DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

NUNO ANDRÉ DE ALMEIDA E SILVA

BSc in Science and Computer Engineering

LIVE PROGRAMMING IN LOW-CODE PLATFORMS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

LIVE PROGRAMMING IN LOW-CODE PLATFORMS

NUNO ANDRÉ DE ALMEIDA E SILVA

BSc in Science and Computer Engineering

Adviser: Doutor João Ricardo Viegas da Costa Seco
Associate Professor, NOVA School of Science and Technology

Co-adviser: Hugo Miguel Ramos Lourenço
Distinguished Architect, OutSystems

Examination Committee

Chair: Doutor Henrique João Lopes Domingos
Associate Professor, NOVA School of Science and Technology

Rapporteurs: Doutora Maria Antónia Bacelar da Costa Lopes
Associate Professor, Faculdade de Ciências da Universidade de Lisboa

Doutor João Ricardo Viegas da Costa Seco
Associate Professor, NOVA School of Science and Technology

Co-adviser: Hugo Miguel Ramos Lourenço
Distinguished Architect, OutSystems

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

Live Programming in Low-Code Platforms

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.11) [7].

Acknowledgements

I want to thank my thesis advisor, Associate Professor João Costa Seco, for his trust in allowing me to do my thesis at OutSystems and for his valuable advice throughout this process. I also want to express my deepest gratitude to my OutSystems advisor, Hugo Lourenço, for his guidance and for always being available to help me. This work would not have been possible without their support.

I want to thank Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa for the amazing past few years as a university student and OutSystems for the opportunity to work on such an interesting and engaging topic.

Last but certainly not least, I want to thank all my friends and family who make life worth living. Special thanks to Daniel and Henrique, who have supported me closely throughout this past year, and to Rita, my mother and father, my brother, and Zila. I love you all.

Abstract

The OutSystems Platform is a visual model-driven development and delivery platform that allows developers to create enterprise-grade web and mobile applications, using Service Studio, the platform's IDE. Low-code development platforms, such as this, aim to improve the agility of the development process, by allowing developers to prototype, test and deploy applications quickly. Compared to other technologies, OutSystems makes it faster to create applications with a modern design without the need to write a single line of code.

However, even in low-code platforms, the develop-compile-test pipeline is explicit, and the experience can sometimes be slow and cumbersome. This limitation becomes more relevant when it is necessary to make many small, iterative changes to an application. This is the case when a developer is fine-tuning the UI, for example. After a developer makes changes to an application, they are required to publish it before being able to see the results. It includes, among other tasks, the transportation of the model and its compilation, that takes place in the OutSystems Platform Server. This process takes a non-negligible amount of time, which breaks workflow fluidity and negatively impacts the developer's experience.

That is the problem we aimed to solve in this work, focusing on the saving, transportation, and loading of the model. To achieve this, we explore two approaches. In the first approach we propose that the IDE and the Compiler share the same model, eliminating the need for transporting and loading it in every publication. One of the challenges that it poses is one of concurrency, since the Compiler needs to be able to see the model while the developer may be changing it in the IDE. The second approach is simpler and consists of only transporting and loading what was modified, instead of the entire model. We developed a prototype which shows that our approaches have the potential to reduce the duration of these operations, reducing the Publication time and improving the development experience in OutSystems.

Keywords: Live programming, Compilation, OutSystems, Low-Code, Model driven development

Resumo

A Plataforma OutSystems é uma plataforma visual de desenvolvimento e entrega baseada em modelos que permite aos programadores criar apicações móveis e web, utilizando o Service Studio, o IDE da plataforma. Plataformas de desenvolvimento low-code, como esta, visam melhorar a agilidade do processo de desenvolvimento, permitindo aos programadores prototipar, testar e implementar aplicações rapidamente. Comparando com outras tecnologias, OutSystems torna mais rápida a criação de aplicações com um design moderno sem a necessidade de escrever uma única linha de código.

No entanto, mesmo em plataformas low-code, o pipeline de desenvolvimento-compilação-teste é explícito, e a experiência pode por vezes ser lenta e desagradável. Esta limitação torna-se mais relevante quando é necessário fazer várias alterações pequenas e iterativas a uma aplicação. É o caso quando um programador está a ajustar a interface de utilizador, por exemplo. Após um programador fazer alterações a uma aplicação, é necessário publicá-la antes de ser possível ver os resultados. Isto inclui, entre outras tarefas, o transporte do modelo e a sua compilação, que ocorre no OutSystems Platform Server. Este processo leva um tempo não desprezível, o que quebra a fluidez do workflow e tem um impacto negativo na experiência do programador.

Este é o problema que pretendemos resolver neste trabalho, focando-nos nas operações de guardar, transportar e carregar o modelo. Para isso, explorámos duas abordagens. Na primeira abordagem propomos que o IDE e o Compilador partilhem o mesmo modelo, eliminando a necessidade de o transportar e carregar em cada publicação. Um dos desafios que isso coloca é um desafio de concorrência, pois o Compilador precisa de poder ver o modelo enquanto o programador pode estar a modificá-lo no IDE. A segunda abordagem é mais simples e consiste em transportar e carregar apenas o que foi modificado, em vez do modelo completo. Desenvolvemos um protótipo que demonstra que as nossas abordagens têm o potencial de reduzir a duração destas operações, reduzindo o tempo de Publicação e melhorando a experiência de desenvolvimento em OutSystems.

