Platypus Platform 3.0

Quick Start Tutorial

Platypus.js applications development

Platypus Platform 3.0 Quick Start Tutorial
Platypus.js applications development
Edition 0

A Platypus.js application step-by-step tutorial.

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

[1] https://fedorahosted.org/liberation-fonts/

> **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

`Mono-spaced Bold Italic` or ***`Proportional Bold Italic`***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **`ssh`** ***`username@domain.name`*** at a shell prompt. If the remote machine is **`example.com`** and your username on that machine is john, type **`ssh john@example.com`**.
>
> The **`mount -o remount`** ***`file-system`*** command remounts the named file system. For example, to remount the **`/home`** file system, the command is **`mount -o remount /home`**.
>
> To see the version of a currently installed package, use the **`rpm -q`** ***`package`*** command. It will return a result as follows: ***`package-version-release`***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **`mono-spaced roman`** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svnbooks_tests  Desktop1
  downloads      images  notes  scripts  svgs
  stuff  svnbooks_tests  Desktop1  downloads      images  notes
```

Source-code listings are also set in **`mono-spaced roman`** but add syntax highlighting as follows:

```
     /**
      * Gets the application server URL
      * @returns {string} a server web application URL
     */
    this.getServerUrl = function() {
        if (serverUrl)
            return serverUrl;
        else
            return "";
    };
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Feedback information

Please connect with us by email: platform@altsoft.biz or by phone: +7 4932 496063.

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 avaliable for download from the Platypus.js project's website.

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

1.1. Platypus.js installation

Java Runtime and JDK 7 are required for Platypus.js applications development.

Install the Platypus.js IDE, the Platyps.js runtime components and Apache Tomcat 7 to enable running a web client:

- Download the Platypus.js installer bundle from the project's website for your operation system.
- Run the installer.
- Follow the master's steps and install the required components.

1.2. Tha sample application requirements

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

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

An adminstrator 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 preceed the check-out date.

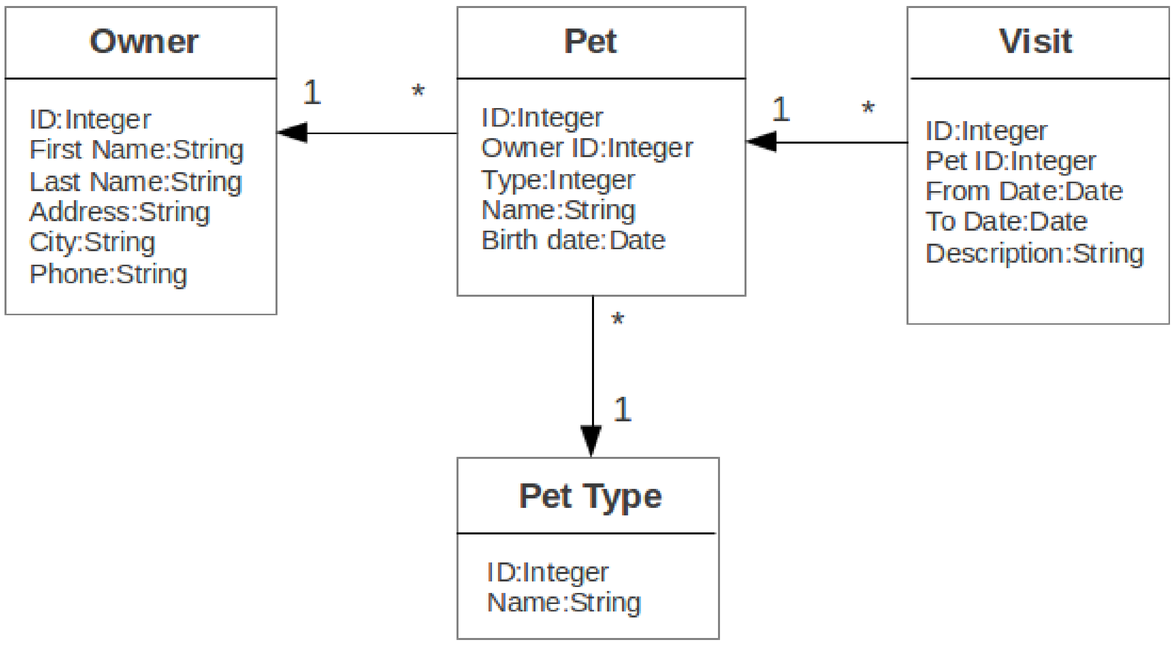

Figure 1.1. Pet Hotel knowledge domain model diagram

1.3. 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.

