



UNIVERSITY OF GOTHENBURG

Form construction and system integration

David Carlberg
Karl Wallin

University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2009

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Form construction and system integration

David Carlberg
Karl Wallin

© David Carlberg, June 2009

© Karl Wallin, June 2009

Examiner: Joachim von Hacht

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2009

1 Abstract

When constructing forms with the help of online tools, these forms will be bound to the online tool. The online tools offer hosting of forms and handling of the submission data. The work in this thesis analyzes the possibilities to move the constructed forms and there data handling to an arbitrary web server and presents what must be done to completely move the forms. It also shortly describes the implementation of such a tool that solves this issue.

2 Preface

This is a master thesis in computer science at University of Gothenburg. The work in this thesis was done from January 2009 to June 2009.

We would like to thank the following:

Joachim von Hacht, who was our examiner and whose support has been very valuable for this thesis.

Märt Kalmo, for giving us feedback on this thesis.

3 Table of contents

1	Abstract	3
2	Preface.....	4
3	Table of contents	5
4	Background.....	7
5	Purpose.....	9
6	Method.....	10
7	Dictionary	11
8	Role Dictionary.....	13
9	Environment	15
10	System integration	16
10.1	No integration	17
10.2	Form to web server integration	18
10.3	Web server integration.....	19
10.4	Database integration	20
10.5	Server side script to web server integration	21
10.6	Complete integration.....	22
11	Web server integration.....	24
12	Database integration	25
13	Evaluation of existing form builder applications.....	27
13.1	System integration.....	27
14	JavaScript	30
14.1	Object oriented JavaScript	30
14.2	Namespaces in JavaScript	30
14.3	Debugging JavaScript.....	32
15	AJAX.....	33
16	Architecture	34
17	Design	35
17.1	Data representation	35
17.2	Test application client.....	36
17.3	Test application server.....	37
18	Implementation.....	38

18.1	Code generation	38
19	Discussion and conclusion	40
20	Appendix A – Implementation issues	41
20.1	Data representation	41
20.2	Database connection	41
20.3	Iframe URL length	42
21	Appendix B – Test application server scripts	43
22	Appendix C – XML database	44

4 Background

The standard for data collection on the internet today is by the use of forms. A form contains controls that collect input data from the user. A typical form can look like this (Figure 4.1).

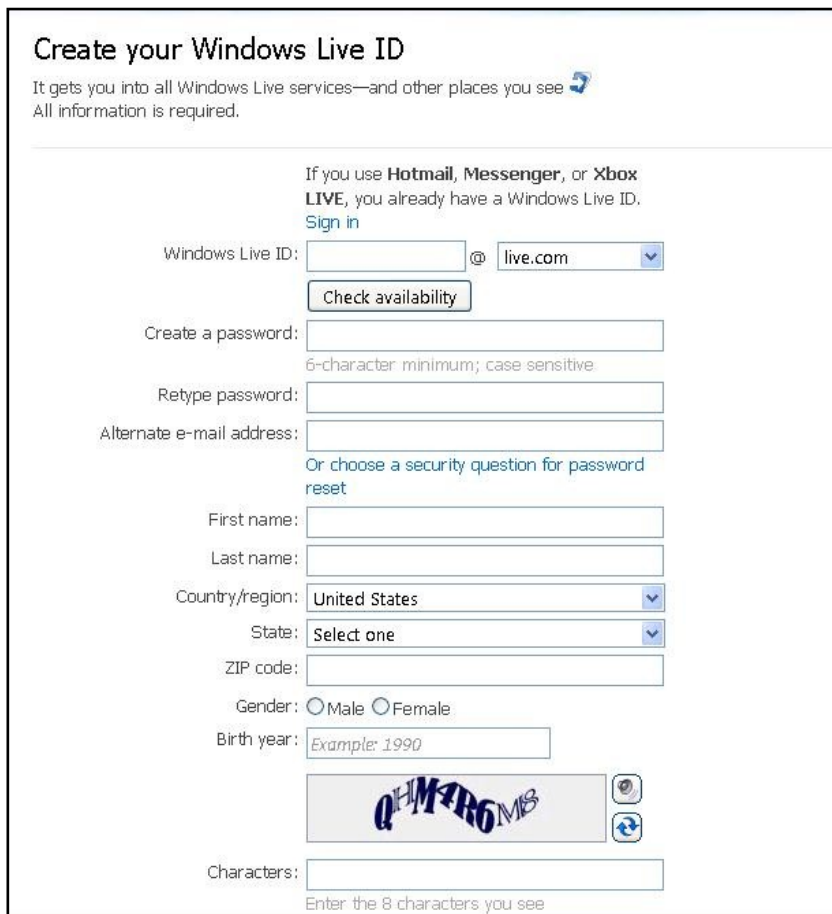


The form is titled "Sign in" and contains the following elements:

- A "Windows Live ID:" label followed by a text input field. Below the field is the text "(example555@hotmail.com)".
- A "Password:" label followed by a text input field.
- A link labeled "Forgot your password?" in blue text.
- Two checkboxes: "Remember me on this computer (?)" and "Remember my password (?)".
- A blue "Sign in" button.

Figure 4-1. A typical login form used online.

A more advance form can look like this (Figure 4.2).



The form is titled "Create your Windows Live ID" and includes the following fields and options:

- Introductory text: "It gets you into all Windows Live services—and other places you see" with a blue arrow icon, followed by "All information is required."
- Text: "If you use **Hotmail, Messenger, or Xbox LIVE**, you already have a Windows Live ID." followed by a blue "Sign in" link.
- "Windows Live ID:" label, a text input field, an "@" symbol, a dropdown menu showing "live.com", and a "Check availability" button.
- "Create a password:" label, a text input field, and the text "6-character minimum; case sensitive".
- "Retype password:" label, a text input field.
- "Alternate e-mail address:" label, a text input field.
- Text: "Or choose a security question for password reset".
- "First name:" label, a text input field.
- "Last name:" label, a text input field.
- "Country/region:" label, a dropdown menu showing "United States".
- "State:" label, a dropdown menu showing "Select one".
- "ZIP code:" label, a text input field.
- "Gender:" label, radio buttons for "Male" and "Female".
- "Birth year:" label, a text input field with "Example: 1990".
- A CAPTCHA image showing the word "QUARONS" with a refresh icon.
- "Characters:" label, a text input field.
- Text: "Enter the 8 characters you see".

Figure 4-2. An advanced form used to register users.

Developing forms that can be used on the web isn't a trivial task, unless the developer possesses knowledge about *HTML*, JavaScript and databases. Even with these skills the process of developing forms can still be very time consuming. Because of this a number of online tools for creating forms were developed to satisfy the needs of both the regular *customers* as well as the professional *customers*.

The service these tools offer includes a visual editor for the form creation, hosting of the completed forms as well as storage and retrieval of submitted data. Because of all these services, almost no resources or even programming knowledge is needed to create and publish a form online. None of today's online tools can however help *customers* to move a form and its data handling into the *customer web server*. All data handling is done by the *form builder application system* and the *customers* have no way of customizing the data handling. So forms created with these tools are bound to the *form builder application system* and there is currently no existing solution for the *customers* to move the forms and data handling into their own system. The term that will be used for moving forms and data handling into a *customer system* will be *system integration*. So what the *customer* gain in simplicity and speed from these services they lose in *system integration*. This is however only problems that need to be considered by the professional *customers* as the option of developing forms by hand is none existing for the regular *customers*. So the problem facing the professional *customers* today is weighing the time issue against the *system integration* issue.

5 Purpose

The purpose of this thesis is to present a solution which enables customers to integrate created forms with their own system.

6 Method

System integration will be solved by implementing a new form builder application that is able to construct forms that can be moved to and integrated with a *customer system*.

7 Dictionary

Server side script

Programming language used on web servers for dynamic web pages. There are many *Server side script* languages available. Some of the more known is ASP/ASP.NET, PHP, Java, Perl, Python and Ruby.

HTML (Hyper Text Markup Language)

HTML is the standard for writing static web pages.

CSS (Cascading Style Sheets)

CSS is used to describe the presentation (font, color, position etc) of a markup language, like *HTML*.

DOM (Document Object Model)

DOM is the object representation of a web page that can be manipulated from JavaScript.

ODBC (Open Database Connectivity)

ODBC is a database access API which is software that makes it possible to connect to an arbitrary database from code.

