A white circle and blue circle

Description automatically generated

**Version: draft**

**Last Updated: 01/2025**

**Author(s): Benjamin Roberts**

End Point Assessment -

Level 4 Software Development

Benjamin Roberts

# Table of Contents

[Table of Contents 2](#_Toc188457227)

[Table of Figures 4](#_Toc188457228)

[Introduction 6](#_Toc188457229)

[1) Apply 1 - Java Decisioning Program 7](#_Toc188457230)

[1.1. Introduction & Requirements 7](#_Toc188457231)

[1.1.1. Project Description 7](#_Toc188457232)

[1.1.2. Acceptance Criteria/Project Outcomes 7](#_Toc188457233)

[1.2. Stakeholders Involved 8](#_Toc188457234)

[1.3. Development – Set Up Dependencies 9](#_Toc188457235)

[1.3.1. CRUD - Database Connection 9](#_Toc188457236)

[1.3.2. Standing up a Database Solution 9](#_Toc188457237)

[1.3.3. Handing Azure firewall 11](#_Toc188457238)

[1.3.4. Uploading Data 12](#_Toc188457239)

[1.3.5. Access Management 13](#_Toc188457240)

[1.3.6. Personal Data (PII) Protection 16](#_Toc188457241)

[1.3.7. Credentials within the Java program 17](#_Toc188457242)

[1.4. Development -Java 19](#_Toc188457243)

[1.4.1. User Input 19](#_Toc188457244)

[1.4.2. Data Model 21](#_Toc188457245)

[1.4.3. Credit Policy built 23](#_Toc188457246)

[1.5. Testing the program 33](#_Toc188457247)

[1.5.1. JUnit Unit Tests 33](#_Toc188457248)

[1.5.2. End to End 34](#_Toc188457249)

[1.5.3. Debugging Errors 36](#_Toc188457250)

[1.6. Conclusions 40](#_Toc188457251)

[1.6.1. Project Outcome 40](#_Toc188457252)

[1.6.2. Future Additions 40](#_Toc188457253)

[2) Apply 2 - React Web App – Decisioning Dashboard 41](#_Toc188457254)

[3) Apply 3 - Application Testing 42](#_Toc188457255)

[4) Apply 4 and 5 – DevOps Framework End 2 End 43](#_Toc188457256)

[5) Apply 6 - Microservices, Databases and API’s 44](#_Toc188457257)

[Introduction 44](#_Toc188457258)

[Discovery 45](#_Toc188457259)

[Relational Databases 45](#_Toc188457260)

[Non-Relational (NoSQL) Databases 47](#_Toc188457261)

[Current Architecture 51](#_Toc188457262)

[Strengths and Weaknesses 56](#_Toc188457263)

[Alternative – Microservice Architecture 57](#_Toc188457264)

[Proposed Architecture 58](#_Toc188457265)

[How to Implement 59](#_Toc188457266)

[Conclusion 61](#_Toc188457267)

[Additional Evidence 62](#_Toc188457268)

[Final Reflections 63](#_Toc188457269)

[Future Development 63](#_Toc188457270)

[Appendices 64](#_Toc188457271)

[Guidance (Delete this later) 64](#_Toc188457272)

[Bibliography 65](#_Toc188457273)

# Table of Figures

[Figure 1 - A Java Class called “AzureJDBC.java which handles a connection to an Azure SQL Database 10](#_Toc188457274)

[Figure 2 - A screenshot showing a Firewall rule in place on my database to only allow access from my network (with IP Addresses redacted) 11](#_Toc188457275)

[Figure 3 - Teams Messages – Uploading Test data Into Azure SQL Database 12](#_Toc188457276)

[Figure 4 - Azure Data Studio flow for loading data into a database 13](#_Toc188457277)

[Figure 5 - SQL Code showing the creating of the integration user for the Java Program to use. 14](#_Toc188457278)

[Figure 6 - SQL Views used within the Java Program 14](#_Toc188457279)

[Figure 7 - SQL Table use to log results history of the Java Program 15](#_Toc188457280)

[Figure 8 - Properties file with the credentials stored in Key-Value pairs, with sensitive information redacted 17](#_Toc188457281)

[Figure 9 - .gitignore file with a record to not include .properties files 17](#_Toc188457282)

[Figure 10 - Implementation of Java reading from an external file to set variables 18](#_Toc188457283)

[Figure 11 - Code used to define the “UserInput” class, importing the necessary Scanner class & defining 19](#_Toc188457284)

[Figure 12 - Java Console showing the Scanner Class in use to capture user input 19](#_Toc188457285)

[Figure 13 - An object method used to prompt the user for a Residential Status 20](#_Toc188457286)

[Figure 14 - the “AppData” class written to act as a Data Model 21](#_Toc188457287)

[Figure 15 - Importing and using the GSON library to create a JSON Representation to log to the console 21](#_Toc188457288)

[Figure 16 - An example of the GSON output, returning a JSON Representation of the Java Program's data model 22](#_Toc188457289)

[Figure 17 - Java Classes for both Scorecards. 23](#_Toc188457290)

[Figure 18 - View of the AppData Class constructor, which the Champion/Challenger split 24](#_Toc188457291)

[Figure 19 - NULL handling for a Scorecard characteristic. 28](#_Toc188457292)

[Figure 20 - Probability of Default formula (Silva et al, 2020) 29](#_Toc188457293)

[Figure 21 - Java Class method to calculate Probability of Default 29](#_Toc188457294)

[Figure 22- SetIndex method for the Champion Scorecard 30](#_Toc188457295)

[Figure 23- SetIndex Method for the Challenger Scorecard 30](#_Toc188457296)

[Figure 24 - Email Chain with colleague for peer review 31](#_Toc188457297)

[Figure 25 - JUnit 5 Test cases written to test individual parts of the program independently 33](#_Toc188457298)

[Figure 26 - A test case written to test a program set up class that reads properties from an external file 34](#_Toc188457299)

[Figure 27 - A demonstration of running the program within the Eclipse Software, with console output 34](#_Toc188457300)

[Figure 28 - Console output for supporting debugging the program 35](#_Toc188457301)

[Figure 29- Unit Test for testing probability of default 36](#_Toc188457302)

[Figure 30 - Unit Test on the Offers class 37](#_Toc188457303)

[Figure 31- Unit Test results showing Illegal Argument Exceptions 37](#_Toc188457304)

[Figure 32 - Java Switch case missing Break statements 38](#_Toc188457305)

[Figure 33- Correcting the Switch Case, allowing the tests to pass 38](#_Toc188457306)

[Figure 34 - Meeting Minutes for presentation on final version with employer 39](#_Toc188457307)

[Figure 35 - Example of a Monolith platform (Atlassian, 2024) 44](#_Toc188457308)

[Figure 36 - Example of tables in a relational database, where a Foreign Key "EmployeeId" is used to relate the tables (Pluralsight, 2020) 45](#_Toc188457309)

[Figure 37 - Example of data storage in a Document store, a type of non-relational database (Microsoft Learn, 2024) 48](#_Toc188457310)

[Figure 38 - Visual representation of a Key-Value store (Microsoft Learn, 2024) 49](#_Toc188457311)

[Figure 39 - Visual representation of a column database (Amazon AWS, 2024) 49](#_Toc188457312)

[Figure 40 - Representation of a graphical database (Microsoft Learn, 2024) 49](#_Toc188457313)

[Figure 41 - Current Architecture for calling External Data Providers 51](#_Toc188457314)

[Figure 42 - Snippet of the Decisioning Engines HTTP Response, which the front-end references to control the next screen shown to applicants 52](#_Toc188457315)

[Figure 43 - Front-end screen shown to applicants for a "Progress" result. 53](#_Toc188457316)

[Figure 44 - Check Record in the Front-End system, saving a record of the decision(s) received from the decisioning system 53](#_Toc188457317)

[Figure 45 - Multiple Check records created in the Front-End system (UAT Environment as to not show any actual customer data) 54](#_Toc188457318)

[Figure 46 - Additional blocks of data within the decisioning engine HTTP Response 54](#_Toc188457319)

[Figure 47 - Check record created from the Scorecard data from the decisioning engine's HTTP response 55](#_Toc188457320)

[Figure 48 - Check record created from the External Info data from the decisioning engine's HTTP Response 55](#_Toc188457321)

[Figure 49 - Visual representation of a Microservice Architecture (Atlassian, 2024) 57](#_Toc188457322)

[Figure 50 - Proposed changes to Architecture, focusing on reusing API Integrations 58](#_Toc188457323)

[Figure 51 - JSON Schema for an Experian API Product as seen within our low-code decisioning engine 59](#_Toc188457324)

[Figure 52 - Example of a RegEx match function being used on a variable within this parsed JSON schema 59](#_Toc188457325)

[Figure 53 - Proposed JSON Schema for the Microservice Payloads 60](#_Toc188457326)

# Introduction

# Apply 1 - Java Decisioning Program

## Introduction & Requirements

The team I work in is primarily responsible for developing and maintaining a Credit Decisioning software platform. The software we use to do this is provided by a 3rd party and provides a low-code graphic user interface (GUI) for developing business logic into a callable Web service, without the need for specialised IT support. Currently I work as one of the Lead Analysts for this, effectively a Technical Lead for mentoring junior members of the team in building business logic and developing more complex functionality to orchestrate executing business logic e.g. modular designs, integrating database connections & integrating external data sources via HTTP.

This software is built within the Java language, with the GUI interface acting as a layer of separation to allow business users to develop without needing to have experience in the Java language. Personally however, I wish to have a greater understanding of Java, due to a desire to have more experience in object-oriented programming and to better understand how a Java program and various libraries scale to a full application.

