Quick start guide

Quick start guide

Table of Contents

List of Figures

Chapter 1. Pet Hotel sample application

In this tutorial you'll learn how to create a simple business application in JavaScript using Platypus.js. The sample source code is available for download from the Platypus.js project's repository [https://github.com/altsoft/pethotel] on github.

Please refer to the Development Guide and the Administration Guide for more details about Platypus.js.

Chapter 2. Platypus.js installation

Java Runtime and JDK 8 update 66 or higher are required for Platypus.js applications development.

Install the Platypus.js IDE, the Platypus.js runtime components and Apache Tomcat 8.0.24+ to enable running a web client:

- Download [http://platypus-platform.org/download.html] the Platypus.js installer bundle from the project’s website for your operating system.
- Run the installer.
- Follow the wizard’s steps and install the required components.

Chapter 3. The sample application requirements

The Pet Hotel is a simple business application for cats and dogs hotel accounting.

The hotel’s administrator role is the only user role defined for this application.

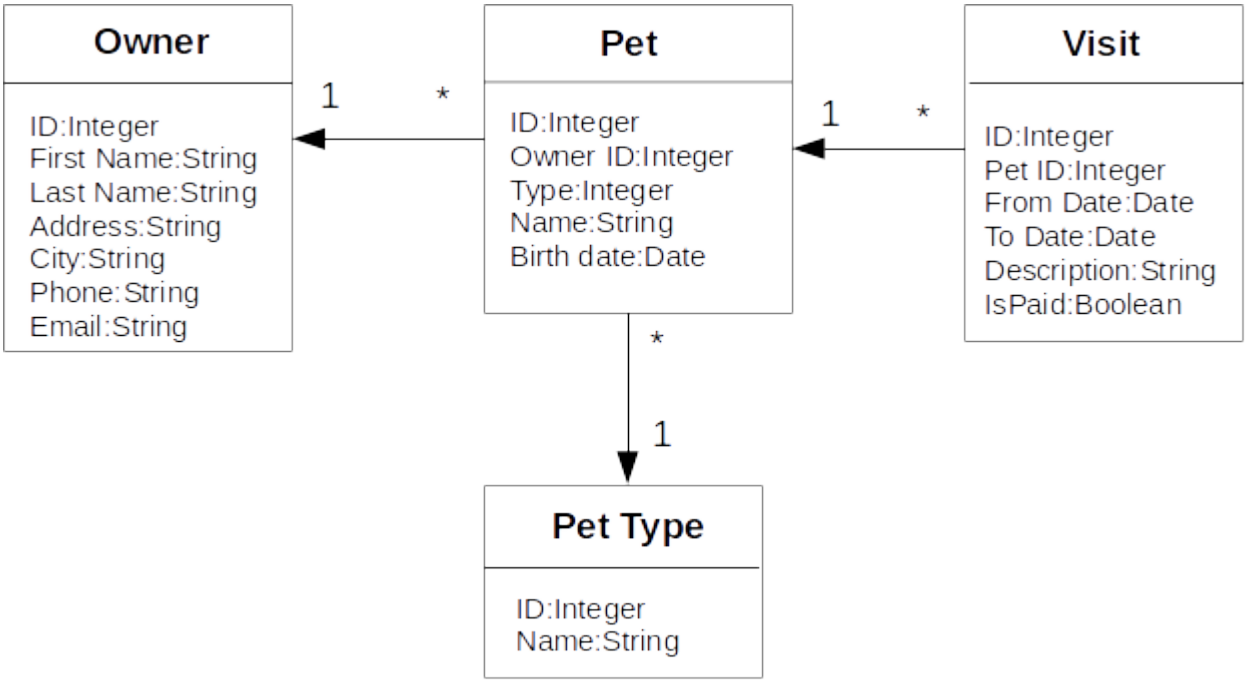
An administrator can perform the following actions:

- Search for the pet owners by her/his name and display the owners list.
- Add, delete and update owner’s data.
- Display an owner’s pets as a list.
- Add, delete and update an owner’s pets records.
- Add, delete and update the hotel attendance data for a pet.
- Display and print the owners report.

The entered data must be validated according to the following rules:

- All owner’s and pet’s data fields are mandatory.
- The check-in date must precede the check-out date. Database structure is shown on following diagram (Fig.model diagram).

Figure 3.1. Pet Hotel knowledge domain model diagram

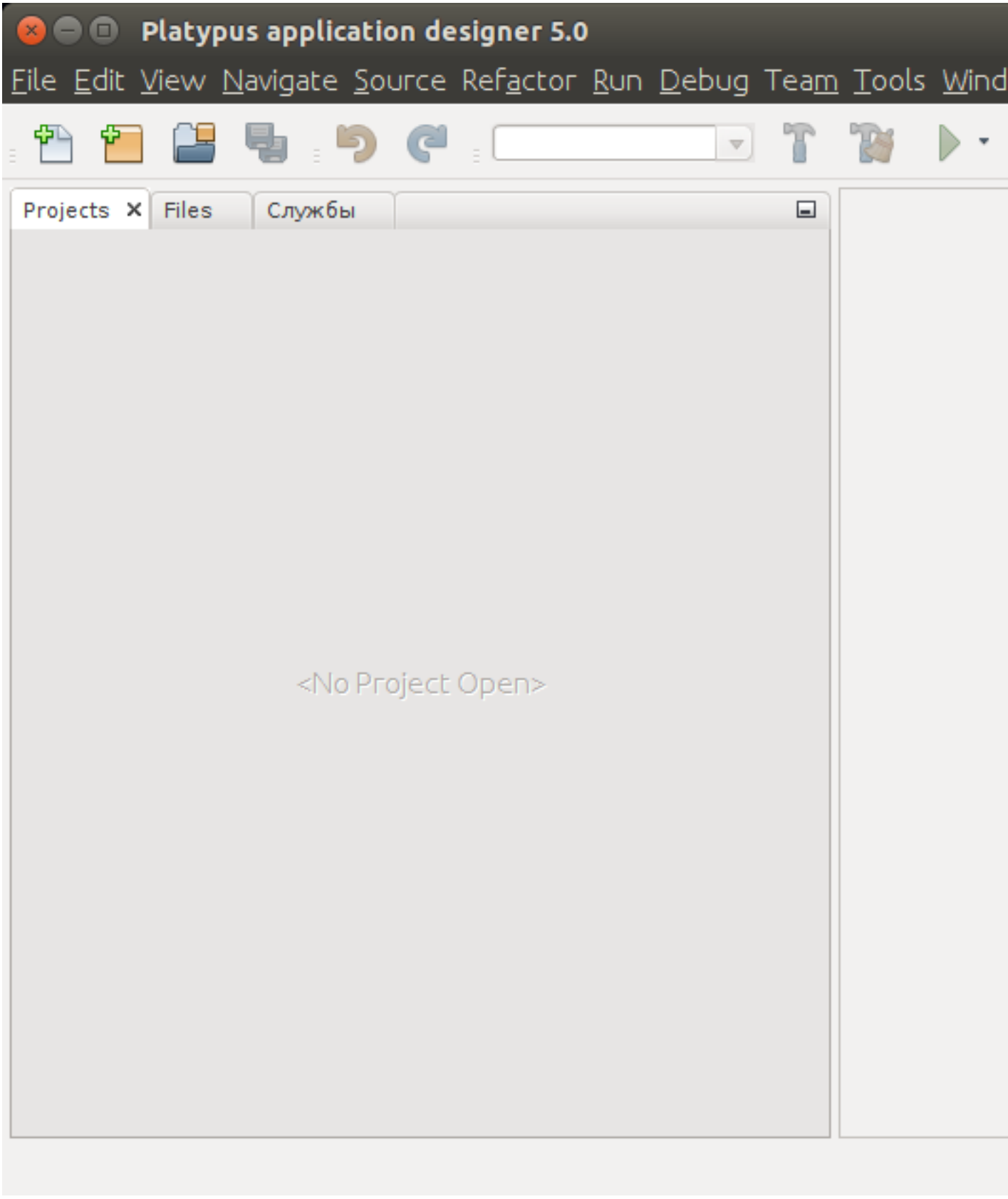


Chapter 4. The sample project creation and setup

During the installation process the Platypus.js IDE is configured for the correct path to the runtime directory, Tomcat and a default database connection. Check the platform’s runtime directory on Tools Platypus Platform menu item in the global menu.

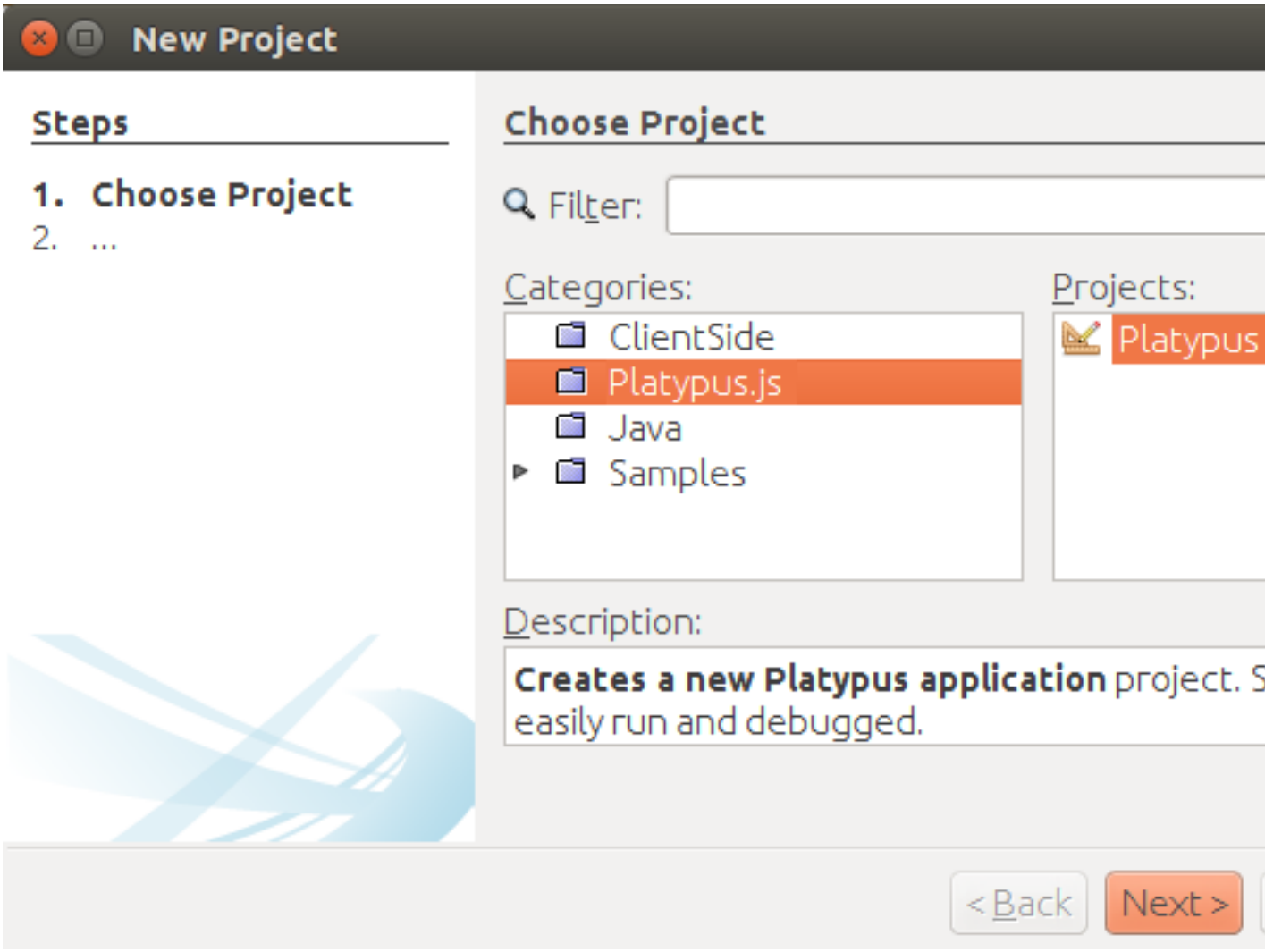
After first launch of Platypus designer you will get window as it shown on Fig. start screen.

