

# **The Wolfram Application Server: Getting Started**

**Build your first Wolfram application**

DRAFT

# The Wolframe Application Server: Getting Started: Build your first Wolframe application

Publication date December 18, 2012 version 0.0.2

Copyright © 2010 - 2013 Project Wolframe

**Commercial Usage.** Licensees holding valid Project Wolframe Commercial licenses may use this file in accordance with the Project Wolframe Commercial License Agreement provided with the Software or, alternatively, in accordance with the terms contained in a written agreement between the licensee and Project Wolframe.

**GNU General Public License Usage.** Alternatively, you can redistribute this file and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Wolframe is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Wolframe. If not, see <http://www.gnu.org/licenses/>

If you have questions regarding the use of this file, please contact Project Wolframe.

# Table of Contents

Foreword .....	iv
1. Introduction .....	1
1.1. Introduction to the Wolfram tutorial .....	1
1.1.1. Requirements .....	1
1.1.2. Sample application .....	1
1.1.3. Basic directory layout .....	1
2. Steps .....	2
2.1. Step 1 - Basic connectivity .....	2
2.2. Step 2 - Basic Authentication .....	3
2.3. Step 3 - Get the whole chain working .....	3
2.4. Step 4 - Implement server side customer list .....	4
2.5. Step 5 - Implement client side customer list .....	8
2.6. Step 6 - Show customer .....	10
2.7. Step 7 - Editing customer .....	15
2.8. Step 8 - Add a new customer .....	16
2.9. Step 9 - Delete a customer .....	18

# Foreword

This is the Wolfram Tutorial.

It describes step by step how to use Wolfram to build a small demo application.

# Chapter 1. Introduction

## 1.1. Introduction to the Wolframe tutorial

### 1.1.1. Requirements

You need the following packages (we assume you are on an Ubuntu Linux, for other platforms there are similar packages available):

- wolframe\_0.0.1-1\_amd64.deb: the Wolframe core server
- wolframe-sqlite3\_0.0.1-1\_amd64.deb: the Sqlite3 database driver module
- wolframe-libxml2\_0.0.1-1\_amd64.deb: the XML filter based on libxml2
- wolclient\_0.0.3-1\_amd64.deb: the Wolframe graphical user interface client

### 1.1.2. Sample application

We want to manage a list of customers with name and address and provide the usual operations:

- list all customers
- create new customers
- edit existing customers
- view customer entries
- delete customers
- search for customers

### 1.1.3. Basic directory layout

We don't use the normal directory layout as it requires root rights to install.

Instead we create two directories, one for server data and one for client data:

```
mkdir ~/tutorial
mkdir ~/tutorial/server
mkdir ~/tutorial/client
```

# Chapter 2. Steps

## 2.1. Step 1 - Basic connectivity

We have to set up the wolframed daemon with a running configuration file:

```
cd ~/tutorial/server
```

We create the central configuration file of the server `tutorial.conf`:

```
; we install a verbose default logger to the shell
logging {
  stderr {
    level DEBUG
  }
}

; one connection, one thread is enough
listen {
  maxConnections 1
  threads 1
  socket {
    address *
    port 7661
    maxConnections 1
  }
}
```

We can start the server now in the shell foreground with:

```
/usr/sbin/wolframed -f -c tutorial.conf
```

We see the following output of the server in the shell:

```
NOTICE: Starting server
DEBUG: Random generator initialized. Using device '/dev/urandom'
DEBUG: AAAA database references resolved
INFO: Accepting connections on ::1:7661
DEBUG: 1 network acceptor(s) created.
DEBUG: 0 network SSL acceptor(s) created.
```

The server is up and listening to port 7661. The server can be stopped anytime by pressing **Ctrl+C**.

If we use a telnet to connect to the server with:

```
telnet localhost 7661
```

we get:

Access denied.

The server tells us:

```
DEBUG: Connection from ::1:52134 to localhost:7661 not authorized
DEBUG: Connection to ::1:52134 closed
```

So we have to configure some basic authorization first.

## 2.2. Step 2 - Basic Authentication

So we have to add a dummy authorization to the server configuration which accepts all connections (not very secure, but for now good enough):

```
; dummy authorization
AAAA {
    Authorization {
        default allow
    }
}
```

If we start the server now, the telnet shows us:

```
Escape character is '^]'.
Wolframe
OK
```

Type **quit**, then **Enter** now and get back to the shell:

```
BYE
Connection closed by foreign host.
```

## 2.3. Step 3 - Get the whole chain working

Now that we have ensured that basic connectivity to the Wolframe server is available, we can configure the basics for the Qt client.

We start the Qt client with:

```
wolfclient -c tutorial.conf
```

First define your connection by selecting "Manage Connections" in the "Connection" menu. Define a new connection called "tutorial" which connects to server "localhost" on port 7661. Leave the SSL connection unchecked.

Now you can try to login to your server by selection "Connection" and then "Login" in the menu.

Because we didn't write any user interfaces yet, we get an error message:

```
Unable to load form 'init', does it exist?
```

To get rid of that error message we will have to create our start form in the Qt designer first. For now we just click away the error message.

We start now the Qt designer and create an empty QWidget form named `~/tutorial/client/init.ui` and save it.

If we restart the client and login in we see the same empty window again, but this time it's the dynamically loaded initial form (which is again empty). The previous error message disappeared.