Create a new database connection for the Pet Hotel applicaton. You can use any of the databases supported by the platform. The easiest way is to use H2 database, wich 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.
- Select the H2 JDBC driver, provide a user name, a password and a JDBC URL in the following format: `jdbc:h2:tcp://localhost/~/pet_hotel`
- 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.
- Select **`PUBLIC`** as the connection's default schema.
- Set **`pet_hotel`** as a 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**.

1.4. Defining the datatase 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.

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

1.5. Owners list form

We are going to build the user inteface 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 detais will be shown on this form. Save this form but for this moment leave it blank.

Now lets edit the **`OwnersView`** form.

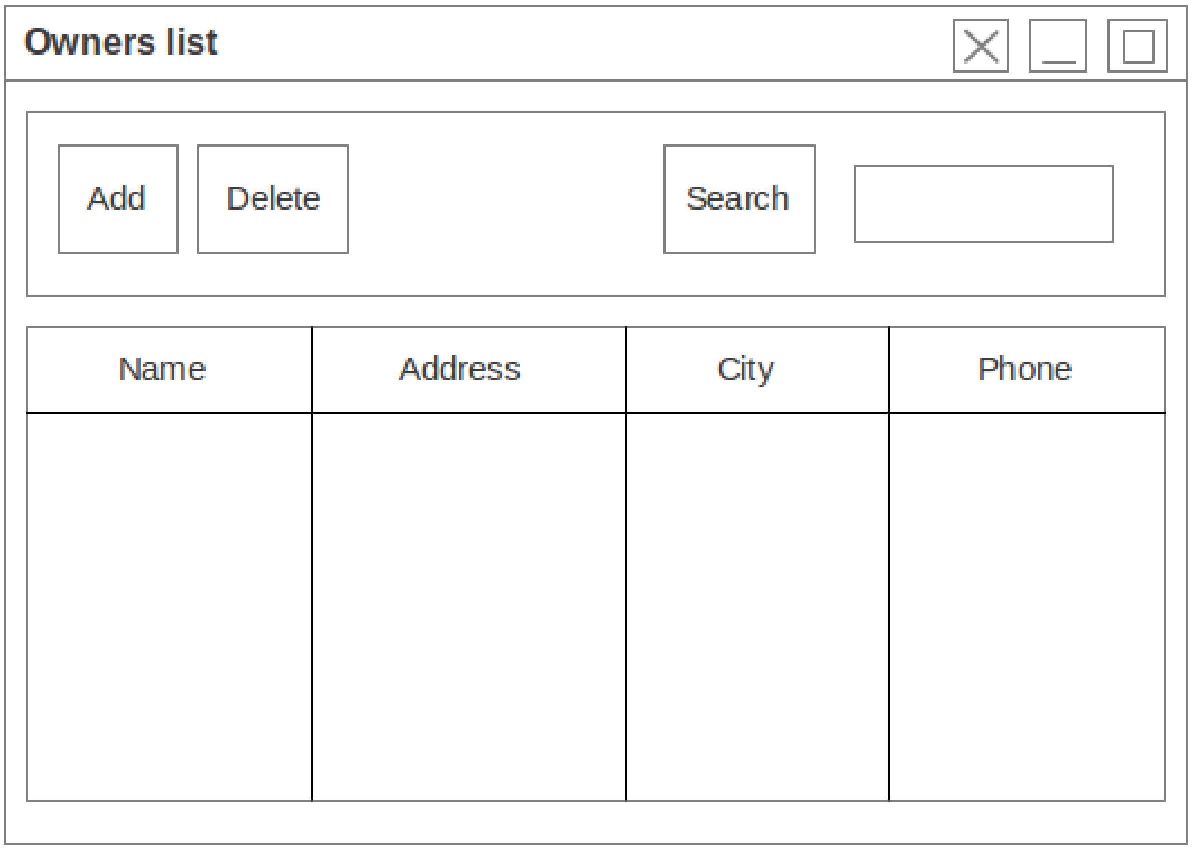

Figure 1.2. Owners list form layout

`OwnersView` form will contain:

- 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.

Add the header panel from the containers pallete 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 Playpus.js data model entities are created on the basis of data sourses. 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:

```
/**
 * Gets all owners.
 * @public
 * @name OwnersQuery
 */
Select t1.owners_id, (t1.firstname || ' ' || t1.lastname) As fullName, t1.address, t1.city,
 t1.telephone
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.

Add **@public** anntotation 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. Go to the new entity's properties and provide its name as **owners**.

Add the new data model parameter named **lastNamePattern** and connect it to the corrsespondent parameter of the entity. We'll use this parameter to perform search on the **Owner** database table.