Palavras-chave: Programação em tempo real, Compilação, OutSystems, Low-Code, Desenvolvimento basedado em modelos

Contents

List of Figures viii

List of Tables x

Listings xi

Acronyms xiii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Motivation . . . 1
- 1.3 Objectives . . . 2
- 1.4 Contributions . . . 2
- 1.5 Document Structure . . . 2

2 State of the Art 4
- 2.1 Live Programming . . . 4
- 2.2 OutSystems Language Elements . . . 7
  - 2.2.1 eSpace . . . 7
  - 2.2.2 Entity . . . 7
  - 2.2.3 Action . . . 7
  - 2.2.4 UI Flow . . . 7
  - 2.2.5 Screen . . . 8
  - 2.2.6 Aggregate . . . 8
  - 2.2.7 Widget . . . 8
  - 2.2.8 Client Variable . . . 8
- 2.3 Developing a simple Task List application in OutSystems . . . 8

3 Related Work 12
- 3.1 Other tools . . . 12
  - 3.1.1 React . . . 12

3.1.2 ASP.NET . . . 16
3.2 Persistent Data Structures . . . 17

4 OutSystems Platform **19**
4.1 Architecture of the OutSystems Platform . . . 19
4.2 OutSystems Application . . . 20
4.3 OutSystems 11 Publication Process . . . 20
4.3.1 Compilation Units . . . 21
4.3.2 Publication Steps . . . 21
4.4 Lazy Loading . . . 29
4.5 Developer Workflow . . . 32

5 Proposed Solutions **33**
5.1 Shared Model . . . 33
5.2 Keeping the Compiler running . . . 36

6 Prototype Implementation **37**
6.1 Meta-model . . . 37
6.1.1 IESpace and ESpaceClass . . . 41
6.1.2 IUIFlow and UIFlow . . . 42
6.1.3 IUIFlowNode, IScreen and Screen . . . 42
6.1.4 IWidget . . . 42
6.2 Compiler . . . 43
6.2.1 Compilation Summary . . . 43
6.2.2 React and Vite . . . 43
6.2.3 Artifact creation . . . 44
6.3 Shared model . . . 49
6.3.1 Versioning solution integration . . . 49
6.3.2 IDE console app . . . 51
6.3.3 Compiler concurrency . . . 52
6.4 Version serialization . . . 58
6.4.1 Serialization . . . 58
6.4.2 Deserialization . . . 68
6.4.3 Transporting versions . . . 72

7 Results **74**
7.1 First approach . . . 75
7.1.1 No changes . . . 75
7.1.2 1 Change in 1 Screen . . . 79
7.1.3 Many changes in 1 Screen . . . 81
7.1.4 1 Change in all Screens . . . 83
7.1.5 Many changes in all Screens . . . 85

7.1.6 Uploading the model . . . 87
7.1.7 Conclusions . . . 87
7.2 Second approach . . . 87
7.2.1 No changes . . . 88
7.2.2 1 Change in 1 Screen . . . 89
7.2.3 Many Changes in 1 Screen . . . 90
7.2.4 1 Change in all Screens . . . 93
7.2.5 Many Changes in all Screens . . . 94
7.2.6 Conclusions . . . 95

8 Conclusions **96**
8.1 Future Work . . . 97

Bibliography **98**

Webography **100**

Appendices

A OutSystems Platform Compiler **103**

B Results tables **113**

List of Figures

2.1 Task Entity creation . . . 9
2.2 Task list screen preview . . . 10
2.3 Task list screen in the browser . . . 10
2.4 Task creation/editing screen preview . . . 11
2.5 Client Action . . . 11

3.1 Task list screen created with React . . . 15
3.2 Task editing screen created with React . . . 15

4.1 OutSystems Platform Server's Architecture . . . 20
4.2 OutSystems Publication Process . . . 22
4.3 Artifacts produced by Compilation Units . . . 25
4.4 Developer workflow . . . 32

5.1 Shared Model approach . . . 34
5.2 Example application model with one version . . . 35
5.3 Example application model with multiple versions . . . 35

6.1 Meta-model class diagram . . . 38
6.2 Example app created with our prototype . . . 41
6.3 Shared model IDE console app screenshot . . . 51
6.4 Example application with different versions . . . 58
6.5 Versioning solution's data structures . . . 59

7.1 Publication steps . . . 74
7.2 Publication steps with the first solution . . . 75
7.3 Save time vs. Screen count (no changes) . . . 77
7.4 Save time vs. Size (no changes) . . . 77
7.5 Load time vs. Screen count (no changes) . . . 78
7.6 Load time vs. Size (no changes) . . . 78
7.7 Save time vs. Screen count (1 change in 1 Screen) . . . 80

7.8 Load time vs. Screen count (1 change in 1 Screen) 80
7.9 Save time vs. Screen count (Many changes in 1 Screen) 82
7.10 Load time vs. Screen count (Many changes in 1 Screen) 82
7.11 Save time vs. Screen count (1 change in all Screens) 84
7.12 Load time vs. Screen count (1 change in all Screens) 84
7.13 Save time vs. Screen count (Many changes in all Screens) 86
7.14 Load time vs. Screen count (Many changes in all Screens) 86
7.15 Publication steps with the second solution 88

List of Tables

3.1 EnC supported code changes 16
3.2 EnC unsupported code changes 17

7.1 Small Model | No Changes 88
7.2 Big Model | No Changes 88
7.3 Saving | Small Model | 1 Change in 1 Screen 90
7.4 Loading | Small Model | 1 Change in 1 Screen 90
7.5 Saving | Big Model | 1 Change in 1 Screen 90
7.6 Loading | Big Model | 1 Change in 1 Screen 90
7.7 Saving | Small Model | Many Changes in 1 Screen 92
7.8 Loading | Small Model | Many Changes in 1 Screen 92
7.9 Saving | Big Model | Many Changes in 1 Screen 92
7.10 Loading | Big Model | Many Changes in 1 Screen 92
7.11 Saving | Big Model | 1 Change in all Screens 94
7.12 Loading | Big Model | 1 Change in all Screens 94
7.13 Saving | Big Model | Many Changes in all Screens 95
7.14 Loading | Big Model | Many Changes in all Screens 95

B.1 No changes 113
B.2 One change in one Screen 115
B.3 Many changes in one Screen 117
B.4 One change in all Screens 119
B.5 Many changes in all Screens 121

