FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

BirdlyData - Data migration tool for OutSystems

Beatriz Ferreira



Mestrado em Engenharia de Software

Supervisor: Nuno Flores

July 24, 2021

BirdlyData - Data migration tool for OutSystems

Beatriz Ferreira

Mestrado em Engenharia de Software

July 24, 2021

Abstract

Data migration has become one of the most demanding tasks inside organizations due to the rapid growth of data. The process of migrate data between environments always requires at least two environments: a source environment to extract the data and a destination/target environment to load transformed data. The procedure includes data profiling, mapping, cleansing, validation and an ongoing process for data quality assurance.

Nowadays, more enterprises are subscribing low code platforms to build powerful applications without being limited by technology. IT leaders are concerned with time-wasting tasks, and prefer to invest the developer time, talent and experience to create software. OutSystems is a a low code/full-stack application development platform, with full IDE integrated where developers can build, deploy and manage their applications faster. It allows developing enterprise applications on a DevOps ready environment that gets everything in production with the click of a button.

Despite being a complete platform, OutSystems has some limitations. Current offerings for the OutSystems Platform lack built-in data synchronization functionalities and/or require licenses and additional developments. The OutSystems platform does not give the developers permissions to perform a direct data synchronization because the database is managed solely by the platform.

This dissertation addresses these issues by introducing a new data migration tool, developed to mitigate the OutSystems related shortcomings. It starts by giving an overview of all the essential processes, concepts and techniques involved in data migration, following by the decision-making carried out in the study and, finally, introduces a new data migration tool able to migrate applicational data between different OutSystems environments, using the Server Studio and Integration Studio integrated in the platform. The tool includes User Interface to display and gather the necessary data for the migration process and a C# extension where the migration runs. Furthermore, some code optimizations, able to turn the software as effective as possible and reduce the migration execution time, are listed.

At last and to testify the completeness and efficiency of the purposed tool a live test is conducted and a case study is evaluated.

Keywords: OutSystems, Data Migration, Big Data, Data Mobility, Data Synchronization, Data Migration Tools, ETL, Data Replication, Automatic Testing, Schema Integration, Data Mapping

Resumo

A migração de dados tornou-se uma das propostas mais exigentes dentro das organizações devido ao rápido crescimento dos dados. O processo de migração de dados entre ambientes requer pelo menos dois ambientes: um ambiente de origem para extrair os dados e um ambiente de destino para carregar/importar os dados transformados. O procedimento inclui examinação de dados, mapeamento, limpeza, validação e um contínuo processo para garantir integridade dos dados.

Cada vez mais, empresas estão a recorrer a plataformas de código de baixo nível para construir aplicações completas sem serem limitadas pela tecnologia aderente. Os líderes de TI preocupam-se com tarefas que desperdiçam muito tempo e preferem investir o tempo, talento e experiência do desenvolvedor para criar um software de qualidade. OutSystems é uma plataforma de desenvolvimento de aplicações de código de baixo nível, com IDE totalmente integrado, onde os desenvolvedores podem construir, publicar e controlar as suas aplicações com mais rapidez. A plataforma permite desenvolver aplicações corporativas num ambiente pronto para DevOps que coloca tudo em produção apenas com o clique de um botão.

Apesar de ser uma plataforma bastante completa, OutSystems possui algumas limitações. As ofertas atuais da plataforma OutSystems não possuem funcionalidades de sincronização de dados integradas e/ou requerem licenças e desenvolvimentos adicionais. A plataforma OutSystems não dá aos desenvolvedores permissões para realizar uma sincronização direta de dados porque a base de dados é gerenciada exclusivamente pela plataforma.

Esta dissertação aborda esses problemas introduzindo uma nova ferramenta de migração de dados, desenvolvida para mitigar as limitações relacionadas ao OutSystems. Esta começa por dar uma visão geral de todos os processos, conceitos e técnicas essenciais envolvidos na migração de dados. De seguida, a tomada de decisão efetuada no estudo é relatada e, por fim, apresenta-se uma nova ferramenta de migração de dados capaz de migrar dados aplicacionais entre diferentes ambientes OutSystems, utilizando o Server Studio e o Integration Studio integrados na plataforma. A ferramenta inclui uma interface de utilizador capaz de exibir e reunir os dados necessários para o processo de migração e uma extensão C# onde a migração é executada. Além disso, são listadas algumas otimizações de código, capazes de tornar o software o mais eficaz possível e reduzir o tempo de execução da migração.

Por fim, e para testar na totalidade a eficácia da ferramenta proposta, um caso de uso é realizado onde o comportamento da ferramenta é avaliado.

Keywords: OutSystems, Migração de dados, Sincronização de dados, Ferramentas de migração de dados, ETL, Replicação de dados, Testes automáticos, Mapeamento de Dados

Acknowledgements

Throughout the writing of this dissertation I have received a great deal of support and assistance.

I would first like to thank my supervisor, Professor Nuno Flores, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback and your patient guidance pushed me to sharpen my thinking and brought my work to a higher level. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly.

I would like to acknowledge my colleagues from my internship at BIT SONAE for their wonderful collaboration. I would particularly like to single out my supervisor at BIT SONAE, Rui Fonseca, I want to thank you for your patient support and for all of the opportunities I was given to further my research.

I would also like to thank my tutor, Bruno Miguel Ferreira, for he's valuable guidance throughout my studies. You provided me with the tools that I needed to choose the right direction and successfully complete my dissertation. Without his persistent help, the goal of this project would not have been realized.

In addition, I would like to pay my special regards to my family for their wise counsel and sympathetic ear. My deep and sincere gratitude to my parents for their continuous and unparalleled love, help and support. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am. I also wish to show my gratitude to my sister for always being there for me as a friend.

Finally, I could not have completed this dissertation without the support of my boyfriend, Gonçalo Espínola, who provided stimulating discussions as well as happy distractions to rest my mind outside of my research. I would like to recognize the invaluable assistance that you provided during my study. I am forever thankful.

Beatriz Fereira

"Success belongs to those who believe in the power of their ideas."

Richard P. Feynman

Contents

1 Introduction **1**
- 1.1 Cost of building software . . . 1
- 1.2 OutSystems as a Low-code platform . . . 2
- 1.3 Motivation . . . 5
- 1.4 No operational data for development testing . . . 7
- 1.5 Goal . . . 8
- 1.6 Document Structure . . . 10

2 Data Migration **11**
- 2.1 Need for Data Migration . . . 11
- 2.2 Migration Risks . . . 12
- 2.3 Data Migration Strategies . . . 13
- 2.4 Data Migration Architecture Approaches . . . 14
- 2.5 Data Mapping . . . 16
- 2.6 Data Quality Validation . . . 18
- 2.7 Data Migration Types and Software Environments . . . 19
- 2.8 Data Migration Automation . . . 20
- 2.9 Data Migration in OutSystems . . . 23
  - 2.9.1 Developing an application with OutSystems . . . 23
  - 2.9.2 Data Migration Guidelines . . . 24
  - 2.9.3 Partners's Products . . . 25
  - 2.9.4 Comparison of OutSystems data movers . . . 26
  - 2.9.5 Limitations . . . 29
- 2.10 Summary . . . 30

3 Using OutSystems for Data Migration **31**
- 3.1 Open issues . . . 31
- 3.2 Metadata collection . . . 32
- 3.3 Metadata analysis . . . 32
  - 3.3.1 System Entities . . . 33
  - 3.3.2 Application Entities . . . 34
  - 3.3.3 Hybrid Entities . . . 34
- 3.4 Technological choices . . . 34
- 3.5 Methodological choices . . . 35
- 3.6 Summary . . . 36

4 BirdlyData **37**
4.1 Architecture . . . 38
4.1.1 BirdlyData . . . 38
4.1.2 BirdlyDataNotifications . . . 40
4.2 Startup Configurations . . . 40
4.2.1 Manage Dependencies - Integrations . . . 40
4.2.2 OutSystems' System Entities . . . 41
4.2.3 Created Entities . . . 41
4.3 User Interface . . . 43
4.4 Functionalities . . . 44
4.4.1 Settings - Data Access Configuration . . . 45
4.4.2 Entity Dependencies . . . 46
4.4.3 Migration . . . 46
4.4.4 Real time Migration Log . . . 48
4.4.5 Reuse Migration Plan . . . 49
4.4.6 Schedule Migration . . . 49
4.4.7 Notifications . . . 50
4.4.8 Migrate *Users* . . . 51
4.4.9 Differential Migration . . . 52
4.4.10 Mask Data - Anonymize . . . 53
4.4.11 Delete Target Data . . . 54
4.4.12 Call Migration through API . . . 54
4.4.13 Self-referencing foreign key . . . 55
4.4.14 Static Entities Dependencies . . . 55
4.5 Optimizations . . . 55
4.5.1 Indexes . . . 55
4.5.2 Mask Object Transformation . . . 56
4.5.3 Dependency Object Transformation . . . 57
4.5.4 BULK INSERT . . . 58
4.5.5 BULK INSERT into Data Mapping table . . . 59
4.5.6 Load Data Mapping table in memory . . . 60
4.6 Summary . . . 60

5 Evaluation **63**
5.1 Data Quality Validation . . . 64
5.1.1 Completeness . . . 66
5.1.2 Correctness . . . 66
5.1.3 Consistency . . . 70
5.1.4 Interoperability . . . 71
5.1.5 Summary . . . 71

6 Conclusion **75**
6.1 Contributions . . . 76
6.2 Further Work . . . 76

References **79**

A Entity Diagrams **83**
A.1 Extended system dependencies diagram . . . 83
A.2 Extended entities dependencies diagram . . . 85

B User Interface **87**
B.1 Connection Settings . . . 87
B.2 Migration Dashboard — Home . . . 88
B.3 New Migration . . . 89
B.4 Application List . . . 90
B.5 Modules List . . . 91
B.6 Entities List . . . 92
B.7 Plan overview . . . 95
B.8 Migration Log . . . 96

C Data Access Configuration **99**
C.1 Test Connection . . . 99
C.2 Encryption algorithm . . . 99

D Entity Dependencies **101**
D.1 Dependency detection . . . 101
D.2 Automatic dependency detection - Recursive . . . 102
D.3 Dependencies order . . . 105

E Migration through an OutSystems extension **111**
E.1 Integration Studio Structures . . . 111
E.2 Get Physical Table Name . . . 112
E.3 Dynamic Insert . . . 113
E.4 Data Mapping - RecordIdRelation . . . 116

F Migration Log **121**

G Reuse Migration Plan **123**

H Differential Migration **125**

I BirdlyDataNotifications **127**
I.1 Send Notifications action - configurations . . . 127
I.2 Real email example . . . 127

J Migrate Users **129**
J.1 BirdlyDataAPI . . . 129
J.2 Inside Service Studio . . . 131
J.3 Inside extension . . . 135

K Mask Data **137**
K.1 Fetch entity's attributes . . . 137
K.2 Import attributes to EntitiesList Screen . . . 137
K.3 Opening an entity's mask popup window . . . 139
K.4 Create MaskAttribute records - database persistence . . . 139
K.5 Anonymize algorithm . . . 140

L Clean target entities **143**

M Migration API **145**

N Special Case: Self-referencing foreign key **147**

O Optimizations **151**
O.1 Mask Object 151
O.2 Dependency Object 152
O.3 Bulk Insert 153
O.4 Bulk Insert into data mapping table 157
O.5 Local data mapping table object 159

P Evaluation Details **161**
P.1 Migration of RocketContactLenses' data using BirdlyData 161
P.2 Correctness of remaining entities 163

Q Migration of Static Entities **167**

List of Figures

1.1 The Low-Code Development Process [19] . . . 3
1.2 Overall OutSystems platform architecture [15] . . . 4
1.3 DevOps lifecycle [14] . . . 6
1.4 DevOps in OutSystems [16] . . . 7
1.5 Production Bug by Cool Data Mover [3] . . . 9

2.1 ETL based architecture [32] . . . 15
2.2 Mapping Sample [32] . . . 17

4.1 Birdly Data Architecture . . . 39
4.2 System entities dependencies compress diagram . . . 42
4.3 Created entities diagram . . . 43
4.4 Migration Workflow Flowchart . . . 45
4.5 Attribute direct dependencies example of Employee entity . . . 47
4.6 Integration Studio Migrate Action . . . 47
4.7 BirdlyDataNotifications - Traditional Email Report . . . 50
4.8 Index of Mapping Table . . . 56

5.1 Rocket Contact Lenses Data Model . . . 65
5.2 Compare completeness of records migrated . . . 66
5.3 Compare Brand records after migration . . . 68
5.4 Compare User FK after migration . . . 69
5.5 Compare LensTypeAttribute after migration . . . 70
5.6 Compare LensType FK after migration . . . 71
5.7 Lens requirements by Type Query . . . 72
5.8 Source and Target environment behaviour and results of filter . . . 74
5.9 Compare generated XML from environments . . . 74

A.1 System Entity Relationship Model . . . 84
A.2 Global Entity Relationship Model . . . 85

B.1 Database Access connection UI . . . 88
B.2 Migrations Dashboard UI . . . 89
B.3 New Migration UI . . . 90
B.4 Applications List UI . . . 91
B.5 Modules List UI . . . 92
B.6 Entities List UI . . . 92
B.7 Automatic dependency detection UI . . . 94
B.8 Attribute list mask UI . . . 94

B.9 Migration Plan UI 95
B.10 Real time Migration Log UI 96
B.11 JavaScript to update MigrationLog screen 98

C.1 Symmetric key encryption [8] 100

D.1 Auxiliary Structures to manage list dependencies 103
D.2 Dependency List Example 104
D.3 Entities List Example 105
D.4 GetAllEntityDependencies server action 106
D.5 GetEntityDependencies client action 107
D.6 DeleteEntityDependencies client action 108
D.7 EntityOrderList action in OutSystems part:1 108
D.8 EntityOrderList action in OutSystems part:2 109

E.1 Integration Studio Migration Configuration Structure 111
E.2 Integration Studio Espace and Entity SSKeys Structure 112
E.3 Integration Studio Connection Structure 112

G.1 CreatePlanClone Server Action 124

I.1 BirdlyDataNotifications - Traditional Send Email 127
I.2 Real email 128

J.1 HTTP POST BirdlyDataAPI 129
J.2 API logic 131
J.3 GetUserDependencies 132
J.4 GetUsers 133
J.5 API logic 134
J.6 CreateUsersOnTarget 135

K.1 Import attributes 138
K.2 AttributeMask structure 139
K.3 UpdateMigrationPlan server action 140

M.1 Migration API 145
M.2 GetStatus API 146
M.3 OutSystems API definition 146

P.1 Creation of a new Migration Plan for RocketContactLenses migration 161
P.2 Selecting RocketContactLenses' entities to migrate 162
P.3 Migration Plan Overview of RocketContactLenses 162
P.4 Migration of RocketContactLenses 163
P.5 Compare VendorName records after migration 164
P.6 Compare LensType records after migration 165

List of Tables

2.1 Types of risks and proportions 12
2.2 Data quality properties validation 19
2.3 Data Movers Features 27

4.1 Bulk Insert - Performance View 59

5.1 Comparison of environments results by Lens Type: *Esférica* 73

P.1 Comparison of environments results by Lens Type: *Tórica Multi-Focal* 166

Abbreviations

API	Application Programming Interface
APP	Application
AWS	Amazon Web Services
BPT	Business Process Technology
CD	Continuous Deployment
CI	Continuous Integration
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
FK	Foreign Key
GDPR	General Data Protection Regulation
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
IP	Internet Protocol
IT	Information Technology
PK	Primary Key
QA	Quality Assurance
RAD	Rapid Application Development
REST	Representational State Transfer
UI	User Interface
URL	Uniform Resource Locator
XML	eXtensible Markup Language

Chapter 1

Introduction

The growth of data due to the increased availability of online repositories makes information technology (IT) companies confront the challenges of migrating data. Modern information systems must be able to fetch, integrate, transform and exchange heterogeneous data. Thereby, data migration is the process of moving data from one location (source) to another final location (target), which can have a different structure from the first. It begins with a deep analysis of the source data and ends up loading the transformed data into the target environment, reconciling the result with well-defined quality measures. To cope with the heterogeneity of the data structures it is essential to match and map data from the source system to the target system. Data migration has been a manual task for a long time. However, as database schemas become more extensive and, consequently, more complex, the data migration process has become very effortful, error-prone, and time-consuming. Building software can be costly, but when companies cannot find exactly what they need, they have no alternative than build a more flexible solution from scratch.

1.1 Cost of building software

The innumerable backlog of applications that some IT departments have to deal with is overwhelming. Even if IT were interested in build some in-house custom applications to boost productivity, they would not find time to do so. However, transforming internal processes is crucial to keep up with the market's ever-changing demands.

The cost of design and developing traditional or custom-made software to meet the company's needs will vary based on the project's scope. Nevertheless, it could range anywhere between $40,000 to $250,000, which is still significantly high [13]. Custom software can be an option for large companies when they need highly customized software, which is difficult to create using a low-code solution. However, for smaller companies, it is often an unfeasible option. Traditional development budgets involve application development from start to finish. At the same time, in

low-code platforms, the organization needs to pay for the access to that specified service, being this last option way cheaper.

Other factors include the software complexity, the required time-to-market, and the required skilled resources. Applications built with traditional development processes are usually highly complex, with various integrated tools and configurations. This inherent complexity requires highly expert resources, i.e., skilled teams with in-depth knowledge on the IT field to be able to operate with these apps accurately. With an inclusive low-code approach it is possible to develop the same application without the master skills in this field, allowing developers with varying levels of experience, like business users, to deliver apps within the required timeline. This way, IT team's backlog and workload are reduced, embracing the available time to work on more complex tasks.

1.2 OutSystems as a Low-code platform

> "Low-code platforms are on the rise, and with projected revenue of $180.7 billion over the next ten years, it's one of the hottest tech trends emerging in the wake of digital transformation." [11]

The low-code market is expanding, platforms offer an innovative and visual software development approach that enables the delivery of applications faster, known as Rapid Application Development methodology, and with minimal traditional hand-coding. This type of development abstracts and automates every step of the application development lifecycle through modeling and a graphical interface, which reduces the complexity, resources, and time needed to deliver customized applications to production.

According to Gartner [12], by 2024, enterprises will have adopted low-code platforms responsible for more than 65% of application development activity. This adoption was accelerated due to the pressure to deliver digital solutions to respond to the COVID-19 pandemic.

Low-code platforms help removing the need for technical expertise in development by offering drag and drop components containing the building blocks of programming [11]. Frequently, the stakeholders can iterate and follow the application progress. Their engagement and satisfaction through the lifecycle is also part of the results of developing in these platforms, which leads to higher return investment with lower risks.

The process of working with traditional development goes roughly through the same steps. Low-code development focuses on delivering something valuable, with fewer steps, as shown in Figure 1.1.

One example of a low-code development platform is the OutSystems platform. It is a solution for the complete application development lifecycle that also enables the continuous management of built applications. The platform can be used to build both web and mobile applications by employing an Agile methodology that is the primary approach companies are investing in to accelerate application development [17]. OutSystems allows developers to stay away from the inherent complexity of implementing a traditional, hard-coded solution.

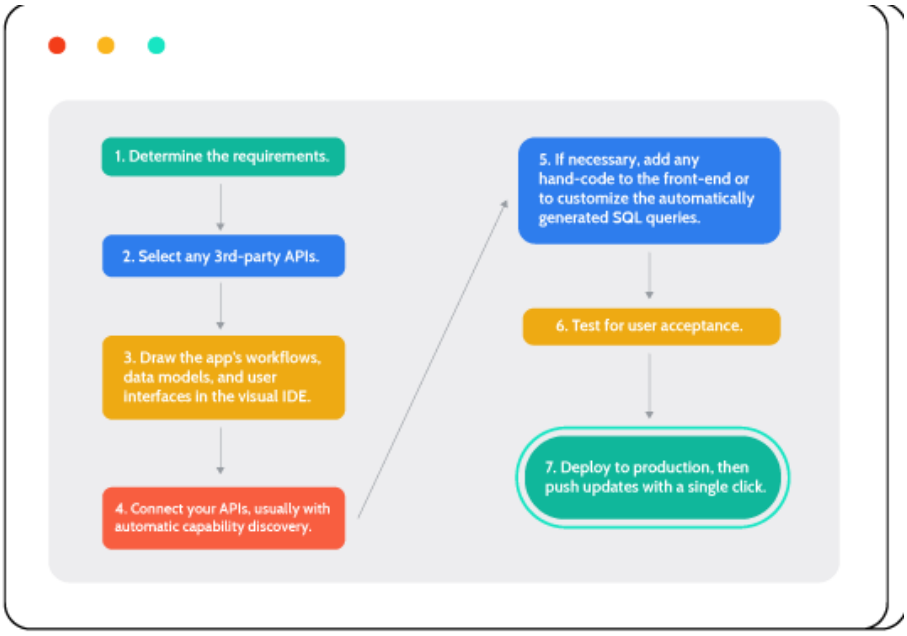


Figure 1.1: The Low-Code Development Process [19]

The operating system required to run OutSystems is Windows with editions supported by Microsoft. Regarding the database management systems, OutSystems integrates with external databases as part of the application data layer, allowing to use data from those databases in a transparent way. The public cloud platform supported external databases are Microsoft SQL Server and Oracle RDS databases.

Internally, OutSystems is divided into five components (Platform Server, Service Studio, Service Center, LifeTime, and Integration Studio), covering all lifecycle steps. The components that are part of the development environment are:

- **Service Studio** — low-code and visual development environment IDE used to create all parts of the application: application logic, create user interfaces (Traditional or Reactive Web applications or Mobile applications), define data model, define business processes and timers, manage dependencies and connect to external code.

- **Integration Studio** — allows developers to create extension modules, i.e., creation of custom components to integrate with existing systems, databases, and code. These extension modules are then consumed by Service Studio modules for use in the applications.

Platform Server is the core of the OutSystems platform since it generates optimized code and orchestrates its compilation and deployment. LifeTime, a centralized console for managing the application versions, IT users, and security, is used to manage the OutSystems infrastructure and applications lifecycle. On the other hand, Service Center is used to manage OutSystems environments and configure settings in different functional areas, such as operation, runtime, and compile-time settings.

Service Studio can also be used to install applications and components from the OutSystems Forge, which basically is a repository for open code modules made by other developers and ready to be used. The reusable components and business solutions can be helpful in some projects, which allow saving development time.

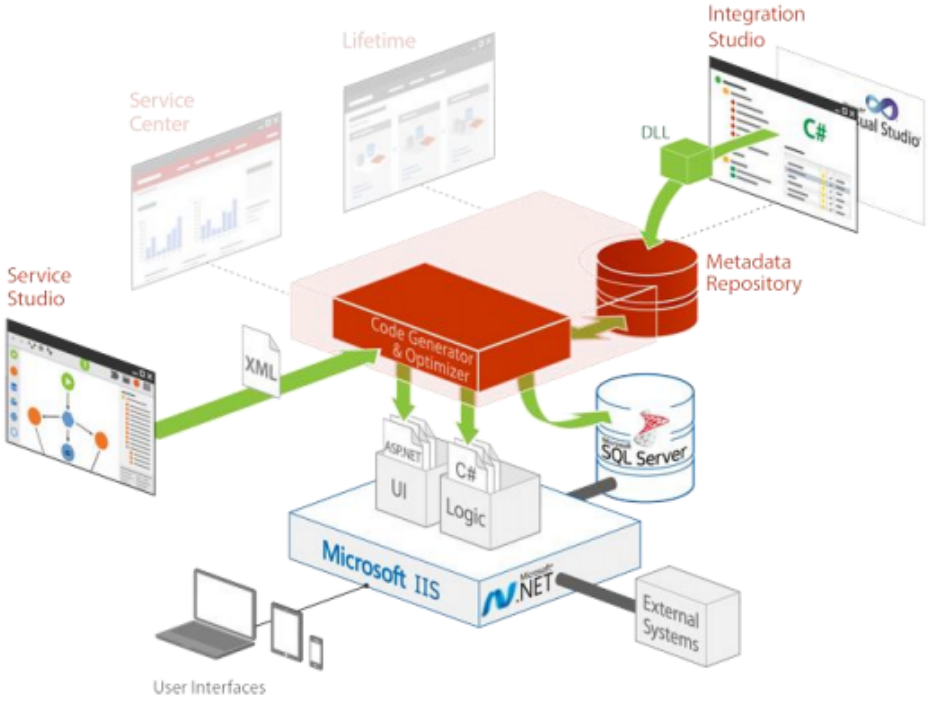


Figure 1.2: Overall OutSystems platform architecture [15]

Integration Studio interoperates with Visual Studio to allow developers to extend the platform using C# .NET code, providing code extension. The overall platform architecture is presented in Figure 1.2.

There are two types of OutSystems applications: Traditional Web and Reactive Web Apps. Reactive Web Apps was a 2019 upgrade created to enable developers to develop the next generation of Web Apps. Both types have inherent differences and benefits as well as different available widgets and functionalities. OutSystems applications are composed into ***Modules***, which provides abstraction by splitting functionalities into different modules, creating an architecture design that is easier to test, maintain and evolve. It also supports the correct isolation of distinct functional modules in which is easier to manage complexity as the developer tackles minor problems. The OutSystems *modules* can be:

- **eSpaces** — which are built-in Service Studio and allow the implementation of the several

layers of the application. This *modules* can have different types such as libraries and services, mapping to the concept of core services in the 4 Layer Canvas[1].

- **Extensions** — which are built-in Integration Studio and which aim to facilitate integration with legacy systems or external data sources.

The result of the OutSystems architecture design framework is a set of modules with more independent lifecycles and without cumbersome dependencies, where each layer in the framework addresses a different aspect of the functionality in a module.

1.3 Motivation

In search of agility and quality in software development, there are methodologies that are capable of enhancing the performance of the professionals involved in their processes. Therefore, companies have increasingly adopted these methodologies so that they help them achieve their goals of coordinating and collaborating to produce better and more reliable products. DevOps is one of these methodologies. Teams that adopt it gain the ability to better respond to customer needs, increase confidence in the applications they create, and achieve business goals more quickly since they build, test, and release software faster and more reliably. The set of cultural practices and values aim to:

- Accelerated time to market
- Adaptation to the market and competition
- Maintenance of system stability and reliability
- Improvement of average recovery time

DevOps influences the application lifecycle (Figure 1.3) through planning, development, delivery and operation phases by implementing *continuous integration and continuous delivery (CI/CD)*. The implementations, then, occur more frequently and in smaller increments, thus result in more agile teams that make software available in short cycles, facilitating risk planning and adapting to changing customer needs.

Rapid Application Development (RAD) teams pretend to improve IT sector deliveries through the evolution of the Agile methodology. RAD is an iterative and incremental software development process model that emphasizes an extremely short development cycle. For this reason, RAD teams are adopting the DevOps culture, practices, and tools in order to become highly productive, creating better products faster and increasing the customer's satisfaction. Since RAD teams' purpose is to implement rapid application development and delivery, they often use the OutSystems platform combined with DevOps to provide better software results. Figure 1.4 presents the integration of the DevOps with the OutSystems set up.

[1] Architecture canvas that reuse common modules. More on [45]

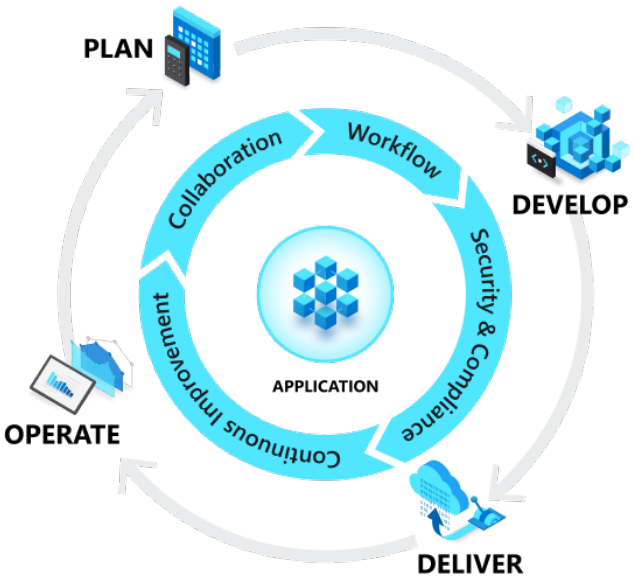


Figure 1.3: DevOps lifecycle [14]

Proper DevOps capabilities allow developers to focus on the business value and not specifically on the technology used. OutSystems is agile enough to integrate many systems that currently provide DevOps capabilities, i.e., it can handle the full cycle of the application life by incorporating the DevOps core values. The low-code platform accomplishes this through a variety of means such as:

- Integrated SOAP/REST web services
- Native OutSystems API's provided by the platform
- Diversification of community connectors available
- Integrations to external systems

Besides, continuous integration uses automatic tests, which are executed whenever new code is consolidated so that the code in the main branch always remains stable. Overall CI/CD includes complete automation of all steps between code consolidation and production implementation.

Despite that, automatic tests are limited in the OutSystems platform since it does not provide an automated mechanism to create reliable test data, which is the input feed for testing the application and helps the developers to find problems during fixes. A complete dataset should transverse different scenarios, but these scenarios are often difficult to create using only the tester imagination because there are infinite ways how the customer can use the software product. Nevertheless, reliable data can be found in other OutSystems' environments, like a production environment, where the data is created by real customers and usually cover different and unexpected scenarios.

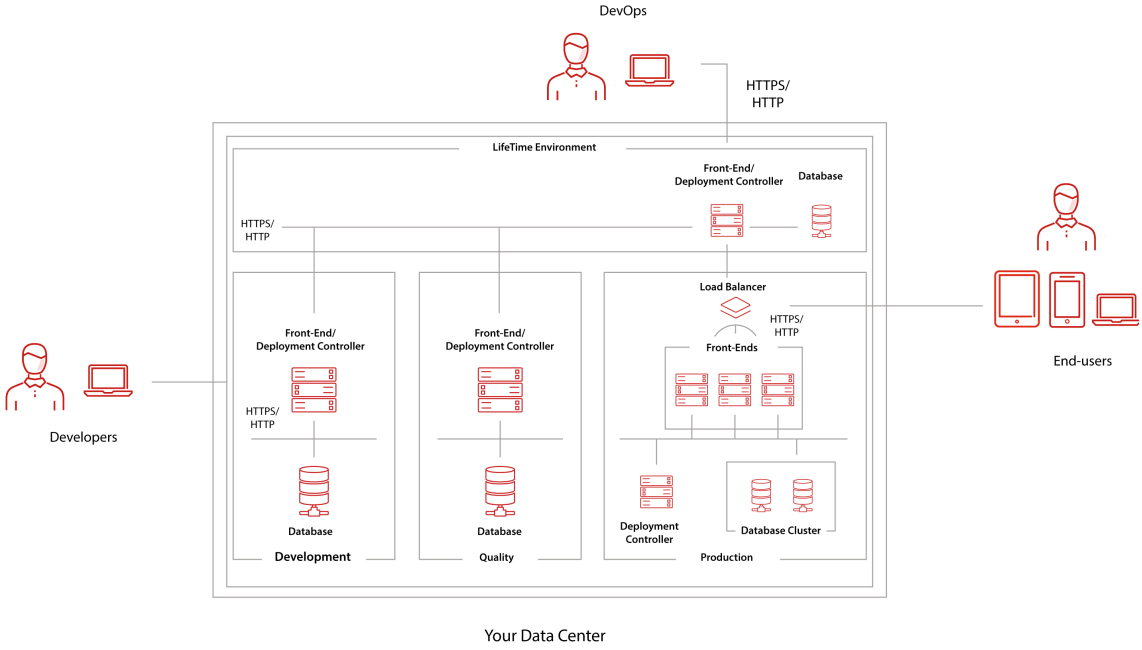


Figure 1.4: DevOps in OutSystems [16]

The data migration process of using data from other environments and automatically transfer it to the testing environment can be the key to provide automatic tests with less human intervention since there is no need to manually create test data.

To keep up with the software demanding, the need to evolve the DevOps model and enable automatic tests in OutSystems applications with reliable data arises. With that purpose, a data synchronization tool should be implemented for the OutSystems platform to consequently save time, money, and resources throughout the process.

1.4 No operational data for development testing

OutSystems changes the way software is built to rapidly create and deploy critical applications that evolve with the business. In general, RAD approaches to software development put more emphasis on an adaptive process. Therefore more projects are completed on time and within budget. Developers rapidly build and deploy their applications using OutSystems. They could be running smoothly in production, but then the first bugs are logged. Flaws can lead to serious security issues later since they are unexpected hardware or software problems that can be prejudice to the correct functioning of some technology.

Provide application updates sooner and moving faster is the main motive that companies switch to DevOps. Software developers are pressured to continually release new features, as a result, they often push code updates as soon as they finish coding it, without even testing. The classic time-consuming development, Quality Assurance (QA), and testing have been compressed to days and sometimes hours. In consequence, teams no longer perform extended beta tests with

customers to catch real-time bugs. Frequently, developers don't know the behaviour of the software after deploying a new software version in production or whether it will even work. As development cycles are increasingly more frequent and fast, makes typical QA testing no longer sufficient. The QA testing methodology commonly used is a set of simulated test suites: simulation is a close imitation of reality but hardly gives a comprehensive insight into the many ways in which customers may actually use the software. Moreover, even if some code works under a set of testing parameters, that does not mean it will be the same in the uncertain customer usage that is the production environment. In "The Bug in Production: What You Don't Know Can – and Will – Harm You" [33], Frank Huerta explains why bugs are detrimental and could lead to errors that bring down the entire service, he stated that:

> "It is estimated that fixing flaws after a software release can be five times as expensive as fixing them during design—and it can lead to even costlier development delays... Software and DevOps teams need a way to test quickly and accurately how new releases will perform with real (not simulated) customer traffic and while maintaining the highest standards."

After finding the bug in the production environment, the developer gets back in development, but it's hard to reproduce the bug. In fact, for understanding and reproducing a bug, it is essential to have detailed *Steps to Reproduce*, that is the path to follow to experience the same bug. These steps include user information that can directly or indirectly cause the bug. Even that data migration solves many issues, this dissertation focus on this particular scenario, when the developer needs more accurate data to test and debug an application properly. The problem is the **"right" data** is often **missing in a non-production environment** and developers need application data from every environment for debugging and testing purposes. But testers need to enter it manually and extensively.

At the OutSystems platform, the developer can only manipulate and visualize the current environment data where he is currently developing code. For example, in case he is working on an application inside the development environment, he can not have access to the same database table information on the test environment, which affects the migration task since the data is inaccessible. There is a figurative "wall" between environments that do not let any information pass between them, as represented in Figure 1.5.

Furthermore, developers who are OutSystems clients do not have any administration permission for data synchronization. What makes data migration unfeasible in the OutSystems platform, since the database can only be managed by the platform itself.

1.5 Goal

Considering the current lack of OutSystems support for data migration between different environments, the objective of this work is to **produce a framework able to automatically migrate**

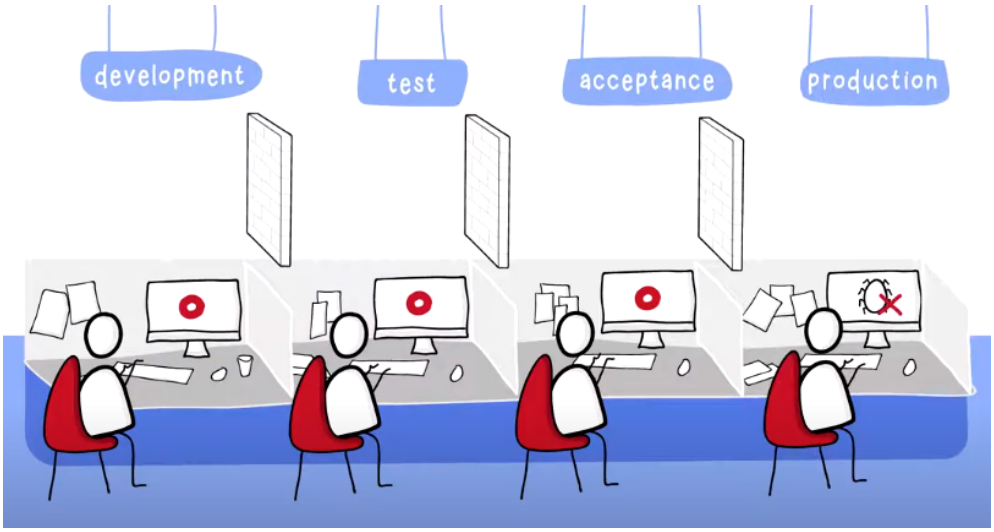


Figure 1.5: Production Bug by Cool Data Mover [3]

database tables (OutSystems Entities) for a given application built with the OutSystems platform. With this, it is intended to synchronize information in order to evolve from the DevOps model and enable automatic testing using reliable data and consequently save time, money, and resources.

The proposed goal aims to fulfill the following objectives:

- build a web application in OutSystems to be used as a visual auxiliary part of the framework with an easy-to-use interface where the user can decide which application entities to migrate between system environments.
- allow the user to choose different database connections for the target and run the framework in different environments to change the database source.
- build an application extension in .NET able to copy and transform the source table data and insert it into the corresponding table of the target database.
- mask sensitive data by transforming some entity's attributes to provide General Data Protection Regulation (GDRP) compliance. The user should be able to identify which attributes are considered sensitive for each entity.
- should correctly map the databases in different environments by providing a priori validations.
- provide some integrated mechanism to visually keep up, in real-time, with the migration process since OutSystems does not provide any support for data migration.
- integrate a re-use feature to clone migration plan to avail from the migration plans that were already created.

- migrate, when necessary, the *User* system table that is commonly used by other entities which require special treatment and cannot be migrated through the extension.
- implementing a differential migration mechanism, when repeating a migration, only the latest data should be migrated since it avoids overprocessing and duplicated records.
- add a set of additional non-functional features to the framework to improve the migration performance because some application entities have a huge amount of records that should be migrated as faster as entities with fewer records.
- validate the framework using real applications.

1.6 Document Structure

This document is organized as follows: Chapter 1 presents the introduction, the motivation, and the problem at discussion and proposed solution, as well as some key concepts about the OutSystems platform. Chapter 2 presents the related work in the field of study of data migration. Chapter 3 presents the relevant issues around data migration using OutSystems and the proposed solution will tackle them, and Chapter 4 the details the implementation of the proposed solution. Finally, in Chapter 5, the results of the evaluation process while Chapter 6 concludes this dissertation with an analysis of the work developed and the proposed solution, as well as future directions of research.

Chapter 2

Data Migration

This chapter outlines overall processes to build an efficient and automatic data migration system, progressively funneling to more detailed guidelines inside the OutSystems platform and available software. Additionally, relevant work will be presented along with some key concepts behind data migration.

2.1 Need for Data Migration

Persistently expanding information size and new prerequisites in information handling have encouraged the creation of many new database systems. Data migration has become one of the most requesting propositions for IT organization supervisors, making migrations of data for business purposes a systematic process. Although replacing a legacy system is the most common reason, many organizations adopt data migration for several reasons.

Simanta [54] stated some reasons to perform data migration:

- Replace an old legacy system, overhaul an entire system.
- Utilize clean and accurate data, e.g., for testing purposes, failure recovery.
- Need for additional storage capacity due to the continuous exponential database growth.
- To limit cost and decrease complexity by moving to a consumable and consistent system.
- Data needs to be transportable from every environment for virtualization.
- Load balancing, to allow backup systems.
- System expansion, data migration improve scalability.

2.2 Migration Risks

In spite of earning such high business benefits, as higher productivity, costs reduction, and ability to manage data, data migration projects are frequently associated with a high level of risk due to the transference of critical data and to the oversized volume of records [48]. Data migration is not a one-time task, it is something that enterprises repeatedly do. The investment should be permanent since it consumes a large amount of the IT budget due to its regular frequency and resources consumed. With the evolution of the storage infrastructures, data migrations are becoming labor-intensive, risky, and complex.

A survey conducted by Hitachi Data Systems [22], represented in Table 2.1, identified many risks for data migration projects. The authors highly recommended that IT executives begin managing this growing problem more effectively through evaluation of new possible storage platforms and migration solutions/planning to solve the expensive problem involved in data center operations.

Table 2.1: Types of risks and proportions

Proportion	Types of risks	Description
66%	Unexpected or extended downtime	Increasing density of applications due to virtualization can lead to loss of availability or access to the data.
64%	Schedule Overrun	Projects last longer than expected and consequently resources too, this can also delay other IT projects.
39%	Budget Overrun	Can affect other IT projects budget and profit-loss for business.
25%	Costumer Impact	Negative public relations impact if any critical customer data is lost or unavailable. Some are interconnected applications where one application data loss affects multiple other applications.

In an organization, when the application data has critical information about the customers, some additional concerns have to be handled. General Data Protection Regulation (GDPR) is a privacy regulation that lays down rules relating to the protection of natural people with regard to the processing of personal data and rules relating to the free movement of that data. Most importantly, this regulation defends principles relating to the processing of personal data. While performing data migration, it is essential to pay extra attention to the process of moving sensitive data since the data can be accidentally lost, destructed, damaged, or accessed/processed by unauthorized or unlawful people. Having a software program that is GDPR compliant means that the data should be processed in a way that ensures proper security of the personal data, using suitable technical or organizational measures (integrity and confidentiality) [7].

Big data drives most modern computer systems, and these modern systems are expected to

be running without interruptions. For that reason, the data migration approach needs to be well-established with continuous processes designed in an adequate manner. The lack of a competent data migration plan can lead to application downtime and performance degradation derived from overwhelming data processes, over-budgeting, technical incompatibilities as inaccurate, redundant, corrupted, and data loss, or even data operations functioning below the initial expectations. As mentioned at the proceedings of the FAST conference on File and Storage Technologies [37], incomplete plans can cause migration projects to fail altogether. To prevent these types of issues, there is a need to develop a steady and reliable methodology that permits to design, plan, relocate and approve the process.

2.3 Data Migration Strategies

The migration process must cause less disruption to the business environment as possible. In a pre-analysis phase, the decision to follow a *big-bang*, also called *cut-and-run* or *source-push*, versus a *target-pull*, also called *trickle*, approach should take place [41, 32, 25].

- *Big-bang/Cut-and-Run/Source-push* - Migrates all data of the old system into the new in one operation, i.e., migrate what the source offers, a single defined cut-off point. It consists of switching off the source system and turning on the replacement system immediately. Live systems experience downtime, in which the system is unavailable while data is migrating. The ideal *big-bang* implementation has zero downtime, but in reality, this cannot be guaranteed since there are resources that remain offline, which leads to high business risk. This could also be very uneconomical for complex applications since it is too expensive to analyze every attribute.

- *Trickle/Target-pull* - Migrating only data necessary for the target system, i.e., migrate what the target requires to reduce the risk involved. This strategy is performed in small and incremental steps (sub-process). If a single step fails, it's only that step that needs to be repeated. Both source and target remain running in parallel, keeping data continuously migrating with no interruptions through a step-wise progress. These implementations, however, may require more complex planning and design.

The decision should be made according to the project data volume, deadline, budget, if the systems can experience downtime and if the target database already contains data. Also, a concrete strategy could be a combination of these approaches to different components of the applications. The risk inherent to the complexity of the incremental migration and the risk of the migration itself should be balanced to complete a successful migration.

Philip Russom, the author of best practices in data migration, defined *big-bang* strategies as risky and should be avoided. He advises breaking the project into a series of manageable chunks/sub-processes [53].

2.4 Data Migration Architecture Approaches

The data migration topic is essentially addressed by IT professionals. Ongoing publications originate from practitioners, consulting professionals and the tool vendors community. Brodie and Stonebraker first proposed the *Chicken Little* methodology [26], allowing the source and target systems to inter-operate during migration. This approach consists of an eleven-step generic migration strategy exploiting complex gateways that mediate the operation software components. A forward gateway is used for source applications to have access to the database environment in the target side of the migration process, this is possible by translating and redirect calls forward and return results. A reverse gateway enables mapping the target data to the source database. Data is stored in both the source to migrate and the growing target systems, consequently, the target system will grow as the migration progresses. Unfortunately, this approach may implicate data duplication, loss of integrity/consistency across databases since the source and target system form a composite information system during migration.

Another tailored pioneering process model for dealing with data migration is the *Butterfly* approach [56]. The main objective is to guide the migration of legacy data in a mission-critical environment. This methodology provides consistency between source and target systems by eliminating, during the migration, the need for simultaneously access both information systems. The gateway-free methodology relies on six phases: (i) Prepare for the migration (requirement engineering). (ii) Analyze and redirect all manipulations of the source and develop the target data schema(s). (iii) Build up a Sample Datastore, based upon the target system data model in the target system, to support the initial development of its components. (iv) Incrementally migrate all the components (except for data) of the source system to the target architecture, using Sample Datastore to design/develop/test newly developed target components. (v) Gradually migrate (step-wise migration) the source data into the target system by introducing a series of TempStores (temporary data stores). (vi) Cut-over, target system is up and ready to run.

These methodologies add more complexity to an already complex process to be actually applied in practice. With the growth of the system's coupling and interactivity, these approaches consume many resources and become difficult and expensive.

Morris provided an extensive work about the whole data migration project management for beginners [44]. This complete set of modules, *PDMv2*, involves contracting external suppliers, source data analysis, data mapping, data quality/testing, extract transform and load (*ETL*) design, migration execution, and legacy retirement (cut-over). It consists of seven modules, each one related to the services of a data migration delivery. However, *PDMv2* is very business-focus, and there is only a high-level overview of the technical issues.

In the context of the *ETL* - extract, transform and load process, any data migration will involve these steps. This means that after extracting the data, it needs to go through a series of functions in preparation so that it can be loaded into a target location. Simple import and export procedures will not work when the source and target databases are different in their internal structures. The data, when migrated, suffers a transformation/restructure such as change of the format or fields

being merged. In other words, data migration is not a simple data movement since it also requires transformation. In practice, *ETL* is considered a critical phase of the data migration strategy, and it is the factor that defines the quality with which raw data is transformed into legible, relevant, and reliable information. Therefore, *ETL*'s main objective is to ensure that there is a strategy for systematic use of data. This involves mapping processes, which is nothing more than establishing rules for data handling to occur in a standardized way. The use of a good quality validation *ETL* tool can minimize the risk of data defects migrated to the target database by providing data reliability.

Years after, Haller [32] purposed another approach centered on *ETL*, represented in Figure 2.1 This approach involves three databases [41]:

- **Source staging database** — This database stores the source application database, as a copy, to decouple the data migration project and the old system. Prevents the downtime of the source application, it stays untouched and is still running.
- **Transformation database** — It stores intermediate results of the transformation step. It is helpful to join, compare and filter data without replacing the actual objects.
- **Target staging database** — The target stores the transformation result as an upload staging database structure to insert on the target database table.

These instances can be joined in one physical database that stores all of them using different schemas.

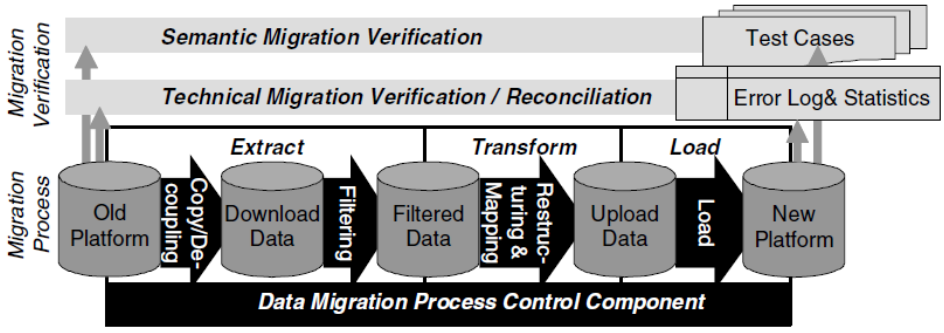


Figure 2.1: ETL based architecture [32]

The migration process progresses as follows:

- **Extract** — Firstly, the extract step downloads the data by copying it to the download area. This download sub-step preserves the running business (decoupling) since no modification is done to the source. Secondly, the downloaded data is filtered by the relevant records being migrated. Any records passing the filter must be in the target system after the filtering sub-step. This step is essential to exclude redundant records or records that are already on the target database. Three main filtering patterns are introduced by [32], these filters consist of different SQL queries.