## 2.4. Step 4 - Implement server side customer list

We want to store the customer data in a sqlite database, so we have to prepare the server:

```
LoadModules {  
    module mod_db_sqlite3  
}
```

This loads the database driver for Sqlite3. Now we also have to create a database and populate the following schema:

```
CREATE TABLE Customer (  
    id            INTEGER PRIMARY KEY AUTOINCREMENT,  
    name          TEXT      NOT NULL,  
    address       TEXT  
);
```

Store this into `schema.sql`. Then execute:

```
sqlite3 tutorial.db < schema.sql
```

Now we have to tell server to use this sqlite database file:

```
database {  
    SQLite {  
        identifier sqlitedb  
        file tutorial.db  
        foreignKeys yes  
    }  
}
```

When we restart the server we see:



```
DEBUG: SQLite database unit 'sqlitedb' created with 3 connections to file 'tut
```

Now we want to use some XML filters to send/receive XML over the protocol, so we have to add the following module to tutorial.conf:

```
LoadModules {  
  ..  
  module mod_filter_libxml2  
  ..  
}
```

In order to see which modules are currently loaded in the wolframed we can use:

```
wolframed -p -c tutorial.conf
```

We see:

```
..  
Module files to load:  
  /usr/lib/wolframed/modules/mod_db_sqlite3  
  /usr/lib/wolframed/modules/mod_filter_libxml2  
..
```

which looks ok.

For mapping the requests to programs we need the directmap module. First add to tutorial.conf:

```
LoadModules {  
  ..  
  module mod_command_directmap  
  ..  
}
```

and

```
Processor {  
  database sqlitedb  
  cmdhandler {  
    directmap {  
      program tutorial.directmap  
    }  
  }  
}
```

Now we have to a file tutorial.directmap. This file maps the requests to the corresponding transaction definitions:

```
CustomerListRequest = SelectCustomerList( xml );
```

This means that we map the 'CustomerListRequest' request which gives us a list of customers to the TDL transaction 'SelectCustomerList'. This also means we need to configure the TDL program in the server:

```
Processor {  
  program Customer.tdl  
}
```

The Customer.tdl file contains the database transaction we want to execute. We also specify that we want the result to be a 'list' which contains 'customer' tags with the data per customer:

```
TRANSACTION SelectCustomerList  
RESULT INTO list  
BEGIN  
  INTO customer DO SELECT * from Customer;  
END
```

We also need a validator for the input when the client sends a 'CustomerListRequest'. we install the simple form DDL compiler in tutorial.conf and register the simpleform program to the list of programs:

```
LoadModules {  
  ..  
  module mod_ddlcompiler_simpleform  
}  
  
Processor {  
  ..  
  program CustomerListRequest.simpleform  
  ..  
}
```

and we add a simple form file CustomerListRequest.simpleform, for now we can leave it empty but for the root element:

```
FORM CustomerListRequest  
{  
  customer {  
  }  
}
```

Now we create a telnet request, which contains basic authentication and a request for the list of customers, called CustomerListRequest.netcat:

```
AUTH  
MECH NONE  
REQUEST  
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE customer SYSTEM 'CustomerListRequest'>
```

```
<customer/>
.
QUIT
```

This we can execute with:

```
netcat -v --wait=2 localhost 7661 < CustomerListRequest.netcat
```

and we get:

```
Wolframe version 0.0.5 OS
OK
MECHS NONE
OK authorization
ANSWER
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<list/>

.
OK REQUEST CustomerListRequest
```

We got an empty list of customers. So we have successfully configured the server for our first command.

Let's also add some demo data now:

```
cat > data.sql
insert into customer(name,address) values('Dr Who','Blue Police Box');
insert into customer(name,address) values('John Smith','The Wheel in Space');
Ctrl-D

sqlite3 tutorial.db < data.sql
```

