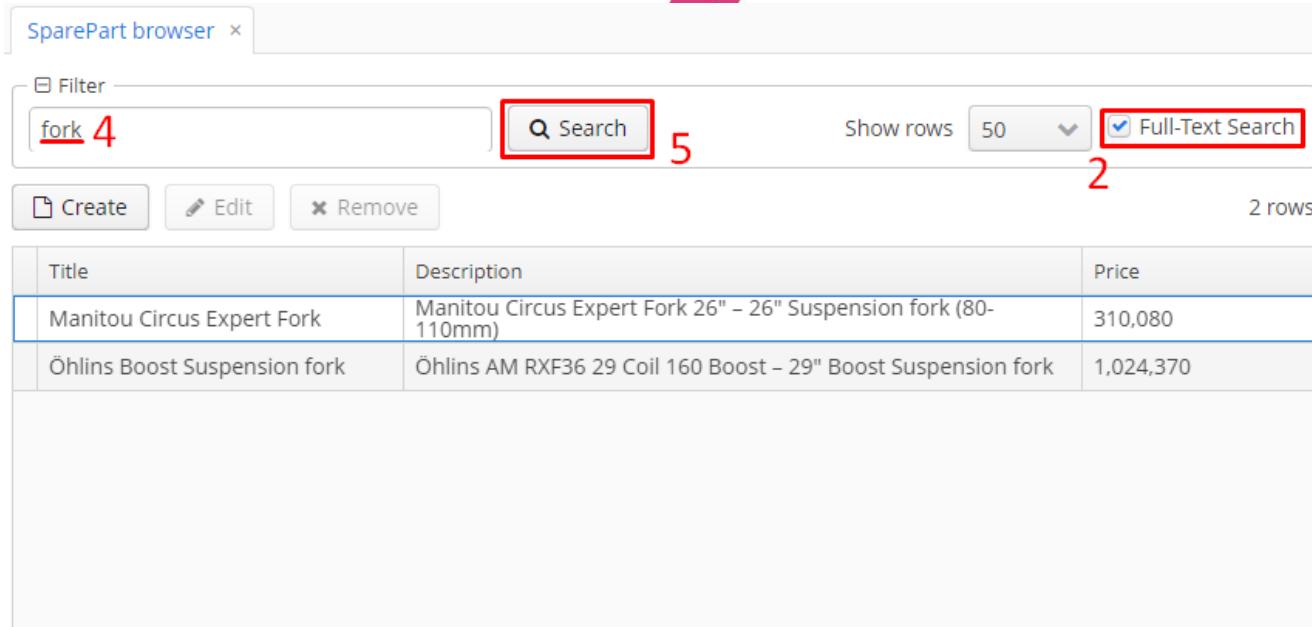




FTS integration with filters

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**



The screenshot shows a 'SparePart browser' window. At the top, there is a filter panel with a text input field containing 'fork' (labeled 4), a 'Search' button (labeled 5), and a checked 'Full-Text Search' checkbox (labeled 2). Below the filter panel is a table with three columns: Title, Description, and Price. Two rows are visible in the table.

Title	Description	Price
Manitou Circus Expert Fork	Manitou Circus Expert Fork 26" – 26" Suspension fork (80-110mm)	310,080
Öhlins Boost Suspension fork	Öhlins AM RXF36 29 Coil 160 Boost – 29" Boost Suspension fork	1,024,370

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 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.