Figure 4.1. Platypus designer interface



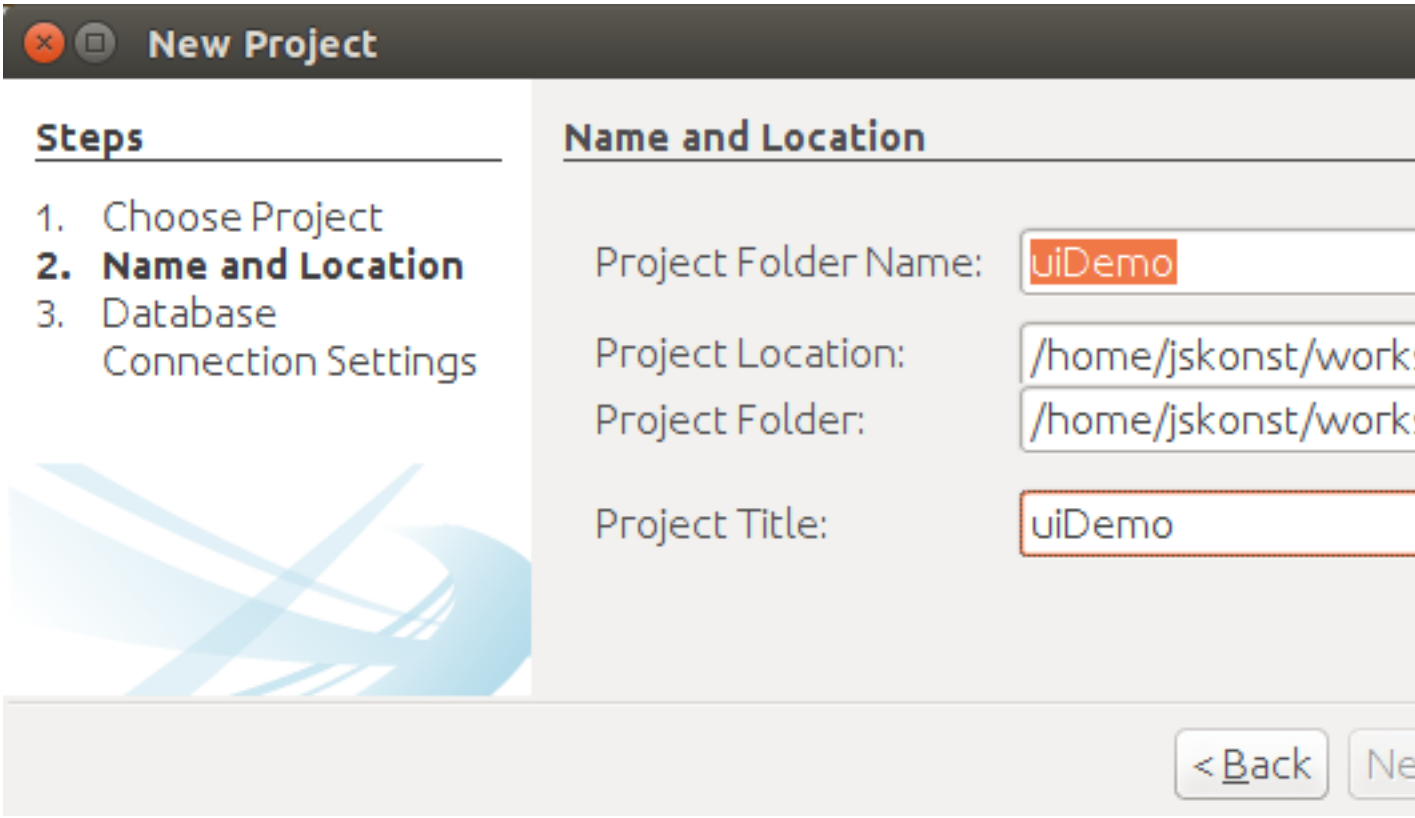
To create Platypus project you should select menu item *File # New Project* (Fig. project creation).

Figure 4.2. Project creation



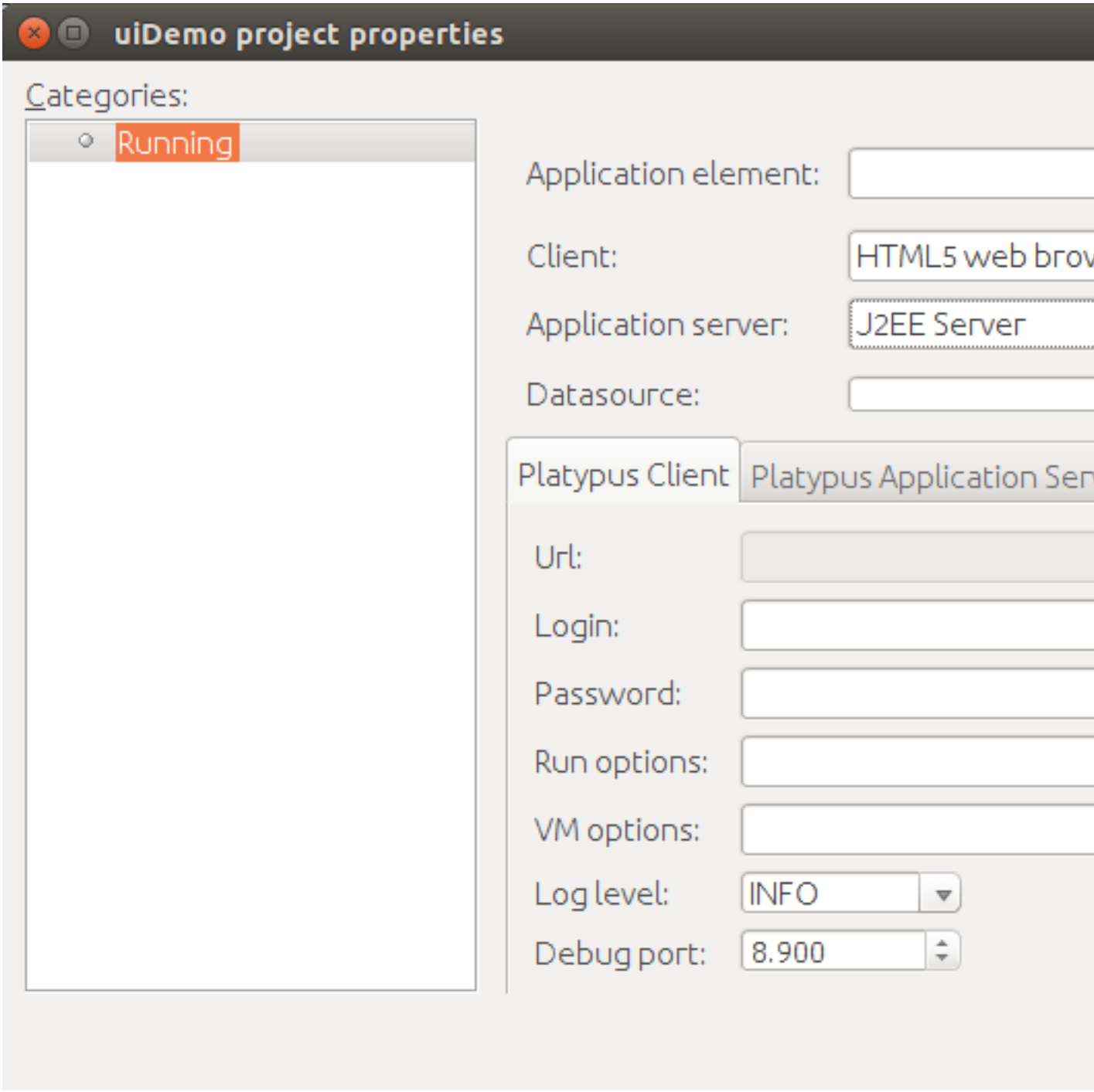
Next, you should set project's name and it's location (Fig. project name).

Figure 4.3. Choose project name



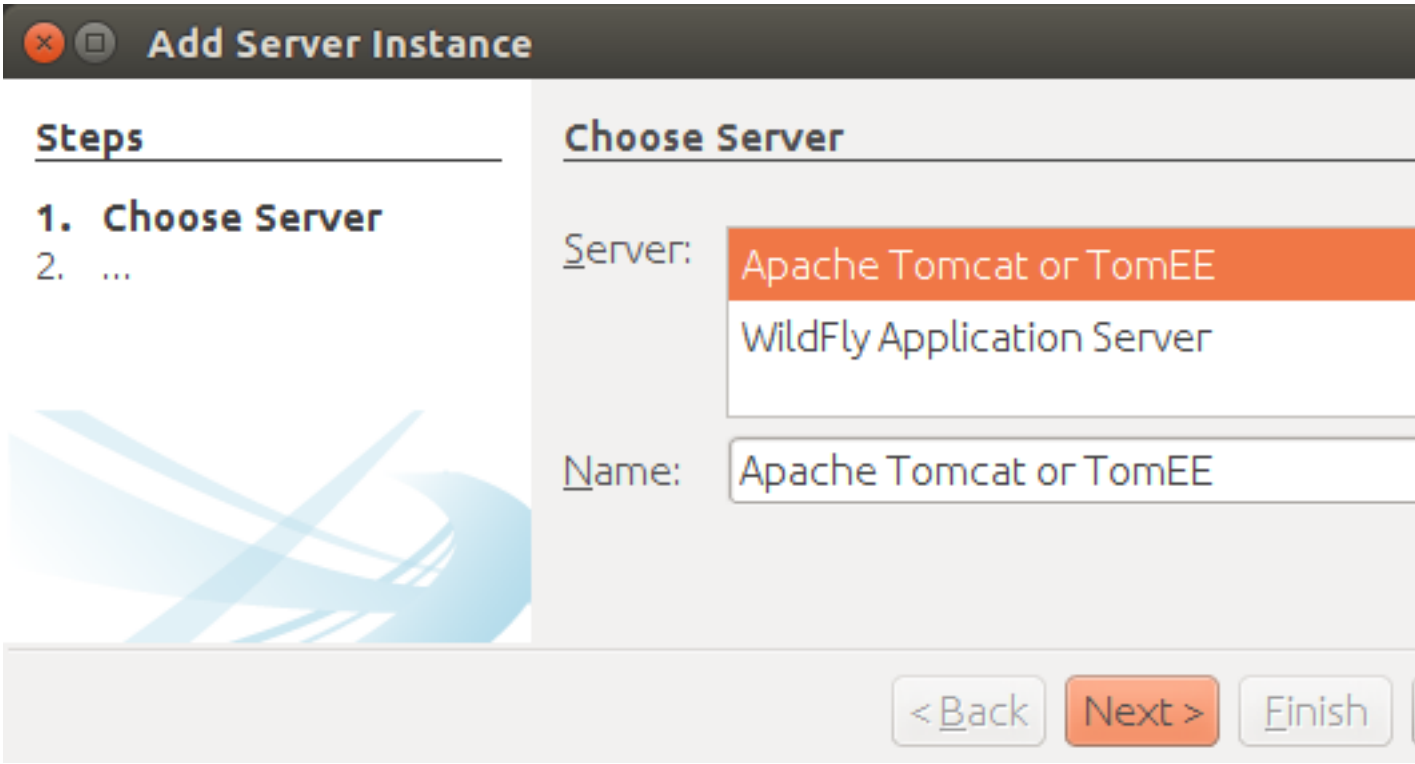
After project was created, we will set it's properties so he could be started as HTML5 application. Perform right mouse button click and in menu choose *Properties*. In modal window,shown on Fig. project properties. Select *Client # HTML5 web browser* and *Application server # J2EE Server*.

Figure 4.4. Setting project properties



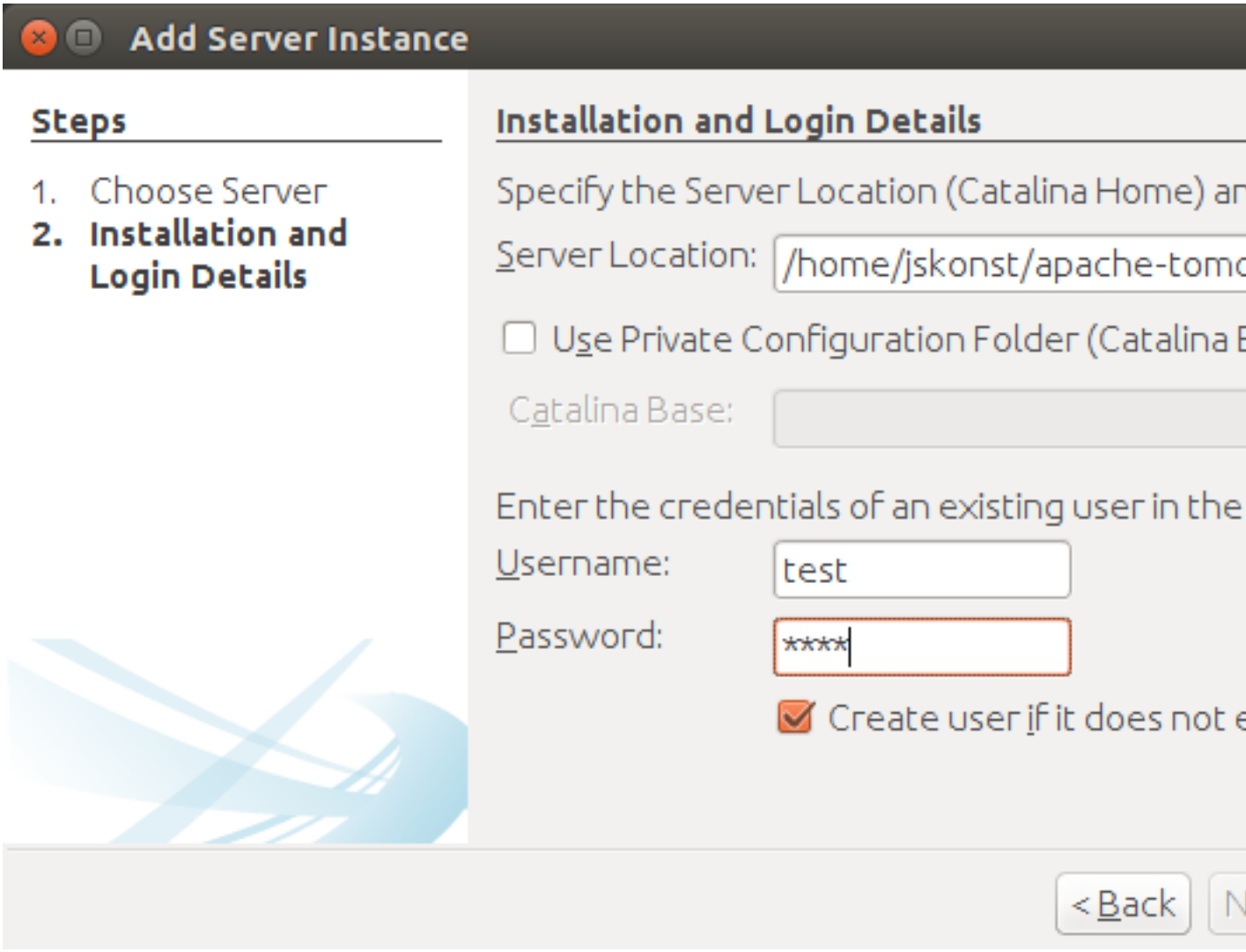
In most cases Apache Tomcat application server is already installed and configured to work with Platypus, so you can skip following steps and check J2EE Server settings tab (Fig. project server). In other cases you should add J2EE server manually. To do this, you should go to Services tab and perform right click on *Servers* item. In context menu choose *Add Server*, on screen you will see following window (Fig. add server).