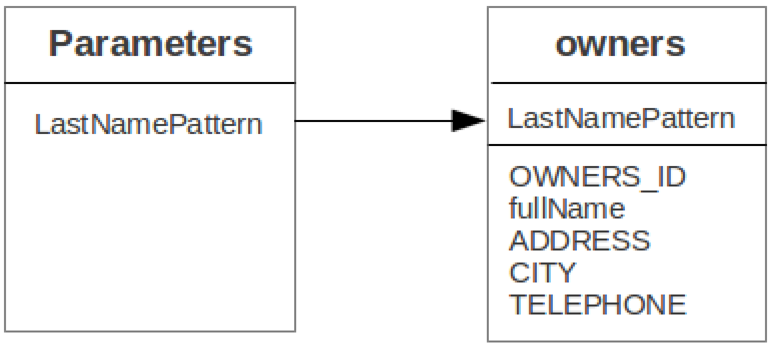

Figure 1.3. OwnersView form data model

Next, bind the **ModelGrid** widget to the **owners** entity. Select the **Model binding** → **entity** parameter and select the entiy 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.

ModelGrid widget enables rows insertions and deletions as well as editing of its its cells. The chandes will be made in the binded 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.

Lets write some JavaScript code for our form.

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

```
/**
 * Add button's click event handler.
 * @param evt Event object
 */
form.addButton.onActionPerformed = function(evt) {
    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 provide the **`refresh`** function as a parameter to enable data model requiery 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 evt Event object
 */
form.deleteButton.onActionPerformed = function(evt) {
    if (confirm("Delete owner?")) {
        ownersQuery.deleteRow();
        model.save();
    }
}
```

On **Delete** button click we are showng 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.

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

```
/**
 * Grid click event handler.
 * @param evt Event object
 */
form.ownersGrid.onMouseClicked = function(evt) {
    if (evt.clickCount > 1) {
        editOwner();
    }
}
```

Write the **`editOwner`** function:

```
function editOwner() {
er() { var ownerView = new OwnerView();
    ownerView.ownerID = owners.cursor.owners_id;
    ownerView.showModal(refresh);
}
```

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 evt Event object
 */
form.searchButton.onActionPerformed = function(evt) {
     model.params.lastNamePattern =
           '%' + txtSearch.text + '%';
}
```