Listings

4.1 Example application model . . . 22
4.2 Compilation Summary does not exist . . . 24
4.3 Compiler version changed . . . 24
4.4 Compilation Summary exists and Compiler version is the same . . . 25
4.5 Cleanup . . . 26
4.6 Example application model with modifications . . . 27
4.7 espace.xml . . . 29
4.8 uiFlows.xml . . . 29
4.9 clientVariables.xml . . . 30
4.10 compilationUnits.xml . . . 30
4.11 nodes.251ee3e2-6626-4ed7-a115-ee17cc11d5e2.xml . . . 30
4.12 nodes.ab3367b1-8a87-47f6-80a3-8c36f6172d4e.xml . . . 31
4.13 widgets.9a640ed5-cba0-4218-b6ee-f4ff0e55c05f.xml . . . 31
4.14 widgets.54d37874-1957-422f-a1f3-a6fdc368d787.xml . . . 31
6.1 example-app.json . . . 39
6.2 Vite's create React app command . . . 44
6.3 Vite's run app command . . . 44
6.4 "Compile" extension method . . . 44
6.5 App.jsx . . . 45
6.6 b865ee99-001b-2814-a312-16c532721767.jsx . . . 46
6.7 c02a48a6-af38-49bf-1439-5a57dcc7f33c.jsx . . . 47
6.8 TextWidget.jsx . . . 47
6.9 ImageWidget.jsx . . . 48
6.10 LinkWidget.jsx . . . 48
6.11 Original Title property implementation . . . 50
6.12 New Title property implementation . . . 50
6.13 SubscribeToTransactionStepEvent method (Compiler class) . . . 51
6.14 Subscribing the Compiler to an application's events . . . 52
6.15 TransactionStepExecuted method (Compiler class) . . . 52

6.16 CompilationQueue 53
6.17 TransactionStepExecuted method continuation 54
6.18 Managing the Compilation Queue and launching compilation threads . 55
6.19 Serialized version 1 60
6.20 Serialized version 2 61
6.21 NewObjects collection serialization 64
6.22 WrittenValues collection serialization 66
6.23 Version deserialization 68
6.24 GetOrCreate method 69
6.25 VersionedObjectFactory class 69
6.26 Version deserialization (continuation) 71
6.27 AddCommittedVersions method (TransactionsManager class) 72
A.1 OutSystems Platform Compiler 103

Acronyms

1CP 1-Click Publish *(pp. 21, 74, 75, 87)*

AI Artificial Intelligence *(p. 33)*
API Application Programming Interface *(pp. 14, 17, 33, 38)*

CPU Central Processing Unit *(p. 75)*

DBMS Database Management System *(p. 17)*
DLL Dynamic-link library *(pp. 24, 28)*
DSL Domain-Specific Language *(p. 7)*

EnC Edit and Continue *(p. 16)*

GUID Global Unique Identifier *(pp. 21, 38)*

HMR Hot Module Replacement *(pp. 12, 13, 43, 44)*

I/O Input/Output *(p. 83)*
IDE Integrated Development Environment *(pp. 1, 2, 5–7, 19–21, 33, 35, 49–52, 72, 75, 87, 96)*

JVM Java Virtual Machine *(p. 6)*

ODC OutSystems Developer Cloud *(p. 21)*
OS Operating System *(p. 75)*

R&D Research and Development *(p. 1)*
RAM Random-access memory *(p. 75)*
RHLR React Hot Loader *(pp. 12, 13)*

SaaS Software as a Service *(p. 19)*

UI User Interface *(pp. 4, 5, 8, 10, 14, 16)*

URL Uniform Resource Locator *(pp. 14, 42, 46, 48, 49)*

1 Introduction

This work was done in the context of a research partnership between the Department of Computer Science of NOVA School of Science and Technology and the Research and Development (R&D) department at OutSystems.

The OutSystems Platform [6] is a visual model-driven development and delivery platform that allows developers to create enterprise-grade web and mobile applications. The platform's Integrated Development Environment (IDE) is called Service Studio, and it enables developers to design in a single place all the aspects of an application. This includes the user interfaces, business logic, database models, and integration with external systems (e.g., via REST services), among others.

1.1 Context

Customers now expect applications to be delivered in very short time frames. It is no longer acceptable for a project to take two years to be completed, which is common when building applications with traditional technologies like Java or .NET. Teams want new applications done in less than 20 weeks and want changes to be made almost instantly [43]. With this in mind, we can make the argument that the ability to change an application's code and quickly test it is of great importance. It has the potential to significantly reduce development times, creating a better experience for the developers and helping satisfy consumer needs.

1.2 Motivation

OutSystems visual development approach, known as low-code, aims to enable faster development cycles, allowing developers to prototype, test and deploy applications quickly. However, even in low-code platforms, the develop-compile-test pipeline is explicit. OutSystems developers currently follow the subsequent process when creating or making modifications to apps:
Opening the app in the IDE from a file or directly from a running server; making changes;

when finished, publishing the application, which corresponds to the transportation, compilation and deployment phases. Publishing an app is only allowed if the app is in a valid state (well typed). The publish process takes some time, after which the app can be tested.

This experience can be slow and cumbersome. It breaks the workflow fluidity since, after making the desired changes, the developer is forced to stop their work and wait, before being able to see and test the results. This is especially noticeable in large apps, where saving the application model and uploading it to the OutSystems Platform Server can be a costly operation.

1.3 Objectives

In this work, our goal is to shorten the time between making changes to an application in OutSystems and being able to see the results, making the development experience more fluid and interactive, and by using live programming techniques. We explored an approach different from the one presented previously, in which the developer does not have to explicitly initiate the publishing process. Instead, the OutSystems Platform will proactively publish the app whenever it is in a valid state. This requires us to focus on the saving, transportation, and loading of the application model. We aimed to optimize these processes or remove some of them from the Publication process.

1.4 Contributions

The prototype we developed consists of a simplified version of the OutSystems metamodel, as well as an abstraction of the OutSystems Compiler. In this prototype we implemented two solutions. In the first solution, saving, transportation and loading of the model are eliminated, by having the IDE and the Compiler running in the same process and sharing the application model. The second solution optimizes these two tasks instead, by only transporting and loading the parts of the model that were changed. Our prototype demonstrated that our proposed approaches are viable and have the potential to provide improvements. Our implementation process also provided insights into these solutions' challenges and how to overcome them. Finally, this work contributed to an ongoing research effort.

1.5 Document Structure

The remainder of the document is organized in the following chapters:

1. State of the Art presents the current state of the art.
2. Related Work presents other tools that try to solve the same problem.
3. OutSystems Platform presents the platform and details the Publication process.

4. Proposed Solutions presents our proposed solutions to the problem.
5. Prototype Implementation presents the relevant details of the implementation of our prototype and our solutions.
6. Results shows and discusses the results obtained from benchmarks, which are used to verify whether our solutions provide improvements and quantify them.
7. Conclusions includes the concluding remarks as well as relevant future work.