- **Transform** — If the old and new database systems differ, then the filtered data goes through the transformation step, where it is restructured to match the target database. The matching process, also called data mapping, takes both source and target schemas as input and produces a mapping between elements of both schemas that correspond semantically to each other [52]. Most mappings include data transformations to perform standardization, provide quality, conflict resolution, and exception handling.

- **Load**— The transformed data is inserted/loaded in the target database. Feedback takes place in this step through error detection for each object regarding if it was successfully migrated or, in case of error, the reason of failure shall be given.

At the end of the data migration process, data is validated. Both validation as well as migration verification techniques that allow verifying if the migrated data is correct and complete are detailed in Section 2.6.

2.5 Data Mapping

Since the schemas are independently developed, integrating them can be difficult due to their different structure and terminology. Even schemas used for the same concept may have naming and structural differences. These schemas can even be expressed in different data models or have similar words with different meanings. To integrate the schemas it is necessary to, at first, identify the interschema relationships, also called data mapping [52]. This relationship composes the basis for translating the data. Data mapping consists of creating a link between two different data models, linking the fields in the source and the fields in the target, and define the transformation logic and rules.

Mapping table [32] is one of the most common implementation patterns. These tables stores the value of the source and the corresponding target value in each row. It corresponds to the transformation database mentioned in the previous section. Figure 2.2 shows a sample of this mapping table, *MAP_COUNTRY*, where the link between the source, *CUS_OLD*, and target, *T_COSTUMER* is stored. When the data is being migrated, this table should be consulted, and every time that one of those values appears on the source database, it should be transformed to a new value, that is the target's match. This new value is determined by a join between the source and data mapping table. Some transformations can also be performed by a mapping function, Figure 2.2 provides an example of a function that merges the *CONTRIBUTION_MARGIN* and *ASSETS* source fields into *CLASSIFICATION* field on the target schema. These functions usually contain business logic or technical transformation rules.

Other common examples are reformatting customer data where the source database has separate fields for the surname and forename, meanwhile, target only has a single field for the full name. This transformation requires the need to split data and merge fields [44]. A mapping

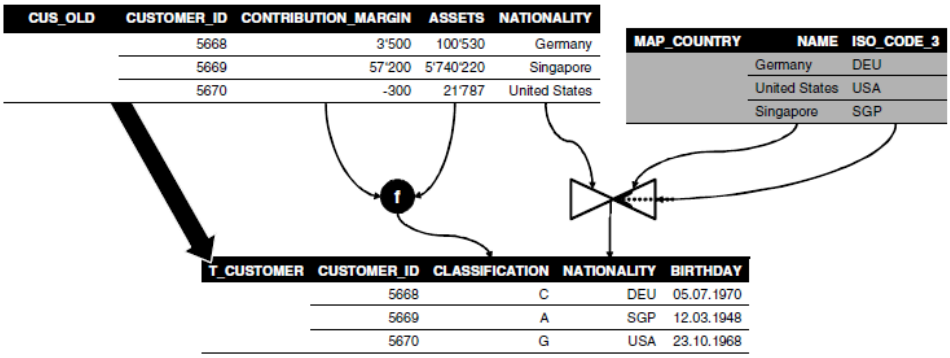


Figure 2.2: Mapping Sample [32]

element with the expression *"Concatenate(SourceTable.Forename, SourceTable.surname) = TargetTable.Name"* describes a mapping between two source schema elements and one target schema element.

Some IDs and data structure change during the data migration, this problem come up to be very complex and challenging to map the primary keys and foreign keys from source to target databases [41]. To prevent transformation failures, a mapping table containing the link between source and target IDs can be created. When these IDs are changed, the table should be updated. This mechanism allows to track the changes and have the right and updated data all the time. A mapping between source and target database schemas should contain a mapping element relating *SourceTable.ID* to *TargetTable.ID* with the mapping expression *"SourceTable.ID = TargetTable.ID"*.

For a long time, schema matching is being performed manually. But as systems have become more complex, so as the databases, the schemas become larger and more complicated due to the increasing number of matches to perform. The IT professionals allocated to data mapping need to have detailed knowledge of the application domain and complex data transformations. Manually specifying schema matches has become labor-intensive, tedious, very time-consuming, error-prone, and therefore expensive.

Some generic, semi-automated implementations of Match built to manage heterogeneity were proposed by [38] "Cupid" (Microsoft Research), [30] "COMA" and [23] "COMA++" (University of Leipzig), [42] "Clio" (IBM Almaden and Univ. of Toronto), [43] "Harmony" (The MITRE Corporation) and [21] "Microsoft BizTalk Mapper" (Microsoft) which is embedded in Microsoft Visual Studio and allows manual mapping through a Graphic User Interface (GUI)-based editor with some automatic support for the determination of match candidates. Most of the latter have a hybrid match approach, which means that the implementation of Match, which suggests correspondences, uses different criteria and multiple match algorithms. These "matchers" provide customization since they can be selected according to the schema type or application domain. With or without software, manual configuration is almost always necessary to perform a successful data mapping. The individual "matchers" can be classified into instance-based matching,

structured-based matching, linguistic matching, constraint-based matching, and auxiliary information [52, 24]. An instance "matcher" uses statistics, metadata, i.e., data contents. A structure "matcher" decision is based on similar relationships, it is considered a match if the schema elements appear in similar-structured groups. Regarding the linguistic "matcher", this is based on semantic similarity, names, and textual descriptions of schema elements like keywords and relative frequencies of words, while constraint "matcher" is based on relationship types, averages, data types, values ranges, character patterns, and nullability. Finally, the auxiliary information "matcher" relies on additional information like dictionaries, user inputs, previous matches, mismatch lists, or thesauri (repositories of lexico-semantic knowledge). To compare larger schemas is also possible to combine multiple matching algorithms (combination of multiple individual "matchers"), most recent tools support the combined use of different "matchers" like linguistic, structural, and instance-based.

ETL systems are a special case of mapping tools since they often provide a GUI and produce transformation scripts as final output. Typically, complex transformation logic must be developed as external ad-hoc functions, coded outside the tools, through programming interfaces [27]. The authors consider ETL tools as a solution with several drawbacks and introduces the Data Fusion DM component as a workaround. This component offers an Interactive Development Environment that communicates with the sever on the data staging area and uses a standard Data Transformation Language (DTL) compiler that generates java code derived from DTL mappers. However, it assumes that the source-target schema mappings are already known, which is not typically true. To first discover correspondences, another data mapping tool, like COMA, should be used.

Some reviews and comparisons between known schema matching tools were also made by [52] and [24]. In [52] a full generic high-level architecture implementation of *Match* is provided with a classification criteria for different schema match approaches. Meanwhile, in [24] are listed some complex techniques and how they are used.

Data migration projects involve managing a large amount of data, which means that the probability of some of this being critical data is big, and consequently, more transformations need to be performed. Enterprises need to be sure that the entire set was migrated and the data is correct, i.e., the transformations cover all migrated records. Once the mapping is done, and the generated correspondences are indeed those intended, then they can be converted into a transformation script, which aims to translate the source to the target schema instance.

The use of a data integration tool with GUI instead of hand-coding is considered by [53] to be a good practice given the complexity of the data mapping process. The graphic support is important for viewing mappings [57].

2.6 Data Quality Validation

To implement a successful migration of high-volume data and reduce risk is necessary to consider how it will be validated. The process of data validation testing is what ensures the user that the data migrated is correct and its integrity was kept along the migration process. A Quality Assurance

(QA) strategy is essential at any migration project since corrupted data is of no use to the user. Data validation testing assists completeness, semantical correctness, consistency of the data, and integration with other applications [32].

Table 2.2: Data quality properties validation

Data Property	Data Validation
Completeness	Identify missing records in the target database or unexpected new records, also called migration verification or reconciliation testing. Covers all data set.
Correctness	Compare records from the source and target application should verify that fields' values are migrated according to the migration specification. An example of an error could be by having the same number, but with a different currency, the semantic should be checked for different test cases. This comparison is usually made with appearance tests.
Consistency	Test how the target application interplay with the imported data. Incompatibilities between the parameterization of the target application and the migrated data can be found with processability tests. The application can crash if, for example, in the case of a currency type of field, the database schema does not impose to set a currency, and the data migration writes directly into the target database table.
Interoperability	Test how the target application with the imported data interplay with other applications, this is possible with integration and end-to-end tests.

As mentioned in the Table 2.2, to identify missing records (completeness), the technique used should cover all data set, while the others can be tested with shorter portions of the data set:

- Sampling — The downtime that it would take to check all possible test cases makes it unfeasible. Randomly select a small number of elements of a large data set is a usual technique to purpose valid conclusions. However, this technique presupposes that the error is uniformly distributed and requires resources to manually compare source and target data.

- Query — An automated approach can be using SQL queries to check missing values. Creating a query that selects the source data' MINUS' (SQL operation) the target-side data allows the tester to check lost values during the migration. To check if any unexpected results appeared on the target data set, the tester needs to do the inverted process. It is also possible to create queries to check duplicate information and filter records for better visualization [55].

2.7 Data Migration Types and Software Environments

Data migration is defined as the transference of data from one computing environment or storage system to another.

As mentioned in Section 2.1, there are many reasons enterprises might need to undertake a data migration project. The most frequent types of data migration performed by organizations are application migration and consolidation, where legacy systems are replaced by new applications, but the database remains the same. Nowadays, data migrations are also performed to move from on-premises IT infrastructures to cloud-based storage and vice-versa. This type of migration between infrastructures is usually implemented to improve optimization.

On the other hand, application migration is the process of moving an application program from one environment to another. This type of migration used to move applications and data to more favorable environments is also called data migration between environments. This type of migration is performed with the aim of adapting the application data to a new environment, making it compatible with the target platform.

Environments are used by software development teams to create "stages" as they progress apps through the development process. Each environment has its own unique purpose, but generally, the process starts at the Development stage and ends with the Production stage.

The process of migrating data from Production to Non-Production environments is very common to automate tests by providing proper data to test different case scenarios. Both environments are logically and physically separated.

The production environment's main purpose is to make applications available to end-users. It contains the customer data. The priority in those environments is reliability, it must always be in working condition, bug-free and available all the time. Unfortunately, production data is not replicated or used in non-production environments unless performed a data migration. The non-production environment's main purpose is to test and debug applications and mitigate the risk of deploying them to production (release). The data provided should be similar to a real scenario (similar to production).

Certain bugs can log in production due to the lack of proper testing. If the mocked data used in tests is not real enough, it can be very hard to reproduce real-life scenarios and, consequently, reproduce those bugs inserted in production. Mocked data can also be very time-consuming to create and is never realistic as the customer data.

Regardless of the target location is a public, private, hybrid cloud, or on-premise, there is a need to find a cost-effective, secure, efficient and possibly automated method of migrating data to the new storage location.

2.8 Data Migration Automation

To enable data mobility, there is a need to transfer data between systems as easily and efficiently as possible to minimize the impact of the migration on the system. Has mentioned in Section 2.3, sometimes the large size of the data objects and a large amount of total data makes migration a process which can easily take several days and is usually performed on a weekly or monthly basis. Identifying the right solution to a data-driven decision process can require several rounds of experiments with large data sets and the adoption of many different technologies, as well as highly

skilled persons. These many requirements lead to a significant amount of time and budget spent along the process [28].

Traditional implementations created for moving data either imply application breaks or strongly impact the performance of other running applications [37]. Perform a data migration from scratch is challenging, incredibly time-consuming and hard to reuse for other projects or cases. The right data tool can simplify migration, performing it efficiently and cost-effectively.

The data framework used to simplify migration should be:

- Scalable — Knowing the data limits of the software and predict if future data exceed those limits. These tools are normally used to migrate large data sets.
- Secure — Data is one of the most valuable resources, and it must remain protected. Data corruption is one of the major data migrations problems.
- Fast — The platform should quickly process large amounts of data. This is a more secondary factor, however, if the software takes days to perform a data migration it would probably not have value to the customer.

One data migration project usually involves different types of technology support, a complete solution would provide data mapping, automated data migration execution and control, and data validation/quality automation. The migration controllers are commonly referred as the ETL tools, they should be able to extract and validate data from the source database, transform/reformatting the extracted data, load the data to the target database, provide a full report of the execution and in case of error provide fallout feedback and manage fallback if the migration was not succeeded then the data should revert to a previous safe state [44].

ETL tools facilitate the data migration process by offering robust features for data profiling, transformation, quality, cleansing and auditing. Oracle Data Integrator (ODI) is a commercial licensed ETL tool that provides interfaces to migrate data from the source to the target database and support a range of data sources. It also monitors the data migration execution and alerts users of the status. The error handling capability on the drag-and-drop functionality makes the perform of the tasks easier. The tool identifies faulty data automatically before moving to the target application and directly loads and transforms data at the target database, which improves the performance because avoids building separate staging infrastructures [36]. ODI comes in both cloud and on-premise versions. Informatica – PowerCenter is a commercial licensed tool for ETL workloads. It works as a data migration, data cloud integration, data analysis - masking and quality, data replication and synchronization and data virtualization but focuses on Business-to-Business (B2B) data exchange. It provides a Metadata Manager to build data flow maps. PowerCenter supports a huge volume of data from any database type [34]. It has a high-performance reputation, however, the tool's high price and difficult learning curve are an obstacle for smaller organizations.

New platforms emerged recently, some of the most popular and cost-efficient are Xplenty and IRI Voracity. Xplenty is a licensed cloud-based tool that provides a complete and simple visual interface for building data pipelines across a wide range of sources and targets, including SQL

data stores, NoSQL databases and cloud storage services. The software processes both structured and unstructured data and handle scheduling, monitoring, security and maintenance, allowing the user to focus on the data [20]. Voracity is a versatile data management platform that combines data discovery, analysis, integration, cleanse and migration within the metadata framework on Eclipse[1]. The data can be structured or unstructured, static and streaming, on-premise or cloud. Voracity includes a standalone data migration utility through IRI NextForm [10].

Nowadays, exists a large list of commercial software, in [51] the authors affirm that is "Needless to say the price tag for those tools often ranges between $50 and $100k per license and includes a lot of unnecessary features. In today's economic conditions when organizations take a closer look at their bottom line trying to find ways to cut costs, it becomes especially relevant to find a leverage to negotiate better deals with software vendor.". Open source tools have become more mature and can be cost-effective alternative to the close source, expensive, proprietary solutions.

Talend Open Studio (TOS) is a free open source software compatible with data sources both on-premise and cloud, that supports data migration, as well as data profiling and data integration. It simplifies ETL processes for large and multiple data sets which ensures optimal execution and scalability. This solution has many inbuilt components, including a graphic mapping component with a drag and drop interface. New versions of TOS provide a Business Modeler that creates a business diagram to monitor and perform impact analysis if any problem occurs. Commonly, it is used for data migration, synchronization or replication of databases. TOS is compatible with Java, SQL and Perl so that the user can edit existing components or build his own. Pentaho Data Integration (PDI), also known as Kettle is also an open-source platform that enables the user to clean data from different sources, move and transform data. Kettle provides a user-friendly graphical interface with drag and drop features to model the flow of the data, a command line to execute the transformation, a graphical tool to build and manage jobs and a web browser that allows remote monitoring of ETL processes. PDI is Java-based and has some shared libraries to simplify the development process [51, 40].

Both presented open source tools have similar features recording core functionality. Talend Open studio output can take any form, pie or bar graphic, table or dataset, while Pentaho system is only able to reproduce tables or datasets. A more detailed comparison was made in [39], where both TOS and PDI fulfill all the given criteria on the pre-evaluation, being the most mature and complete open source software. The main goal of this research was to evaluate the performance of these tools regarding execution time, CPU and memory load. Overall, TOS shown significant speed advantages and since the other metrics evaluated are similar in both tools, TOS higher efficiency settles the decision of TOS over PDI.

The tools mentioned above are some of the available products to any data migration software where the software language is not an obstacle. Both open source software presented revealed to use Java, while the OutSystems integration platform uses .NET/C#.

[1] Integrated development environment (IDE) used in computer programming.

Many recent articles compare tools and provide similar ranking lists of licensed and free, close and open source software, as in [1].

2.9 Data Migration in OutSystems

To properly migrate OutSystems' data it is important to comprehend the development components and guidelines provided by the platform. Some platform's internal components were already defined on Section 1.2.

2.9.1 Developing an application with OutSystems

The development environment - Service Studio - is divided into several tabs, the Interface, Logic, Data, and Processes tab. The Interface tab is used to create *Screens* using drag-and-drop. The Data tab is used to create ***Entities*** and define the database model. The data can be fetched with an *Aggregate*, that is an OutSystems visual query to manipulate data, or with standard SQL queries for more custom queries and stored procedures. At the Process tab, the developer is able to define timers and business processes. In the Logic tab, the developer can create and/or import server and client ***Actions***, it is also possible to consume or expose integrations (REST or SOAP).

An OutSystems' *Entity* is similar to a database table, and it is used to persist information into the database and implement the database model. It can also be a view. Some are *Static Entities* which are basically a set of named values (like an Enum Type - a group of constants). On the Logic side, *Actions* are used to manipulate *Entities* and provide business logic to the application. The *Client Actions* are tied to a particularly *Screen*, adding logic to a functionality, like clicking a button or a link and open another window or make an operation. Therefore, only *Screen* elements can access them. The *Client Actions* can call *Actions* that are similar to a regular method in object-oriented programming, called *Server Actions*. These last are used for encapsulating reusable logic, and they can also be public or private. If the action is set to public, it means that it can be referenced by other ***eSpaces/Espaces***. Most common scenarios in *Extension Modules* are writing custom .NET code, as *Actions* and use external database tables, as *Entities*.

While developing applications in OutSystems, sometimes it is interesting to have a variable holding a collection of variables with different data types grouped together and use it in the logic or UI. For example, to assign the values returned by an action and avoid creating one output for each value. In OutSystems, it is possible to create structured values by using a *Structure*, which is a compound data type used to encapsulate groups of related attributes. On top of that, sometimes it is helpful to execute logic to configure an application after deployment or even execute batch tasks like sending emails at a predetermined time. To support these functionalities, OutSystems uses *Timers* that allow executing asynchronous logic. *Timers* have associated actions and are handled by the OutSystems Scheduler Service that checks for *Timers* that are ready to run and executes their actions.

Every OutSystems Module has a ***User Provider Module*** associated. By definition, the *User Provider Module* specifies a source for the end-user records to be used by a specific module. By

default, all applications have their *User Provider module* set to *"Users"*. The *User Provider* is bonded to one ***Tenant*** - each *eSpace* has a *User_Provider_Name* attribute that corresponds to the *Tenant* name. A *Tenant* is an "instance" of an application that uses isolated data. Typically, OutSystems' applications run on a single *Tenant*. For scalability, OutSystems support **multi-tenancy** by allowing the same servers to be used in a hosted environment to host multiple users. Each user is given a separate space within those servers to store data, providing data isolation among clients. These *tenants* share screens and the business logic of the application, but data, sessions, processes, and end-users are isolated from each other. It is possible to turn on "Multi-tenant" as an *eSpace* property, which is especially useful to a Software as a Service (Saas) application. For multi-tenant scenarios, the platform creates a *Tenant_ID* column in every table and then creates a *VIEW* for each *Tenant*. These views are automatically used in queries instead of the original tables. Setting an *Entity* as multi-tenant constrains their data per *Tenant*. Therefore, *Aggregates* and SQL are performed with automatic data isolation: only data belonging to the correct *Tenant* is returned.

2.9.2 Data Migration Guidelines

To migrate data between different OutSystems installations there are many factors to consider and steps to follow. OutSystems provides guidelines [47] and recommendations to follow when copying data from production to non-production environments.

Some relevant instructions refer to the importance of:

- there exists a high probability of migrating data that is no longer needed, because it is dirty or the target already contains it. For this, the solution given is to create a first process step to delete the target data and its dependencies.

- the matching process is quite complex and extremely important since the *entities* can have different physical table names between environments. Besides that, *entities* or *attributes* can also be duplicated. The physical table name changes when the developer edits the *entity*, incrementing an index that appears after the name. However, only one *entity* and *attribute* is active in any environment and only these ones match between environments. The match is done using the *Entity SS_KEY* always combined with the related *Espace SS_KEY* (it is possible to have repeated *Entities SS_KEY* but not if combined with the *Espace SS_KEY*).

- the application *entities* to be migrated can have static *entities* as foreign keys. OutSystems Platform does not allow to change *static entities*, so it is also necessary to match these *static entities* between environments - enumerated *entity* records correspondent identifiers can be different in source and target.

- one *entity* can have several foreign keys, each one of the dependencies can also have other dependencies, in these cases the solution should deal with the direct dependencies and indirect cyclic dependencies.

- to match data between different environments could also be useful to create an auxiliary supporting table for mapping and matching *entity* identifiers between environments. This supporting table will then be used to replace the foreign keys on the target environment accordingly to the map. This is also called Foreign Keys Map and should be processed in a different way according to the foreign key type: *static entities* records identifiers, users identifiers, other foreign keys identifiers.

OutSystems also supply common general questions and respective answers associated with data migration [46]. These instructions, applied to the database engine Microsoft SQL Server and cover topics such as disable an application, get application users' info and *BPT* triggers with auxiliary queries. These queries are useful to understand how the OutSystems database is structured and what are system entities and relations.

2.9.3 Partners's Products

Despite not providing specific data migration tools, OutSystems has partners who developed products to deal with the specific use case of copying data from production to a non-production environment. All these products have subscription plans for organizations to be able to use them.

- DMM - Data Migration Manager

  Created by Infosistema Solutions [9] since November 2017, enables data migration across and into OutSystems environments or servers. It is the most mature software available that provides GDPR compliance features like data anonymization and data scramble. DMM is complete in terms of features, detecting inconsistencies due to non-selected dependent tables, filter data, reporting errors during the migration and even reuse migration plans. This web application is available to download on the OutSystems Forge.

- CDM - Cool Data Mover

  Published on May 2019 by CoolProf [3], the OutSystems application allows automating the export and import of data between environments on any platform installation (PaaS, On Premise and Hybrid). It is database independent, GDPR compliant, supports data synchronization and data backups. CDM has a two-step approach, where the import and export mechanisms are made separated and manually. It is also available at Forge to download into OutSystems service studio.

- Sync iT

  Recently powered by Do iT Lean [6], allows migrating application data from one OutSystems infrastructure to another, regardless if it is on-premise or hosted in the OutSystems cloud. This product is not available to download at the Forge but it is possible to ask for a demo or contact the partner, the features available are directly dependent on the subscription plan, the basic plan contains only the main features to perform an optimized migration. Sync iT consists of two core applications, one for the source and one for the target environment.

The two core applications are supported by a migration engine that houses the brains of the solution.

2.9.4 Comparison of OutSystems data movers

Features of a data migration software are not wide diverse, all products emerge from the same needs. Solutions that derive from the same problem have similar features. Meanwhile, the implementation, configuration and utilization of these features can be different. Table 2.3 presents some of the more relevant features that the different OutSystems data movers own. Those features are described as:

- Primary Keys integrity — The primary key (PK) imposes the *entity* integrity. It is typically a column or combination of columns of the table that contains values that identify each row on the table in a unique way. It is impossible to guarantee integrity, accuracy and consistency of data over its life cycle, in any data migration, without guaranteeing primary keys integrity.
- Reusable migration plans — Data migration is always an iterative process that continues beyond the go-live, so the fact that the framework is completely reusable means that the client can continue executing further changes in data or structures for as long as he wants. After a migration plan is created for the first time, it could be used in later stages by re-playing the migration configurations as many times as needed.
- Pre-Migration validations — The pre-migration activities check the migration tasks settings before launch, these validations are extremely important when complex applications are to be migrated since they avoid errors, unnecessary processor work and save the developer time. For example: check database connection, inspect target schema and check the source data against the requirements of the target system.
- Automatic dependency injection — It is also a form of pre-validation that provides to the user an automatic approach of selecting all entities on which the selected table depends, without the need of knowing any table dependencies. This is, support for detecting inconsistencies due to non-selected related tables (validation of the foreign keys).
- Mask data — Studies of data protection in software systems [35] enforce that data protection within large data processing applications is increasingly becoming a very important issue. To ensure privacy is crucial to comply with GDPR , masking the confidential data used in the migration process.
- Business Process Technology (*BPT*) migration — A business process technology, also called a process, is designed to support an *entity*'s lifecycle through all stages, as a flow of activities to be complete in each phase, for example, the user executing a task in the application, or wait for a determined condition to progress to the following or execute a task that did not need any end-user mediation [18]. One of the most critical topics in data migration relates to *BPT* [2].

- Online and Offline mode migration — Online migrations permit non-disruptively since that the applications are still up and running during the process of migrating data (no performance impact). Offline migrations process is, otherwise, disruptive to users and applications since there is a downtime associated. Meanwhile, this mode ensures greater data reliability because there is no new data coming into the applications, assuring data consistency along the process.

- Schedule Migration Plans — Scheduling allows to set the migration to run automatically at the specific day and hour the user sets, so that he can easily run migrations in intervals or at off-hours.

- Customize migration — The multi-step process of selecting which application, module and entities to migrate provide a more customizable migration. It acts as a filter, allowing the user to select only the necessary data.

- Differential migration — A differential migration only transfers the data that was changed since the last transference, it does not allow data replication. The software recognizes what data was already moved and what is the new data, preventing data consistency.

- *User* migration — This feature provides user synchronization by allowing to migrate OutSystems' *entities*, where the *entity* is the *User*. Typically, applicational data has references to the *User entity*. One step of the migration process is to cross all foreign keys that are understood as user records to align the corresponding *User* between environments.

- Execution logs — Monitoring the migration process by accessing a full audit detailed log of executions. This feature allows the user to find where errors occur.

Table 2.3: Data Movers Features

Features	**Products**		
	DMM	***CDM***	***Sync iT***
Primary Keys Integrity	✓	X	X
Reusable Migration Plans	✓	✓	✓
Pre-Migration Validations	✓	✓	✓
Automatic Dependency Injection	✓	✓	✓
Mask Data	✓	✓	✓
BPT Migration	✓	X	✓
Online and Offline Mode	✓	✓	✓
Schedule Migration Plans	✓	✓	✓
Customize migration	✓	✓	X
Differential Migration	✓	X	X
User Migration	✓	✓	✓
Execution logs	✓	✓	✓

De Ruijter [29] compared both Cool Data Mover and Data Migration Manager applications, both available at Forge, in metrics such as ease of installation, configurability, performance and reporting.

Regarding the installation process, all products have some obstacles to overcome. Before performing the migration process itself, there is some configuration to be done. While CDM has a larger amount of site properties to set, it has no external dependencies and thus can, in theory, be installed in all environments without interfering with the existing applications. DMM has some additional database configuration requirements (it is necessary to have database access), dependencies and if using an OutSystems cloud environment it will be necessary to contact OutSystems support to arrange access to several places and have settings applied. If the user plan to use it on an on-premise or AWS/Azure environment, then this administrative overhead of DMM might be a downside since DMM setup has a steep learning curve. Sync iT, as a business solution, is not so accessible since it is not available until you contact the partner directly.

The configurations/steps to perform before the migration itself are different in all applications. CDM performs the migration with an import/export manual mechanism (two-step approach), where it is necessary to create packages to perform these actions. First, one import package should be created to download the data and then after defining the operations that package can do, it is necessary to configure where to export the package to. Only after the package is created, the user can define which entities from which modules the package should contain. The reason this process is done this way is to improve reusability by providing the possibility of export one set of data and import it in various environments multiple times. However, in DMM the steps are shorter, starts by selecting the source and target database connection from which to export and import data. Then select from which applications and *espace*'s to move the *entities*. There is also the possibility to upload a configuration file from your computer files. In the end, it is possible to save the configuration, also providing a simpler, reusable plan. Sync iT, on the other hand, allows data migration through two core applications, one for the source and one for the target environment. This division is not done on the other competitors, which can be a positive or negative aspect depending on the user preferences. On the source web application the user can choose a target server in which the data is configured. The user can add applications and consult their entities but can not consult the *modules/espace*'s. Both DMM and CDM provide the user the option to chose which *modules* and *entities* to migrate, but, unfortunately, Sync iT does not provide the customization migration feature, allowing to migrate unnecessary data and losing time along the process. Afterwards, the user has to sign in the target web application and schedule a migration. Sync iT provides an innovated dashboard with information about the migration duration, planned migrations, last created migrations and last executed migrations. All migrations are saved automatically and then the user can consult them later and execute them again.

The speed of transfers depends on multiple factors such as network speed, CPU utilization and so on. One test with 1367 entries performed by De Ruijter revealed that the migration process itself was faster with DMM than CDM. The reason could be because Infosistema DMM runs dozens thousands of records per minute. That's because it get the most out of the database itself

not having the huge overhead of working at a higher layers of abstraction. The performance in Sync iT is tuned by built in serialization, data compression and customized batch sizing.

Data protection is very similar in all applications, the option is usually present on the *entities* list, where the user can define which *entities* attributes to mask and how. The products support different types of mask configurations. DMM supports scramble and anonymization, CDM supports anonymization and Sync iT supports scramble.

The reporting of the migration is available to get a clear view of what happened and the speed of the different actions. It is good to have a detailed audit trail where, in case of error, the user can see what the problem is based on and where it emerged. CDM provides a very differential view of this logs by switch the verbosity level. DMM logging does not have different configuration possibilities, it logs, by default, all exports, migrations and scramble actions on data. Sync iT provides a very detailed log with information about how many processed requests, which *entities* and the number of records.

In terms of user interface all applications have a user friendly and easy to use interface, being DMM one of the most straightforward regarding this topic. Meanwhile, Sync iT provides a better and more appealing design compared to the other applications.

All applications have different approaches that provide a similar service. Selecting the best application suited depends on different variables as the type of storage (on-premise, cloud or hybrid), and the aspects the user find most important, e.g. speed, data consistency, data security, between others. There is no one size fits all solution.

2.9.5 Limitations

None of these products are open-source software, i.e., the code of this software cannot be edited or extended. The software complexity of closed-source often control the price that consumers will pay for access to it's benefits. The hard costs include integration, licensing (with specific max number of user licenses), and ongoing supports. DMM is the only one with no additional costs for month subscription, for the other products, a bigger investment is needed to have access to more features and more capacity for data movements, costs are according to the actual consumption of resources (pay per use). However, in DMM the cost increases proportional to the organization size. Further aggravating, it is not possible to change the code to meet the costumer needs, since they are closed-source software and there is no permissions to change the code or add new features. The business is in constant changes and it is important to meet it is needs by providing flexible tools that can be customized. If the company develop an internal feature this cannot be added to the software unless the software providers decided to implement it on the product, to all costumers, what is not ideal since the company would be given free features to any competitor with the same product. Other limitation happens when the costumer finds a bug or something to fix and he needs to wait for the company providers to fix it, which can take hours, months or years and with no assurance that they will even fix it at all. If the situation to fix is very important to the business the costumer/company can start to loose money. Finally, the software can also be updated anytime, sometimes this updates can affect the costumer platform due to it is frequent changes. The inability

of controlling this updates can lead to unexpected and unintended results. The costumer is always dependent of the product supplier, while in an open-source software it is possible to constantly expand, customize, and update the platform.

2.10 Summary

In many organizations, data migrations are regular tasks inside the IT department. The data migration topic has been extensively studied. The major issue is the amount of steps involved that work together to build an abstract implementation of the problem. Data migration is a complex project that often causes major issues with a very high level of risks such as downtime, schedule overruns, budget and poor performance. Usually, the solution to attain an automated migration is to perform it through software tools. A reliable and consistent methodology is needed to plan, analyze, migrate and validate the migration. Using the right ETL tool can simplify the data migration process and minimize the risk of possible data defects on the target system. The ETL software/tool should be combined with a validation system to assure data quality. In case of dealing with heterogeneous structures where source and target have different structures, then a tool for matching mapping data should also be introduced. The chosen tools have to support the specific migration requirements, including storage platforms, operating systems and performance. Currently, data migration in OutSystems is a problem since the platform does not allow developers to manipulate the database and there is no permissions to perform any data synchronization. Besides some limitations, DMM[2] proves to be a more complete solution than the competition. However, all the available tools seem to be very inflexible.

[2]OutSystems' partner product for data synchronization. More on Section 2.9.3.

Chapter 3

Using OutSystems for Data Migration

OutSystems platform uses components that allow to increment its functionality and integrate with external systems. In OutSystems, advanced and automating integration is accomplished through *extensions*. Integration Studio is the component used to extend existing functionalities and data model of OutSystems. With *extensions*, it may be possible to integrate one or more external relational database and manipulate data from both databases at the same time, allowing the possibility of migrating data between database connections. Since the OutSystems platform concedes many benefits, such as rapidly and efficiently creating extensions, their data migration limitation options may be overcome by using the platform benefits to create a framework able to solve the problem.

3.1 Open issues

The underlying question behind this work alerts for the possibility of creating a software product able to match developers needs and overcome the OutSystems drawbacks: **"Is it possible to, automatically, provide data migration between different environments using only the OutSystems platform?"**. The main objective is reducing human intervention to the minimum, saving developers valuable time. To solve some of the platform limitations, a cause-and-effect relationship was established, "Is it possible to create a tool using only the OutSystems components such as Service Studio and Integration Studio?", "Is implementing an extension to migrate data enough?". The approach to solving the data migration problem of the OutSystems platform was developing a tool that is able to do this data migration (cause) and evaluate the effect of one deployed application with the data migrated by the tool (effect).

The criteria for validity and reliability in this type of research is usually the evaluation of the migrated data integrity and consistency. However, additionally, efforts were made to compare the behaviour of one environment's application with migrated data and the behaviour of the original environment's application. This method allows to better analyze the system data such as User roles, data displayed and processes. Methods that only analyze migrated data, such as using

samples and creating SQL queries, can fail to detect errors since it is almost impossible to verify a large number of records.

Even knowing that the OutSystems platform has some drawbacks, it does have many benefits like performance and scalability. The chosen approach allows taking advantage of the platform using the built-in integration with a new external service that implements the data migration process. Developers manage integration configurations without the need to write custom code, significantly reducing time and effort and eliminating human errors. From a developer's perspective, there is almost no difference between invoking an OutSystems method, a REST API, or an *extension* method.

This approach can be generalized for any enterprise that uses OutSystems with SQL Server database to develop and deploy applications to the customers. Regarding other data migration tools available, there is no public information about the software used to implement them and neither any code instructions.

3.2 Metadata collection

The data needed was referent to the OutSystems platform specifications, thus some questions were raised at the initial stage: "Does OutSystems provides informational data (metadata) about applicational data?, "Where is the metadata stored and what types of relationships are there?". For this research, the data collected was basically existing data, stored at the (System) database and imported to the project tool. The system had a variety of informational data, but only some entities were relevant to the data migration (listed in Section 4.2.2).

The data collected is originally produced by the Factory Database[1]. This information is managed and maintained in the *system's entities* while creating new *eSpaces* and publishing new versions with *entities*. The data used for the migration process is *applicational data*, but managing this data in an **abstract** and **automatic** way requires some system/informational data to be collected and handled from the platform, also called *metadata*. This includes information details about the structure/schema of the applicational data tables, table relationships, and table storage location.

3.3 Metadata analysis

The qualitative approach used in this research was based on observations of the OutSystems platform behaviours, i.e., what processes OutSystems runs on the background, how data is stored and used by the platform. The OutSystems Platform metamodel was analyzed from the data migration point of view.

Entities (tables) can be grouped into three main types:

- System Entities

[1] Application manufacturer.

- Application Entities
- Hybrid Entities

3.3.1 System Entities

Creating an entity forces the developer to give it a logical name and the OutSystems Platform creates a physical name for it internally, this is then used to represent the *Entity* at the SQL Server level. The mapping between the physical table name and the logical name is in the *OSSYS_ENTITY* table, along with other relevant information regarding the entity details. *Systems entities* are recognized by the prefix *"OSSYS_"* in their names and are pure system tables used in the OutSystems Platform's metamodel, such as: *OSSYS_APPLICATION* , *OSSYS_APP_DEFINITION_MODULE*, *OSSYS_ESPACE*, among others. These entities can be sub-grouped by different topics accordingly to which component they support. For this research, the ones analyzed were the ones that support the OutSystems Platform Factory. This factory includes *Applications* and *Modules*, *Espaces* and *Extensions*, *Entities*, *Attributes* and *Records*.

Some system tables have a column named ***SS_KEY***, which corresponds to an identifier used by the platform to identify something regardless of which environment or installation it is deployed to. Usually, these *SS_KEYs* are unique but can get a new one if they are copied. The *Espace* always have a unique *SS_KEY* generated and stored in the metamodel and equal in all environments (it will be the same no matter where that *Espace* is published). However, the *SS_KEY* does not always have a unique value. After analyzing all entities that support the OutSystems Platform Factory, it was identified that the other *SS_KEYs* are not unique, for example, in *Entity* and *Entity_Attr*. These could be duplicated when the *Espace* is cloned in order to support operations like merging. When this happens, all the information inside the *Espace* is replicated to the Clone *Espace* (Cloned *Espace* generates a new *SS_KEY* for the *Espace* but all other objects keep the *SS_KEY*). The new (generated) entities have the same *SS_KEY* and *Primary_SS_Key* from the original *Espace* and have the same structure, the difference between the original entities and cloned ones can be detected by their *Physical Table Names* and the related *Espace SS_KEY*. To be unique, it has to be combined with the *Entity* related *Espace SS_KEY*, which is the way to match and map the same *Entity* Record between two different environments.

Another common column in system entities is a flag called *IsActive*, this attribute is used to include or exclude inactive soft-deleted entities. For this research, only active records are relevant since they are those available to the end-users. Being able to migrate tables that have relationships with other tables (foreign keys) alerted to the attribute - *Type* - of the *Attribute* entity, which contains:

- Run-time types — *rtText*; *rtLongInteger*; *rtInteger*; *rtDecimal*; *rtDateTime*; *rtDate*; *rtBoolean* ...
- Reference Id types with the format :*'bt'*+<*Espace_SS_Key*>+*'*'*+<*Entity_SS_Key*>

The reference ID types expression contain the composed unique key that can map the same *Entity* between different environments.

3.3.2 Application Entities

These tables are usually recognized by the prefix *"OSUSR_"* in their name and relate directly to the application development. Applicational entities are associated with the business and do not exist until the application is created. The prefix is assigned for entities created inside an application, for example, *"OSUSR_0o7_Employee2"*. The *"0o7"* is a three character code that groups tables by *eSpace*. It is also unique for *eSpace*. Other applications can also have, for example, an entity with the name "Employee", but its physical tables names have to be different, like: *"OSUSR_aew_Employee"*, *"OSUSR_bla_Employee1"*, or similar others. Duplicating an application would result in creating entity replicas with a different physical table name since the eSpace would also be different.

Applicational entities usually have foreign keys to the systems model, like *Users*, *Tenants* and *Multi-Tenant*, which are some of the most used system entities that might cause dependencies and have to be mapped and aligned between environments in order to migrate final applicational entities.

The data migration process created must be allowed to read and write on these entities.

3.3.3 Hybrid Entities

These tables are similar to pure system tables (managed by the OutSystems Platform). On the other hand, they also interact with applications (that create records on these tables). Due to this composition of different elements, these are the tables that require more attention and work from a data migration perspective. To read and write on these entities is necessary to understand these types under certain conditions. To this research, a Hybrid entity called *User* was analyzed since applicational data typically has references to this entity (for example, by the *CreatedBy* type of audit columns). For the migration process of an application entity, the software has to traverse all foreign keys to understand all records that are holding said user records and treat them accordingly.

3.4 Technological choices

The database used, in this dissertation, to integrate with the OutSystems platform is a **SQL Server** database with Microsoft SQL Server Management Studio as a script auxiliary. Additional software requirements require the set up of **Microsoft .NET Framework** on the developer machine.

SQL Server Management Studio was used as an auxiliary tool to easily connect to different environments and manage their data. This software allows connecting to any database inside the server with a given SQL Server Authentication credentials. Both system table contents and schemas were analyzed with this tool.

Inside OutSystems Service Studio, it is only possible to visualize the entities related to the current eSpace, which makes it difficult to analyze the data. Using SQL Server Management Studio allows analyzing all entities of a given environment with a single SELECT statement, even if these entities are in different applications or eSpaces. In summary, it allows to globally visualize OutSystems data using the physical table name of the entities.

3.5 Methodological choices

From the several different strategies for undertaking a data migration in the design phase, **Trickle** Migration is the one that is linked to an agile/incremental approach. For the aim of this study, a strategy with zero downtime of source and live target systems was preferable since it can reduce risks. The migration process should be done in phases and both systems run in parallel without operational interruptions. On the contrary to the *Big Bang* strategy, that a complete rollback is required if the migration fails, the *Trickle* Migration strategy have easy rollbacks, i.e., if a single-phase fails, it is possible to detect the exact location of the error and proceed with the rollback.

As regards software architecture, an **ETL** (Extract, Transform, Load) based architecture was chosen to support the study, mainly because it implements a transformation, using a transformation database or data mapping table to store intermediate results of the transformation step. This is essential to build a data migration tool that provides foreign keys integrity, using the data mapping table to match and map the source with the target records. Other architectures, like the *Butterfly* approach and *PDMv2*, were analyzed but disapproved since they incorporate complex gateways or are very business oriented with low technical details.

For this solution development, the most valuable methodology to implement should be based on delivering value faster, with greater quality and responsiveness to changes due to the substantial amount of software requirements suggested. The software development used in this solution was **Agile** since it is iterative and can improve flexibility and control while delivering shorter software portions instead of just one unreliable final product.

The validation process is not over after testing the code during the development phase. It is also important to test the data migration software with real applications to ensure the accuracy of the implementation and completeness of the application. The validation method in order to confirm if the hypothesis is, in reality, fulfilled is to conduct a Live Test. The live test offers an analysis of the migrated application behaviour. Additionally, the use of random samples and SQL queries to check the data integrity complement the technique to gather valid conclusions.

Finally, the use of external software was rejected while analyzing the complex OutSystems requirements since some transformations were very specific. A program may require one or more other programs to run (the "dependencies"), but dependencies can also be an overhead. For this specific study, the implemented software tool does not rely on almost any other software since this adds complexity and uncontrollable dependencies make the code difficult to maintain and to add new functionalities. This is also known as "Big Ball of Mud" or "Dependencies hell", the costs of integrating existing systems and try to shape them to meet our requirements does not, in

this case, justifies its use. In summary, the methodology chosen in the face of integration with external services was to build software from scratch with fewer dependencies possible to have better control over the features.

3.6 Summary

In this chapter, the dissertation hypothesis is formulated. The choice of methodologies was made in order to save developers time by automatically run the data migration process between any environment, without any manual intervention. For this, a new OutSystems *extension* is planned for development and integration, using the Integration Studio, with an easy to use User Interface built-in Service Studio. This limits the amount of software used to build the product, using only another auxiliary tool - SQL Server Management Studio - to complement the development and testing phases.

Chapter 4

BirdlyData

In chapter 1 this dissertation goal was identified: build a framework that supports automated data migration between different environments using the OutSystems platform. A list of functionalities to achieve the overall goal could be defined as:

- Allow the developer to create and test a source and target database connection configuration.
- Pre-Migration validations to ensure a successful migration: automatically detect recursive entity dependencies; order entities list by dependencies.
- Automatically migrate data between environments respecting FK integrity.
- Provide a real-time migration log so that the developer could be able to visualize the background process state.
- Generate notifications via email with information regarding the finished migration.
- Ability to reuse a saved migration plan data for proceeding with another migration with the same information.
- Allow the developer to schedule a migration to any time desired.
- Provide differential migration of system table *Users* (end-users of the applications).
- Ability to migrate differential data, i.e., only the last records inserted after the last migration.
- Provide GDPR compliance by allowing the developer to mask sensitive data.
- Integrate the migration with CI/CD.

BirdlyData's goal is to manage developer's applications data, so the end-users of the framework should have a certain level of specific IT skills, like developers and testers, to be able to manipulate the program. However, knowledge about the construction of database queries and

schemas is not mandatory but would be essential to understand the background mechanism and processes taking place.

This chapter provides more information about the BirdlyData framework implementation in order to allow developers to test applications with real data from any environment, which consequently promotes continuous integration and delivery in the development process. It is organized as follows:

- First, the architecture for the implementation of the framework is presented.
- Next, some pre-implementation (startup) configurations were properly defined, which are important initial steps required to proceed with the solution. Such as the definition of the data model as well as the justification of which OutSystems' Entities to manipulate.
- After that, the user interface workflow guide is presented, with information regarding the implementation of each Screen.
- Subsequently, a more detailed information guide about the functionalities and their implementation process is reported as well as the decision-making and problem-solving process among possible alternatives options.
- Finally, some notes regarding software optimizations are documented.

4.1 Architecture

The solution integrates two OutSystems applications that work together as a whole, the BirdlyData, a Reactive App, and BirdlyDataNotifications, a Traditional App. As explained in Section 1.2 the practices were improved in Reactive, but there are some Traditional features that were not migrated, like the email sender action. For this reason, bound with the notification objectives, the BirdlyDataNotifications was created.

4.1.1 BirdlyData

The BirdlyData application framework was divided into three different types of modules. The division was based on the different functionalities of each Module. The modules are:

- Core module: BirdlyData — Reactive Web App

  The Core module - Reactive Web App - involves the User Interfaces that are used by the developers. This Module is used to collect all the information that is considered relevant to the migration process and is where all the core processes take place like essential timers activation, store client variables, CRUD operations and overall logic. All except the migration process itself, which is not feasible using only the Service Studio platform[1].

[1] OutSystems low-code and visual development environment.

- Service: BirdlyDataAPI

  The Service module provides an Application Programming Interface (API) using Representational State Transfer (REST)[2]. This API provides the content required by the consumer (call) and the content required by the producer (the answer). The RESTful API was created to deal with the specific case of creating *User* records on the target environment, communicating between source and target environments, where the source is the consumer and the target is the stateless server or producer.

- Integration service: BirdlyData (extension) — Extension integrated with .Net and C#

  The Integration service is the extension module, which represents the connection with the Integration Studio[3], i.e. the connection with Microsoft Visual Studio IDE running the .NET Framework and software language C#. This Extension module is essential to the whole framework since it is where the migration process takes place. Inside the extension, some database connections were made, allowing the management of data, otherwise not possible.

The framework architecture diagram is shown in Figure 4.1.