With the support of my employer (after a 1-to-1 meeting), I decided to make an attempt to build a similar decisioning application within Java directly. Goals and acceptance criteria have been defined below:

### Project Description

Create a Java program that implements Credit Decisioning logic at an introductory level. This must include:

1. Some ability to capture relevant applicant information required to make a credit decision and enter this information into the program.
2. Execute credit decisioning policy rules that conditionally reject an application if set criteria is met.
3. Connect to an external database to retrieve more data to use for decisioning.
4. Use additional data gathered from this database to execute Scorecards in a champion/challenger approach (requiring at least 2 scorecards to be created)
5. Return a final decision and an offer to the user depending on the decisioning logic.
6. A record of the program run is then saved into an external database (for monitoring and further credit decisioning use)

### Acceptance Criteria/Project Outcomes

1. Final program must have an interface for a user to input application data
2. Application data input must include:
   1. A Match ID of some kind to look up in the external database.
   2. Residential Status
   3. Employment Status
3. A database connection is made to allow CRUD operations for logging activity made in the program.
4. A database connection is made to allow a lookup to be made based on the data received in the program.
5. Multiple credit decisioning policy rules are built and executed in the program flow.
   1. Policy rules are grouped in a modular way to breakdown testing.
   2. A mix of the application input data and data retrieved from the external database is used.
6. Multiple credit decisioning scorecards are built and executed.
   1. A credit offer is made in a champion/challenger style i.e. only 1 is used as the basis for the offer despite both being executed

## Stakeholders Involved

To develop the project, I consulted a number of stakeholders for support. A summary of the input from each stakeholder is recorded below. Communication methods used are also logged as:

* **F** = Face to Face
* **IM** = Instant Messaging via Microsoft Teams
* **C** = Call via Microsoft Teams
* **E** = Email

|  |  |  |
| --- | --- | --- |
| **Stakeholder** | **Methods** | **Input** |
| Credit Risk | **F, IM, C** | * Sign-off for project with Line Manager * Consulted for specific business logic that could be applied to the project & would be appropriate to share * Final review of the project |
| IT Delivery Managers | **F** | * Consulted for advice on how to host the application, specifically for the database link for CRUD operations |
| Asset Management | **E** | * Arranged the installation of all software and licenses required for this project, including:   + Eclipse IDE   + Visual Studio 2022 subscription with Microsoft Azure credits |
| Data Delivery | **IM** | Consulted for:   * advice on setting up an Azure SQL Database user. * importing data into a database that could be used for the program |
| Model Management | **E** | * Fact-checking on the theory sections of the Scorecard Development |

## Development – Set Up Dependencies

### CRUD - Database Connection

In practice, Decisioning Systems like these use a backend database to log application & API call data. They may also store data used in part of the Decisioning Strategy e.g. matching to internal customer data. Creating an integration therefore was a natural requirement. The below sections cover the work required for this.

### Standing up a Database Solution

2 Options were considered for this:

1. MySQL localhost database, a solution I know to be open-source (and therefore free to use)
2. An Azure SQL Database, which was recommended by my employer’s IT Delivery Managers as we hold licences for Visual Studio 2022 subscription, which come with $50 of monthly Azure Credits

The Azure SQL Database was chosen due to:

* The localhost nature of the MySQL solution limited the program to only working on the developer machine.
* Azure representing the opportunity to work with Cloud Computing resources.
* Azure potentially providing a host to deploy this application to if it went further.

Once set up, a connection was established within Java using the MS SQL JDBC library. This is contained within the “AzureJDBC.java” class within my project:

A screenshot of a computer program

Description automatically generated

Figure 1 - A Java Class called “AzureJDBC.java which handles a connection to an Azure SQL Database

### Handing Azure firewall

One issue I had with connecting to the database was that Azure protects its resources with a firewall.

To resolve this, I added my local IP Address to the Firewall rules within Azure to allow devices on my network to interact with the database.

A screenshot of a computer

Description automatically generated

Figure 2 - A screenshot showing a Firewall rule in place on my database to only allow access from my network (with IP Addresses redacted)

### Uploading Data

Credit Decisioning logic typically relies on external data provided by credit bureaus e.g. for the UK those are Experian, TransUnion and Equifax. Being able to retrieve a source for this data was vital to the program.

The scope I set for this project meant that I needed a batch of data the program could access. For this, I used a set of data based on Experian’s DelphiSelect API. This provides users with access to the data Experian hold on a person, which is used to create a credit file for that person. Lenders can then use this to inform their credit decisions (Experian, 2024).

For my program to work, I needed a way to upload a Dataset to the SQL database. For support on this, I consulted our internal Data Delivery team. This was done over Microsoft Teams:

A screenshot of a chat

Description automatically generatedA screenshot of a chat

Description automatically generated

Figure 3 - Teams Messages – Uploading Test data Into Azure SQL Database

From this conversation, 2 solutions for uploading a dataset based off this API into the Database were identified:

1. **Import Wizard** via the Azure Data Studio Software, the main software used for working with Azure Databases
2. **Azure Data Factory** – another cloud-based tool offered by Microsoft for ETL (Extract, Transform, Load) Data flows.

The Import Wizard was preferred due to its simplicity, but this kept timing out while loading. Azure Data Factory was the alternative I discussed with our Data Delivery team (see Appendix). The tool provides a beginner friendly “Ingest” template, which I was able to use to get my test dataset loaded.

Shown below is an example of this in action. The pipeline was run in “Debug” mode, which caters for one-off scenarios like this,

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Figure 4 - Azure Data Studio flow for loading data into a database

### Access Management

When creating the database, I was required to create a root/admin user, with full permissions to edit the database. It’s not appropriate to use this for my application as it has far more permissions than it actually needs, introducing security risks (Snowflake, 2024).

To set an appropriate level of access, I created a new Integration User within my database & granted only permissions that were needed.

A screenshot of a computer program

Description automatically generated

Figure 5 - SQL Code showing the creating of the integration user for the Java Program to use.

Permissions granted to this user included:

* SELECT access to the views required by the program to execute (Microsoft, 2024a)

A screenshot of a computer

Description automatically generated

A black background with white text

Description automatically generated

Figure 6 - SQL Views used within the Java Program

* INSERT access to the “dbo.JavaDecisioningHistory” table where transaction history will be logged.

A screenshot of a computer program

Description automatically generated

Figure 7 - SQL Table use to log results history of the Java Program

### Personal Data (PII) Protection

This **access management configuration can also be used for another purpose**: **protecting Personal Data (PII).** Credit Decisioning Systems like this must collect personal data to be able to conduct a credit search on a person e.g. Experian’s DelphiSelect API requires at least:

* Full name of the person in question
* Date of Birth
* At least the current address of the person in question
* Previous addresses are also often needed if the person in question has resided at their current address for less than 3 years

(Experian, 2024)

Each of these Data Points are protected under the UK's Data Protection Act 2018, which also codifies the GDPR into UK Law (UK Government, 2018). The program is currently collecting the Full Name of the user as a starting point to allow a full API integration in future. This means the program must be mindful of Data Protection/GDPR requirements.

Using a SQL databases within the program creates the risk of SQL Injection, where malicious input into the program can lead to arbitrary execution e.g. a user passes in a SQL command instead of a name (W3Schools, 2024). This is where the SELECT and INSERT permissions assigned above come into effect. The Full Name collected in the application is only saved to the **“dbo.JavaDecisioningHistory**” , which the integration user only has INSERT permissions on. This prevents a malicious user from entering a query to retrieve data from this table, as the database permissions will result in the query being rejected.

The only other permission the Integration User has been granted is the SELECT permission on the “**Delphi.VW\_DelphiPremiumValueData**” and the “**Delphi.VW\_DelphiSummaryData**” views. These are the only other views required by the program, so by limiting the access like this the possibility of risk incurred by SQL Injection is minimised and therefore acts as a control to protect Personal Data.

### Credentials within the Java program

To avoid hard coding these credentials within the program, a “.properties” file was created:

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

Figure 8 - Properties file with the credentials stored in Key-Value pairs, with sensitive information redacted

For security purposes, this properties file is not committed to the GitHub repository used for this project. It falls into a general rule to exclude all .properties files from the repository. This is done by adding a record to the “.gitignore” file, which tells Github files matching the names or expressions within should not be monitored. The \*.properties line acts as a catch-all to filter out all properties files like this.

A screen shot of a computer

Description automatically generated

Figure 9 - .gitignore file with a record to not include .properties files

Java can read these files by the “key” names e.g. for this file, Java can reference “jdbcUser” within the program and extract the value. The advantages of this are:

* Credentials are not hard coded, making them easy to change
* They can be stored in files that are not tracked by Version Control software like Github.

Below is code showing how this is implemented in the program:

A screen shot of a computer program

Description automatically generated

Figure 10 - Implementation of Java reading from an external file to set variables

The try-catch logic is used to catch scenarios where the config properties filed cannot be found. Without this, the program can’t run. To avoid any unnecessary behaviour if this occurs, the “System.Exit” method is used to terminate the program immediately.

## Development -Java

### User Input

The Java Scanner class was used to prompt the user for input. Defined below is a class called “UserInput” which has been written to group all methods related to capturing user input together:

A screenshot of a computer program

Description automatically generated

Figure 11 - Code used to define the “UserInput” class, importing the necessary Scanner class & defining

Upon running the program, the user will be asked a series of questions:

1. Their Forename
2. Surname
3. The ResidentialStatus and their EmploymentStatus, presented as a menu.

Console output for this is below:

A screenshot of a computer

Description automatically generated

Figure 12 - Java Console showing the Scanner Class in use to capture user input

An example of the code that generates this console output is annotated below:

A screen shot of a computer program

Description automatically generated

Figure 13 - An object method used to prompt the user for a Residential Status

What each part of this does:

* “println()” functions print out a menu of options to the console for the user
* A custom defined “getIntInput” method calls the Java Scanner & writes out another prompt to the console with instructions to the user
* A “while” loop traps the user in this loop until they provide a valid option
* The “try-catch” block is included to capture invalid characters e.g. string values when expecting a value of 1 to 6
* The “loopCount” increment logic is included as a failsafe for ending the while loop in case of too many bad inputs and/or preventing an infinite loop

### Data Model

One issue I ran into early on was how to logically structure the data used within the program. When working with Credit Decisioning systems in the workplace, we use a Data Model to define what data the application has access to and how it is stored.