OLEDB (Object Linking and Embedding Database)

OLEDB is a database access API which is software that makes it possible to connect to an arbitrary database from code (Microsoft only).

JDBC (Java Database Connectivity)

JDBC is a database access API which is software that makes it possible to connect to an arbitrary database from code.

Database driver

Software that is used by the database access APIs (*JDBC*, *ODBC*, *OLEDB*) for communication with proper database.

Database metadata

Metadata is data about data. In the case of databases, metadata describes the structure of the database.

INFORMATION_SCHEMA

INFORMATION_SCHEMA is part of the SQL standard. *INFORMATION_SCHEMA* is a table that provides access to the database metadata.

Connection string

Connection string specifies information about a database and the means of connecting to it. This information includes database driver, host, port, username and password.

Equation

A form *equation* is when values from one or more controls together forms a mathematical expression which result will be placed in a single separate control.

Validation

Form validations is restrictions that has been placed on controls within the form. These restrictions can vary. Some common restrictions that are often placed on controls are:

- Required: When the form is submitted the restricted control must have a value.
- Number: A control with a number restriction will only pass the *validation* if its value is a numeric value.
- Email: This restriction means that the value of the control must have the form of a valid email address.
- Value length (max/min): This restriction means that the value in the control can only have a specified number of characters.

8 Role Dictionary

End user

Fills out a form, using a web browser.

Form builder application

An online application for building forms where no programming knowledge is needed.

Form builder application web server

The web server where the *form builder application* is located.

Form builder application database

The database used by the *form builder application* and the completed forms.

Form builder application system

All hardware and software used by the *form builder application* including the *form builder application web server* and database.

Customer

A user of the *form builder application*.

Customer database

The database owned by the *customer*.

Customer web server

The web server owned by the *customer*.

Customer system

All hardware and software owned by the *customer* including the *customer web server* and database.

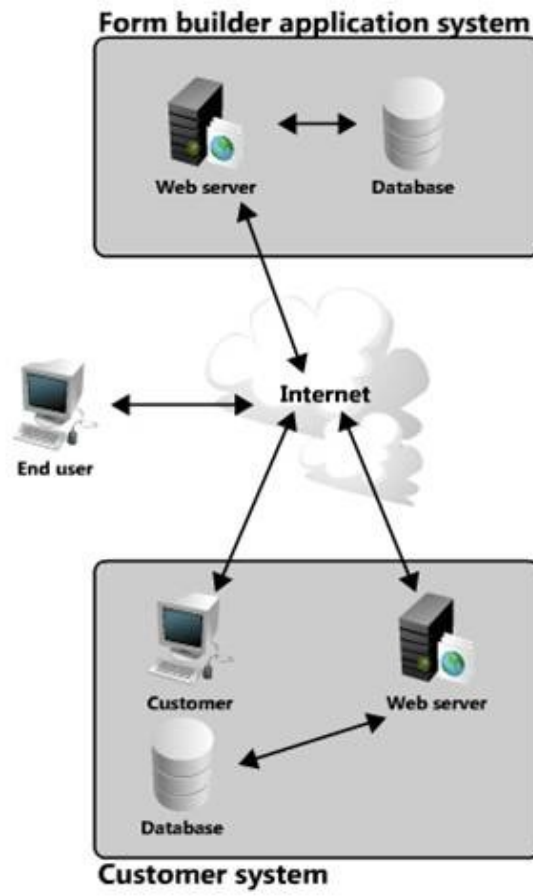


Figure 8-1. A view of the different roles.

9 Environment

A form is a collection of *HTML* controls within a web page that can be filled out and submitted by *end users*. The form is viewed in a browser and the code that the browser uses to display the form is written as *HTML* and *CSS*. A form can also have dynamic functionality with the help of *JavaScript*.

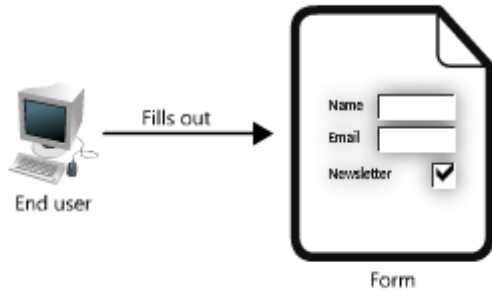


Figure 9-1. Common view of a form

This is the common view of what a form is but in reality there is a lot more functionality in the background that is not visible to the *end user*. When the data is submitted it is sent to a web server where it is handled by a *Server side script*. The *Server side script* can have different tasks including verifying, modifying and store the submitted data.

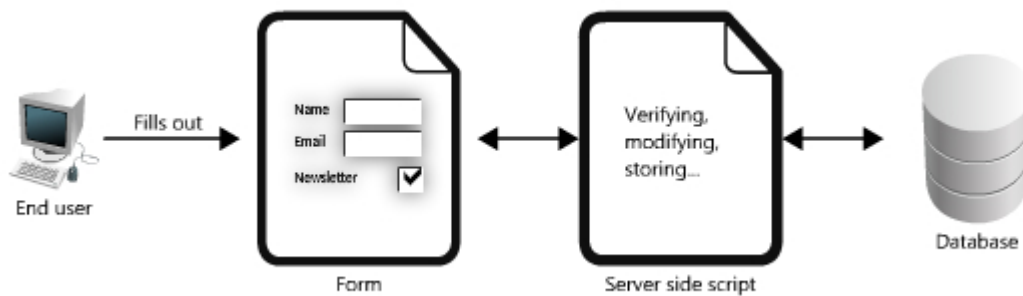


Figure 9-2. Server side script and database functionality

10 System integration

Form builder applications offers a service that helps the *customer* create complicated forms without needing any programming knowledge. One thing that can be viewed as a problem with this service is that the *customer* to some extent can become dependent of the *form builder application system* that provides this service. This level of dependency depends on how much of the created form needs the *form builder application system* to perform its functions. The level of dependency can be reduced by increasing the level of integration. Integration of the form to the *customer system* can be divided into two parts. The first part involves the adjustment of the form as well as the *server side script* which handle the form submission data, so they can be published on the *customer web server*. This will be referred to as *web server integration*. The second part of the *system integration* is where the *server side script* associated with the form, is adjusted so that the submitted data can be sent to the *customer database*. This will be referred to as Database integration.

In order to measure the level of *system integration*, different categories was created that shows how integrated a build form is to the *customer system*. The range of these categories goes from no integration to complete integration. These categories can also be viewed as a measurement of how dependent of the *form builder application system* the *customer* of the service becomes.

10.1 No integration

The form is in this case located on the *form builder application web server*. When the *end user* fills out the form, they need to access it from the *form builder application system*. When a form is not integrated at all the *server side script* associated with the form is also located on the *form builder application web server*. The submission data is stored in the *form builder application database*. This is the most common scenario used by *form builder applications* because it is the most convenient alternative for the *customer* as no external resources is needed.

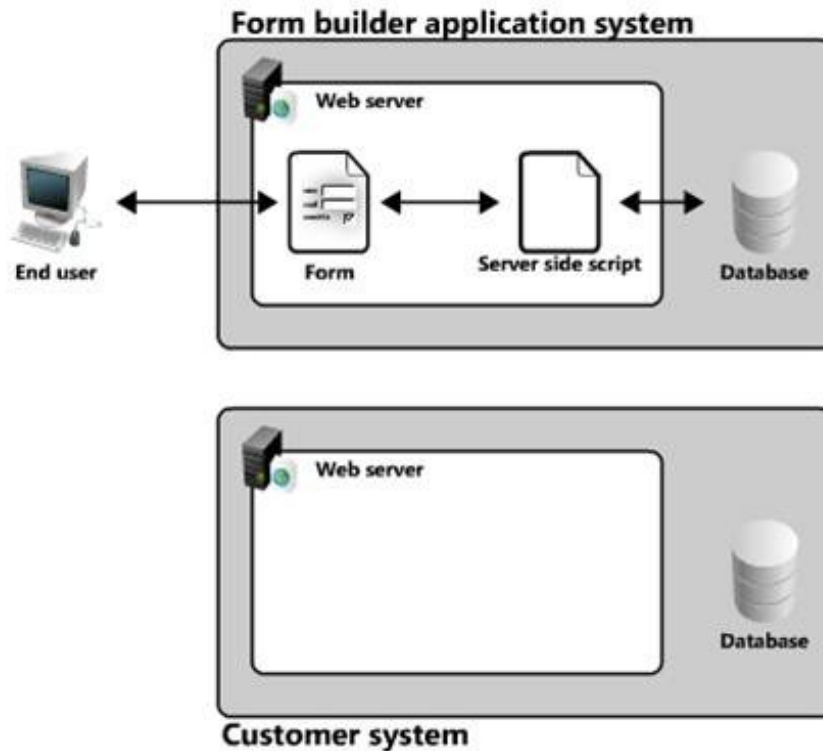


Figure 10-1. No integration

10.2 Form to web server integration

This is when the form can be exported and published on the *customer web server*. The *server side script* is still located on the *form builder application web server* and all submission data is stored in the *form builder application database*. So whenever *end users* fills out a form it is accessed from the *customer web server*, but the submission data is still sent back to the *form builder application web server* where it is handled by the *server side script*. This step requires more from the *customer* since the *customer* need to have access to a web server where the form can be published. This alternative can provide the possibility to make small changes to the form by hand, but for this some programming knowledge is needed.