Figure 4.5. Selecting Java EE server type



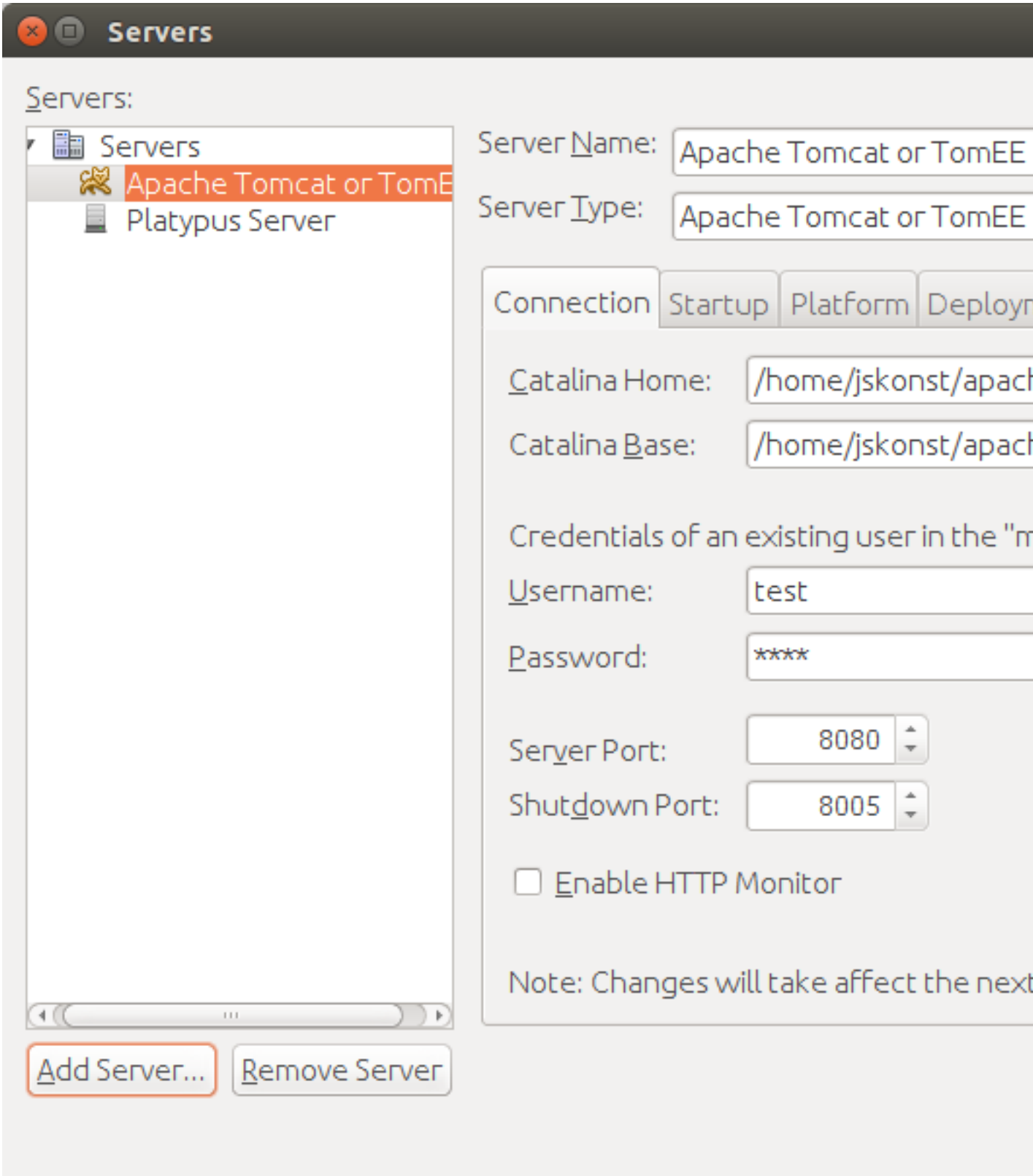
After you have selected server, click the *Next* button. We will show how to set *Apache Tomcat* server. Using next menu (Fig. adding Tomcat), browse server folder location and set Username and password for user manager role.

Figure 4.6. Adding Apache Tomcat server folder



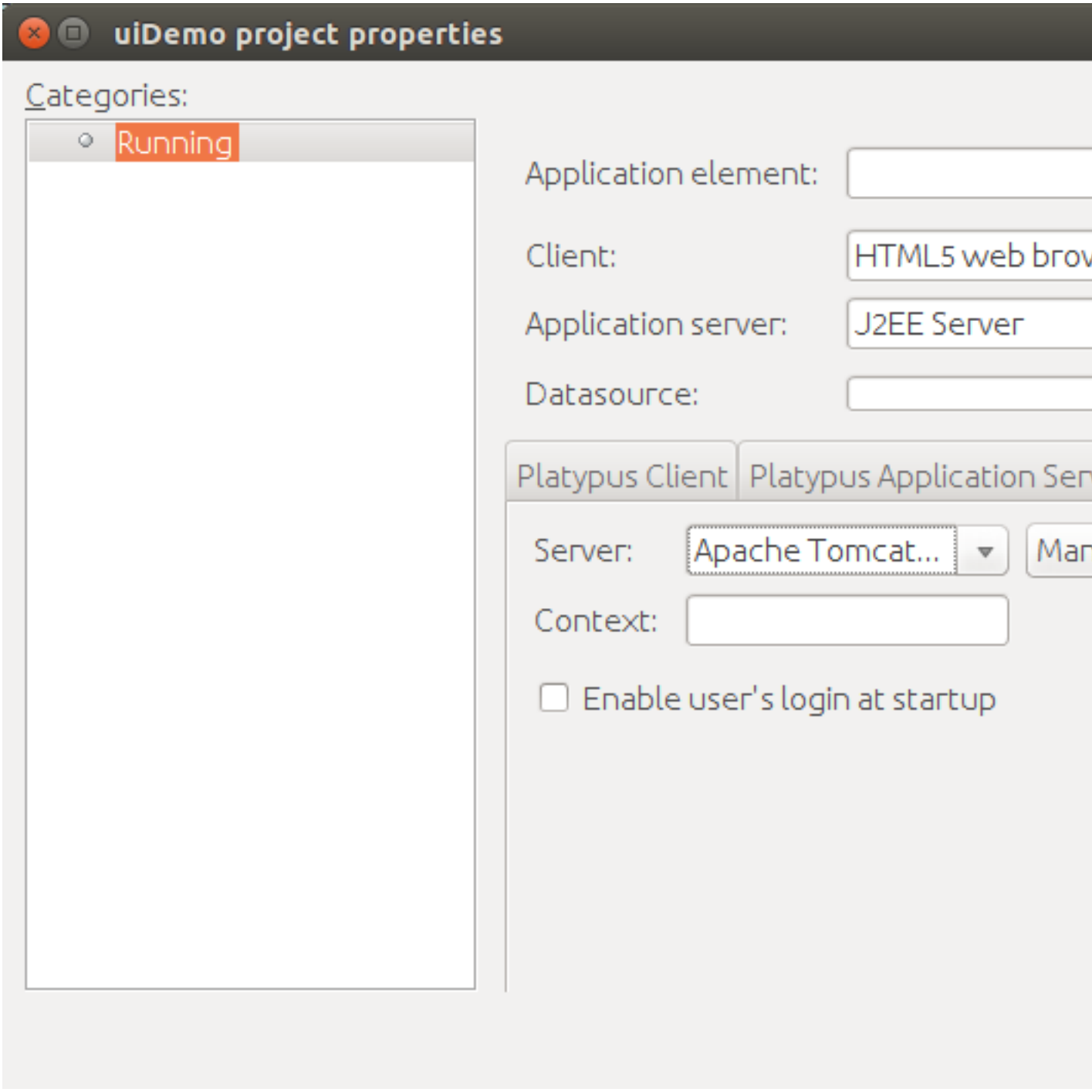
Fine adjustment of Tomcat such as server port, could be performed on Fig. tomcat properties.

Figure 4.7. Tomcat properties



Finally you should set J2EE Server settings for project properties (Fig. project server).

Figure 4.8. Choosing project's Java EE server

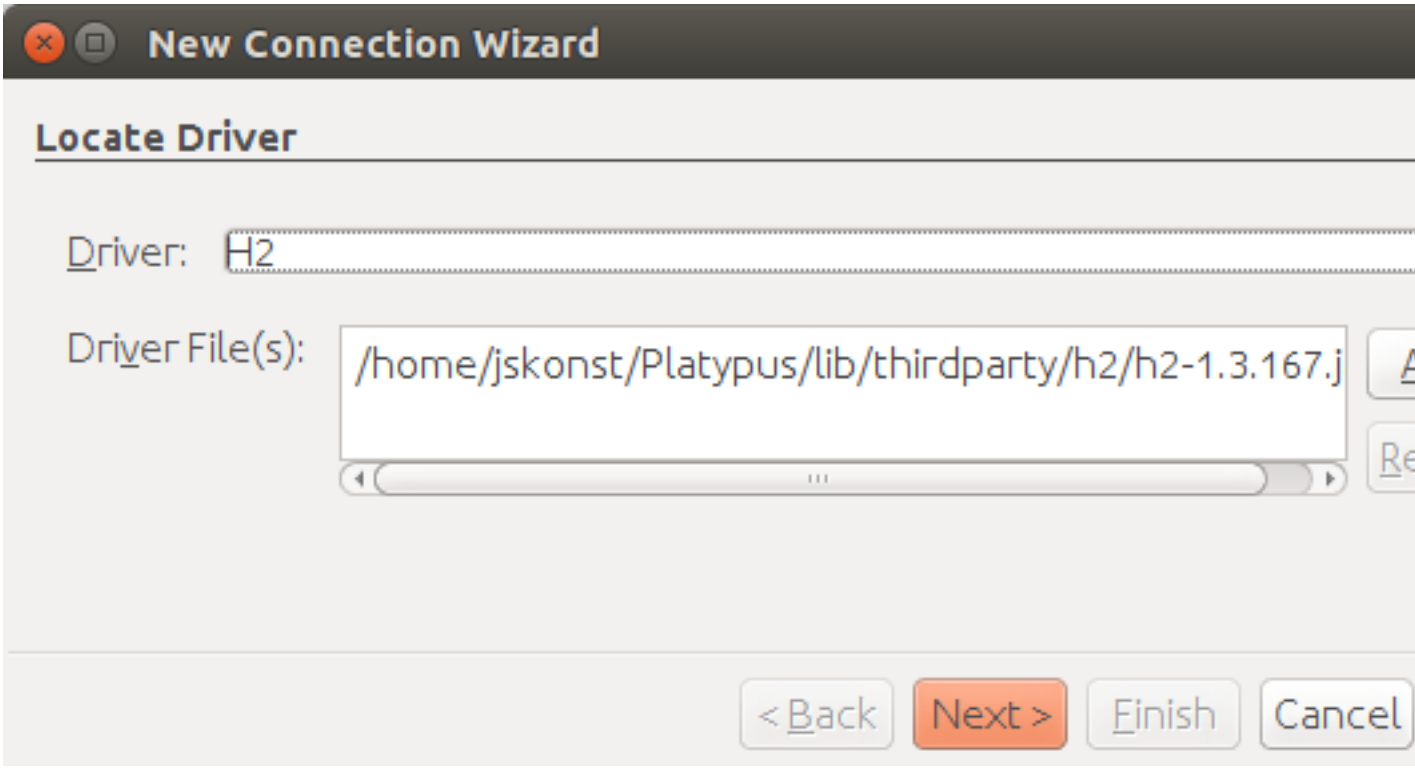


Create a new database connection for the Pet Hotel application. You can use any of the databases supported by the platform. The easiest way is to use H2 database, which is supplied with the platform and does not require any additional configuration or administration.

Use the instructions below to create the H2 datasource connection:

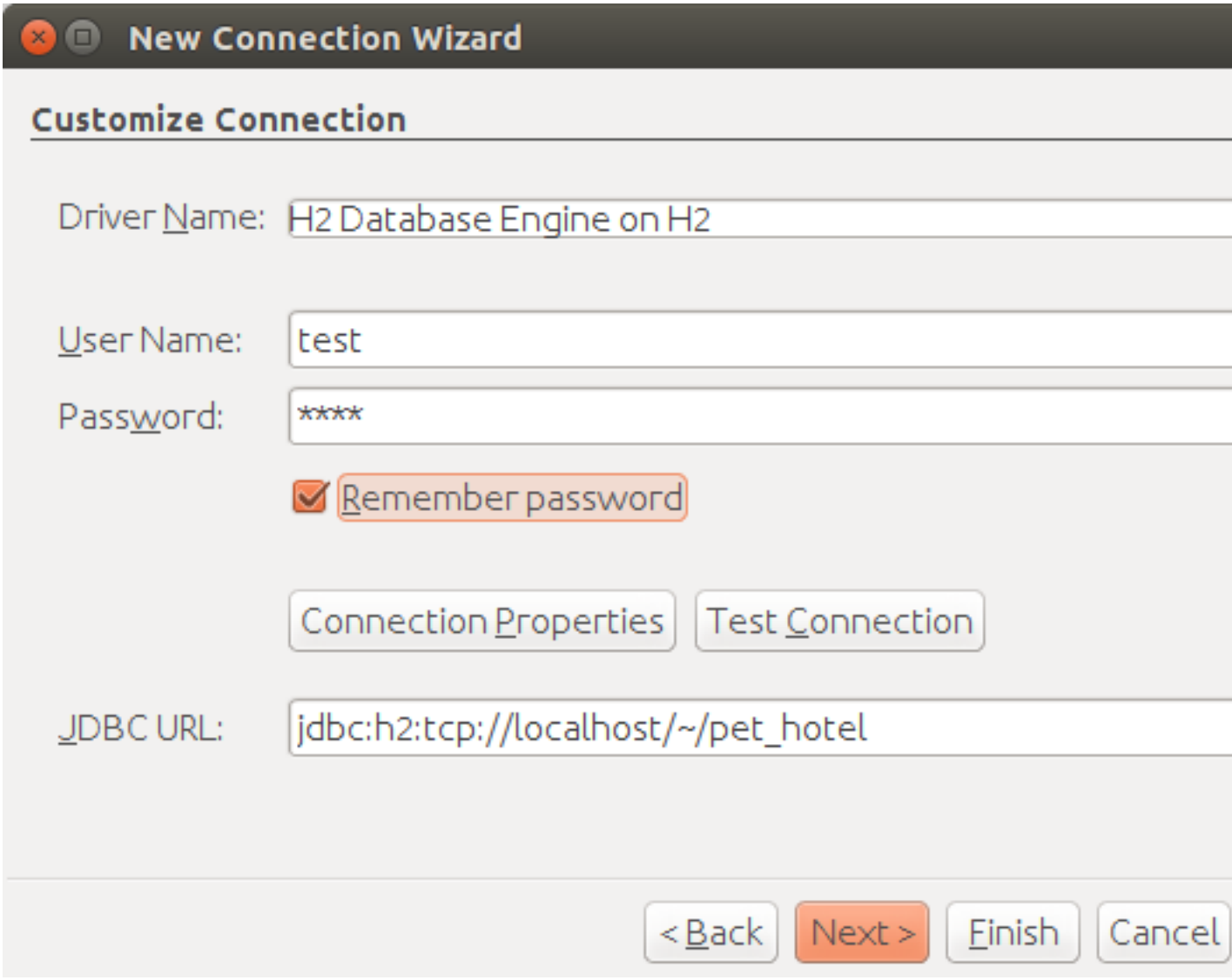
Go to the Services panel. Select the *New Connection…* menu item from the Databases node context menu.