I chose to apply a similar approach to this program, by defining an “AppData” class:

A screenshot of a computer

Description automatically generated

Figure 14 - the “AppData” class written to act as a Data Model

Doing this gave these advantages:

1. All application data is logically grouped
2. A single data object can be logged for review
3. Java Methods could be written to accept this model, allowing some flexibility around Java’s strongly typed data constraints when writing methods

Once the program completes its run, this Data model is populated with all the data used during the run. This allows the data to be logged easily for review. To do this, I made use of the GSON library to get a JSON representation of this (Google, 2024):

**A screen shot of a computer

Description automatically generated**

Figure 15 - Importing and using the GSON library to create a JSON Representation to log to the console

An abridged example output of this is below:

{

"appl": {

"forename": "T",

"surname": "T",

"employmentStatus": "FullTime",

"residentialStatus": "HomeOwner"

},

"decData": {

"pk": 4874,

"E1B07": "0",

"E1B08": "0",

"E1B09": 2,

"TRD\_A\_13": -3,

"E1A09": 0,

"E5S051": 777,

"NDSPCII": -4,

"TRD\_A\_06": 2,

"E1B13": 0,

"E1B01": 0,

"TRD\_STL\_14": -2,

"TRD\_STL\_19": -2

},

"scorecardRngResult": 77,

"strategyPath": "CHALLANGER",

"scorecards": [

{

"scorecardName": "ChampionScorecard",

"score": 733,

"index": 0,

"probOfDefault": 0.0024726231566347745,

"stringCharacteristics": [

{

"characteristicValue": "0",

"name": "E1B07",

"bandingScore": 27

},

{

"characteristicValue": "HomeOwner",

"name": "ResidentialStatus",

"bandingScore": 40

}

],

"numberCharacteristics": [

{

"characteristicValue": 2,

"name": "E1B09",

"bandingScore": 10

},

{

"characteristicValue": -3,

"name": "TRD-A-13",

"bandingScore": 40

},

{

"characteristicValue": 0,

"name": "E1A09",

"bandingScore": 46

},

{

"characteristicValue": -2,

"name": "TRDSTL14",

"bandingScore": -30

}

]

},

{

"scorecardName": "ChallangerScorecard",

"score": 691,

"index": 0,

"probOfDefault": 0.01798620996209156,

"stringCharacteristics": [

{

"characteristicValue": "FullTime",

"name": "EmploymentStatus",

"bandingScore": 45

}

],

"numberCharacteristics": [

{

"characteristicValue": -4,

"name": "NDSPCII",

"bandingScore": -40

},

{

"characteristicValue": 2,

"name": "TRD-A-06",

"bandingScore": 14

},

{

"characteristicValue": 0,

"name": "E1B13",

"bandingScore": 26

},

{

"characteristicValue": 0,

"name": "E1B01",

"bandingScore": 34

},

{

"characteristicValue": -2,

"name": "TRD-STL-19",

"bandingScore": 32

}

]

}

],

"flags": [

{

"policyCode": "D\_001",

"ruleOutcome": "PASS",

"decisionType": "DECLINE"

},

{

"policyCode": "D\_002",

"ruleOutcome": "PASS",

"decisionType": "DECLINE"

},

{

"policyCode": "D\_003",

"ruleOutcome": "PASS",

"decisionType": "DECLINE"

},

{

"policyCode": "A\_001",

"ruleOutcome": "PASS",

"decisionType": "ACCEPT"

}

]

}

Figure 16 - An example of the GSON output, returning a JSON Representation of the Java Program's data model

### Credit Policy built

The below section details the Credit Decisioning Logic built into the program.

#### Scorecards

Credit Scorecards are a tool in credit risk management used by Credit Risk Management teams to model/predict the likelihood of default for a given subject borrower (i.e. an applicant/customer) over a period (Huang and Scott, 2017). They are used for evaluating the creditworthiness of a customer, making them critical part of the Credit Decisioning process. This made them a natural choice to build in this project.

Scorecards from a program point of view are made up of:

* Characteristics which receive a score based on their value
* A final calculated score based on the sum of the characteristic scores + a starting score
* A “Score Index” used to group ranges of scores together. Typically translates into groups the business can use to simply communicate which applicants are more risky
* A “Probability of Default”, which translates the raw score into a percentage chance the applicant would default on a credit offer i.e. they are unable to pay

This project implements 2 scorecards in a champion/challenger fashion. This is an approach that deploys multiple competing strategies into a production environment and monitoring which provides the best (FICO, 2020). Using this is an effective demonstration of using Object-Oriented programming to build Credit Decisioning logic.

Note that both scorecards are not based on any actual statistical model. They are only for demonstrating how the concept can be implemented in an Object-Oriented program.

The Characteristics form the core of any scorecard. Given that each scorecard is unique in what variables it uses and how it assesses each variable, I chose to use classes containing static methods to implement these. This is shown in the figure below:

A screen shot of a computer

Description automatically generated

Figure 17 - Java Classes for both Scorecards.

##### Decide between Champion and Challenger

For a Champion/Challenger strategy, the goal is to send a percentage of applicants down the challenger path and compare the outcomes to the champion. To start, most of the volume would be directed to the known good model (i.e. the champion). This allows the business to test the challenger model with a limited volume to analyse if better business outcomes are given by the challenger.

To apply this to my program, I added logic into my AppData model constructor to decide which strategy path the program should follow. This logic is below:

A screen shot of a computer program

Description automatically generated

Figure 18 - View of the AppData Class constructor, which the Champion/Challenger split

Defining this logic in the constructor of the Data Model allows me to easily ensure this split is made early in the program. It is implemented as:

* 60% of the users will use the Champion Path
* 40% use the Challenger Path

The “Random” Java class is used to generate a random number to enforce this split.

##### Characteristics

These scorecards consist of a starting/base score plus a series of characteristics, variables with values that represent a statistical insight into the applicant/customer e.g. the E1A09 characteristic in the Champion scorecard count how many delinquent (i.e. in arrears) account they have present on their account (Experian, 2024). Specific values in that variable get assigned Banding Scores (or Points) which represent if they are a positive or negative indication of the applicant’s credit worthiness.

Once calculated, all the Banding scores are added together, along with an initial starting/base score to calculate a final score for the applicant, which represents their credit worthiness. A financial institution can use this as a basis for deciding a credit offer i.e. do they offer them a product and what terms they will offer if they do.

Below are specifications of both the Champion and Challenger scorecards built into the program, documented in a style typical in the workplace. The columns mean:

* **Characteristic** = the name of the variable used. Can reference ether data the applicant enters on an application form or data available on their Credit records accessed via Credit Bureaus like Experian
* **Data Type –** documents the type of values the characteristic can hold. Useful for guiding developers in how to handle the values within
* **Min Value & Max Value –** These relate to the **Points** column & defines the ranges used to set a specific Banding Score/Points. For String values, this is a 1-to-1 mapping for value to points
* **Points –** The actual Banding score assigned to the characteristic, which is then added to the overall score on the scorecard

##### Champion Specification

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Characteristic** | **Data Type** | **Min Value** | **Max Value** | **Points** | **Notes** |
| Base Score | Numeric | [default] | | 600 | This is a static value that all instances of this scorecard will start from. |
| E1B07 | String | T | | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Worst Status in the last 6 months of all Active CAIS accounts (SP)***  (Experian, 2024) |
| N | | 0 |
| D | | 0 |
| U | | 0 |
| 0 | | 27 |
| 1 | | 14 |
| 2 | | -10 |
| 3 | | -23 |
| 4 | | -23 |
| 5 | | -23 |
| 6 | | -23 |
| 8 | | -50 |
| [default] | | -50 |
| E1B09 | Numeric | -1 | 0 | -20 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as***: Number of Active CAIS Accounts (SP)***  (Experian, 2024) |
| 1 | 2 | 10 |
| 3 | 4 | 20 |
| 5 | 6 | 30 |
| 7 | max | 18 |
| [default] | | -20 |
| TRD-A-13 | Numeric | -3 | -3 | 40 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as***: Number of Months Since Most Recent Arrears on all Accounts allowed***  (Experian, 2024) |
| -2 | -2 | -20 |
| -1 | 0 | -20 |
| 1 | 1 | 10 |
| 2 | 2 | 5 |
| 3 | max | -30 |
| [default] | | -30 |
| E1A09 | Numeric | -1 | -1 | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Number of Delinquent CAIS Accounts, excluding Mail Order accounts***  (Experian, 2024) |
| 0 | 0 | 46 |
| 1 | 2 | 24 |
| 3 | max | -26 |
| [default] | | -26 |
| TRD-STL-14 | Numeric | -2 | -2 | 26 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***TSMR Active Short Term Loans account opened (Active CAIS)***  (Experian, 2024) |
| -1 | -1 | 0 |
| 0 | 0 | 26 |
| 1 | 6 | -24 |
| 7 | 18 | -15 |
| 19 | 36 | 6 |
| 36 | max | 15 |
| [default] | | -24 |
| ResidentialStatus | String | HomeOwner | | 40 | This data point is one of the User Input questions.   The status indicates the type of accommodation someone lives in. |
| PrivateTenantFurnished | | 26 |
| PrivateTenantUnfurnished | | 34 |
| CouncilTenant | | 10 |
| Cohabiting | | 34 |
| LivingWithParents | | 10 |
| [default] | | 10 |

##### Challenger Specification

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Characteristic** | **Data Type** | **MIn Value** | **Max Value** | **Points** | **Notes** |
| Base | Numeric | [default] | | 580 | This is a static value that all instances of this scorecard will start from. |
| NDSPCII | Numeric | -3 | -1 | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Consumer Indebtedness Index (SP)***  This is an index that represents how in debt a person is. Higher values indicate the subject is using more of their available credit.   (Experian, 2024) |
| 1 | 10 | 45 |
| 11 | 20 | 23 |
| 21 | 25 | 14 |
| 26 | 40 | 5 |
| 41 | 50 | -20 |
| 51 | max | -40 |
| default | | -40 |
| TRD-A-06 | Numeric | -2 | -2 | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Number of Consistently up-to-date accounts L12m*** (Experian, 2024) |
| -1 | -1 | -10 |
| 0 | 0 | -20 |
| 1 | 1 | 6 |
| 2 | 2 | 14 |
| 3 | max | 21 |
| default | | -20 |
| E1B13 | Numeric | -1 | -1 | -10 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Number of CAIS Status 3 or worse, within the last 6 Months for all Active CAIS accounts (SP)***  (Experian, 2024)  CAIS Status refers to the arrears position of the account e.g. 3 means 3 missed payments on the account. |
| 0 | 0 | 26 |
| 1 | 1 | -12 |
| 2 | max | -56 |
| default | | -56 |
| E1B01 | Numeric | -1 | -1 | 0 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Number of Active CAIS Accounts Opened in the Last 3 Months (SP)***  (Experian, 2024) |
| 0 | 0 | 34 |
| 1 | 1 | 12 |
| 2 | 2 | -11 |
| 3 | max | -33 |
| default | | -33 |
| TRD-STL-19 | Numeric | -2 | -2 | 32 | External data provided by Experian’s Credit Record API: DelphiSelect  Defined as: ***Total Number of Default Short Term Loans accounts (Default CAIS)***  (Experian, 2024) |
| -1 | -1 | 0 |
| 0 | 0 | 28 |
| 1 | 1 | -14 |
| 2 | max | -27 |
| default | | -27 |
| EmploymentStatus | String | FullTime | | 45 | This data point is one of the User Input questions.   The status indicates the type of employment an applicant is in. |
| PartTime | | 24 |
| Student | | 6 |
| SelfEmployed | | 33 |
| Unemployed | | -10 |
| Retired | | 22 |
| default | | -10 |

##### NULL Handling in characteristics

NULL values are technically possible within the External Data variables detailed in the specifications above. In this program the expectation is that those should fall into the “default” bands and get assigned the relevant points.

