# LoanAPound Guide

## Fairsail Technical Test

Iain MacDonald – 17/10/2017



Table of Contents

[LoanAPound Guide 1](#_Toc497292418)

[Fairsail Technical Test 1](#_Toc497292419)

[Requirements matrix 3](#_Toc497292420)

[Installation instructions 5](#_Toc497292421)

[Browsing, building and running the code in a development environment 5](#_Toc497292422)

[Database 5](#_Toc497292423)

[Installation to IIS 6](#_Toc497292424)

[Data model 7](#_Toc497292425)

[Summary of the code 8](#_Toc497292426)

[Testing 9](#_Toc497292427)

[Miscellaneous notes 12](#_Toc497292428)

# Requirements matrix

In the description of the Fairsail Developer Test, there are a number of requirements. The table below summarises them, and describes how each of them is met by this solution.

|  |  |
| --- | --- |
| Requirement | How requirement is met |
| **General solution requirements** | |
| It should be an object model solution. | It is an object model solution. The “Summary of the code” section in this guide discusses the basic architecture. |
| There should be a service layer. | The solution is a layered solution, consisting of SQL Server database, data access layer, business logic layer, and web-enabled service layer. |
| Tests should be included. | A series of manual tests is described in the “Testing” section of this document. |
| Installation instructions should be included in the documentation. | There is an “Installation instructions” section in this guide. |
| The data model should be included in the documentation. | There is a “Data model” section in this guide. |
| The core logic/backend should be in Java/C#/Salesforce. | The logic is in C#. |
| The frontend (if any) should be in JavaScript (any framework). | There is a frontend test harness web page, which uses JavaScript/jQuery. |
| All code must be submitted with GitHub. | The code is on GitHub. |
| **Required Tasks** | |
| Design a data model to support the LoanAPound solution. | Done. |
| Implement a Credit Check Service (user stories 3 and 4) and a basic UI to test it. | See next 3 requirements below. |
| *User story 3* As a loan engine, I want to get a credit score from a third party system, so that it can be used to inform whether or not the applicant will get his loan accepted or rejected. | The code and data model support “plugging in” third-party credit score providers. It is anticipated that in “real life” these credit score providers would expose various APIs, each of which would need code written to integrate to. Since the APIs are unknown, it may be that the stored Applicant data in this implementation is not sufficient; in real life, a superset of the applicant data required by all of the credit score providers would need to be stored. The exact information would be specified in advance of implementation. Three example plug-ins are included in the solution, to show how the mechanism for including multiple credit score providers would actually be implemented.  These credit score providers can be called to obtain a credit score for an applicant. |
| *User story 4*  As a loan engine, I want to be able to implement multiple credit score third party system, so that I can chose to use any of them based on various criteria. | It is not absolutely clear from this requirement what kind of criteria are anticipated for selection of the credit score system, and in “real life” this would be bottomed out before design and coding. The solution includes the ability to query for a list of available credit score providers, and also to obtain credit scores from a subset of those providers for any particular client. However, no mechanism for actually making the choice of credit score engine(s) has been provided. |
| *Basic UI to test the Credit Check Service* | Both of the methods that the credit check service requires can be tested from the provided test web page, by calling them with different parameters and observing the output |
| **Other user stories and additional requirements** (These are features which the system should support but which do not require implementation as part of this solution) | |
| *User story 1*  As a LoanAPound administrator, I want to be able to setup different types of loans with criteria such as (but not limited to) term/years, borrowing rate, provider… | The data model supports the setting up of multiple loans (i.e. types of loans). It is not 100% clear from the user story, but it sounds as though there is a requirement to invent ad-hoc criteria as part of the solution, and that these may or may not be attached to any particular loan type. (The exact requirement would be clarified with the user in real life.) The data model therefore supports setting up criteria. In this solution, these are restricted to three types (string, number or date), but in a “real life” situation, the requirement would have been pinned down further to determine whether these types are sufficient.  Types of loans could then be configured to have criteria of specified values attached to them. (e.g. Provider – AcmeBank; Term in months – 3; Rate – 15%; Loan offer end date – 31/12/2017) |
| *User story 2*  As a loan applicant, I want to be able to apply for a loan online so that I get a decision on money being lent to me or not | The data model supports this: applicants and applications are stored. There is a mechanism for credit scoring. |
| *User story 5*  As loan engine, I want to be able to approve the loan or not based on the credit score&loan amount so that I can refer it to the underwriter decision or not | This is possible with the data model. Credit scores for applicants, and loan amounts for applications are stored (as are the applicants applying), so that code could be written to approve the loan according to specified business logic and referred to the underwriter if necessary. |
| *Additional requirement 6*  As an applicant, I want to be able to view online the progress of my application | The data model supports this. Application statuses and the dates that they reached those statuses are stored. So code could be written to allow applicants to see online their application’s progress, represented by the flow through status stages. |
| *Additional requirement 7*  As an underwriter, I want to see a queue of pending applications so that I can approve or reject them | The data model supports this: applications are stored (and who has applied for them), as are the statuses and date when the applications reached those statuses. So code could be written to allow the underwriter to see, for example, applications that have already been credit scored ordered by date of when the applications were initially started (or indeed when they were credit scored). |
| *Additional requirement 8*  As a loan engine, I want to notify by email the applicant of the progress of its application, so that he is aware of what is happening | The data model supports this: the email address of the applicant is stored, as is the application’s status. Code could be written to inform the applicant of changes to the status of their application(s) by email. |