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

1.6. Owners detalis, pets and visits form

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

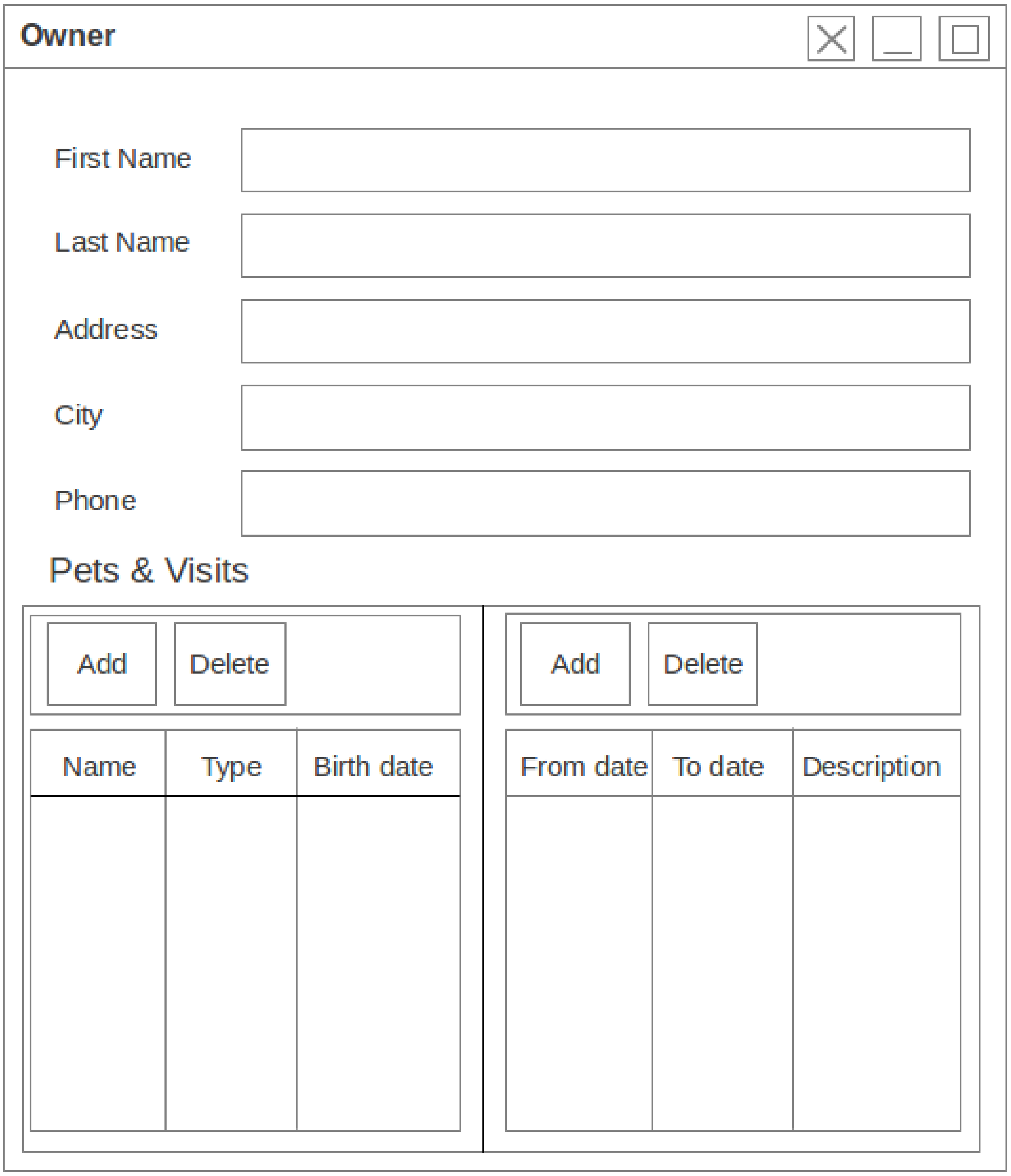

Figure 1.4. OwnerDetails form layout

Add the **Name**, **Last Name**, **Address**, **City** and **Phone** 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 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 conrete 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 concrete 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 concrete 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
```

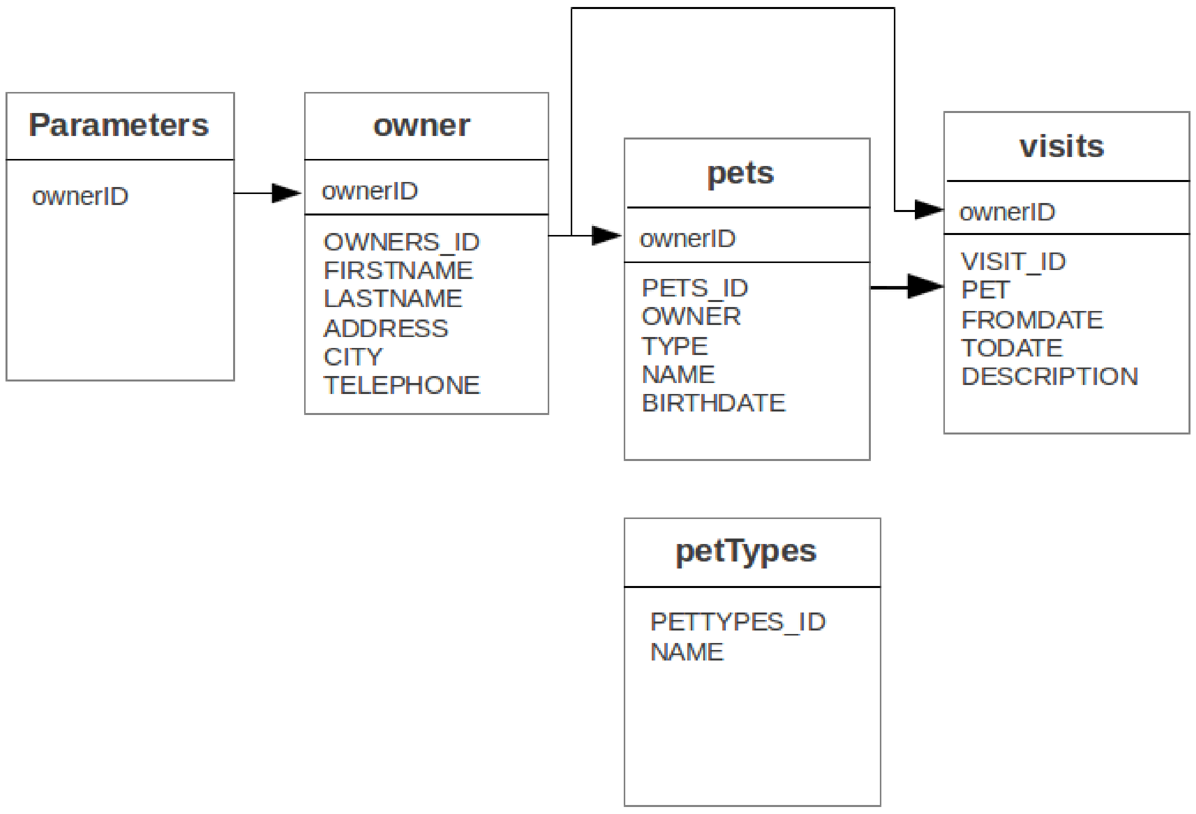

Figure 1.5. OwnerView form data model

Add a data model parameter **`ownerId`** and set its type to **`Decimal`**. Add new entities based on the **`OwnerQuery`**, **`PetsQuery`**, **`VisitsQuery`**, **`PetTypesQuery`** queries and set the correspondent entites names to **`owner`**, **`pets`**, **`visits`** и **`petTypes`**.

Connect the **`ownerID`** data model parameter to the **`ownerID`** parameter of the **`owner`** entity. In runtime this entity will contain the owner's data selected according to the parameter value. Notice that this entity will contain only one row.

Connect the input parameters of the **`pets`** and **`visits`** entities to the current owner's identifier.

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. For this, add a filtering link between the **`pet_id`** field of the **`pets`** entity and the **`pet`** field of the **`visits`** entity. Notice that filtering take place on a client and does not spawn any new database requests.

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 **`owner`** entity.

Create new columns for the **`pets`** and **`visits`** grids and bind this columns to the correspondent fields of the **`pets`** and **`visits`** entities. Provide the correct text for the columns headers.

Ulike 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 pet type column on the pets grid. For this component specify the **`displayField`** и **`valueField`** by connecting them to the **`name`** и **`pettype`** fiels of the **`petTypes`** entity.

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

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