To handle this in Java, try-catch blocks are used to specifically catch a NullPointerException that would arise from attempting to use these values e.g. the screenshot below shows this applies to the E1B09 characteristic on the Champion scorecard:

A screen shot of a computer program

Description automatically generated

Figure 19 - NULL handling for a Scorecard characteristic.

##### Probability of Default formula

The calculated score on a Scorecard is intended to be a representation of how likely a subject is likely to default on any offered credit. In its natural form however, this isn’t a clear value to communicate probability to non-technical stakeholders. This becomes important when collaborating with product teams in credit decisioning, who typically do not have the subject matter expertise to interpret raw credit scores. The probability of default works as a more effective communication tool to help them set product offers e.g. max loan amounts and terms.

The Probability of Default formulas allow us to translate this raw score into a percentage value that can be more cleanly communicated to non-technical stakeholders. It makes use of coefficients (called “beta” β) and model variables defined early on in the Scorecard Model Development process to convert the raw score into a percentage value.

An accepted model for calculating this value (based on Logical Regression) is provided below:

|  |  |
| --- | --- |
| Probability of Default formula | **e** = Eular’s Number (i.e. exponent)  **β** = beta (i.e. the model coefficients)  **x** = Model variables defined in the Scorecard Model Development process |

Figure 20 - Probability of Default formula (Silva et al, 2020)

Beta i.e**. β** is a static value defined as part of the development process. Doing so requires using historical data on the population we’re modelling for. This data is run through statistical analysis (e.g. logical regression) to estimate a co-efficient to use.

X represents model variables defined in the scorecard process. For the purpose of this mock model, those are our Scorecard Characteristics covered above. The Points assigned to each variable are added together to form this.

Defining the constants to use in this model is typically done in the model development process. As only mock scorecards are used for this program to demonstrate the scorecards concept, the output of this will have no statistically meaningful value. The focus is solely on the Java implementation.

This screenshot below shows this coded in the “Scorecard.java” class:

A screen shot of a computer program

Description automatically generated

Figure 21 - Java Class method to calculate Probability of Default

Where:

* Math.exp() function provides the Exponent value required for the formula
* baselineScore & pointsJump represent the constants to be uses in the formula

##### Score Indexes

Score indexes are another method of translating the raw score from the Scorecard into an easier to understand metric for non-technical stakeholders. They help group ranges of scores/probability of defaults together to allow other business teams to make decisions on how they treat the person in question. One key example of this is deciding terms for offering a person a credit agreement e.g. how much money the business is prepared to lend over a set period of time (months, years etc).

To demonstrate this, I’ve coded “**setIndex**” methods into both my Champion and Challenger scorecards:

A computer screen shot of a program

Description automatically generated

Figure 22- SetIndex method for the Champion Scorecard

A computer screen shot of a program

Description automatically generated

Figure 23- SetIndex Method for the Challenger Scorecard

##### Peer Review

Given the complexity of the Scorecard Development process, I wanted to have the information above peer reviewed to ensure accuracy. For this I reached out to our internal Model Monitoring team. They act as an Internal Audit team who reviewing the performance and development process of how models like these Scorecards are built, making them perfect technical stakeholder contact for validating this information.

Below is the email chain that came from this:

Hi Ben,

I have modified some wordings in the scorecard section and added my comments on the probability of default formula(p12).

Hope this helps.

Kind Regards,

[Colleague]

**From:** Benjamin Roberts (Credit) <Benjamin.Roberts@vanquis.com>  
**Sent:** Wednesday, May 29, 2024 1:53 PM  
**To:** [Colleague]

**Subject:** Re: Scorecard training docs?

Thanks [Colleague]🙂

Could I take you up on that please?

One thing I need to do for the project is explain the concept of a scorecard & the parts of it. I've put something together in the attached word doc from my basic understanding.

Could you give the Scorecard section a read please & fact check it?

**Sent:** Monday, May 20, 2024 15:07  
**To:** Benjamin Roberts (Credit) <[Benjamin.Roberts@vanquis.com](mailto:Benjamin.Roberts@vanquis.com)>  
**Subject:** RE: Scorecard training docs?

Hi Ben,

It’s a while back (maybe 9 years ago) when [Colleague] and people at that year joined the company I did run some model/scorecard development training courses. I don’t know if I keep those and have moved them to SharePoint as PFG had been transformed greatly. I will do a digging to see if I could find anything.

In the meantime, I am very happy to help you to understand any steps/procedures on the Scorecard development life cycle if you have any questions.

Kind Regards,

[Colleague]

**From:** Benjamin Roberts (Credit) <[Benjamin.Roberts@vanquis.com](mailto:Benjamin.Roberts@vanquis.com)>  
**Sent:** Monday, May 20, 2024 1:03 PM  
**To:** Weidong Peng (Credit) <[Weidong.Peng@vanquis.com](mailto:Weidong.Peng@vanquis.com)>  
**Subject:** Scorecard training docs?

Hi Weidong,

Hope your well 🙂

Rizwan suggested I approach you on this question: Do you have any internal training documentation on Scorecard Model Development I could look at please?

I'm doing an Apprenticeship for personal development. One of the things I'm doing for that is building a Scorecard within the Java language & as part of it I need to explain what one is to a non-technical stakeholder. My own knowledge admittedly isn't on the end-to-end Model Development so I was hoping we had some documentation around?

Kind Regards,

Benjamin Roberts

Lead Credit Risk Analyst – Credit Systems

Phone: 07798 848049

Email: [Benjamin.roberts@vanquis.com](mailto:Benjamin.roberts@vanquis.com)

Address: 1 Godwin Street | Bradford | West Yorkshire | BD1 2SU

Logo

Description automatically generated

Figure 24 - Email Chain with colleague for peer review

#### Policy Rules

Policy rules check the available data on the applicant for specific signs of financial difficulty or any other indication that the applicant may be outside the credit institutions risk appetite.

Policy Rules can be of 1 of these types:

* **Decline** = If the logic is met, no offer of credit given to the applicant
* **Accept** = Valid only if the logic for all instances of the other 2 types are not met. Represents no concerns found on the provided data, an offer of credit can be given

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Rule Name** | **Decision** | **Logic** |
| D\_001 | Worst Arrears status is 2+ payments behind | Decline | IF DecisioningDataRow.E1B08 regex match 2|3|4|5|6|8 THEN FAIL |
| D\_002 | Experian Score below threshold | Decline | IF DecisioningDataRow.E5S051<= 600 THEN FAIL |
| D\_003 | Scorecard | Decline | IF (  decisionPath = “CHAMPION” AND Scorecards.Champion.score < 600  )  OR (  decisionPath = “CHALLENGER” AND  Scorecards.Challenger.score < 580  ) THEN FAIL |
| D\_004 | Unemployed | Decline | If EmploymentStatus == “Unemployed” |
| A\_001 | Accept | Accept | If none of the above Policy rules are hit |

## Testing the program

To be sure my program worked as intended and to the project requirements, testing was necessary. This was approached in 2 ways:

* Unit Tests via Junit 5
* End to end tests via running the program in the Eclipse IDE

### JUnit Unit Tests

When programming logic within the bespoke tool I use in my workspace, Unit Tests are used to ensure components of the code work as expected before a deployment is done. These are small scale tests that can be separated from the main program & re-used when any changes are made, allowing code quality to be maintained (AWS, n.d.). JUnit 5 is a Java package that provides a framework for writing these test & executing them (TutorialsPoint.com, n.d.). Examples written for this project are covered below:

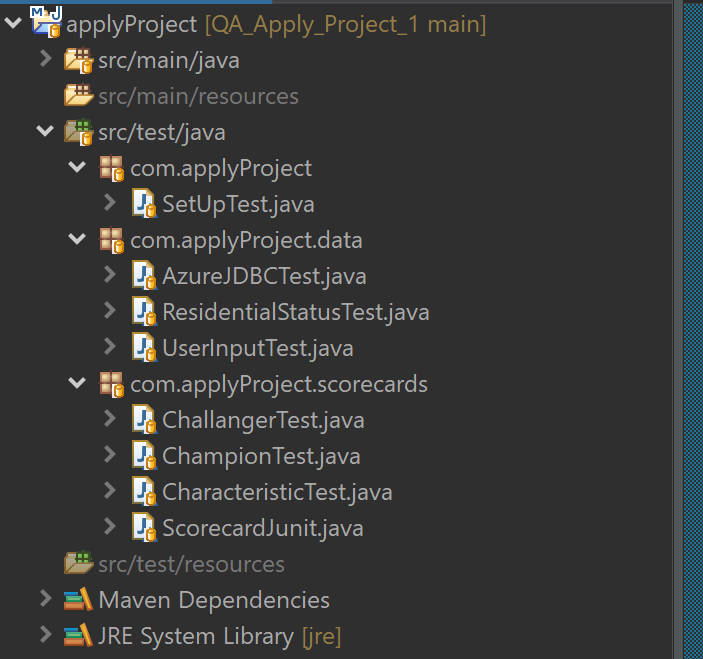


Figure 25 - JUnit 5 Test cases written to test individual parts of the program independently

A screen shot of a computer

Description automatically generated

Figure 26 - A test case written to test a program set up class that reads properties from an external file

### End to End

To fully test the program, it was executed within the Eclipse IDE:

A screenshot of a computer

Description automatically generated

Figure 27 - A demonstration of running the program within the Eclipse Software, with console output

Towards the end of the program, there are steps to output decisions made to the console. This includes the full Data Model used. This allowed manual review of the decisions made vs the specifications covered in this document.