When we reexecute the netcat command we see:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<list>
  <customer>
    <id>1</id>
    <name>Dr Who</name>
    <address>Blue Police Box</address>
  </customer>
  <customer>
    <id>2</id>
    <name>John Smith</name>
    <address>The Wheel in Space</address>
  </customer>
</list>
```

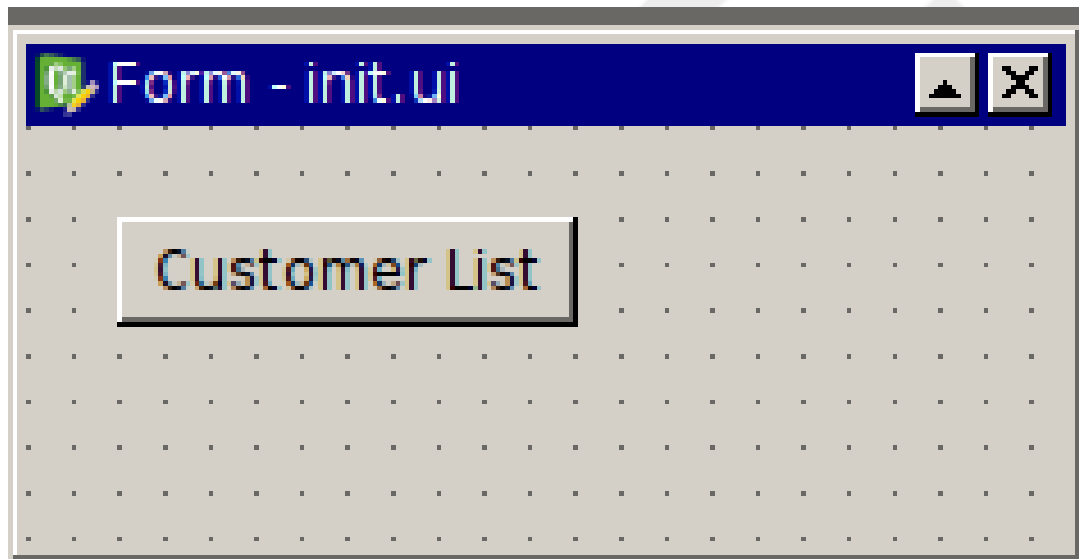
So, the data is now correctly retrieved from the database.

We can move now to the wolfcient to make our first request visible.

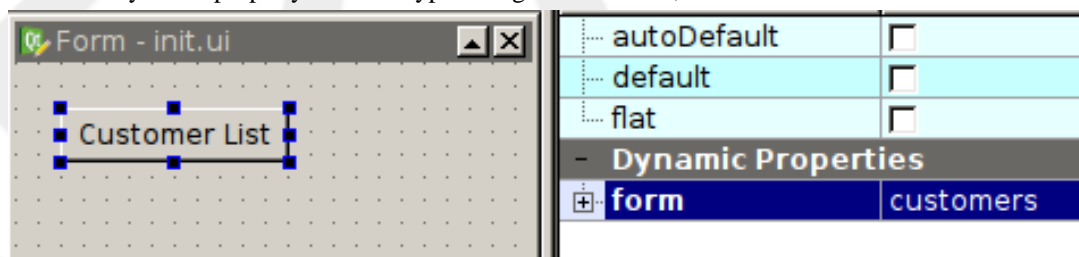
## 2.5. Step 5 - Implement client side customer list

It's time now to get something visual working, so we start to add a first simple interface to our wolfcient.

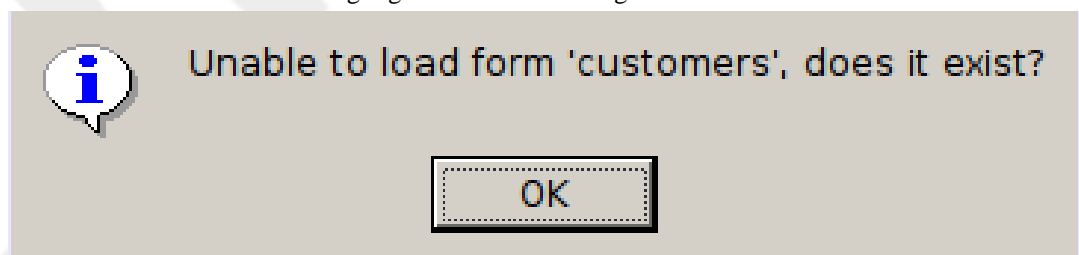
For this we need the 'Qt designer'. We open the file `~/tutorial/client/init.ui` again draw a single button with the text "Customer List":



We add a dynamic property 'form' of type String to this button, which has the value 'customers':

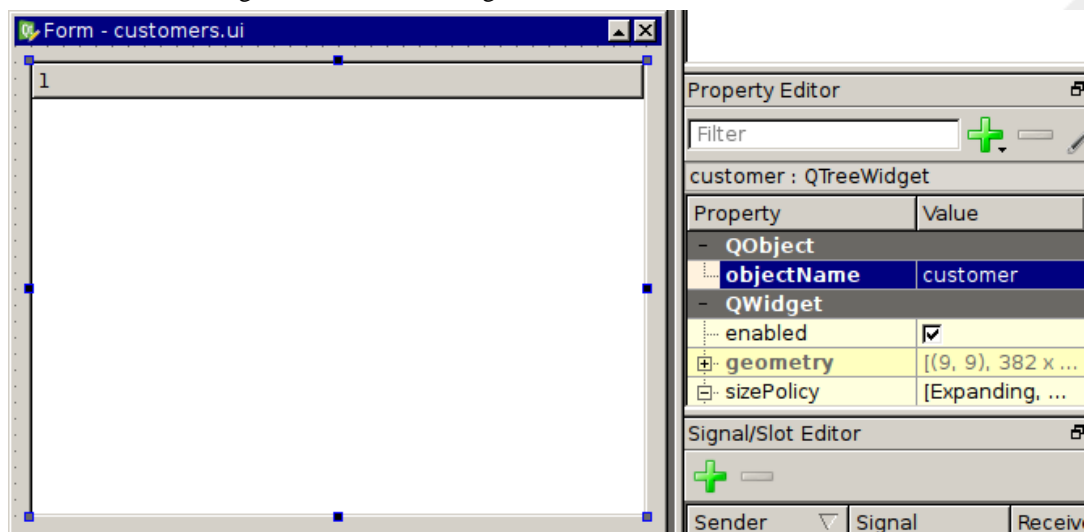


When we save the form and start the wolfcient, we get (after logging in) the first page with the "Customer List" button. Pressing it gives the error message:



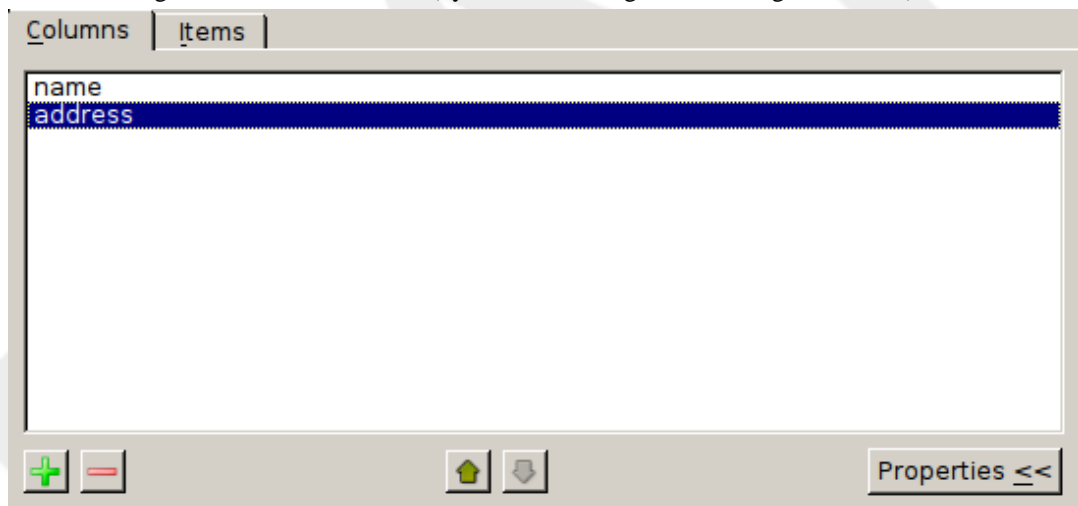