```
/**
 * Save button's click event handler.
 * @param evt Event object
 */
form.okButton.onActionPerformed = function(evt) {
    if (model.modified) {
        var message = validate();
        if (!message) {
            model.save(function() {
                close(owner.owner_id);
            });
        } else {
            alert(message, title);
        }
    }
}
```

In the handler code snippet above validation function is invoked and if successfull 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 falsy value if form is valid
 */
function validate() {
    return null;
}
```

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

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

To ensure that a new owner row is inserted add the **onRequeried** event hanlder for the **`owner`** entity:

```
/**
 * Data model's OnRequired event handler.
 * @param evt Event object
 */
model.owner.onRequeried = function(evt) {
    if (!model.params.ownerId) {
        owner.insert();
    }
}
```

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.addPetButton.onActionPerformed = function(evt) {
    model.pets.insert();
    model.pets.cursor.owner = model.params.owner_id;
}
```

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

```
/**
 * Delete pet button's click event handler.
 * Deletes the selected pet.
 * @param evt Event object
 */
form.deletePetButton.onActionPerformed = function(evt) {
    if (confirm('Delete pet?', title)) {
        model.pets.deleteRow();
    }
}
```

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.addVisitButton.onActionPerformed = function(evt) {
    model.visits.insert();
}
```

Дважды кликните мышкой на кнопке **Delete** на панели визитов и добавьте код для удаления визита:

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.deleteVisitButton.onActionPerformed = function(evt) {
    if (confirm('Delete visit?', title)) {
        model.visits.deleteRow();
    }
}
```

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 (!owner.firstname) {
        message += "First name is required.\n";
    }
    if (!owner.lastname) {
        message += "Last name is required.\n";
    }
    if (!owner.address) {
        message += "Address is required.\n";
    }
    if (!owner.city) {
        message += "City is required.\n";
    }
    if (!owner.telephone) {
        message += "Phone number 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 = "";
    visits.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;
}
```

Please notice that the pet's visits validation must be invoked not only before the model save action, but also on any pets grid cursor movement. To do that, implement the **`willScroll`** event of the **`pets`** entity:

```
/**
 * Pet's entity cursor movement event handler.
 * @param evt Event object
 */
model.pets.onWillScroll = function(evt) {
    Logger.info('Pets scroll event.');
    var message = validateVisits();
    if (message) {
        alert(message);
        return false;
    }
    return true;
}
```

The cursor will not move if the **`onWillScroll`** event handle will return **`false`** value.

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 JavaSctipt works correctly.

By default the anonymous mode is enabled, but you can activate a user's login dialogue. For this set the correspondent checkbox in the application project properties. The user name **`admin`** with **`masterkey`** password are the default credentials you can use to login.

1.7. Improving the owners list form

In this section we are going to add a new column on the owners grid list to display pets names.

Open for edit the **`OwnersView`** form layout and drag-and-drop a new **Model Grid Column** widget on the owners list grid component. Set the column name and its header text. Do not connect this column with any of the entities fields. For this column we are going to use a separate SQL query to retrieve the pets list and output it using the specific JavaScript code.

Create a new query named **`OwnersPets`** to select the owners and their pets. Configure this query using the visual editor. Notice that in this query we use the **`OwnersQuery`** subquery. Add the **`OwnersQuery`** and the **`Pets`** table and connect the keys fields with a link. The result SQL is to be like the follows:

```
/**
 * Gets the owners and their pets.
 * @public
 * @name OwnersPets
 */
Select q1.owner_id, t.name
From OwnersQuery q1
 Left Outer Join Pets t on t.owner = q1.owner_id
```

Add the **`lastNamePattern`** parameter to the query and bind the parameter to the **`OwnersQuery`** subquery.

Please notice that a client's data model has asynchronous nature. Thus the queries execution results are delivered to the client asynchronously. In the case when we need to display the combined data on a single widget (a column cell in our application) we have to take care about it.