A black screen with white text

Description automatically generated

Figure 28 - Console output for supporting debugging the program

### Debugging Errors

These forms of testing allowed me to identify unexpected behaviour (i.e. bugs) in my program. Below are 2 examples I’ve pulled out for demonstration.

#### Probability of Default - Result always Infinity

One issue the Unit Tests on the Scorecards uncovered was that the Probability of Default function would always return an “Infinity” value, regardless of the formula input. This made the calculation effectively useless for its intended purpose.

To start debugging this, I added a specific check to my Unit Test to check for infinity values:

A screenshot of a computer program

Description automatically generated

Figure 29- Unit Test for testing probability of default

With some trial and error when running this test, I was able to correct the error (unfortunately I did not have a specific commit in my GitHub repository capturing this).

#### Offers class returning exceptions

One Issue I faced with my program was that the Credit Offers the “Offers.java” class I wrote was not returning any of the expected results.

Examples of these tests are below:

A computer screen shot of code

Description automatically generated

Figure 30 - Unit Test on the Offers class

This test works by creating a new Scorecards & assigning a Score (via the constructor) and setting a Score Index of 1. The Score Index is what determines the maximum offer an applicant can receive e.g. in this test, we’re expecting the Index of 1 to mean that person could ask for up to £3000.

When I ran this test however, this was not what I was getting. Instead. The Fail trace should Illegal Argument Exceptions being raised:

A screenshot of a computer program

Description automatically generated

Figure 31- Unit Test results showing Illegal Argument Exceptions

After reviewing the code, I spotted I had neglected to add in the “break” statements for the Switch-Cases used within the “**setMaxOffer**” method this Unit Test was testing:

A screen shot of a computer program

Description automatically generated

Figure 32 - Java Switch case missing Break statements

The “**break**” statements are essential in Java’s implementation of the switch case, as they prevent Java from executing the next case (Geeks for Geeks, 2024a). Without it, what happens here is that every switch case gets evaluated, even if a matching case is found. Ultimately, this ends up with the “default” case running, which generated the Illegal Argument Exception causing my unit test to fail. Once corrected, the test pass as expected:

A computer screen shot of a program

Description automatically generated A screenshot of a computer program

Description automatically generated

Figure 33- Correcting the Switch Case, allowing the tests to pass

Attendees (by initials):

* BR, TD [Head of Function, cover for RK]

Apologies (by initials):

* RK (on leave)

Positives

* Reusable code is shown via the Scorecard and Policy rule templates.
* JSON output of the decisions made provides a full view of the data used for auditing.
* Champion/Challenger scorecards considered and implemented logically.
* Some Error Handing steps in place to prevent redundant program execution if the program could not complete e.g. database credentials missing.

Possible Improvements

* Error Handling has room for improvement e.g. if the program is unable to connect to the database, a specific Policy Rule could be built for reporting.
* Scorecard points, index boundaries etc could be better placed as configurable values (ether via external config file or in the SQL database), which avoids hard-coding them into the complied program (and difficult to change)
  + Also creates possibilities to hand controls of this to business teams, potentially creating competitive advantage via fast to implement change.
* Database timeouts should be explicitly considered e.g. 30 second timeout.

Agreed overall the program met the Acceptance criteria set out and had scope to improve should the project be developed further.

Figure 34 - Meeting Minutes for presentation on final version with employer

Overall, I believe this went well. My employer agreed the project criteria has been met and provided many valuable points of feedback from a less technical, more business practical point of view.

## Conclusions

### Project Outcome

Overall, the project succeeded in meeting the criteria set out in the introduction. The major functionality (Scorecards and Policy Rules) used within the bespoke Credit Decisioning software platform my team used has been developed into the program with re-usable classes that are suitable for creating many instances. A Credit decision is also made based on data retrieved from a backend database.

Additional Java Libraries were also discovered to expand the functionality of the program to effectively meet some of the requirements. For example, using the GSON library allows representations of the Java objects to be converted into JSON. That greatly simplified the requirement for retaining a record of the transaction, as this meant that all the data could be added to a Data Model object. All that was needed then was to save this Data Model to the logged history.

There was a significant amount of pre-requisite steps needed to get this project working e.g. setting up an Azure SQL database and loading data into it. This will be useful for later projects as a re-usable resource.

### Future Additions

Below are a few ideas I have in mind to expand this project in future:

* Front end interface to collect more data on an applicant e.g.
  + Addresses
  + Income
  + Expenditure
* Submit the collected applicant data to the program as a semi-structured message body e.g. JSON
* Use previous runs of the application within business logic e.g. check the history for a count of same person re-running the program
* Replace the random database lookup with a connection to the Delphi Select Sandbox API
* Automate execution of the JUnit test cases & generate a report on the results

Reviewing with my employer also identified areas for improvement:

* Change approach to setting values to allow configurations like Scorecard points to be done outside the program, making them easier to change (potentially by business users as well)
* Improved Error Handing, in ways that made Management Information (MI) available to business users
  + Specifically, an explicitly defined timeout on the database connection

# Apply 2 - React Web App – Decisioning Dashboard

# Apply 3 - Application Testing

# Apply 4 and 5 – DevOps Framework End 2 End

# Apply 6 - Microservices, Databases and API’s

## Introduction

Currently our Credit Decisioning system is set up as standalone “**monolithic**” platforms, a traditional architecture approach where all components of the software are compiled into a single network and code base, tightly coupling it all together (Atlassian, 2024). The figure below shows this visualised:



Figure 35 - Example of a Monolith platform (Atlassian, 2024)

This has been useful to date for getting our platforms ready for production on tight project timelines but has led to issues. Specifically, we see occurrences where we integrate with multiple HTTP API’s provided by 3rd parties across multiple platforms, but each platform uses its own code base to do this, leading to small differences in implementation.

This paper will examine the current architecture and tooling, look for potential issues and what alternatives are possible.

## Discovery

Before exploring existing architecture, topics relevant to examining the current decisioning system architecture are explored in the sections below.

### Relational Databases

A Relational Database (full name “Relational Database Management System", or RMDBS when abbreviated) is a storage solution that stores data in rows and columns, across various tables within the database. Examples of these types of databases are:

* Microsoft SQL Server
* MySQL
* OracleDB
* SQLite

The rows and columns make up “records” within the tables. Columns are used to capture specific data points, while rows hold the value for those columns. The table structure allows the data to be structured logically e.g. columns that all relate to an order on an e-commerce site (e.g. identifier, datetime) would all be in 1 table.

#### Keys

Tables relate to each other through “keys”. Different types of keys are used to structure data:

* **Primary Keys** – Column(s) in the table that uniquely identify a record. Ideally this would be a single column within the table, but it can be made up of multiple fields (these types being called “Composite Keys”
* **Foreign Keys** – A column within a table that references a record within another table. These are used to join tables to each other as needed e.g. analytics. When a column is declared as a foreign key, it references another column in a different table to establish a relationship with it. Once in place, values for the foreign key much match the references column in the other table.

The figure below visualises how these are used to relate tables. In it, “EmployerID” is the Primary Key of the “Employees” table. This is also referenced in the “Sales” table.

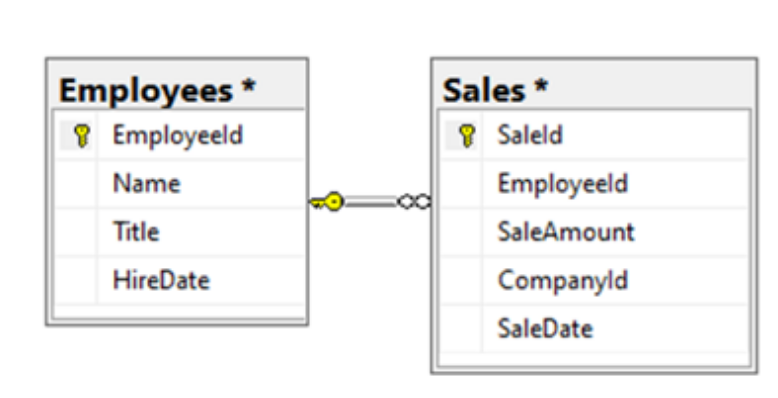


Figure 36 - Example of tables in a relational database, where a Foreign Key "EmployeeId" is used to relate the tables (Pluralsight, 2020)

#### Data Relationships

Tables within a relational database can have many types of relationships, which dictate the result you’d expect when joining tables together. At the high level, these relationships are:

|  |  |
| --- | --- |
| **Type** | **Description** |
| One-to-One | Records within Table A will join onto exactly one record from Table B, and no more than that.  Where it is possible there is no matching record in Table B, we call that a One-to-Zero or One relationship. |
| One-to-Many | One record within Table A will join onto ether one or multiple records from Table B (dependent per record).  The “to-Many” part of this relationship is strictly 1 way e.g. the multiple Records in B will only join onto 1 Record in Table A. |
| Many-to-Many | Multiple records within Table A will join onto multiple records from Table B.  This type of relationship should be discouraged, as they lead to complex joins that require significant computational power to calculate, resulting in slow queries.  One way to avoid this relationship is to create a table between the 2 tables, each with a One-to-Many relationship to this new table. For example: Table A and Table B have a Many-to-Many relationship. To avoid this, we create a Table C and create a One-to-Many relationship to it for both Table A and B. Doing so creates an intermediary that pre-stores the matching records between Tables A & B, minimising the computational resources needed to calculate the otherwise complex join when querying the database. |

#### Normalisation

Splitting the data like this allows “**normalisation**” to be achieved, meaning the data is organised to eliminate data anomalies (DataCamp, 2024). Examples of these anomalies are:

* Duplicate data points across several tables e.g. employee details. Without normalisation, any update would have to be done in multiple tables
* Redundancy – The same duplication means redundant data is in the system, which requires more storage space (and cost) to store

Normalisation is important to a relational database as it helps achieve referential integrity i.e. data is consistent across tables. Organising the data into structured tables prevents these data anomalies from occurring by ensures that updates, deletions, or additions in one table are reflected consistently across related tables. Attempting such an operation will return an error, preventing orphaned records or inconsistencies in the data. This help support reliable data management and data quality.