This means we have to define a new form called `customers.ui`, which will show the list of customers, for now we leave it empty. When we start the wolfcient and press the "Customer List" button again, we see that the form gets changed to the "customer" form (empty).

We add now a `QTreeWidget` item to the `customer.ui` form and choose a grid layout for the whole form. We change the name of the widget to 'customer':

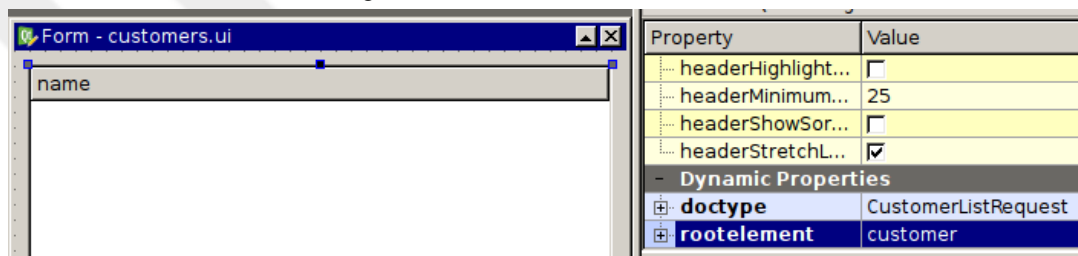


We also disable the 'rootIsDecorated' tick (we have a list, not a tree of customers). We also set 'selectionMode' to 'SingleSelection' and 'selectionBehaviour' to 'selectRows' to get the default expected behaviour of a list.

Now we change the columns of the list (by double clicking into the widget data area):



Now we add the 'doctype' and 'rootelement' dynamic properties to the customer list widget, so that it loads the domain when the form is loaded. The request to send is the 'CustomerListRequest' with root element 'customer' we have configured before in the server:



When we start the wolflclient, we get a funny error message:

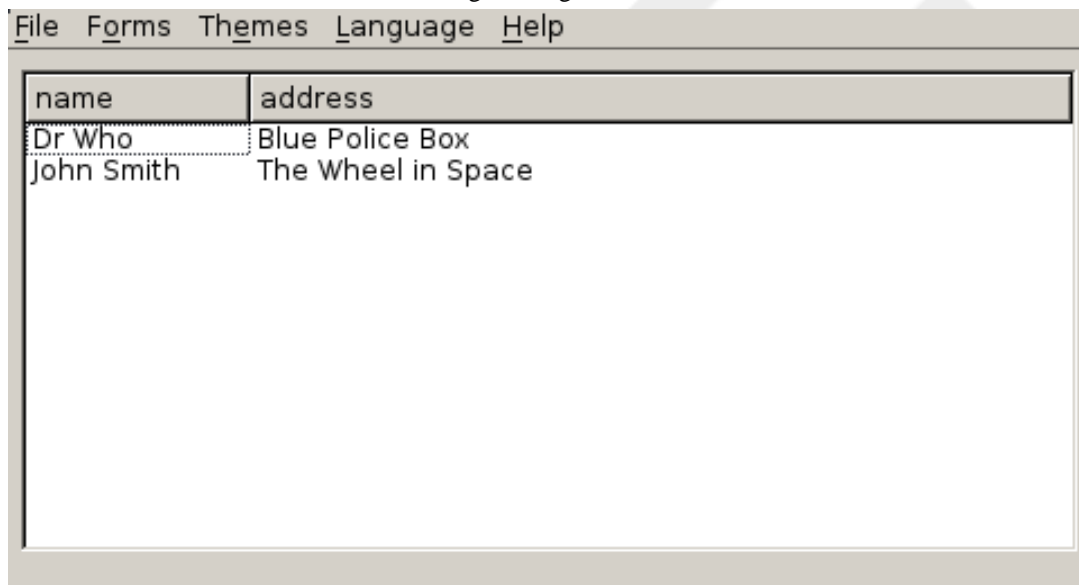
```
Error calling procedure: unknown attribute (id) at /customer
```

This means we have to configure the id attribute in the server in the simpleform. The wolfcient adds an id automatically to the customer tag in the request.

We declare the customer to have an optional 'id' attribute in `CustomerListRequest.simpleform`:

```
FORM CustomerListRequest
{
  customer {
    id ?@string
  }
}
```

If we restart the server and run wolfcient again we get the desired result:

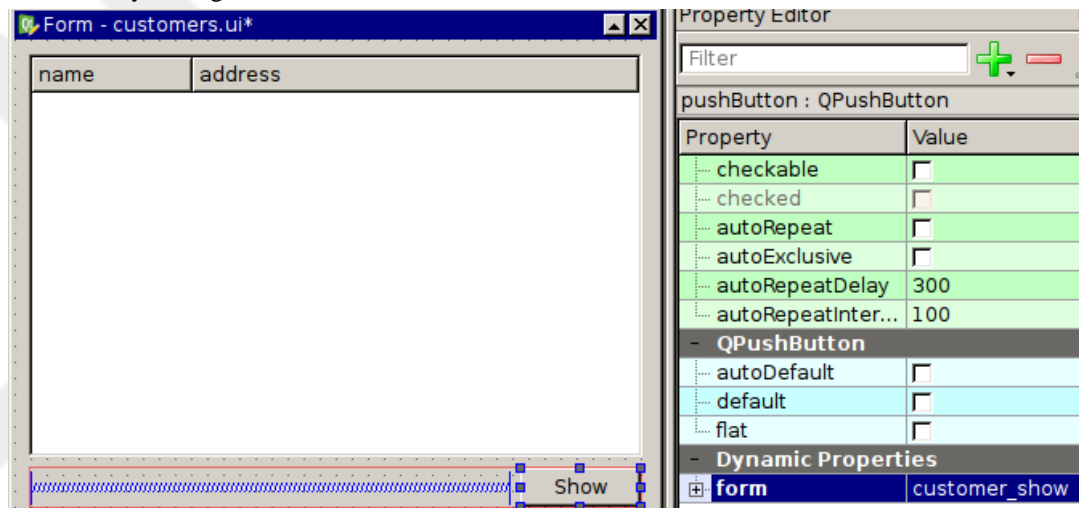


name	address
Dr Who	Blue Police Box
John Smith	The Wheel in Space

## 2.6. Step 6 - Show customer

As next step we want to show how data is communicated between the forms by implementing a simple "show me customer data" use case.

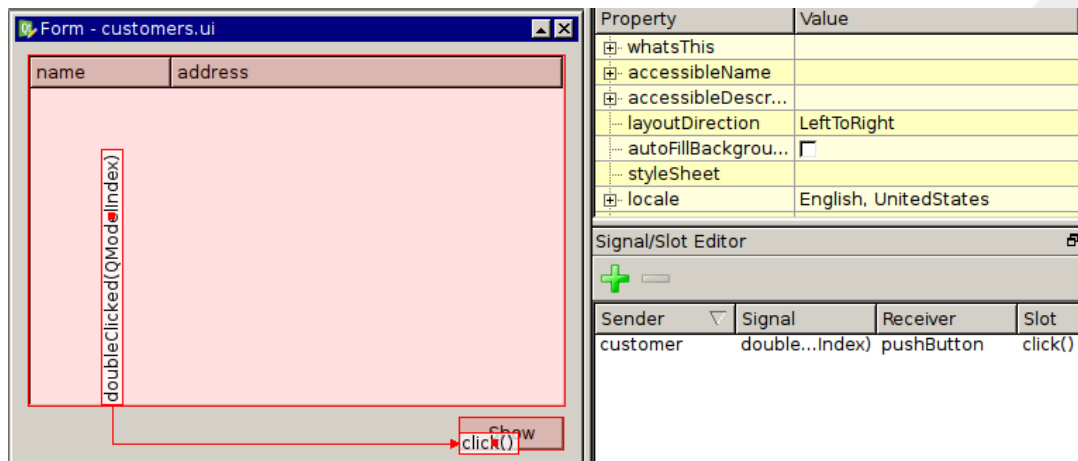
Let's start by adding a button in `customers.ui` called 'Show':



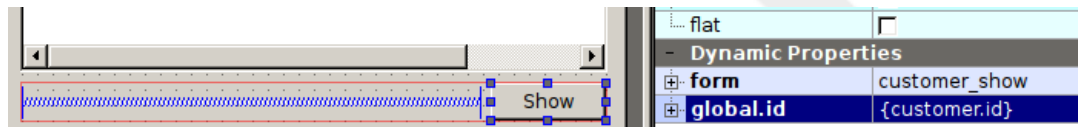
Property	Value
checkable	<input type="checkbox"/>
checked	<input type="checkbox"/>
autoRepeat	<input type="checkbox"/>
autoExclusive	<input type="checkbox"/>
autoRepeatDelay	300
autoRepeatInter...	100
- QPushButton	
autoDefault	<input type="checkbox"/>
default	<input type="checkbox"/>
flat	<input type="checkbox"/>
- Dynamic Properties	
form	customer_show

We add dynamic properties, one is the 'form' which we set to 'customer\_show'.

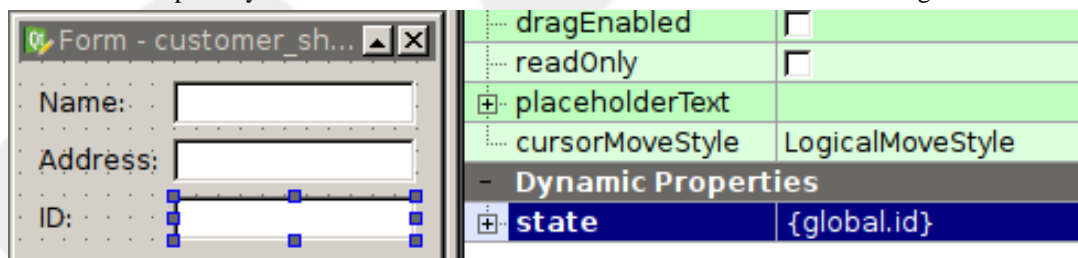
We also add some signals for the double click on the customer list to click the "Show" button:



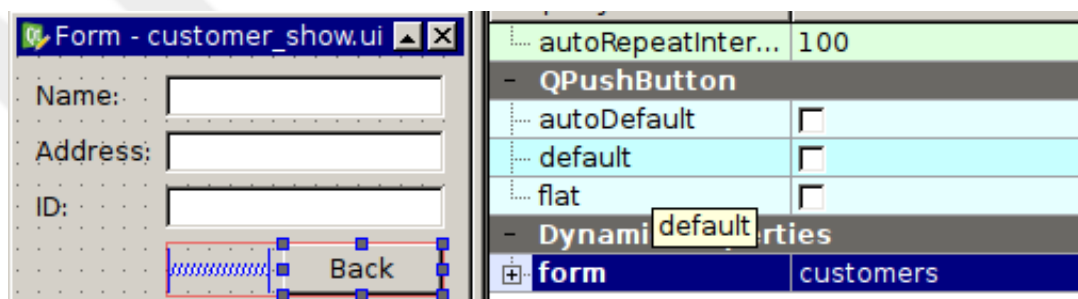
We want the currently selected customer to be accessible in the `customer_show.ui` form, so we have to remember the current selection of the `QTreeWidget` named 'customer' as global variable 'id'. We add a dynamic property with the name 'global.id' and the value '{customer.id}':



Now of course we have to create a new form called `customer_show.ui`. We choose a form layout and add two fields with labels 'Name:' and 'Address:' and each of them having a `QLineEdit` widget. The names of the widgets should be 'name' and 'address' in order to match the future read request from the server. Temporarily we also add an 'id' field which shows us the current value of 'global.id':



Finally we also need a button which brings us back to the customer list by simply adding a 'form' action with the value 'customers':



If we start the `wolfclient` and select a customer and press 'Show' we will notice that the whole "Show Customer" form is empty, even the id. What's wrong? Well, the widgets in `wolfclient` have the default behaviour of using 'id' attributes attached to the columns of a list for instance. We must make sure, the server is mapping us the XML correctly with 'id' attribute and not 'id' as element.

This is the moment we go back to the server and start output form validation. We add a `CustomerList.simpleform` form which describes the result of the 'CustomerListRequest' more precisely and especially declares the 'id' as mandatory attribute of the customer:

```
FORM CustomerList
{
  list
  {
    customer []
    {
      id !@string
      name string
      address string
    }
  }
}
```

We have to declare output form validation in the 'CustomerListRequest' in `tutorial.directmap` as well:

```
CustomerListRequest = SelectCustomerList( xml ) : CustomerList;
```

and we have to include the program in `tutorial.conf` of the server:

```
Processor {
  ..
  program CustomerList.simpleform
  ..
}
```

Checking with:

```
netcat -v --wait=2 localhost 7661 < CustomerListRequest.netcat
```

we get now:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE list SYSTEM "CustomerList.simpleform">
<list>
  <customer id="1">
    <name>Dr Who</name>
    <address>Blue Police Box</address>
  </customer>
  <customer id="2">
    <name>John Smith</name>
    <address>The Wheel in Space</address>
  </customer>
</list>
```

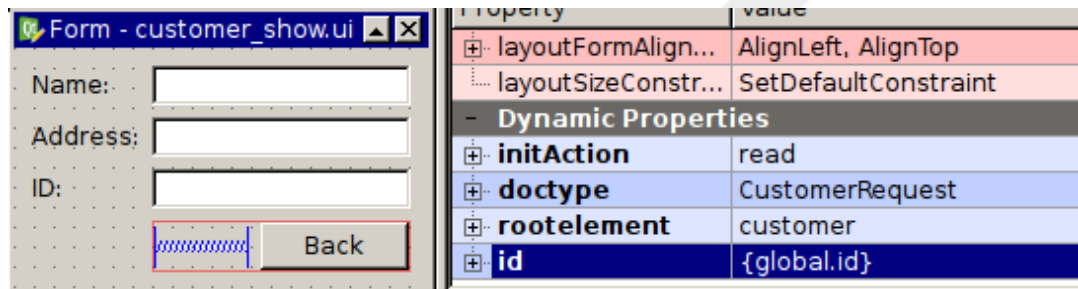


So, that looks ok.

Now also the wolflclient shows the id of the customer, when the user double clicks the customer in the list.

The 'name' and the 'address' fields are still empty. We could of course use global variables again to propagate the values between forms, but if the form gets more complex, this is not a good idea.