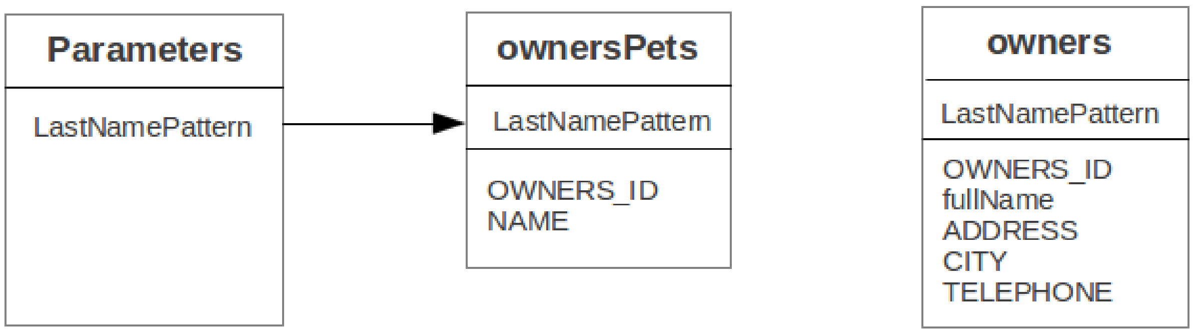

Figure 1.6. OwnersView form data model

Add the **`OwnersPet`** query to the **`OwnersView`** and set the new entity's name to **`ownersPets`**. Delete the existing link between the **`owners`** entity and the **`lastNamePattern`** parameter and bind this parameter to the correspondent parameter of the **`ownersPets`** entity. This way the parameter change will refresh the pets entity, not the **`owners`** entity.

The **`owners`** entity will be requiered by the code logic. This code will be invoked on the event of the getting new data in the **`ownersPets`** entity.

Add the **`onRequired`** event handler for the **`ownersPets`** and insert some code to its body:

```
/**
 * Called then data is ready in ownersPets entity.
 * @param evt Event object
 */
model.ownersPet.onRequeried = function(evt) {
    model.owners.params.lastNamePattern =
           model.ownersPets.params.lastNamePattern;
    owners.requery();
}
```

The owner's list data grid will be repained after the fresh data will come to the binded **`owners`** entity. At that moment we'll have for sure actual pets data because the correspondent query already have executed.

Provide the **`onRender`** event handler to the owners **`ModelGrid`** widget to perform the specific rendering for the new **`pets`** column:

```
/**
 * Pet's column onRender handler.
 * @param evt onRender event object
 * @returns true to apply changes to the cell
 */
form.ownersGrid.onRender = function(evt) {
    var pets = model.ownersPets.find(
              model.ownersPets.schema.owner_id,                            evt.id);
    var txt = '';
    pets.forEach(function(aPet) {
       if(txt.length > 0) {
           txt += ' ';
       }
       txt += aPet.name ? aPet.name : '';
    });
    evt.cell.display = txt;
    return true;
}
```

The function above is invoked for every element of the owners list. We get the owner's pet, create the string representation and render it in the grid cell.

1.8. 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 a string parameter named **lastNamePattern** and the owners entity based on the **OwnersQuery**. Bind the model's and the entity's parameters.

Start Excel to edit the report template. Provide the report's header, owners tables columns headers and the columns tags as it shown below:

Name	Address	City	Phone
${owners.fullname}	**${owners.address}**	**${owners.city}**	**${owners.telephone}**

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

```
/**
 * Report button click event handler.
 * @param evt Event object
 */
form.reportButton.onActionPerformed = function(evt) {
    var ownersReport = new OwnersReport();
    ownersReport.params.lastNamePattern =
        form.params.lastNamePattern;
    ownersReport.show();
}
```

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

Appendix A. Revision History

Revision 0-0	**8.08.2013**	**Vadim Vashkevich** *`vv@altsoft.biz`*, **Maxim Lipnin** *`ml@altsoft.biz`*

Initial revision

Index