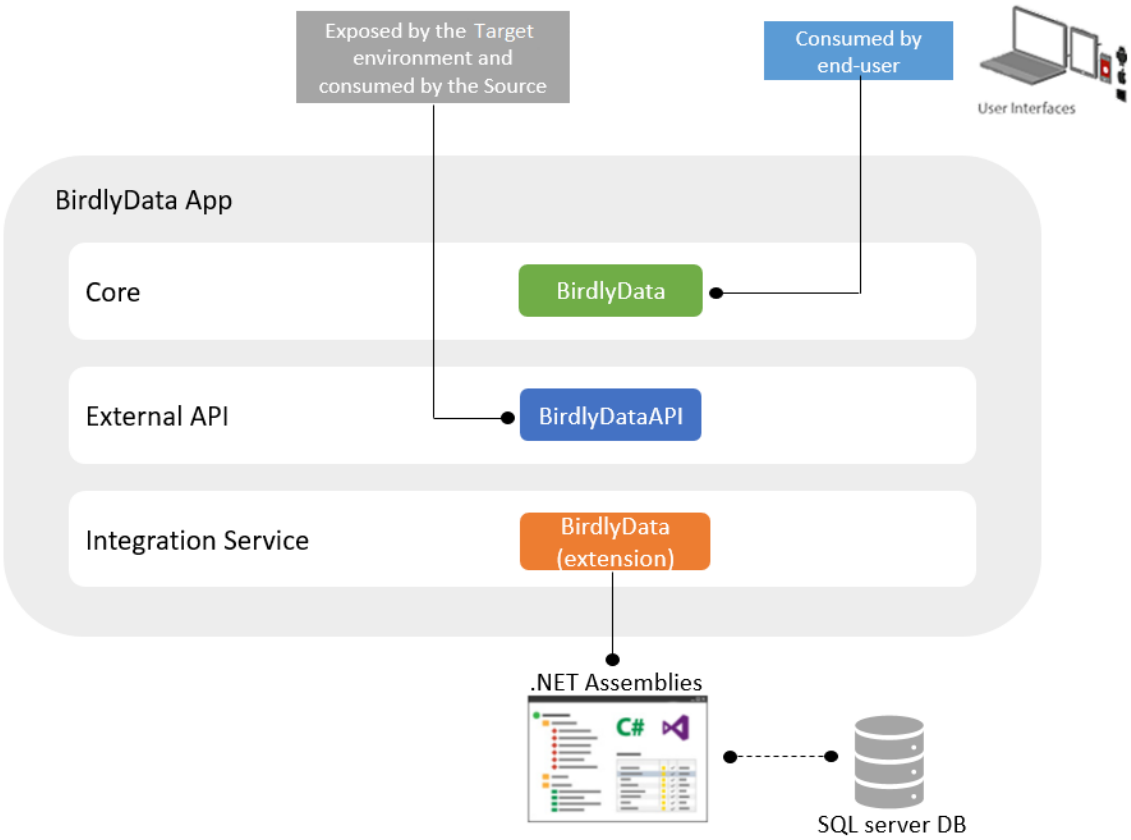


Figure 4.1: Birdly Data Architecture

[2]Software architectural style that uses the Hypertext Transfer Protocol (HTTP).
[3]Desktop tool that allows you to create and manage extensions.

4.1.2 BirdlyDataNotifications

The BirdlyDataNotifications application has only one Traditional Web module. This application consists only of one Email Screen and one Public Server Action responsible for sending notifications as emails.

4.2 Startup Configurations

For proper functioning, BirdlyData should be installed in all environments. One concrete objective of the framework is to be versatile, allowing source and target environments to be any (Production and Non-Production) existing environment. For instance, the developer should be able to choose the source environment, like Production, Development, Testing or Acceptance[4]. In addition, with the last set of environments, it is possible to make 4! = 24 combinations. The agility of choosing any possible combination makes it possible to migrate data for a variety of reasons and not only the specific case of migrating from Production to Testing. Moreover, the source environment should always be locked to the current environment,i.e. if the developers desire to change the source environment he needs to use the respective one. In the case where he intends to make a migration from, for example, Testing to Development, then he needs to install the application in the Testing environment and run the BirdlyData application there.

4.2.1 Manage Dependencies - Integrations

Some processes, widgets and entities were required for the implementation of the application tool and added to the dependencies as soon as needed. By default, OutSystems imports the basic widgets to build screens. However, some actions to manipulate lists, treat transactions, encrypt values and dependencies to other modules or entities were required.

The dependencies in BirdlyData Core include:

- (System) — System Objects: entities, commit and abort transaction and list management actions.
- Crypto_Lib — Crypto (Framework Library Module), able to encrypt and decrypt a text using the AES-256 algorithm.
- OutSystemsUI — User Interface widgets, all necessary data to build UI Flows.
- Users — Backoffice to manage users, groups and roles.
- BirdlyData (extension) — Responsible for the migration itself.
- BirdlyDataNotifications – Created Module that provides notifications via email.

[4]Production and Non-Production environments. See section 2.7.

4.2.2 OutSystems' System Entities

Essentially, the data migration process manipulates database tables, called entities by the OutSystems platform. There are a couple of entities to be managed alongside the project. Some of them are system tables that are embedded in the platform and contain information about the developer's applications to be migrated. The system entities added as dependencies were:

- Application — Applications that are used in the current environments. Old applications are kept inactive.
- Espace — Espaces defined in Service Studio. Older and deleted eSpaces are kept inactive.
- Module — An eSpace or an extension.
- App_Definition_Module — Modules for a specific application definition. Internal use only.
- Entity — Entities defined in Service Studio. Older Entities defined in Service Studio are kept as inactive.
- Espace_Entity — Entities referred by each eSpace as defined in Service Studio add/remove references.
- Entity_Attr — Attributes for each entity defined in Service Studio. Old attributes are kept inactive.
- User — End-user of the applications. Shared between eSpaces with the same user provider (defined in Service Studio).
- Tenant — Disjoint set of users, processes, and application data. Tenants are shared across eSpaces with the same user provider (defined in Service Studio). Used to segment data, users, and processes in a multi-tenant scenario.

To create a tool to move data between different OutSystems environments, first it is necessary to fully understand how the development factory[5] (*Applications*, *Modules*, *Espaces*, and *Extensions*) is managed by the platform. It is equally important to understand how the platform manages the factory applicational database info (*Entities*, *Attributes* and *Types*). The Figure 4.2 represents the system entities dependencies (FKs) considered more relevant to the project.

The explanation of the system data model, as well as the extended diagram, is annexed in Appendix A.1.

4.2.3 Created Entities

The Figure 4.3 represents the created entities attributes and connections between themselves and system entities.

At the first stage, with the objective to store the necessary information about a Migration, three main entities were created: *Connection*, *MigrationMaster* and *MigrationDetail*.

[5] See OutSystems' platform details in Section1.2.

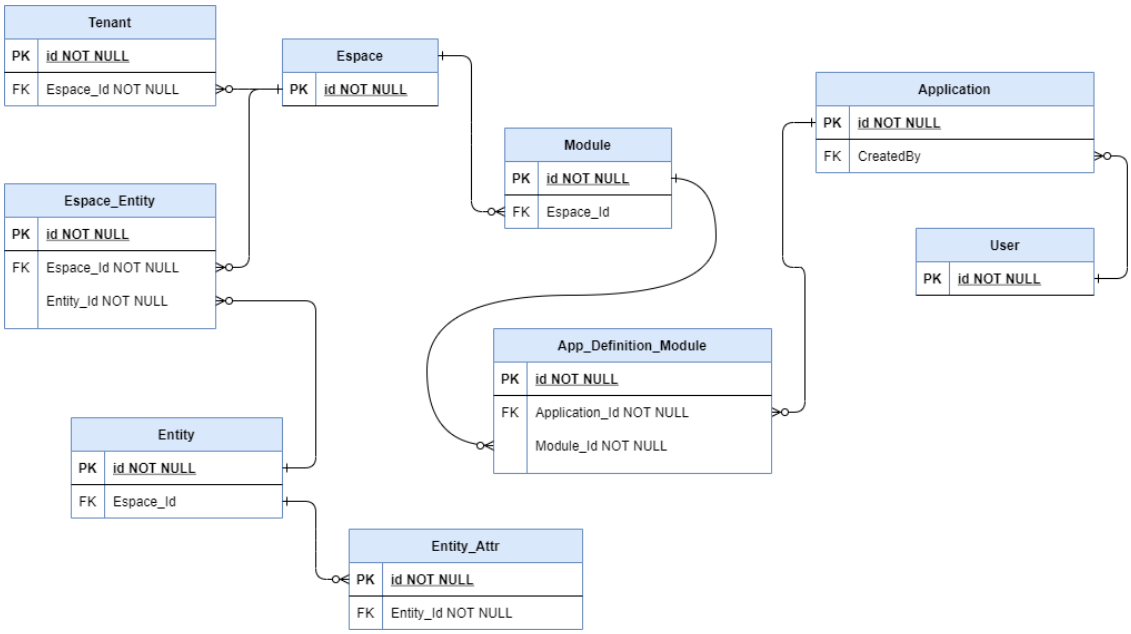


Figure 4.2: System entities dependencies compress diagram

- Connection — Entity used to store several database connections, which are related to an environment. The *Catalog* attribute stores the database name used to connect to that environment. IP is the attribute that identifies the database server. Furthermore, some authentication attributes were also required.

- MigrationMaster — Main entity responsible for storing the information provided by the developer, such as Source and Target *Connections*, chosen *Application* and *Modules*, and selected *Entities* (including their dependencies), i.e. all ordered entities necessary to proceed with a successful migration. Some log attributes are also stored in this data table to provide a detailed log of the migration process, such as *IsFinished*, *IsExecuting* and *TotalRows*. Other attributes were added with the evolution of the software and the construction of new features. Every attribute creation is justified inside the correspondent feature sub-section in Section 4.4.

- MigrationDetail — This entity stores all data relative to one specific entity. It was built mainly to log purposes since it is able to give a perspective about what is happening at an entity level and not only at a global migration level.

Several other entities were necessary to build a complete and extended solution. These entities were created to provide more complex functionalities, such as mask sensitive data, differential migration, and migrate dependent entities properly, respecting the integrity of the data FKs.

- MaskAttribute — This data table stores which attribute name (column) of a migration entity is configured as sensitive and set to mask.

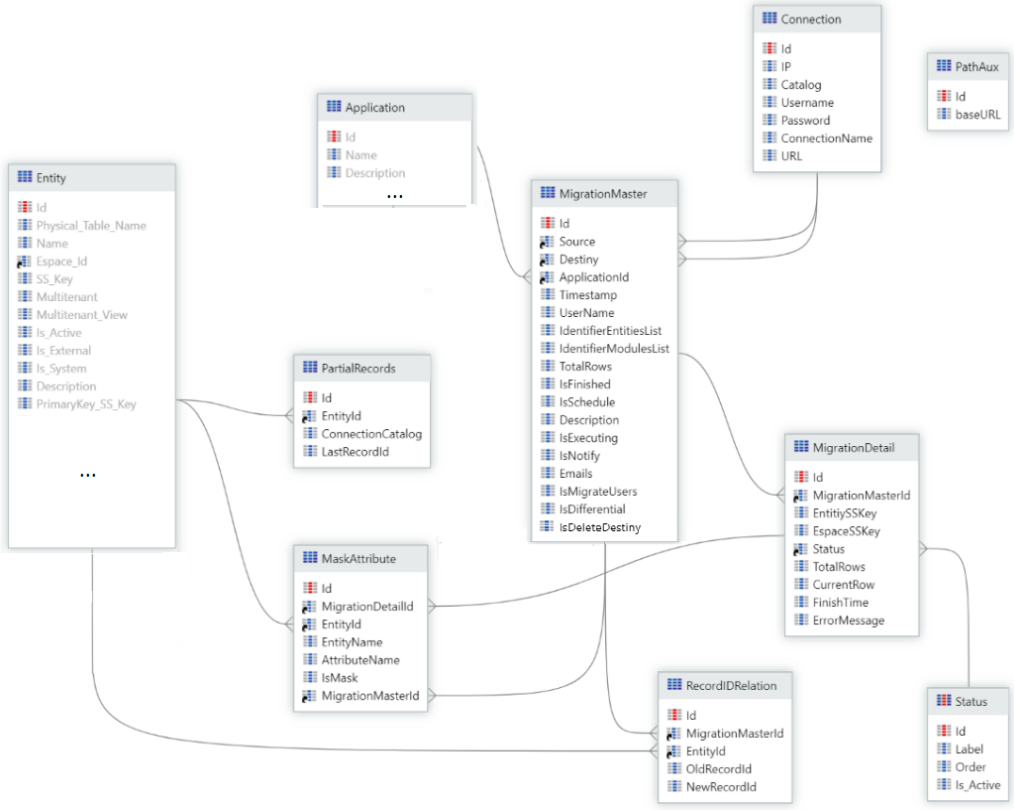


Figure 4.3: Created entities diagram

- PartialRecords — Entity responsible for storing the last record migrated of an entity at a given target environment.
- RecordIdRelation — Entity created to store the entity's old (record identifier before migration - at source) and new records (record identifier after migration - at target) for a given migration.

Some auxiliary entities were also created:

- Status — An OutSystems' static entity used to save the progress status of an entity while migrating. It is a group of constants(Enum): Done, In progress or Error. It is an attribute used by *MigrationDetail* entity for log purposes.
- PathAux — Auxiliary entity used to save the target environment URL. Created especially for the API management.

4.3 User Interface

To have access to the data migration tool, the developer needs to be properly authenticated. The initial task the developer needs to carry on is to login. Once logged in, he will be redirected to

the main dashboard. This has information regarding the latest data migrations executions and scheduled data migrations plans. From this screen, it is possible to navigate to all other screens. The workflow of a migration includes the several screens:

1. Settings Screen and New Connection Screen — Add at least two database access configurations, one for the target and another marked as source. Screenshot example and screen's logic on B.1.

2. Migration Plans Dashboard Screen — View finished and scheduled migrations. Reuse a migration plan or create a new migration plan. Screenshot example and screen's logic on B.2.

3. New Migration Screen — Add plan description, select the target environment, and, facultatively, activate secondary features such as: schedule a migration (by choosing a date from the calendar), send email notifications with the summary report (to one or more email addresses), migrate *Users*, migrate partial records (differential migration) and delete target entities. Screenshot example and screen's logic on B.3.

4. Applications List Screen — Select the application to migrate. It is also possible to search an application with the search bar. Screenshot example and screen's logic on B.4.

5. Modules List Screen — Select one or more modules/eSpaces that contain the pretended entities to be migrated. Screenshot example and screen's logic on B.5.

6. Entities List Screen — Select pretended entities, and optionally, add a mask to entity attributes. Screenshot example and screen's logic on B.6.

7. Migration Plan Overview Screen — Confirm the migration plan settings. If it is a schedule migration, then it redirects to the main dashboard screen. Otherwise, the screen is redirected to the Migration Log Screen. Screenshot example and screen's logic on B.7.

8. Migration Log Screen — Screen presents the migration status updates in real-time. Screenshot example and screen's logic on B.8.

The Flowchart represented in Figure 4.4 contains the diagrammatic representation of the user interface workflow within a simple migration scenario.

4.4 Functionalities

Implementation and decision-making process used to build a complete solution. These features support the belief that it is possible to create a data migration tool for migrating data between OutSystems environments, using only OutSystems components, SQL advanced queries and C# as an OutSystems extension.

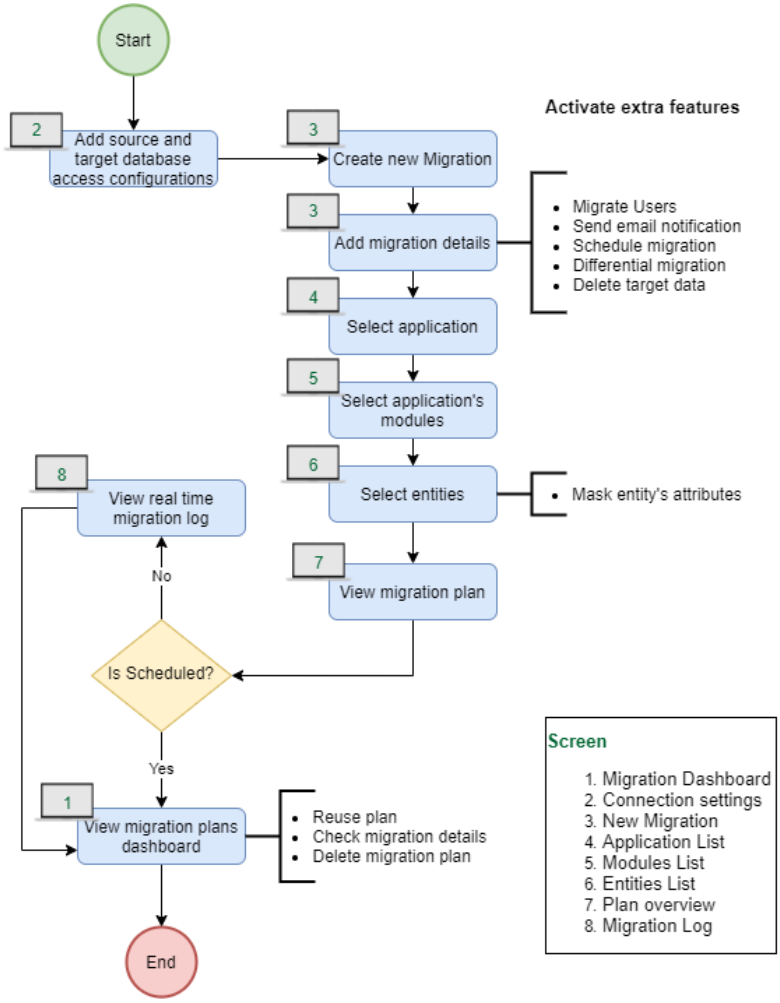


Figure 4.4: Migration Workflow Flowchart

4.4.1 Settings - Data Access Configuration

The purpose of the database configurations, called *Connections*, is to bind one environment to it's necessary specifications. In the extension, a **SqlConnection** represents a connection to a SQL Server database using *System.Data.SqlClient*[6]. To access database data is necessary to first set the **ConnectionString** property before using a SQL Server database connection.

The information required to build a *ConnectionString* is:

- DataSource — name or network address (IP) of SQL Server to connect to.
- InitialCatalog — name of the database associated with the connection.
- UserID — existing user identifier with an SQL Server account.
- Password — valid user password of the SQL Server account.

[6]Namespace .NET Data Provider for SQL Server

When creating a new database configuration, BirdlyData verifies this information and saves it as a *Connection*. Some auxiliary fields are also added, such as the name to display and the environment URL. The verification process uses an extension action - *TestConnection* - to validate the authenticity of the *ConnectionString*, described in C.1. For flexibility, the developer is able to create a set of connections. After one connection is created as the Source Connection, then it will be fixed for that environment. However, after the developer creates other valid connections, he can use any of these as Target Connections as long as the application tool is installed in those environments.

For security purposes, the password introduced is not saved into the database in the same form the developer introduces it. To prevent unexpected unauthorized access to the database credentials, an encryption algorithm was used since it is one of the most common methods to protect sensitive data. With this initiative, a Framework Library Module called **Crypto_Lib** was added as an application dependency. This Module allows to encrypt and decrypt a text using the AES-256 algorithm.

The encryption algorithm details and the use of *Crypto_Lib* in BirdlyData are explained in Appendix C.2.

4.4.2 Entity Dependencies

Migrating data with dependencies between entities (usually mapped by Foreign Keys) needs to consider the entire cascading depth of those dependencies. For instance, entity A has a **direct dependency** on entity B if A has an FK to B. If entity B has a direct dependency to entity C, then A has **indirect dependency** to entity C. An so on, a so forth.

An OutSystems entity depends on other entities if the format of the column/attribute **Type** inside *Entity_Attr* is: **bt<Espace_SS_Key>*<Entity_SS_Key>**, where both *Espace_SS_Key* and *Entity_SS_Key* identify the entity at which the first entity relies on. One concrete example of direct dependencies in an OutSystems entity - Employee - is presented on Fig 4.5. The *Id* attribute is the identifier of the Employee entity. The attributes *JobId*, *ManagerId*, *DepartmentId* and *UserId* correspond to FKs, entities on which Employee depends, accordingly to the *Type* attribute format.

Through a series of SQL queries, these FKs were extracted and the direct dependent entities were retrieved. **Recursively**, all indirect dependencies were also retrieved and properly **ordered** so that the migration process could proceed properly. The order respects the entity dependencies, adding in first place entities with no dependencies and then adding the ones that only depend on the entities already inserted. This was done as a pre-validation feature prior to the migration process to provide the developer with the full list of entities to migrate. All details on retrieving and ordering the dependencies can be found in Appendices D.1 through D.3.

4.4.3 Migration

The majority implementation of the migration mechanism is done inside the OutSystems extension. At Integration studio, a new action was created called **Migrate**. The input and output

ENTITY_ID	NAME	TYPE
10149	Id	rtLongInteger
10149	FirstName	rtText
10149	LastName	rtText
10149	Email	rtEmail
10149	Phone	rtPhoneNumber
10149	HireDate	rtDate
10149	JobId	bt1f98f437-11c2-4f21-b1d7-af3c9469c125*39fa3a99-211d-47c7-8db9-a952c7190a9c
10149	ManagerId	bt1f98f437-11c2-4f21-b1d7-af3c9469c125*b7aaa0ce-c593-43f6-82aa-add2cfcccd64
10149	DepartmentId	bt1f98f437-11c2-4f21-b1d7-af3c9469c125*4d63f0dd-3504-4095-81a3-6bcef7fce6e3
10149	UserId	bt478870b9-2d60-4f73-9eb3-7cd8b994a737*a416822d-845f-46cb-a270-70f5e383544c

Figure 4.5: Attribute direct dependencies example of Employee entity

parameters of the action are also defined in the Integration Studio as shown in Figure 4.6. The action receives a *MigrationConfig* Record with all necessary information provided by the developer and gathered by using the User Interface. The *isSuccess* output parameter allows to verify if the migration was finished without errors, otherwise, the *errorMessage* parameter will contain a text error message.

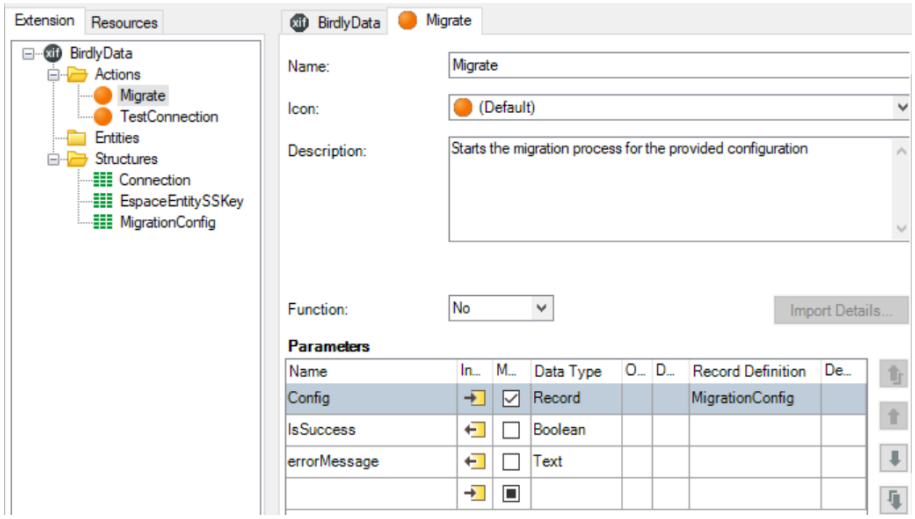


Figure 4.6: Integration Studio Migrate Action

The Integration studio structures, such as *MigrationConfig*, are identified in Appendix E.1. Three main steps were essential to the migration process:

- Discover the physical table name of the entity in both source and target databases.

  If the entity was modified in the target environment then, the same entity would have another physical table name[7] from the source. The implemented solution uses a composite key[8],

[7]The physical table name is essential to identify the OutSystems' entity in SQL queries.
[8]Unique identifier of the entity through environments. See Section 2.9.

globally unique, of both *Entity* and *Espace SS_KEYs* to retrieve this data from the target SQL connection. An example of this situation and the implemented query are present in Appendix E.2.

- Implement a dynamic INSERT query able to parse any table with different columns and data types.

  The *INSERT INTO* query needs the entity's physical table name, the ordered columns and values to insert. This data is retrieved through a couple of other auxiliary SQL queries. For example, the column headers are fetched from the table's schema info and the values are fetched from another SELECT query to the source database connection. The detailed string builder process of the INSERT query is in Appendix E.3.

- Map entities' identifiers between the two environments to provide FK integrity.

  Entity records' identifiers, in OutSystems, are auto-incremented[9], which means that the migrated target records will not necessarily have the same identifier, even if the target was previously cleared. The implemented solution uses a data mapping table (Foreign Keys Map, called *RecordIdRelation*) to store the identifier of the source and the corresponding target match identifier. While migrating the records, both environment identifiers are stored in the data mapping table. In the presence of an FK, the data table is consulted and the source identifier replaced by the target identifier. The detailed implementation of these steps is present in Appendix E.4.

4.4.4 Real time Migration Log

While running the migration process inside the extension, the developer does not have any information about what is happening in the background. This is mainly because OutSystems does not have access to what is concretely running, just its final output. Besides, external data modifications are not detected. However, any change applied to the system should be documented in an audit log[10]. From a developer's perspective, audit logs give some insight into how a complex system, as data migration, arrived at its current state. Frequently, sharing data between different infrastructures is made by a common database, so the proposed solution uses the entities: *MigrationMaster* and *MigrationDetail* to provide real-time updates of the migration status, some new log attributes were added, at the entity level, to *MigrationDetail*: *TotalRows*, *CurrentRow*, *Status*, *FinishTime* and *ErrorMessage*, while at the migration level, *MigrationMaster*, other two were created: *IsFinished* and *TotalRows*.

The enumerated values, assigned while creating the *Status* Static Entity in OutSystems Service Studio, are:

- 1 — Done

[9] Incremented by 1, each time a new record is inserted.

[10] Audit logs are typically used for diagnostic performance and error correction.

- 2 — In Progress
- 3 — Error

While migrating one entity, the extension uses the correspondent *MigrationDetail*, to update the entity status to one of these enumerated values. It also counts the total rows of each entity to provide information about the migration progress of that entity and from all entities to provide the overall migration progress.

In OutSystems, the MigrationLog Screen uses a couple of **Aggregates**[11] to fetch the necessary data and a JavaScript function to refresh the results every second, forcing OutSystems to detect the data changes.

The complete explanation of the functions that provide real-time updates is in Appendix F.

4.4.5 Reuse Migration Plan

Some migration plans can be useful for the future. The creation of a mechanism that allows the developer to reuse an already existing plan can save development time. With this feature, the developer does not have to create a migration plan every time he wants to migrate the same application's entities.

The re-utilization is done by creating a new plan that receives a new auto-incremented ID, and cloning only the necessary information (excluding logs). Whereas re-running the same migration plan would cause loss of data, i.e., the log information saved would be overwritten by the new migration log and this information would be lost, making it impossible to consult the log of the original migration.

The clone can be performed at the MigrationDashboard Screen, which provides the full list of executed migration plans.

The server action responsible for fetching the plan and creating a new one is present in Appendix G.

4.4.6 Schedule Migration

In data migration projects, it is very common to configure migration for the future. Many developers limit data migrations to evenings, weekends, and other times when system usage is low. Scheduling a migration allows the developer to do a set of other functions without having to wait for the migration to finish.

An auxiliary boolean attribute was added to the *MigrationMaster* entity called *IsExecuting* that acts as a software flag. This attribute can only be set to true if the migration is not yet finished, this means that the *IsFinished* attribute has to be false when the *IsExecuting* is true.

To execute asynchronous logic, OutSystems uses **Timers**[12]. It also provides a Scheduler Service that checks for actions that are ready to run and execute their action.

[11] Used to fetch data using an optimized query. See more on Section 1.2.

[12] Used to execute application logic periodically on a scheduled time. More on Section 1.2.

To benefit from the optimized OutSystems' tools, a solution was built to provide schedule migrations on BirdlyData, using Timers. First, a Timer, called *GetSchedules*, was created in the solution, this timer is activated every five minutes and executes an action to check if there is something already executing (*IsExecuting* column), if not, then checks if there is a pending migration ready to run. One migration is ready to run if the scheduled date of the migration is the same or before (less) the timer check. This timer then awakes another timer responsible for starting the migration. When wake, this other timer sets the *IsExecuting* attribute of the *MigrationMaster* plan to true, runs the migration and sets the attribute again to false after finishing.

This approach permits running a schedule migration five minutes after its schedule date, but, in the end, all migrations ready to run are executed in a small scope of time.

4.4.7 Notifications

For each migration plan, the developer should be able to be notified that the migration was finish, as well as the possibility to check the detailed log. This feature is especially helpful in schedule migrations since the developer does not know when the migration is finished. Nonetheless, the feature was also implemented for not scheduling migrations, allowing the developer to always chose if he wants to be notified or not.

Early in the process, at the New Migration Screen, the developer can determine if the migration plan is set to **notify** and, if true, add several email addresses. The notify flag is going to be checked at the end of the migration in both schedule or normal migrations.

This feature was added through the creation of a new application. The application created, *BirdlyDataNotifications*, is a Traditional Web OutSystems Application, which provides mechanisms to send Emails, on the contrary of Reactive Web Applications like the BirdlyData Core.

OutSystems provides tools that allow using built-in emails similar to Web Screens, and it is possible to create, compose the email content and send it. To create the Email content, an Email layout was used with some input variables, such as: *MigrationMasterId*, description (expression with a link) and the boolean *IsSucess*. With this variables, it is possible to provide more detail about the migration, building a link to consult the migration log. The email informs the developer if the migration was successful or if it was completed with errors, as presented in Figure 4.7.

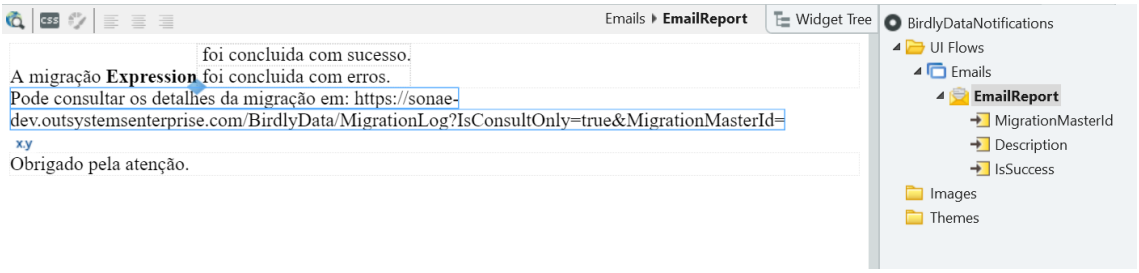


Figure 4.7: BirdlyDataNotifications - Traditional Email Report

Still, inside this application, a PUBLIC server action[13] was created to send Email notifications. The Email addresses are also used as input of this server action, with the format of a text separating the addresses with commas. More information details about BirdlyDataNotifications are illustrated on Appendix I.

At BirdlyData, the dependencies were managed, adding BirdlyDataNotifications as one of them. This allows calling the PUBLIC *SendEmails* action after the migration is finished. The log information result is collected and sent to the action as input.

4.4.8 Migrate *Users*

Applicational data typically has references to the hybrid[14] *User* entity. OutSystems platform even provides a Users Application Tool, where the developers can manage their end-users and create them to test their applications. But to create a *User* record at run-time is a more complex task.

So, a step of the migration process has to traverse all foreign keys and understand all records that are holding said *User* records, matching the corresponding *User* between environments and then take that into consideration when migrating the application data. This process consults and inserts into the data mapping table as any other entity, but it is not implemented inside the extension. Instead, the references *User*s are migrated before calling the extension with Service Studio.

The following specifications were in the origin of the functionality implementation:

- The only way to create a new *User* record in Service Studio is by adding a dependency to the *Users* back-office and import the action **CreateUser**, which allows creating a *User* record. Otherwise, OutSystems does not give developers permission to insert records into this entity.
- The amount of *Users* records created into the OutSystems platform has additional licensing costs, because of this, the migration of users is made differently, i.e. only relevant (referenced by the entities to migrate) *Users* records are migrated.
- An application's *Module* can have different *User Providers*. The *Tenant* ID is used to find the *User Provider* of the *User* record, which means that different *User Providers* have different *Tenant* IDs. The case of migrating entities that belong to different *User Providers* (Multi-tenant) is common and should also be treated.
- The unique attribute of *User* is its *Username* combined with the *Tenant* ID to match the *User* entity between different environments.

The solution is not so straightforward because, in the extension, it is possible to access other environments just with the environment database credentials, but in OutSystems, it is only possible to access the current database environment. For this reason, running the *CreateUser* server action on the current environment would only duplicate the same *User* record because the action creates

[13] Public server actions allow the element to be added as a dependency by other modules.
[14] Similar to system entities but also interact with applications. More on 3.3.3.

the record inside the current environment where it is called. This leads to the explored solution: call the *CreateUser* server action on the target environment instead of the source through an API.

The full functionality implementation required changes through all BirdlyData application, such as the creation of the new Service API, creation of new server action in BirdlyData and some modifications inside the .Net extension. These are documented in detail in Appendix J.

1. BirdlyDataAPI Service

   The service exposes one REST API with a single HTTP POST, which receives the following parameter types:

   - Body: *User* record — the record to be created in the target database (contains the Username)
   - Header: API Key — for security/authorized calls
   - Header: Tenant name (*UserProvider* name) — to provide multi-tenancy[15] and compose the unique *User* identifier

   The target environment exposes the API, but it is the source environment that consumes the API through a call. The HTTP POST function uses the *CreateUser* server action from the system User back-office to create the user in the target database.

2. BirdlyData Core

   In Service Studio, a new flag was created in order to activate the migrate *Users* functionality.

   Before starting the migration, the relevant *User* records, i.e., the ones that are referenced by the selected entities, are fetched through a series of advanced SQL queries. Calling the POST method for each one and updating the data mapping table with the source and target identifiers match.

3. BirdlyData Extension

   While migrating the entities, system's *User* entity is ignored since it was already migrated in Service Studio. If another entity has a foreign key to *User*, then the data mapping table is consulted as usually.

4.4.9 Differential Migration

Migrating every single record present in the source environment every time the developer wish to migrate an application could be a very heavy process for large tables. Differential migrations can be very fast compared to creating full migrations. The option to migrate only the last inserted records on the source environment, since the last migration (partial migration), has several benefits such as: reduced waiting time and avoid duplicated records on the target environment.

[15] Single instance of a software application serves multiple customers. See Section 1.2.

Some possible solutions were discussed. One of them was based on the comparison of records between source and target environment, which is the most "correct" way to properly capture the only data that has changed since that full migration. However, this option would imply searching for each source record if any target record matches it, and if there is not any match, then insert it into the target. For every source record, it is necessary to iterate all target records, which leads to a very complex and over-processing $O(n^2)$ algorithm. For this reason and respecting the performance's goal, this option was refused.

Eventually, a more simple solution was found to solve the inherited problem. First, a boolean called *IsDifferential* was added to the *MigrationMaster* entity to flag the feature. Then, an auxiliary entity (**PartialRecords**) was created to store the last migrated value's identifier of each entity.

On the extension, the last entity's record inserted is saved into *PartialRecords*. Follow migrations with the activated flag will only migrate records that have a bigger identifier than the one stored at the data table, i.e., new records.

A detailed explanation of the functionality and use of SQL queries is in Appendix H.

4.4.10 Mask Data - Anonymize

Production data can include intellectual property, personally identifiable data, protected health data, as well as financial data, such as payment card information. Data masks are used to protect this types of data. When doing a data migration, the dataset with sensitive data is copied, which has a high risk of a data breach, that can compromise the responsible organization. To comply with the **GDPR** norms, a data security technique to protect sensitive data was implemented. The obfuscated replica is then used instead of the authentic data, where individuals are no longer identifiable.

There are several techniques able to obfuscate this type of data before migrating. The technique chosen to obfuscate data was *Anonymization*, which is a data processing technique that removes all sensitive information by substituting the values to their data type's default value. Other forms were also measured, but the performance of manipulating and mix data (scramble) would always be worst than just retrieve the default value.

This functionality is defined at the EntitiesList Screen since the mask is set at the entity's attribute level. The first step is to fetch the list of attributes of each entity through a SELECT SQL. These attributes are then added to a Structure containing the attributes names and the *IsMask* boolean flag. After the developer defines which entities to migrate and sensitive correspondent attributes to mask, the columns' names are saved into the database table **MaskAttribute**.

While migrating the records, on the extension side, this data table is consulted, and if the current entity's attribute being migrated is considered sensitive, then the value is replaced by its default type value.

The detailed information of how to fetch the entity's attributes, import them into the EntitiesList Screen using a *Structure*, manipulate the attributes' list view and the mask algorithm is presented from Appendix K.1 to K.5.

4.4.11 Delete Target Data

For testing purposes is very common to delete the data of the target environment before migrating new data. This allows a clean start of the test, where the selected entities are cleaned before receiving the new information. This is also helpful to prevent duplicated records.

Manually, it is possible to clear the data tables before the migration but would require building and run an external SQL script with the DELETE statements for the entities. However, the user of the tool, even being a developer with knowledge of the field, should not be obligated to build and run scripts to support the migration.

To automatically delete the selected entities before the migration, another flag was added to the entity *MigrationMaster*, called *IsDeleteTarget*. Setting the option to empty the target database will directly block the option to perform a differential migration, and set the *IsDifferential* attribute to False, i.e., cleaning the target will also clean the *PartialRecords* information regarding that environments entities. This is essential to prevent errors because if another migration is done after, with the differential migration triggered, then it would only migrate new records, but in reality, the target was empty and should receive all records present on the origin.

The extension's function uses several SQL queries to build this feature. The entities are deleted in the reverse order of their dependencies to successfully clean the target entities without breaking foreign keys. More details on Appendix L.

4.4.12 Call Migration through API

Developers continuously build, test, and deploy iterative code changes. Continuous Integration and Continuous Delivery (CI/CD) automation seek less human intervention (or no intervention at all) from the development process until the code deployment. To help and automate the CI/CD process, a new API was created to support the migration tool. This allows to more easily integrate the migration tool with the CI/CD automation, for example, with Jenkins, which is an extensible automation server that can be used as a CI server or as a continuous delivery hub for any project. The ability to preview environments using migrated data is essential to optimize pipeline automation.

The new API called *MigrationAPI* was created at OutSystems Service Studio, with two REST API methods, one to perform a migration and another to get the status of the migration. The REST API documentation is annexed in Appendix M.

- /Migrate

  The HTTP method first fetches the migration plan given its identifier and creates a clone. This cloned plan is the one that will be used to perform the migration. For security, the URL also asks for an APIKEY parameter to prevent unauthorized calls. The introduced APIKEY must match the stored value that is in the site properties.

The call retrieves a text result with a link to the other HTTP call, the GetStatus, such as: "Go to /GetStatus?MigrationMasterId=*MigrationMasterIdOfClone* to check the migration status", where *MigrationMasterIdOfClone* is the identifier of the clone plan.

- /GetStatus

  The HTTP GET method *GetStatus* uses the identifier of the plan to fetch the *MigrationMaster* and checks the attributes *IsExecuting* and *IsFinished* to provide the status of the migration that could be: finished, executing or not started.

4.4.13 Self-referencing foreign key

Some entities can have an FK constraint referencing themselves. This is not very common, but it is possible and can lead to complications while migrating the column that represents a self-reference (FK integrity lost). More details about this case and discussion of the implemented solution are present in Appendix N.

4.4.14 Static Entities Dependencies

Application entities usually have static entities[16] as foreign keys. As already mentioned in 2.9.2, the OutSystems platform does not allow to change these entities. To provide FK integrity with static entities, it is also necessary to match these between environments. The solution to this particular case involves storing both target and source entities locally since usually they have very few records, before starting the migration. Then find which target record has the same label as the source record, retrieving its ID.

More details about the implementation of the static entities' FK matcher is in Appendix Q.

4.5 Optimizations

Automation helps the data migration making it simple and fast. After the implementation of all functionalities, some load and stress tests were performed to ensure system stability. These tests detected several migration delays and connection timeouts. For a more scalable data migration approach, some performance activators were implemented into the solution.

4.5.1 Indexes

Initially, the extension could read, transform, and insert into the target approximately 57 records per second. But migrating applications with extremely loaded entities (millions of records) could be a very slow process, taking several hours to finish. The beginning of the migration was always faster since the mapping table did not have many records, however, with the migration going and the constant consult and insert into the mapping table *RecordIdRelation*, the process started to become slower. The mapping table size is directly proportional to the number of rows being

[16] More information regarding Static Entities is present in 2.9.1.

migrated, i.e., for each row migrated, a new row is created on the mapping table. The algorithm inserts into the mapping table before moving to the next record, but for every FK column, the mapping table is consulted, and if the entity has several FK, then the consults to the table also increase. The size of the *RecordIdRelation* increasing will also imply more time to perform a consultation and, consequently, the migration process delay.

To speed up searching in the database, a new index was created in *RecordIdRelation*, as shown in Figure 4.8. An index is used to efficiently find all rows that match some column(s) in a query and then walk through only that subset of the table to find exact matches. Without indexes, the SQL server has to walk through the whole table and check every row to see if it matches, which may be a slow operation on big tables.

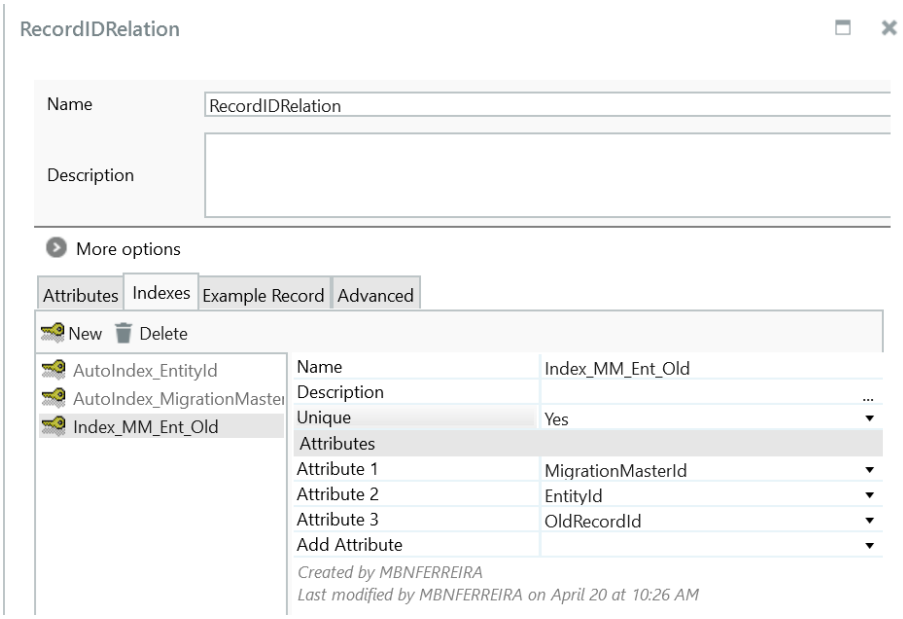


Figure 4.8: Index of Mapping Table

The **Index_MM_Ent_Old** index was created with the attributes used to retrieve the target ID when consulting the data mapping table, which are *MigrationMasterId*, *EntityId* and *OldRecordId* (the source record identifier). It is a UNIQUE index, which means that indexed records cannot have repeated values in the index attributes, i.e., for a determined entity in a determined migration plan, there is only one target record identifier, given its source record identifier.

4.5.2 Mask Object Transformation

The first implementation of the mask feature required a database call per column. This means that, for each attribute, the algorithm was calling the database to search into the *MaskAttribute* entity if the current column being migrated had the mask flag (boolean set to true). The number of calls to the database increases for bigger tables that have many attributes and delay the data migration

process since database calls were made for each column in each record. For example, an entity with 10 records and 10 attributes would call the database 100 times.

The code optimization implemented to solve this problem was fetching the information before starting iterating the entity records and save it locally on an object. Then, instead of calling the database for each column, the local object is consulted.

The object created for this effect was the C# class: **Mask** that contains a long attribute corresponding to the migration master identifier, another to the entity identifier and a *Hashtable* to store both column name and respective *IsMask* boolean. A *Hashtable* (hash map) is a data structure that implements an associative array abstract data type. The structure maps keys to values and uses a hash function to compute an index (hash code) into an array of slots, from which the desired value can be found. In this case, the keys correspond to the columns' names of the entity and the values to the boolean *IsMask* stored at the *MaskAttribute* entity. The structure also allows the search to be faster since it uses the index to search for a determined key's value.

The local manipulation of the object's *Hashtable* allowed to avoid unnecessary mask searches. The complete object constructor code, creation and use of the local object are explained in Appendix O.1.

4.5.3 Dependency Object Transformation

Another heavy process was the one responsible for, for each column, check if it is an FK and, in that case, retrieve the identifier of the entity that corresponds to that FK. This entity identifier is then used, along with the migration plan identifier and the source record identifier, to retrieve the correct target record identifier from the mapping table. Two drawbacks of the algorithm were detected at the optimization phase:

1. The algorithm already checked if the current entity being migrated had dependencies or not, so it did not have to check, for all attributes, if the current corresponds to an FK. However, a list of dependencies (composed with the entity's identifiers) fetched from the database was being consulted in every column.

2. Another function with two SQL queries was used to find if the current attribute was an FK and retrieve its entity identifier. Again, this function uses the query to get dependencies which lead to the duplication of code and, again, another heavy process performed in every column of every entity with dependencies. If the current column is not even an FK, the query would also be called. Additionally, if the current column was indeed an FK, the dependencies list was being iterated to search if the column is a dependency. This method adds unnecessary processes that exponentially slower the migration of entities with many dependencies.

The solution found to obtain a better performance was similar to the creation of the *Mask* object, but in this case, created class was: *Dependency*. This class has the attribute *TableName* that stores the physical table of an entity and a *Hashtable* called *AttributeDependencyId*. This

structure contains the attributes/columns' names as Keys and the corresponding entity identifier FK (*entityId* of the dependency) as Values.

The local object allows to fully eliminate the dependencies list by using the *HashMap* method *ContainsKey()* to check whether a particular key is being mapped into the *HashMap* or not. Additionally, the database calls used to retrieve the target record identifier from the data mapping table ,for each FK attribute, were also eliminated. The full explanation of the object's manipulation is present in Appendix O.2

The performance obtained by the last optimizations was not as good as expected. The average of records migrated per second increased to approximately 143, which makes the application 86x faster. Immediately after these changes, the time to migrate, for example, **5 entities with 420,934 records**, decreased from 2 hours to 40 minutes.

Despite the performance being improved, it was still not enough since this amount of records is just a sample of the real enterprise database tables, which can have the double size. For large database migrations, these times are still not acceptable and should be improved. Ideally, for this sample, the time of migration should be around 5 minutes.

4.5.4 BULK INSERT

The performance of ETL processes for large quantities of data can always be improved by experiment alternative techniques. Some researches [49] about different load methods and accordingly to Microsoft's Data Loading Performance Guide [4], the best way to insert a million rows of data into SQL Server is by using the BULK INSERT command because multiple rows insert consistently outperformed single row insert. The BULK INSERT is a process provided by the database management system to load multiple rows of data into a database table.

Before adding the BULK INSERT modification, the application was inserting the data with a series of INSERT statements, where the INSERT was done while reading the source records, i.e., one INSERT for each record (each row of the data table). For each INSERT a database call is made, with a sample size of 420,934 records the program will make 420,934 calls to the database, what is one of the program slowness' motive.

At the same time, implementing the BULK INSERT has one disadvantage, it blocks the possibility of retrieving the migration log updates row by row because the BULK operator inserts multiple rows at a time. However, the performance boost is still preferable over the lack of this feature.

SQL has an inbuilt mechanism to import a large volume of data and there are inbuilt classes in .NET which support bulk insert that help insert multiple records in the database, for example, the *SqlBulkCopy* class. Although it is possible to easily tweak the behaviour with some options, this class does not allow to output the newly generated identity value, which is necessary to store into the mapping table.

While searching for similar solutions, another resource was found, the C# *Bulk Operations* by ZZZ Projects, an improved version of *SQLBulkCopy*. This package allows to overcome *SQLBulkCopy* limitations and use flexible features to output inserted identity and concurrency column values and can be installed by managing the *NuGet* packages for the solution in Visual Studio IDE.