2

State of the Art

In this chapter we present the current state of the art. Following this, we describe the current experience of developing an application in OutSystems.

2.1 Live Programming

In Burckhardt et al. paper [2], the authors define Live Programming as the ability to edit the code of a running program and immediately seeing the effect of the code changes. They argue that, at a minimum, Live Programming requires creating a snapshot of the state of the program, re-executing the changed code and re-displaying the results. In our work, however, we will not focus on the preservation of state, which could be a challenge for future work. The authors also argue that one major difficulty that developers face is that they have to simulate parts of the execution of the program in their minds during development, creating a gap between program code and execution. This gap takes two forms. The first one is the time consumed by rebuilding the program, executing it and guiding it (possibly with manual input) to the place where changes can be observed. The second form is the cognitive distance between looking at the code and understanding what it will do when run. Shortening this perception gap requires an environment where code can be edited in a continuous manner with uninterrupted live feedback. The authors claim that this has the potential to improve the learning experience of novice programmers while boosting the productivity of experienced ones. They propose a formal model, where a program consists of both code and persistent data. A code change is simply one possible transition of the program in this model. They separate the User Interface (UI) state from ordinary state, and the render code that builds the UI from ordinary code. This way, upon code changes, the UI state is discarded and is rebuilt, for the currently displayed page, using the separated render code for that page. This separates the rendering and non-rendering aspects of a UI program, allowing the display to be refreshed on code changes without restarting the program. The authors modified the TouchDevelop language, implementing the model they proposed. This goes beyond the scope of our work. Our focus will be the rebuilding process of OutSystems applications, and our goal is to make it faster.

In this paper, the authors state that languages based on mostly declarative programming models, including many visual languages, already provide a live programming experience. However, these languages are not expressive enough for more complex general purpose programs. They also state that many IDEs for languages such as LISP, Smalltalk, Java and C# support a "fix-and-continue" feature, where the programmer can modify code without restarting the debugging process. Nonetheless, this often results in non-responsive feedback, since usually the program initially builds a tree of widget objects to be rendered for the UI, but changing the code that does this is meaningless as it will not execute again. Furthermore, many programming environments already offer a limited live experience by separating declarative style declarations from the imperative code that generates content, such as HTML and CSS. However, this live experience only handles aesthetic tweaks.

Steven L. Tanimoto [13] presents four levels of program liveness that reflect the degree of feedback a program provides to the user. In a 2013 article [11], they added two more degrees of intensity, to account for possibilities that the author considered to be on the horizon for programming environments.

1. Informative - a representation of a program on paper such as a program flowchart.

2. Informative and significant - simple program execution.

3. Informative, significant and responsive - the system provides feedback in response to user-initiated edits. Includes event-driven response.

4. Informative, significant, responsive and live - the program is designed with the possibility to be continually active. It updates the display continuously to show the results of processing a stream of data, that vary with time.

5. Tactically predictive - the computer not only runs the program and responds, but also predicts the next programmer action (with possibly multiple alternatives), and runs one or more of the predicted versions of the program. The environment does not lag behind, nor just keeps up with the programmer, instead it stays a step ahead. This is feasible through the use of machine learning technology.

6. Strategically predictive - rather than simply making tactical predictions, a system might be able to make strategic predictions. These predictions would cover the desired behavior of a larger unit of software. The system would synthesize a program with that behavior from a combination of the current program and a large knowledge base.

In our work, the aim is to make improvements in the OutSystems Platform liveness, operating at the third level of liveness. It could be the purpose of future work to bring the system closer to liveness level 4.

In this article, the author also provides a perspective into the history of Live Programming. They state that liveness as an attribute of programming environments was first studied in the context of visual languages. These languages attempt to solve a similar problem to the one that liveness addresses, which is making it easier to understand quickly what a program is doing or supposed to do, thus making programming easier. The author gives a few examples of early systems that had live qualities. The first one is Sutherland's Sketchpad [10], which allowed users to specify graphical objects interactively, and their appearances and properties were computed and displayed in real time. Although it is not considered to be a programming system, it was not a big leap to tools that supported the drawing and running of "executable flowcharts", which exemplify level 2 liveness. The author then mentions a visual programming environment that they proposed [13], called VIVA. It specified a system for fully live executable dataflow diagrams. An implementation of it [1] achieved level 3 liveness. They continue, referring to a detailed study of the implementation requirements for achieving level 4 liveness [3] in declarative visual languages. Then, the author states that the release of the computer game "Widget Workshop" marked an important point in the availability of live programming-like systems to the public. They also mention Gamut [8], a programming-by-demonstration system that did not have a run/build distinction, making it fully live. Finally, the author refers to "Data Factory" [12], a fully Live Programming system they developed, for experimental use in education.

According to the author, the Java Virtual Machine (JVM) includes a "hot-swap" feature, since version 1.4, that allows for the replacement of a class file while the overall program is running. This enables IDEs to quickly compile a new version of a class or object and insert it into the running JVM.

Furthermore, the author emphasizes the importance of live coding in performance arts. They then state that Live Programming is appearing in more and more contexts, such as web-server scripting, learning environments and professional tools. They predict that this trend is likely to continue.

Miguel Domingues and João Costa Seco [4] introduce a programming model for safe and incremental construction of live data-centric applications. Their approach allows for both code and data updates to be safely applied during execution and provides immediate feedback. The static verification of each programming step is combined with a scheduling discipline that ensures the absence of runtime errors and interferences between execution and development. Their model is instantiated in an imperative and reactive programming language, where typing ensures that the system is always sound. To achieve this, the authors introduce three main ingredients: State variables that model the persistent data layer; Data transformation expressions, that represent either a query over persistent data, or code that processes results to serve a view of the system and that model the application logic layer; And actions, which are delayed computations modeling event handlers that enclose queries to the data layer. The system has a remote interface to manage the application code. Construction operations are interpreted at the systems' interface level to

create or change parts of an application and trigger data updates. Submitted operations are statically checked and are rejected if they are ill-typed. They follow data-driven operational semantics to ensure changes in data are proactively pushed to interfaces. Once more, the focus of our work is different, since our goal is to speed up the compilation process of OutSystems applications.