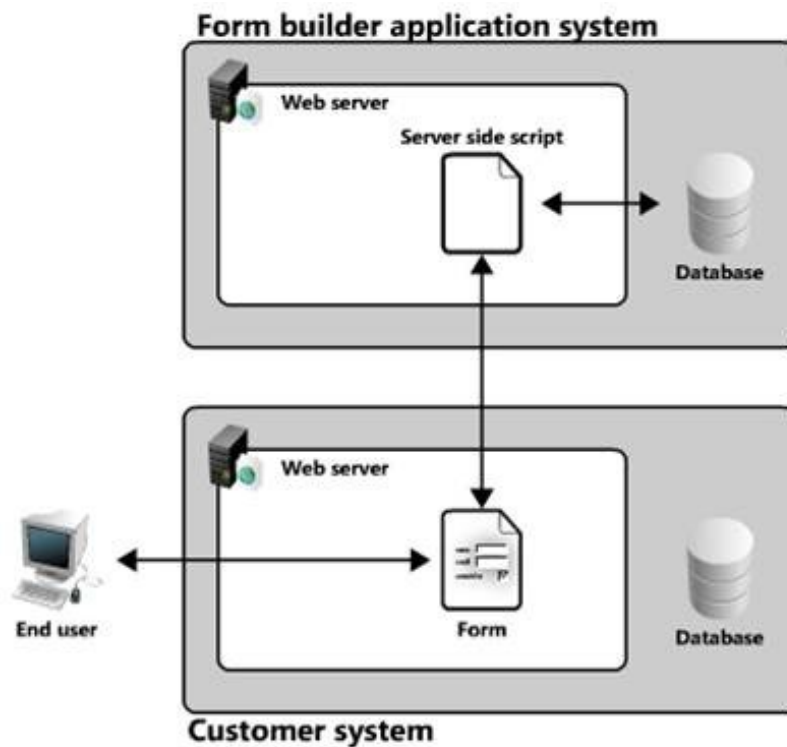


Figure 10-2. Form to web server integration

10.3 Web server integration

This is when both the form and the *server side script* are integrated with the *customer web server*. The *server side script* however, still sends the submission data back to the *form builder application database*. This solution may however impose a security risk since vital information about connection to the *form builder application database* is included in the *server side script*.

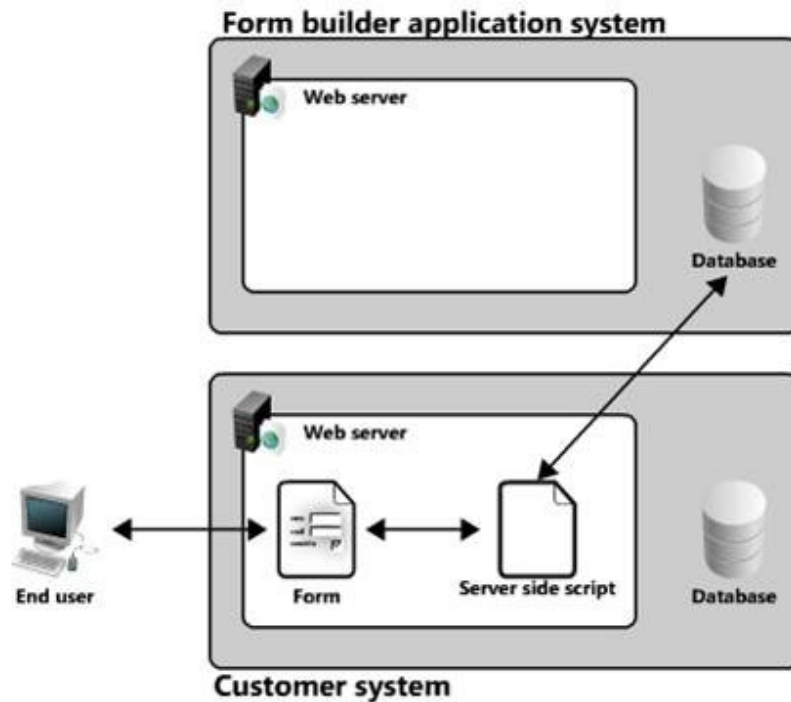


Figure 10-3 Web server integration

10.4 Database integration

Database integration is when the form and the *server side script* both are located on the *form builder application web server*, but submission data is sent to the *customer database*. This requires that the *customer* provides a database and have some knowledge about databases.

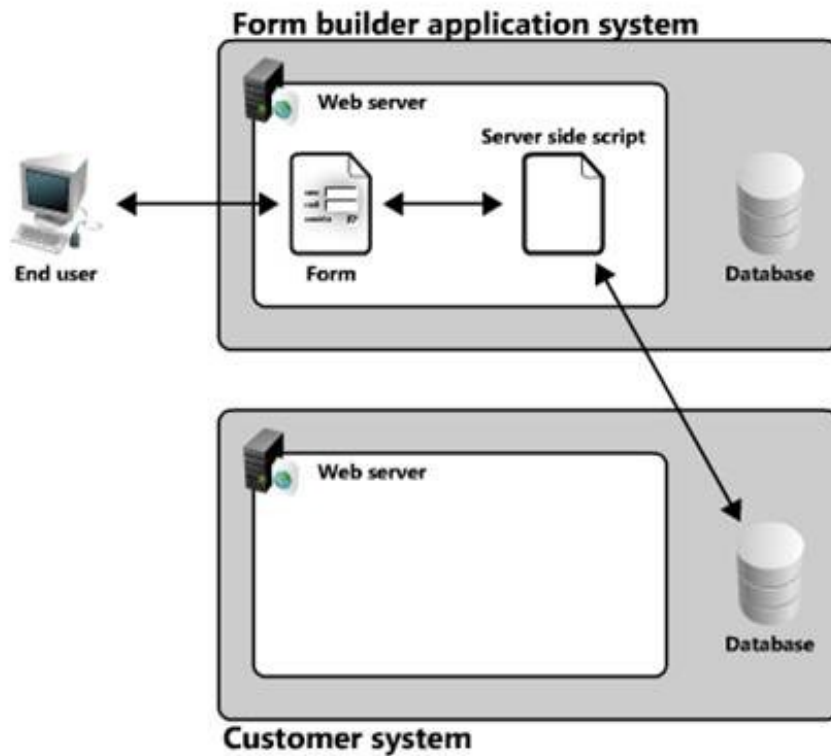


Figure 10-4. Database integration

10.5 Server side script to web server integration

In this case the *server side script* is located on the *customer web server* but the form is still located on the *form builder application web server*. The submission data is sent to the *customer database*. This solution is similar to database integration. As the *server side script* only is a connector between form and the database, its location is of small difference when the form is located at the *form builder application web server*.