The solution's website [50] indicates the average times that the Bulk Insert Operation takes for different table sizes, this is represented in milliseconds on Table 4.1. The benchmark can be affected by several factors, such as index, column type, latency, throttling, etc. This method turned out to be more flexible and increased the performance of the solution.

Table 4.1: Bulk Insert - Performance View

1,000 Entities	**2,000 Entities**	**5,000 Entities**
50 ms	55 ms	75 ms

The construction of the INSERT string query disappears, instead, data is loaded into a local *DataTable*[17] object, used by Bulk Insert to insert the data into the target database. While reading the source records, new rows are added to the *DataTable* with the transformed data, but the insert is only done from X to X records, for example, 200,000 in 200,000 rows. To preserve the mapping table's logic, some alterations have to be made because the Bulk Insert mechanism migrates several records at once and there is no information regarding the new target ID while filling the *DataTable*. The insert into the data mapping table is divided into two steps: an INSERT while reading the records, with the source ID of the record and then after the Bulk Insert, an UPDATE is done with the correspondent target record's ID. After each Bulk Insert, the local objects are cleaned, which provides better performance since smaller blocks of data are used. The detailed decision-making and implementation process of this optimization is on Appendix O.3.

Previously modifications on the same sample (4.5.3) were taking 40 minutes, but with this last one, the tool migrates the same entities in 25 minutes (around 15 minutes faster).

4.5.5 BULK INSERT into Data Mapping table

The last optimization (constant clear *DataTable*) did not solve the latency[18] found on the UPDATE SQL calls made to the data mapping table (*RecordIdRelation*). The alternative was implementing the Bulk Insert to update the data mapping table as well.

DataMap is a new auxiliary C# class created to support the Bulk copy into the target database. Instead of the several INSERT queries, it creates a local *DataTable* similar to *RecordIdRelation*. The class provides methods to insert and update into the local object and, finally, persist the *DataTable* rows data into the database (*RecordIdRelation*). Implementation details are presented in Appendix O.4.

[17] .NET class that is a database table representation and provides a collection of columns and rows to store data in a grid form.

[18] Division of the INSERT into two steps required two database calls instead of just one. More on O.3.

This optimization had a really big impact on the overall performance of the migration process. For the same sample used before (4.5.3), the time to migrate the same 420,9897 rows was only 4 minutes, 21 minutes faster.

4.5.6 Load Data Mapping table in memory

As proven before, database calls are usually heavier than manipulate the data locally. The last database call left inside the loop was to get the target record's ID, for each dependency of each entity.

To cut all unnecessary database calls, a new auxiliary class **LocalMap** was built to fetch the data mapping table information (*RecordIdRelation*) and store it locally in a *DataTable*. For each entity, a new instance of this class is created, fetching the data mapping table for the current migration plan. *LocalMap* contains a method called *GetNewId*, that replaces the method *GetRelationId* already implemented[19], which allow consulting the data mapping table locally. Additionally, the size of the local data table is always shorter since it only contains the information of the current migration plan. The class methods and usability are explained in Appendix O.5.

For the same sample mentioned on the other implementations (420,9897 rows - 4.5.3), this optimization reduced the migration time to only 24 seconds.

The combination of these last three optimizations helped to migrate 748633 records from 6 entities in 80 seconds (1min20sec), given an average of 9358 records per second, which provides a performance boost of 164x faster than the original tool implementation (without optimizations).

4.6 Summary

The process of building a solution to solve the migration obstacle in OutSystems was iterative, with several steps to develop a complete and efficient solution. Overall, the proposed solution solves the concrete problem of migrating OutSystems data from a production to a non-production environment, raised in Section 1.4. Equally, the data migration tool that sets up the solution can solve many other problems since it allows to perform migrations between any environments, independently if it is a production or a non-production environment.

The final result of the proposed solution is the possibility of performing a data migration between any OutSystems environment without any manual work (no additional code). The tool allows the user to automatically perform all operations that are necessary to run a data migration. The original objective of building a tool able to manage and stage a data migration was reached with the combination of the OutSystems platform and C# code integration. Taking advantage of some information stored in the OutSystems platform system tables is indeed possible to retrieve important information about the data used by the applications. With this information is possible to build a GUI for other testers and developers to create and manage their own migrations plans and data access configurations. By taking advantage of the Integration Studio, it was proved that

[19] Function that executes a SQLCommand call to *OSUSR_m7a_RecordIdRelation* in order to find the new/target ID. More on E.9

it is possible to build an extension capable of executing the data migrations plans by open SQL connections to both source and target environments at the same time. Additionally, the extra features offer extra efficiency and solidify the final software product.

Several new obstacles of the OutSystems platform were found during development. To overcome those difficulties, many solutions were found by trying to find some workarounds. These workarounds were basically ways of forcing OutSystems to manipulate data from other environments that not the current one (origin). In conclusion, the tool is able to transform several routine tasks, structurally complex and manual (by constantly writing custom scrips), that require a lot of time, into an automated and effective process, with very little human intervention.

Chapter 5

Evaluation

In this chapter, the software built to fulfil the current dissertation hypothesis was tested from different perspectives: non-functional testing and functional application testing. The non-functional testing stage involves Data Quality validation, where the data migrated is analyzed and compared with the source data and the transformation requirements. On the other hand, functional application testing is based on the application's behaviour with the migrated data and how the processes and functionalities are affected.

The objectives of the data migration testing/verification are:

- Verify if all the required data was transferred.
- Verify if the target tables are populated with accurate values (including FK integrity).
- Verify the absence of data loss.
- Verify the performance of the application with the new data.

To provide a real case scenario, a deployed application from the enterprise's backlog, called Rocket - Contact Lenses, was used to test if the application tool can migrate data between different environments and keep the data integrity untouched. RocketContactLenses is an application created to manage contact lenses, and it is used by a bigger enterprise application called Well's. Well's is a reference brand of health, beauty, well-being and optics for all Portuguese families. Eyecare is a strand of Well's offers, and that is where the application RocketContactLenses gains importance. The major functionalities of RocketContactLenses are:

- CRUD operations of relevant entities that support the contact lenses information.
- Filter lenses information.
- Export lenses information to Microsoft Excel.
- Generate catalogs as XML documents.

- Import Microsoft Excel lenses catalogs.

The application selection was based on its complexity and data types: RocketContactLenses uses 8 regular entities, 3 static entities and the system's entity - *User*. In total, The application data has 11806 records plus the referenced user records. The RocketContactLenses' data model is presented in Figure 5.1. *Lens* is the main entity that references (has FK to) the *LensType*, *VendorName*, *Brand*, *User*, *ReplacementType*, *Capacity* and *LensColor* entities, where the last three are static entities, *User* is hybrid/system and the others are application entities. All applicational entities reference the system table *User*, which requires migrating users to be activated in this test set. The *Lens* entity depends directly on the *VendorName*, *Brand* and *LensType* and depends indirectly on the *LensTypeAttribute* since *LensType* depends directly on/references the *LensTypeAttribute*. This amount of complexity was ideal for testing the migration tool, given that it owns a network of dependencies. To the correct operation and simulation of the RocketContactLenses in the target environment, it is not necessary to select the *FileImport*, *TrackImport* or *Config* entities to the migration process, these entities are auxiliary to the configuration of the import and export mechanism and the data can be generated with the information from other applicational entities. Therefore, the entities used in the test suit were: *User* (only records referenced by the other entities - 3), *LensType* (5 records), *LensTypeAttribute* (5 records), *Brand* (7 records), *VendorName* (3 records) and *Lens* (11654 records), 6 entities with the total of 11677 records. The simulation of the migration through the BirdlyData's UI is annexed in Appendix P.

Once the application is migrated successfully, end-to-end system testing is performed in the testing environment. For that, it is necessary to identify separate environments for Pre (source) and Post (target) Migration environments to carry out any verification that is required as part of testing. The source environment selected for this validation was the Development (dev) environment, while the target environment was the Test (tst). This choice was made based on the minimal impact that the data migration would have, preventing possible errors in production environments. Since this dissertation was done in an enterprise context, it was not allowed to use any production environment of major applications since this is a test stage and the possible loss of data integrity could be critical for the company.

5.1 Data Quality Validation

Data migration projects' mistakes are more likely to be made in the mapping and transformation logic. Additionally, runtime failures such as network failures or broken transactions can lead to several issues such as missing records or values, incorrect values, duplicated records, non-formatted values and broken relationships across tables. Quality Assurance (QA) ensures that the source data is correctly loaded into the target tables and additionally that the application substitutes default values, alter, ignores, and reports invalid data correctly (transformations). This process is explained in Section 2.6.

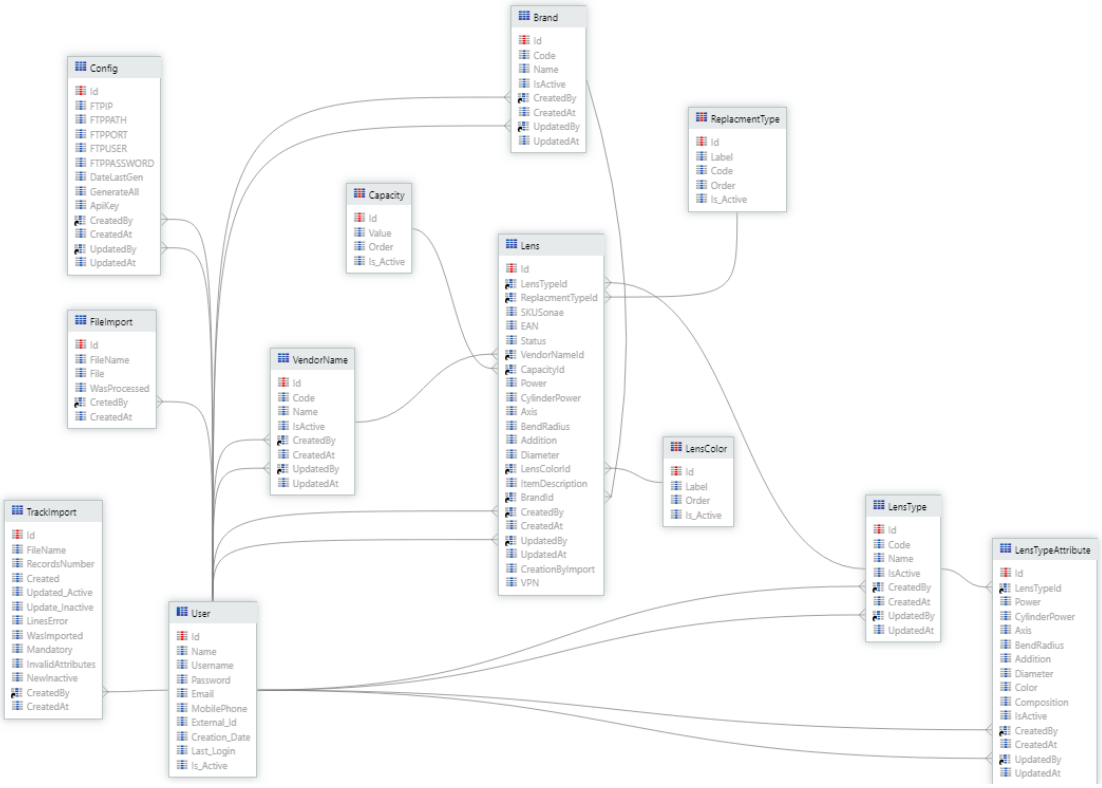


Figure 5.1: Rocket Contact Lenses Data Model

The first step of the validation process was migrating all entities used by the test application RocketContactLenses. However, some prerequisites must be fulfilled to perform the data migration:

- BirdlyData should be installed in both source and target environments.
- The test application (RocketContactLenses) should be installed in both source and target environments.
- The database connections, both source and target, should be created in the BirdlyData application tool inside the source environment (where the migration call is performed).

The test strategy starts by comparing source and target data sets, and the main concern is to check the referential integrity between tables, transformation and data validation for inserted records. After the data migration, the supporting tool was analyzed according to the following QA strategy's scope which was previously identified on Table 2.2:

- Completeness
- Correctness

- Consistency
- Interoperability

5.1.1 Completeness

The data reconciliation/verification testing stage prevents issues from going unnoticed, avoiding possible damage to the overall accuracy of the data and inaccurate insights. In this stage, the target data is analyzed, searching for missing or unexpected records. The target dataset is considered complete if all the records of the source dataset were fully migrated - Record Count Verification.

In order to get a number of a record in a target or source table, we can use a simple query (Listing 5.1) to validate the data before and after migration. The use of the same query is due to, in this case, the physical table names of the entities are the same in both environments. The results obtained by running the same query in both target and source environments are the same (since the target environment was cleaned before migrating), as revealed by Figure 5.2, where the *osDEV1* is the database name, also known as *Catalog*, of the source environment and the *osNOP2* is the database name of the target environment.

```
1 SELECT
2 (SELECT COUNT(*) FROM OSUSR_hew_Lens)+
3 (SELECT COUNT(*) from OSUSR_hew_LensType)+
4 (SELECT COUNT(*) from OSUSR_hew_LensTypeAttribute)+
5 (SELECT COUNT(*) from OSUSR_hew_Brand)+
6 (SELECT COUNT(*) from OSUSR_hew_VendorName)
7 AS SumCount
```

Listing 5.1: Count records

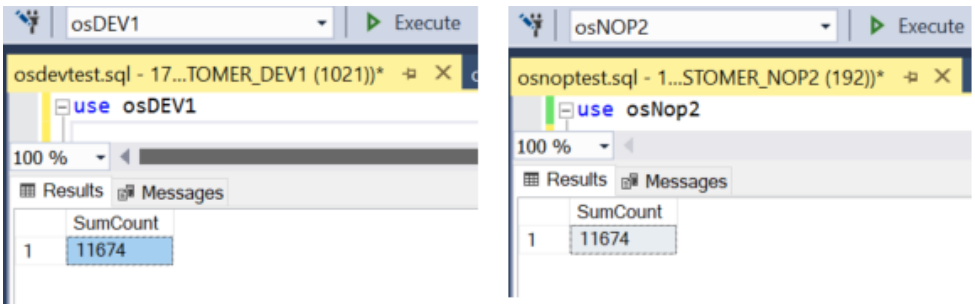


Figure 5.2: Compare completeness of records migrated

5.1.2 Correctness

A standard method for checking the functional correctness of applications is using *test cases*. In a data migration project, this means checking if the attributes of selected entities are correctly

migrated. With a large dataset, the focus goes to checking relevant but selected attributes of all objects. The large test set used, 11674 records, does not allow a deep review verifying the records one by one because it would be very time consuming and error-prone. The strategy implemented to verify the correctness of the data migrated was based on each table or characteristic attributes. The test suite was divided into similar groups, performing comparisons into random records inside those groups. This methodology uses random samples inside a similar group but is not completely random, which reduces the risk of a validation failure since it does not assume that the error is uniformly distributed.

The simplest entities can be fully tested since they have a low amount of records and columns, these entities are: *Brand*, *VendorName* and *LensType*. The detailed comparison of *Brand* is presented in Figure 5.3, where the top image represents the data on the source environment *(dev-osDEV1)* and the bottom image represents the target environment *(tst - osNOP2)*. The results are obviously similar except in three columns: the *ID*, *CREATEDBY* and *UPDATEDBY*. The justification for the difference between *ID* columns is that this value is always auto-incremented since it is the record's primary key, which was an expected result. The *ID* starting at 322 only means that there were already 321 records inserted but deleted before the migration. On the other hand, *CREATEDBY* and *UPDATEDBY* are references (FKs) to the system table *User*, these columns have different values because the *ID* (auto-incremented) that they refer to is different in the environments. Since this *User* was being used by these *Brand*'s records, was migrated, and the data mapping table registered the old ID (5156) and the new inserted (or already existent) *User ID* (2514). Every time the mechanism reads a column referencing as *User* records with the value 5156, it replaces to 2514 - the target and correct value. To prove that the *User* referenced by those environments is the same, another simple query is called as in Figure 5.4. The query searches for each *User ID* in each environment and the result obtained is the same, proving that both *ID* reference the same *User*: *CL_MANAGER*.

Both *VendorName* and *LensType* were also successfully migrated, the source and target records comparison is presented with more detail on Appendix P.

The entity *LensTypeAttribute* has references to *LensType* and *User*, but it has only 7 records, so the migration can also be fully verified. *LensTypeAttribute* represents a common database table where the database from source and target environments do not have the same order, additionally, the source *(dev)* has one more column - *BRENDRADIOUS* - that does not exist in the target and should not be migrated. The present data migration tool performs verification of which source columns exist in the target to prevent errors while trying to migrate data to a non-existent column. The results from the migration of this entity are presented in Figure 5.5. In this Figure, it is possible to verify that there exist 16 columns on the source environment but only 15 columns on target because the column *BRENDRADIOUS* does not exist on *osNOP2* (target). The *LENSTYPEID* values are justified by the same reason as the *CREATEDBY* and *UPDATEBY User*'s FKs, the column references another entity (*LensType*). The source and target ID values difference correspond to the mapping of the environments records. Figure 5.6 represents an example of one of these mappings where the source ID 2 corresponds to the target ID 331 both the same *LensType* object - The *Tórica*

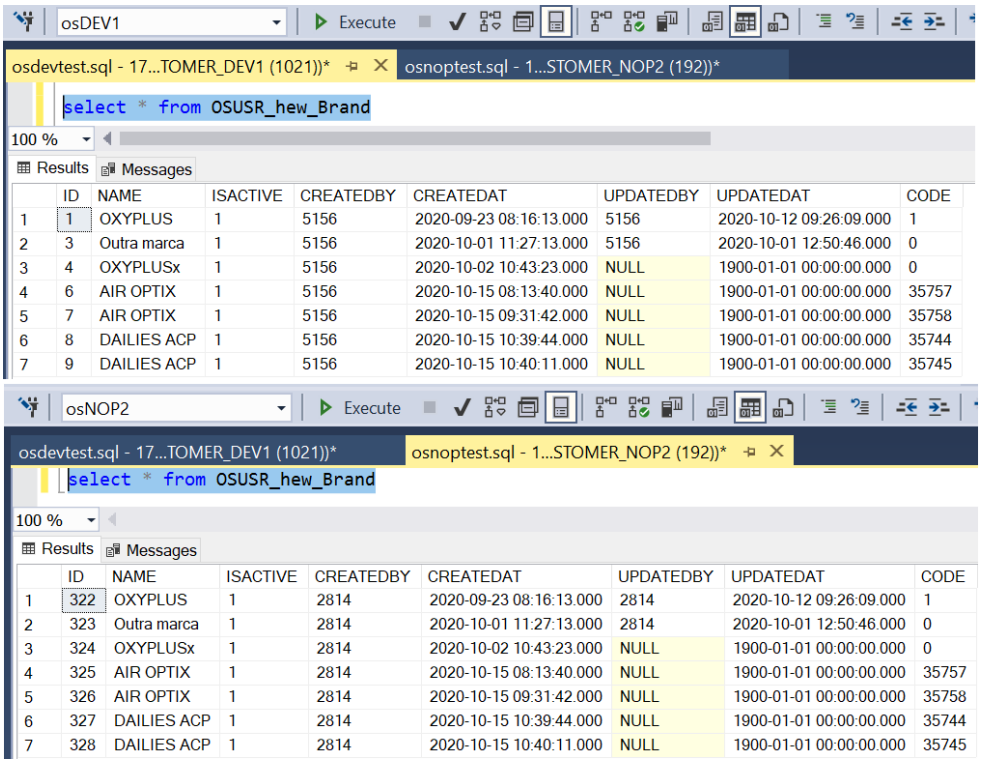


Figure 5.3: Compare Brand records after migration

Lens with code T003.

Finally, the entity *Lens*, which has more dependencies, more columns and more records, has to be analyzed by sections. These sections were grouped by business logic concepts, and some records were randomly chosen from these groups to be tested.

The business test groups were defined by *Lens*' type. This decision was based on the most important business rule of RocketContactLenses: Every *Lens* of a given *Type* should have the same *Lens Type Attributes*. For example, any *Lens* of type *Tórica* should indicate its *Power*, *Bend Radius*, *Diameter*, *Cylinder Power* and *Axis*, but it should not have any information about the *Addition*. But, for instance, *Tórica Multi-Focal* Lens should have all the attributes. These group requirements are represented in Figure 5.7.

With the current test data, the division into groups originates 5 different groups. From these 5 groups, some random record was compared with the source. The query that retrieves the data set group by *LensType* is listed in 5.2. In this case, the *LensType* that corresponds to the ID equals 1 is *Esférica*, for the other groups, their respective ID replaced the 1.

```
1 select * from OSUSR_hew_Lens
2 where  LENSTYPEID= 1
```

Listing 5.2: Group by test set for validation

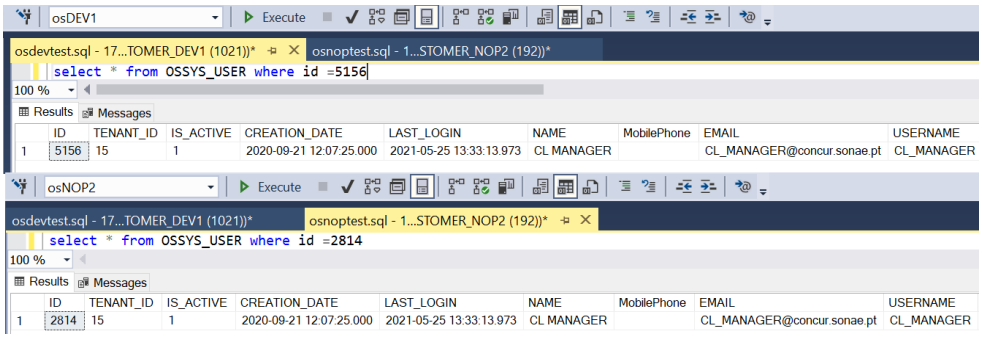


Figure 5.4: Compare User FK after migration

To select the target data set for the *Esférica* Lenses, the same query was called but with the respective target ID, that is 330. The justification is the same as the *Tórica* Lens environments mapping on Figure 5.5.

The dataset retrieved by this *Type* of *Lens* has 398 records in source and target environments, which is also already another confirmation about the completeness of the data.

To guarantee that the selection of one record from this data set was indeed random, an algorithm [31] that offers true random numbers was used to generate a random number between this interval of data - from 1 to 398 - and the number generated was 356. The following comparison represented in Table 5.1 is from both environment records at the same position, i.e., same record (the record will be the same since the target was previously cleaned).

The values presented in Table 5.1 are the same for almost all columns except the columns that have references to other entities, such as *BRANDID*, *VENDORNAMEID*, *LENSTYPEID*, *CREATEDBY* and *UPDATEDBY*. These entities' data were also inserted into the target database before the *Lens* entity, and their ID provided by the Insert is fetched from the data mapping table that stores the value of the record in source and target environments. Further, the IDs points to the same entity object with the same data. Some columns are FK to static entities like *REPLACEMENTID* and *CAPACITYID*, so the value is static (the same in both environments). This dataset test covers data from the following types: *bigint* (long), *int*, *nvarchar* (string), *DateTime*, *decimal* and *bit* (boolean), which are the most common data types used by OutSystems applications. In addition, the business rules that originate the division of the test data into groups were verified in this dataset. As demonstrated in 5.7, the *LensType Esférica* requires values from the attributes: *Power* , *Bend Radius* and *Diameter* , but no information about: *Cylinder Power* , *Axis* or *Addition*. In both environments, the columns *POWER*, *DIAMETER* and *BENDRADIUS* present valid values, on the contrary of CYLINDERPOWER, *AXIS* and *ADDITION* that present the default value of their type, since the *ADDITION* is a *nvarchar* field than the default is empty, the *CYLINDERPOWER* as a *decimal*, has the default null value of 0.00 and the *AXIS* as an *int* has the value 0. By compliance with this business rule and the correctness of the other values in the test set, the group can be set as successfully tested.

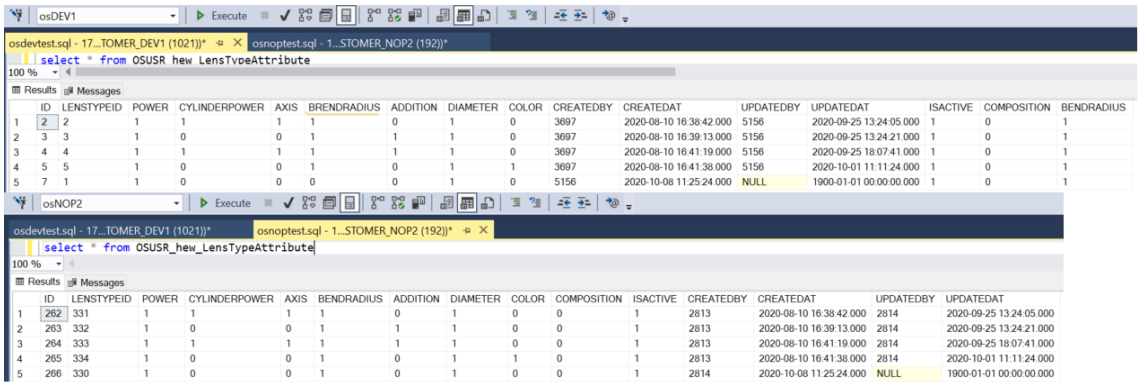


Figure 5.5: Compare LensTypeAttribute after migration

The other remaining groups' correctness tests are similar to the *Esférica LensType* group and are mentioned in Appendix P.

5.1.3 Consistency

Data validation makes sure that data migration was done, it can meet all business rules and mapping requirements, but it does not necessarily mean that in the target environment, everything works as predicted. Running the business processes in the target environment is important because some issues or special cases still need to be resolved before completing.

RocketContactLenses is an application that supports CRUD functionalities, but it is also possible to search the *Lens* data table using filters. To validate the application's behaviour with the migrated data, some filters were set, and the results compared with the source environment. An effective processability test should duplicate process conditions and result in information directly related to the processing behaviour of the test application.

The filters set in the test use FK attributes to better validate the integrity of the relationships between entities, the selected filters are:

- *VendorName* — ALCON VISION CARE
- *LensType* — Multifocal
- *Brand* — AIR OPTIX

The top screenshot on Figure 5.8 represents the behaviour of RocketContactLenses running in the source environment, which is evident by the URL since this uses the **dev** acronym to represent the development environment. On the other hand, the bottom screenshot represents the behaviour of the test filters on the target environment, represented by **tst**. The application behaviour is the same in both environments, retrieving the only record with the given characteristics.

Others filters were randomly selected, and the behaviour of the application and the data results remained the same in both source and target environments. Running the same business process allows confirming that the target environment works as expected (just like the source).

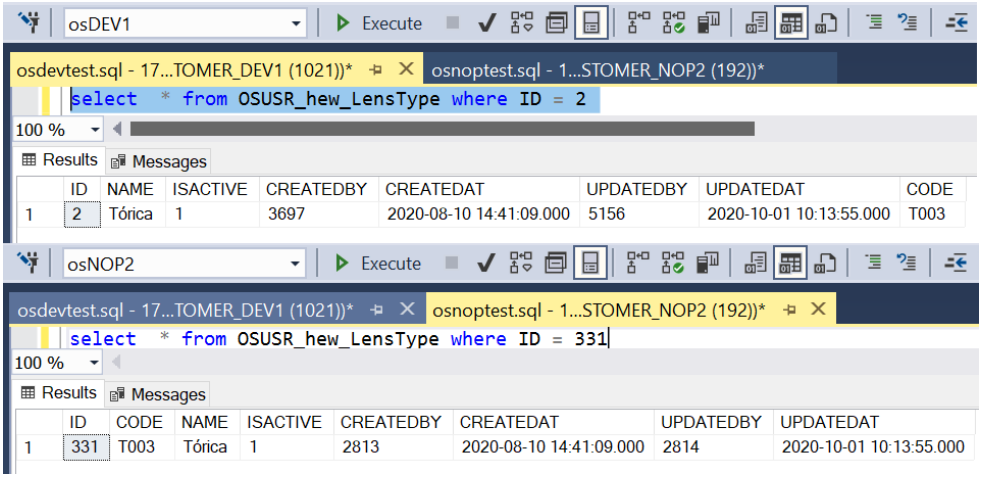


Figure 5.6: Compare LensType FK after migration

5.1.4 Interoperability

The last step to fully validate the data migration tool is to test how the target application with the imported data interoperates with other applications, namely, how the RocketContactLenses interoperates with the Well's application. Well's owns a mechanism to fetch the RocketContactLenses generated XMLs and use their information to show and configure the lenses information at the online store. The behaviour of the Well's application depends on the XML, i.e., if the generated XML is the same in both environments, then the Well's application must have the same behaviour on the target environment with the migrated data. To prove that both original data and migrated data originate the same XML, two new XML files were generated through the RocketContactLenses application UI, one for each environment.

There are several tools built to compare files, but after some research, a simple and easy to use plugin was selected, the plugin is called (*notepad++ file compare plugin*) and supports XML files comparison in NotePad++ (source code editor that supports several languages).

After the installation of the plugin and the download of the source and target generated XML files from the UI, both files were opened on NotePad++ and the comparison was made by running the plugin, the result can be consulted in Figure 5.9, which shows the editor with both files open and a successful message: "Files Match".

Identical XML files guarantee an identical behaviour of the Well's application while importing information (communicating) from the integration with RocketContactLenses.

5.1.5 Summary

This chapter provided a series of demonstrations of the main data migration tool functionalities and produced a full report of the validation steps performed on the migrated data. Some objectives were defined at this stage to validate the accuracy of BirdlyData, building a Quality Assurance strategy to test the tool with an appropriate and complete test dataset, where the data migrated was

```
SELECT t.Name, a.Power, a.CYLINDERPOWER, a.AXIS, a.BENDRADIUS, a.ADDITION, a.DIAMETER
FROM OSUSR_hew_LensTypeAttribute a, OSUSR_hew_LensType t
WHERE a.LENSTYPEID= t.ID
```

100 %

Results | Messages

	Name	Power	CYLINDERPOWER	AXIS	BENDRADIUS	ADDITION	DIAMETER
1	Tórica	1	1	1	1	0	1
2	Multifocal	1	0	0	1	1	1
3	Tórica Multi-Focal	1	1	1	1	1	1
4	Colorida	1	0	0	1	0	1
5	Esférica	1	0	0	1	0	1

Figure 5.7: Lens requirements by Type Query

compared with the original data. These several tests' results met the expectations, allowing us to consider the data migration successful. The validation of the application tool allows to fully accept the formulated hypothesis, proving that it is possible to migrate data between different OutSystems environments using the platform, where data integrity is never lost.

Table 5.1: Comparison of environments results by Lens Type: *Esférica*

Lens' Record at position 365		
	Source(dev)	Target(tst)
ID	32939	336658
LENSTYPEID	1	330
REPLACMENTTYPEID	1	1
SKUSONAE	32939	336658
EAN	630175476289	630175476289
COMPOSITIONS	*empty*	*nonexistent column*
ISACTIVE	0	*nonexistent column*
CREATEDBY	5156	2814
CREATEDAT	2020-10-15 10:55:29.000	2020-10-15 10:55:29.000
UPDATEDBY	5156	336658
UPDATEDAT	2020-12-18 14:44:34.000	2020-12-18 14:44:34.000
POWER	3.25	3.25
CYLINDERPOWER	0.00	0.00
AXIS	0	0
BRENDRADIUS	0.00	*nonexistent column*
ADDITION	*empty*	*empty*
DIAMETER	14.00	14.00
COLOR	*empty*	*nonexistent column*
VENDORTYPECODE	*empty*	*nonexistent column*
LENSCOLORID	NULL	NULL
COMPOSITION	*empty*	*empty*
CAPACITYID	6	6
CREATIONBYIMPORT	1	1
BENDRADIUS	8.70	8.70
ITEMDESCRIPTION	LC DAILIES ACP CX 90UN G	LC DAILIES ACP CX 90UN G
BRANDID	8	327
VENDORNAMEID	3	212
VPN	*empty*	*empty*

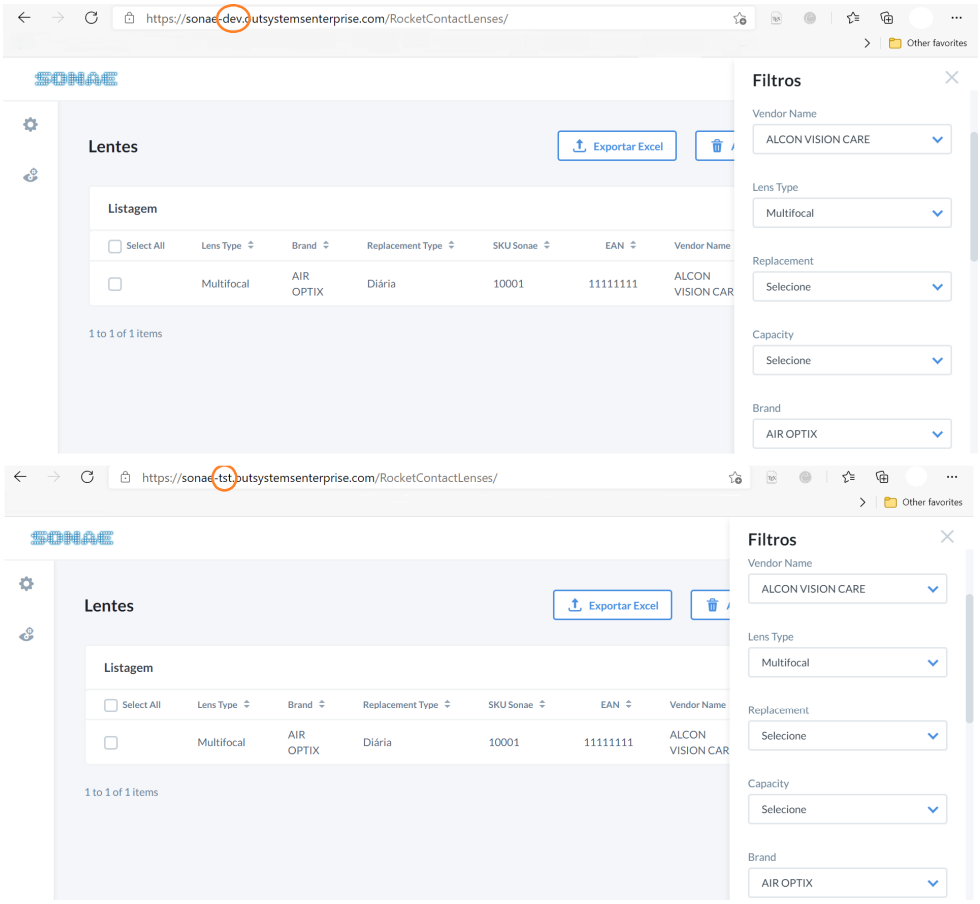


Figure 5.8: Source and Target environment behaviour and results of filter

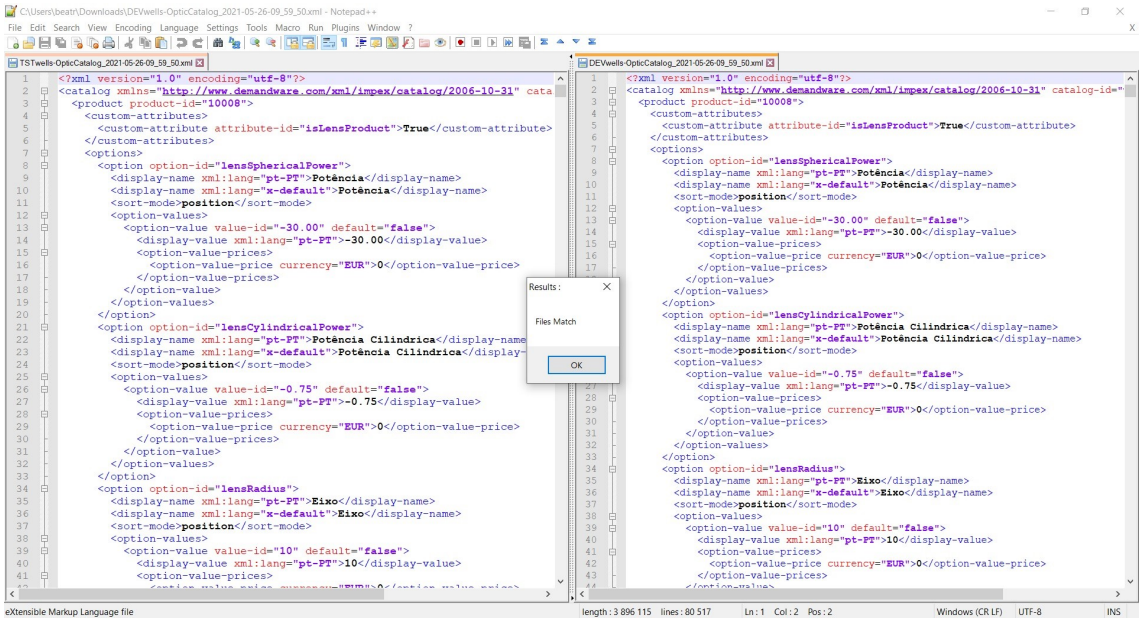


Figure 5.9: Compare generated XML from environments

Chapter 6

Conclusion

Nowadays, businesses generate ever-growing amounts of data and increasingly face urgent pressure to maximize the value they extract from it. The success of software applications depends on meeting the specified requirements and producing the desired results with faster feedback loops. Faulty software projects costs enterprises money and could potentially harm customer satisfaction, that is why quality assurance is critical for any software project, building processes for testing software and ensuring quality. Data migration is one of these processes, it is used to reject simulated data since, even if similar to reality, it rarely gives a comprehensive insight into the many ways in which customers may actually use the software and instead use the migrated data from production environments to test the applications.

OutSystems is a modern application platform designed to productively accelerate the development of applications, with faster development and deployment stages, keeping up with the changing needs of the business to drive future innovation. However, the production of quality software still requires long and complete testing phases and the fast rhythm, for which OutSystems is known, is undermined by it. OutSystems limitations include the lack of some data migration mechanism to transfer data between its environments.

This dissertation presents the design, development and use of a data migration mechanism in OutSystems, able to migrate accurate data from any environment, keeping the integrity of primary and foreign keys.

As such, a new efficient tool, BirdlyData, was created and validated. The results obtained show fast migration times with complete and correct migrated data, proving that it is possible to create a data migration framework to support testing in OutSystems' applications, using components provided by the platform itself.

6.1 Contributions

The presented solution, when in place, will enhance the testing ability of OutSystems applications by providing a full test environment setup. With BirdlyData, testers do not need to generate huge volumes of test data to ensure their increasing quality contribution in the delivery of the product for real-world use, instead, they could, in a few seconds, migrate thousands of records from other environments that contain more realistic data, like production. Then, developers could deliver software updates to customers faster and with higher quality because more bugs could be detected and solved at the testing phase. This is the result of shorter times between introducing new functionalities and the results of testing them.

6.2 Further Work

As future work, other migrations can be planned that use more realistic and complete applications of larger sizes using the mechanism implemented with the developed BirldyData tool. The interest in running more complete applications is that they could contain different types of data, like, for example, BinaryData or some unusual characters that were not tested.

Some implemented functionalities were constrained or even excluded due to the main goal of having a fast migration. Initially, the real-time migration log could provide a line-by-line update because the insert into the target environment was done record by record, but with the implemented optimizations (bulk inserts) this functionality could not offer such update details, limiting the audit log updates from x in x (constant) records. Another functionality on the backlog that was suspended was mask data using Scramble (crossing actual data), it requires mixing different columns or rows values, which would require extra processes and affect the overall performance.

It would also be interesting to add an option to define the constant (amount of records migrated) for each entity since some entities can be heavier than others. This would be a bonus functionality that is easy to implement.

The special case where the entity being migrated has foreign keys to itself (cyclic/self foreign keys), is a slower process comparing to regular entities. Further work could involve trying to improve their migration performance.

In the case of encrypted columns, each environment has its private encryption key, as a result, different environments have different encrypted strings, e.g. a source environment's (like DEV) encrypted string is different from the target's (TST) string. Migrating encrypted strings can generate runtime errors at the time of decryption. In short, there exist two approaches to solve this limitation: or this data is not migrated or, the most complex one, create an API in the target environment that encrypts data on demand, and at the time of the migration, the encryption string migrated has already the target environment encryption key.

To commercialize the tool, it would also be interesting to adapt it in order to be able to accept different databases like MySQL and Oracle.

Finally, several OutSystems senior developers have shown interest in an important feature for them: BPT Migration. Some important applications are built entirely around BPT and have many advantages, such as explicitly launch a process execution or explicitly open a task to be executed by an end-user. BPT entities are not changed by the application development and are managed by the OutSystems Platform, nonetheless, the data is populated into these by the application usage in runtime. The scope of this project did not include this functionality since it is one of the most critical topics in data migration.

References

[1] 2020's best etl tools for data management | 33rd square. https://www.33rdsquare.com/best-etl-tools/. (Accessed on 12/28/2020).

[2] Business process technology (bpt) - outsystems. https://success.outsystems.com/Support/Enterprise_Customers/Maintenance_and_Operations/Data_Migration_From_Production_to_Non-Production_Environment/Business_Process_Technology_(BPT). (Accessed on 11/09/2020).

[3] Cool data mover. https://www.coolprofs.com/cool-data-mover-en/. (Accessed on 10/12/2020).

[4] The data loading performance guide | microsoft docs. https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008/dd425070(v=sql.100)?redirectedfrom=MSDN. (Accessed on 06/03/2021).

[5] Default values on database - outsystems. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Data/Database_Reference/Default_Values_on_Database. (Accessed on 06/02/2021).

[6] Do it lean | syncit. https://www.doitlean.com/syncit#! (Accessed on 10/12/2020).

[7] General data protection regulation (gdpr) – official legal text. https://gdpr-info.eu/. (Accessed on 03/10/2021).

[8] How does aes-256 encryption work to protect your data. https://www.atpinc.com/blog/what-is-aes-256-encryption. (Accessed on 06/01/2021).

[9] Infosistema | data migration manager. https://www.infosistema.com/dmm-data-migration-manager/. (Accessed on 10/12/2020).

[10] Iri voracity - total data management. https://www.iri.com/products/voracity. (Accessed on 12/28/2020).

[11] Low code development is on the rise - rtinsights. https://www.rtinsights.com/low-code-development-is-on-the-rise/. (Accessed on 03/01/2021).

[12] Low-code: The full guide to low-code platforms and development | outsystems. https://www.outsystems.com/low-code-platforms/. (Accessed on 03/01/2021).

[13] No code/low code vs. traditional development - which team should you pick? https://marutitech.com/no-code-low-code-vs-traditional-development/#Cost. (Accessed on 03/07/2021).

[14] O que é o devops? uma explicação do devops | microsoft azure. https://azure.microsoft.com/pt-pt/overview/what-is-devops/. (Accessed on 03/07/2021).

[15] Platform runtime | evaluation guide | outsystems. https://www.outsystems.com/evaluation-guide/platform-runtime/. (Accessed on 03/01/2021).

[16] Possible setups for an outsystems infrastructure - outsystems. https://success.outsystems.com/Documentation/11/Setting_Up_OutSystems/Possible_setups_for_an_OutSystems_infrastructure. (Accessed on 03/11/2021).

[17] Top application development trends in 2019 | outsystems. https://www.outsystems.com/1/state-app-development-trends/. (Accessed on 03/01/2021).

[18] Use processes (bpt) - outsystems. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Processes_(BPT). (Accessed on 11/09/2020).

[19] What is low-code? [2021 update]. https://www.outsystems.com/blog/posts/what-is-low-code/. (Accessed on 03/01/2021).

[20] Xplenty. https://www.xplenty.com/. (Accessed on 12/28/2020).

[21] Creating maps using biztalk mapper - biztalk server | microsoft docs. https://docs.microsoft.com/en-us/biztalk/core/creating-maps-using-biztalk-mapper, 2005. (Accessed on 11/30/2020).

[22] Patrick Allaire, Justin Augat, Joe Jose, and David Merrill. Reducing costs and risks for data migrations. *Santa Clara, CA, USA*, page 31, 2010.

[23] David Aumueller, Hong Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with coma++. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 906–908, 01 2005.

[24] Philip Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *PVLDB*, 4:695–701, 08 2011.

[25] Jess Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information system migration: A brief review of problems, solutions and research issues. 06 1999.

[26] Michael L Brodie. *Migrating legacy systems: gateways, interfaces & the incremental approach*. Morgan Kaufmann Pub, 03 1995.

[27] Paulo Carreira and Helena Galhardas. Efficient development of data migration transformations. 02 2004.

[28] C. Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314–347, 08 2014.

[29] Michael de Rujiter. Comparison of outsystems data movers | by michael de ruijter | itnext. https://itnext.io/comparison-of-outsystems-data-movers-abf15afa2d6f, Oct 2019. (Accessed on 10/19/2020).

[30] Hong-Hai Do and Erhard Rahm. Chapter 53 - coma — a system for flexible combination of schema matching approaches. In Philip A. Bernstein, Yannis E. Ioannidis, Raghu Ramakrishnan, and Dimitris Papadias, editors, *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*, pages 610 – 621. Morgan Kaufmann, San Francisco, 2002.

[31] Dr Mads Haahr. Random.org. `https://www.random.org/`. (Accessed on 05/26/2021).

[32] Klaus Haller. Towards the industrialization of data migration: Concepts and patterns for standard software implementation projects. pages 63–78, 06 2009.

[33] Frank Huerta. The bug in production: What you don't know can – and will – harm you - devops.com. `https://devops.com/the-bug-in-production-what-you-dont-know-can-and-will-harm-you/`, May 2019. (Accessed on 03/07/2021).

[34] Informatica. How to achieve flexible, cost-effective scalability and performance through pushdown processing. `https://www.informatica.com/downloads/pushdown_wp_6650_web.pdf`, 11 2007. (Accessed on 12/28/2020).

[35] Zsolt Istvan, Soujanya Ponnapalli, and Vijay Chidambaram. Towards software-defined data protection: Gdpr compliance at the storage layer is within reach, 2020.

[36] Tom Laszewski and Prakash Nauduri. Chapter 5 - database schema and data migration. In Tom Laszewski and Prakash Nauduri, editors, *Migrating to the Cloud*, pages 93 – 130. Syngress, Boston, 2012.

[37] Chenyang Lu, G. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST*, 2002.

[38] Jayant Madhavan, Philip Bernstein, and Erhard Rahm. Generic schema matching with cupid. *Proc 27th VLDB Conference*, 07 2001.

[39] Tim A. Majchrzak, Tobias Jansen, and Herbert Kuchen. Efficiency evaluation of open source etl tools. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, page 287–294, New York, NY, USA, 2011. Association for Computing Machinery.

[40] Nilesh Mali and Sachin Bojewar. A survey of etl tools. *International Journal of Computer Techniques*, 2(5):20–27, 2015.

[41] F. Matthes, C. Schulz, and K. Haller. Testing and quality assurance in data migration projects. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 438–447, 2011.

[42] Renée Miller, Mauricio Hernández, Laura Haas, Ling-Ling Yan, C. Ho, Ronald Fagin, and Lucian Popa. The clio project: Managing heterogeneity. *SIGMOD Record*, 30:78–83, 03 2001.

[43] Peter Mork, Len Seligman, Arnon Rosenthal, Joel Korb, and Private Profiler. The harmony integration workbench. *J. Data Semantics*, 11:65–93, 01 2008.

[44] John Morris. *Practical Data Migration*. BCS Learning Development Ltd., Swindon, GBR, 2006.

[45] OutSystems. Architecture canvas. https://success.outsystems.com/Support/Enterprise_Customers/Maintenance_and_Operations/Designing_the_Architecture_of_Your_OutSystems_Applications/The_Architecture_Canvas. (Accessed on 19/06/2021).