# Installation instructions

## Browsing, building and running the code in a development environment

To build the LoanAPound code and run it in a development environment, Microsoft Visual Studio is required. The code on GitHub can be built with Visual Studio Community Edition 2017, which is freely downloadable from here:

<https://www.visualstudio.com/vs/visual-studio-express/>

You should run the development environment as Administrator to allow publishing of the web application to IIS.

## Database

A SQL Server database is required. The solution has been tested against Visual Studio 2017 Express, a free download available from here:

<https://www.microsoft.com/en-us/sql-server/sql-server-editions-express>

The “Basic” installation can be selected. At the end of the installation, there is an option to additionally install SSMS (SQL Server Management Studio), which should be done. The SSMS download actually comes from here: <https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms>

There is a script in the database folder on GitHub to create the database structure and populate the database with some example data. This script should be opened and run in SQL Server Management Studio.

The connection string in C:\temp\LoanAPound\WebServiceLayer\bin\LoanAPoundDataAccessLayer.dll.config should be set to point to the SQL Server database that has been created, replacing the machine name and SQL Server instance, like this:

<LoanAPoundDataAccessLayer.Properties.Settings>

<setting name="ConnectionString" serializeAs="String">

<value>Data Source=**<Machine name>\<SQL server instance>**;Initial Catalog=LoanAPound;Integrated Security=True</value>

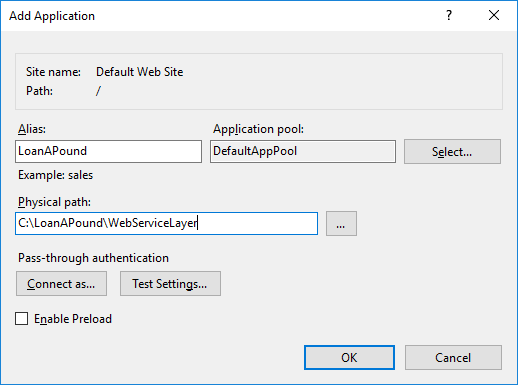
</setting>

</LoanAPoundDataAccessLayer.Properties.Settings>

## Installation to IIS

To deploy LoanAPound externally to Visual Studio, IIS (Microsoft Internet Information Services) is required. This can be turned on as a feature of Windows.

To deploy LoanAPound to IIS, right click on the Default Web Site node, and choose “Add Application”. Choose an alias of “LoanAPound” and navigate to the “WebServiceLayer” directory downloaded from GitHub (in the iis directory).



A second web application should be created to contain the test harness. So, in IIS, right click on the Default Web Site node, and choose “Add Application”. Choose an alias of “LoanAPoundTest” and navigate to the “TestHarness” directory (in the iis directory) downloaded from GitHub. (The solution anticipates the default security settings and users are in place in IIS and SQL Server. If this is not the case, it is beyond the scope of this document to describe how to restore them.)

# Data model

The data model is designed to support the requirements and user stories. Foreign keys have been created as appropriate, but no indexes have been created – these would be required in a production environment for performance.



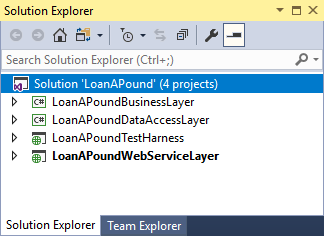
Result

The function of most of the tables will be self-explanatory, but these explanatory notes may be helpful:

* The **Applicant** table contains applicant data. In a real system, there is likely to be more required information, and this would be specified before implementation. For credit scoring, it is usual to supply a list of previous addresses for an applicant, so the schema would probably be expanded to include an **Address** table, which would include dates of residence.
* **Status** is just a reference-data table containing the possible statuses of an application. Dummy data has been inserted for this exercise (although it is not actually used).
* The three **CriterionValue** tables are designed to allow criteria of three different types (string, numeric, date), and they are all joining tables between **Loan** and **Criterion**. The **CriterionType** table lists those three types and is present to aid construction of queries.
* The **CreditScoreEngine** table contains the .NET type names of the classes that integrate to credit score provider APIs. In this exercise, there is no real integration and the classes simply use algorithms based on the applicant’s name and email address to assess whether they are worthy of credit.
* In a real system, a log of when emails have been sent out may well be desired by the business. There is no table for storing such a log in this schema, as it was not explicitly mentioned as a requirement in any of the user stories.
* The **CreditScoreResult** table stores the result as a single numeric value. It may be that this is not actually the result type supplied by all engines. During proper specification of the system, the nature of the results would be pinned down.