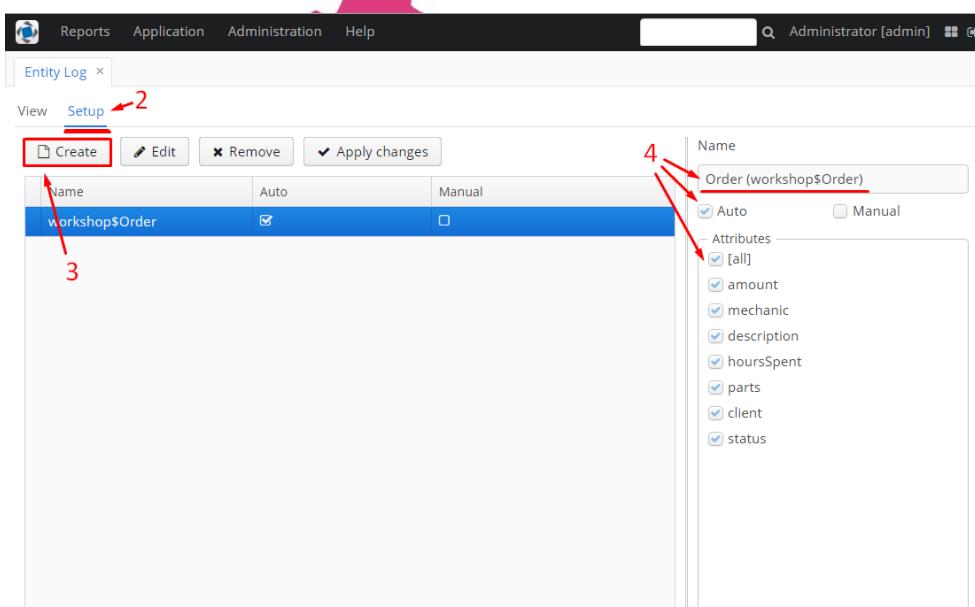
Let's apply it for logging order changes.

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**



The screenshot shows the CUBA application interface with the following details:

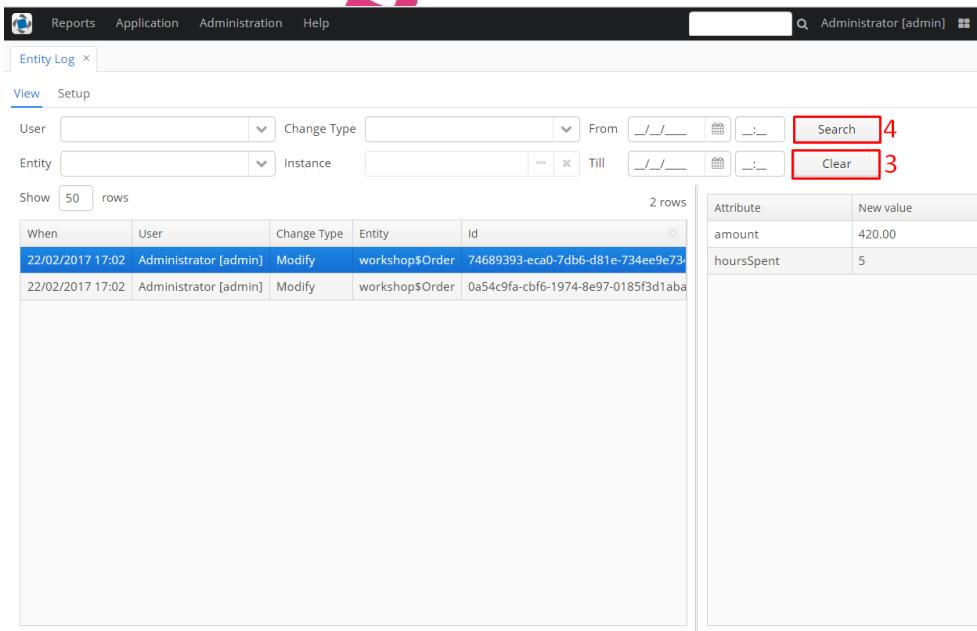
- Top Bar:** Reports, Application, Administration, Help, Search bar, Administrator [admin], and a user icon.
- Entity Log Tab:** The "Entity Log" tab is selected.
- Setup Tab:** The "Setup" tab is active, indicated by a red arrow labeled "2".
- Create Button:** A red box highlights the "Create" button, with a red arrow labeled "3" pointing to it.
- Form Fields:** Name: workshop\$Order, Auto: true (checkbox checked), and Manual (checkbox unselected).
- Right Panel (Attributes):**
 - Name:** Order (workshop\$Order) (highlighted with a red box and arrow labeled "4").
 - Auto:** Auto (checkbox checked), Manual (checkbox unselected).
 - Attributes:** A list of checked checkboxes: [all], amount, mechanic, description, hoursSpent, parts, client, and status.