Figure 4.9. Database type



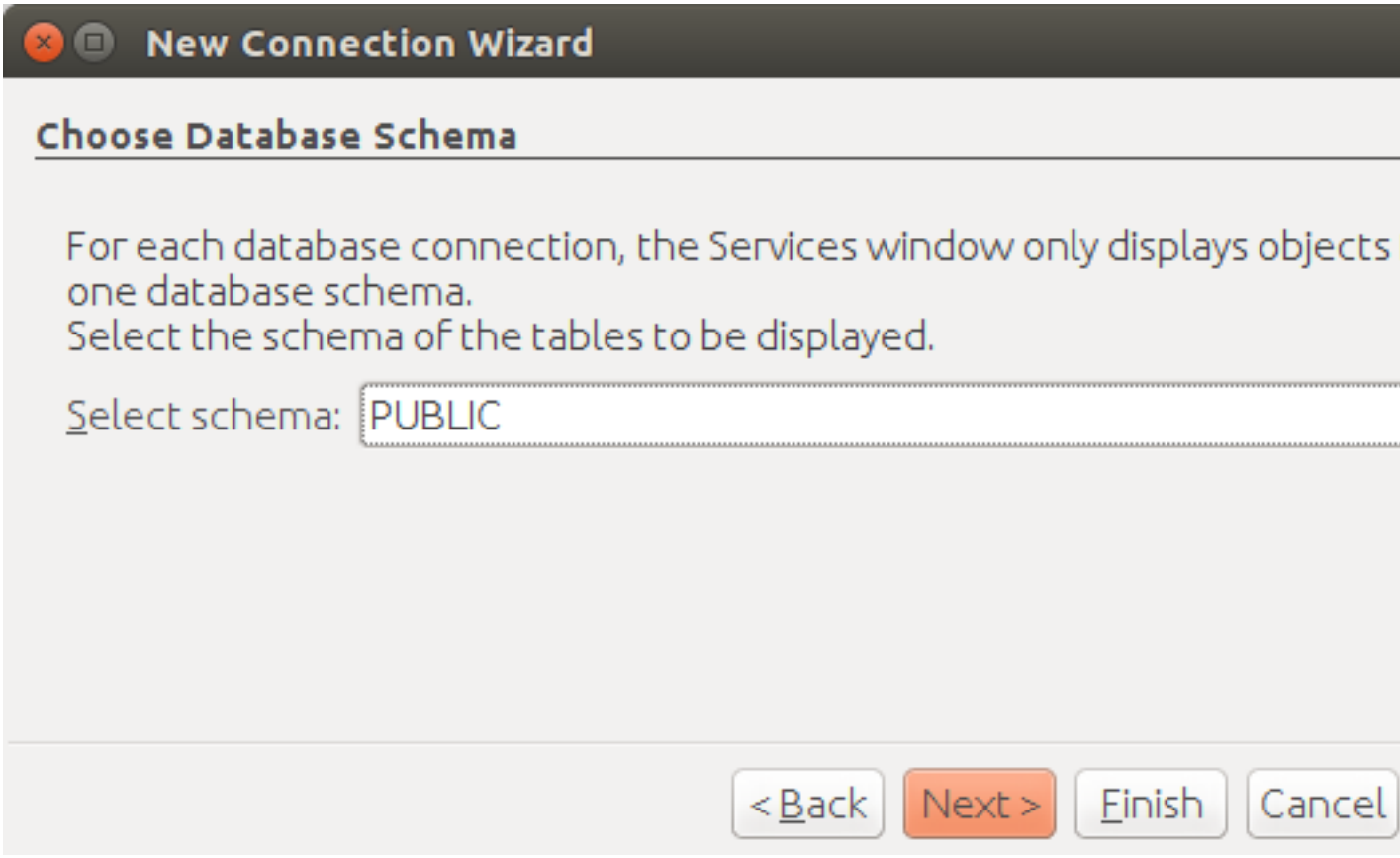
Select the H2 JDBC driver, provide a user name, a password and a JDBC URL in the following format:

Figure 4.10. Connection properties



Select `PUBLIC` as the connection's default schema.

Figure 4.11. Schema type



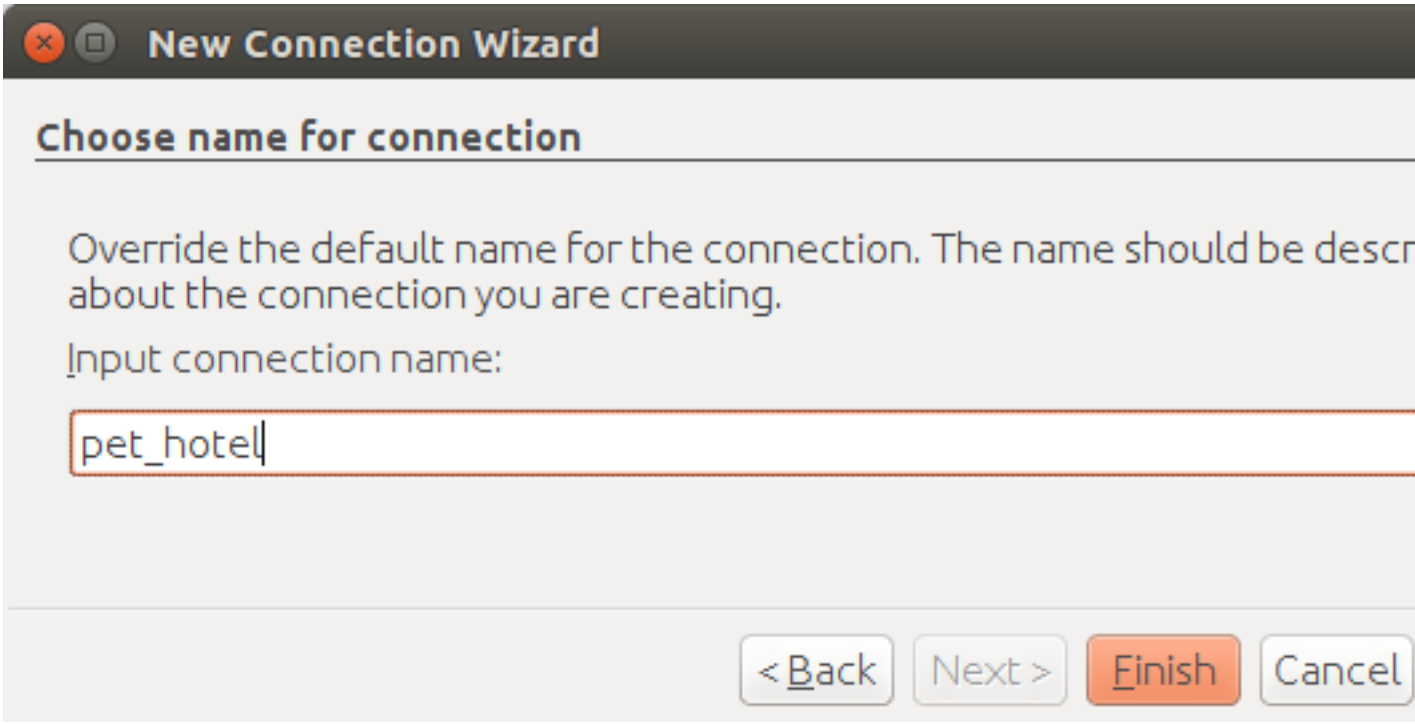
Click Next button. H2 database will be started and `pet_hotel` database will be created in the user's directory if it is not exist yet.

Caution

Connection name should be named by the JavaScript variables rules.

Set `pet_hotel` as a connection name.

Figure 4.12. Connection name



Create a new project for Pet Hotel application and provide the project's name and home directory. In the project's properties select the `pet_hotel` from the list as a default Datasource.

Chapter 5. Defining the database structure

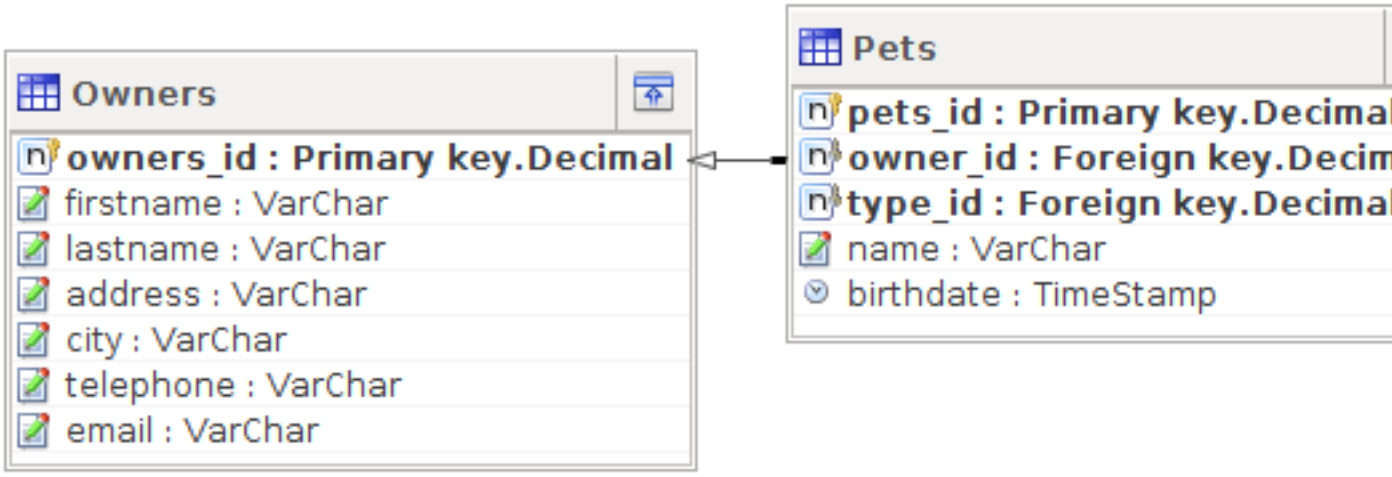
One way to begin building your application is to start from creating its database structure. When using Platypus.js, you need to create a database structure diagram.

Add a new Database structure diagram application element. *File # New File # Database structure diagram*.

On the diagram create new database tables named `Owner`, `Pet`, `PetType` and `Visit` according to the knowledge domain model. A numeric primary key is automatically created for each new table. Add all the required fields for the tables.

Create the foreign key links by connecting foreign key fields with the correspondent primary keys files. Please note that the connected fields must have the same data type.

Figure 5.1. Database structure



Chapter 6. Owners list form

We are going to build the user interface allowing to display the owners list.