# Summary of the code

There is a single solution containing four projects. These are compiled into separate .NET assemblies, emphasising the separation of concerns.



The projects are:

* **LoanAPoundDataAccessLayer** – uses ADO.NET to access the SQL Server database. Only queries required for this exercise have been implemented. The static Database class implements the required methods. The DataObjects file contains light-weight (method-free) data-object classes corresponding to the data in the tables.
* **LoanAPoundBusinessLayer** – contains classes pertinent to the business logic of the application. It contains “managers” (Applicant manager and Credit score manager) to orchestrate sub-elements of the solution. It is envisaged that in a full solution, further managers would be implemented to introduce the required functionality. The business layer also includes classes corresponding to the entities required for this exercise (Applicants, Credit score engines and Credit score results). Finally, it implements a mechanism for extra “plug-in” integrations to third-party credit score systems. Each integration has to implement the ICreditScoreEngine interface, but its type name is stored in the database, and this would potentially allow separate assemblies to be written containing the necessary implementation code.
* **LoanAPoundWebServiceLayer** – provides an external interface to the system. It is essentially a thin façade around methods provided by the credit score manager. It is envisaged that it would be expanded to provide access to additional functionality as this is added to the system. It uses ASP.NET Web API as a quick-and-easy way to implement a REST-ful web service over HTTP that can consume and return data in JSON format from and to clients.
* **LoanAPoundTestHarness** – this is an ASP.NET project, but in reality it consists of just a single HTML page which contains the UI for the test harness (i.e. there is no server-side code). Calls from this client requests are achieved using the jQuery JavaScript library to implement AJAX.

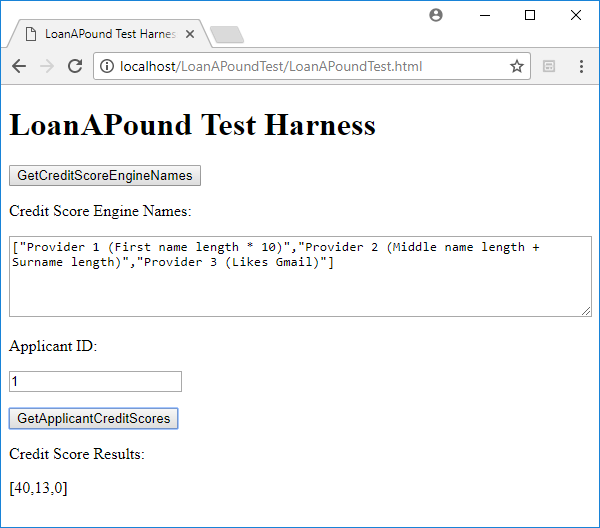
# Testing

The following tests should be carried out using the supplied LoanAPoundTest.html test harness. (This will have URL <http://localhost/LoanAPoundTest/LoanAPoundTest.html> if IIS has been configured as above.)

The test harness uses jQuery AJAX to call the two exposed methods in the WebServiceLayer, namely:

* **GetCreditScoreEngineNames**, which takes no parameters
* **GetApplicantCreditScores**, which takes two parameters: a list of credit score engine names to be used for the credit check, and the id of the applicant to be checked

The test harness has text fields where these parameters can be supplied. Furthermore, the return result of **GetCreditScoreEngineNames** is written directly into the text box that is the input for the list of credit score engines for the **GetApplicantCreditScores** call. This means that correctly formatted JSON is available for manipulation for the call to **GetApplicantCreditScores**.



The envisaged sequence of actions when testing is therefore:

* Click **GetCreditScoreEngineNames** button.
* Examine returned JSON in **Credit Score Engine Names** text box.
* (Optionally) change content of **Credit Score Engine Names** text box.
* Enter desired input for applicant in the **Applicant ID** text box.
* Click **GetApplicationCreditScores** button.
* Examine returned JSON in **Credit Score Result** area of the page.