2.2 OutSystems Language Elements

In OutSystems, development is made in Service Studio, the platform's IDE. In Service Studio, developers work with a proprietary Visual Domain-Specific Language (DSL) that provides graphical metaphors with which they can define the data model, compose user interfaces and define business logic. These metaphors abstract the development of an application from the implementation details. They are the OutSystems language elements [9]. We will introduce a subset of these elements, focusing on the ones that are relevant for our explanations and examples.

2.2.1 eSpace

An eSpace can be both a running deployable application and a module [9]. All the elements that we describe next are contained in it. Modules are used to structure applications and aggregate related functionality, implementing a specific purpose. They are wrapped in a pluggable interface, so they can be used by other systems.

2.2.2 Entity

Entities are elements that allow to persist information in the database and to implement the database model. They represent a table in the database [37, 38]. An Entity is defined through Entity Attributes that store the information related to it.

2.2.3 Action

Actions are used to create the business logic of the application. As the name implies, Server Actions define the logic that runs on the server and Client Actions define the logic that runs on the client device (i.e. web browser or mobile device). Logic is encoded through the composition of visual elements [9]. An Action resembles a graph, where the nodes are the action elements and the edges define the control flow. An Action element can be a control structure, such as an if or foreach, calls to other actions, queries to the database, among others. Actions can have input parameters and, in the case of Server Actions, they can produce output values.

2.2.4 UI Flow

UI Flows are elements that group Screens into logical units with common settings [44].

2.2.5 Screen

A Screen is a UI element that contains other UI elements for users to interact [42].

2.2.6 Aggregate

Aggregates allow the developers to fetch data using an optimized query, tailored to the usage. They automatically absorb changes in the data model and can load the local database's data from the server. They support combining several Entities and advanced filtering, and bring only the attributes that are used. Aggregates can be part of Screens or Actions [33].

2.2.7 Widget

Widgets are contained within Screens. They are visual elements used to design and organize the app's UI. They serve as building blocks to accelerate UI development [36]. There are many types of widgets, including Text Widgets, whose purpose is to display text; Image Widgets, which display an image; and Link Widgets, which can contain other widgets within themselves, like text and image widgets, and, when clicked, redirect the user to other resources, typically a different screen.

2.2.8 Client Variable

Client Variables are used to store data in the client-side, in a key-value format. They can store, for example, configurations and app context data. They can only store basic data types (except Binary data, which cannot be stored) [34].

2.3 Developing a simple Task List application in OutSystems

To be able to understand and describe the experience of developing an application using the current version of OutSystems, we created a simple Task List application. We wanted the Task List application to have two screens. The main screen should display a list of all tasks. Each task should have a description, a due date and an indication of whether it is still active or not. The second screen should allow for the creation of new tasks or modification of existing ones. It should display a form where the user can insert the necessary input. To navigate between screens, the main screen should have a button to create a new task, that redirects the user to the second screen. The user must also be able to click a task and be redirected to the second screen to modify that task. The second screen should have a save and a cancel buttons, that redirect the user to the main screen and either save or discard the changes, respectively.

To create the Task List application in OutSystems, we started by creating a Task Entity, in the Data tab. Then, we added the Entity Attributes Description, DueDate and IsActive (see figure 2.1).

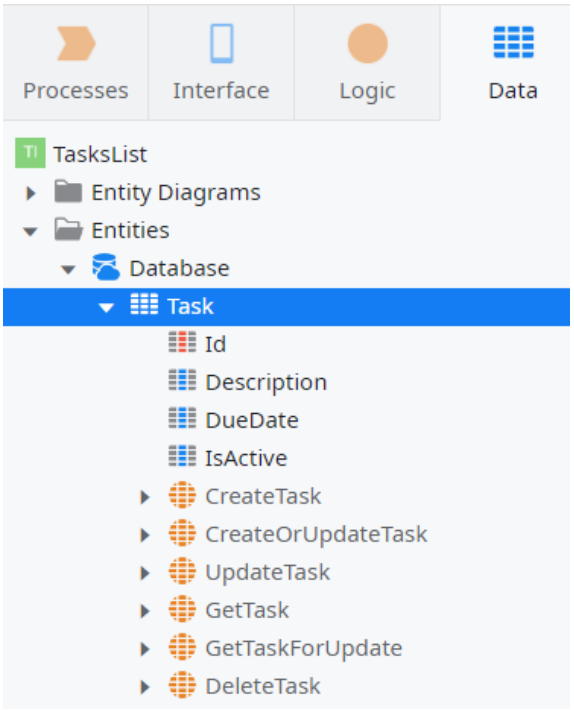

Figure 2.1: Task Entity creation

After that, we created the Tasks Screen that lists all the tasks. All we had to do was to use the Scaffolding feature of Service Studio by simply dragging and dropping the Entity into the Screen. This automatically creates the Aggregate that fetches the data from the database, the list Widget that displays that data, and all the necessary logic that allows for more data to be loaded when scrolling. Service Studio shows a preview of the created Screen and, if the developer is happy with the results, they can go ahead and Publish the application (see figures 2.2 and 2.3).

Finally, we created the task creation/editing Screen. We used the available Widgets by dragging and dropping them into the Screen. A Form Widget was used, along with Input Widgets for the description and due date, a Checkbox Widget for the IsActive attribute and save and cancel Buttons (see figure 2.4).

This Screen needs to have an input parameter, that stores the ID of the task being edited (its value is null if a new task is being created). We created an Aggregate that fetches the task from the database with the given ID. Some logic was implemented in a Client Action to save the new data in the database (see figure 2.5). A few more steps were also made to ensure that the Widgets are linked to the correct attributes of the Task Entity and that the Buttons trigger the necessary Client Action and redirect the user to the main Screen.

All of this can be done in a matter of minutes. The developer does not need to write a single line of code and the learning curve is small, so development will be fast even if they are inexperienced.