### Non-Relational (NoSQL) Databases

A non-relational database (also called NoSQL) also stores data, but does not use a table, row, identifying and relational keys structure like a traditional relational database does (Pluralsight, 2020). There are instead various forms of non-relational databases that are tailored to specific requirements but the data not being stored in a relational manner is the commonality between them. These database types have gained the term “NoSQL” due to this, as the lack of a relational model removes the traditional use case for the SQL language (Microsoft Learn, 2024).

Examples of these are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Types** | **Details** | **Use Cases** | **Example solutions** |
| **Document** | Contains data in a set of named string fields and object data values, referred to as a document (Microsoft Learn, 2024). Typically, this will take the form of a semi-structured format of data e.g. JSON, XML, YAML.  Multiple data entities can be combined into this document (e.g. customer details, order details etc), unlike a relational database which would separate these entities into tables. These documents also do not need to have the same structure.  Specific documents can be retrieved by a key, which uniquely identifies the document.  See Figure 37 for a visualisation of this. | * E-commerce data where items attributes differ product to product * Customer Data Management * Real-time analytics | * MongoDB * Azure Cosmos DB * Google Cloud Firestore |
| **Key-Value** | A simple table structure that stores data in 2 columns: the “key” and the “value”. The “key” column is used as the identifier for the data and the “value” columns stores the data point. Unlike a document store however, this is just a simple value.  The data is stored using a hashing function on the “key” value (hashing functions are one-way transformations, hence why this is not done on the “value” as well). Using the hash function allows for an efficient lookup.  These are optimised for simple lookup operations, where data need not be compared across multiple tables e.g. configuration options.  See Figure 38 for a visualisation of this. | * Caching session data on an application * Storing user preferences * Shopping cart data | * Couchbase * Amazon DynamoDB * Azure Table Storage |
| **Column-oriented** | These are like traditional Relational databases in that data is stored within rows & columns. Unlike relational databases however, data is stored within the columns of the table. Each record is still stored with a unique identifier, but the columns are grouped into “families” and these store logically related data points, which can be retrieved together. (Amazon AWS, 2024). This allows large amounts of data to be stored together and retrieved quickly.  These columns are not fixed. They can scale as needed and do not need to have pre-defined data types, removing the normalisation constraints of a typical relational database.  See Figure 39 for a visualisation of this. | * Logs * Read-heavy workloads e.g. analytics * Time-stamped data | * Amazon DynamoDB * Azure Cosmos DB * Google Cloud Datastore |
| **Graph** | Data is stored in the form of “nodes” and “edges”:   * “Nodes” represent entities e.g. a person, department * “Edges” represent the relationship between these nodes e.g. 1 person reports to another   (Microsoft Learn, 2024)  This storage allows the data to be represented graphically, hence the name.  See Figure 40 for a visualisation of this. | * Social Networks, for relationships between users * Supply chain management, via mapping out dependencies | * Azure Cosmos DB * ArangoDB |

*Example solutions sourced from* (Strauch, n/a)

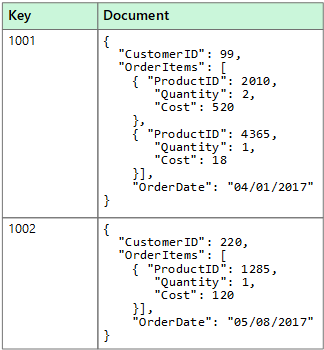


Figure 37 - Example of data storage in a Document store, a type of non-relational database (Microsoft Learn, 2024)

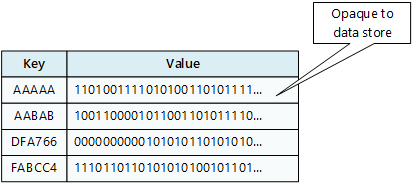


Figure 38 - Visual representation of a Key-Value store (Microsoft Learn, 2024)

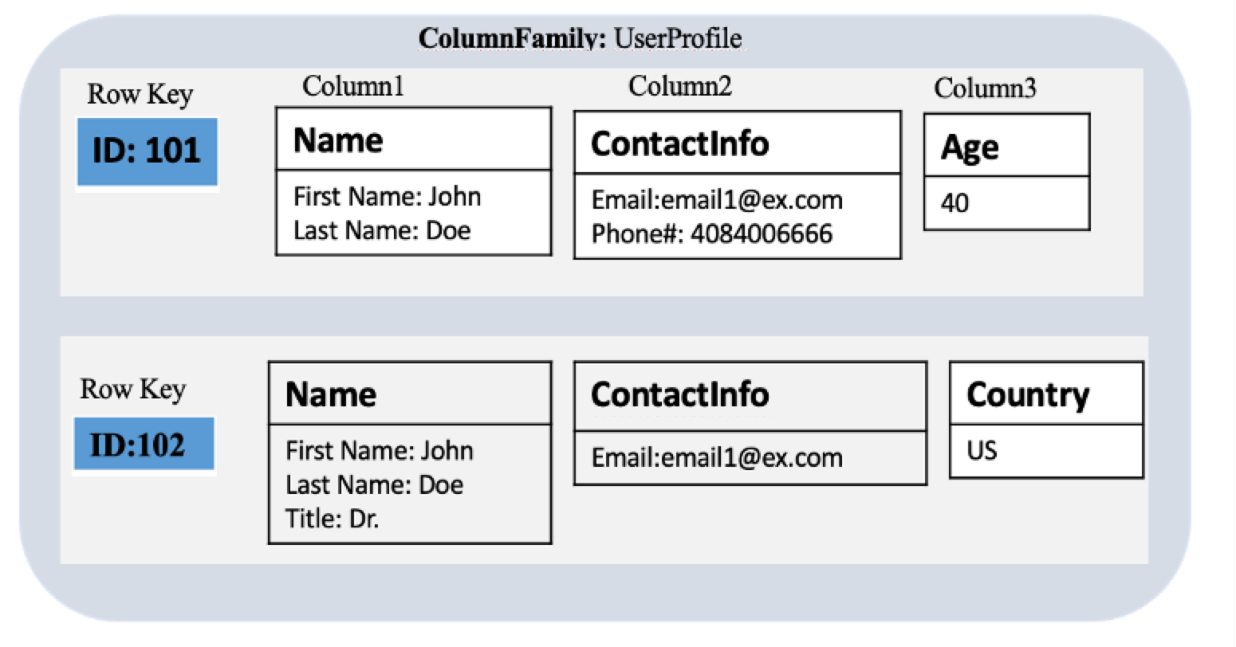


Figure 39 - Visual representation of a column database (Amazon AWS, 2024)

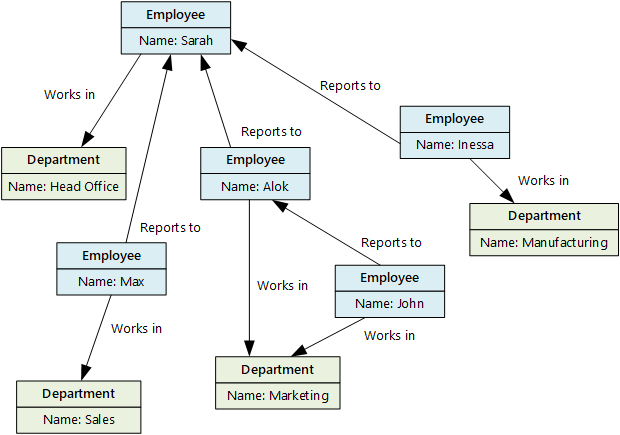


Figure 40 - Representation of a graphical database (Microsoft Learn, 2024)

#### Vs Relational Databases

There are some common differences between the various types of non-relational databases compared to relational ones. These are explored below:

|  |  |
| --- | --- |
| **Non Relational** | **Relational** |
| Non-Relational databases do not specifically have to confirm to a set schema the same way a relational model does.  This makes non-relational databases more reliable for data without fixed structures, or data that frequently changes | Data must be structured e.g. a table in a relational database must define columns with set datatypes. |
| Data is de-normalised, leading to potentially redundant/duplicate data that is unsuitable for analytics or reporting | Data is normalised, creating referential integrity in the data. |
| Data retrieval is simple and normally does not require joining other tables | Complex joins may be required |
| Excel at high availability and performance, where latency times are measured in milliseconds while performing millions of transactions per seconds |  |

(Microsoft Learn, 2022)

A summary takeaway I see form this is that non-relational databases are preferred for applications that require fast read times and do not require complex operations. For use cases that do require this however (e.g. analytics and reporting on key business performance indicators), a traditional relational database would be more suitable.

### Current Architecture

The below sections contain details of the existing architecture of our decisioning systems and how the data they generate is stored and used by front-end systems.

#### Overview

Below, I have summarised the architecture into a diagram:

A diagram of a data flow

Description automatically generated

Figure 41 - Current Architecture for calling External Data Providers

Our Decisioning Systems is currently split on a by-product basis i.e. for every product we offer, we use an individual instance of the decisioning system e.g. one for Loans, one for Cards etc. This is done so that individual decisioning logic that only applies to 1 product can be developed and maintained without the risk of introducing breaking changes into other products.

The decisioning system itself takes the form of a HTTP API. Front-end systems will send POST HTTP Requests to the system containing necessary data collected from the customer to make an informed credit decision. This data includes:

* Name
* Address
* Income
* Expenditure

The system uses this data to call Data Provider services offered by Credit Bureaus (e.g. Experian, TransUnion & Equifax) to obtain additional data. All this data is then used to execute a Credit Decisioning strategy to judge the credit risk of lending to an applicant. The system will then send a Response back to the front end with this decision, allowing the front-end to handle ether the onboarding steps (if the applicant is approved) or the decline process (if they are not).