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.

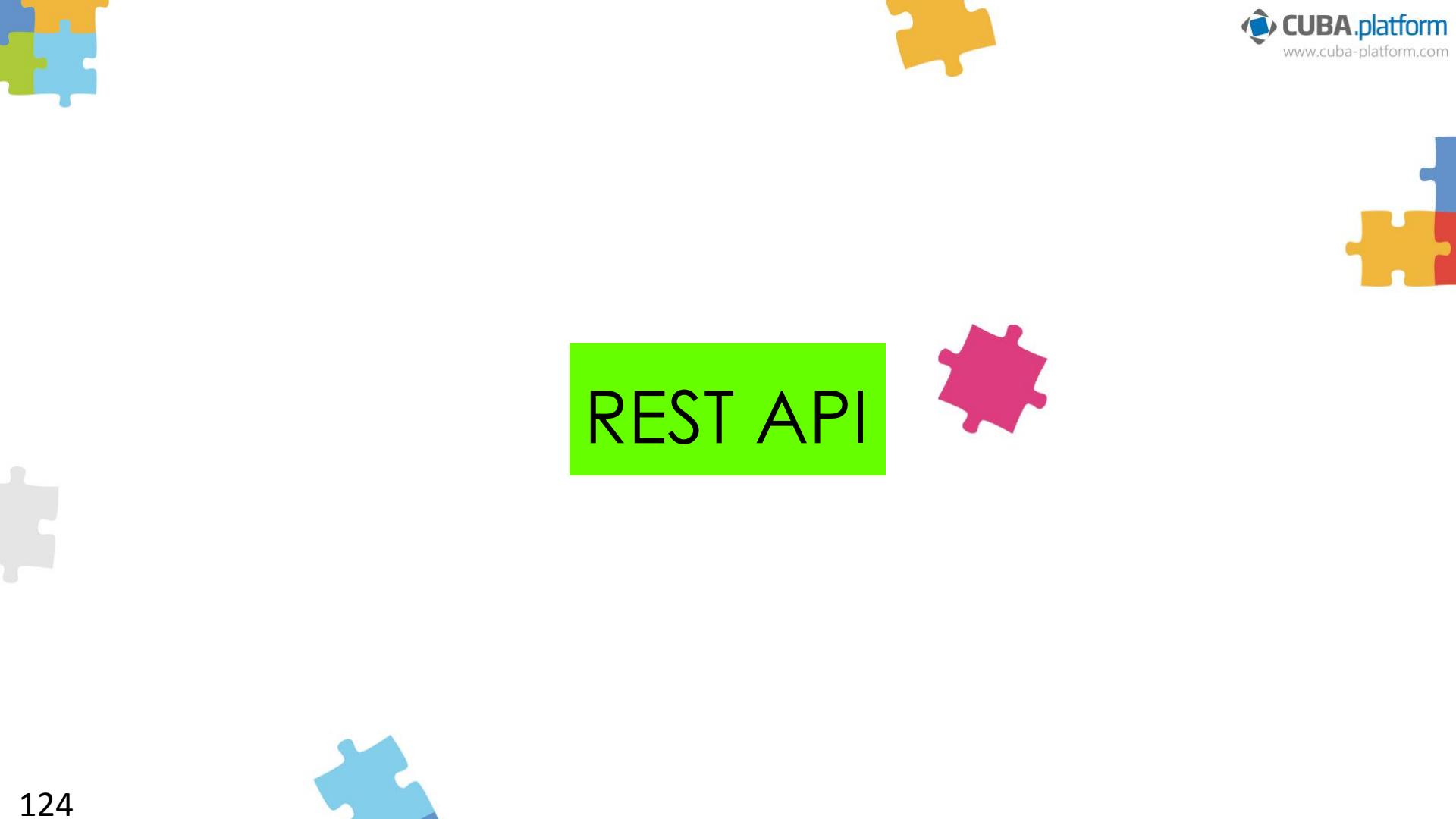


The screenshot shows the Entity Log page with the following details:

- Search Bar:** Contains fields for User, Change Type, From, Till, and a Search button (labeled 4).
- Filter Bar:** Contains fields for Entity, Instance, and a Clear button (labeled 3).
- Table Headers:** When, User, Change Type, Entity, Id.
- Table Data:** Two rows of audit log entries.