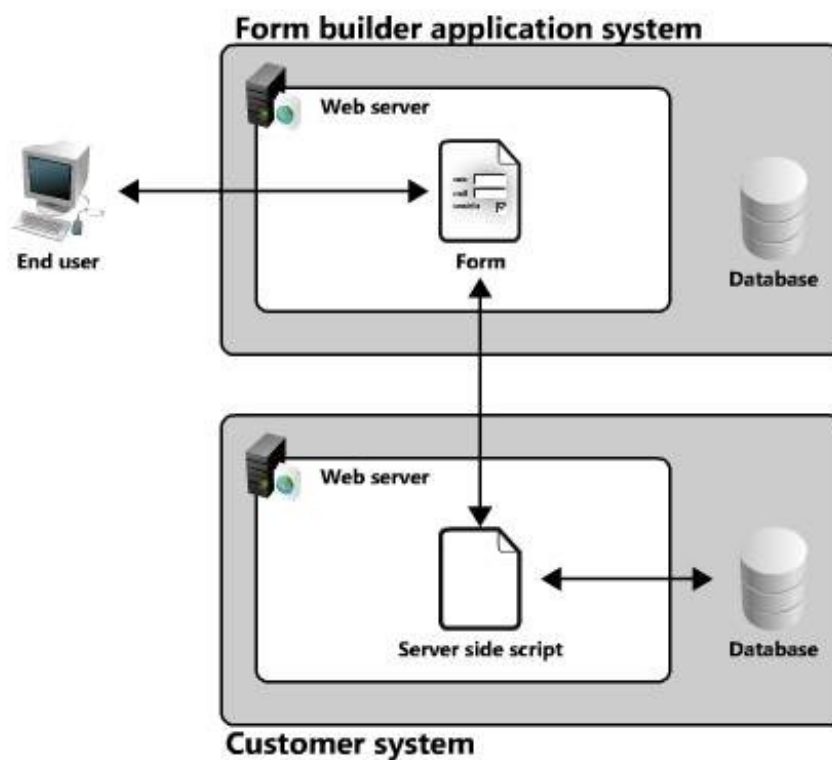


Figure 10-5. Server side script to web server integration

10.6 Complete integration

This is when both *web server integration* and database integration is achieved. Both the form and the *server side script* are located on the *customer web server* and the submission data is sent to the *customer database*. This means that the form has been completely integrated with the *customer system* and has no association with the *form builder application system*. This step requires that the *customer* can provide a web server and a database.

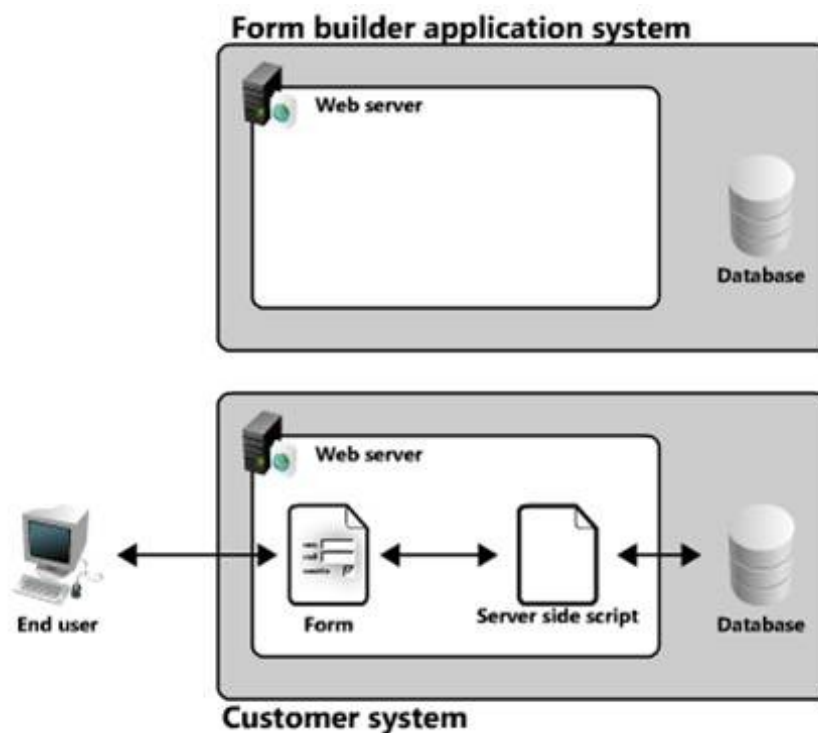


Figure 10-6. Complete integration or System integration

There is another solution to integrate forms into a *customer system* that is used by some *form builder applications* but still doesn't fit into the definition of *system integration*. In this solution the *customer* must install a copy of the entire *form builder application system* on its own system. With this solution the *customer* is in no way dependent on the *form builder application system* but the forms are still dependent on the installed copy of the *form builder application system*.

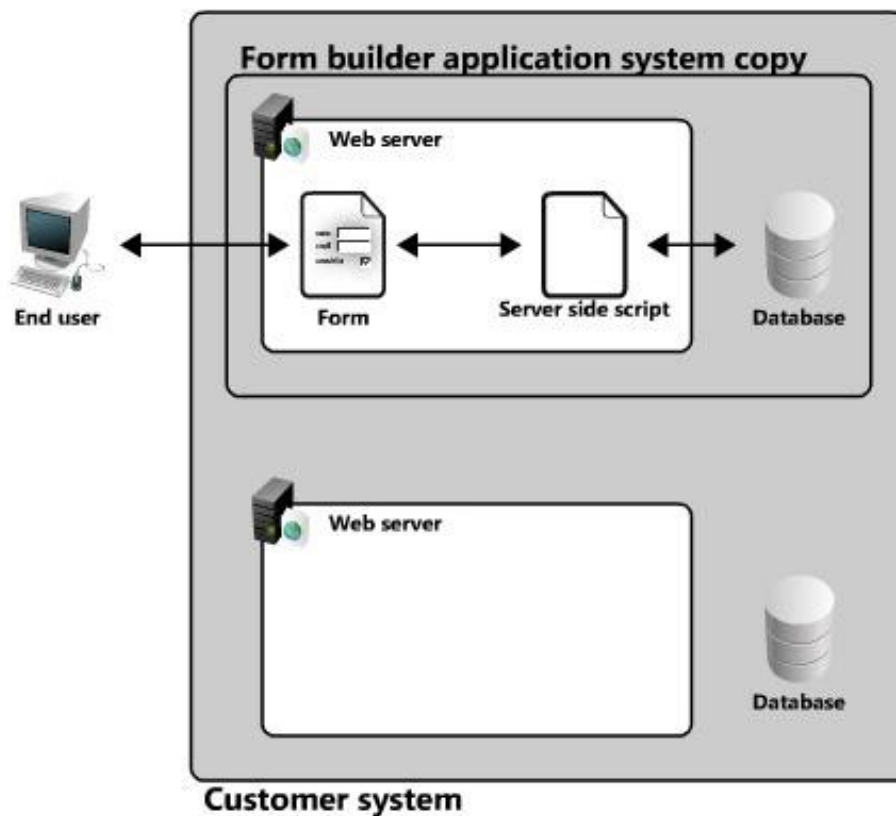


Figure 10-7. Special type of system integration

11 Web server integration

Web server integration consists of two parts, the integration of the form as well as the *server side script* to the *customer web server*. In the Evaluation of existing *form builder applications* chapter, there is a closer description of how existing *form builder applications* have solved *system integration*. None of the existing form builder applications have a solution to web server integration. Most of them supports form to web server integration but none of them supports integration of the *server side script*.

When adding *server side script* integration capabilities to a *form builder application*, some new aspects must be considered. One of the more significant new things to consider is the need for adapting the *server side script* to the *customer web server's* platform. The problem here is that the *customer web server* usually only supports one specific web server software and this may differ from *customer* to *customer* as there are many competing web server software on the market. Different web server software may also support different server side scripting languages. When integrating the *server side script* with the *customer web server* it must be of the same server side scripting language as the *customer web server* supports. To solve this difficulty the *form builder application* must be able to construct the *server side script* in different server side scripting languages. Some of the more popular server side scripting languages are ASP/ASP.net, PHP, Cold Fusion, JSP, Python, Ruby and Pearl.

The server side scripting languages differs a lot and they are all executed on the web server. The form however is executed by the end user's web browser and the support differences between web browsers are much less complicated since the form uses *HTML*, *CSS* and JavaScript which all browsers have almost the same support for. The small problem here is JavaScript where some support differences may occur but this is insignificant in comparison to the differences between different server side scripting languages.

As mentioned earlier, a second form of *system integration* exists, where the *customers* had the possibility to integrate the complete *form builder application system* on their own system. This only works if the *customer system* and the *form builder application system* are compatible. If they are not, the *customer* may have to install new software and hardware which supports the *form builder application system*.

12 Database integration

Database integration means that the *server side script* is able to communicate with the *customer database*. This communication can be divided into two areas. The main area is handling submission data from the form. The other area is reading data from the database which can be used to interact with the form. This interaction can be to fill the form with predefined values or to control form actions depending on existing data.