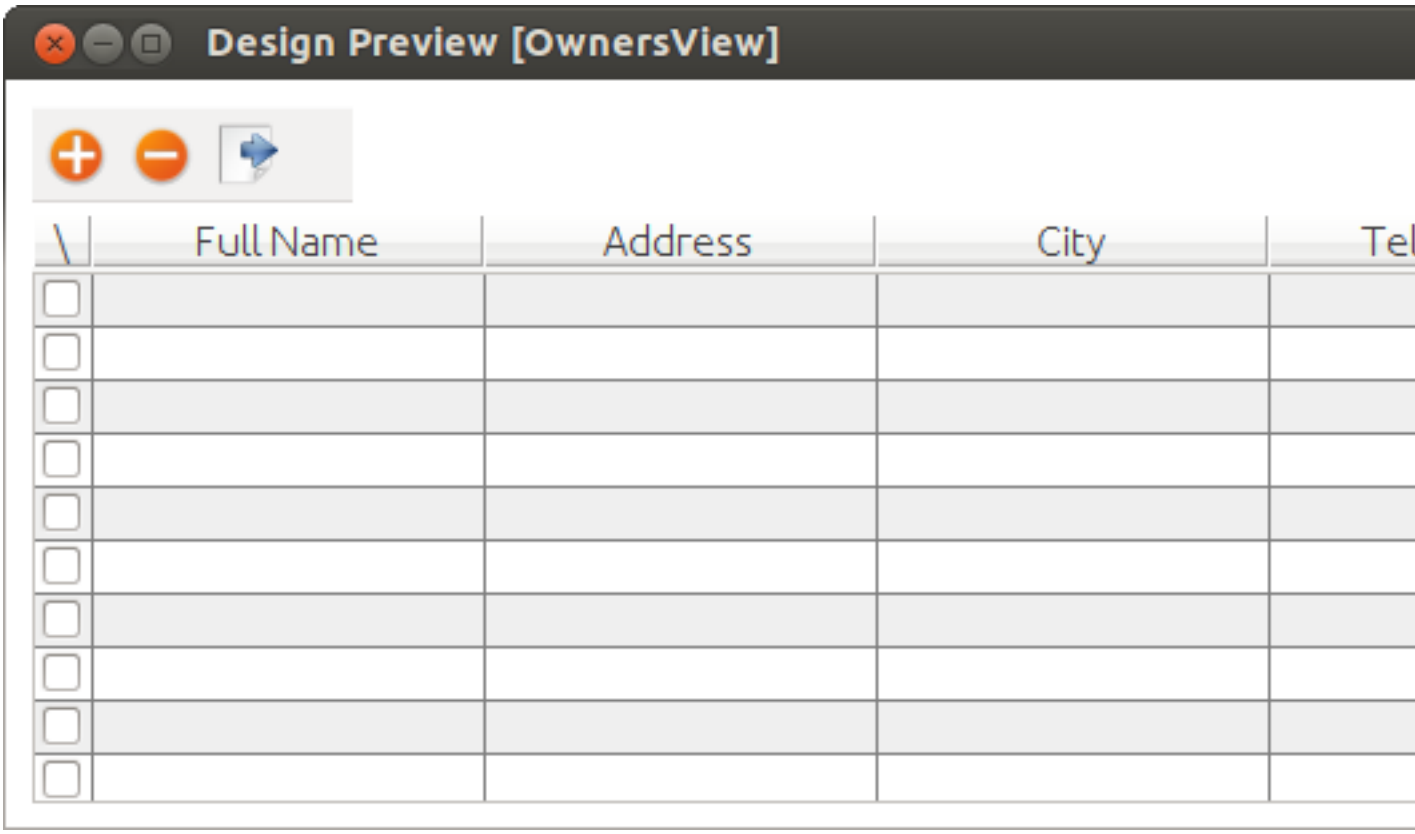
Create a new Form application element named `OwnersView`, check that the JavaScript constructor for this form is also set to `OwnersView`. This form will display the owners list.

Also create a new Form application element named `OwnerView` , check that the constructor is also set to `OwnerView`. The owners details will be shown on this form. Save this form but for this moment leave it blank.

Now lets edit the `OwnersView` form. `OwnersView` form will contain (Fig. owners view):

- On the top of the form: the panel with the Add and Delete buttons as well as the search text field and the Search button.
- The `ModelGrid` widget to display the owners list.

Figure 6.1. Owners view



Using *Palette* tool add elements to panel by drag-drop. Add the header panel from the containers palette on the form, put the buttons and the text field from the standard components palette on the panel. Provide meaningful names for the added components. Set texts to the added buttons. Drag-and-drop a `ModelGrid` from the model widgets palette on the form below the header panel and also provide a name for it.

Next lets configure the data model for our `OwnersView` form. Data model allows persistent data to be read and written from/to the database. In Platypus.js data model entities are created on the basis of data sources. To access relational data create data sources from SQL queries.

Create a new Query application element named `OwnersQuery` with SQL to get filtered records from the `Owner` table:

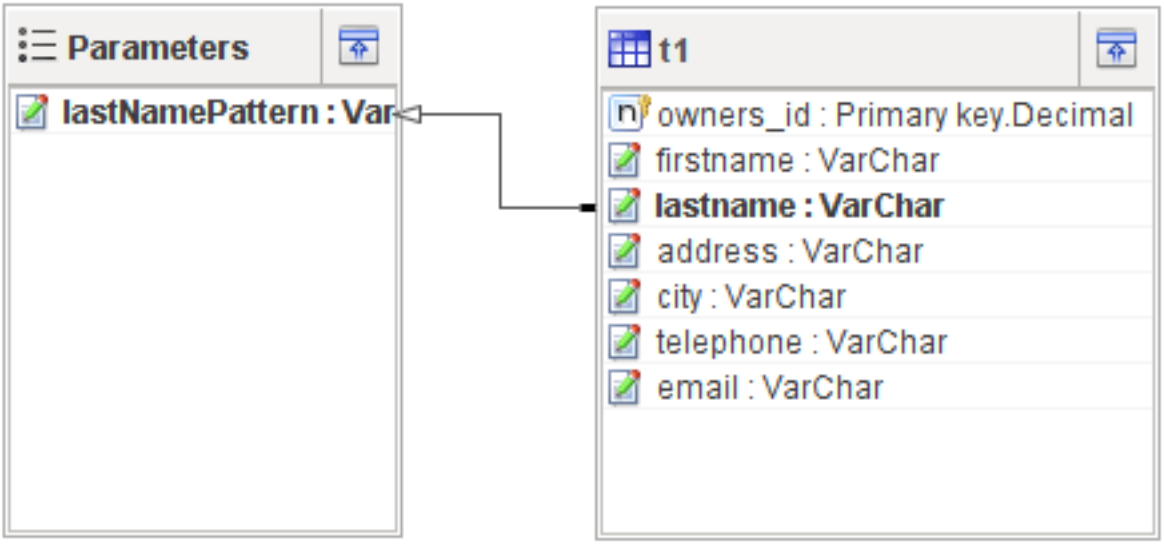
```
/**
 * @name OwnersQuery
 * @public
 */
Select t1.OWNERS_ID, (t1.FIRSTNAME || ' ' || t1.LASTNAME) AS fullName, t1.ADDRESS AS add
, t1.CITY AS city, t1.TELEPHONE AS phone, t1.email AS email
From OWNERS t1
 Where t1.LASTNAME Like :lastNamePattern
```

In this SQL query we are concatenating the `firstname` and `lastname` fields to return an owner's full name. Use the `:lastNamePattern` to provide a search pattern for the owner's last name. Adding alias to the fields allows us to use ORM (Object Relation Mapper) on any database in future.

Add `@public` annotation to the query's header to enable access via network from a remote data model running on a client and save the query.

Drag-and-drop it to the `OwnersView` data model. You may also go to the new entity's properties and provide its name, for example `owners`.

Figure 6.2. Connection name



Next, bind the `ModelGrid` widget to the `owners` entity as it shown in figure grid binding. Select the Model binding data parameter and select the model entity to bind. Create the grid's columns using *Fill columns* context menu item. After that provide the meaningful columns names and correct the columns captions (Fig. Columns settings).

Figure 6.3. Binding data model to grid

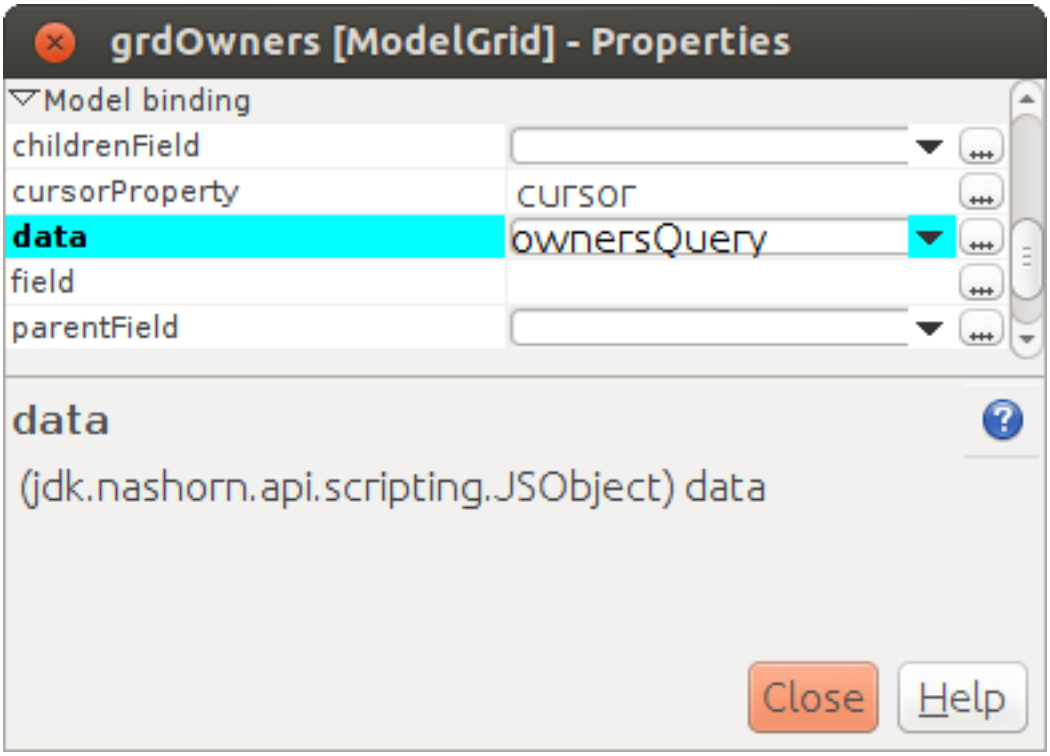
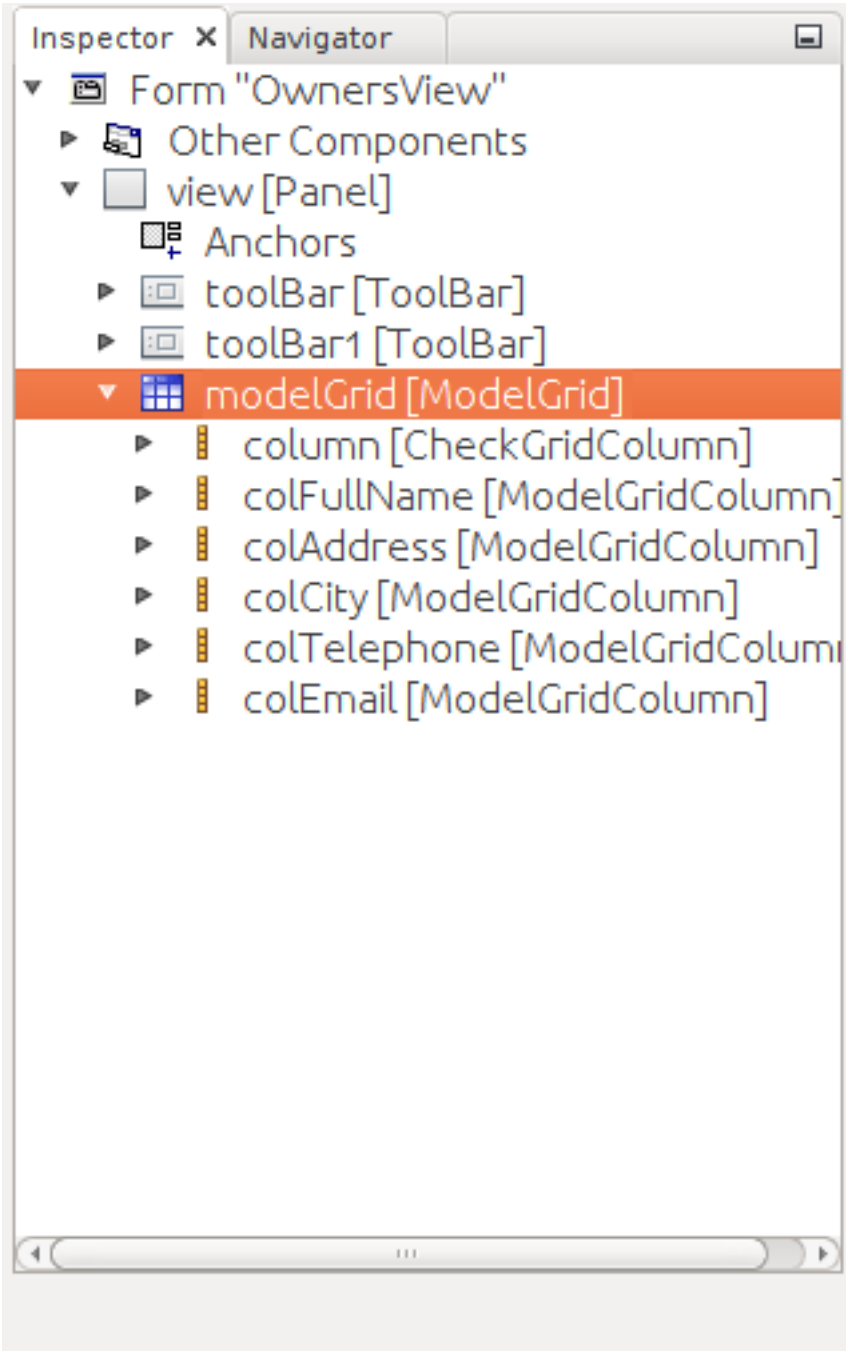


Figure 6.4. Setting grid columns



`ModelGrid` widget enables rows insertions and deletions as well as editing of its its cells. The changes will be made in the bounded data model entity. This way we can create a simple CRUD functionality even without any coding. For our grid we disable this feature, because we are going to use a separate form to edit a single owner's record — disable deletable, insertable and editable properties of the grid in it's properties menu.

Lets write some JavaScript code for our form.