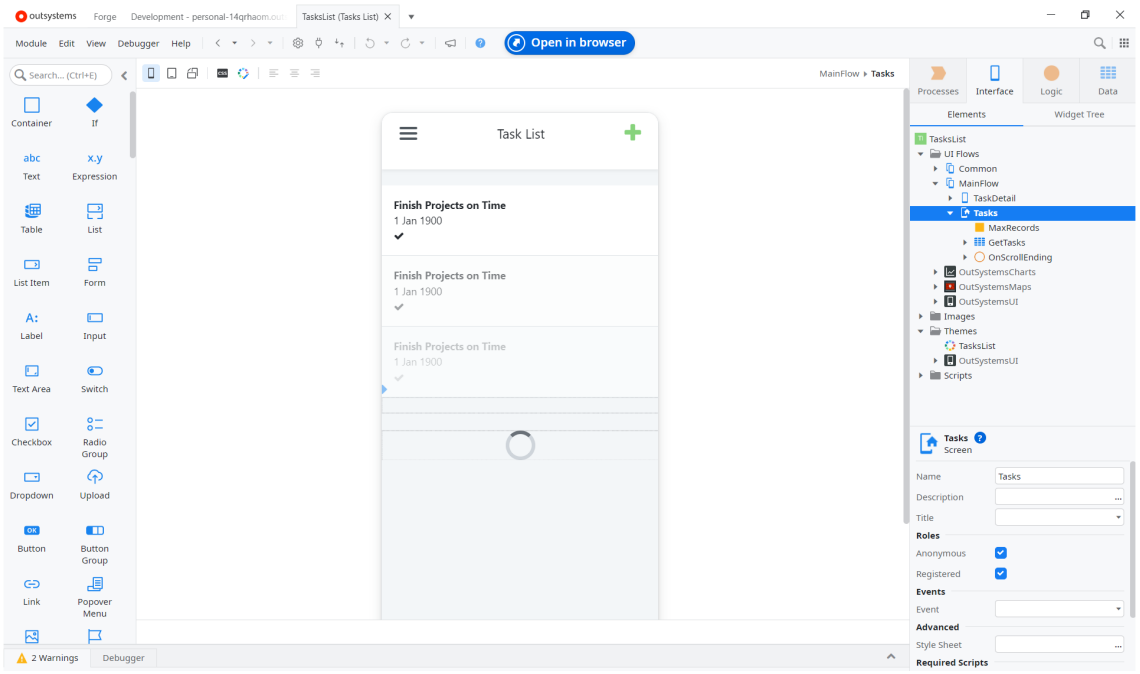

Figure 2.2: Task list screen preview

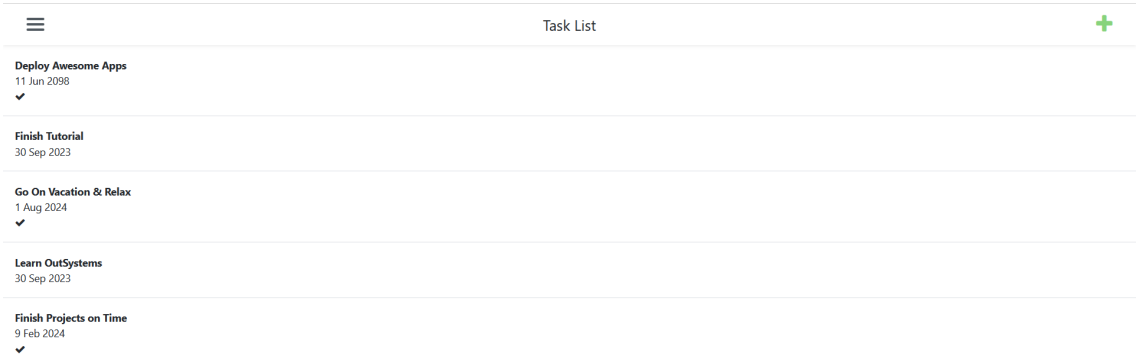

Figure 2.3: Task list screen in the browser

However, when we tried to make changes to the style of the UI, or make changes to the business logic, the experience was not as fast. We changed the color and font of some text and, between clicking the Publish button and seeing the results in the browser, around 18 seconds passed. Then we changed the message that is shown to the user when there are errors in the input of the form to create/edit a task. This was done by changing the feedback node in the Client Action in 2.5, that is in the false branch of the condition that checks the form's validity. The time interval to be able to test those changes was similar. This is just an example and there are a lot of factors that contribute to this time interval, so it can vary greatly. Nonetheless, we can make the point that the experience is not fluid, even in a very simple application like this one.