[46] OutSystems. How to migrate data - outsystems. https://success.outsystems.com/Support/Enterprise_Customers/Maintenance_and_Operations/Data_Migration_From_Production_to_Non-Production_Environment/How_To_Migrate_Data. (Accessed on 12/30/2020).

[47] OutSystems. Introduction to data migration - outsystems. https://success.outsystems.com/Support/Enterprise_Customers/Maintenance_and_Operations/Data_Migration_From_Production_to_Non-Production_Environment/Introduction_to_Data_Migration. (Accessed on 12/29/2020).

[48] Priyanka Paygude and P.R. Devale. Automated data validation testing tool for data migration quality assurance. *International Journal of Modern Engineering Research*, 3:599–603, 01 2013.

[49] Eugene Philipov. Comparing multiple rows insert vs single row insert with three data load methods. https://www.red-gate.com/simple-talk/sql/performance/comparing-multiple-rows-insert-vs-single-row-insert-with-three-data-load-methods February 2013. (Accessed on 05/11/2021).

[50] ZZZ projects. bulk-insert | .net bulk operations. https://bulk-operations.net/bulk-insert. (Accessed on 05/11/2021).

[51] Val Pushkarev, Henry Neumann, Cihan Varol, and John Talburt. An overview of open source data quality tools. pages 370–376, 01 2010.

[52] Erhard Rahm and Philip Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10:334–350, 12 2001.

[53] Philip Russom. *Best Practices in Data Migration*. TDWI Monograph Series, 04 2006.

[54] Simanta Shekhar Sarmah. Data migration. *Science and Technology*, 8(1):1–10, 2018.

[55] Manjunath T N, Ravindra Hegadi, and HS Mohan. Automated data validation for data migration security. 01 2011.

[56] Bing Wu, Deirdre Lawless, J. Bisbal, R. Richardson, J. Grimson, Vincent Wade, and Declan O'Sullivan. *The Butterfly Methodology: a gateway-free approach for migrating legacy information systems*. IEEE Computer Society, 10 1997.

[57] Zohra Bellahsene, Angela Bonifati, Fabien Duchateau, and Yannis Velegrakis. *On Evaluating Schema Matching and Mapping*. Springer Publishing Company, Incorporated, 1st edition, 2011.

Appendix A

Entity Diagrams

Extended diagrams version containing the data tables connections (FK).

A.1 Extended system dependencies diagram

The connectors between tables represent the relationships among entities. The Application contains information about installed applications and the system end-user that created them. To be able to comply with the Layered Architecture pattern, is also possible to create several Modules inside an Application. When one Application Module is created OutSystems generates an corresponding Espace, where this new Module will be integrated. The App_Definition_Module entity is an intermediary table used to bind each Module to the correct Application. Since there is no direct relationship between Applications and correspondent Modules, this table is necessary to find the corresponding Espace and Entities of the Application. A Tenant is an "instance" of your application that uses isolated data. The Tenant entity has a corresponding Espace and it is used to provide the capability to reach out multiple costumers, while enforcing an effective isolation of data, configurations and end-users. For example, if the application in question is a gym management software application, set to be multi-tenant, each gym has it's own tenant, notice that all gyms use the same application but users logged in a certain gym can't see data from other gym because they have different User Providers. The Espace_Entity is also an intermediary entity to store the Entity's Espace. One Entity can only belong to a Module, i.e. one Espace and inside each Espace there could be several Entities. The Entity table stores all entities, all data tables created inside a given Application. This is the main entity consulted since it is where resides the data about a database table, it contains information such as the Name given at the OutSystems platform, the Physical SQL Table Name, if it is a system table, if it is active, between others. Both Entity and Espace entities contain an attribute called SS_KEY. The Espace always have a unique SS_Key generated and stored in the metamodel and equal in all environments. For Entity the attribute SS_Keys is not unique, for example, if the Espace is cloned. However, the combination of the Entity SS_KEY and the Espace SS_KEY make a unique composed key, that allows to identify one determined Entity between different environments. Finally, the Entity_Attr is the table that

stores every Attribute of a given Entity. This table is used to consult the Name of the columns/attributes of an Entity as well as their Type. The Type column allows to discover if the attribute is an integer (rtInteger), a data time (rtDateTime), a text (rtText), a long integer (rtLongInteger - used for entities identifiers), between others, or if it is a reference identifier type (a dependency). Figure A.1 represents the extended System Model generated by the OutSystems platform.

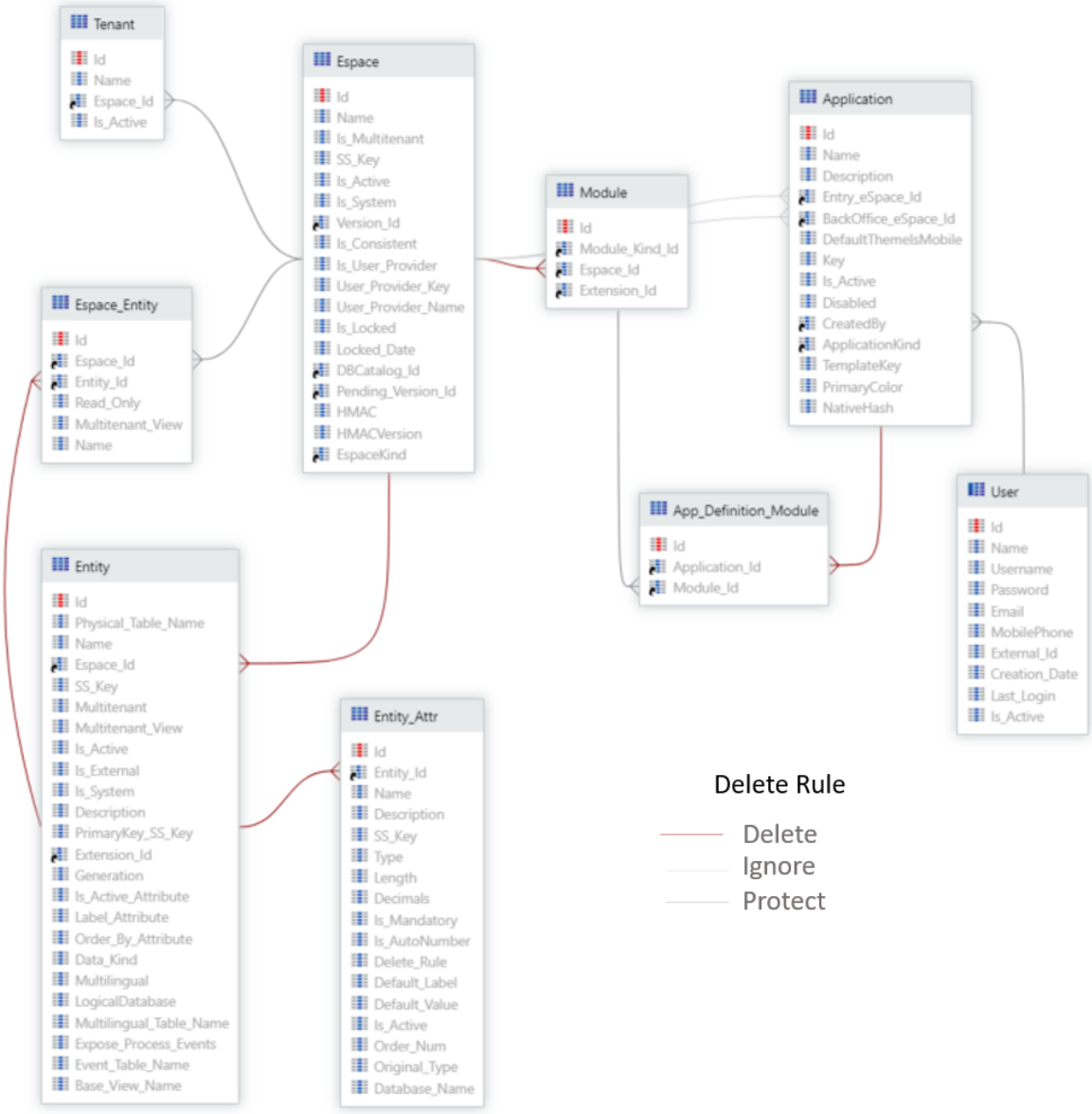


Figure A.1: System Entity Relationship Model

A.2 Extended entities dependencies diagram

Figure A.2 —Entity Model with both system tables and created data tables generated by the OutSystems platform.

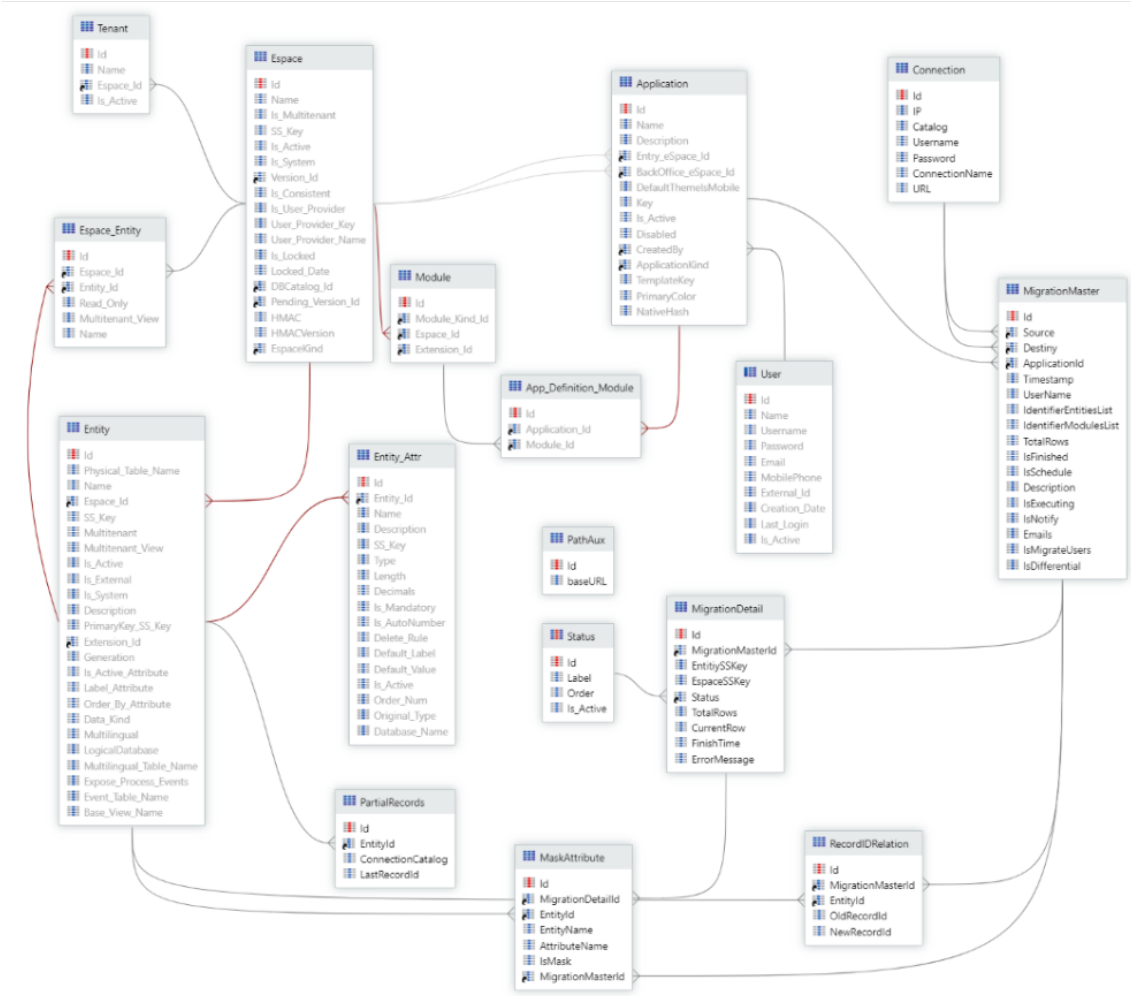


Figure A.2: Global Entity Relationship Model

Appendix B

User Interface

BirdlyData Screens display filtered database data. In particular, more information regarding the screen aggregates and data sources used to display data are presented on the next sections. The data workflow between screens is also commented, as well as inherent features of each one.

B.1 Connection Settings

A **Connection** is a set of data access configurations, that allow to connect to specific database environment. This screen, shown on Figure B.1, contains a list of created connections. Selecting one of this connections opens a more detailed view with the information of the connection access. It is also possible to create new connections by clicking on the New Button, this will redirect the user to a new screen form called New Connection screen. The form requires information about the instance of SQL Server: server/host name of the SQL server database, which is the IP address and the database name, also called *Catalog*. The name of the connection is only used to identify the connection and the URL was later added to support the migrate Users functionality. This screen also has a pre-validation functionality to test if the connection is valid before saving it, this verification is done by calling the extension. Inside the Integration Studio an action called TestConnection action is called, this receives a Connection Structure Record and allows to test that connection by simply opening it, if the connection can not be opened is because is not valid and the action returns error.

The client used for the SQL Server Authentication (User details: Username and password) must have write and read permissions since both actions will be required while the migration is taking place. Since this information represents some user credentials, the password should be encrypted, for security purposes. Even for visualizations, the password will always be encrypted.

The **source connection** is allocated to just one connection, i.e. there could only be one source environment. This connection retain the credentials to access the database of the **current environment** where the data migration tool is installed. This is one of the main reasons why the tool should be installed in all environments. If the user intends to migrate data from another source he needs to install the application tool at the respective environments and run a migration inside

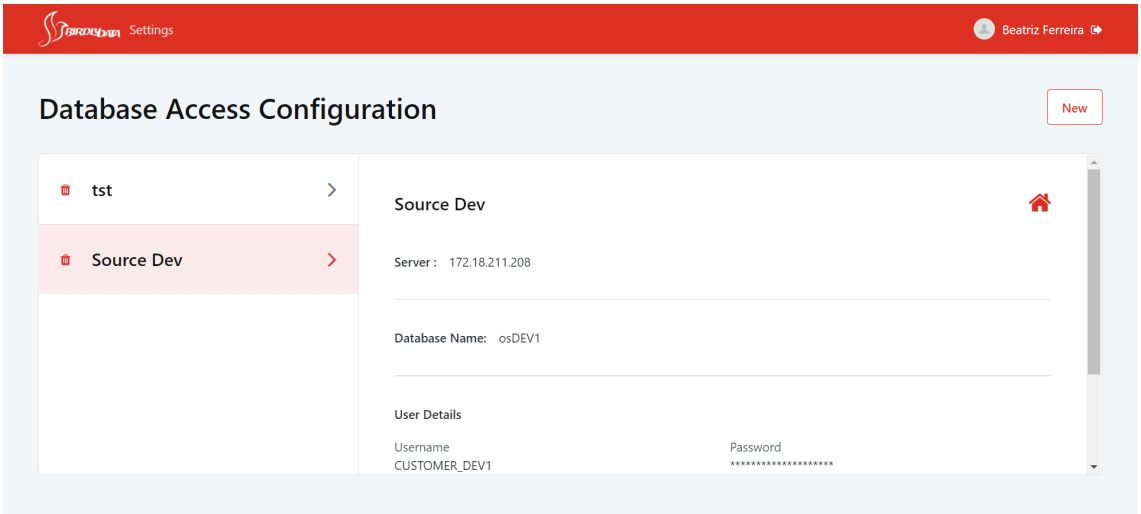


Figure B.1: Database Access connection UI

that environment, the connection of the current environment should always be added as source connection to provide the correct user credentials used for the SQL Server authentication. To migrate to other environments, one or more target connections can be created, and will be added automatically to the list of connections. To migrate to this target connections the tool should also be installed in the respective target environments to migrate end-users of the applications.

To delete a connection the tool first verify if the connection is not being used by any saved plan. If is being used by a plan, then it is mandatory to first delete the respective plan before deleting the connection.

B.2 Migration Dashboard — Home

From this screen is possible to navigate to other screens such as new migration screen, access the result of finished migrations in the Migration Log Screen or the migration plan overview screen for scheduled migrations (selecting the info icon of a migration plan), and go to the database access configurations screen.

This screen, represented on Figure B.2, displays only finished migrations and future (schedule) migrations. It is set as a home screen since it allows to visualize and manage all information regarding last migrations and scheduled ones, as well as create new migrations plans or environment connections, being able to preform all functionalities when redirected to other screens. The data displayed for the finished migrations is fetched from the entity MigrationMaster, filtered by true IsFinished values. The same logic is done for the scheduled migrations but with two attributes being filtered: the IsSchedule set to true and the IsFinished values set to false.

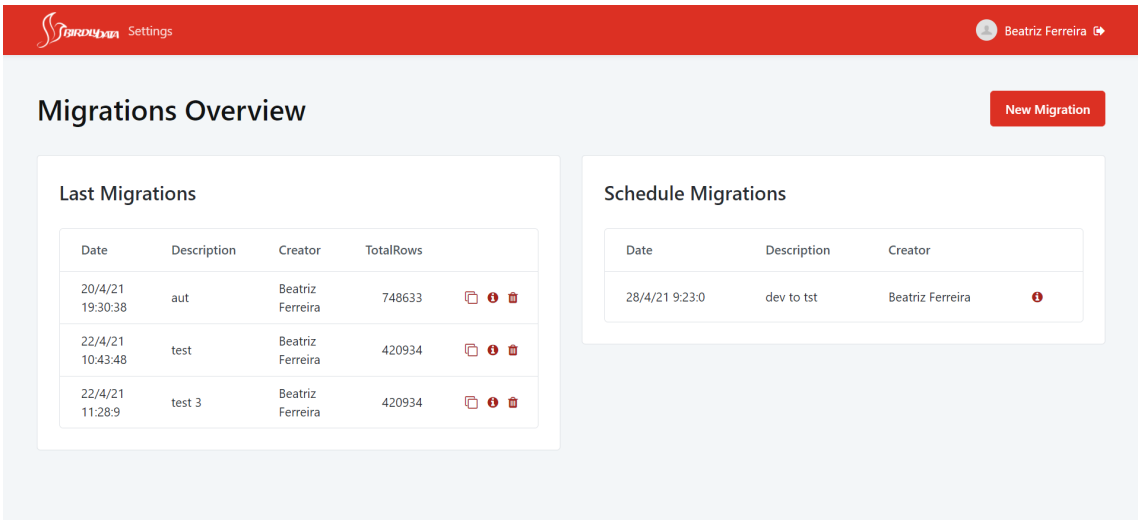


Figure B.2: Migrations Dashboard UI

B.3 New Migration

From the Dashboard screen is possible to navigate to the New Migration screen, represented on Figure B.3. It is a form screen, which display content for a specific single record plan. It contains a set of boolean values used as flags to trigger several functionalities. The required fields are the plan description, that allows to better identify the plan, and the source and target environment. This attributes alone are able to perform a simple and successful migration.

Some database's data is displayed into the screen form, such as target connections and the source connection. The target connections are fetched through an aggregate and filtered by the attribute Fixed (determines if it is the source) of the connection with negative values. The source connection has the filter on Fixed to allow only true values. The source environment widget is not enable or editable, which allocates the source connection to the current environment and prevents further errors.

By default the boolean flag responsible to trigger the differential migration functionality is set to True, since migrating just the latest records will have less impact on the migration performance. However, activating the switch of Empty Target option will disable the differential migration switch, since they are opposite actions, where the first one saves the last records migrated and the last deletes them.

Regarding the Migrate Users option, even if the migrate Users switch is set to True, it is only actually triggered if the system table User is added to the entities list to be migrated, since only referenced User records should be migrated. If it is set to false, then even if the User entity is referenced, the entity records will not be migrated.

Both scheduled migrations and notifications functionalities require an extra attribute, the start date and the emails address, respectively. Scheduling a migrations allows the user to pick a date from the calendar and the respective hour he intends to run the migration, this data will be saved

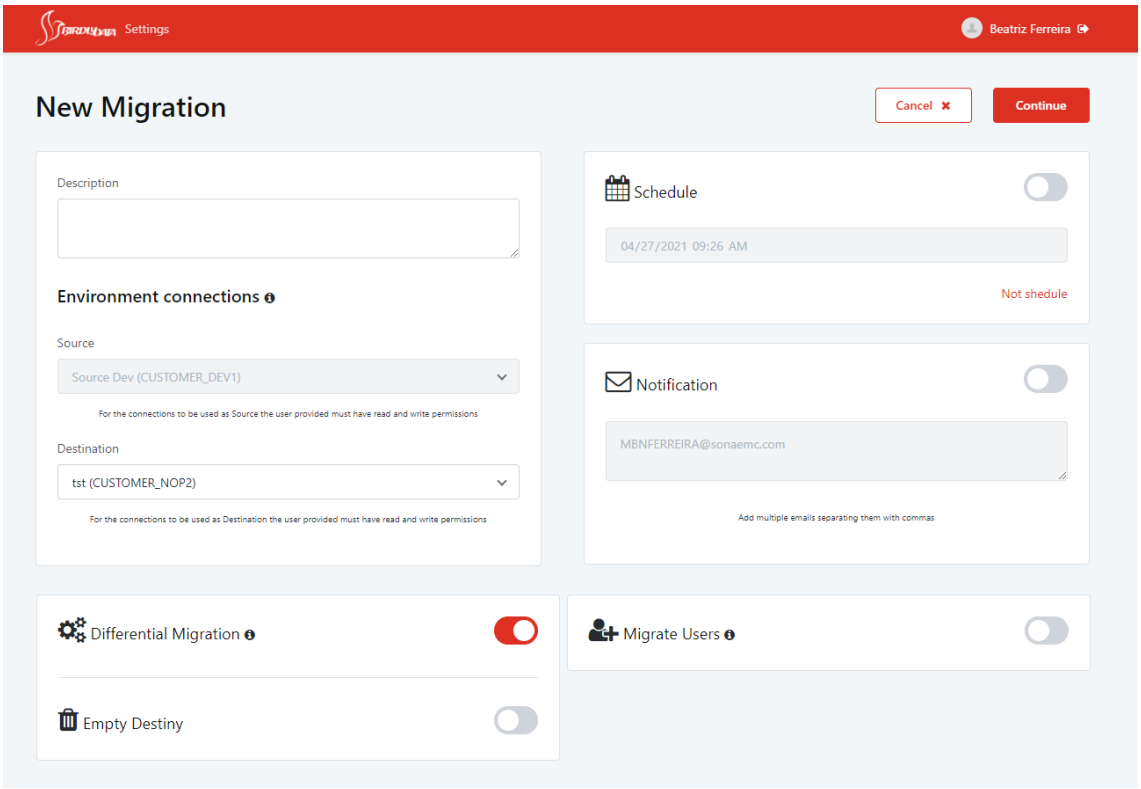


Figure B.3: New Migration UI

as a DateTime local variable. To receive a notifications update after a migration is finished then the user needs to input the emails into a text field, separated by commas in case of more than one email address. This auxiliary inputs are disabled until the user enables the switch that triggers the functionalities.

When the user decides to continue with the migration then a new plan is created, this plan is filled with the local variables provided by the user (a MigrationMaster record is created), such as description, source and target environments and the extra migration functionalities flags. However the plan configurations is still incomplete until the user navigates to the next couple of screens.

B.4 Application List

The Application List Screen, represented on Figure B.4, fetches data from the Application entity. However, not all applications are displayed since some of them don't even contain entities. The applications should have at least one module with entities. Applications that are not active and do not contain any active module with active entities should not be displayed. With this purpose, the aggregate joins the Application, App_Definition_Module, Module, Espace and Entity entities and filters them by: active applications, eSpaces and entities. Other filter is the Entity attribute Data_Kind being "entity", since other types of entities are not intended to be migrated, like static entities. Additionally, system applications are also ignored.

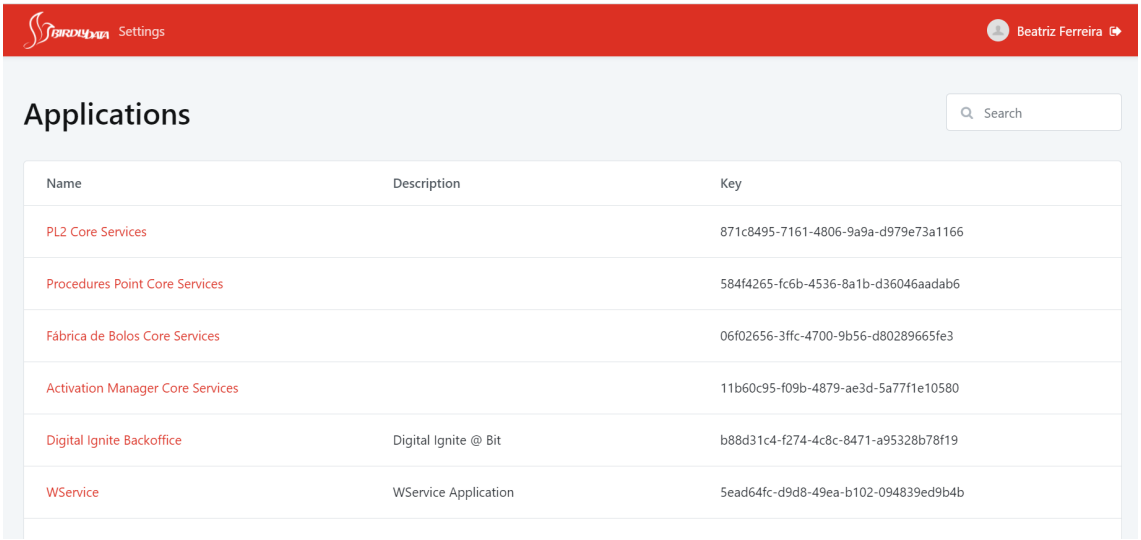


Figure B.4: Applications List UI

Since the amount of applications could be very extended, the screen automatically provides table pagination and search bar. To proceed with the workflow the user has to click on the application name desired to be migrated. This means that it is only possible to migrate one application at a migration plan and to migrate another application entities, then another migration plan should be created.

B.5 Modules List

The Modules List Screen, represented on Figure B.5, is similar to the Application List Screen. The screen fetches the chosen application name to display it at the header with the application identifier. For the modules list then the information fetched by the aggregate follows the same logic as the aggregate used in the Application List Screen. It fetches the active modules with active entities using the application identifier. This modules are then added to a local list called EspacesList that corresponds to a structure with a small portion of the attributes and a new boolean attribute: IsSelected.

To select the respective modules then the user has to click on the checkbox of the respective module, that sets the IsSelected attribute of that record to True.

The user should select at least one module. When the user has his set of modules selected then the list is iterated and checked records identifiers are added to a text variable, separated by commas. The reason behind this text variable is because is not possible to pass lists as input to another screen in OutSystems.

Finally, the screen is redirected to the Entities List Screen.

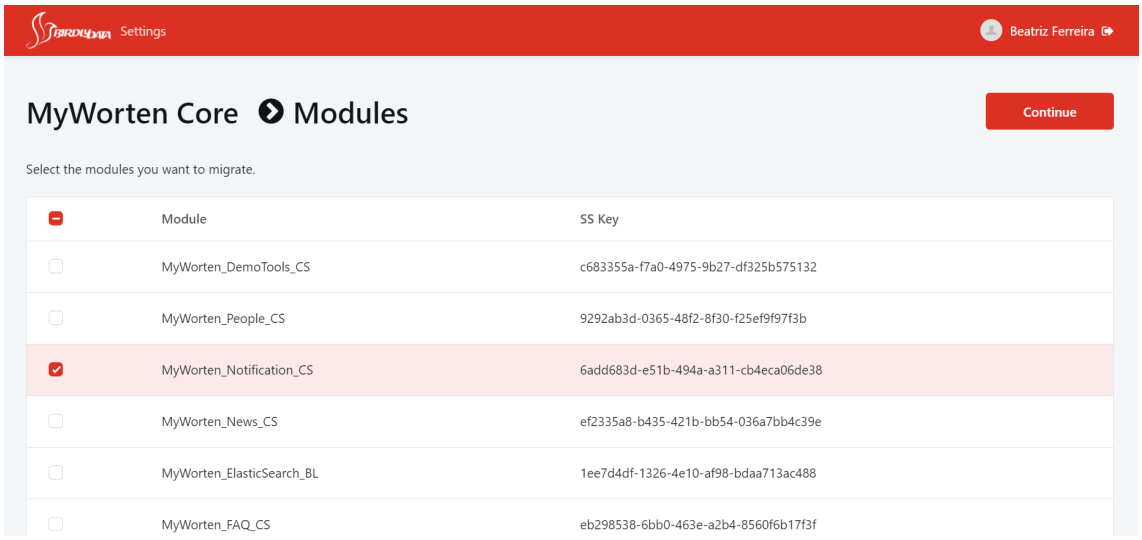


Figure B.5: Modules List UI

B.6 Entities List

At the Entities List screen the user can select which entities he wants to migrate and define which entities's attributes should be mask. This screen is exemplified on Figure B.6.

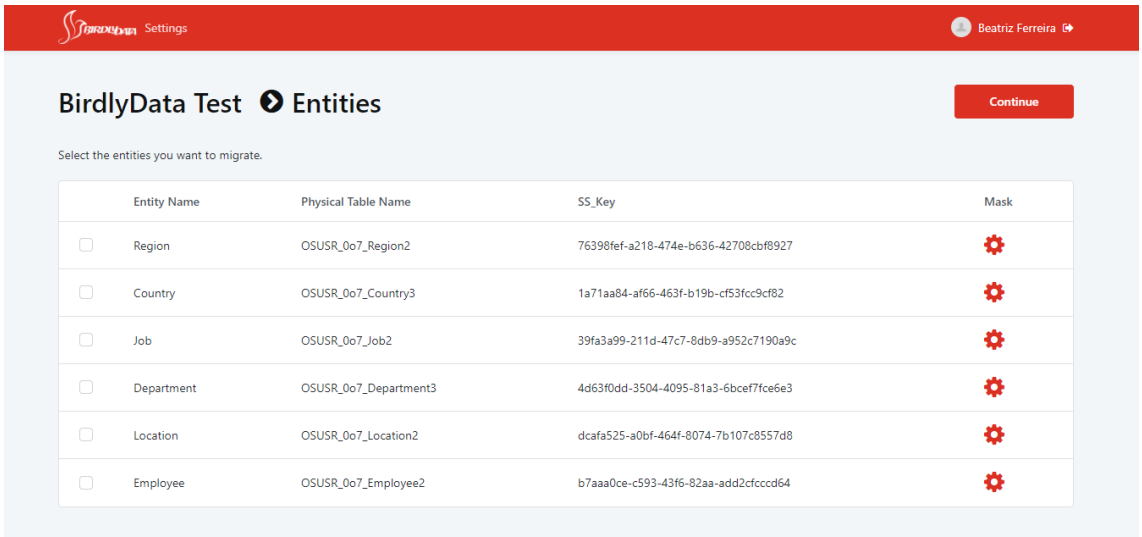


Figure B.6: Entities List UI

To fetch the entities list it was necessary to build a query that searches for entities inside the chosen module/eSpaces. The eSpaces identifiers list already obtained is used as parameter to the follow query B.1. This query joins the entities Espace and Entity and returns all active entities created inside those eSpaces. Additionally, and since the IN operator is used, the Expand Inline property of the parameter was also set to yes. In OutSystems the Expand Inline property of the

attribute has to be set to yes to allow to implement part of the query at runtime.

```
1 SELECT {Entity}.* FROM {Entity} , {Espace}
2 WHERE  {Entity}.[Espace_Id]={Espace}.[Id]
3 AND  {Entity}.[Is_Active] = 1 AND  {Entity}.[Data_Kind] LIKE 'entity'
4 AND {Espace}.[Id] IN (@EspaceListIDs)
```

Listing B.1: Get Entities list screen

After fetching the entities, a local list structure (EntitiesListSelection) is filled with the relevant information about them. Inside the client action, GetEntitiesOnAfterFetch, a list of attributes, for each entity, is also fetched and added to a new Mask object, a list of AttributeMask structures. The information displayed is the entity's name, it's physical table name and SS_KEY. The reason behind this decision was to represent an entity by it's unique attributes. The physical table name corresponds to the entity name at the SQL level, and it is also used used for the migration at the extension level. The SS_KEY is an attribute that identifies the entity cross-cut environments.

Selecting one entity with dependencies selects and disables the checkbox of the direct and indirect dependencies, as shown on figure B.7. In this example, the entity Employee directly depends of the entities User, Employee, Department and Job. But the entity Department depends of Location and this depends of Country. In turn, the entity Country depends of Region. This means that Employee will also depend, indirectly, of Location, Country and Region. That is the reason why all entities of the Figure are selected and disabled except Employee. Dependencies that correspond to entities of the list are marked as selected and the checkbox widget is disabled, not allowing the user to edit this value. If the dependencies are not on the entities list then they are added, such as the system entity User.

Disabling the checkbox widgets allows to prevent errors while migrating and maintain the FK integrity by mandatory migrate all related entities. Deselecting one entity, in this case Employee, will revert the process. The checkbox widget triggers an OnChange event, every time it is selected, this event uses one client action to affect the current row style and check if it is a select or a deselect so it can proceed with a different logic for both cases. Selecting an entity will call the client action GetEntityDependencies, responsible to find dependencies and provide the logic to select them, while deselecting it will call DeleteEntityDependencies, responsible to remove the dependencies from the display screen.

For the attributes list, represented on Figure B.8, another data action was created, called GetEntityAttributesByEntityId. The query uses one entity identifier to fetch the attributes names by joining the tables: Entity_Attr and Entity. It also excludes attributes that can not be masked like primary keys (PK) and foreign keys (FKs). The attributes' names and a boolean flag are added to a compose mask structure list.

When the user selects the mask icon then a new event onclick is triggered and is handled by the client action: AttrDetailOpenPopUp. This actions opens a new pop with the local list of attributes by assign the selected entity identifier to a local/auxiliary variable called EntityId.

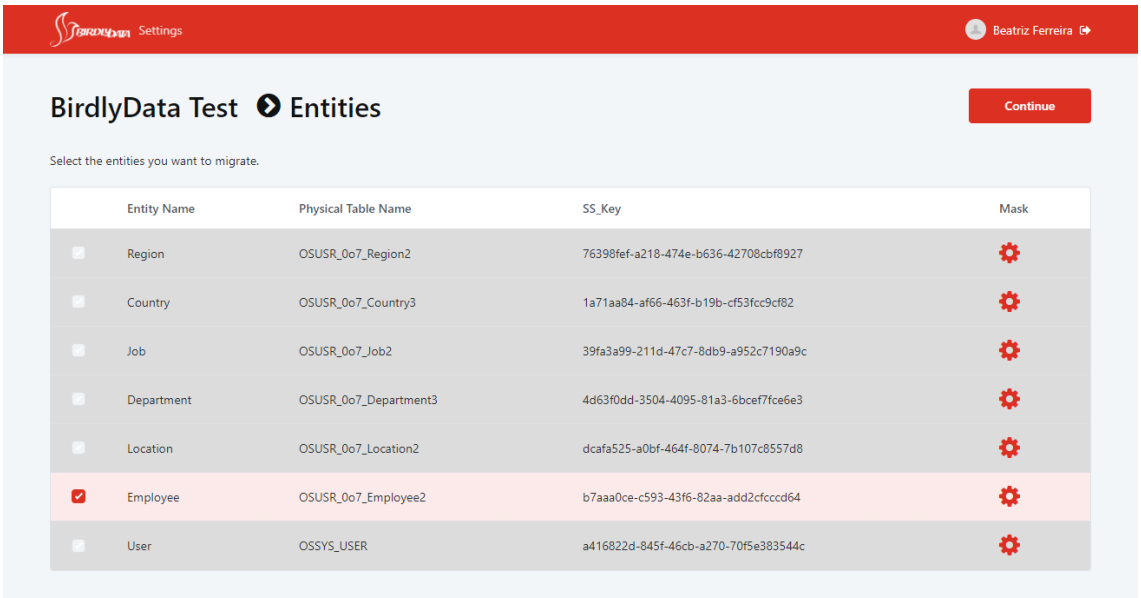


Figure B.7: Automatic dependency detection UI

This variable stores the current entity selected and it is used to fetch the attributes relative to that entity. However, the entity identifier alone is not enough to fetch the correct data, so another local variable was created, this one stores the position of the selected entity inside the attribute list (mask structure list). So the expression that fetches the attribute name look like this: *AttributeMask[PositionMask].AttributeNameMask.Current.AttributeName*, where the AttributeMask is the name of the local structure that stores the attributes names and mask flags and the PostionMask is the local variable that indicates the position of the selected entity into the AttributeMask list. After set the mask and press the Ok Button then the pop up is closed.

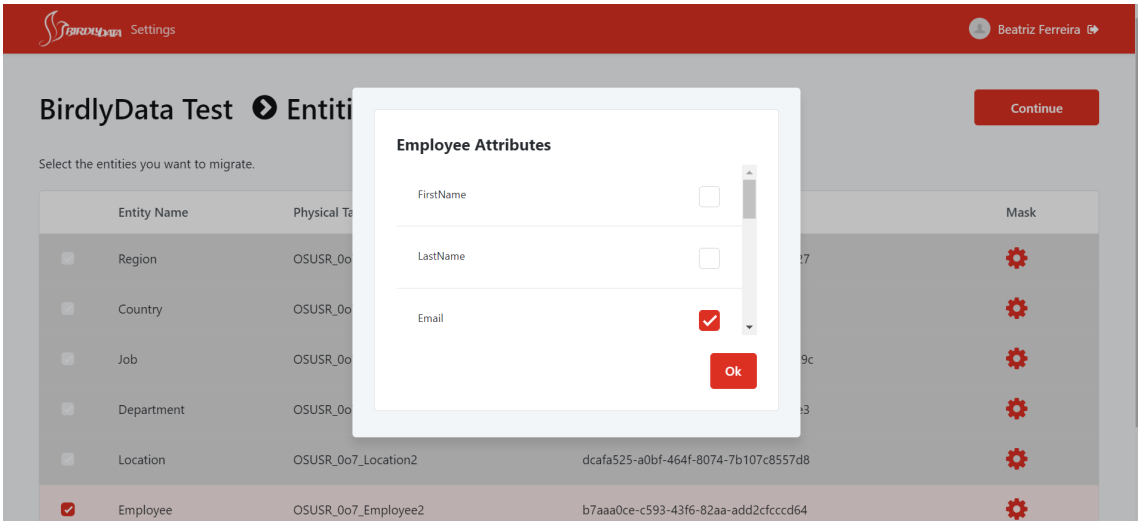


Figure B.8: Attribute list mask UI

When the user decides to continue then the selected entities are ordered by dependencies and converted to a text list separated by commas, the same way as the modules list. After, the plan is updated with the values of the remaining attributes of MigrationMaster record plan. At this step, both MigrationDetail records and MaskAttribute are created, every MaskAttribute record corresponding to a particular MigrationDetail record. The MigrationDetail entity contains information about a particular entity of the migration plans, including it's Espace and Entity SS_KEYS that provide a composed unique identifier through the all OutSystems platform. Additionally, the MaskAttribute entity has information regarding every column of an entity, it stores the attributes, also called columns or headers of the table/entity, where each record represent a column of the entity and it's masks setting.

B.7 Plan overview

The migration plan overview screen is for consultation only, since contains all information regarding a migration plan. An representation of the screen is presented on Figure B.9.

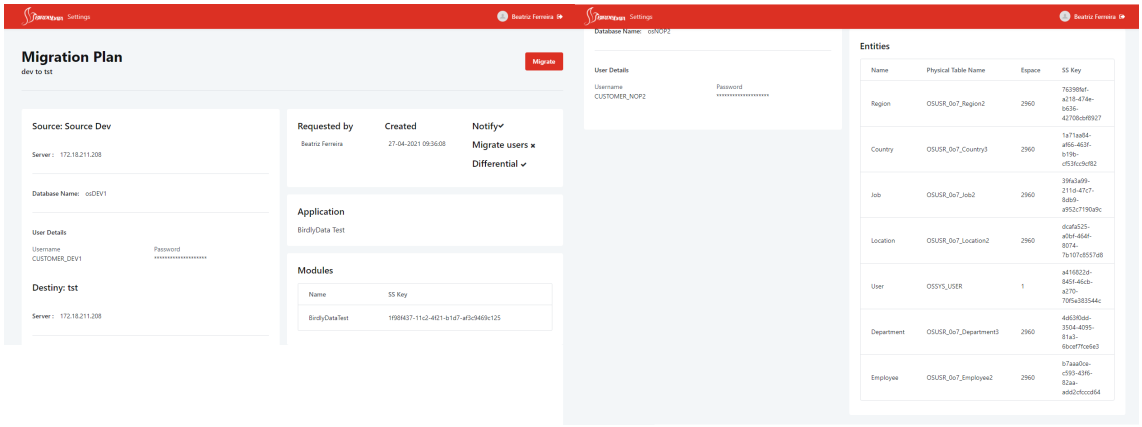


Figure B.9: Migration Plan UI

This screen fetches both source and target Connection data, information about the migration flags, the selected application, modules and entities. Both modules and entities are fetched from a text with the records identifiers separated by commas. This implies fetching the data with a SQL data action. For the modules, a simple IN operator was used at the IdentifierModulesList attribute of the MigrationMaster, this operator allows to specify multiple values in a WHERE clause.

The text attribute that stores the entities list has the records identifiers already ordered by dependencies. However, the IN operator alone could not provide a ordered output. To force this order and display the entities list properly, then two parameters were added to the SQL, one with the Expand inline property set to yes and another set to no. Both parameters should receive exactly the same text value and the query should look like the follow query B.2. The ORDER BY clause forces the output order to be the same as in the IdentifierEntitiesList text attribute of the MigrationMaster entity, otherwise the entities would be displayed randomly.

```
SELECT {Entity}.* FROM {Entity}  WHERE  {Entity}.[Id] IN (@EntitiesListIDs)
ORDER  BY CHARINDEX(CAST(ID AS VARCHAR),  @EntitiesListIDs2  );
```

Listing B.2: Workaround SQL to retrieve ordered entities

B.8 Migration Log

For not schedule migrations, the Plan Overview screen will redirect to the Migration Log screen as represented on Figure B.10. This screen provides an audit report about the migration process and it's status.

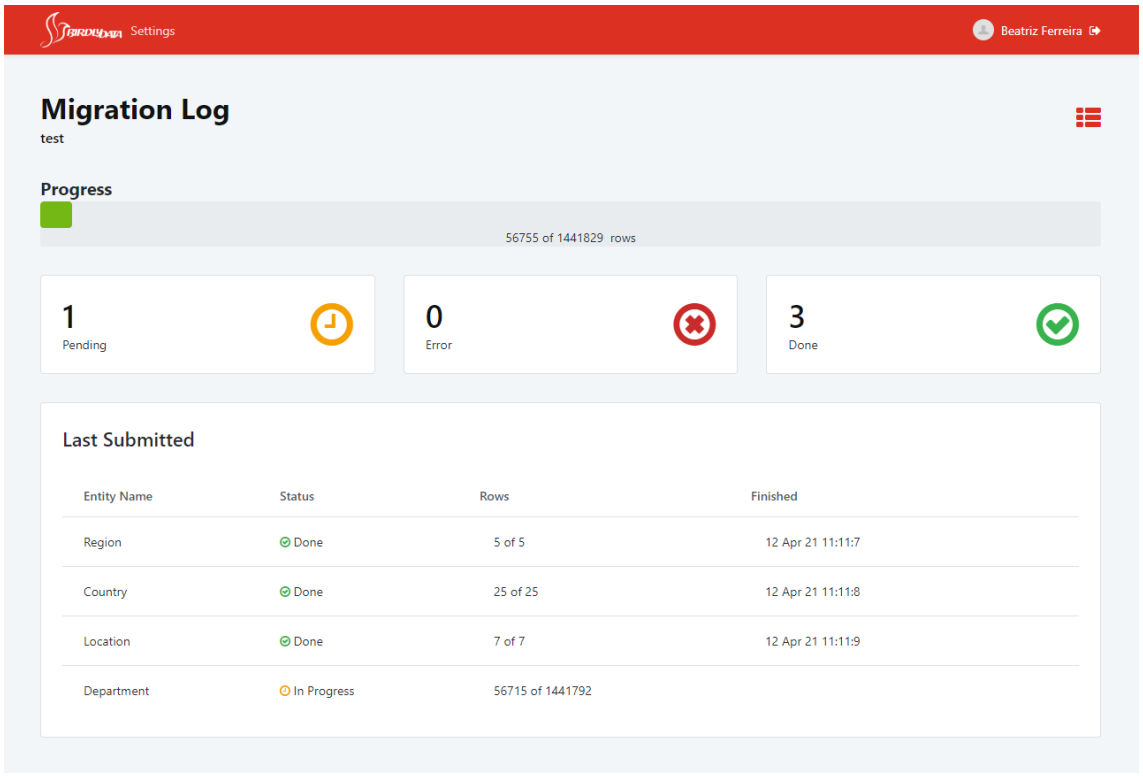


Figure B.10: Real time Migration Log UI

After fetching (OnAfterFetch event) the information of the migration plan, using the MigrationMaster, MigrationDetail and Connection entities, then the logic is implemented. This action runs when the Screen is ready, i.e. after it's first render. First, it uses the Crypt_Lib actions to decrypt both source and target password to be used by the extension. The extension action (Migrate) uses a list of entity and eSpace SS_KEYS to globally identify one entity, in any environment. To support this, a new structure was created with both entity SS_KEY and eSpace SS_KEY, each record of this list structure represents an entity. On the after fetch action, the list of entities selected are iterated and the SS_KEY values added to the structure.

If the migration has a flag trigger to migrate Users and the table was indeed found on the entities list then an action to create Users on target is called to migrate referenced ones. After that, the Migrate action of the extension is called with the fetched data and the list of SS_KEYs.

The screen retrieves multiple data and it is divided in three sections, which are:

- The progress bar with the overall migration progress, that includes the number of records of all entities combined
- Entity status count: progress, error and done
- Entity level status, record progress (row-by-row), entity migration finish time and error details

One aggregate gets the migration details to extract the table - entity name, status, rows (current and total) and finish time. Some counting were also added to get how many entities exist at a given state: done, in progress and error entities.

The progress bar expression uses a global CurrentRow value as well as the TotalRows of the whole process to complete the expression: *sum(MigrationDetail.[CurrentRow]) / MigrationMaster.TotalRows * 100*. This will retrieve a value inside a range of 0% to 100%. The overall progress consults the aggregate that joins the MigrationMaster with the MigrationDetails, Espace, Entity and Status (static entity) to fetch the TotalRows attribute of the MigrationMaster entity and uses a data action to fetch the sum all CurrentRow attribute of the MigrationDetails, summing all current rows of all entities - current row of the whole process.

Finally, for the Last Submitted data table, the same aggregate of the progress bar was used. It is able to fetch the attribute name and physical_table_name of the Entity, the finish time of the MigrationMaster and possible error message (in case of an entity failure). For the format *x* out of *y* rows, where *x* is the current row being migrated and *y* is the total row size of the table, both attributes CurrentRow and TotalRows of the MigrationDetail entity were consulted, respectively.

In OutSystems, when the data displayed is modified by external sources it does not automatically detects changes so it does not refresh it's widgets. Nonetheless, this refresh is of most importance for a real time operation. The workaround to solve this issue was based on a creation off a hidden button (not displayed to the users) and a javaScript, represented on Figure B.11 that simulates the click on the button every second. This button will then call a client action created just to refresh the aggregates and data actions necessary for the log view.

Figure B.11 shows the script used to update the displayed data at real-time (every second), in MigrationLog Screen.

This screen can also be used to consult the progress of a finished migration, for this, a local boolean variable called IsConsultOnly was created, and at the OnAfterFetch action it is verified, if it is just a consult view then the data is only fetched and displayed with no more logic associated.