The platform adaption problem that was mentioned in web server integration also exists in database integration. There are several number of database systems that need to be supported. Most database systems use a standard communication protocol, SQL, which in its most basic functions behaves in the same way for most database systems. Because of this, the adaptation problem for database systems is not as significant as with web server platform adaptation. The problem here is the initial connection as it's often not possible to directly connect to the database from a *server side script*. *Server side scripts* often use a database access API like *ODBC*, *JDBC* or *OLEDB* to communicate with arbitrary databases. This is where the largest platform adaption problem occurs for database integration as these database accesses APIs uses drivers to communicate with the database, and these drivers are independent services that need to be installed on the *customer web server*. These drivers are also different for different databases.

On a *customer web server*, the database drivers are already installed if database communication has been used. Some server side scripting languages have optimized support for specific database systems which means that no database access API is needed for communication and therefore no drivers are needed.

The platform adaption problem however is not the main problem whit database integration. The main problem is to adapt the *server side script* to the structure of the *customer database* which of course varies a lot from database to database.

For the *server side script* to be integrated with the database it need to know the structure of the database. This can be done in two ways, reading the structure from the database itself or having the *customer* manually describe the structure. Reading the structure directly from the database is much less error prone, as the whole process is automatic. But both alternatives must still be supported to accommodate all *customers*, since connection to the database isn't always possible. Both of these alternatives require that the *customers* have database connection information knowledge. The information needed to connect to a database is the location of the database (host, port), the driver to the database, the user name and password to the database. When entering the structure manually *customers* also need to know the database's exact structure.

The structure of a database can be very complicated for large systems, but as there are very strict rules on how to structure the database this is of little importance. So by following all the rules of the database structure the solution will be scalable. The core of the database is its relations which are built up using keys. The database relation structure is needed for inserting the data correctly into the database. Because of the relation dependencies, data sometimes

must be inserted in a specific order. This means that the *form builder application* needs to know the database relation structure.

To learn the structure of a database the form application system must read the metadata information from the database. There is a standard protocol which is a part of the SQL standard that can be used to retrieve metadata from a database. This part of the SQL standard however is not very well supported by many of the larger database systems which instead implements their own way of retrieving the metadata. This means that to support automatic learning of the database structure, the *form builder application* need to be able to adjust to different databases.

13 Evaluation of existing form builder applications

There exist a number of *form builder applications* online and all of them have very similar ways in handling forms both in the construction phase as well as integration phase. Four of these applications were evaluated more thoroughly.

Wufoo - <http://www.wufoo.com>

FormSpring - <http://www.formspring.com>

Frevvo - <http://www.frevvo.com>

Comfact designer - <http://designer.hubbus.com>

13.1 System integration

The main features of almost all the evaluated *form builder applications* lies under the form to web server integration category. Of the evaluated *form builder applications* there was only one exception to this. The exception is Comfact designer which falls under the no integration category. All forms developed by these *form builder applications* handle submission data in the same way, by storing it in the *form builder application database*.

No integration

This level of integration is supported by all evaluated *form builder applications* and in the case of Comfact designer, this is the only level that is supported. In order to access forms at this level, users must navigate to the *form builder application web server*. The most common way of achieving this is to provide a link directly to the form. Some *form builder application systems* also supports access restrictions which forces users to log in before accessing forms.

Form to web server integration

The form to web server integration is also widely supported among the *form builder applications*. The difference between this level and the no integration level is that *end users* can access forms from the *customer's web server*. The evaluated *form builder applications* offer a few integration alternatives to the *customers*. These integration alternatives includes form download and form embed. The form download alternative gives *customers* a web page containing full client side source code that can be placed on their own web server. The embed alternative is when forms are placed within an existing web page which can be done by the use of embed, iframe or script tags.

As mentioned before there is an alternative of *system integration* which requires *customers* to install a copy of the *form builder application system* on their own system. This feature is supported by both Comfact designer and Frevvo.

None of the *form builder applications* found has a higher level of support than the form to web server integration.