We start to use the 'initAction' property on the `customer_show.ui` form as follows: we want it to execute a read with a document type 'CustomerRequest' which searches for a single customer by customer id:



We see, that we use the value of 'global.id' again and add it to the 'initAction' request as attribute 'id' (we can use the wolflclient with -d to see the communication on the wire):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE customer SYSTEM 'CustomerRequest'>
<customer id="2"/>
```

We also get in the client:

```
CRITICAL: ERROR: error in network data loader "Protocol error in command REQUEST"
```

and in the server:

```
ERROR: error redirect input: no command handler for 'CustomerRequest'
```

We see, that the request is sent to the server, but we didn't define the necessary things in the server yet. So we add another map:

```
CustomerRequest = SelectCustomer( xml ) : Customer;
```

to `tutorial.directmap`.

We define a new file `CustomerRequest.simpleform` which contains the validation of the customer request (for now, this looks exactly like the request for the list of customers, but we may want to change that later):

```
FORM CustomerRequest
{
  customer {
    id ?@string
  }
}
```

We register the form in `tutorial.conf`:

```
Processor {
  ..
  program CustomerRequest.simpleform
  ..
}
```

And of course we have to define a transaction function 'SelectCustomer' in `Customer.tdl`:

```
TRANSACTION SelectCustomer
RESULT INTO customer
BEGIN
  INTO . DO SELECT * from Customer WHERE id=$(customer/id);
END
```

The '\$(customer/id)' refers to the ID we pass down for the customer record to retrieve. The 'RESULT INTO customer' makes sure the result will again be in a table with 'customer' as root element.

We also have to define how the result should be mapped, so:

```
Processor {
  ..
  program Customer.simpleform
  ..
}
```

and `Customer.simpleform`:

```
FORM Customer
{
  customer
  {
    id !@string
    name string
    address string
  }
}
```

Now if we restart client and server and we click on the second customer in the list we get:

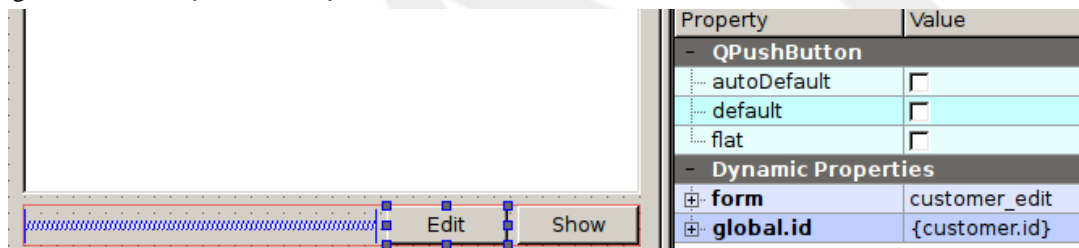
So we successfully read the data of a customer into a form.

## 2.7. Step 7 - Editing customer

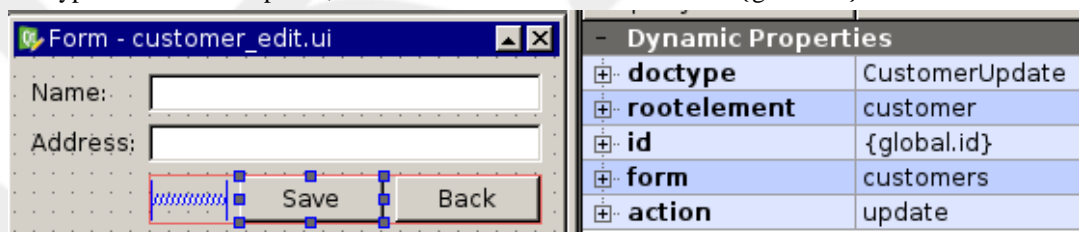
We change the show customer use case slightly, so that we can also edit the customer in the `customer_show.ui` form.

Let's first make a copy of `customer_show.ui` and name it `customer_edit.ui`.

As before we add first a "Edit" button to the `customers.ui` with 'forms' set to 'customer\_edit' and 'global.id' set to '{customer.id}':



We change the form `customer_edit.ui` and remove the line with 'ID' as we don't need it anymore and because nobody should be able to edit the id of a customer and change it! We also add another button and label it 'Save', this button get the properties 'form' set to 'customers', 'action' set to 'update', 'doctype' to 'CustomerUpdate', 'rootelement' to 'customer' and 'id' to '{global.id}':



The other fields will be sent along automatically in the 'CustomerUpdate' XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE customer SYSTEM 'CustomerUpdate'>
<customer id="2">
  <name>John Smith</name>
  <address>The Wheel in Space</address>
</customer>
```

Similar to to the "show customer" case we add now a new map:

```
editCustomerUpdate = UpdateCustomer( xml );
```

We also add a simple form `CustomerUpdate.simpleform` which looks very similar to the `Customer.simpleform`:

```
FORM CustomerUpdate
{
  customer
  {
    id !@string
    name string
    address string
  }
}
```

add it to `tutorial.conf`:

```
Processor {
  ...
  program CustomerUpdate.simpleform
  ...
}
```

finally we write the transaction function 'CustomerUpdate' in `Customer.tdl`:

```
TRANSACTION UpdateCustomer
BEGIN
  DO UPDATE Customer SET name=$(customer/name), address=$(customer/address)
  WHERE id=$(customer/id);
END
```

Note, that this time the database transaction doesn't return a result.