#### Databases

Each instance also contains its own database; a MySQL relational database which is used for various purposes:

* Each HTTP Request/Response is logged for auditing
* Previous requests from the same applicant can be looked up e.g. if someone changes their details between applications, to get a preferable decision to them
* **Responses from external data providers are cached**, in case the same applicant re-enters the system. This allows the data to be reused, saving costs on calling these services (which are chargeable per call).

MySQL was not a specific tool choice by us. It was chosen by our 3rd Party partner & is deeply integrated into the Zoot system. As such, we are unlikely to be able to change it. MySQL however is a flexible solution, which provides some mitigation against this risk e.g. it can store and parse JSON documents, allowing it to effectively function as a Document-type non-relational database.

#### Connecting Data to User Interfaces

The HTTP Response created by the decisioning systems is critical. By itself, this is just a data set. The front-end system interprets this data to direct where the applicant journey should go next.

Figure 42 below for example shows part of the response that contains the decision made by the decisioning system on an application. The “finalDecisionReason” and “decisionReasonCodes” are the critical fields here, as differing values can be used to control what the next steps for an application should be.

A screenshot of a computer code

Description automatically generated

Figure 42 - Snippet of the Decisioning Engines HTTP Response, which the front-end references to control the next screen shown to applicants

The “Progress” values seen here are an instruction for the front-end to show a “Congratulations” screen to the applicant, to indicate we can provisionally offer then a credit product (subject to other checks in the journey). An example of this screen is shown in the figure below (note that this is for an older build of the front-end system).

A screenshot of a credit card

Description automatically generated

Figure 43 - Front-end screen shown to applicants for a "Progress" result.

That only covers the parts the customer sees. In the background, the front-end system also creates several records within its own system to record relevant information on the application. This is to cover audit scenarios where a full view of how the applicant went through the application journey is needed & what decisions are made. Another use case is that this data can be referenced by the Operations teams when an application needs to be manually approved before a credit product can be offered.

Decision records like the one shown in Figure 42 above are saved in what the front-end calls “Check” records. The figure below shows an example:

A screenshot of a computer

Description automatically generated

Figure 44 - Check Record in the Front-End system, saving a record of the decision(s) received from the decisioning system

That is not the only record created however. Multiple check records are created based on data received in the HTTP Response from the decisioning system, covering other data points seen besides the main decision, as shown in the figure below:

A screenshot of a computer

Description automatically generated

Figure 45 - Multiple Check records created in the Front-End system (UAT Environment as to not show any actual customer data)

Figure 45 shows additional check records. These are based on other data blocks received in the response e.g. the “scorecards” data maps to the data seen in the Check record show in Figure 47, while one of the entries in the “externalInfo” data maps to the record shown in Figure 48. Data mapping logic within the front-end system parses the JSON response received to create as many records as required, allowing the front-end to maintain a complete record of the application, in a cleaner User Interface then a raw JSON can provide.

A screenshot of a computer code

Description automatically generated

Figure 46 - Additional blocks of data within the decisioning engine HTTP Response

A screenshot of a computer program

Description automatically generated

Figure 47 - Check record created from the Scorecard data from the decisioning engine's HTTP response

A screenshot of a computer

Description automatically generated

Figure 48 - Check record created from the External Info data from the decisioning engine's HTTP Response

## Strengths and Weaknesses

From reviewing the architecture and including reflections on recent work experience, I have identified the following strengths and weaknesses of the architecture:

|  |  |
| --- | --- |
| **Strengths** | |
|  | Using isolated instances of the decisioning system per product helps make the business logic implementation more maintainable.  For example, if changes are only needed in the business logic for evaluating Cards credit risk, this split means we would not risk creating breaking changes in the business logic for evaluating credit risk for a Vehicle Finance product. The split platforms mean the logic is isolated from one another. |
|  | The MySQL database solution can support both relational and non-relational database design paradigms, making it flexible for different use cases.  It also specifically has options for horizontal scaling, allowing the database to gain more processing power as demand increases. |
| **Weaknesses** | |
|  | HTTP Integrations with 3rd Parties are currently being built per monolithic product platform. These effectively means we are duplicating code per platform and as these are independent, they can differ.  This has been a risk that has materialised in recent history e.g. in 2024 we found a critical issue where the integration between an Experian API differed between 2 of our platforms. The way address data was being mapped into the request to the Experian API differed and resulted in the Experian API not consuming these properly. This negatively impacted the data we received back from that API and presented regulatory and compliance implications, requiring us to roll back the impacted build i.e. revert to an earlier production build. |
|  | Data stored in the Database cannot be re-used per platform, as they are all isolated from each other.  For example, the Response Cache use case stores responses from external data providers and attempts to re-use them before making new API calls. As each call is chargeable, this reduces costs. The isolation however means it is theoretically possible for an applicant to apply for each individual product we offer & we’d call the Data Provider service each time. Assuming the request data is the same (i.e. name, address, date of birth), then we’d get the same response from the data provider for each call.  This would be a barrier if we wanted to build a customer management process that checks a customer’s eligibility for each product (i.e. cross sell). |
|  | If there is Business Logic that is shared across platforms, we’d currently need to implement this on each individual platform.  This could be mitigated however by importing the necessary business logic between platforms (e.g. with version control). A design doing this however must be modular i.e. not have any dependencies on processes specific to a single platform. |

## Alternative – Microservice Architecture

The weaknesses identified in the above section highlight pain points in our current architecture. These weaknesses are symptoms of the monolithic approach we’ve taken so far. To resolve these, we can look at using a “microservice” approach.

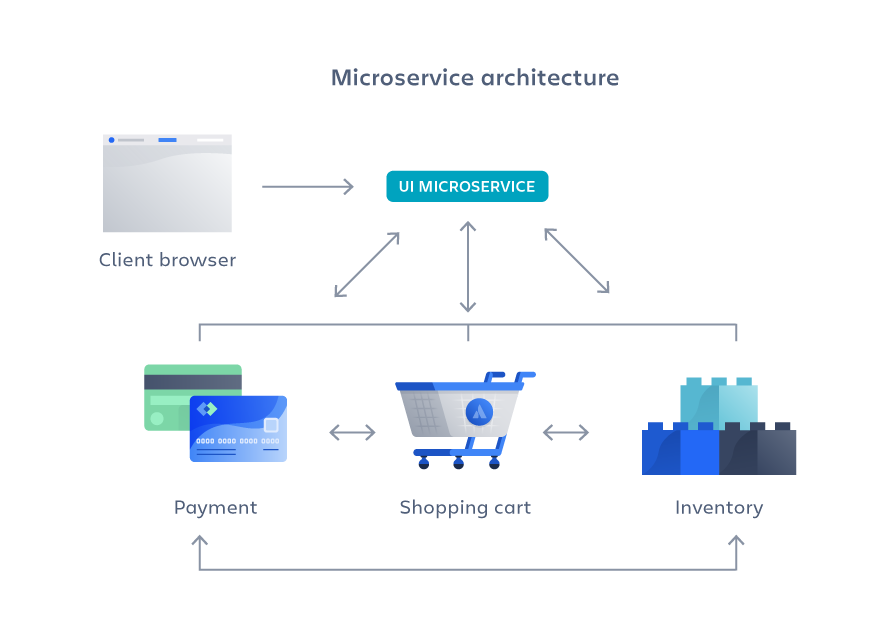


Figure 49 - Visual representation of a Microservice Architecture (Atlassian, 2024)

The Microservice architecture differs from monoliths in that instead of packaging all components into a single code base, multiple independent services are created and split into their own code bases, each performing a specific business logic. The resulting services then communicate with each other as needed e.g. via HTTP calls (Atlassian, 2024).

This approach allows for business logic to be decoupled from each other, bringing benefits over traditional monoliths:

* The independent services can be re-used across other solutions as needed e.g. if a later business project required a payment service, we would not need to build a new one
* The services are independent of each other, allowing changes to be made without the risk of breaking another service
* Splitting the services creates smaller code bases, making the solution easier to maintain and test, and therefore minimising time to market for changes
* The small code base also creates less opportunity to be “locked in” to specific technology choices, bringing more freedom in upgrading

(Atlassian, 2024)

### Proposed Architecture

Based on this idea of Microservices, I propose the to split off our 3rd party data provider integrations into Microservices, where our product platforms would instead call these services instead of using integrations built within the platform itself. Figure 50 shows this visualised:

A diagram of a database

Description automatically generated

Figure 50 - Proposed changes to Architecture, focusing on reusing API Integrations

Instead of integrating 3rd party APIs per product platform, we instead set up a new platform solely for calling these services. This will allow for **a consistent integration between these services to be used**, resolving the 1st weakness identified previously.

Using a single platform with a single Response Cache database also resolves the 2nd weakness, as this singular entry point to the integrations means we can share a database for these services**, allowing cached data to be re-used across platform**s.

Strictly speaking, a **true microservice architecture would use separate platforms for each 3rd party service we’d use**, however **doing so increases the costs** (as our 3rd party partner bills us per platform). Having a singular platform allows the above benefits while minimising this platform cost overhead, trading off not fully isolating each 3rd party service.