Figure B.11: JavaScript to update MigrationLog screen

Appendix C

Data Access Configuration

C.1 Test Connection

Before creating a new Connection, on the UI, a validation is made to check if the connection is valid or not. This was not possible using solely the OutSystems Service Studio platform since OutSystems only has access to the current environment's data. Inside the Integration Studio, on the extension module, a new Structure called Connection was created, where only the essential attributes to the creation of a ConnectionString were added, such as IP (DataSource), *Catalog* (database name), Username (user identifier) and Password. The created action, called TestConnection, receives an Connection structure record as input and retrieves a boolean as result, and in case of error it also retrieves an error message.

After compiled, the action is generated in the source code as a function. There are several ways to check if the ConnectionString is valid, the method used was simple but effective: checking if the connection is able to be opened, otherwise catching the SQL Exception as error.

C.2 Encryption algorithm

The Advanced Encryption Standard (AES) is the first and only cipher approved for protecting top secret information.

Encryption works by taking plain text, in this case a password, and converting it into cipher text, made up of apparently random characters. One special key is required to decrypt it.

The AES algorithm takes plain text in blocks of 128 bits and converts them to ciphertext using keys of 128, 192, and 256 bits. AES-256 has a key length of 256 bits, so it supports the largest bit size. It is practically unbreakable, for this reason is considered secure and the most robust and worldwide standard for encryption methods.

AES is a symmetric key encryption algorithm, that uses only one secret key to cipher and decipher information.

The illustration C.1 shows how symmetric key encryption works.

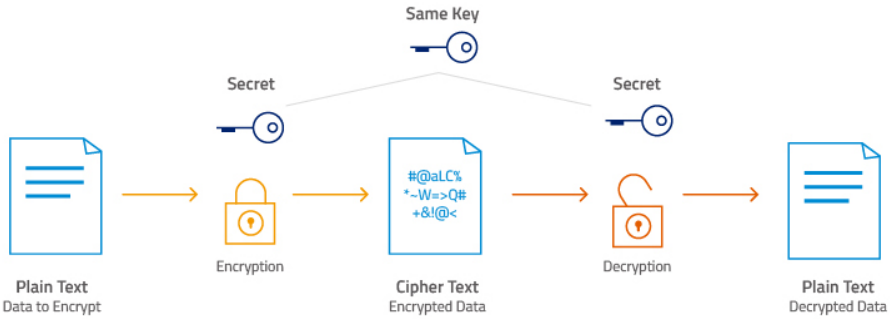


Figure C.1: Symmetric key encryption [8]

The AES-256 algorithm used in the BirdlyData app contains two actions: one that allows to encrypt any plaintext and convert it to a cyperthext, and another to decrypt the ciperthext to the original plaintext.

Before creating the connection (before saving it to the database), the password is encrypted using the Encrypt server action. This allows to mask the sensitive data at the database level. Then it is decrypted using the Decrypt server action of the same module, when sent to the extension to generate the ConnectionString properly.

Appendix D

Entity Dependencies

Migrating data without any references to other entities is simple, but is not a regular case. The most common entities in databases have **Foreign Keys** (field in one table, that refers to the PRIMARY KEY in another table) to other entities, called **dependencies**. A dependency is a constraint that defines the relationship between two entities. For example, a direct dependency considering two entities could be between: Employee and Department. The Employee entity has an attribute DepartmentId, wich is a FK to entity Department. This means that Employee depends directly on the table Department, and Department is an Employee dependency. The Employee is related to one Department, and if some information regarding the Department changes, the Employee will be affected since it uses the Department information.

D.1 Dependency detection

The Type attribute (mentioned on Section 4.4.2) informs if it is a dependency but not which one (only presents the unique identifier of the entity - SS_KEYs), this information has to be retrieved with a query. For the process of finding some entity's dependencies it is necessary to find out what entity correspond to each FK columns through an advanced SELECT query in OutSystems such as the one listed in D.1. This is possible by decomposing the string to get both SS_KEY values and use these parameters to search for the entity information, for all attributes of the selected entity (query's input). Since the espace SS_KEY is between "**bt**" and "*" and entity SS_KEY is presented on the right side of "*", the query separates the Type attribute by these characters. The query's output retrieves all **direct** dependencies and represents the base of the dependencies detection mechanism.

```
1 SELECT  {Entity}.* FROM  {Entity}, {Espace}
2 WHERE  {Entity}.[ESPACE_ID] = {Espace}.[ID]
3 AND @EntitySSKey  = (select right({Entity_Attr}.[Type], charindex('*', reverse({
    Entity_Attr}.[Type])) - 1))
```

```
4 AND @EspaceSSKey = (select SUBSTRING({Entity_Attr}.[Type], CHARINDEX('bt', {
      Entity_Attr}.[Type]) + 2, CHARINDEX('*', {Entity_Attr}.[Type]) - CHARINDEX('bt'
      ,{Entity_Attr}.[Type]) - 2))
5 AND {Entity}.[Data_Kind] = 'entity' AND {Entity}.[IS_ACTIVE] = 1
```

Listing D.1: OutSystems advanced query: find entity dependencies

The RIGHT() function extracts a set of characters from a string (starting from a given index) and it is defined as: RIGHT(string, number_of_chars), where <number_of_chars> is the index. In different records the <number_of_chars> can change, to prevent errors, this was made dynamically, using the operator CHARINDEX(expressionToFind , expressionToSearch). This function allows to search for the "*" symbol (expressionToFind) and the function REVERSE(stringExpression) gets the right part of the string without the symbol (expressionToSearch).

The SUBSTRING() function extracts some characters from a string, in this case, the wanted substring is between "bt" and "*" that corresponds to the espace SS_Key. The operator definition is: SUBSTRING(string, start, length) where the start is the start position and the length is the number of characters to extract. The start value is replaced with a CHARINDEX operator that jumps the first two characters "bt", being the length value another CHARINDEX that counts how many characters exist from the third position until the "*" symbol.

The data kind is set to "entity" to exclude static entities and others that are not important to the migration process, the is active attribute is set to 1 because only active entities have value.

D.2 Automatic dependency detection - Recursive

At this stage and with the aim of migrating the data successfully, one pre-validation feature was implemented to detect and select entity dependencies. This validation allow to treat the data before being consumed by the extension. In the background, when the developer selects one entity to be migrated, the tool searches for all dependencies of the selected entity (using Listing D.1 recursively) and automatically adds them to the entities list. These added dependencies will be already **selected** and **disabled**. The fact that they are disabled is because allowing the user to deselect those entities could have several problems at the migration level, since the migration of the selected entity would not be successfully without also migrating their dependencies (data loss, break relationships or lack of data integrity).

Nonetheless, detect direct dependencies is not enough to the migration process since this entities will also be migrated, which means that it is necessary to search again the dependencies of those. Given the same example as before, considering that an Employee entity depends on another entity called Department and Department also depends on (has a FK to) Location, this means that Employee will also depend on Location, as an **indirect** dependency.

The solution found to solve this problem was creating a **recursive** action, calling the SQL query D.1 that retrieves the dependencies of a given entity. First the SQL would retrieve it's direct dependencies, i.e. Department (DepartmentId), Job (JobId), Employee(ManagerId) and

User(UserId), represented by the FK columns on Figure 4.5. Those entities will be added to the DependenciesList, which is going to be iterated, and the action will be called again, for every record in this list. When the dependencies list reaches the Department entity then the SQL call will retrieve the Location entity, repeating this process until the SQL retrieves an empty list and the dependencies list finish iterating.

The checkbox responsible to select the entities, on EntitiesList Screen, implements an OnChange event client action called OnChangeBulkRow to manage the selects and deselects of an entity. This action uses the SetSelectedTableRow action to complement Bulk Actions Logic, it uses the row number and the entity ID to select the current row. However, the mechanism should detect if it is a select or a deselect, i.e. if the entity was already selected or not. For that type of information it is necessary to store a boolean value IsSelected inside each record of the EntitiesList, for this reason and subsequent steps, the EntitiesList data type was changed to a Structure. The structure has all relevant information about the entity itself as well as another Structure attribute called Dependency, as presented on Figure D.1. The Dependency object of each entity has the attributes IsDependency, IsAddedDependency and a list of entity identifiers. Additionally, DependenciesList was built as a structure, composed by the entity identifier, name and a attribute called Dependencies that is composed by the data type: entity list.

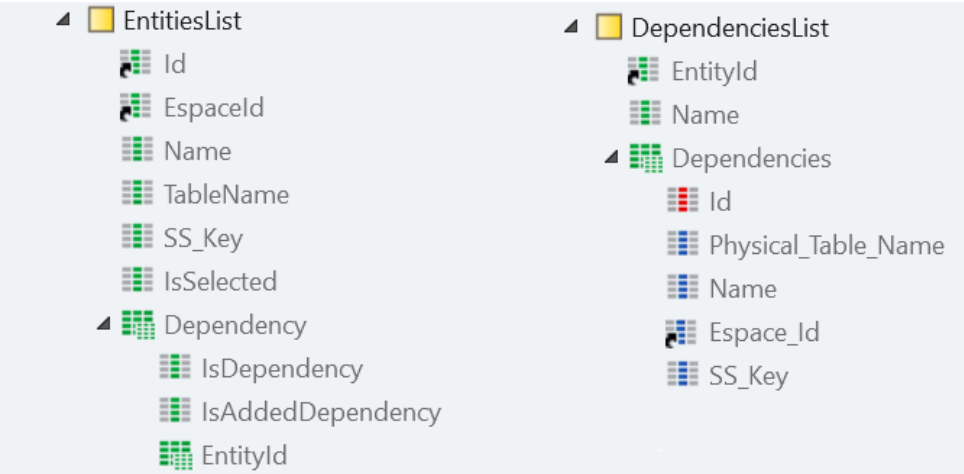


Figure D.1: Auxiliary Structures to manage list dependencies

When an entity is selected, the software calls the GetEntityDependencies client action that searches for all direct and indirect dependencies, if there exist dependencies, then the entity selected identifier and name are saved into the DependenciesList. The list of dependencies found is added to the Dependencies attribute of the DependenciesList, allowing to create an object that works like a DTO - Data Transfer Object which maps the selected entity with it's own list of entity dependencies.

This would be enough if the developer does not deselect any previously selected entity or if all the dependencies were already on the list, which can probably not be the case. The algorithm takes care of this specific cases by iterating the list of entity dependencies retrieved by the SQL

and check if the current entity ID already exists on the EntitiesList. This is where the attributes IsAddedDependency and IsDependency have utility. The problem is that the original entities list only has the entities of the selected application modules, but an entity could depend of other entities that belong to other modules. So, when this happens, the dependency is added to EntitiesList and set the IsAddedDependency to true, which indicates that this entity was added to the list and was not there originally. The IsSelected and IsDependency attributes of the EntitiesList are also set to true and the selected entity identifier is added to the list of EntityId. Otherwise, if the dependency already exists on the EntitiesList the algorithm just sets the IsDependency flag and IsSelected to true and the IsAddedDependency attribute to false, also adding the selected entity identifier to it's list of Entity IDs.

Both Figure D.2 and Figure D.3 represent possible values of the DependenciesList and EntitiesList structures, respectively. The lists examples are associated to the selection of the entity Employee with the identifier number 10149.

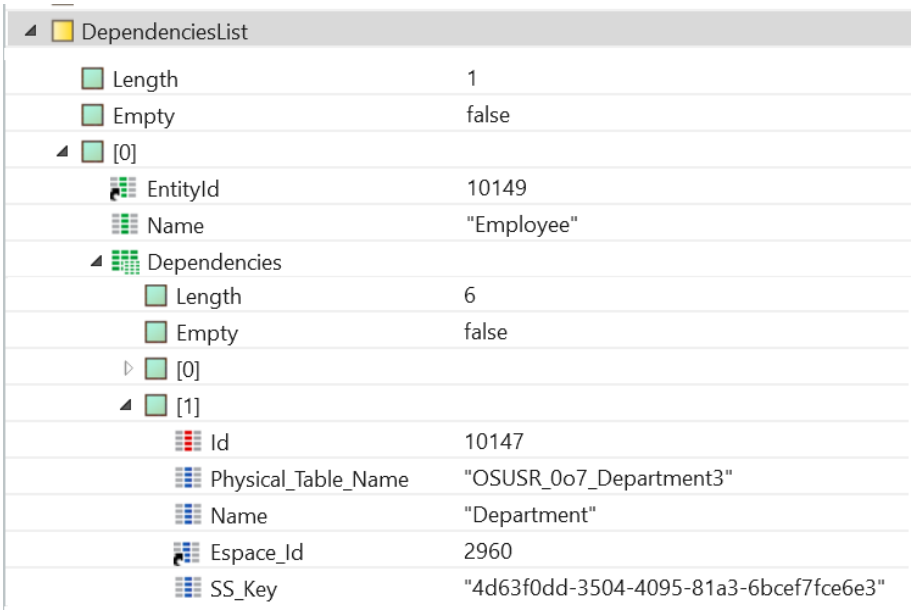


Figure D.2: Dependency List Example

Figure D.4 represents the OutSystems server action GetAllEntityDependencies that uses an entity ID, espace ID and an entity list and retrieves the dependencies as an entity list. This server action calls itself recursively, and it is first called by the client action responsible to get the selected entity's direct and indirect dependencies, represented on Figure D.5.

Figure D.6, on other hand, shows the client action that is responsible to the logic of deselecting an entity from the Entities List Screen. It uses the select entity identifier as input.

The specific case of the developer deselecting an entity is also treated with this auxiliary lists. First, the algorithm checks if the IsSelected attribute was already true, if yes then the DeleteEntityDependencies Client Action is called. This action checks their dependencies list, for each entity, if the dependency list has the selected entity ID (input-deselected entity) then check if it's the

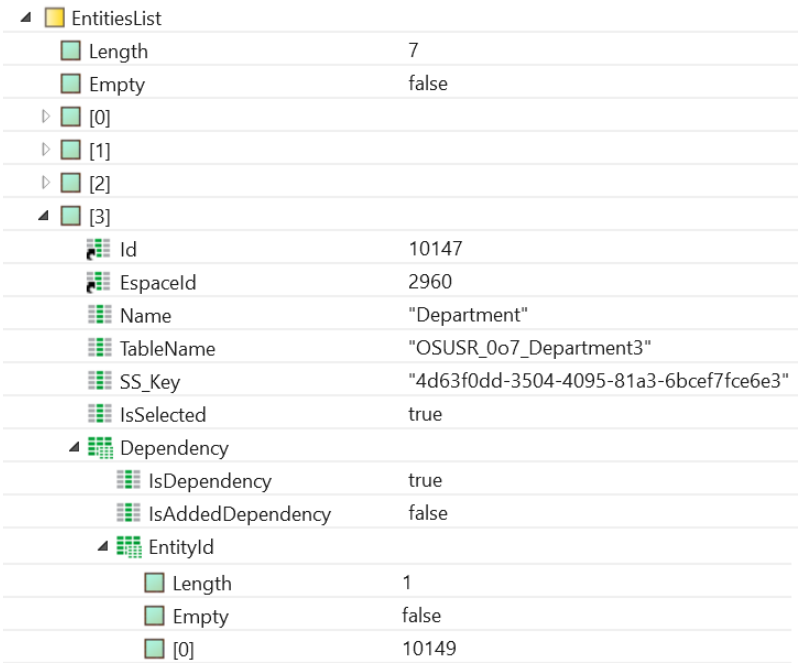


Figure D.3: Entities List Example

only dependency. Being the only dependency allows to delete it, but if it is not the only one, just removes the entity ID of the list (remove the dependency FK). Additionally, the attribute IsAddedDependency is checked and in case of true the software deletes the entities that were added to the original EntitiesList, otherwise just deselects them.

When deselecting entities a persistent bug appeared inside the DeleteEntityDependencies action's logic. The bug revealed the special case of deselecting one entity that depends of other already selected. For example, select the entity Location and then select the entity Department (that depends of Location). This bug did not allow the developer to deselect the entity Location, since the IsSelected attribute was still true. If the entity the developer want to deselect is on the DependenciesList is because it was automatically selected so he is not allowed to deselect it. This was solved by searching the deselected entity identifier at the DependenciesList, if found then set the attribute IsDependency to false, otherwise sets both IsDependency and IsSelected to false.

D.3 Dependencies order

Another form of validation to be performed before the migration is to ensure the dependencies list are ordered by dependencies. Ordering the list allows to keep the FK integrity by guarantee that entities with dependencies are only migrated after their dependencies. While inserting an entity with dependencies at the target environment, the identifier of the record should be already defined (inserted), for the attribute/column that identifies a dependency. Otherwise, the relationship

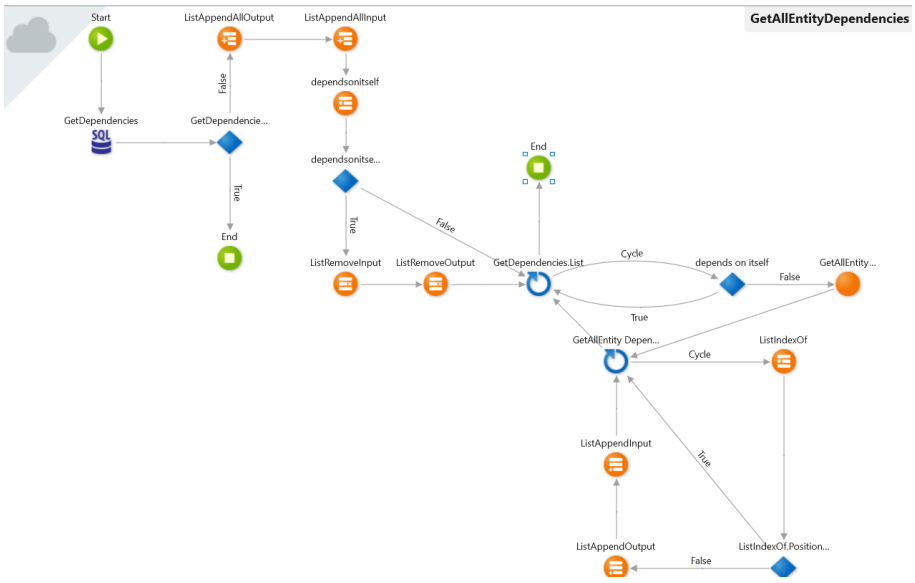


Figure D.4: GetAllEntityDependencies server action

between tables can be broken. Therefore, all entity dependencies should be migrated before the entity that depends on them.

When developers (tool users) finish selecting which entities they want to migrate and click on the Continue button, the EntitiesList is iterated and the records with the attribute IsSelected set to true are added to a list of selected entities. This list is used by a Server action, EntityOrderList, that receives the list of entities and retrieves the same entity list but now ordered by their dependencies.

The algorithm created to solve this issue, first adds the entities without dependencies to the ordered list and then verifies which entities depend only on the entities already included in the list and adds them, and so on until all selected entities are present on the output list. Inside the action, some local auxiliary variables were created such as AddEntity, of type boolean and EntitiesToRemove, of type entity list. The mechanism implemented iterates the entity list received as input and use the query to fetch the dependencies (D.1) of each entity. This is done inside a server action called GetEntityDependencies created to get the entity's dependencies, it checks if the result is empty, if not, then possible duplicated entities are removed and the flag AddEntity is set to true.

If the list without duplicates is empty is because it doesn't have dependencies. In this case the entity is added to the output list (goes first on the list because doesn't depend on others). Then the mechanism adds the current entity to the EntitiesToRemove list and continue iterating. Otherwise, if the list after the duplicates is not empty, it is iterated and, for each dependency, check if all entities(dependencies) are already on the output list, if they are then nothing is done since the AddEntity is already set to true, but if it is found at least one dependency that is not on the output list then the AddEntity is set to false. After iterating the dependencies, the algorithm verifies the AddEntity flag, if true (was found) it adds the current entity to the output list and to the list of EntitiesToRemove. The mechanism continue iterating until the input list finish. Finally,

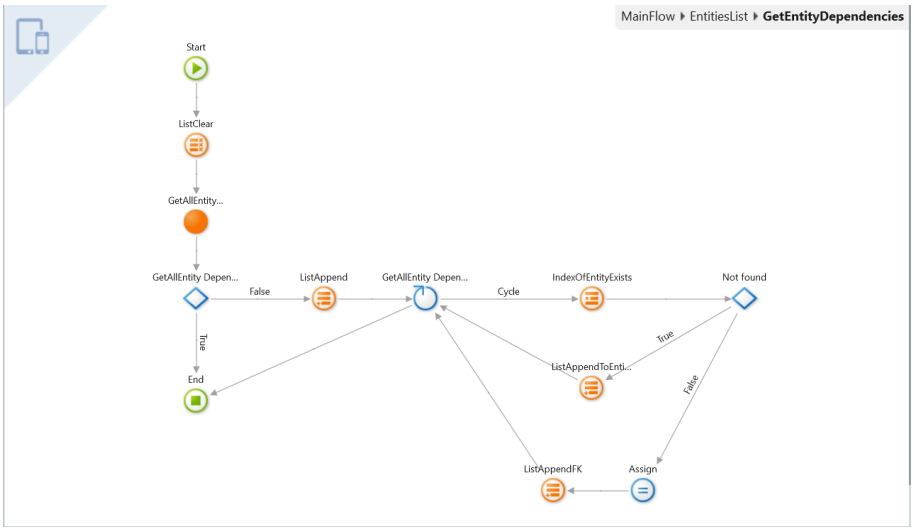


Figure D.5: GetEntityDependencies client action

all entities on EntitiesToRemove list are eliminated from the input list (this entities could only be deleted after the cycle ends, since deleting records of a list while it is being iterated is not feasible in OutSystems), if this input is still not empty it iterates it again until it's empty.

The algorithm's implementation in OutSystems is displayed in Figure D.7 and D.8, that represent the server action EntityOrderList.

The client action that calls the server action EntityOrderList then iterates the output, inserting each entity ID into a text separated by comma. This is used to update the ordered entities on the MigrationMaster table and create the MigrationDetail for each entity. After this step, the developer does not have to provide any more information about the migration, and a display Screen called MigrationPlanOverview is invoked with all the migration plan data. This Screen presents, between other essential data, the entities list. To fetch the entities by order another advanced query B.2 was used in this screen.

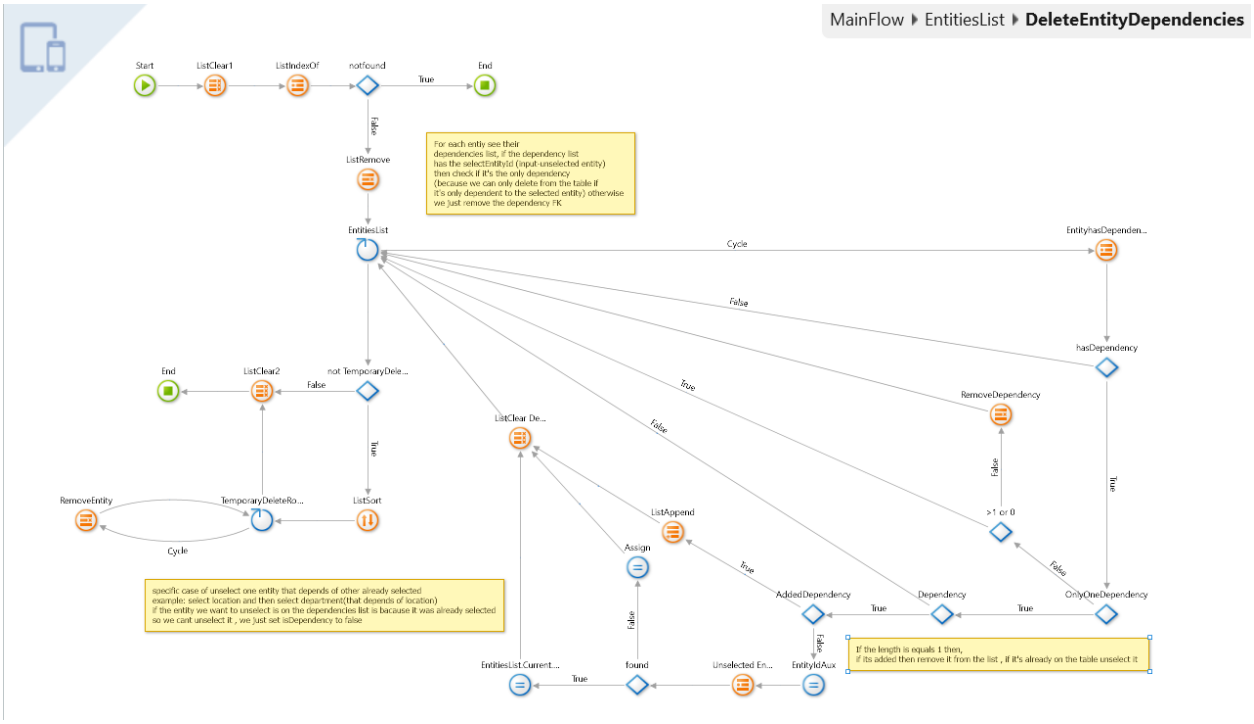


Figure D.6: DeleteEntityDependencies client action

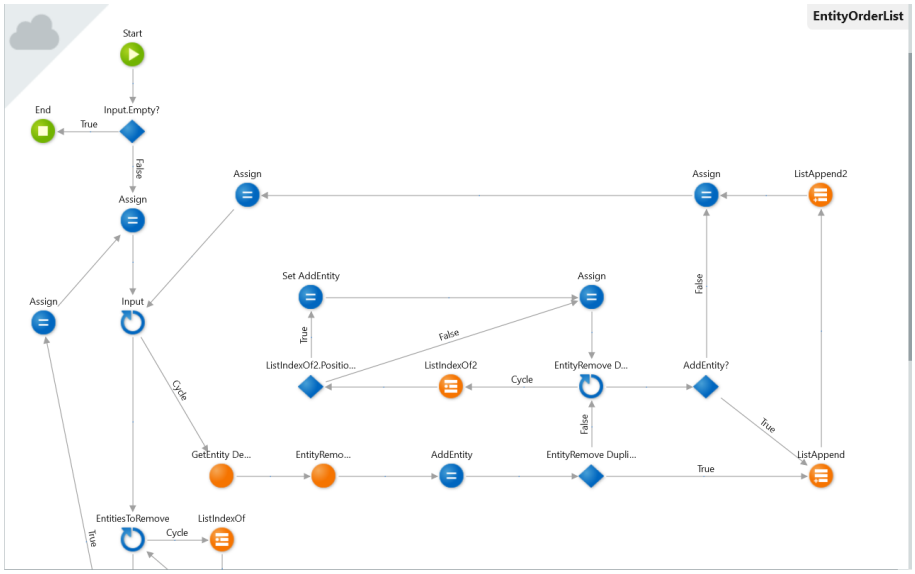


Figure D.7: EntityOrderList action in OutSystems part:1

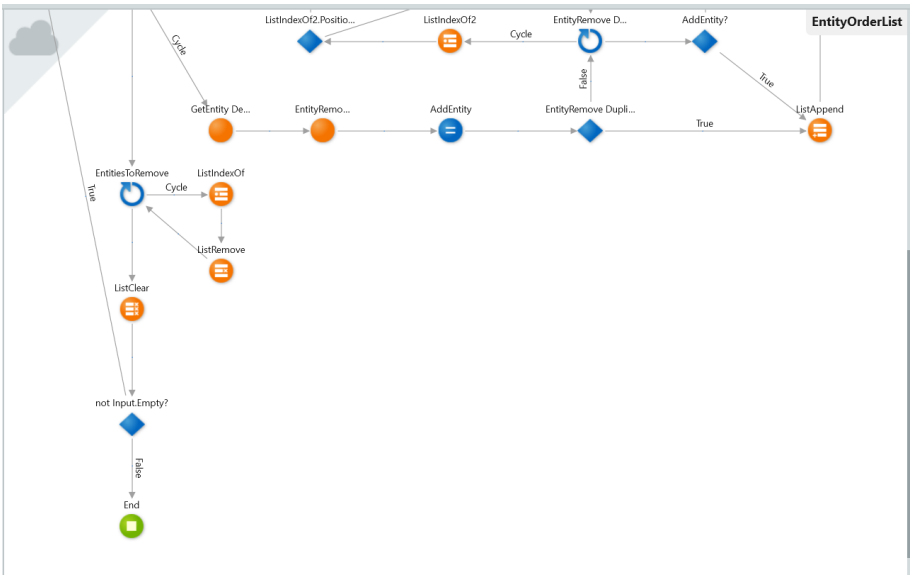


Figure D.8: EntityOrderList action in OutSystems part:2

Appendix E

Migration through an OutSystems extension

E.1 Integration Studio Structures

Integration Studio uses a set of structures and actions to connect with the Visual Studio and the OutSystems' Service Studio.

The MigrationConfig structure is presented on Figure E.1. This structure has the Migration-Master identifier (long integer) as the StagingId, both source and target connection records (also used to test connection) and the username as a text parameter called User. The boolean parameter IsMigrateUser is used to verify if the developer pretends to migrate the User system table and the IsDifferential flag to check if the developer pretends a full migration or just migrate the new records inserted since the last migration. Finally, isDeleteDestiny is the flag responsible to the deletion of the target environments tables.

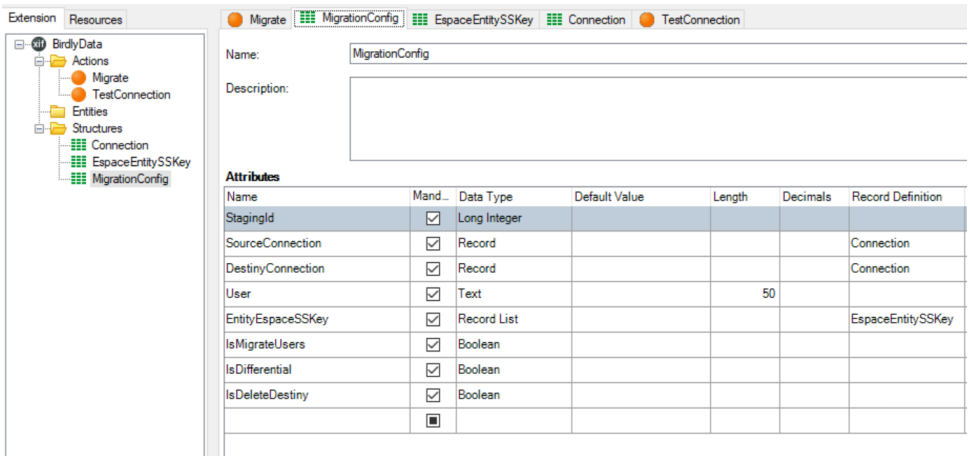


Figure E.1: Integration Studio Migration Configuration Structure

Another structure called EspaceEntitySSKeys contains two text parameters for each SS_KEY

value. This structure was added as a record parameter to the MigrationConfig structure to allow to identify entities. The EspaceEntitySSKeys structure is presented on Figure E.2.

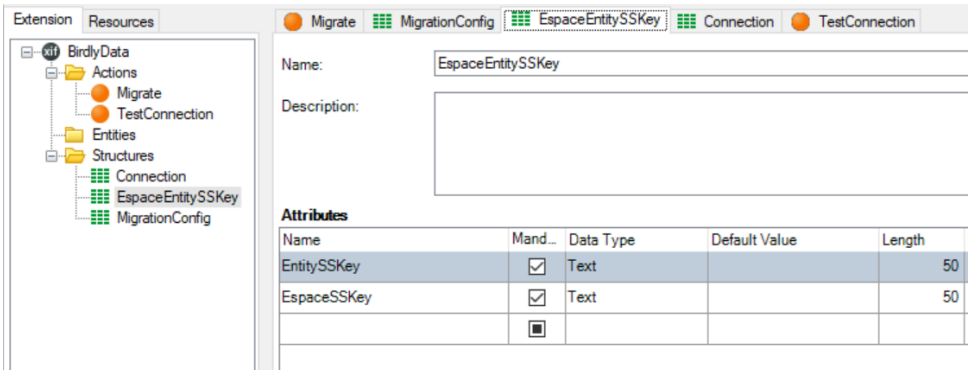


Figure E.2: Integration Studio Espace and Entity SSKeys Structure

The Connection structure is presented on Figure E.3. This structure is used for both TestConnection and Migrate actions.

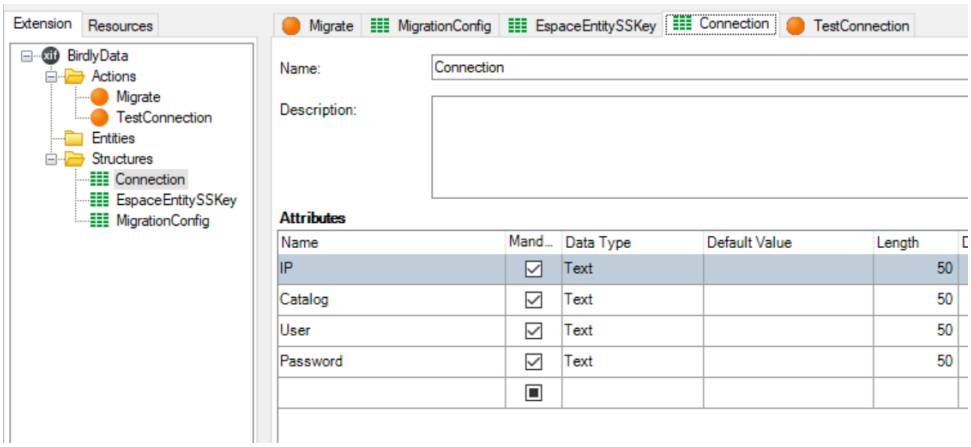


Figure E.3: Integration Studio Connection Structure

Inside Visual Studio solution, the solution is built with several generated classes. The Records.cs and Structure.cs are the classes which have the generated code of the structures created in the Integration Studio. For example, the structure STMigrationConfigStructure represents the Integration Studio's structure and RCMigrationConfigRecord corresponds to a record of this structure.

E.2 Get Physical Table Name

Different schemas may have naming and structural differences as already mentioned on the Data Mapping Section 2.5. In OutSystems, is common to have name dissimilarities between different environments. An entity can be modified in the target environment, changing it's physical

table name version, for example, the entity called "OSUSR_0o7_Employee" can be called "OSUSR_0o7_Employee1" at the target environment. OutSystems manage changes by adding an auto incremented number after the physical table name. Even if the implemented change is renaming the entity, the name is still the same on the database but with an incremented number of the new version. This could be a problem since, at source, there is no information about the correspondent table name of the target and the same entity can have a different identifier in different environments. The entity attribute SS_KEY is the only entity attribute that is the same in both environments. Unfortunately, when duplicating an application, the SS_KEY attribute of their entities is also duplicated.

The solution found to this problem was joining the SS_KEY attributes of both Entity and Espace data tables. This composite key with the Entity SS_KEY and Espace SS_KEY is the only unique identifier throughout all system.

The function GetPhysicalTableName, listed at E.1, was created with the aim to find the physical table name of an entity. GetPhysicalTableName receives a connection and the composite key of SS_KEYS and retrieves the correct physical table name of that environment. This function is called for both source and target environments, saving them on a string SourcePhysicalTableName and TargetPhysicalTableName respectively. The entity's physical table name will be essential to the SELECT as well as the INSERT query for the migration.

```
1 SqlCommand cmd = new SqlCommand(
2 "SELECT PHYSICAL_TABLE_NAME FROM ossys_entity e, ossys_Espace s " +
3 "WHERE  e.ESPACE_ID=s.ID " +
4 "AND e.SS_KEY = '" + entityss_key + "' " +
5 "AND s.SS_KEY = '" + espacess_key + "'", sqlConnection);
```

Listing E.1: Get Physical Table Names

E.3 Dynamic Insert

To be able to parse any given table with different columns and data types the insert query should be abstract. There are three elements necessary to build the insert query:

- Physical table name
- Table columns or headers/entity attributes names
- Values of the record to insert

The INSERT INTO statement is used to insert new records in a table. The syntax E.2 used specifies both the column names and the values to be inserted, this ensures that the values match their respective columns.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);

-- Example:
--INSERT INTO OSUSR_0o7_Employee(name,jobId...) VALUES('Jon',1,...)
```

Listing E.2: Insert Syntax

There is another INSERT syntax that was considered. This syntax does not specify the columns names in the SQL query and it is used when adding values for all the columns of the table. However, it is necessary to ensure the order of the values is in the same order as the columns in the table, what makes this last option to be rejected since different environments can have different structural schemas, binding the attributes to the respective values ensures the order is correct and guarantees that no error occurs.

- **Headers**

  To find the table headers names is also necessary to know how many columns exist. The number of columns is used to iterate the list of columns and add the values to the string query (the insert statement).

  The function GetColumnNames E.3 uses the source physical table name to fetch its columns names and join the ordered columns in a string, returning a string in the format: "(column1, column2, column3, ...)". It consults the Information_schema, SQL Server database table, that stores information regarding the schema of data tables, like it's columns for example. With *information_schema.column* it is possible to find information about the columns of a given entity using the physical table. The variable *headers* store the complete string and the list stores each column retrieved with the SELECT. After adding each column name to the list, then the string is build, aggregating all results (separated by commas) and composing the string with the parenthesis. However, the identifier of the record (ID column) is ignored because it can not be migrated because OutSystems does not allow us to specify an ID, it is auto incremented. The solution ignores this column, since that, when inserting, a new ID is going to be assigned to that record. The function GetColumnCount is similar to GetColumnNames, using COUNT() to retrieve the number of columns with the same except for the column "ID".

```
string headers = "(";
SqlCommand cmdHeaders = new SqlCommand("SELECT COLUMN_NAME FROM
    INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = '" + tableName + "';",
    sqlConnection);
SqlDataReader reader = cmdHeaders.ExecuteReader();
var list = new List<string>();
while (reader.Read())
{
```

```
7       if (!reader.GetString(0).Equals("ID")) {
8          list.Add(reader.GetString(0));
9       }
10  }
11  headers += list.Aggregate((prev, cur) => prev + ", " + cur);
12  headers += ")";
```

Listing E.3: Get column names for dynamic insert

- **Values**

  The values to be inserted into the target are provided by the select query which gets data from the source with a SqlCommand E.4, and uses the source physical table name already stored at string variable SourcePhysicalTableName. To iterate trough the records a SQL-DataReader (way of reading a forward-only stream of rows from a SQL Server database) was used to execute the command. Each iteration corresponds to a record of the source table (while reader.Read()).

```
1  cmd = new SqlCommand("SELECT * FROM " + SourcePhysicalTableName, source);
2  SqlDataReader reader = cmd.ExecuteReader();
```

Listing E.4: Get Source Data

With the number of columns, already discovered, it is possible to iterate the headers' list and check if the column's name is "ID", if it is then it's value is ignored (not added to the string query). If the value of the reader is empty it should be added to the list as "NULL". If the value is not empty and neither corresponds to the value of the column "ID" then it's data type has to be analyzed. Strings and dates have special treatment since they require an apostrophe, for example 'Jon' or '1900-01-01 00:00:00'. Other data types like integers and decimals do not need any special treatment since they will just be added to the string as they are retrieved. The verification is done by comparing the column type with "nvarchar", "datetime" and "varchar", that are the most common data types for data that requires an apostrophe on the SQL query. The values corresponding to columns with this data types will be added between apostrophes to the list. After all records of one entity are read then the list of values is aggregated on a string as done to the headers string.

Dependencies, also known as Foreign Keys, need to be treated separately. Before iterating the headers a new function GetTableDependencies is called, this function uses a SQL query similar to the one already done before to get the dependencies in OutSystems. It returns the list of entity identifiers that are dependent of the current entity being migrated. If the output list of GetTableDependencies is empty it means that the current entity is not dependent of any entity and is ready to be migrated. Otherwise, if the output list contains any IDs then

it is because the current entity is dependent of other entities (it has FKs) and it has to be treated differently.

At the end of the headers cycle, the string query is finished and ready to be used by an INSERT statement as in E.5, where *TargetPhysicalTableName* contains the target table name, *headers* is the string that stores all columns and *data* is a string with the correspondent values read.

```
1 SqlCommand cmdInsert = new SqlCommand("INSERT INTO " +
    TargetPhysicalTableName + headers + " OUTPUT INSERTED.ID VALUES" +data,
    target);
```

Listing E.5: Dynamic insert with variables

E.4 Data Mapping - RecordIdRelation

The established solution that allows to maintain the FK integrity and be able to migrate dependencies without loosing or corrupt data is using a data mapping table. The table stores the value of the source and the corresponding target value in each row. It is one of the most common implementation patterns because it is simple and effective, and developers do not need to rely on external heavy and unnecessary software dependencies.

When the data is being migrated, the mapping table has to be consulted and every time that the current attribute/columns read corresponds to a FK it should be transformed to the correspondent target value. This new value was previously stored on the data mapping table (after inserted) and is determined by a SELECT statement to the data mapping table with the source record identifier, entity identifier and the MigrationMaster identifier. The mapping table, named **RecordIdRelation**, contains the link between source and target records IDs for every entity inside a migration plan. The data mapping table allows to monitor the old and new ID of the migrated element.

Keep track of the old ID (record ID on the source) and the new ID (new record ID of the target after inserted) is necessary because the IDs on different environments can be different, i.e, the target environment can already have other records, or deleted records (the ID number is never reassigned to the next record), and since the ID is auto incremented, even when the table is empty, it does not mean that the ID starts in 1. The problem is that when inserting a record that maps to another table (FK), this ID-FK can be pointing to the wrong record. To solve this, the two identifiers are stored on an intermediary table while inserting and then, if migrating a table that has a FK to the other already migrated, the data mapping table is consulted to check the new value and change the FK from the old ID to the new.

A few steps were identified for the migration process:

1. Fill table while inserting on target database
2. Consult table to identify the target ID(FK)

3. Clear table after migrating all entities

This steps are presented with more detailed within the following items:

- **Save into RecordIdRelation while inserting**

  To persist the data on the data mapping table it is necessary to discover the old and new record ID. This step is done independently if the entity has or not dependencies.

  By default, the old (source) ID is the 1º position but this can not always be true. To guarantee that the right value is inserted the mechanism iterates the column headers of the current entity to discover the one that is called "ID" and save that value to an auxiliary variable called *oldId*, as shown on listing E.6.

```
1 long oldId = Convert.ToInt64(reader.GetValue(0)); //by default the the ID is
      the 1rst position
2
3 for (int j=0; j< headersCount; j++ ) {
4     if (reader.GetName(j).Equals("ID")){
5        oldId = Convert.ToInt64(reader.GetValue(j));  //retrieve old Id
6     }
```

Listing E.6: Get old record ID

Inserting the record into the target database, with the INSERT statement E.5, allows to obtain the new ID, retrieved by the insert command thanks to the OUTPUT clause. It returns information from each row affected by an INSERT, the results can also be inserted into a table, in this case into RecordIdRelation - the data mapping table. The clause used was INSERTED.ID, which retrieves the ID of the inserted record, adding it to an auxiliary variable called *newId*.

With both IDs it is possible to insert the relation into the data mapping table. The function UpdateRecordRelationTable, listed on E.7, uses the migration staging ID (MigrationMaster identifier) as well as the entity ID and the new and old record ID, to create a command that inserts an entry into RecordIdRelation.

```
1 public void UpdateRecordRelationTable(SqlConnection sqlConnection, long
      entityId, long oldId, long newId, long stagingId)
2 {
3     // creating  SqlCommand with parameters
4     SqlCommand cmd = new SqlCommand("INSERT INTO OSUSR_m7a_RecordIdRelation(
          MIGRATIONMASTERID,OLDRECORDID,NEWRECORDID,ENTITYID)" +
5     " VALUES("+stagingId.ToString()+","+oldId.ToString()+","+newId.ToString()
          +","+entityId.ToString()+")", sqlConnection);
6
7      cmd.ExecuteNonQuery();
```

```
8
9 }
```

Listing E.7: Update data mapping table

- **Consult RecordIdRelation while inserting**

  This step is done when the entity has dependencies. The algorithm consult the data mapping table RecordIdRelation to use the newId instead of the old in the FK columns. As already mentioned, a FK can be pointing to one record in the source database and this can be different or non existent in the target database.

  The first step is to check if the current entity has dependencies because only entities with dependencies need to consult the data mapping table. The same mechanism used for non-dependent tables is used also here, if the current column is "ID" then it is ignored, if the value is null, add value as "NULL". Otherwise, a function called FindFKByColumnName, partially listed in E.8, is called to check if the current column is a FK, if yes the query return the correspondent entity ID of the dependency, if not, returns a default value: -1. This function used two SQL queries: one that retrieves the Type of the attribute given the column name and another that is similar to the last query to get dependencies (D.1) but with two differences, the output is not the entity as a hole but just the attribute "ID" and the other difference is replacement of the Type to the value *type* retrieved by the first query result. The attribute Type of a column is composed by the eSpace and entity SS_KEYs of the FK, knowing the Type attribute makes it possible to find which entity corresponds to that dependency by matching it's SS_KEYs.

```
1 string query1 = "select a.TYPE from ossys_Entity_Attr a, ossys_entity e "
2     + "where a.ENTITY_ID=e.ID AND e.PHYSICAL_TABLE_NAME ='" + tableName + "'
          AND a.NAME = '" + columnName + "'";
3 SqlCommand cmd1 = new SqlCommand(query1, sqlConnection);
4 string type = (string)cmd1.ExecuteScalar();
5
6 string query2 = "SELECT DISTINCT e.ID FROM ossys_entity e, ossys_Entity_Attr
      a " +
7       "WHERE e.SS_KEY = (select right('" + type + "', charindex('*', reverse('
          " + type +'" //....
```

Listing E.8: Find entity FK by column name

The next steps are different if the current column is indeed a FK. If the current column is just a regular attribute then the mechanism is the same as already mentioned on Dynamic Insert E.3. On the other hand, if the column is a FK then the algorithm checks which it is by comparing the ID retrieved by the FindFKByColumnName method with the dependencies

list to check which one it is in the current column. Finally, the newId is searched by given the entity ID, staging ID and the old record ID as input to a function called GetRelationId E.9.

```
1 SqlCommand cmd = new SqlCommand("SELECT NEWRECORDID FROM
    OSUSR_m7a_RecordIdRelation " + "WHERE MIGRATIONMASTERID = " + stagingId.
    ToString() + " AND ENTITYID = " + entityId.ToString() + " AND OLDRECORDID
    = " +oldRecord.ToString()+ ";", sqlConnection)
```

Listing E.9: Retrieve target FK-ID

The output of the GetRelationId function is the target ID of that record, and is the value that is added to the text query for the insert.

As systems become more complex also the databases become larger which increase the number of matches to perform. The data mapping table contain entries of all records of several entities, which can lead to a considerable large amount of records. To solve this issue the records of one migration added to the table are always cleared at the end of the migration. This provides better performance to the next migration since it is easier to iterate the table searching for the FK values. No backup is done since this information is not important for subsequent migrations and neither for visualization.

Appendix F

Migration Log

The updates were created, inside the Extension, as functions such as: UpdateTotalRowsOfAllEntities, used to save the total rows of all entities - for the progress bar and SetMigrationStatusToDoneState, needed to update the IsFinish column of the MigrationMaster that indicates if the migration is finished. Both functions use a SqlCommand to update the MigrationMaster entity given it's ID. Particularly, for each entity counter and status update three functions were created: UpdateCurrentRow, UpdateTotalRow and UpdateStatus.

Before the migration, the list of entities was iterated, using each physical table name to count how many rows exist, for each entity. The local integer variable totalRowsOfMigration sums the total rows of all entities and it is used to update the MigrationMaster entity. This first iteration was necessary to the progress bar, since it joins all entities records for a global view. The entities list are then iterated again, while the migration is taking place, updating the status of each entity row. The first update is setting the entity to a Progress state, which is done when the list advances to the next entity. The function UpdateStatus is called with the entity's composite key (entity ss_key and eSpace ss_key), the MigrationMaster ID and the status to the correspondent enumerated value of "In Progress" (2). The UpdateTotalRows is also called, using the physical table name to get the total entity rows as listed on F.1.