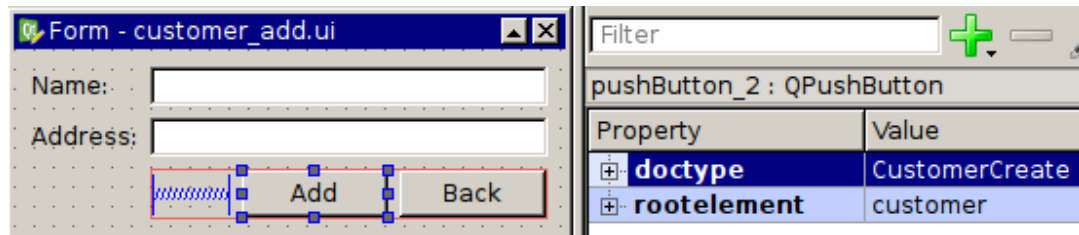
Restart server and client and start to edit the customers.

## 2.8. Step 8 - Add a new customer

We also want to add new customers to the list. The "add customer" case is very similar to the "edit customer" case.

Let's copy the `customer_edit.ui` to `customer_add.ui`. We remove the 'initAction' and the other dynamic properties from the form itself as we don't want to read anything when creating a new customer. But of course we could execute here a "Get new customer initial data" request too which initializes certain values.

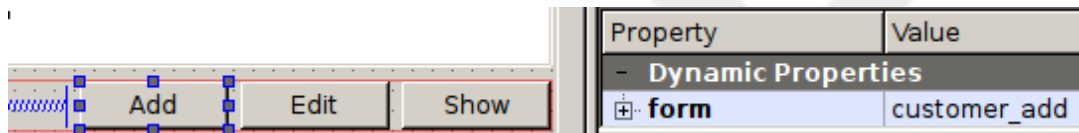
We change the "Save" button and rename it to "Add". We also change the attribute 'doctype' there from 'CustomerUpdate' to 'CustomerCreate' and we remove the 'id' property (as this one is automatically chosen by the sequence in the database). We also have to change the 'action' from 'update' to 'create':



We also introduce a new element here, the 'initialFocus' property. We set it on the 'name' QLineEdit, so that it gets the initial keyboard focus when the form is loaded:



In the `customers.ui` form we have to add a 'Add' button which has one property 'form' with value 'customer\_add':



Now for the server side. We add a new mapping for customer creation in `tutorial.directmap`:

```
editCustomerUpdate = UpdateCustomer( xml );
createCustomerCreate = CreateCustomer( xml );
```

We also have to add the simple form `CustomerCreate.simpleform`:

```
FORM CustomerCreate
{
  customer
  {
    name string
    address string
  }
}
```

This is the same as `CustomerUpdate.simpleform` with the exception that we don't accept an 'id' attribute to be passed to the server.

We register this form in `tutorial.conf`:

```
Processor {
  ..
  program CustomerCreate.simpleform
  ..
}
```

Last we add a 'CreateCustomer' transaction function:

```

TRANSACTION CreateCustomer
BEGIN
  DO INSERT INTO Customer( name, address )
    VALUES( $(customer/name), $(customer/address) );
END

```

When we restart the server and client we see the following request being passed to the wolframe server:

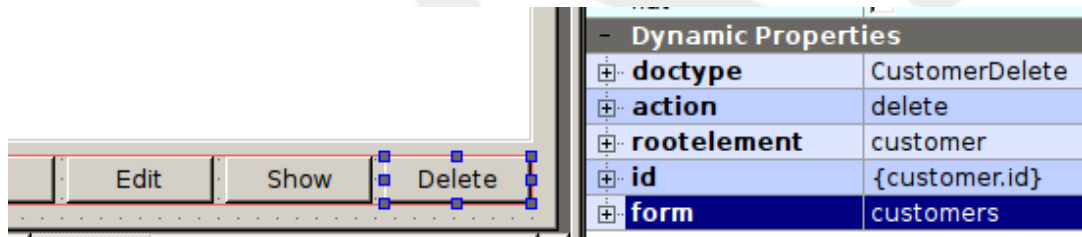
```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE customer SYSTEM 'CustomerCreate'>
<customer>
  <name>New Customer</name>
  <address>New Location</address>
</customer>

```

## 2.9. Step 9 - Delete a customer

We want to get rid of customers. For this we have to change little in the `customs.ui` form: a button "Delete" with takes the following properties: "action" set to "delete", "doctype" set to "CustomerDelete", "rootelement" to "customer", "form" to "customers" (this is the simplest way to reload the list of customers after the deletion) and "id" to "{customer.id}":



We add another map for the 'deleteCustomerDelete' request in `tutorial.directmap`:

```
deleteCustomerDelete = DeleteCustomer( xml );
```

A new `CustomerDelete.simpleform` which allows us only the specify an 'id' attribute of the customer to delete:

```

FORM CustomerDelete
{
  customer {
    id !@string
  }
}

```

and in `tutorial.conf`:

```

Processor {
  ..

```

```
program CustomerDelete.simpleform
..
}
```

Finally follows the implementation of the delete transaction in `Customer.tdl`:

```
TRANSACTION DeleteCustomer
BEGIN
  DO DELETE FROM Customer WHERE id=$(customer/id);
END
```

Executing the request we see in the wolflclient debug output:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE customer SYSTEM 'CustomerDelete'>
<customer id="3"/>
```

Seems ok, customer gone. :-)