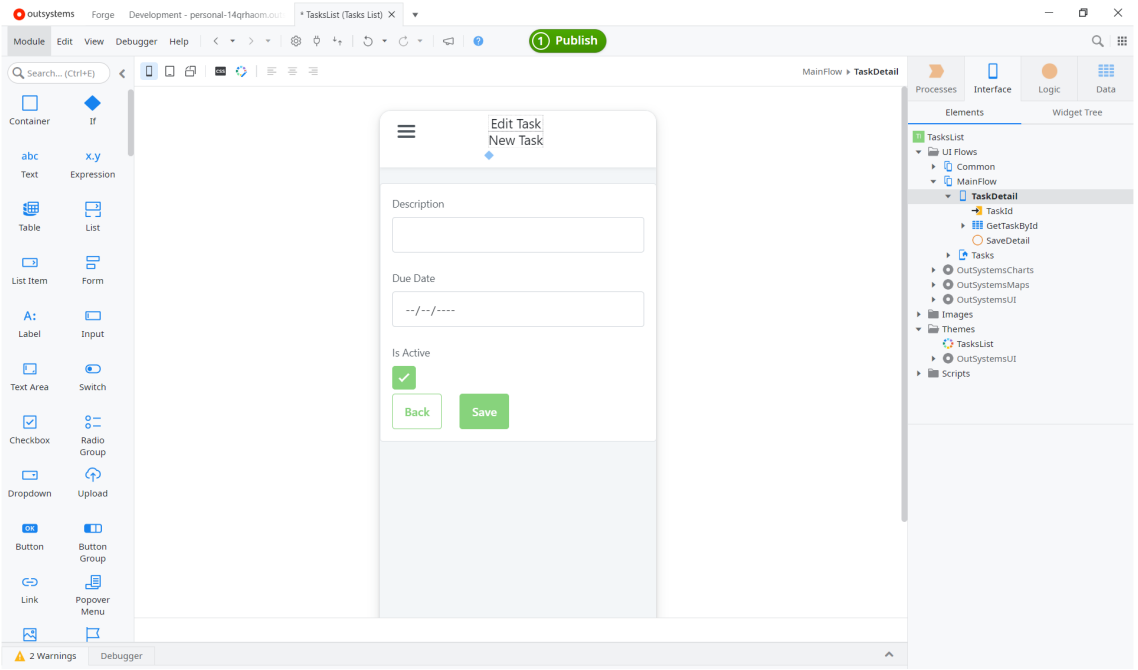

Figure 2.4: Task creation/editing screen preview

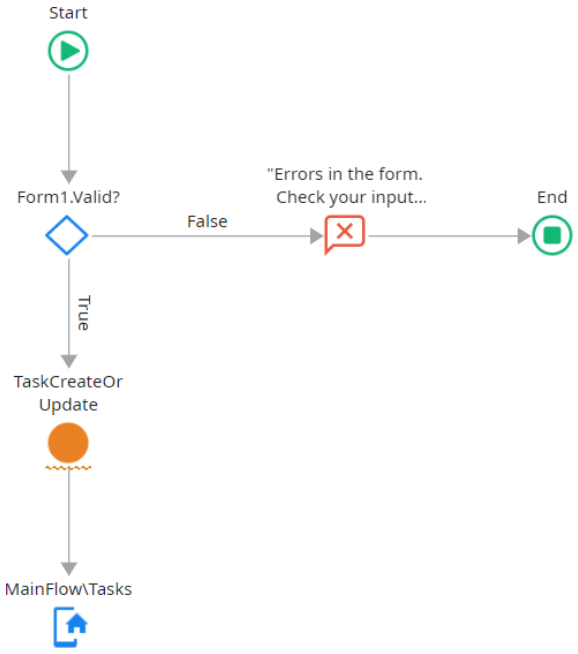

Figure 2.5: Client Action

3

Related Work

In this chapter we present other tools that have mechanisms to shorten the times between making changes to an application's code and being able to test the results. We also explore other work that is relevant for our solutions.

3.1 Other tools

In the preparation work, we explored different tools that aim to shorten the times between making changes to the code and being able to see the results. The goal was to make an assessment of the development experience provided by these tools and compare them with the experience provided by OutSystems. To achieve this, we developed the same Task List application, previously developed in OutSystems, using React for the frontend, ASP.NET for the backend. We will introduce both technologies and the mechanisms that they use, describe the experience of developing the task list app with them and compare it with OutSystems.

3.1.1 React

React is a free and open-source front-end JavaScript library for building web and native user interfaces composed of individual pieces called components. It is maintained by Meta and a community of individual developers and companies. [29, 46]

In React, it is possible for developers to make changes to their code and immediately see the results in their running applications, upon saving the file. There are many ways to achieve this, but one of the most popular options was to use React Hot Loader (RHLR), which is a plugin that allows React components to be live reloaded without the loss of state. It works with Webpack and other bundlers that support Hot Module Replacement (HMR) and Babel plugins [15]. Webpack is a free and open-source module bundler for JavaScript. It is designed mainly for JavaScript, but it also works with front-end assets such as HTML, CSS, and images if the corresponding loaders are included. It takes modules with dependencies and generates the corresponding static assets [27]. Babel is a JavaScript compiler [17]. HMR is a feature offered by Webpack. It allows modules to be updated,

added or removed at runtime without the need for a full reload [25]. Here is a summary of how HMR works:

- Some steps are necessary to allow modules to be swapped in and out of an application. Firstly, the application asks the HMR runtime to check for updates. Then, the runtime asynchronously downloads the updates and notifies the application. After that, the application asks the runtime to apply the updates. Finally, the runtime synchronously applies the updates.

- To allow for HMR, besides producing the normal assets, the compiler need to emit an update. This update consists of two parts. The first part is a JSON file that contains a list of all the updated chunks, as well as the new compilation hash. The second part contains the new JavaScript code for all updated modules, or a flag indicating that a module was removed.

- HMR only affects modules that contain HMR code. If a module does not have code to handle HMR, the update bubbles up. Thus, a single handler can update a complete module tree. If a single module from a tree is update, the entire set of dependencies is reloaded.

- The module system runtime has to emit additional code to track modules' parents and children. When checking for updates, the runtime makes an HTTP request to get the previously mentioned JSON file. If this request fails, there are no updates available. Otherwise, the list of update chunks is compared to the list of currently loaded chunks. For each loaded chunk, the corresponding updated chunk is downloaded. All module updates are stored in the runtime. Then, the runtime switches to the ready state. To apply the updates, the runtime flags all updated modules as invalid. For each invalid module, there needs to be an update handler in the module, or the invalid flag bubbles up, invalidating the parent, until it reaches an update handler or the app's entry point. If it bubbles up from an entry point, the process fails. Afterwards, all invalid modules are disposed and unloaded. The current hash is updated, the runtime switches back to the idle state and everything continues as normal.

RHLR has been deprecated and is, at the time of writing, being replaced by React Fast Refresh [14]. React Fast Refresh is considered to be more reliable and is officially supported by React [18]. Nonetheless, the underlying mechanisms are the same.

When Creating the frontend for the Task List application in React, we started by creating the main application's component, which uses React's Router library to allow for navigation between screens. The entire component is wrapped with a Router component that sets up the application's navigation. The main component also defines the structure of the application and defines navigation links, using Link components. We created a title ("Task List") that points to the root ("/") of the application and a button with text "Create

Task" that points to "/create". Then we configured the routes using the Routes and Route components. When the path is "/", it renders the TaskList component. If it is "/create", it renders the TaskCreate component. And when it is "/edit/:id", it renders the TaskEdit component (with a parameter id). We proceeded to create each of these components.

The TaskList component fetches a list of tasks from a specified Application Programming Interface (API) endpoint and renders them in a list format. For each task, its description, due date and active status are displayed. The description is linked to the edit page based on the task's ID.

The CreateTask component is responsible for rendering a form to create a new task. It uses state variables to store the task's description, due date and active status. We defined event handling functions that update these variables when the user interacts with the form inputs. An event handler for the form submissions was also created. It is responsible for creating a new task object from the state variables and sending a POST request to the specified API endpoint. Finally, we created the form with input fields for the task description, due date, and an active checkbox. It also contains a button with text "Create" that submits the form.