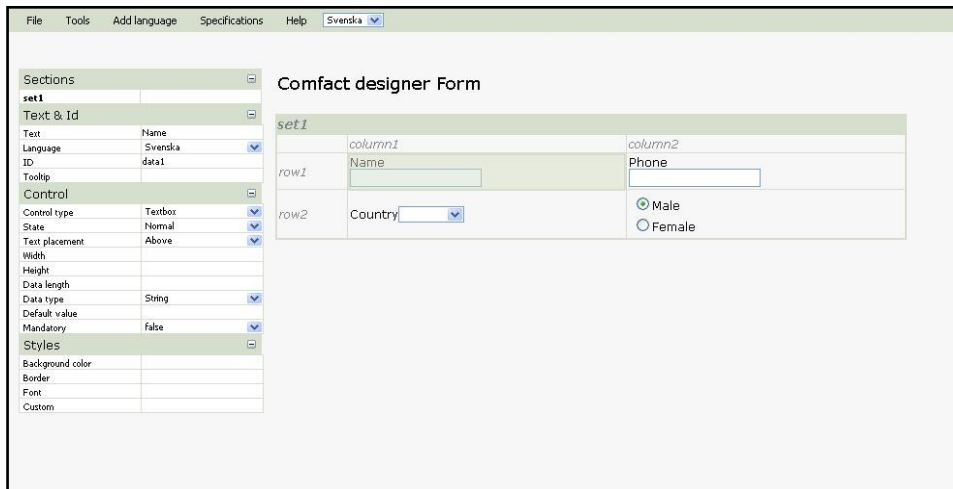


Figure 13-1 Comfact designer overview

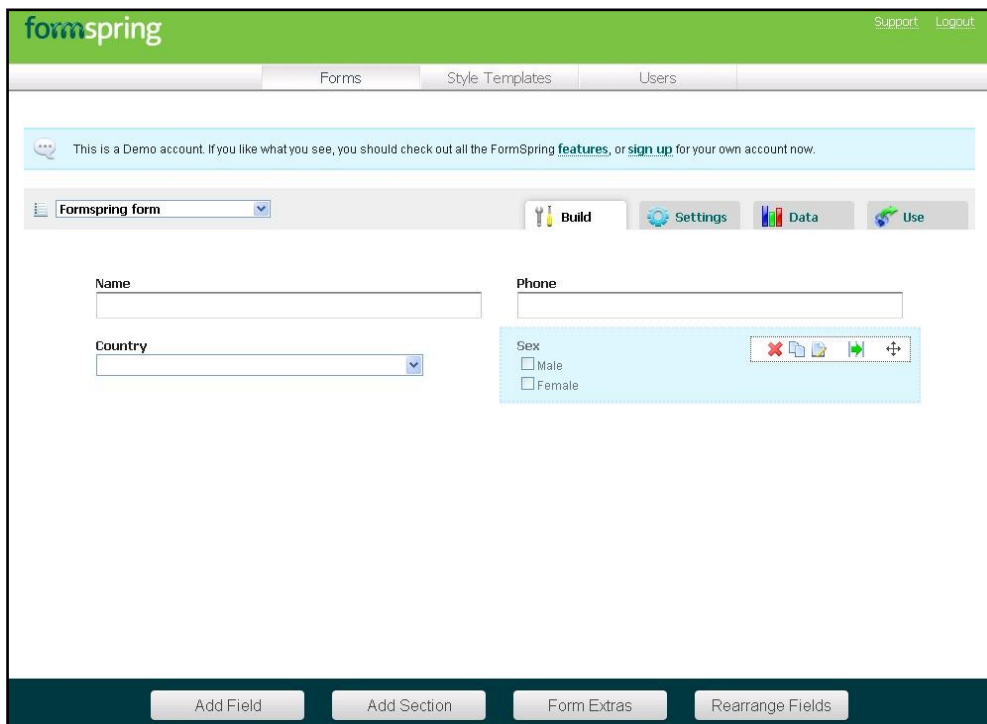


Figure 13-2. FormSpring overview

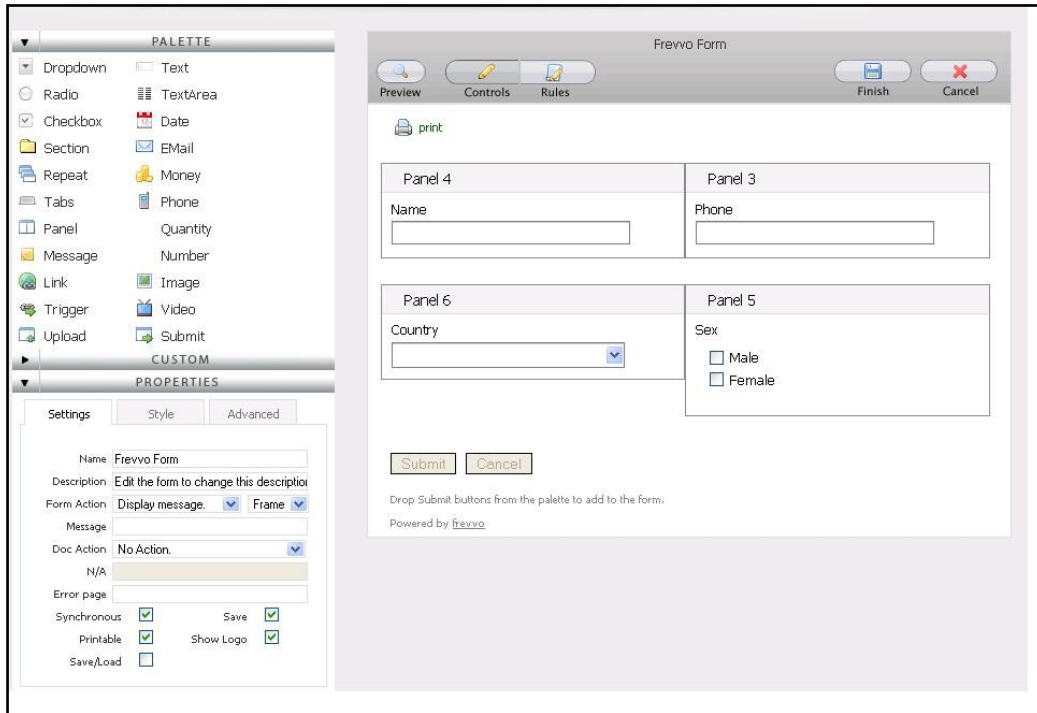


Figure 13-4 Frevvo overview

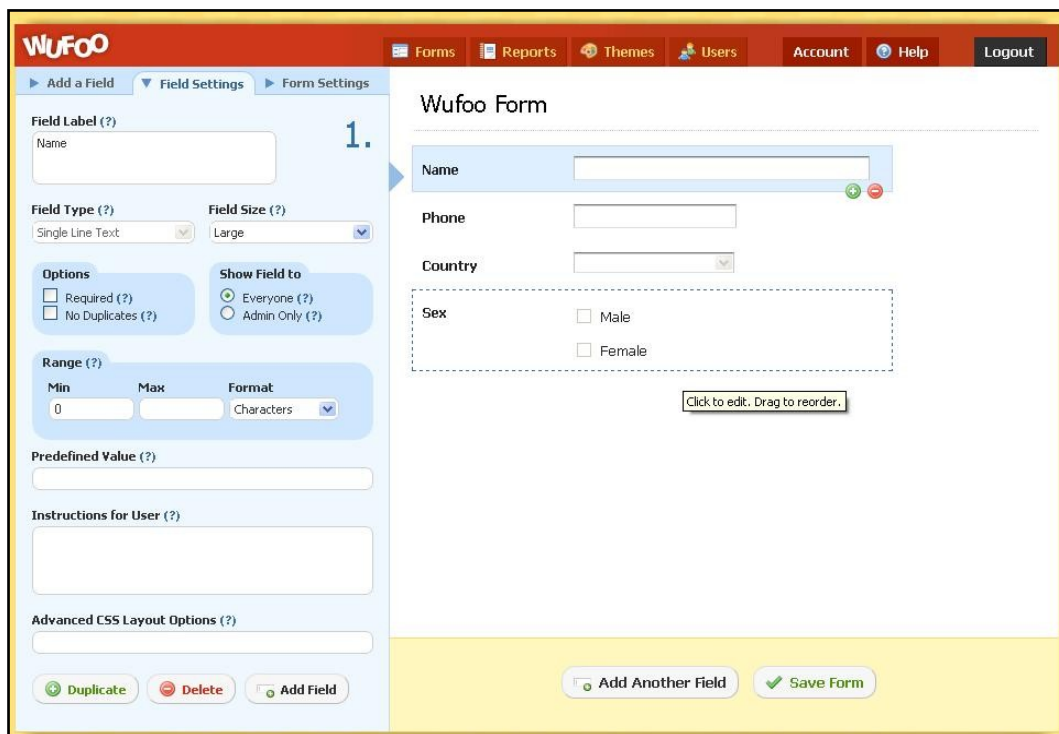


Figure 13-3 Wufoo overview

14 JavaScript

JavaScript is the main language used in the development of the *form builder application* implemented to solve the *system integration* issue. This chapter gives a short description about the language and also highlights a few useful features for developing larger JavaScript projects.

JavaScript is used to create dynamic web pages and executes on the client browser. It is often used to only enhance small parts of the web page, like visual changes in real time. This is often done using small snippets of JavaScript code that is applied to the web page. Using JavaScript for small tasks is a fast and effective way of manipulating a web page, but when creating a larger project mainly built using JavaScript, the code can easily become unstructured. The reasons why JavaScript can be messy in large projects can be many. One of the reasons is the lack of type checking which forces the developer to keep track of all variables. One of the major problems is the browser incompatibility which can require different solutions for the same problem. This can lead to large if statement code which often makes little sense and may be hard to read and understand.

14.1 Object oriented JavaScript

JavaScript is not an object oriented language but it is still possible to use JavaScript as if this was the case by simulating it. This is achieved using the keywords `function` and `new`.

```
function Class1()
{
    var counter = 0;
    this.getCounter = function()
    {
        return counter;
    }
}

var class1 = new Class1();
var counter = class1.getCounter();
```

14.2 Namespaces in JavaScript

One problem that can occur when using pre-built JavaScript libraries or toolkits is name collisions. This problem is solved in other programming languages using namespaces which doesn't exist in JavaScript. As with object orientation in JavaScript namespaces can be simulated. This is done using global objects. Here is an example on how to simulate a very simple namespace in JavaScript.

```
var namespace1 = {};  
namespace1.counter = 0;  
  
var namespace2 = {};  
namespace2.counter = 0;  
  
namespace1.counter++;  
namespace2.counter++;
```

In this example two namespaces are created (ns1, ns2) each with a variable named counter. The variables are accessed through the namespace objects. Both counter variables will have the value 1.

It is also possible to simulate nested namespaces if needed, by creating namespace objects within an existing namespace.

```
namespace1.nestednamespace = {};  
namespace1.nestednamespace.counter = 0;
```

There is also an option to add functions to namespaces which can be used to add simulated classes to namespaces, as seen in Object oriented JavaScript.

```
namespace1.inc = function()  
{  
    namespace1.counter++;  
};  
  
namespace1.EmptyClass = function(){};  
  
var newEmptyClass = new namespace1.EmptyClass();
```

Using these methods the developer can protect it's code from name collisions when using other libraries or toolkits. This also helps the developer to structure the code and separate it logically which can increase the structure of the code significantly.

14.3 Debugging JavaScript

When writing large projects independent of programming language, one important part is debugging. This can be especially difficult in JavaScript as it is a dynamically typed language and there is no help from a compiler.

Another difficulty with debugging JavaScript is the implementation differences between browsers. This means that debugging must be done in all browsers for the project to achieve full browser support.

There are some useful debugging tools for JavaScript in almost all of today's larger browsers. These tools are essential when developing and testing large JavaScript projects as well as smaller JavaScript snippets. There are also some aids that control that the syntax is correct and almost work as a small compiler.

15 AJAX

AJAX stands for Asynchronous JavaScript and XML and is used in web pages for retrieving data from the server without reloading the web page which will make the web page faster and act more like a desktop application. Some known web sites that use this functionality are Gmail, Google Maps and Facebook. AJAX can for example be used for real time *validation* of form input data, to give text proposals when *end users* start writing text in text fields.

The *form builder application* developed during this thesis will use AJAX for server communication to avoid page loads.

16 Architecture

To solve *system integration* a test form builder application (test application) was developed. When it comes to server technologies the difference in functionality is small and the choice is of small importance. The test application was implemented using .NET and C# as server technology and the client side where implemented using *HTML*, JavaScript and AJAX. The database used by the test application is of type MySQL.