```
1  cmd = new SqlCommand("UPDATE d SET d.TOTALROWS = ( SELECT COUNT(*) FROM " +
       tableName + " ) FROM OSUSR_m7a_MigrationDetail d " +
2  "WHERE  d.MIGRATIONMASTERID = " + stagingId.ToString() + " AND d.ENTITIYSSKEY = '"
       + entSSKey + "' AND d.ESPACESSKEY = '" + espSSKey + "';", sqlConnection);
```

Listing F.1: Update totalRows of MigrationDetail

Next, while iterating each row of a given entity (while (reader.Read())) the function UpdateCurrentRow is called. This function executes another SqlCommand to increments the CurrentRow attribute given the entity composite key identifier. At the end of an entity's migration, the entity status is set to done, using UpdateStatus with the correspondent enumerated of the "Done" status (1).

Finally, when all entities were iterated and migrated, the function SetMigrationStatusToDoneState is called to set the migration status to finished. The current entity is always stored at an auxiliary variable so that, in case of error (catch the exception- block of code to be executed, if an error occurs in the try block), the variable is used to update the status of that entity to "Error" Status (3) and add the message error provided by the exception.

At Service Studio, the MigrationLog Screen was created to visualize the results in real time. It fetches everything that is needed to start a migration, the source and target connection and the migration master. The Screen is composed by a set of aggregates to fetch the database data to be displayed, every second. These aggregates are mentioned in Section B.8. When it is ready (screen preparation) it fetches the aggregates data as input to call the Migrate server action from the extension. After fetched, the migration starts and the data is updated according to the extension.

Appendix G

Reuse Migration Plan

The MigrationDashboard Screen was created and set as Home Screen to provide a view were the developer can see all executed plans and perform some actions on each one. It displays the migrations that are already finished, with a button to reuse/clone in each row, i.e., each migration plan.

A new server action, CreatePlanClone G.1, was created to receive the migration master identifier as input and run a SELECT advanced query to fetch all information regarding that plan, this information is then added to another server action called CreateMigrationPlan (which is also used for normal plan creation). In this last action the plan (MigrationMaster table record) is created in the database. After that both MigrationDetail and MaskAttribute tables are updated with new data to create the full plan support. When the developer chooses to clone one migration plan the CreatePlanClone action is called, retrieving the new plan identifier.

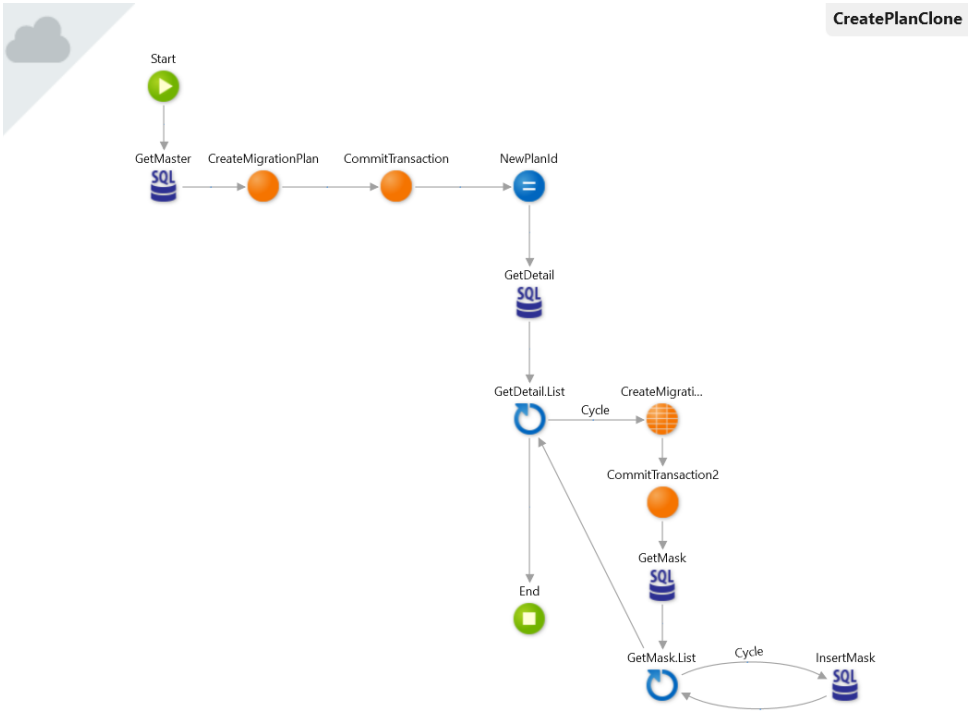


Figure G.1: CreatePlanClone Server Action

Appendix H

Differential Migration

The implemented solution involves the creation of a new entity, called PartialRecords. This data table stores the identifier of the entity and it's environment, since storing the migration plan was not an option because the plan can be deleted by the user. The other, most relevant, attribute is the identifier of the last record migrated (*LastRecordId*). At the MigrationMaster entity an attribute called IsDifferential, representing a flag to this feature, was also added. This attribute allows the developer to chose if the migration will only have the latest records or just migrate everything. The last record of the source environment inserted on the target (migrated) is always saved on the PatialRecords data table, even if the IsDifferential boolean is set to false because it could be used in the next differential migration. The tool inserts, for each entity, the record into PartialRecords, which contains the entity identifier, target environment database name and last identifier (last entity identifier migrated).

First, when there is no data at the target and no previously migrations were executed, all records are migrated, because the PartialRecods table is consulted and does not contain any value for that environment and entity.

At the subsequent migrations, this table will also be consulted with a SELECT query, as listed in H.1, and if the IsDifferential value is true then the source SELECT will filter the entities records identifiers that start from the last ID, as in H.2, where the *last* variable is the result of the SELECT query that retrieves the LastRecordId of a given entity identifier and target environment.

```
1 SqlCommand cmd = new SqlCommand("select LASTRECORDID from OSUSR_m7a_PartialRecords
      where ENTITYID = " + entityId.ToString() + " AND CONNECTIONCATALOG = '" +
      connection + "'", sqlConnection);
2 long last =  cmd.ExecuteScalar();
```

Listing H.1: Get Last record migrated for one entity

```
1 //verifications- isDifferential?
2 new SqlCommand("SELECT * FROM " + table + " WHERE ID>" +last, source);
```

Listing H.2: Selecting partial

This code is present on the extension, executed with and SqlDataReader that retrieves all records to be migrated by each entity.

Appendix I

BirdlyDataNotifications

I.1 Send Notifications action - configurations

Figure I.1 represents Service Studio configurations of the BirdlyDataNotifications Public action.

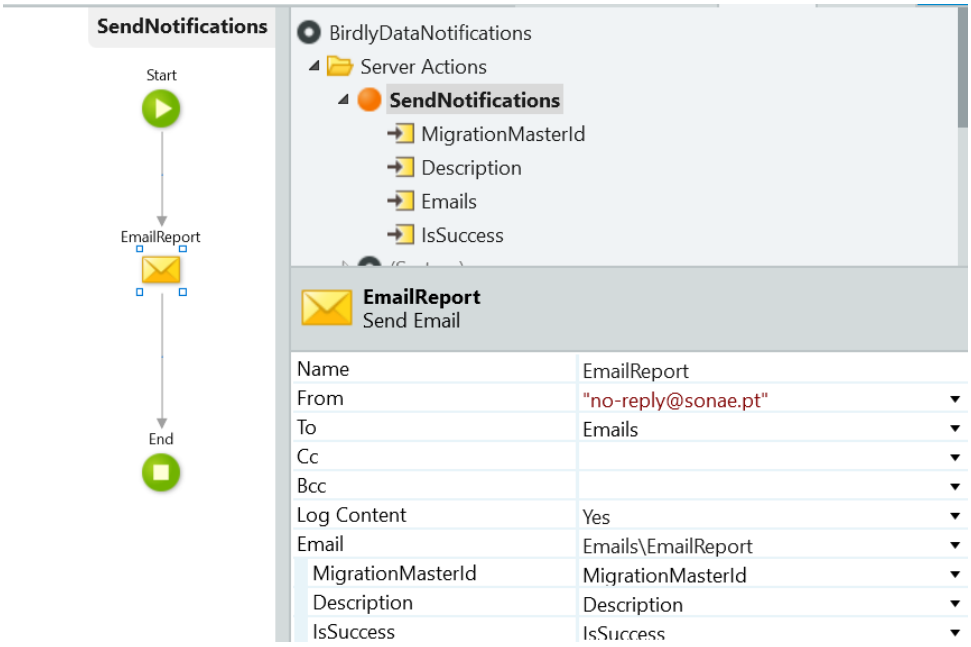


Figure I.1: BirdlyDataNotifications - Traditional Send Email

I.2 Real email example

Figure I.2 represents a real email, sent after the migration of the "test all" plan, with ID:284.

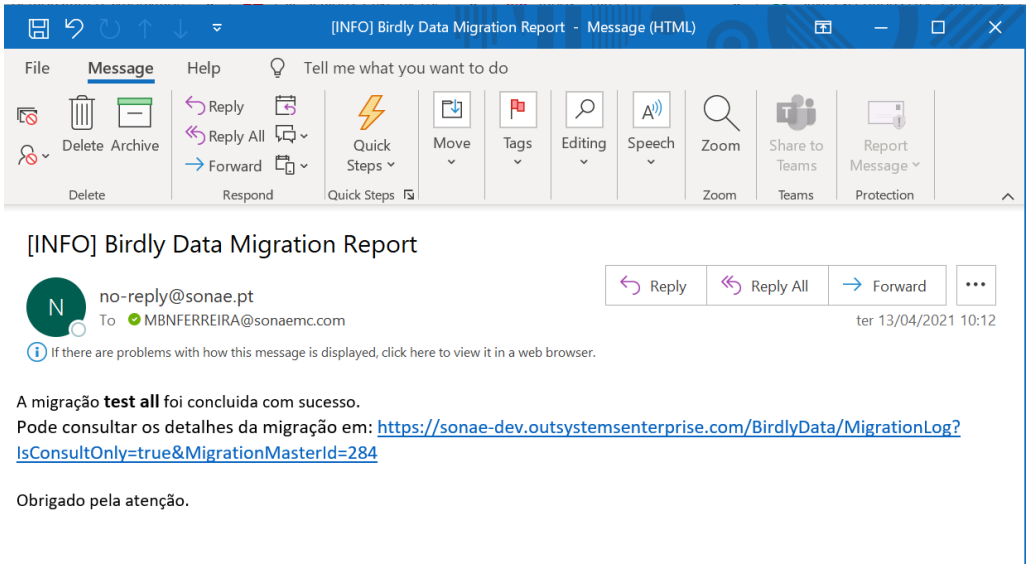


Figure I.2: Real email

Appendix J

Migrate Users

J.1 BirdlyDataAPI

To call the CreateUser action on the target environment, one service, called BirdlyDataAPI, was created in the BirdlyData application. This module just exposes one REST API with a HTTP POST as in Figure J.1. This receives an User record (body), API KEY and Tenant Name (User Provider Name).

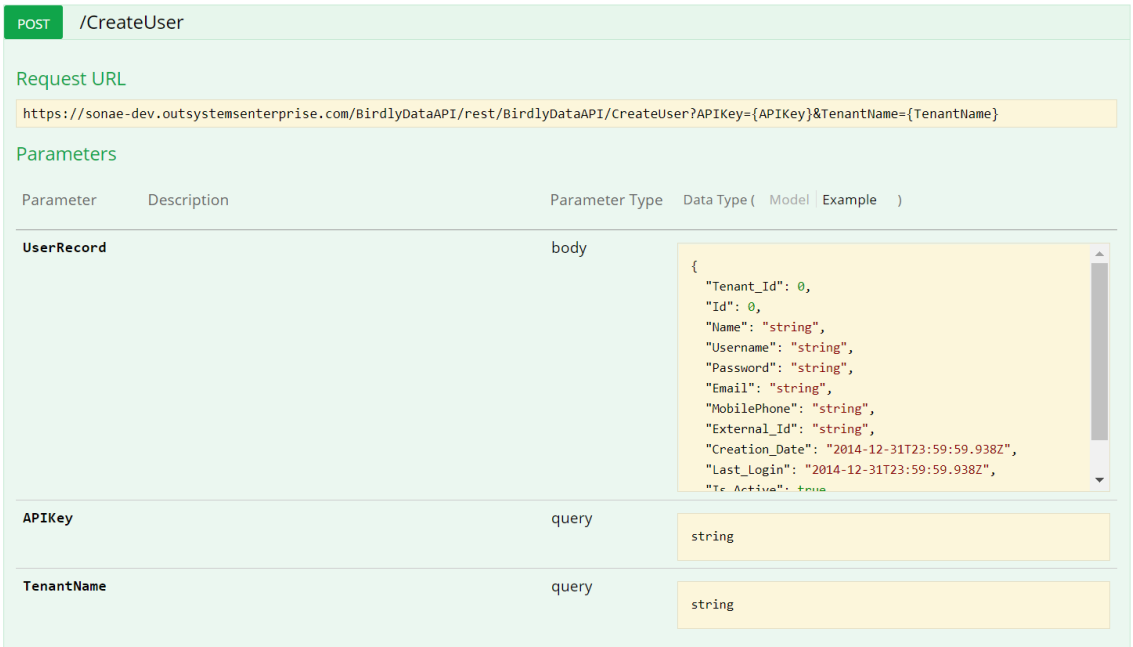


Figure J.1: HTTP POST BirdlyDataAPI

For security reasons (unauthorized calls), one APIKEY was added to the Site Properties. When the API is called, the input key is checked and compared with the key stored at the Site Properties. If the input key does not match the stored, it means that an unauthorized call is taking place and it returns the server error.

Since OutSystems advanced queries use the entity name in braces (e.g. Employee), which is different than the traditional SQL queries that use the physical table name (e.g. OSUSR_0o7 _Employee2), OutSystems' queries only retrieve entities inside the current eSpace where they are called. This limitation does not allow to create a query to select information about other application entities data or other environments, because one application only has access to it's own data. To check if an User exist in the target there was a need to force the SQL to accept the physical table name instead of the entity name. This was possible by creating a query in text format which will be the input parameter of the SQL query. To allow use the query parameter to implement all of the SQL query at runtime the **extend inline** has to be set to yes, otherwise OutSystems will only retrieve Users in the current User Provider (Tenant). Additionally, the API Module property **IsMultiTenant** should be set to yes because it enables multi-tenancy to this Module (where the User is created).

The text query, on J.1, injected into an OutSystems advance query, allows to **check if the source User is already created on the target environment**. It depends of two variables: User-Record.Name and Tenant.Id (composed unique identifier of the User record between environments). Where the GetTenant is a created aggregate that fetches the Tenant ID of the target, by filtering the Tenant that has the same source Tenant Name (there could only be one).

```
1 declare @user VARCHAR(50) = 'ossys_user';
2 declare @tenant VARCHAR(50) = 'ossys_tenant';
3 EXEC('select u.id from ' + @user + ' u,  ' + @tenant + ' t
4 where u.TENANT_ID = t.id and u.USERNAME = ''" + UserRecord.Name + "'' and t.id = "
      + GetTenant.List.Current.Tenant.Id + " ')
```

Listing J.1: Get Users on target (to verify if they already exist)

With this query is possible to check if the User of a determined User Provider already exists in the target (differential migration), if it exists, the SQL retrieves the User ID of the target environment as output. If the User does not exist on the target then the CreateUser server action will be called by importing it and pass the input body content: User record. Either way, the ID will be retrieved by the API to be used later to update the entity RecordIdRelation (mapping table).

To create the User record another server action should be imported and used: GrantUserManagerRole, this is an UserManager action used to grant the manager role to the current User (the one that performs the migration), essential to create users.

Figure J.2 represents the BirdlyDataAPI logic , the CreateUser HTTP POST uses an User-Record(DataType:User) to verify if the target already contains any User with the same Username and Tenant ID, creating the User if no User is found.

The API will be exposed inside the target environment and consume it (call) on the source. Which, again, needs BirdlyData **installed** in **both origin and target environment**.

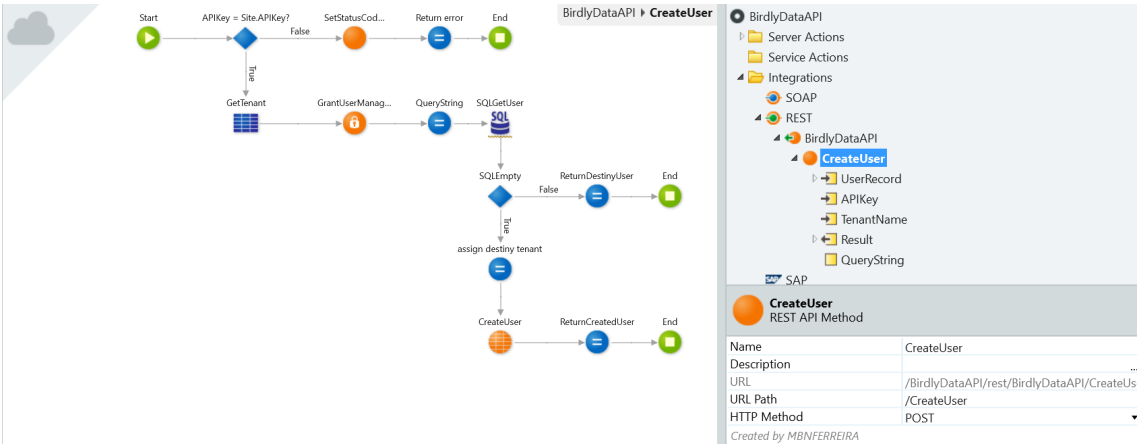


Figure J.2: API logic

J.2 Inside Service Studio

To flag this functionally, another boolean IsMigrateUsers was added to the MigrationMaster plan.

At BirdlyData Module three server actions were created: CreateUsersOnTarget, and two more auxiliary actions GetUserDependencies and GetUsers:

1. CreateUsersOnTarget — called before the Migrate action, that invokes the extension, and is the one that consume the REST API BirdlyDataAPI.

2. GetUsers — called by CreateUsersOnTarget to fetch User records which will be used as body of the API call (retrieves a list of User records).

3. GetUserDependencies — called by GetUser to find the User IDs that are referenced by the other entities (retrieves a list of User IDs).

- **GetUserDependencies**

  To set the body content of the HTTP call (JSON format) a list of User records is needed. An OutSystems query, similar to the get dependencies query D.1, replaces the entity SS_KEY to the @UserSSKey value and is used to identify which Users are used by the other entities. For each entity to be migrated, the query is called, searching for User dependencies. For each record of the SQL output a new text query was also created.

  This is an important step to provide a differential migration, since avoids creating non-essential Users and spend unnecessary money for additional Users license.

  The content of GetUserDependencies is illustrated in Figure J.3.

  The built of the text query was made in two parts. First, the SQL output was iterated, for each record, the DECLARE statement was added to the text value, using the current TableName and ColumnName. Second, the same list output was iterated once more, to fill the rest of the text query, starting with the EXEC statement. In this section, all entities'

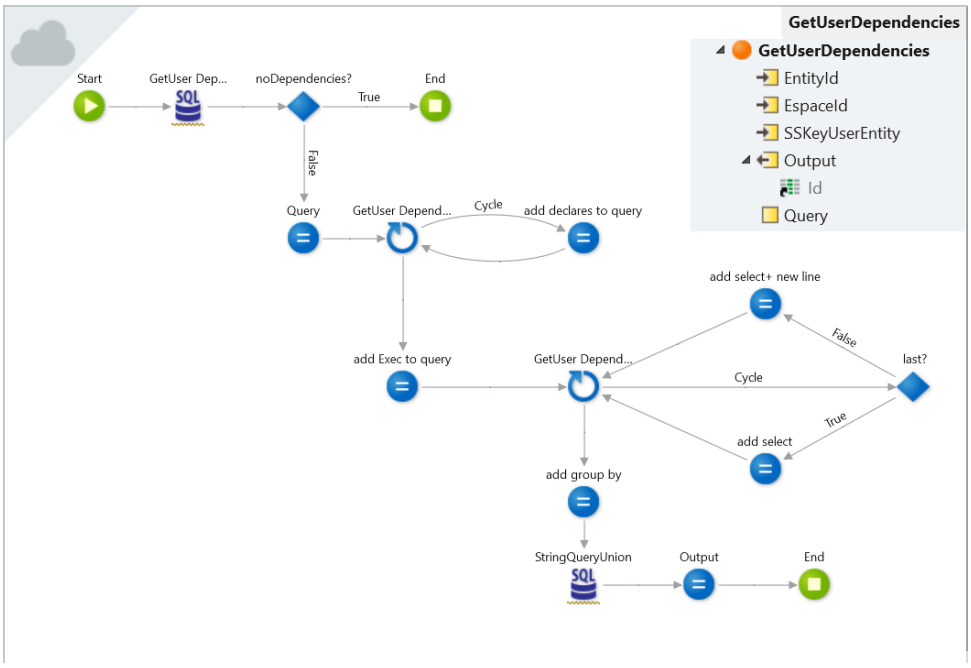


Figure J.3: GetUserDependencies

attributes that reference Users are selected, for each entity, and grouped by ID. Finally, another SQL advanced query was built from this text query (expand inline), that joins all Users IDs referenced by the other entities. An example of how the query could be is stated in J.2.

```
1 declare @PRODUCT_UPDATEdby VARCHAR(500) = 'OSUSR_UZ2_PRODUCT' ;
2 declare @PRODUCT_createdby VARCHAR(500) = 'OSUSR_UZ2_PRODUCT' ;
3 EXEC('
4 select t1.userid from (
5 select UPDATEdby as userid from '+ @PRODUCT_UPDATEdby + ' where UPDATEdby is
      not null group by UPDATEdby
6 union all
7 select createdby as userid from '+ @PRODUCT_createdby + ' where createdby is
      not null group by createdby
8 )
9 as t1 group by t1.userid ')
```

Listing J.2: Example query: Get used Users ID used by other entities

The creation of the text query is the only way to force the OutSystems advanced query to manage an entity of another application or eSpace.

- **GetUsers**

With the Migration Master ID it was possible to retrieve the Structure list of Entity ID and Espace ID that correspond to the selected entities (entities to migrate). To find which Users to migrate, this entities were iterated, calling the action GetUserDependencies, which retrieves an User IDs list in case of the current entity having User dependencies.

For each record in J.2 another query was called to get all User attributes (content), needed to build the body of the HTTP POST request. The SELECT query J.3 specify the attributes because the attributes order of the ossys_user table can be different of the OutSystems system table User (founded after a constant error message: "switched attributes").

```
1 declare @user VARCHAR(50) = 'ossys_user';
2 EXEC('select ID,NAME,USERNAME,PASSWORD,EMAIL,MOBILEPHONE,EXTERNAL_ID,
      CREATION_DATE,LAST_LOGIN,IS_ACTIVE from ' + @user + '
3 where ID = " + GetUserDependencies.Output.Current.Id + " ')
```

Listing J.3: Get used Users

The content of GetUsers is illustrated in Figure J.3.

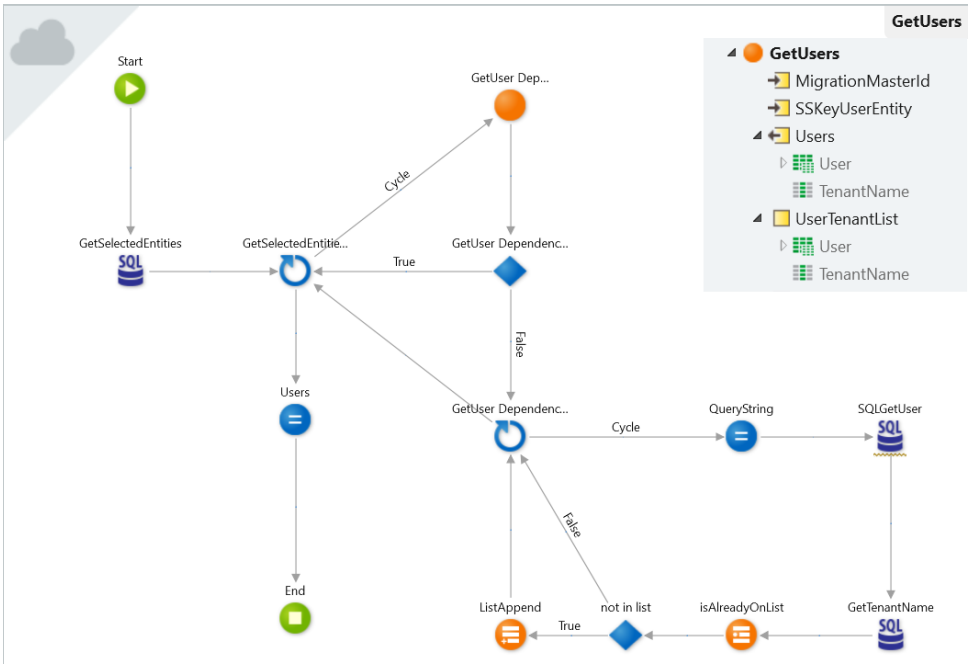


Figure J.4: GetUsers

The eSpace was also used to get the User_Provider_Name (Tenant Name) that is the environment unique identifier of the tenant. Both values were added to an OutSystems Structure list of results, composed with an User record and corresponding Tenant Name. This list is the output of the action used by CreateUsersOnTarget.

- **CreateUserOnTarget**

This action is called before the migration of the other entities. It starts with the update of the value stored in the entity PathAux, setting the value to the target connection URL.

At the Integrations section of the Logic tab, it was necessary to add a configuration to consume (call) the API. The swagger specifications are added to facilitate consumption using the URL provided by OutSystems.

However, a new problem emerged since the HTTP call URL needs the target as base URL (instead of the source) and OutSystems base URL is fixed to the source environment. That was the reason that lead to the creation of a new field called URL at the Connection entity and to the creation of the PathAux entity. PathAux stores a single record: target Url, which is updated every time that a new migration with Users is run.

There is no direct way to customize the request, since the base URL is already set. Alternatively, an action to customize a request before sending it: OnBeforeRequest (RestAPICallback) was created to solve this problem. This action does not allow to set any input variable, that is why the PathAux was created, since the only way to pass a base URL value was to store it in the database.

Inside the OnBeforeResquest, represented on Figure J.5, the value stored in PathAux is fetched and it is assigned to Request.BaseURL, changing the request to a customized request.

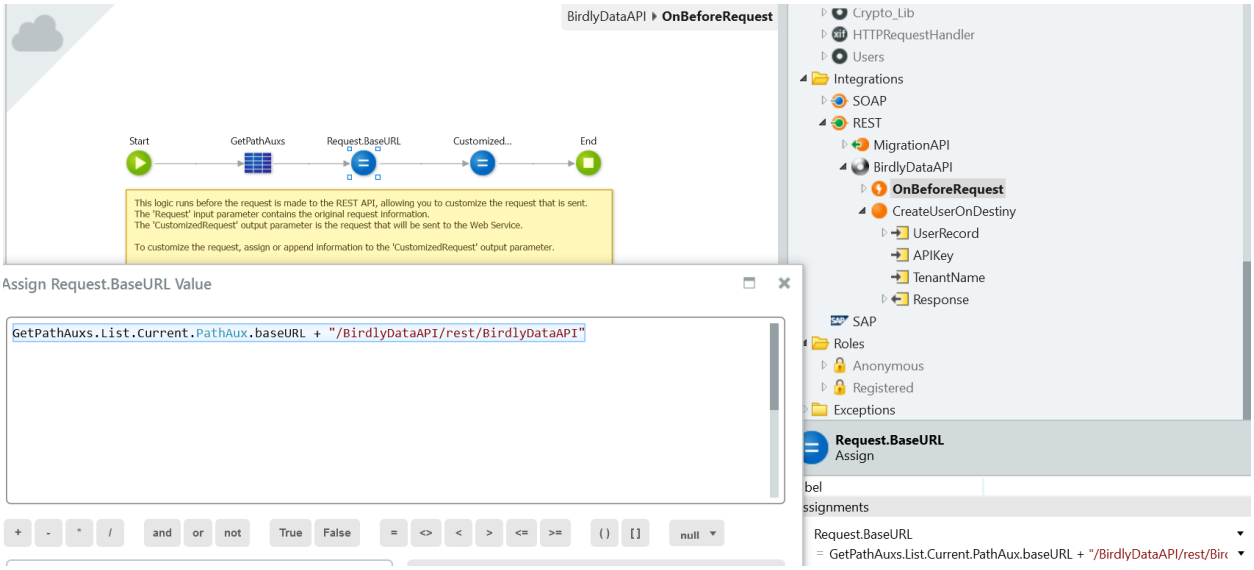


Figure J.5: API logic

The action GetUsers is called given the MigrationMaster identifier. For each User, a HTTP call is made, consuming the BirdlyDataAPI. Here is where both body and headers are filled. The APIKEY is also given for security. Two SQL queries are created, one before the User creation and another after, updating the log status of the entity migration (attributes of MigrationDetail: CurrentRow, TotalRows, Status and FinishTime). Every time an User record

is created with success on the target environment then a new value is inserted in the auxiliary RecordIdRelation table, storing the source(old) and target(new) ID of that record. This process is illustrated on Figure J.6.

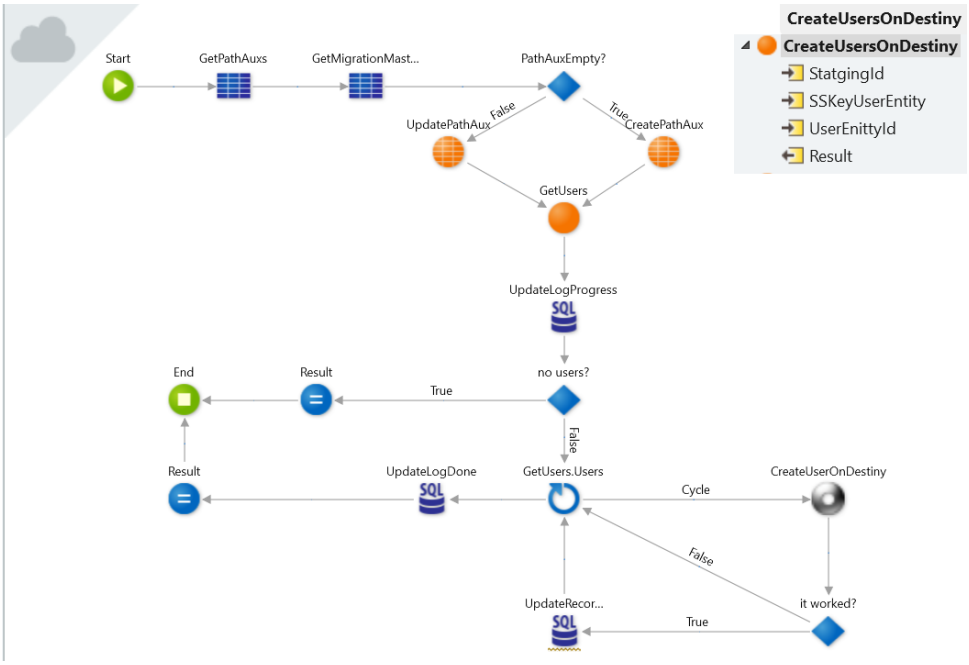


Figure J.6: CreateUsersOnTarget

J.3 Inside extension

In .Net, the User table is ignored. For the other entities, when some User dependency is founded, then the boolean (IsMigrateUsers) saved on the MigrationMaster plan is checked. If set to false, means that the Users records were not migrated so the insert value will be NULL. Otherwise the data mapping table is consulted as always to fetch the target ID.

Appendix K

Mask Data

K.1 Fetch entity's attributes

For fetching the list of attributes of one entity, the GetEntityAttributeByEntityId Data Action was created at the EntitiesList Screen. This action uses an SQL K.1 to retrieve all attributes of one entity, excluding the ones that are not allowed to mask since they can compromise the data logic, such as the entity identifier (ID) and entity dependencies (FK). An entity dependency can be detected through the Type column of the Entity_Attr table, where types starting with "bt" characters are considered entity dependencies. To exclude them the operator NOT LIKE was used, that retrieves the results that do not match this expression. To exclude primary keys the operator different (!=) was used to filter the attribute name "Id".

```
1 SELECT {Entity}.[Name] , {Entity_Attr}.[Name]
2 FROM {Entity} , {Entity_Attr}
3 WHERE {Entity}.[Id] = @EntityId
4 AND {Entity_Attr}.[Entity_Id]= {Entity}.[Id]
5 AND {Entity_Attr}.[Type] NOT LIKE 'bt%'
6 AND {Entity_Attr}.[Name] != 'Id'
```

Listing K.1: Selecting entity's attributes

K.2 Import attributes to EntitiesList Screen

Consequently, a new auxiliary entity was created called MaskAttribute with the follow attributes: entityId, AtrributeName, MigrationMasterId and MigrationMasterDetailId. The MaskAttribute is related to Migration Detail (many-to-one relationship) since one Migration Detail record correspond to one entity, but one entity can have several attributes.

In the EntitiesList Screen, the mask can be set before the user select the entities, so there is no exact way to know which attributes are going to be relevant until after the developer selects which entities he desires to migrate. To manage this, a new local Structure, similar to the

MaskAttribute entity, was created on Service Studio to manage the mask configurations with the following attributes: EntityId (long), EntityName (text) and AttributeNameMask (list of another structure containing the attribute name and a boolean - IsMask - that triggers the anonymization of the attribute). The UI uses this structure in a local list to store the information of all entities' attributes.

While saving the fetched entities at the local list (OnAfterFetch client action represented by Figure K.1), the AttributeMask local list is also filled with the entityId and the entityName, creating records for each entity on the mask list. The attribute missing is the AttributeNameMask list, to find this information, the entities list is iterated again and the local variable EntityId of the screen is set to the current entity being iterated (this identifier is used as a filter on the GetEntityAttributesByEntityId data action). To fetch the attributes' names one refresh operator is used to the GetEntityAttributesByEntityId data action (using the entityId to find the attributes of the current entity), refreshing the data action is essential to fetch the attributes of a given entity, otherwise the attributes fetched could not correspond to the current entity being iterated.

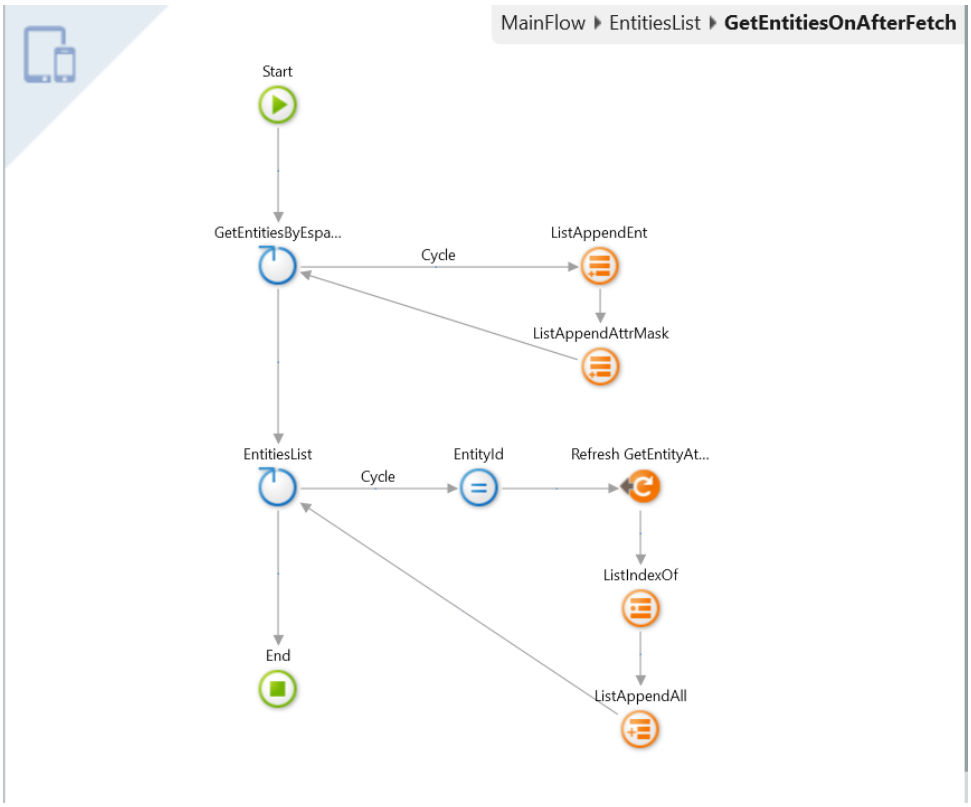


Figure K.1: Import attributes

The action in Figure K.1 first saves, for each entity, the identifier and name's entity into the Structure AttributeMask. The second loop adds the GetEntityAttributesByEntityId data action's result (list of attributes' names) to the AttributeNameMask.AttributeName of the AttributeNameMask Structure and the AttributeNameMask.IsMask is first assigned to False, in all at-

tributes. Figure K.2 illustrates the Structure used by GetEntitiesOnAfterFetch event of EntitiesList Screen.

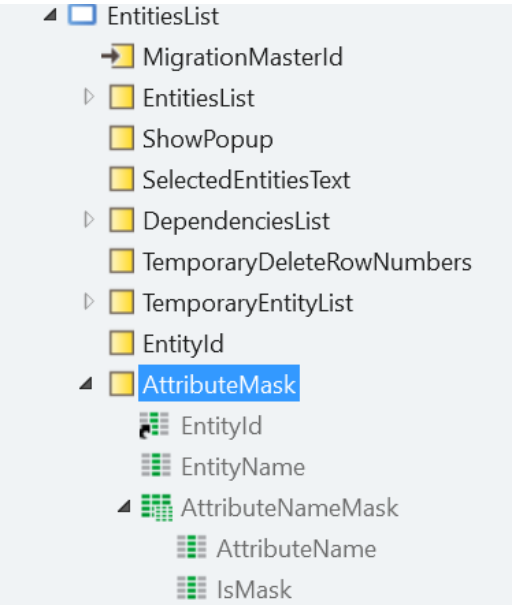


Figure K.2: AttributeMask structure

K.3 Opening an entity's mask popup window

Since the local mask list already has information of all entities identifiers and names, then another client action to get the index (Position) of the correct record of the mask list was called (ListIndexOf), this uses the entityId to get the position of that entity inside the mask list and uses it to add all fetched attributes to the respective AttributeNameMask structure list, and assign the isMask boolean to False to all records.

Every time the developer wants to mask some attribute he clicks on the respective entity Mask icon, this will set the local EntityId to the selected entity identifier and open a new pop up with all the entity's attributes and respective checkbox widget corresponding to the IsMask flag. Another local variable used to manipulate data is the PositionMask integer, the logic of opening the pop up also sets the PositionMask to the position of the selected entity in the AttributeMask local list. This position is used to display the attributes of a selected entity using the expression: *AttributeMask[PositionMask].AttributeNameMask* to get the attributes names and isMask boolean saved on the AttributeNameMask structure.

K.4 Create MaskAttribute records - database persistence

Finally, when the developer selects the entities to migrate and clicks continue, the MigrationMaster is updated with the Application, Modules and selected Entities information. The MigrationDetails

records are also created at this phase, for each entity. Nonetheless, it is still essential to store, into the database, the information of which attributes to mask. The database entity that can receive this information is the MaskAttribute entity and the inserts are made while creating the MigrationDetail records, binding each entity that corresponds to a MigrationDetail record to the set of attributes stored for that entity at MaskAttribute. This set of actions and SQL queries are performed inside the server action: UpdateMigrationPlan, illustrated in Figure K.3, that receives the AttributeMask local list and, for the selected entities, calls a Server Action to insert the mask information of the given AttributeMask List into the MaskAttribute entity. This means that, each record in the entity MaskAttribute has a FK to the MigrationDetail and, for example, if an entity has three attributes then the MaskAttribute should have three records, one for each attribute, and the three records have one MigrationDetail and one MigrationMaster in common.

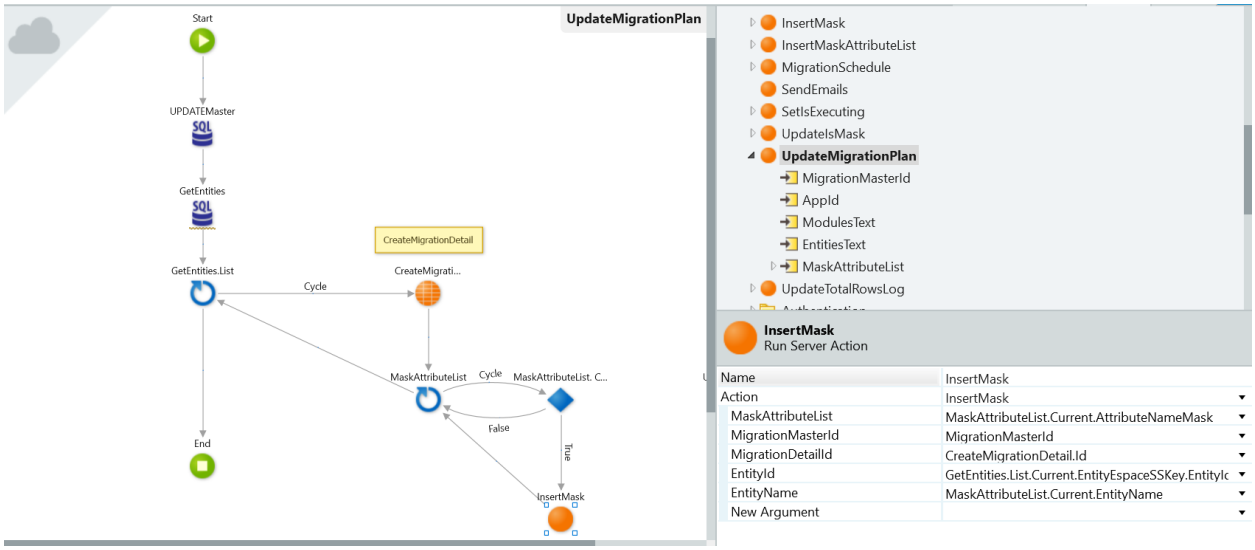


Figure K.3: UpdateMigrationPlan server action

K.5 Anonymize algorithm

After the reunion of the columns that should be masked (at the UI), a mask technique called Anonymization is applied to this columns in the extension. The anonymized data corresponds to the default data, specified also by OutSystems [5], of each data type, for example, the text type default is an empty string (""), DateTime default is "1900-01-01 00:00:00", for numbers the default value is 0, etc.

Each column type is verified before masking the data. The code listed on K.2 is called in the extension, the function MaskDataByType receives the column name and is responsible to retrieve the masked data accordingly to it's type. The function retrieves always a srting because this value will tbe used into the INSERT statement query that has a text format. Other formats not specified are retrieved as NULL because they aren't relevant to the project.

```
public string MaskDataByType(string columnTypeName)
{

   if (columnTypeName.Equals("nvarchar") || columnTypeName.Equals("varchar"))
  {
       return "''"; // returns an empty string
  }else if (columnTypeName.Equals("datetime"))
  {
    return "'1900-01-01 00:00:00'"; //default DateTime value for OutSystems
  }
     else if (columnTypeName.Equals("int")|| columnTypeName.Equals("decimal") ||
         columnTypeName.Equals("numeric") || columnTypeName.Equals("bigint"))
    {
       return "0";
     }
    else
   {
        return "NULL";
    }
}
```

Listing K.2: Anonymize data

Appendix L

Clean target entities

First, the extension fetch the entities' identifiers list from the MigrationMaster entity to use as parameter input to a SELECT SQLCommand. This SELECT uses the IN operator to search the composed SS_KEY of each entity present in the input list and a ORDER BY DESC since inserting data into the tables should respect the dependencies order, but clearing (DELETE) the tables should use the reverse order. The result of this query is a SQLDataReader composed by the entities' SS_KEYS, which are used as input to the function: GetPhysicalTableName. Finally, with the SQL table name, of all entities, a string query to delete the tables is created and executed.

After deleting the entities on the target, the tool uses the target connection info to match the *Catalog* (which identifies the environment) and delete all PartialRecords records that correspond to those environment entities. This is the only moment where the information about the last records migrated for a determined entity and environment, saved at the PartialRecords, is deleted.

The complete function is listed on L.1.