### How to Implement

Currently, our decisioning engine instances parse the JSON/XML responses from these 3rd party services into a Schema, allowing the data within to be used by the platform. This schema must pre-declare every object and variable within the structure. Anything not captured is dropped. An example of one such schema is shown below:

A screenshot of a computer program

Description automatically generated

Figure 51 - JSON Schema for an Experian API Product as seen within our low-code decisioning engine

Making this architecture change could have implications on how this is referenced. For example, variables/objects using this schema reference the exact path, with the below figure showing a variable called “MatchTo” being used in an IF-condition RegEx logic:

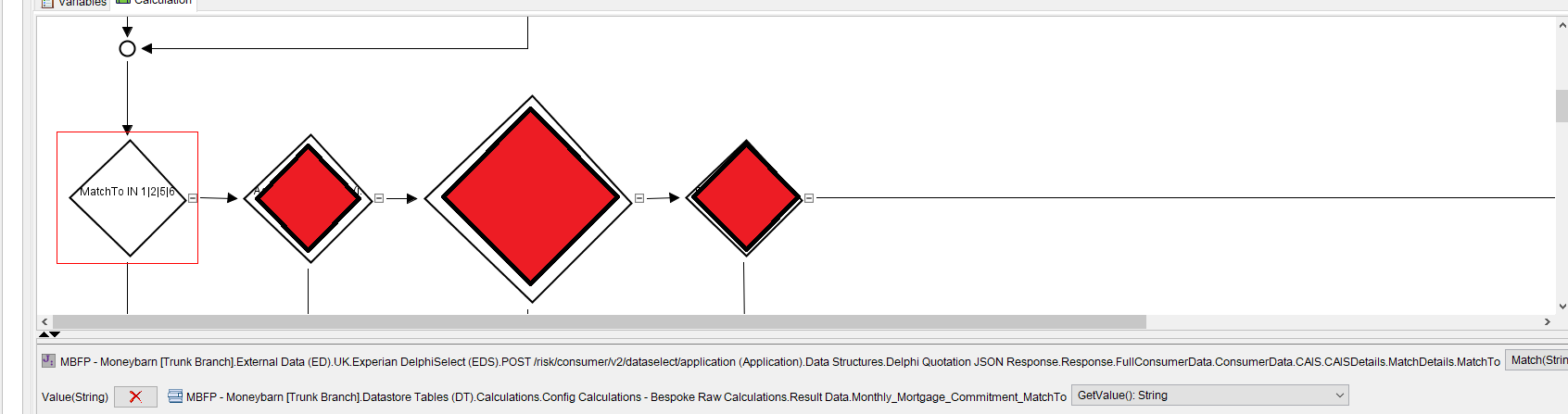


Figure 52 - Example of a RegEx match function being used on a variable within this parsed JSON schema

To avoid having to refactor any code, we’d have to pass the response from the 3rd party data provider exactly as is when designing our microservice payloads, so that it would still conform to this schema. This could become a limitation, as it limits our ability to create useful logging information from the HTTP call to the 3rd party service.

We could instead wrap the raw response from the data providers inside a “providerPayload” object, which opens the schema for additional metadata to be passed. A proposal for how this could look is below:

A screenshot of a computer

Description automatically generated

Figure 53 - Proposed JSON Schema for the Microservice Payloads

This structure will:

* Allow the payload from the data providers to be captured exactly as is, meaning if new data is added, we don’t need to update the schema for it to be captured. The exact format that can accomplish this however needs investigating. I have a concern that system limitations within our 3rd party decisioning engine tool mean that this may not be true if the raw payload from the data provider is saved as an object. It may be required to save the raw payload as a string instead to avoid this (preferably as a Base64 encoded value to avoid issues with special characters).
* Additional data from the microservice platform can be recorded e.g. call duration to the data provider in milliseconds, allowing us to record this for MI Purposes e.g. measuring the overhead using the microservice architecture adds compared to the current direct integration done per platform

## Conclusion

There are some painful weaknesses within our existing monolithic platforms around how they integrate with 3rd party services. The single code base nature of monolithic platforms means we duplicate the code required for these integrations to each platform. Each platform then makes changes to this code, resulting in differing implementations. We’ve already seen a critical bug in production because of this.

Moving the 3rd party data integrations to a separate platform in a microservice fashion would mitigate this and provide us with re-usable services. Ideally, we’d split off each data provider into its own microservice, however this would incur additional cost, as the core architecture of the decisioning engine software provided by our 3rd party partner means that a separate platform would have to be created for each service. They bill us per platform, so this approach would significantly increase costs. As a compromise, the proposed solution outlined in this document uses a shared platform to host these integrations, allowing the benefits to be realised with minimal cost.

There is one potential problem with the proposed solution, where the need for a predefined schema means we could lose data if the 3rd party provided add new variables or objects. How to mitigate this is an action to work with our 3rd party partner with. There is however a workaround identified here, where we could encode the raw payload from the providers in Base 64 and provide in the microservice response as a string, avoiding the risk at the cost of requiring decoding within the platforms.

# Additional Evidence

# Final Reflections

## Future Development

# Appendices

## Guidance (Delete this later)

Portfolio Formatting

- Cover page - name, date, professional design, company branding

- Table of contents

- Overall introduction - Introduce yourself, your role, your company, the apprenticeship etc.

- Apply activities inserted one after another

- Any further evidence they wish to include (communication, problem solving, creative thinking etc if it is relevant to the workplace)

- Overall reflection - talk about what went well, the challenges, future plans etc...

Consider the following;

- Page numbers

- Header, footer, subheadings

- Consistent text font, size, style

# Bibliography

Amazon AWS, 2024. *Types of NoSQL databases.* [Online]   
Available at: https://docs.aws.amazon.com/whitepapers/latest/choosing-an-aws-nosql-database/types-of-nosql-databases.html  
[Accessed 17 01 2024].

Atlassian, 2024. *Microservices vs. monolithic architecture.* [Online]   
Available at: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith#:~:text=A%20monolithic%20architecture%20is%20a%20singular%2C%20large%20computing%20network%20with,of%20the%20service%2Dside%20interface.  
[Accessed 16 01 2024].

AWS, n.d.. *What is Unit Testing? - Unit Testing Explained - AWS.* [Online]   
Available at: https://aws.amazon.com/what-is/unit-testing/  
[Accessed 22 01 2024].

Bechtold et al, 2024. *JUnit Jupiter API.* [Online]   
Available at: https://junit.org/junit5/docs/current/user-guide/  
[Accessed 05 06 2024].

Bionic Turtle, 2011. *FRM: Logistic distribution maps credit score to probability of default (PD).* [Online]   
Available at: https://www.youtube.com/watch?v=HnQ6QjeGMm0  
[Accessed 09 05 2024].

DataCamp, 2024. *Normalization in SQL (1NF - 5NF): A Beginner’s Guide.* [Online]   
Available at: https://www.datacamp.com/tutorial/normalization-in-sql  
[Accessed 20 01 2024].

Experian, 2024. *DelphiSelect API Documentation - for Credit Decisioning data.* [Online]   
Available at: https://developer.experian.com/products/uk/delphi-select/delphi-select-api-docs   
[Accessed 17 05 2024].

FICO, 2020. *A Guide to Champion/Challenger Testing Approaches: Overview and Benefits..* [Online]   
Available at: https://community.fico.com/s/blog-post/a5Q2E000000YNqoUAG/fico2227  
[Accessed 17 05 2024].

Geeks for Geeks, 2024a. *Switch Statements in Java.* [Online]   
Available at: https://www.geeksforgeeks.org/switch-statement-in-java/   
[Accessed 06 17 2024].

Google, 2024. *GSON Libary.* [Online]   
Available at: https://mvnrepository.com/artifact/com.google.code.gson/gson  
[Accessed 05 06 2024].

Microsoft Azure, 2010. *Adding Users to Your SQL Azure Database.* [Online]   
Available at: https://azure.microsoft.com/en-us/blog/adding-users-to-your-sql-azure-database/  
[Accessed 03 05 2024].

Microsoft Learn, 2022. *SQL vs. NoSQL data.* [Online]   
Available at: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/relational-vs-nosql-data  
[Accessed 21 01 2024].

Microsoft Learn, 2024. *Non-relational data and NoSQL.* [Online]   
Available at: https://learn.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data  
[Accessed 16 01 2024].

Microsoft, 2023a. *Northwind Database - TSQL.* [Online]   
Available at: https://github.com/microsoft/sql-server-samples/blob/master/samples/databases/northwind-pubs/instnwn  
[Accessed 01 05 2024].

Microsoft, 2024a. *Database-level roles.* [Online]   
Available at: https://learn.microsoft.com/en-us/sql/relational-databases/security/authentication-access/database-l  
[Accessed 05 06 2024].

Microsoft, 2024b. *MS SQL JDBC.* [Online]   
Available at: https://mvnrepository.com/artifact/com.microsoft.sqlserver/mssql-jdbc  
[Accessed 05 06 2024].

Pluralsight, 2020. *Relational vs Non-Relational Databases.* [Online]   
Available at: https://www.pluralsight.com/resources/blog/software-development/relational-vs-non-relational-databases#:~:text=There%20are%20five%20popular%20non,used%20for%20a%20single%20application.  
[Accessed 16 01 2024].

QA Apprenticeships, 2024. *Module 1 - Java JDBC Slides.* [Online]   
Available at: https://github.com/broberts-gtw/LVL\_4\_Software\_Dev\_Repo/blob/main/Module%201%20-%20Intro%20to%20Software%20Development/Live%20Course%201%20Materials/\_\_Course%20Stretch%20files/Java/presentation/JDBC.pptx  
[Accessed 01 05 2024].

Silva et al, 2020. *A logistic regression model for consumer default risk - National Library of Medicine.* [Online]   
Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9041570/  
[Accessed 09 05 2024].

Snowflake, 2024. *Access Control.* [Online]   
Available at: https://docs.snowflake.com/en/user-guide/security-access-control-overview  
[Accessed 05 06 2024].

Stack Overflow , 2021. *Adding objects to an array list.* [Online]   
Available at: https://stackoverflow.com/questions/70406298/adding-objects-to-an-array-list-cannot-invoke-xxx-add-b  
[Accessed 14 05 2024].

Stack Overflow, 2008. *Can an abstract class have a constructor?.* [Online]   
Available at: https://stackoverflow.com/questions/260666/can-an-abstract-class-have-a-constructor  
[Accessed 03 05 2024].

Strauch, C., n/a. *NoSQL Databases.* [Online]   
Available at: https://www.christof-strauch.de/nosqldbs.pdf  
[Accessed 17 01 2024].

TutorialsPoint.com, n.d.. *JUnit - Test Framework - Tutorialspoint.* [Online]   
Available at: https://www.tutorialspoint.com/junit/junit\_test\_framework.htm  
[Accessed 22 01 2025].

UK Government, 2018. *Data Protection Act 2018.* [Online]   
Available at: https://www.legislation.gov.uk/ukpga/2018/12/contents/enacted   
[Accessed 17 06 2024].

W3Schools, 2024. *Java Abstraction.* [Online]   
Available at: https://www.w3schools.com/java/java\_abstract.asp  
[Accessed 03 05 2024].