By default after creation form's code looks like this:

```
function OwnersView() {
    var self = this
            , model = P.loadModel(this.constructor.name)
```

```
            , form = P.loadForm(this.constructor.name, model);

    self.show = function () {
        form.show();
    };

    // TODO : place your code here

    model.requery(function () {
        // TODO : place your code here
    });
}
```

Double click on the Add button and enter the code responsible for showing the `OwnerView` form:

```
/**
 * Add button's click event handler.
 * @param event Event object
 */
form.btnAdd.onActionPerformed = function (event) {
   var ownerView = new OwnerView();
   ownerView.showModal(refresh);
};
```

In this event handler we create a new instance of the owner's details form and show it as a modal window. We will create showModal method in detailed OvnerView later. We provide the `refresh` function as a parameter to enable data model requery when closing the owner's details form:

```
function refresh() {
    model.requery();
}
```

Double click on the Delete button and provide the code fragment responsible for an owner's record deletion:

```
/**
 * Delete button's click event handler.
 * @param event Event object
 */
form.btnDelete.onActionPerformed = function (event) {
        if (confirm("Delete owner?")) {
            for (var i in form.modelGrid.selected) {
                model.owners.splice(model.owners.indexOf(form.modelGrid.selected[i]), 1)
            }
       model.save();
   }
};
```

On Delete button click we are showing a confirmation dialogue and if the action is confirmed the current row in the owners query will be deleted. Then all changes will be saved to the database. Data model is JavaScript array, so we use method *splice* to delete selected rows. Information about selected rows we get from modeGrid.

Provide a handler for the `onMouseClicked` event of the grid widget:

```
/**
 * Grid click event handler.
 * @param event Event object
```

```
 */
form.modelGrid.onMouseClicked = function (event) {
        if (event.clickCount > 1) {
            var ownerView = new OwnerView();
            ownerView.showModal(refresh, model.owners.cursor.OWNERS_ID);
        }
    };
```

The code is seems familiar except the handling of the `ownerID` parameter containing the grid's current owner's record identifier.

Double click on the Search button to provide the search by a last name action logic:

```
/**
 * Search button click event handler.
 * @param event Event object
 */
form.btnSearch.onActionPerformed = function (event) {
    var searchText = "%" + form.txtSearch.text + "%";
    model.owners.params.lastNamePattern = searchText;
    model.owners.requery();
};
```

When a new value is assigned to a model's parameter the model's data linked to this parameter is automatically required according to the new value.

At this point we are ready to run and debug our application. Some test data can be added to the database tables using our SQL query. When a query is run the result are shown in a separate results window. You can also insert, delete and update database records using this window.

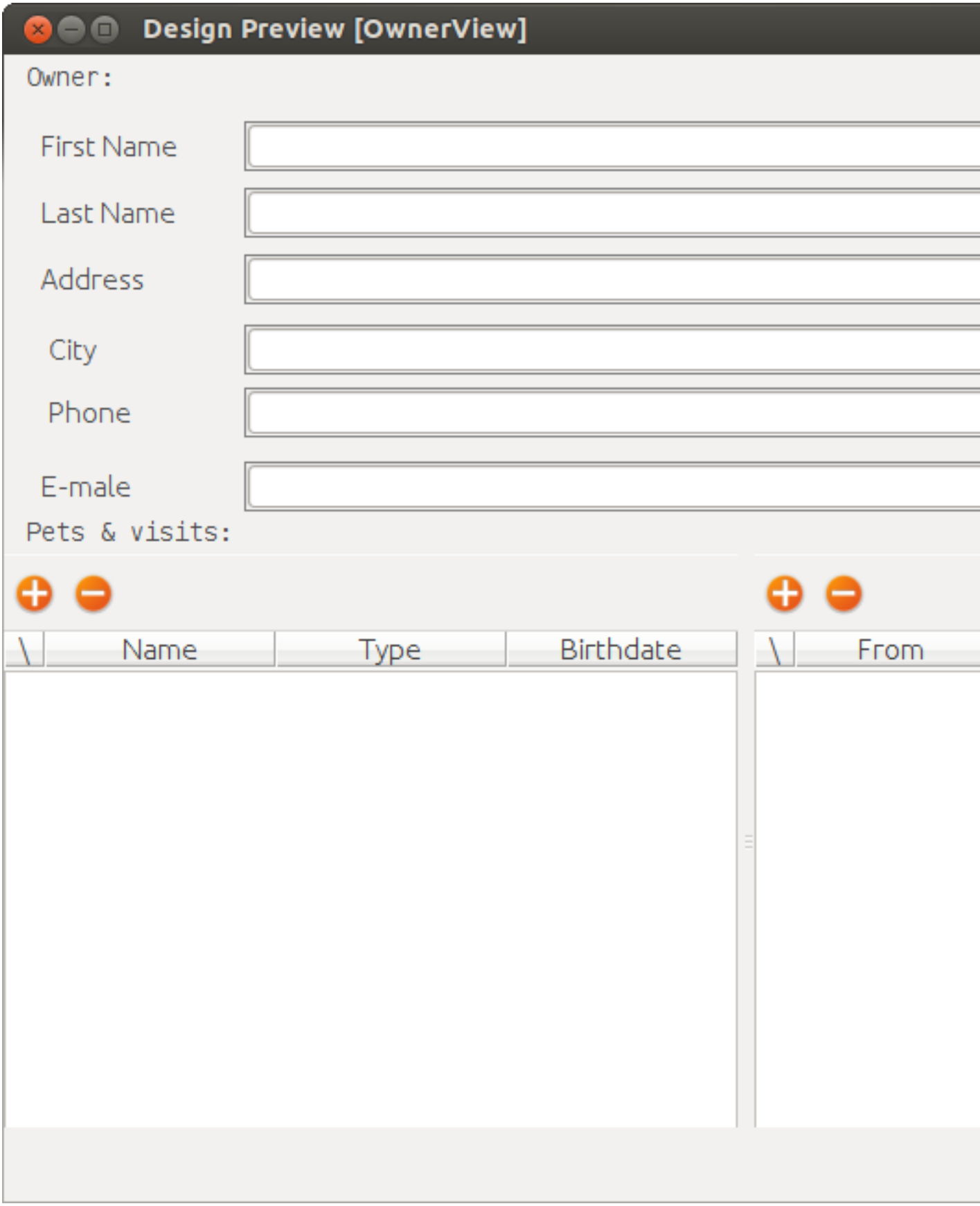
To obtain all data on form load we will add following code to method *show*:

```
self.show = function () {
       form.show();
       var searchText = "%%";
       model.owners.params.lastNamePattern = searchText;
       model.owners.requery();
   };
```

Chapter 7. Owners details, pets and visits form

Open the OwnerDetails form we’ve created earlier. This form will contain the user interface related to a concrete owner, her/his pets and hotel visits.

Figure 7.1. Owner detail



Add the Name, Last Name, Address, City, Phone and Email model `TextField` widgets for an owner's fields. Align this components to the right. Add `Label` components to the left of the correspondent input text field. Provide meaningful names for all components and set the labels texts.

Drag-and-drop a `SplitPane` container from the containers palette and set its separator orientation to vertical.

Add a panel container on the left and right sides of the `SplitPane`. The left panel is for an owner's pets and the right side is for the pet's visit to the hotel.

Place the Add and Delete buttons on top of the pets and the visits panels.

Add `ModelGrid` widgets on the left and the right panels to display pets and the concrete pet's visits list.

At the bottom of the form add Ok and Cancel buttons to save an owner's data, as well as the pets and the pet's visits data.

At this moment we have our owner's details form mock layout. Next we will configure the form's data model based on the SQL queries and write some JavaScript code.

Add a new application element for a SQL query selecting data for the specific owner by her/his identifier:

```
/**
 * Gets the owner by its ID.
 * @public
 * @name OwnerQuery
 */
Select *
From Owners t1
 Where :ownerID = t1.owner_id
```

Add a query for the pets list for the specific owner:

```
/**
 * Gets the pets for concrete owner.
 * @public
 * @name PetsQuery
 */
Select *
From Pets t1
 Where :ownerID = t1.owner
```

Next, add a query for getting all the hotel visits for the all pets of the specific owner:

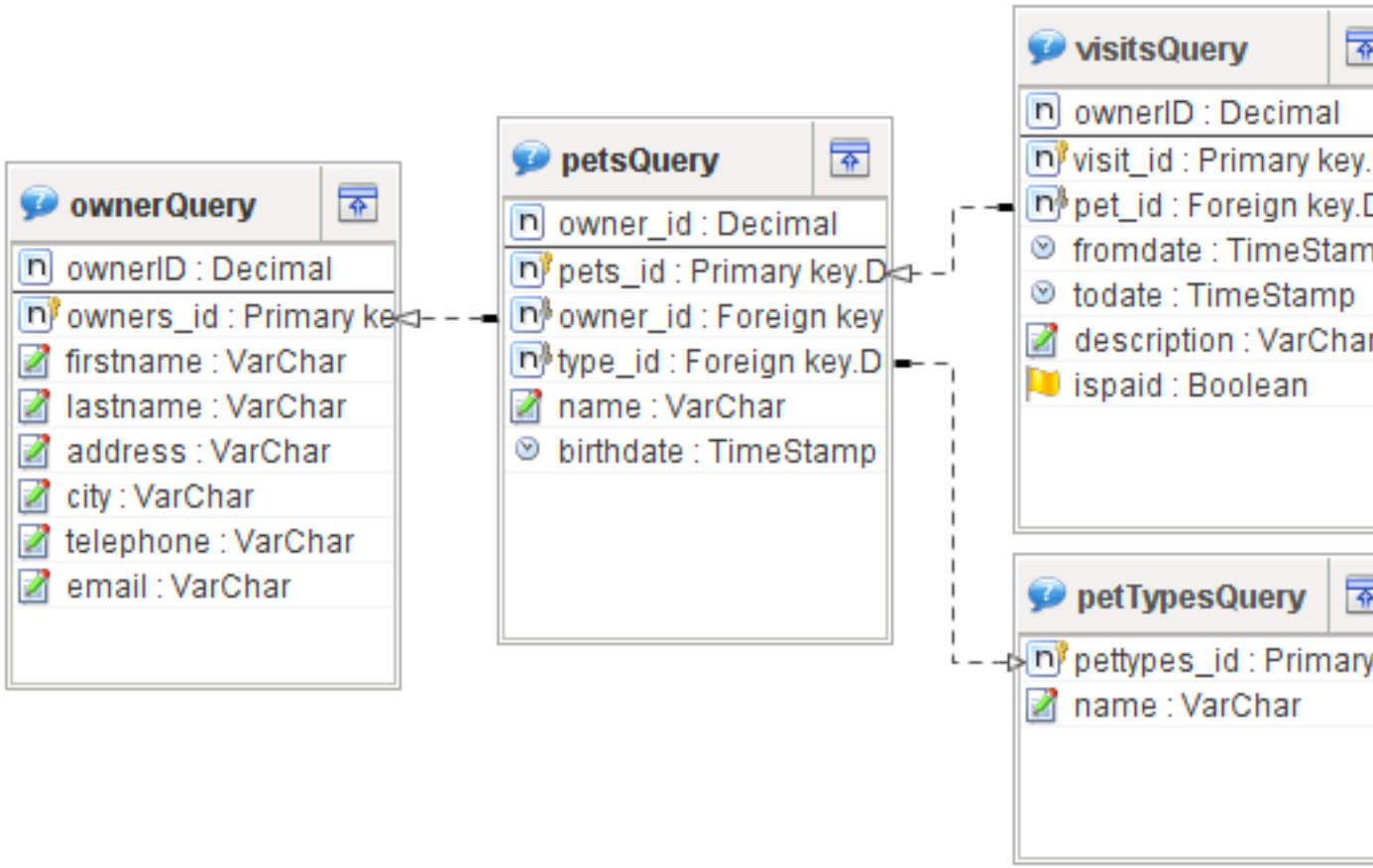
```
/**
 * Gets all visits for concrete owner.
 * @public
 * @name VisitsQuery
 */
Select t1.visit_id, t1.pet, t1.fromdate,
 t1.todate, t1.description
From Visit t1
 Inner Join PetsQuery t2 on t1.pet = t2.pet_id
```

Add a simple query for selecting all pets types:

```
/**
 * Gets all types for pets.
 * @public
 * @name PetTypesQuery
 */
Select * From PetType
```

In data model of *ownerView* form, add this four queries so that our model will looks like shown in fig. owner view data model

Figure 7.2. Created datamodel



Chapter 8. Scalar and collection properties

We need to create some scalar and collection properties for our *OwnerView* datamodel. You should select link between *petsQuery* and *petTypesQuery* and perform right mouse button click. In properties window (Fig. pets Collection set name for Scalar property name and Collection property name. Perform same task on connection between *petsQuery* and *visitsQuery* (Fig. visits collection).