```
1  public void emptyDestiny(SqlConnection source, SqlConnection destiny, long
       migrationId)
2  {
3      SqlCommand cmd = new SqlCommand("select IDENTIFIERENTITIESLIST from
           OSUSR_M7A_MIGRATION where id = " + migrationId.ToString(), source);
4
5      string ids = cmd.ExecuteScalar().ToString();
6
7      //reverse order because it is a delete
8      SqlCommand cmd2 = new SqlCommand("SELECT e.SS_KEY as entity, s.SS_KEY as espace
            FROM ossys_entity e , ossys_Espace s WHERE e.ID IN (" + ids + ")  " +
9      "AND s.id = e.ESPACE_ID " +
10     "ORDER  BY CHARINDEX(CAST(e.ID AS VARCHAR), '" + ids + "') DESC", source);
11
12     string query = "";
13     SqlDataReader reader = cmd2.ExecuteReader();
14     while (reader.Read())
15     {
```

```
16          string table = GetPhysicalTableName(destiny, reader["entity"].ToString(),
                reader["espace"].ToString());
17          if (!table.Equals("OSSYS_USER"))
18          {
19             query += "DELETE FROM " + table + ";\n";
20          }
21       }
22
23      SqlCommand cmdDelete = new SqlCommand(query, destiny);
24      int result = cmdDelete.ExecuteNonQuery();
25
26      //delete partial records
27      SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder(destiny.
            ConnectionString);
28      string database = builder.InitialCatalog;
29
30      SqlCommand cmdDeletePartial = new SqlCommand("Delete from
            OSUSR_m7a_PartialRecords where CONNECTIONCATALOG = '" + database + "'"
31      + " AND ENTITYID IN (" + ids + ")", source);
32      int resultpartial = cmdDeletePartial.ExecuteNonQuery();
33
34  }
```

Listing L.1: Delete target tables

Appendix M

Migration API

Figure M.1 — Migrate HTTP REST Documentation.

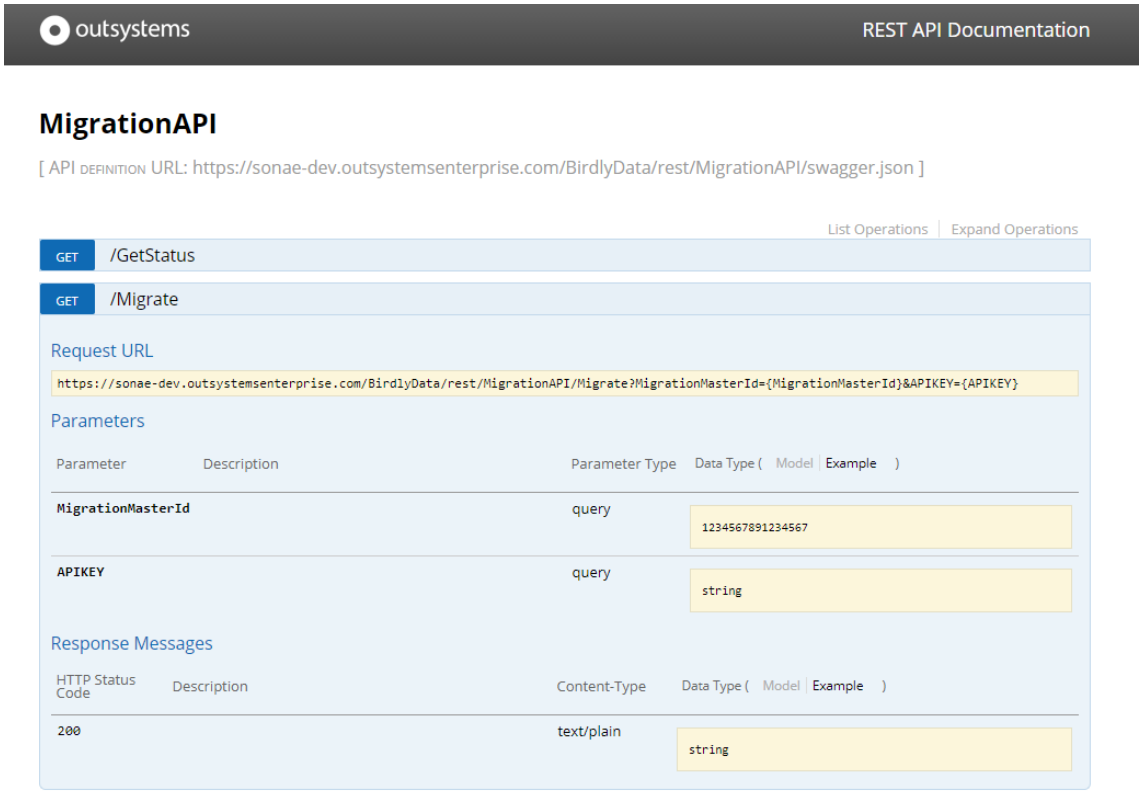


Figure M.1: Migration API

Figure M.2 — GetStatus HTTP REST Documentation.

Figure M.3 — OutSystems MigrationAPI inputs.

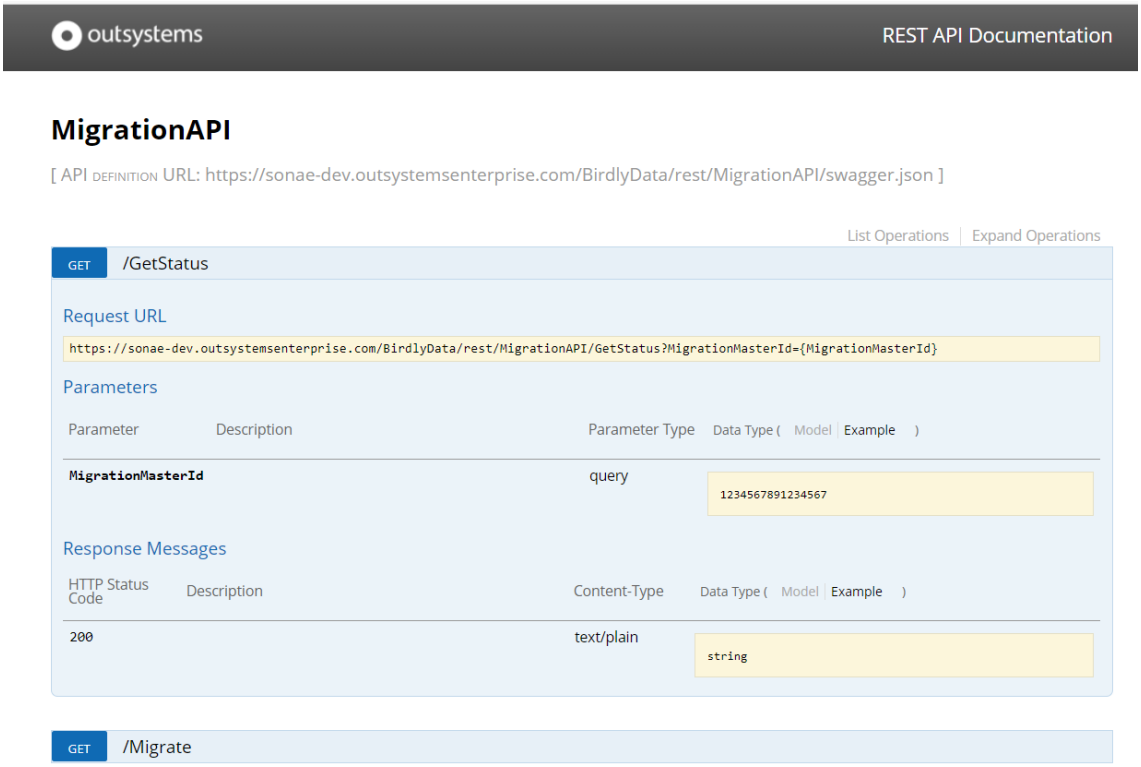


Figure M.2: GetStatus API

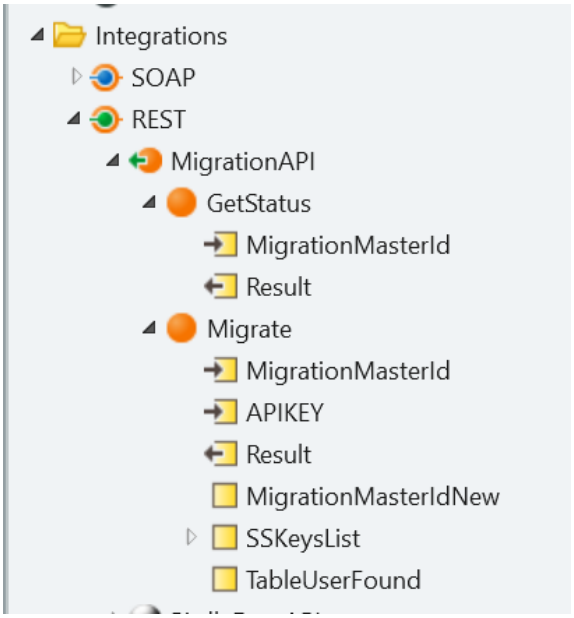


Figure M.3: OutSystems API definition

Appendix N

Special Case: Self-referencing foreign key

Sometimes, the child and parent tables may refer to the same table. In this case, the foreign key references back to the primary key within the same table.

A good example is a table of employees with the following columns:

- EmployeeID- Primary Key
- EmployeeName
- ManagerID - Foreign key going back to the EmployeeID

The ManagerID column is a foreign key that refers to the EmployeeID column which is the primary key of the employees table. This relationship allows the employees table to store the managing structure between employees and managers. Each employee is managed from zero or one employee and an employee can have zero or many subordinates.

The foreign key on the column ManagerID is known as a cyclic or self-referencing foreign key.

Usually, when inserting a record into the table, the value of the column corresponding to a FK should already exist as a PK on the table. For example, the first employee that we add to the table cannot have a managerId because there are no more employees yet registered. The most common order of the references is that records inserted later always reference previous records.

However, this FK value can then suffer an update, building the reference after inserting the record. This means that the references are not necessarily ordered, i.e., the record present on line 5 can reference a record present on line 6 because after inserting the record number 5 without managerId and add the record number 6, it is possible to update the FK of the record 5 to point to the record 6.

To the line-by-line migration process, this last case can be a potentially problem. Since migrating the data with one record referencing another record with bigger index (that was not wet migrated - so it does not exist) then an error would occur.

The solution found to solve this problem was migrating this data in two phases. Where first the table is migrated but the FK column values that reference itself are inserted as: Null, and secondly, the values are updated with the correct (target) FKs, building the references only at this later stage.

The implementation is done in the extension. When reading an entity records, in the case where the current column is detected as a FK, then a new block of code N.1 was added that checks if the entity dependency is itself triggering a local flag that indicates if self-references were found in the table and adding the value as NULL. An auxiliary local list of oldFKIds is also filled by order, line by line, with the old (source) FK identifier. Finally, the column name is also stored to assist the update query.

Afterwards, when the migration of all the data of the table is finished N.2, the flag is checked and if it was activated (the entity references itself) then the local list is iterated parallelly with the records stored in the data mapping table. The Update SQLCommand uses target physical table name already stored on TargetPhysicalTableName, as well as the column name of the FK. The SQLCommand also uses the value to update (to the SET clause) - which is retrieved by searching into the data mapping table for the NewRecordId that has a OldRecordId matching the current oldFKId of the auxiliary list. The final variable used by the SQLCommand is the current NewRecordId of the data mapping table for the WHERE clause. The data in the data mapping table is stored locally into a DataTable object, what is useful to search the correct(target) Id of the reference.

```
1  //if the current column in a FK ..
2  //special treatment for entities that reference themselves - if the FK is itself
3  if (Convert.ToInt64(dependencies.AttributeDependencyId[reader.GetName(j)]) ==
       entityID)
4  {
5      found = true;
6      if (reader.GetValue(j).Equals(DBNull.Value))
7      {
8          oldSelfFK.Add(null);
9      }
10     else
11     {
12         oldSelfFK.Add(Convert.ToInt64(reader.GetValue(j)));
13     }
14     columnAutoFK = reader.GetName(j);
15     dataRow[reader.GetName(j)] = DBNull.Value; //insert fk as null to update later
16 }
```

Listing N.1: Self-FK detection

```
1
2  if (found) //if the entity references itself
3  {
```

```
4     setIsMigratingSelfFK(source, stagingId, entitySSKEY, espaceSSKEY, 1);
5     int mapIndex = 0;
6     SqlCommand cmdLoad = new SqlCommand("select * from OSUSR_m7a_RecordIDRelation
          where MIGRATIONMASTERID = " + stagingId.ToString() +
7     " and ENTITYID=" + entityID.ToString(), source);
8
9     DataTable aux = new DataTable();
10    SqlDataAdapter da = new SqlDataAdapter(cmdLoad);
11    da.Fill(aux);
12    da.Dispose();
13
14    foreach (DataRow row in aux.Rows)
15    {
16        long? oldselfFK = oldSelfFK.ElementAt(mapIndex);
17        if (!oldselfFK.Equals(null))
18        {
19            DataRow finalMap = aux.Select("OLDRECORDID= " + oldselfFK.ToString()).
                  Single();
20
21            if (!finalMap.IsNull("NEWRECORDID"))
22            {
23                SqlCommand updateFK = new SqlCommand("UPDATE " +
                      TargetPhysicalTableName + " SET " + columnAutoFK + "=" +
                      finalMap["NEWRECORDID"].ToString()
24                + " WHERE ID=" + row["NEWRECORDID"].ToString(), destiny);
25             }
26
27        }
28        mapIndex++;
29  }
30
31setIsMigratingSelfFK(source, stagingId, entitySSKEY, espaceSSKEY, 0);
32}
```

Listing N.2: Self FK treatment

Appendix O

Optimizations

O.1 Mask Object

The object constructor, listed in O.1, receives the SQL source connection, the migrationMasterId and the entityId and sets the received parameters as the Mask object attributes. It also creates a new AttributeMask Hashtable to store the data fetched from the database.

```
1  public Mask(SqlConnection sqlConnection, long migrationMasterId, long entityId)
2  {
3      MigrationMasterId = migrationMasterId;
4      EntityId = entityId;
5      AttributeMask = new Hashtable();
6
7      SqlCommand cmd = new SqlCommand("select a.ATTRIBUTENAME, a.ISMASK FROM
           OSUSR_m7a_MaskAttribute a " +
8      "WHERE a.MigrationMasterId = " + migrationMasterId.ToString() + " " +
9      "AND a.ENTITYID = " + entityId.ToString(), sqlConnection);
10
11       SqlDataReader reader = cmd.ExecuteReader();
12       while (reader.Read())
13       {
14         AttributeMask.Add(reader["ATTRIBUTENAME"].ToString().ToUpper(), Convert.
               Toboolean(reader["ISMASK"]));
15     }
16 }
```

Listing O.1: Mask constructor

In the Migrate function, a new Mask object is created for each entity, given the entity identifier and the migration plan identifier. Additionally, if the entity does not have any attribute to be masked, is not really necessary to check if the current column is set to mask. For this, another function, listed in O.2, was created inside the Mask class to check if the entity has any isMask attribute set to true. This function iterates all Hastable Values (isMask booleans) and checks if exist **at**

least one true Value, in this case it returns immediately true otherwise the function returns false. This function is called before iterating the records of an entity and the result is stored into a local boolean: hasMask. Then, for each column, this boolean is checked and if true the Value of the current attribute's name is fetched through this expression *mask.AttributeMask[reader.GetName(j)]* where *reader.GetName(j)* correspond to the current attribute name of the entity being read by the SqlDataReader. Otherwise if the entity does not have any attribute to mask (local variable hasMask is false) then the Hastable is not even consulted, avoiding unnecessary mask searches.

```
1  public bool IsAnyMaskSetToTrue()
2  {
3   // To get the values alone, use the Values property.
4     ICollection valueColl = AttributeMask.Values;
5
6      foreach (bool isMask in valueColl)
7      {
8          if (isMask) //if we find at least one true then its not empty
9          {
10         return true;
11         }
12     }
13     return false;
14 }
```

Listing O.2: Function to check if the Mask object has attributes to mask

O.2 Dependency Object

The object constructor, listed in O.3, receives the origin SqlConnection and a physical table name and fills the Hashtable with database data.

The constructor runs a SQLCommand query that uses the table name to get all it's attributes names and types. For each attribute on the query's result, another query to get the dependencies identifiers is called by using the SS_KEYs present on the reader["Type"]. This last adds the result (FK) as the Value of the respective Key (attribute) to the Hashtable.

```
1  public Dependency(SqlConnection sqlConnection,  string tableName)
2  {
3      TableName = tableName;
4      AttributeDependencyId = new Hashtable();
5
6      SqlCommand cmd = new SqlCommand("select a.Name, a.Type from ossys_Entity_Attr a
           , ossys_entity e "+
7                  "where a.ENTITY_ID = e.ID AND e.PHYSICAL_TABLE_NAME = '"+tableName+
                       "' AND a.TYPE LIKE 'bt%'", sqlConnection);
8
```

```
9     SqlDataReader reader = cmd.ExecuteReader();
10    while (reader.Read())
11    {
12        SqlCommand cmd2 = new SqlCommand("SELECT e.ID FROM ossys_entity e,
              ossys_espace s " +
13        "WHERE  e.ESPACE_ID = s.ID " +
14        "AND e.SS_KEY = (select right('"+reader["Type"]+ "', charindex('*', reverse
              ('" + reader["Type"] + "')) - 1)) "+
15         \\... dependency query continuation
16         long id = Convert.ToInt64(cmd2.ExecuteScalar());
17
18          if (id!=0)
19          {
20            AttributeDependencyId.Add(reader["Name"].ToString().ToUpper(), id);
21          }
22    }
23 }
```

Listing O.3: Dependency constructor

In the migration, the implementation was done by creating one Dependency object before iterating the records of the entity. Now, to check if the entity has dependencies it is only necessary to create a local variable called hasDependencies and use the object to check if the Hashtable has values, through: *dependencies.AttributeDependencyId.Count!=0*.

The rest of the logic remains the same but instead of verify if there is any dependency on the dependencies list, this boolean (hasDependencies) is verified. The biggest step of this optimization is done by removing the function called FindFKByColumnName, this function was the one calling the database, in every column, to check if the current entity had dependencies and what were those. But now, all attributes of a given entity are fetched only one time (before iterating the entity records) and the information about this attributes that are FK are all in the local object. The new algorithm just check if the column is a dependency with one local call to the Hashtable, like: *!dependencies.AttributeDependencyId.ContainsKey(reader.GetName(j))*, where *dependencies* is the object of the Dependency class. The search in the dependencies list was also removed since the Hashtable already contains all dependencies information, i.e., attribute name and entity identifier. To get the Value (entity identifier) of the current column the expression was reduced to: *dependencies.AttributeDependencyId[reader.GetName(j)]*.

O.3 Bulk Insert

The Bulk Insert mechanism can receive a DataTable (.NET class that is a database table representation and provides a collection of columns and rows to store data in a grid form) or a C# Entity (relate to the database entity as to a set of objects). To this solution, a DataTable object was created since it is easier manipulate parts of the DataTable using DataRow and DataColumn and has a similar form to the database tables.

The string INSERT query E.5 used until now disappears, and it is replaced with the DataTable. The data is loaded into a local DataTable object while reading the source records and perform some necessary transformations. The SQLDataReader only stores the original data and can not be used to perform the bulk insert. When the reading ends, then this local object is consumed by the bulk insert and copied to the target database. The migration data is manipulated locally, which allows to improve the performance.

To update the RecordIdRelation the bulk has to return the identity, to do this the code must specify the **AutoMapOutputIdentity** attribute of the Bulk Operations class to true and the primary key of the table ("ID") should be mapped as the Output. When the bulk insert is done, then the "ID" column of the DataTable will be filled with the target identifiers and this value can then be used to update the RecordIdRelation table. The Bulk should also define the target physical table name of the database, as listed on O.4. A new DataTable object is created for each entity in the queue.

```
1 BulkOperation bulk = new BulkOperation(target);
2 bulk.DestinationTableName = TargetPhysicalTableName;
3 bulk.ColumnMappings.Add("ID", ColumnMappingDirectionType.Output);
4 bulk.AutoMapOutputIdentity = true;
```

Listing O.4: Bulk Operations Config

Building the DataTable object requires knowing the schema of the source and target table. The source entity's schema is retrieved with the SQLDataReader but the target schema must be fetched as well, this can be done by also creating a SQLDataReader to the target database and use the method *GetSchemaTable()*, as listed on O.5.

```
1  DataTable dtSchema = reader.GetSchemaTable();
2  DataTable dt = new DataTable();
3  SqlCommand targetSchema = new SqlCommand("SELECT * FROM "+TargetPhysicalTableName,
       destiny);
4  SqlDataReader targetSchemaReader = targetSchema.ExecuteReader(CommandBehavior.
       SchemaOnly);
5  DataTable dtSchemaTarget = targetSchemaReader.GetSchemaTable();
6
7  if (dtSchema != null && dtSchemaTarget.Rows.Count >0)
8  {
9      foreach (DataRow drow in dtSchema.Rows)
10     {
11         string columnName = System.Convert.ToString(drow["ColumnName"]);
12         DataColumn column = new DataColumn(columnName, (Type)(drow["DataType"]));
13
14         var r = dtSchemaTarget.Select("ColumnName = '" + columnName + "'").
               SingleOrDefault();
15
```

```
16          if (r!=null) //if the columns exists in the destiny
17          {
18              dt.Columns.Add(column);
19              if (!columnName.Equals("ID"))
20              {
21                bulk.ColumnMappings.Add(columnName, columnName);
22              }
23          }
24      }
25  }
26  targetSchemaReader.Close();
```

Listing O.5: DataTable creation

The source and target schemas may have different number of columns, which can be a problem since the iteration of columns is done following the source schema. When trying to insert data into the target to a nonexistent column an error occurs without even start the actual migration process. The solution compares the columns in the source and target schemas and only adds the target columns into the local DataTable object.

The table columns **DataColumns** are created and added to the DataTable object, consulting both schemas. Then it is still necessary to map the columns of the DataTable class to the columns of the target database table, which in this case, have the same column names. The primary key column ("ID") is ignored since it was already mapped to *ColumnMappingDirectionType.Output*.

New **DataRows** are created for each row of the source (*while(reader.Read())*) and added to the DataTable after reading and transforming all columns. The DataRow uses the current column name to store the value, whereas, before the optimization, the value was added to one list to be later converted into the string query. As presented on O.6, several lines used to build the string query were replaced with the DataTable object, commented lines represent the code before implementing Bulk Operations.

```
1  DataRow dataRow = dt.NewRow();
2  //...
3  //if (columnTypeTransformationToString(reader.GetDataTypeName(j))) // string
       especial treatment
4  //{
5  //    list.Add("'" + reader.GetValue(j).ToString() + "'");
6  //}
7  //else
8  //{
9  //    list.Add(reader.GetValue(j).ToString());
10 //}
11 dataRow[reader.GetName(j)] = reader.GetValue(j); // new assignment
12 //...
13 dt.Rows.Add(dataRow);
```

Listing O.6: DataTable rows

Some changes had to be implemented for the Insert into the data mapping table. Inserting multiple rows into the target all at once does not allow to insert the old and new record identifier into the mapping table directly. The most practical solution was decompose the INSERT into two steps: an INSERT and an UPDATE. First, INSERT the data available (migrationMasterID, entityID and oldRecordID), while reading the source record and then, after the bulk insert (after reading one entity), iterate the ID column of the DataTable and UPDATE the mapping table with this value - newRecordID, represented by *row["ID"]*, on O.7. To fetch the correct RecordIdRelation record a function called *getMap* gets the first record of the current entity's migration plan, this is then incremented while iterating the DataRows to update the data mapping records respectively. Now, the log is updated at the end of each entity, using a variable called count to store the current row index, in this case, stores the last index of the table.

```
1  bulk.BulkInsert(dt);
2  UpdateCurrentRow(source, stagingId, entitySSKEY, espaceSSKEY, count);
3  long startId = getMap(source, entityID, stagingId);
4
5  foreach (DataRow row in dt.Rows)
6  {
7      UpdateRecordRelationTable(source, Convert.ToInt64(row["ID"]), startId);
8      startId++;
9  }
10
11 dt.Clear();
```

Listing O.7: Bulk Insert

With this modifications, the same sample migrating in 40 min now migrates in 29 minutes (around 11 minutes faster). However, after some tests, it was detected that the algorithm took a lot of time performing the UPDATE SQL calls into the mapping table. This is mainly because before this modification there was only one INSERT call to the database and now there is two (insert and update) calls for each row of each data table, which potentially grows the latency of the migration. In this solution, the DataTable has the size of the original source table, which can be a problem if the source has millions of rows. To agile the process, a new constant (X) was added to the program, performing Bulk Inserts from X in X rows and clearing the DataTable every time a bulk insert is done (in every X). Clearing the local object allows to migrate in incremental steps and obtain better performance because manipulates smaller blocks of data.

The constant chosen was 200,000 rows, which means that one entity with less than this will perform the Bulk Insert at the end of reading it's records, but entities with more than this will perform Bulk Inserts between 200,000 in 200,000 rows and also at the end of the entity. For example, an entity with 420,897 rows, will perform three bulk inserts, one at the row number 200,000, another at the row 400,000 (200,000 + 200,000) and another at the 420,897 (the rest of the records - 20,897). The same logic for performing a Bulk Insert is added with the condition of

the current row being between constant rows, like in O.8.

```
const int amoutOfRowsPerBulk = 200000;
int x = amoutOfRowsPerBulk;

while (reader.Read())
   count++;
//...
//After reading all entity rows
if (count == x)
{
//same bulk insert logic
long startId1 = getMap(source, entityID, stagingId);
    if (count > amoutOfRowsPerBulk)
     {
         startIndex += count;
     }
x += amoutOfRowsPerBulk;
```

Listing O.8: Bulk Insert from X in X rows

This new addition set the migration time, for the same sample, to 25 minutes, 4min faster then before.

O.4 Bulk Insert into data mapping table

To use the data mapping table locally a new C# was created, called **DataMap**, which contains the migration plan, entity identifiers and a DataTable. The constructor of the class initializes the DataTable with the same columns used by the RecordIdRelation database table and uses a migrationMasterID and entityID (input) to fill the object attributes. Inside this class, the following functions were also implemented: Insert, Update and BulkCopy.

The Insert method listed in O.9, on DataMap.cs, has the old record ID as input and creates a new DataRow with all the information, except the new record ID. Finally, it adds the DataRow to the DataTable.

```
public void Insert(long oldRecordId)
{
    DataRow dataRow = dt.NewRow();
    dataRow["MIGRATIONMASTERID"] = migrationMasterId;
    dataRow["ENTITYID"] = entityId;
    dataRow["OLDRECORDID"] = oldRecordId;
    dt.Rows.Add(dataRow);
}
```

Listing O.9: Insert into DataMap (local object)

The Update method listed in O.10 uses the new record ID and a DataRow as input to update the column NEWRECORDID of the DataTable by assigning the ID to the *dataRow["NEWRECORDID"]*.

```
public void Update(long newRecordId, DataRow dataRow)
{
    dataRow["NEWRECORDID"] = newRecordId;
}
```

Listing O.10: Update DataMap

The last method, BulkCopy listed in O.11, uses a SqlConnection (source) to create a SqlBulkCopy object. It copies all rows in the supplied DataTable to a target table specified by the bulkCopy.DestinationTableName property of the object. The first function tests retrieved the error message: "Bulk Insert with another outstanding result set should be run with XACT_ABORT on". This issue occurs because SQL Server incorrectly detects a MARS deadlock. The addition of the SQLCommands to turn on and off the XACT_ABORT property solved the problem.

```
public void BulkCopy(SqlConnection sql)
{
    using (SqlBulkCopy bulkCopy = new SqlBulkCopy(sql))
    {
        bulkCopy.DestinationTableName = "OSUSR_m7a_RecordIDRelation";
        // How many records to send to the database in one go (all of them)
        bulkCopy.BatchSize = dt.Rows.Count;

        SqlCommand on = new SqlCommand("SET XACT_ABORT ON",sql);
        on.ExecuteNonQuery();

         // Load the data to the database
         bulkCopy.WriteToServer(dt);

        SqlCommand off = new SqlCommand("SET XACT_ABORT OFF", sql);
        off.ExecuteNonQuery();

        // Close up
        bulkCopy.Close();
    }
}
```

Listing O.11: SQLBulkCopy into Mapping table

In this case, the SqlBulkCopy built-in mechanism was used instead of the BulkInsert package because it is a simple insert, without the need to retrieve the identity of the inserted values.

On the main class, the DataMap object is created for each entity. The methods that call the database to insert and update the RecordIdRelation are replaced with the Insert and Update method of the object. The getMap method that call the database to retrieve the first identifier inserted for

that migration plan's entity is deleted since now the position will start always in the beginning of the DataTable, because this local DataTable table will be deleted after each Bulk Copy. Cleaning the table after each bulk operation guarantees that new records inserted will start again from the position 0. The algorithm increments this position (startIndex) while updating the DataTable, row by row. An excerpt of the code is presented in O.12, where the *dm* is the instance of the DataMap object and *dt* is the DataTable instance used to bulk insert the entities. Both *dt* and *dm* should be cleared at the end of a bulk insert to match the number of rows in the following bulk inserts.

```
1  bulk.BulkInsert(dt);
2  int startIndex = 0;
3
4  foreach (DataRow row in dt.Rows)
5  {
6    dm.Update(Convert.ToInt64(row["ID"]), dm.dt.Rows[startIndex]);
7    startIndex++;
8  }
9
10 dm.BulkCopy(source);
11 dm.dt.Clear();
12 dt.Clear();
```

Listing O.12: Update Mapping table

O.5 Local data mapping table object

The LocalMap class constructor receives the source SQLConnection and a migration master plan identifier and fetches all records from the RecordIdRelation that match the migration master plan, using a SqlDataAdapter to fill the DataTable with this information, as listed in O.13.

```
1  public LocalMap(SqlConnection conn, long MigrationMasterID)
2  {
3       migrationMasterID = MigrationMasterID;
4       sql = conn;
5       SqlCommand cmd = new SqlCommand("select * from OSUSR_m7a_RecordIDRelation
           where MIGRATIONMASTERID = " + migrationMasterID.ToString(), sql);
6
7      // create data adapter
8      SqlDataAdapter da = new SqlDataAdapter(cmd);
9      // this will query your database and return the result to your datatable
10     da.Fill(dataTable);
11
12     da.Dispose();
13 }
```

Listing O.13: LocalMap constructor

The LocalMap object is created at the same time as the DataMap, for each entity, before iterating it's records. The creation of a new object to every entity is essential because the algorithm is storing the information locally, and after each entity the RecordIdRelation is updated and the next entity would not have access to the new inputs.

Additionally, another optimization is inherent since the DataTable is only filled with information from the current migration plan, what reduces the size of the DataTable compared to the size of the database RecordIdRelation, that can contain more information regarding other migration plans.

The GetNewId method, represented on O.14, uses the same input as the GetRelationId method, the migration plan identifier, entity identifier, old record identifier and two variables that flag the User migration. If the entity being migrated is the User entity then the method return -1.

```
1  public long? GetNewId(bool isMigrateUsers, long userEntityId, long entityId, long
       oldRecord)
2  {
3      if (!isMigrateUsers && entityId == userEntityId) // SYSTEM TABLE USER
4      {
5           return -1;
6      }
7
8      var r = dataTable.Select("ENTITYID=" + entityId.ToString() + " AND OLDRECORDID=
           " + oldRecord.ToString()).SingleOrDefault();
9
10     if (r == null || r.IsNull("NEWRECORDID"))
11     {
12         return null;
13     }
14
15    return Convert.ToInt64(r["NEWRECORDID"]);
16 }
```

Listing O.14: LocalMap GetNewId function

To find a single record of the DataTable the Select method was used, that is similar to a SQL SELECT. The Select gets an array of all DataRow objects that matches the filter criteria, in this case, that match the entity identifier and the old record identifier. After that, the value stored at the column name: NEWRECORDID, if not null, is retrieved.

Appendix P

Evaluation Details

P.1 Migration of RocketContactLenses' data using BirdlyData

The use of BirdlyData to create and run a migration plan for RocketContactLenses' data is documented on the following figures. To simulate a migration, a new migration plan was created, with a description and both Empty Destiny (target) and Migrate Users flags selected, as shown on Figure P.1.

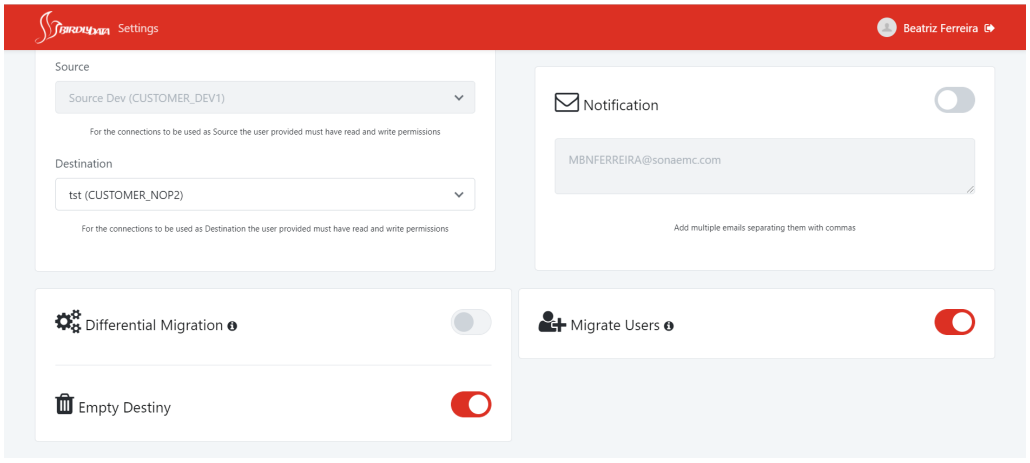


Figure P.1: Creation of a new Migration Plan for RocketContactLenses migration

After selecting the RocketContactLenses application from the several available applications, the RocketContactLenses_CS module (the only module of the app) was also selected. With this information, the tool is able to retrieve the application's entities. Due to the automatic dependency detection feature, selecting the Lens entity also selects the entities in which this one depends, which are: LensType, Brand, VendorName and User. For this test scenario, the entity LensTypeAttribute was also selected, this UI is demonstrated on Figure P.2.

With all the important data selected, BirdlyData presents the Migration Plan Overview Screen that stores all information about the planned migration, here the entities are shown by ordered of

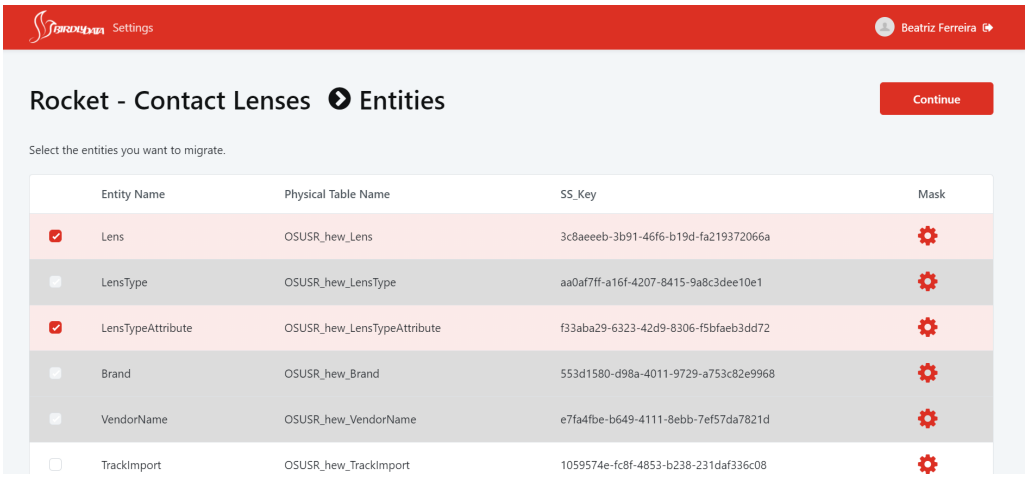


Figure P.2: Selecting RocketContactLenses' entities to migrate

dependencies, being the first ones of the list independent entities, following by the entities that depend on the previous ones of the same list. In this particular case, User should be the first entity since all other entities depend on it and the next entity on the list should only depend on User. The Lens entity is the last because it depends on the other entities (except LensTypeAttribute that could also be the last since it only has to be next it's dependency: LensType). The ordered entities information results provided by the data migration tool interface are on Figure P.3.

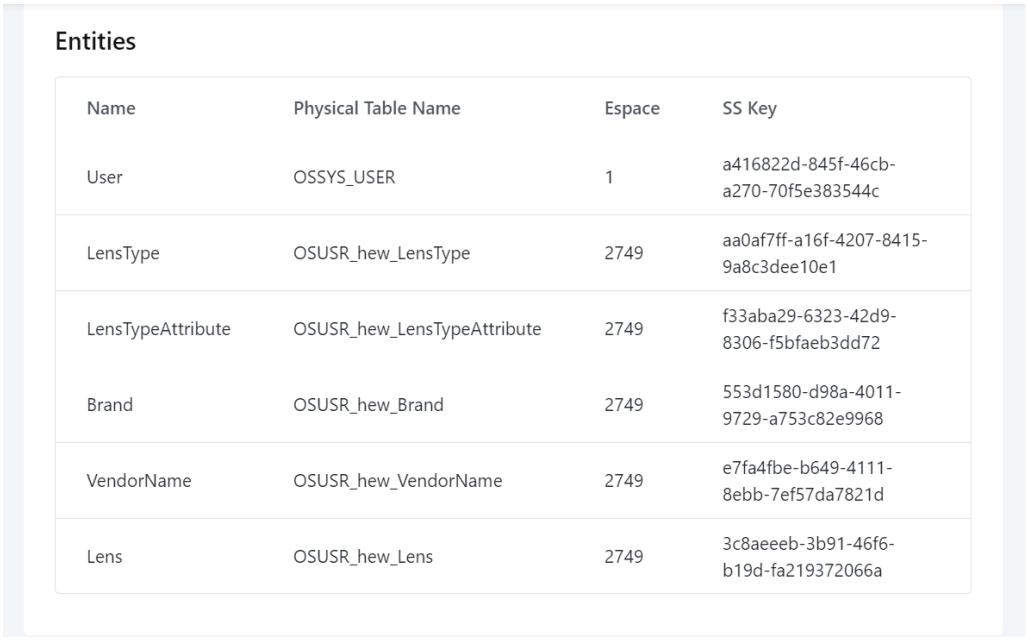


Figure P.3: Migration Plan Overview of RocketContactLenses

Finally, the migration process starts, taking 5 seconds to migrate 11674 records plus the system table User records (3). Any error on the migration process would appear on the right side of the

entity space and the migration would be set as Finished. The Figure P.4 represents the real-time Screen Log of the migration process.

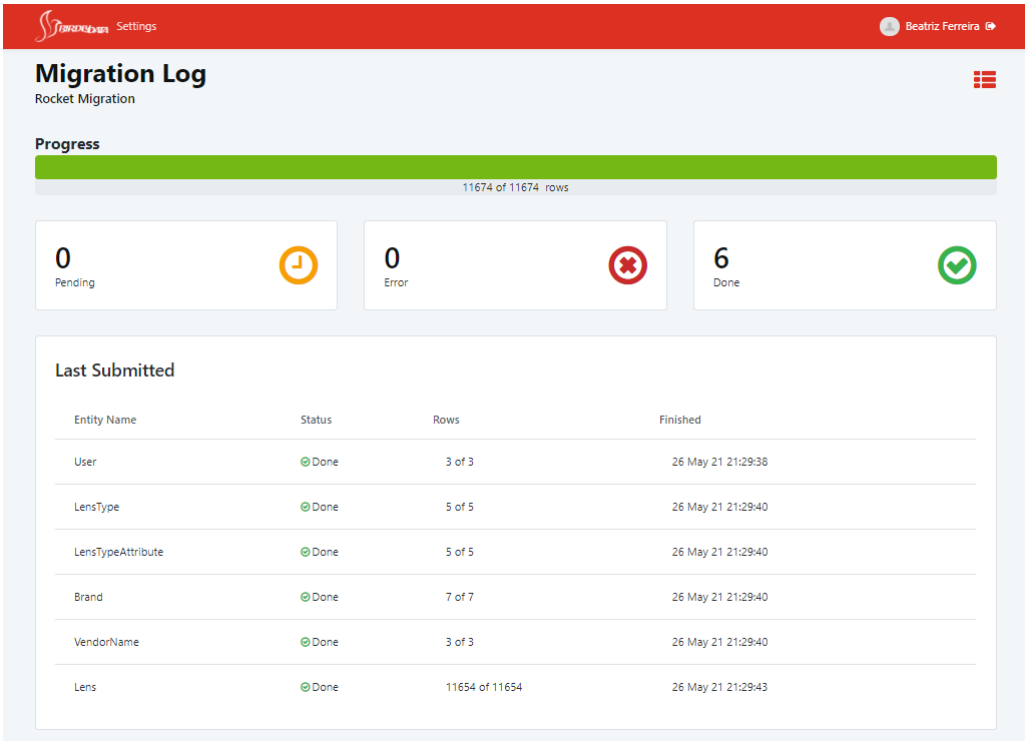


Figure P.4: Migration of RocketContactLenses

After running the tool, the data from the source environment is available in the target environment as well.

P.2 Correctness of remaining entities

The comparison results of the other two smaller entities migrated are represented by Figure P.5 and Figure P.6, where the first Figure portrays the VendorName entity and the second the LensType entity. Despite some changes in the columns order, the migrated values are the same for every record, while the User FKs point to the same User object, as already proven.

The remaining 4 datasets used to test the migrated data of the Lens entity, are grouped by the LensTypes: *Tórica*, *Multifocal*, *Tórica Multi-Focal* e *Colorida*. The logic of the implemented correctness tests was the same as the *Esférica* Type, a random record from each group is selected and compared with the source database. An example of another results' comparison with a record from the group:*Tórica Multi-Focal* can be found on Table P.1. In this group there was only one existent record, so it was not necessary to randomly select one record.

Following the business rule, lenses of the Type:*Tórica Multi-Focal* should have all their attributes (POWER, DIAMETER, BENDRADIUS, CYLINDERPOWER, AXIS and ADDITION) filled, i.e., with valid values and not the default/null value, the results demonstrate the validation of

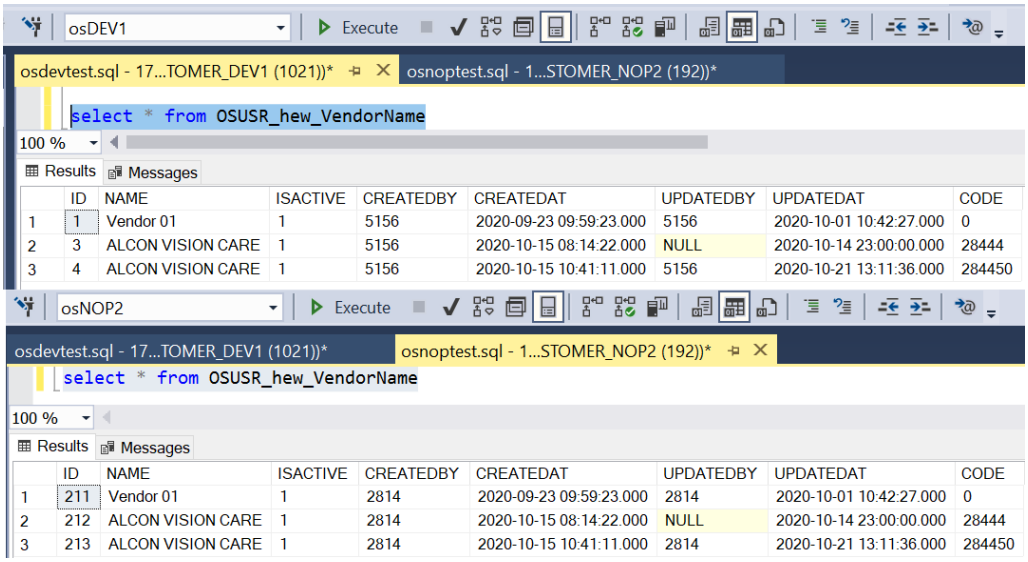


Figure P.5: Compare VendorName records after migration

this rule and accuracy of the data. The columns that represent dependencies to other entities were also checked in detail and the references represented correct mappings between environments.

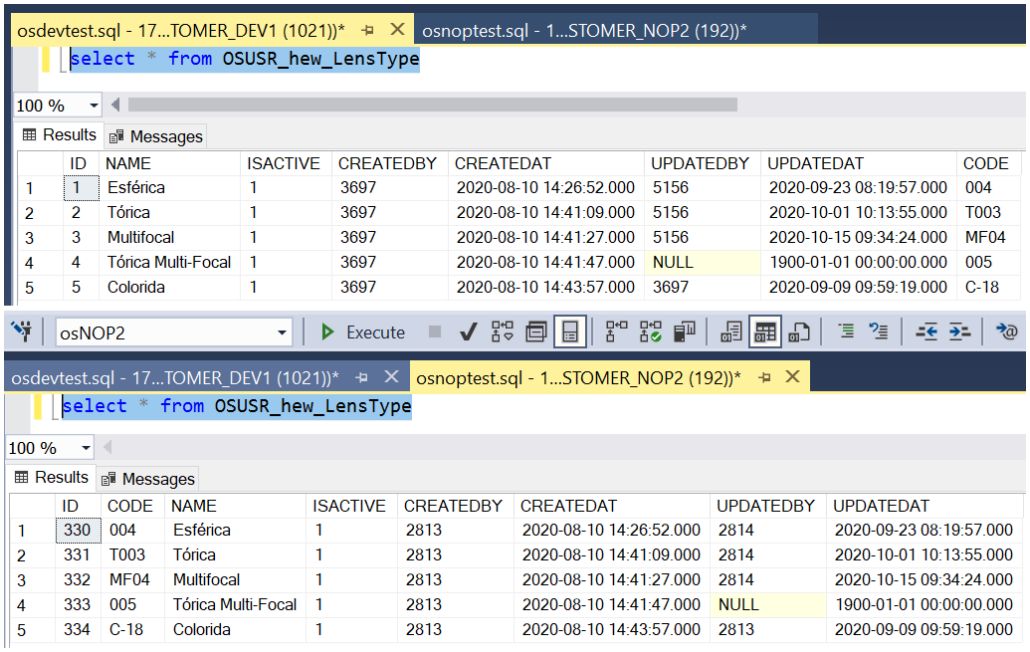


Figure P.6: Compare LensType records after migration

Table P.1: Comparison of environments results by Lens Type: *Tórica Multi-Focal*

Lens' Record		
	Source(dev)	Target(tst)
ID	5255	327704
LENSTYPEID	4	333
REPLACMENTTYPEID	1	1
SKUSONAE	1111	1111
EAN	121212121212	121212121212
COMPOSITIONS	*empty*	*nonexistent column*
ISACTIVE	0	*nonexistent column*
CREATEDBY	5156	2814
CREATEDAT	2020-10-02 12:30:00.000	2020-10-02 12:30:00.000
UPDATEDBY	5156	2814
UPDATEDAT	020-12-18 14:43:12.000	2020-12-18 14:43:12.000
POWER	22.00	22.00
CYLINDERPOWER	-1.00	-1.00
AXIS	2	2
BRENDRADIUS	0.00	*nonexistent column*
ADDITION	1	1
DIAMETER	14.00	14.00
COLOR	*empty*	*nonexistent column*
VENDORTYPECODE	*empty*	*nonexistent column*
LENSCOLORID	NULL	NULL
COMPOSITION	*empty*	*empty*
CAPACITYID	NULL	NULL
CREATIONBYIMPORT	1	1
BENDRADIUS	8.60	8.60
ITEMDESCRIPTION	*empty*	*empty*
BRANDID	3	323
VENDORNAMEID	1	211
VPN	*empty*	*empty*

Appendix Q

Migration of Static Entities

The same mechanism implemented to find an entity dependencies was used, but now with the possibility of searching for regular entities or static entities. For each static dependency found a new local object is created to manage static entities.

An auxiliary Hashtable was created to store the entity ID as Key and the object LocalStaticEntity as Value. LocalStaticEntity stores both source and target data tables, between other attributes, with this information is possible to make a comparison between records to retrieve the target matcher of a given source identifier.

First, two DataTables from source and target static entities, are filled with the database data, using SqlDataAdapter. Then, while migrating, the source record's identifier is transformed to the target's, i.e., it is possible to discover the target identifier of the same value, for quantitative static entities, the value that is compared is the VALUE, but for qualitative static entities the column compared is LABEL. The function getDestinyValue receives an object which is the value read by the SLQDataReader and searches for the same value in both databases, retrieving the one that corresponds to the target database or null if there is no correspondence. There is also a secondary function to verify if the static entity is qualitative or quantitative.

An excerpt of the getDestinyValue function for qualitative static entities is listing on Q.1, where *destinyStaticEntity* is the target DataTable and the *sourceStaticEntity* is the source DataTable stored on the object LocalStaticEntity.

```
1 DataRow Sdr = sourceStaticEntity.Select("ID = " + value.ToString()).SingleOrDefault
      ();
2
3 if ((bool)isLabel) {
4     var r = destinyStaticEntity.Select("LABEL= '"+ Sdr["LABEL"].ToString() + "'").
          SingleOrDefault();
5     if (r == null)
6     {
7         return null;
8     }
9     else
```

```
10      {
11       DataRow Ddr = (DataRow)r;
12       return Convert.ToInt64(Ddr["ID"]);
13      }
14  }
```

Listing Q.1: GetDestinyValue of Static Entity

If any static entity was detected, then the mechanism searches inside the auxiliary Hashtable to find the correct object and call the getDestinyValue function, as in Q.2. The value of the target FK is saved into the DataTable that is responsible to insert the data into the target database. This feature benefits from the already created data structures to manipulate the dependencies and the local DataTables since it was implemented after the optimizations stage.

```
1   if (hasStaticDependencies)
2   {
3      if (dependenciesStatic.AttributeDependencyId.ContainsKey(reader.GetName(j)))
4      {
5           if (reader.GetValue(j) == DBNull.Value)
6           {
7              dataRow[reader.GetName(j)] = DBNull.Value;
8           }
9           else
10          {
11             long staticId = Convert.ToInt64(dependenciesStatic.
                   AttributeDependencyId[reader.GetName(j)]);
12             LocalStaticEntities localStatic = (LocalStaticEntities)staticHash[
                   staticId];
13
14             var destinyStaticId = localStatic.getDestinyValue(reader.GetValue(j));
15
16             if (destinyStaticId == null)
17             {
18                 dataRow[reader.GetName(j)] = DBNull.Value;
19             }
20             else
21             {
22                 dataRow[reader.GetName(j)] = Convert.ToInt64(destinyStaticId);
23             }
24          }
25     }
26  }
```

Listing Q.2: Call GetDestinyValue