When	User	Change Type	Entity	Id
22/02/2017 17:02	Administrator [admin]	Modify	workshop\$Order	74689393-eca0-7db6-d81e-734ee9e73c
22/02/2017 17:02	Administrator [admin]	Modify	workshop\$Order	0a54c9fa-cbf6-1974-8e97-0185f3d1aba
- Table on the Right:** Shows attribute changes with their new values.

Attribute	New value
amount	420.00
hoursSpent	5



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.

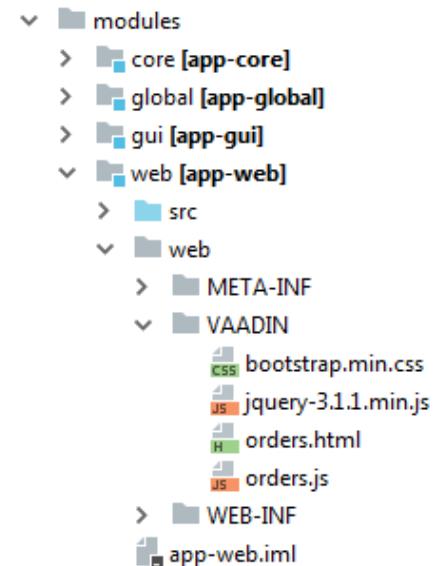
REST API is enabled by default in the web client, thus we can use it right now for a simple web page that will show the list of recent orders for a logged in user.

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:

Bike Workshop

Login:

admin

Password:

.....

Submit

```
<!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.

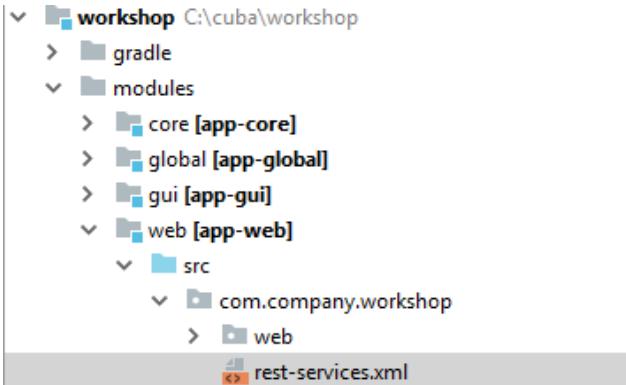
```
@Override
public BigDecimal calculateAmount(UUID orderId) {
    LoadContext<Order> context = LoadContext.create(Order.class)
        .setQuery(LoadContext.createQuery("select o from workshop$Order o where o.id = :orderId")
            .setParameter("orderId", orderId).setView("order-view"));
   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



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

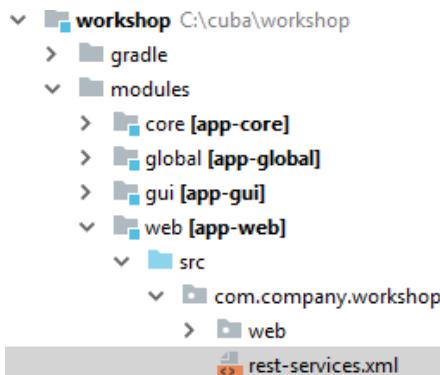




Service Method Invocation (GET)

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



```
#####
# Configuration
#####
cuba.rest.servicesConfig = +com/company/workshop/rest-services.xml

cuba.springContextConfig = +com/company/workshop/web-spring.xml

cuba.dispatcherSpringContextConfig = +com/company/workshop/web-dispatcher-spring.xml

cuba.persistenceConfig = +com/company/workshop/persistence.xml

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 w 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:

Bike Workshop

Logged in successfully

Orders

Description	Amount
Broken chain	37730
Special job for Jack	1627730
Front break upgrade	93146
Wheel problem	54

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/>