Figure 8.1. Pets collection

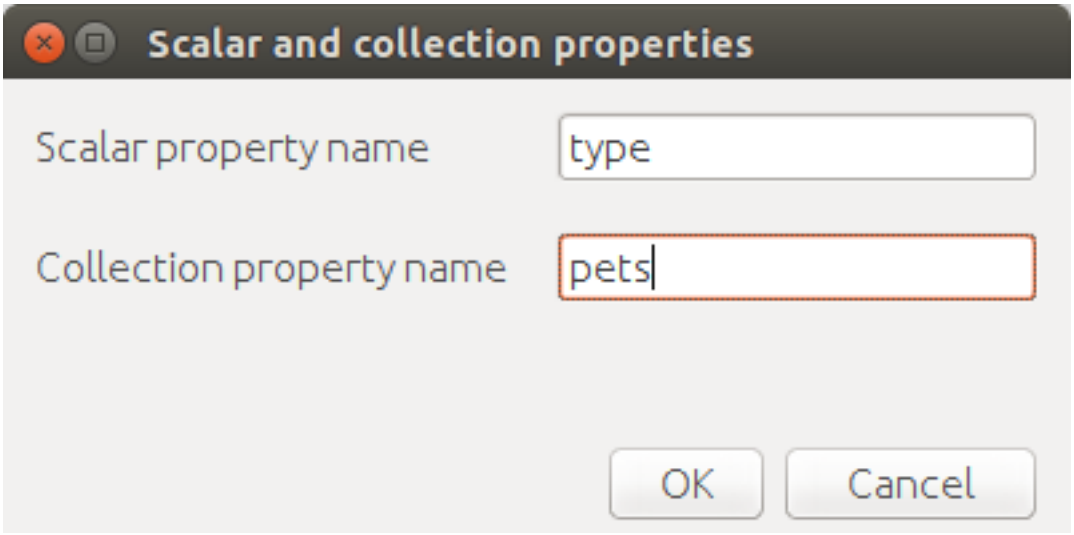
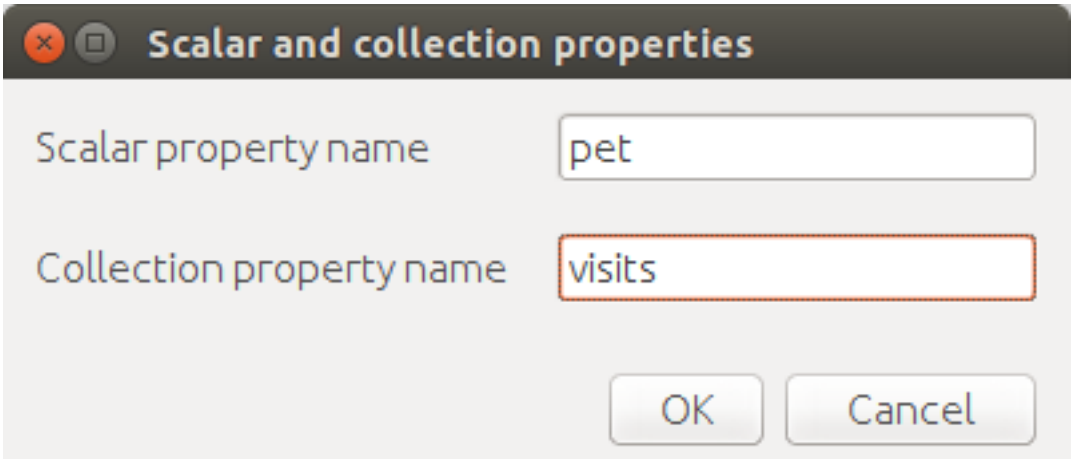
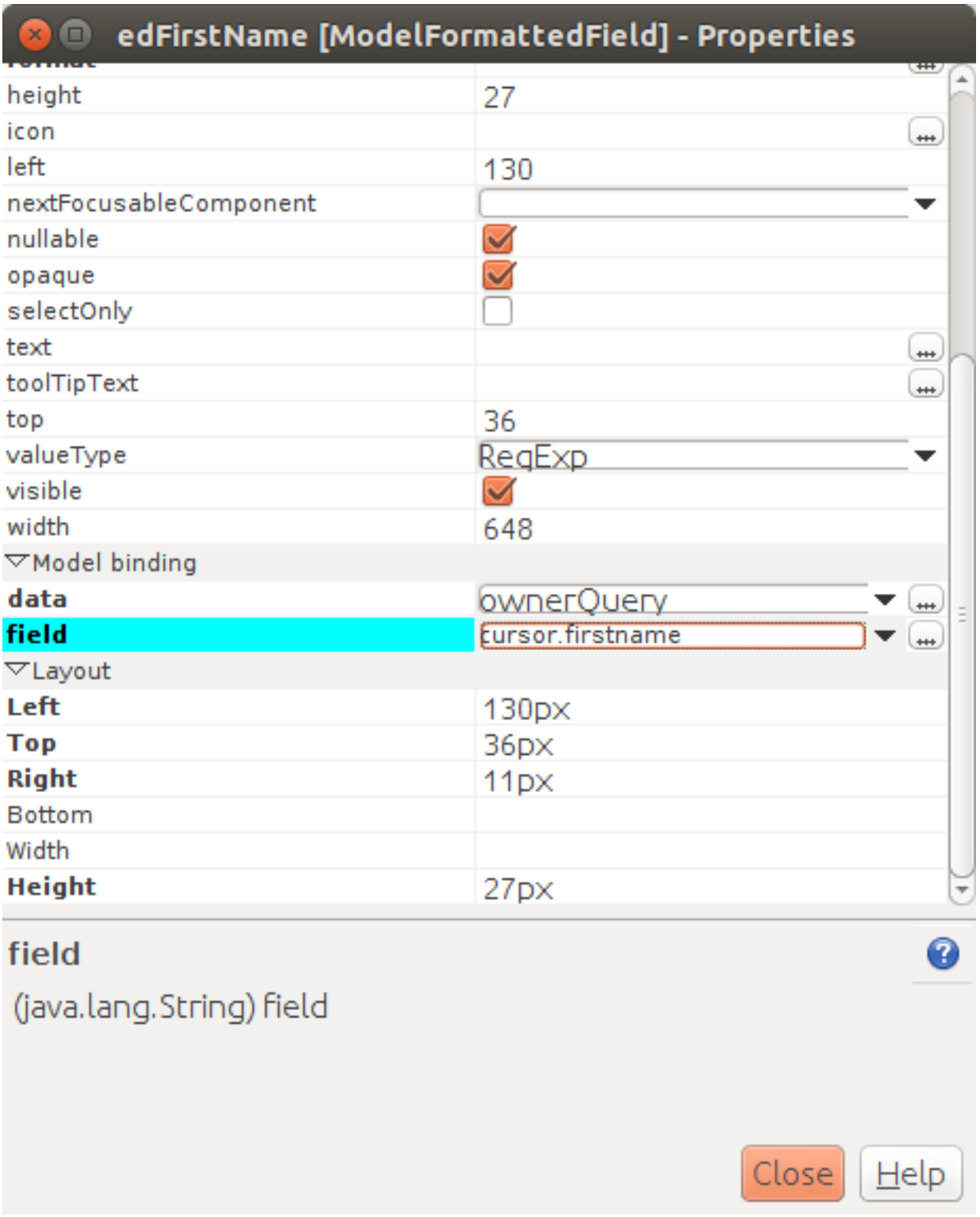


Figure 8.2. Visits collection



As the form's data model configuration is completed, bind the form's model widget to the model. Set the Model binding field property for the ModelText widgets on the form and bind them to the name, last name, city and telephone fields of the `ownerQuery` entity and set field's property as corresponding data field;

Figure 8.3. Field binding



The `visits` entity will hold all the visits for the all pets of the concrete owner, but we want to show on the right grid only the visits for the currently selected pet. To solve this issue we will use our collections, and create master-detail view.

In Pet's grid bind data to corresponding query (*petsQuery*) and use *Fill columns* context menu to create columns. Using inspector (as it was shown earlier in Fig. Inspector) delete unnecessary columns with id's and delete service column. Provide the correct text for the columns headers. Add Check grid

column, so the user could select multiple pets. Unlike the owners list grid the pets and visits grids will allow edit their cell data.

Provide the ModelCombo widget as a cell component for the *petType* column on the pets grid (Fig. Combo View). Set scalar property name, that we have defined earlier (pets collection) *type* to this column. For *ModelCombo* set *displayField* property to *name* and *displayList* to *petTypesQuery* (Fig. combo properties).

Figure 8.4. Combo view

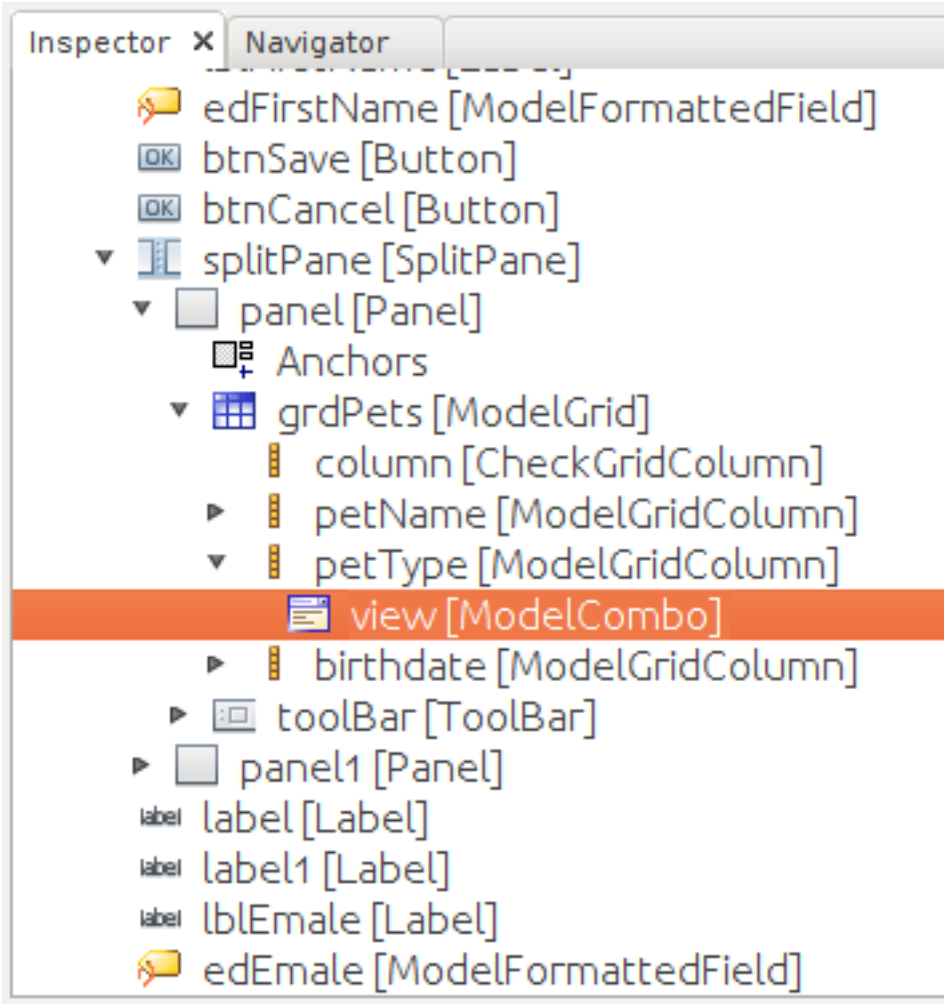
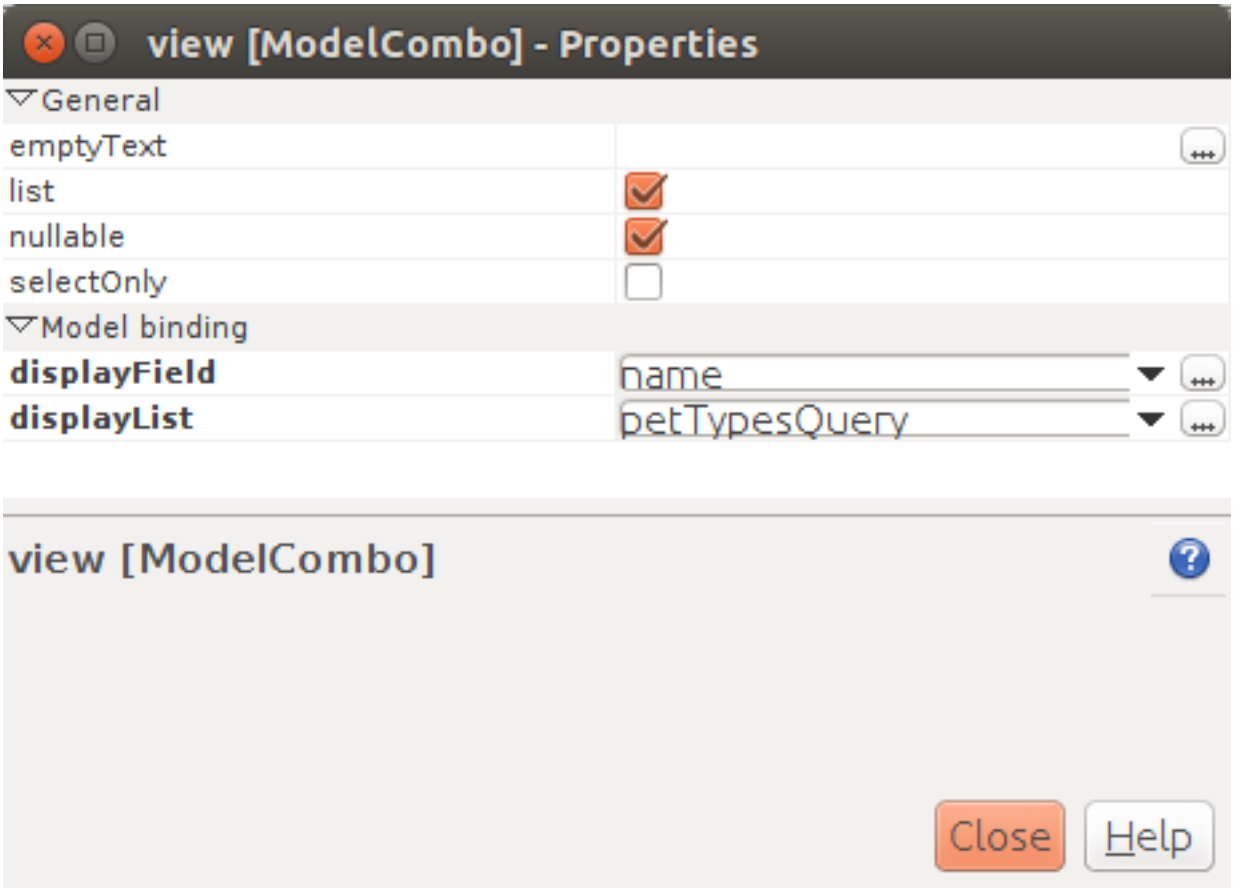
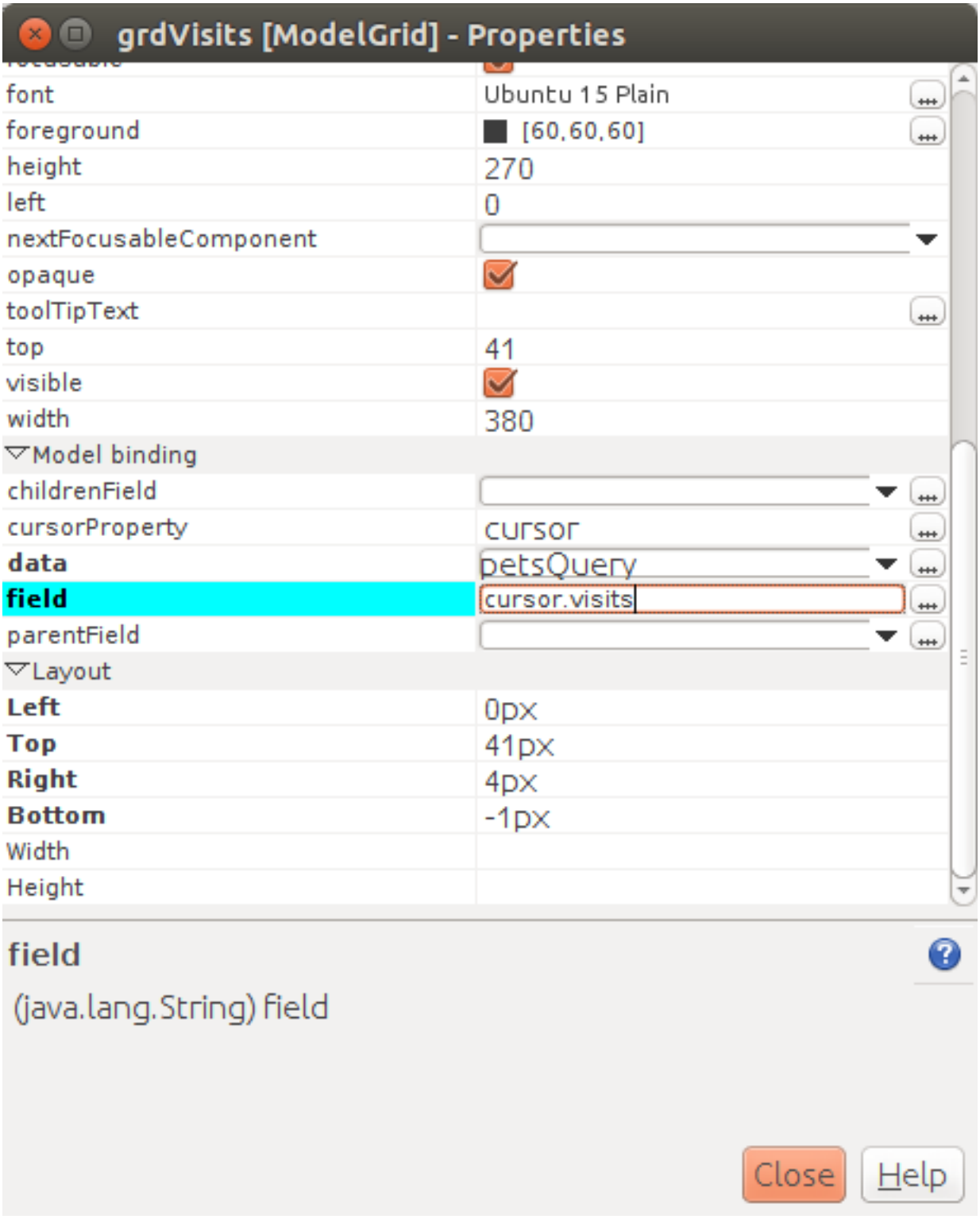