17 Design

The purpose of test application is to handle the *system integration* feature that other *form builder applications* don't support. Before implementing the test application a few important decisions regarding what platform to run on as well as the languages and technologies to use had to be considered. When looking at other *form builder applications* the majority of them focus on JavaScript and AJAX. These technologies will also be the focus for the test application, but the focus, in comparison to other *form builder applications*, is more centered around JavaScript and less on AJAX and server communication. The reason for this is to avoid communicating with the server at every step of the building process and by this increase the application speed.

This chapter describes the different design parts of the test application and is divided into three parts. Each part describes one major design area.

17.1 Data representation

As the test application is going to store forms, both completed and uncompleted, a structured format that could represent the form is needed. The complete form will be exported to *HTML* and JavaScript, but a way to store the information that can represent a form before it is exported is needed. The form should be stored in such a way that it can be easily analyzed and modified. The combination of *HTML* and JavaScript is not well suited for these specifications so another format to store the not yet exported forms had to be developed. So an XML structure was developed for this purpose. The XML structure is a representation of a form used by the test application during form development.

In order for the database integration to work the XML structure store connection information, the structure of the database as well as the connection between database fields and form controls.

Here is a very small example of how the database structure could look like.

```
<Database name="Database" databaseType="MySQL" username="username"... >
  <DatabaseTable name="users">
    <DatabaseColumn name="username" isPrimaryKey="true" isForeignKey="true".../>
    <DatabaseColumn name="email" isPrimaryKey="false" isForeignKey="false"... />
    <DatabaseRow>
      <DatabaseCell columnName="username" controlId="textBox1" />
      <DatabaseCell columnName="email" controlId="textBox2" />
    </DatabaseRow>
  </DatabaseTable>
</Database>
```

17.2 Test application client

The test application was designed to be as fast as possible and also to have the look and feel of a desktop application. This is achieved by using as much JavaScript as possible and by placing almost the whole application on the client. As most of the application is placed on the client, both GUI code as well as most of the application logic is in JavaScript.

On the client, the *XML* file is converted to JavaScript objects which are modified by the application during the form building process. The main usage for these objects is to construct the GUI of the form by generating *DOM* elements which are inserted into the *DOM* tree. These objects can also be converted back to *XML*.

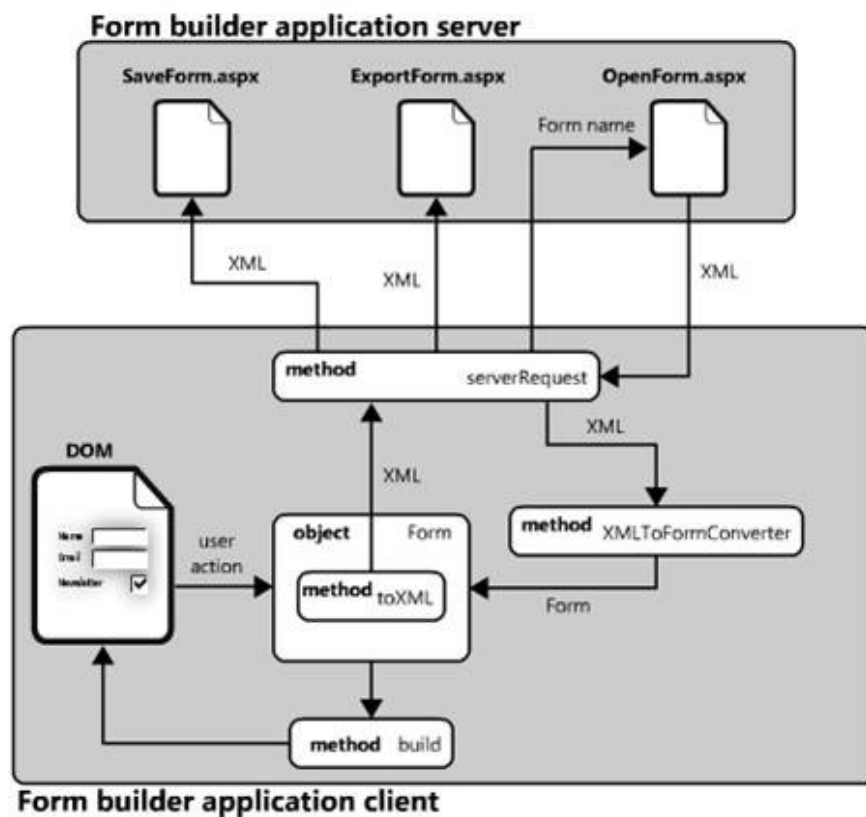


Figure 17-1. Test application client

17.3 Test application server

The test application server consists of a number of server side scripts implemented as aspx. The test application client communicates with the test application server with the use of AJAX calls. Many of the server side scripts only perform small tasks which include communication with the test application database.

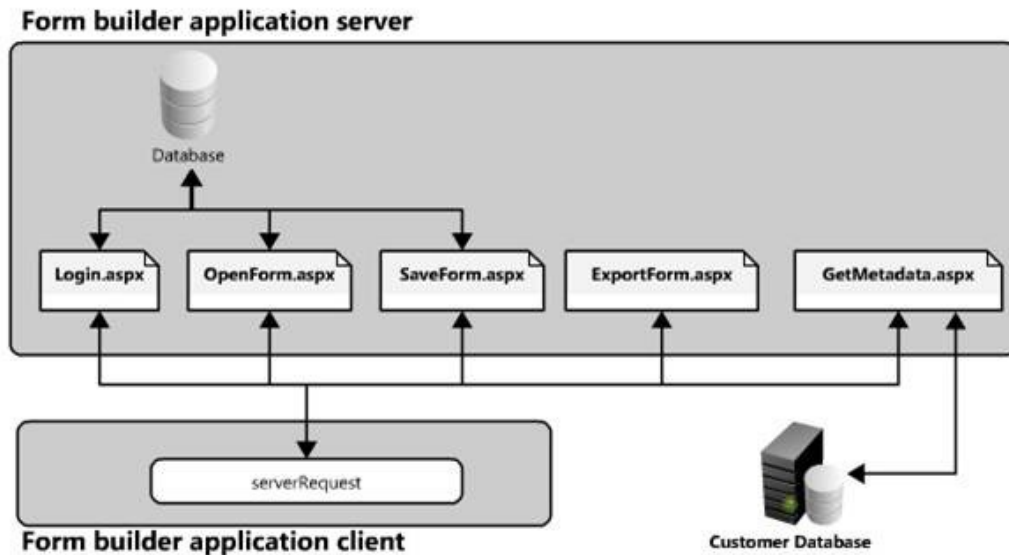


Figure 17-2. Test application server

18 Implementation

18.1 Code generation

In order to support web server integration the test application need to generate the form as well as its *server side script*. For the form to be fully functional it also must be able to generate the *server side script* in different languages depending on the *customer web server* platform.

When exporting, the generation of the form and the *server side script* are done separately. All generation is done using the information in the *XML* file.

Form generation

The form generation includes generating the *HTML* and the JavaScript needed. The *HTML* and JavaScript generation will be done in the same way independent of the *customer web server* platform since the form is handled by web browsers. First all tags in the *XML* that describe the visual part of the form will be used to generate a *HTML* string. For the more advanced functionality, like *Equation* and *validation*, JavaScript code is generated and added to the *HTML* string.

Server side script generation

When generating the *server side script* the type of web server platform where it should be placed must be taken into account. The *server side script* will be generated as a string and the string will differ depending on the web server platform. The test application will, like with the form, use the information in the *XML* file to generate the sever side script string, but only the information from the Database tags will be used.

The first though was to have the form as a separate *HTML* file that communicated with the *server side script*, but since all *server side scripts* supports *HTML* code as well, the form code was moved into the *server side script*. The test application now only generates one file that represents the form which includes both the form *HTML* code and the *server side script*.