The Applicants, with their IDs that have been prepopulated in the database are as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | Title | FirstName | MiddleName | Surname | EmailAddress |
| 1 | Mr | Adam | Alan | Arkwright | adam@yahoo.com |
| 2 | Mrs | Beatrice | Bianca | Baines | bbb@gmail.com |
| 3 | Mr | Colin | Carl | Crighton | c\_c\_c@yahoo.com |
| 4 | Mrs | Dianne | Doris | Dawes | d.dawes@b.com |
| 5 | Dr | Eric | Edward | Engels | eric@outlook.com |
| 6 | Dr | Fiona | Fatima | Fontaine | fi@gmail.com |
| 7 | Mr | Gareth | Gordon | Griffiths | garethgriffiths@yahoo.com |
| 8 | Miss | Hannah | Hazel | Haynes | hh@outlook.com |
| 9 | Mr | Igor | Ian | Irvine | igor.irvine@yandex.com |
| 10 | Miss | Jane | Joanne | Jones | JJJones@gmail.com |

The rules that the three dummy credit score engines are coded to apply are:

|  |  |
| --- | --- |
| Credit Score Engine Name | How credit score is calculated according to applicant |
| Provider 1 (First name length \* 10) | FirstName.Length \* 10 |
| Provider 2 (Middle name length + Surname length) | MiddleName.Length + Surname.Length |
| Provider 3 (Likes Gmail) | EmailAddress.Contains("gmail")? 100 : 0 (i.e. 100 if they have the string “gmail” in their email address, but 0 otherwise) |

Here is a list of tests to be carried out:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test | Input(s) | Expected output |
| 1 | Calling GetCreditScoreEngineNames returns expected JSON | N/A – just click GetCreditScoreEngineNames button | The Credit Score Engine Names text box is populated with the names of the three dummy scorers (note that their names actually contain a brief description of how their credit score engines work for convenience of testing them):  ["Provider 1 (First name length \* 10)", "Provider 2 (Middle name length + Surname length)", "Provider 3 (Likes Gmail)"] |
| 2 | Calling GetApplicantCreditScores gives expected results for applicants (This can be repeated for all 10 applicants. The database can be changed or supplemented to do extra tests.) | Enter application id in Applicant ID field.  Leave full returned JSON in the Credit Score Engine Names text box. Click GetApplicantCreditScores button. | A list of three scores corresponding to the three credit scorers should be returned. E.g. for Applicant 1 (Adam Alan Arkwright, [adam@yahoo.com](mailto:adam@yahoo.com)):  [40,13,0]  Length of “Adam” is 4, hence first engine returns 4 \* 10 = 40; Lengths of “Alan” + “Arkwright” = 4 + 9 =13;  Email address is not Gmail, so third result is 0. |
| 3 | Calling a subset of the engines only returns a subset of results (This can be repeated for a selection of combinations of credit engines and applicants) | Edit the Credit Score Engine Names text box so that it contains only one or two of the names, e.g. ["Provider 1 (First name length \* 10)", "Provider 3 (Likes Gmail)"] | Result corresponds to credit score engines in the input, e.g.  [40,0]  For applicant 1. |
| 4 | Calling no engines results in an empty return set of scores | Edit the Credit Score Engine Names text box so that it contains empty JSON, i.e. [] | Return value is empty, i.e. [] |
| 5 | Supplying defective JSON results in no returned credit scores | Edit the Credit Score Engine Names text box so that it contains corrupt JSON | Return value is null |
| 6 | Supplying a non-existent Applicant ID results in no returned credit scores | Edit the Applicant ID so that it contains an id that is not in the database (e.g. 9999), or even a non-numeric value. Use correct JSON for the credit score engines. | Return value is null |
| 7 | Calling non-existent credit score engines gives expected value for credit score | Edit the Credit Score Engine Names text box so that it contains correctly-formatted JSON, but with some non-existent engine names, e.g. ["Provider 1 (First name length \* 10)", "Provider 1.5", "Provider 2 (Middle name length + Surname length)", "Provider 3 (Likes Gmail)", "Provider 4"] | Return value is -1 for each non-existent engine, e.g. for applicant 1:  [40,-1,13,0,-1] (The second and last names were not real engines.) |
| 8 | Calling credit score engines inserts into the CreditScoreResult table in the database | Perform credit check(s) on applicants using the test harness. | In SQL Server management studio (or Visual Studio), perform this query:  select \* from CreditScoreResult  New rows corresponding to the credit check should have been inserted into the database. (See table below for an example of query results.) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | ApplicantID | CreditScoreEngineID | DateAchieved | Result |
| 32 | 1 | 1 | 2017-10-17 14:24:37.3133333 | 40.0000000000 |
| 33 | 1 | 2 | 2017-10-17 14:24:37.3266667 | 13.0000000000 |
| 34 | 1 | 3 | 2017-10-17 14:24:37.3333333 | 0.0000000000 |

# Miscellaneous notes

* There is no logging in this solution – an omission that would need to be present in a production environment.
* The solution is not scalable as it stands: there is no multithreading, for example. Again, an enterprise-scale solution would require this.
* The calls to the host systems may be slow in real life, and therefore asynchronous techniques (to allow for parallel calls to credit score engines) would be implemented in real life, as would time-outs.