Figure 8.5. Combo view properties



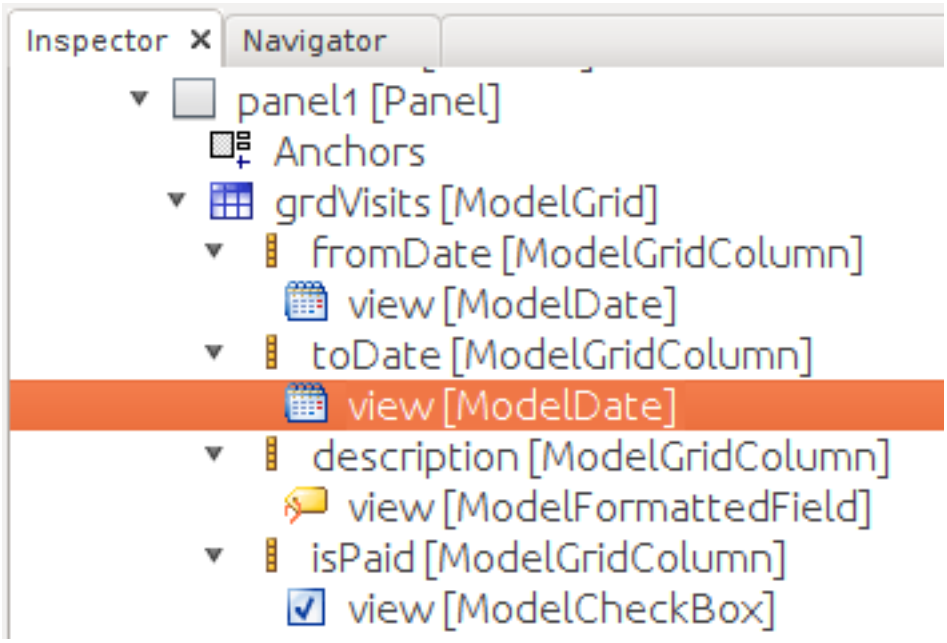
Master-detail view is creating by using two model grids, on same form. Master - is our pets, detail - pet's visit. We should set grid properties for visits. Set data field - *petsQuery*, but field must be set as *cursor.visits*. This collection we have also defined earlier (visits collection) ORM of *Platypus.JS* will automatically return certain collection for certain pet (Fig. visit grid properties).

Figure 8.6. Visit grid properties



Add model grid columns and set they field's as corresponding names of *visitsQuery* fields. Set presentation type in inspector as it shown in Fig. visit grid column properties.

Figure 8.7. Visit grid columns view



At the next step we'll write some JavaScript code for the OwnerView form.

```
self.showModal = function (aCallback, aID) {
    callback = aCallback;
    if (aID) {
        model.ownerQuery.params.ownerID = aID;
        model.requery();
    } else {
        model.ownerQuery.push({});
    }
    form.showModal();
};
```

Double click on the OK button and insert the handler code to save the owner's data:

```
form.btnSave.onActionPerformed = function (event) {
        if (model.modified) {
            var message = validate();
            if (!message) {
                model.save(function () {
                    callback();
                }, function () {
                    P.Logger.Info("Failed on save");
                });
                form.close();
            } else {
                alert(message);
            }
        } else {
            form.close();
        }
    };
}
```

In the handler code snippet above validation function is invoked and if successful then changes are saved to the database. Write the `validate` function stub we'll return to its code later.

```
/**
 * Validates the view.
 * @return Validation error message or false value if form is valid
 */
function validate() {
    var message = validateOwner();
    message += validatePets();
    message += validateVisits();
    return message;
}
```

Double click on the Cancel button and insert JavaScript code to perform the form close action:

```
/**
 * Cancel button's click event handler.
 * @param event Event object
 */
form.cancelButton.onActionPerformed = function(event) {
    form.close();
}
```

The event handler above will be invoked on form initialization.

Now it is time to add the code for the pets and their visits management.

Insert pets Add button `onActionPerformed` event handler to add a new pet:

```
/**
 * The add pet button's click event handler.
 * @param evt Event object
 */
form.btnAddPet.onActionPerformed = function (event) {
    model.petsQuery.push({});
};
```

Insert pets Delete button `onActionPerformed` event handler to delete a pet:

```
/**
 * Delete pet button's click event handler.
 * Deletes the selected pets.
 * @param evt Event object
 */
form.btnDeletePet.onActionPerformed = function (event) {
    if (confirm("Delete selected pets?")) {
        for (var i in form.grdPets.selected) {
            model.petsQuery.splice(model.petsQuery.indexOf(form.grdPets.selected[i]), 1)
        }
        model.save();
    }
};
```

Insert visits Add button `onActionPerformed` event handler to add a new visit to the hotel:

```
/**
 * Add visit button's click event handler.
 * @param evt Event object
 */
form.btnAddVisit.onActionPerformed = function (event) {
    model.visitsQuery.push({});
```

```
    model.visitsQuery.cursor.FROMDATE = new Date();
};
```

Insert visits Delete button `onActionPerformed` event handler to delete a pet's visit:

```
/**
 * Delete visit button's click event handler.
 * @param evt Event object
 */
form.btnDeleteVisit.onActionPerformed = function (event) {
    if (confirm("Delete selected visits?")) {
        for (var i in form.grdVisits.selected) {
            model.visitsQuery.splice(model.visitsQuery.indexOf(form.grdVisits.selected[i
        }
        model.save();
    }
};
```

Next we will provide the logic for the form validation. Edit the `validate` function and implement it as follows to perform the owner's and the pets and visits validation:

```
/**
 * Validates the view.
 * @return Validation error message or empty String if form is valid
 */
function validate() {
    var message = validateOwner();
    message += validatePets();
    message += validateVisits();
    return message;
}
```

Add owner's fields validation code:

```
/**
 * Validates owner's properties.
 * @return Validation error message or empty String if form is valid
 */
function validateOwner() {
    var message = "";
    if (!form.edFirstName.value) {
        message += "First name is required.\n";
    }
    if (!form.edLastName.value) {
        message += "Last name is required.\n";
    }
    if (!form.edAddress.value) {
        message += "Address is required.\n";
    }
    if (!form.edCity.value) {
        message += "City is required.\n";
    }
    if (!form.edPhone.value) {
        message += "Phone number is required.\n";
    }
    if (!form.edEmale.value) {
        message += "E-Mail is required.\n";
    }
    return message;
```

```
}
```

The pets validation code is as follows:

```
/**
 * Validates pets entity.
 * @return Validation error message or empty String if form is valid
 */
function validatePets() {
    var message = "";
    pets.forEach(function(pet) {
        if (!pet.name) {
            message += "Pet's name is required.\n";
        }
        if (!pet.birthdate) {
            message += "Pet's birthdate is required.\n";
        }
        if (!pet.type) {
            message += "Pet's type is required.\n";
        }
    });
    return message;
}
```

Insert the visits validation code for the currently selected pet:

```
/**
 * Validates visits entity.
 * @return Validation error message or empty String if form is valid
 */
function validateVisits() {
    var message = "";
    form.grdVisits.data.forEach(function (visit) {
        if (!visit.fromdate) {
            message += "Visit from date is required.\n";
        }
        if (!visit.todate) {
            message += "Visit to date is required.\n";
        }
        if (visit.fromdate >= visit.todate) {
            message += "Visit 'from' date must be before 'to' date.\n";
        }
    });
    return message;
}
```

At this stage you need to run and test your application. To do that, run the application with desktop client and direct connection to the database. Use step-by-step code debugging to make sure your JavaScript works correctly.

Chapter 9. Owners report

In this section we are going to create a simple report about the owners.

Create a new Report application element with the `OwnersReport` name. Add `OwnersQuery` to data model.

```
self.execute = function (onSuccess, onFailure) {
    model.ownersQuery.params.lastNamePattern = "%%";
    model.requery(function () {
        var report = template.generateReport();
        report.show(); //| report.print(); | var savedTo = report.save(saveTo ?);
//            onSuccess(report);
    }, onFailure);

};
```

On layout tab click on *edit report template* to edit the report template. Provide the report's header, owners tables columns headers and the columns tags as it shown below:

Full Name	Address	City	Phone	E-mail
${model.ownersQuery.firstName} ${model.ownersQuery.lastName}	${address}	${city}	${model.ownersQuery.telephone}	${model.ownersQuery.email}

Go to the `OwnersView` form and add the Report button. Change the button name, the caption text and provide its press event handler code:

```
/**
 * Report button click event handler.
 * @param evt Event object
 */
form.btnReport.onActionPerformed = function (event) {
      var oReport = new OwnersReport();
      oReport.execute();
  };
```

Here we can create a new report instance, set its parameter to the similar parameter of the`OwnersView` form and display the report.

But this will work only in SE client. To make this work on HTML5 application, we should create new server module, and place code as shown below:

```
/**
 *
 * @constructor
 * @public
 */
function serverModule() {
    var self = this, model = P.loadModel(this.constructor.name);

    self.execute = function (reportSuccessCallback) {
        var oReport = new OwnersReport();
        oReport.execute(reportSuccessCallback);
    };
}
```

Because we are going to call our server module over network, we should add annotation @public (like in queries).

On the next step we need to modify report generation code, where we will return generated report to callback.

```
self.execute = function (onSuccess, onFailure) {
    model.ownersQuery.params.lastNamePattern = "%%";
    model.requery(function () {
        var report = template.generateReport();
        //report.show(); | report.print(); | var savedTo = report.save(saveTo ?);
        onSuccess(report);
    }, onFailure);
};
```

We also should rewrite code in Report button:

```
var reportCallback = function (report) {
    report.show();
};

form.btnReport.onActionPerformed = function (event) {
    var srvModule = new P.ServerModule("serverModule");
    srvModule.execute(reportCallback);
};
```

After report has been generated, it will be returned to client. If you are launching your's application as HTML5 client, then report will be downloaded by browser when it's method *show* is called, otherwise it will launch assosiated application (Excel for example).

Thanks for you attention.

Chapter 10. Revision History

0-0 8.08.2013 Vadim Vashkevich vv@altsoft.biz [mailto:vv@altsoft.biz] Maxim Lipnin ml@altsoft.biz [mailto:ml@altsoft.biz] Initial revision

0-0 29.04.2015 Eugeny Konstantinov jskonst@altsoft.biz [mailto:jskonst@altsoft.biz] Marat Gainullin mg@altsoft.biz [mailto:mg@altsoft.biz] Update doc to platypus version 5.0