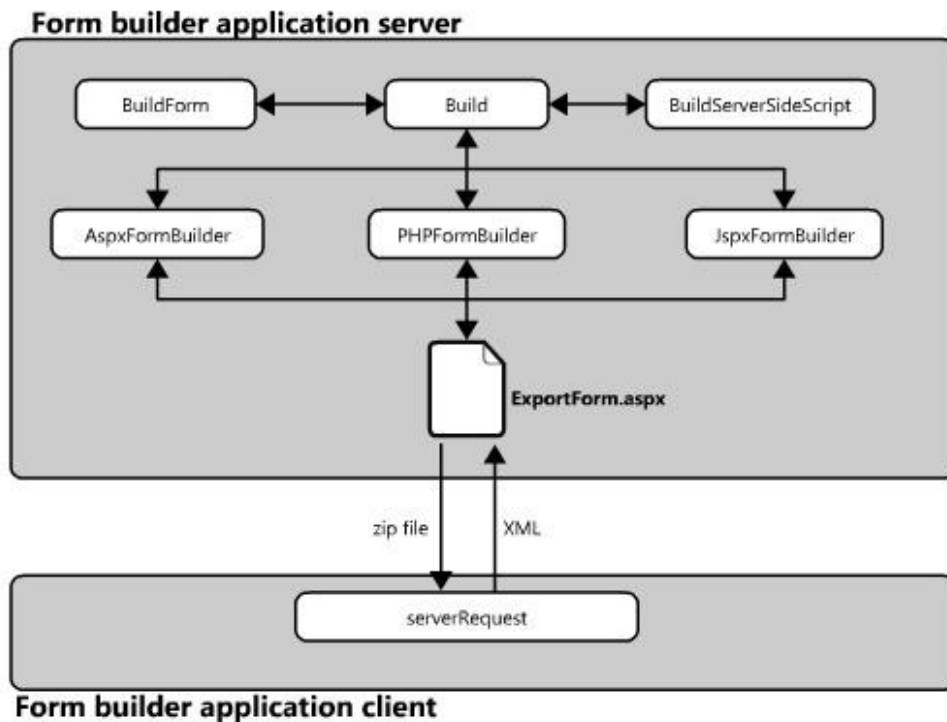


Figure 18-1. Code generation at test application server.

Database connection and analyzing

To read and analyze the structure of a database automatically, the connection information is needed. *Customers* are prompted for most of the necessary connection information and use this to construct a *connection string* which is used to connect to the *customer database*. If the connection was successful the metadata is read to obtain the structure of the database. The metadata is then used to create an *XML Database tag* that will be added to the *XML structure*. *Customers* can then select different form controls and associate them to the appropriate database column which modifies the Database tag accordingly.

Key sorting

When inserting data into a database, the data must be inserted in a correct order which depends on the relations of the columns. Therefore the *DatabaseRow tags* are sorted before generating the *server side script*. The key sorting algorithm implemented in the test application looks for these dependencies and sorts them accordingly so that there will be no insertion errors.

19 Discussion and conclusion

The purpose of this thesis was to present a solution to the *system integration* issue that no form builder application currently handles. The method used to solve this was an implementation of a new form builder application which could handle *system integration*. The implementation of the test application was successful and even if the implementation is in an early stage the *system integration* issue has been solved.

20 Appendix A – Implementation issues

20.1 Data representation

One of the first discussions was about the underlying structure that would represent a form during development. As mentioned in the Implementation section an XML based protocol was created for this purpose. Before XML was chosen however, some other alternative formats that would be well suited for data representation was considered. The three formats considered were SQL, XML and JSON.

SQL

Using SQL gives a lot for free, like speed, structure, security and wide support. SQL could therefore be used for the exact this purpose. The problem is that most of the work is done at the client side of the application, which means that the form data must be sent between client and server and SQL isn't well suited for easy distribution. SQL is built for extracting specific data from its structure which is needed, but this is needed on the client side and a SQL database is located on the server side of the application. So what is needed is a format that can represent a form in a structured way so that specific data can easily be extracted on the client side.

XML

XML is good for representing data in such a way that the test application could make use of it. XML is well structured and can be read easily. The forms can with XML be separated into single XML files. XML can be handled in many different environments and be easily distributed because it's a stand alone document that does not rely on an engine to work like SQL. Two important things with XML is the support in the languages chosen to implement the test application. Both .NET and JavaScript has good support for XML.

JSON

JSON is many ways similar to XML and as with XML, JSON can store forms in different files. Since the test application is going to use a lot of JavaScript and AJAX, JSON would be a perfect way of storing the forms as JSON is very easily handled with JavaScript.

The choice was between XML and JSON as they are very similar and both fitted the needs of the test application. The decision to go with XML was made simply because XML has wider support and is more recognized.

20.2 Database connection

In order to connect to a database using a *connection string* the following information is needed:

- *username* - the user login name for the database
- *password* - the password associated with the username
- *host* - the address to the database (name or IP)
- *port* - the port number that the database is listening on
- *driver* - what driver to use when connecting

For some *customers* this information can be difficult to acquire, especially the port number and the driver. The first three (username, password and host) is information that is very *customer* specific and must be supplied by the *customer*. The port and driver however can both be database specific so the process of retrieving them can in most cases be done automatically. The amount of drivers is limited so they can all be tested for connectivity to determine the correct driver. The different database types use standard port numbers that will be used by default. The port can however be set manually by the database administrator (*customer*) and in that case there is no effective way of automatically choose the port since all ports would have to be tested. So if the port has been set manually, the *customer* must specify the port as well as the host, username and password.

To make this as user friendly as possible the test application will have automated test for ports and drivers whereas the host, username and password always must be supplied by the *customer*.

Metadata

Acquiring metadata isn't a trivial task when it comes to databases because of the differences in the implementation between the database systems. There is a section in the SQL standard that describes how to obtain metadata from a database using the *INFORMATION_SCHEMA*. The *INFORMATION_SCHEMA* is a built in group of tables containing metadata about the database. The problem here is that the support for this part of the SQL standard is not widely implemented by the different database systems. The systems which do support this standard still have differences in their interpretation. Systems that don't implement this standard have often implemented their own way of obtaining metadata from their database. So even if there is a standard for obtaining metadata from a database, each individual database system must be thoroughly analyzed so the test application can read metadata from all different systems.

20.3 Iframe URL length

The test application use AJAX for transmitting data from the client to the server. A problem was encountered during the exportation of the form. When exporting a form, the test application client sent the full *XML* structure as a parameter in the iframe URL. The *XML* was handled at the server by generating the form and the *server side script* and the test application server then sent them back to the *customer* as a compressed file. The problem, which only occurred in Firefox and when the URL exceeded 2048 characters, was that the compressed file was blocked for download for unknown security reasons. Because of this the test application had to use AJAX to send the *XML* to the server first and then fetch the compressed file using the iframe with an URL without parameters.

21 Appendix B – Test application server scripts

SaveForm.aspx

This *server side script* receives the *XML* from client through the URL as a parameter and stores it at the correct place in the database.

OpenForm.aspx

After receiving a form name from the client, the correct *XML* file is read from the database and is sent back to the client.

BrowseForms.aspx

Reads all form names, for a specified *customer*, from the database and sends them to the client along with other information about the form.

ExportForm.aspx

This *server side script* receives the *XML* from the client in the same way as *SaveForm.aspx* does. Then it uses the *XML* to generate a form as well as the *server side script* associated with the form. The generated form and *server side script* is sent back to the client as a compressed file.

GetDatabaseMetadata.aspx

This *server side script* receives information about connecting to a specific remote database. The *server side script* uses this information to connect to the specified database and read its metadata. It then constructs an *XML* structure from the metadata and sends it back to the client.

Login.aspx

After receiving *customer* login information, this *server side script* validates the information against the login information stored in the database. If the login information given by the *customer* is correct the *customer* is logged in.

22 Appendix C – XML database

Database

The Database tag holds all the mentioned information for one database. The XML structure can have one or many Database tags which make it possible to connect one form to multiple databases. The Database tag's attributes holds all connection information needed. It also contains zero or more children of the XML tag DatabaseTable.

DatabaseTable

The DatabaseTable represents a table in the database. It can contain two different child tags, DatabaseColumn and DatabaseRow. It also has the table name as an attribute.

DatabaseColumn

The DatabaseColumn corresponds to a column in a database table and holds database relation information data. This includes what data type the value of the column is, what type of key, if any, the column is and also the reference information if the key is of the type foreign key.

DatabaseRow

The DatabaseRow tag represents one insertion into the database table. There can be zero or more DatabaseRow tags inside a DatabaseTable so that it can be possible to insert multiple rows in the database table from a single form. DatabaseRow contains one or more DatabaseCell tags.

DatabaseCell

This tag describes the database field to form control connections by having two attributes, columnName and controlId. The columnName is the name of the database column in the database table where the value from the control with id equal to controlId will be stored.