The EditTask component is responsible for editing and updating an existing task. This component receives an 'id' parameter, extracted from the Uniform Resource Locator (URL) and starts by fetching the task with that ID from the API endpoint. The state variables are then initialized with the correct values from the fetched task. The rest of the component works similarly to the previous one, but instead of sending a POST request it sends a PUT request when the form is submitted. We also added an event handler for the deletion of a task. It sends a DELETE request to the API to delete the current task and is triggered by a "Delete" button in the form.

Although not mentioned previously, we also had to write code to handle errors in all of these components. The task list screen can be seen in figure 3.1 and the task editing screen can be seen in figure 3.2.

Using React requires the developer to be familiar with it and its underlying technologies (JavaScript, HTML, CSS). If that is not the case, the learning curve is significantly steeper than when using OutSystems. This can greatly affect the time required to write the code for an application. Whereas with OutSystems it is much easier for any developer, experienced or inexperienced, to start using the technology and quickly creating an application. Also, OutSystems provides UI elements that allow developers to easily create an application with a modern design. React does not provide this experience out-of-the-box. There are libraries that provide UI elements for developers to use, but the process is not as straightforward.

It is also important to emphasize that all the work detailed here was only to create the application's frontend. More work needs to be done for the backend, using ASP.NET, as we will see next. In OutSystems, all aspects of the application are created faster and more easily, in the same environment.

However, React allows developers to modify specific modules in a running application

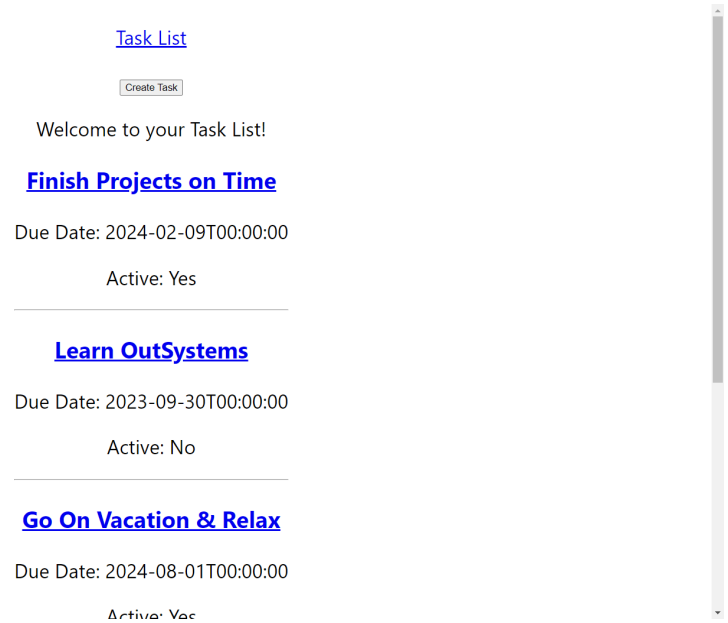

Figure 3.1: Task list screen created with React

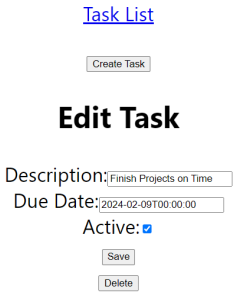

Figure 3.2: Task editing screen created with React

without requiring a full page reload, with the mechanisms detailed previously. This is significantly different from the experience when using OutSystems, that we explained in chapter 2. In OutSystems, after publishing the application, it has to be reloaded in the web browser or mobile device as well, in order for the changes to be seen. Besides taking a significant amount of time and breaking workflow fluidity, state is not preserved.

A common scenario where this matters is to work on a feature that is multiple screens away from the launch screen. Every time the developer reloads, they have to click on the same path again to get back to the feature. These aspects are especially problematic when tweaking the UI, since the task is usually an iterative process that requires many small changes and testing. To change the color and font of some text in React, we just had to make changes to the stylesheets file and, upon saving it, only the affected elements were reloaded and the changes could immediately be seen in the browser.

3.1.2 ASP.NET

In OutSystems, the backend of applications is implemented using the ASP.NET framework. ASP.NET is a server-side web-application framework designed for web development to produce dynamic webpages [28] and is part of Microsoft's .NET framework.

It has a feature called .NET Hot Reload that enables developers to modify their apps source code, including changes to stylesheets, while the application is running, without needing to manually pause or hit a breakpoint and without losing app state [32, 23]. .NET Hot Reload is powered by the Edit and Continue (EnC) mechanism, which has some limitations and does not support all types of code changes. It can handle most types of code changes within method bodies. However, most changes outside method bodies and a few changes within method bodies require the app to be restarted [24]. In table 3.1 it is possible to see some of the code changes that are supported by EnC, and in table 3.2 some unsupported ones. The complete list is available in [21]. .NET Hot Reload is not open source and more information about its implementation is not publicly available.

Table 3.1: EnC supported code changes

Supported Code Changes
Add methods, fields, constructors, properties, events, indexers, field and property initializers, nested types and top-level types (including delegates, enums, interfaces, abstract and generic types, and anonymous types) to an existing type
Add and modify iterators, yield statements
Add and modify async methods, await expressions
Add and modify operations with dynamic objects
Add lambda expressions
Modify generic code
Modify lambda expressions
Edit partial class
Deleting members other than fields
Renaming members other than fields
Rename method parameters
Modify method parameter types
Change return type of a method/event/property/operator/indexer

Add and modify custom attributes
Adding and modifying namespace declarations

Table 3.2: EnC unsupported code changes

Unsupported Code Changes
Modify interfaces
Add a method body such that an abstract method becomes non-abstract
Add new abstract, virtual, or override member to a type
Modify a type parameter, base type, delegate type
Modify a catch-block if it contains an active statement
Modify a try-catch-finally block if the finally clause contains an active statement
Delete types

To create the application's backend we first had to decide which Database Management System (DBMS) we were going to use, to store its data. We decided to use SQLite for its simplicity and lightweight nature. Then, using the ASP.NET framework, we had to define the task entity, with all its attributes (ID, description, due date and active status), as well as connecting the backend service to the DBMS. Finally, we created each necessary endpoint that was needed for the API: an endpoint to get a list of all tasks in the database, an endpoint to get a task given its ID, an endpoint to create a new task, an endpoint to modify a task given its ID and an endpoint to delete a task given its ID. This requires the developer to be familiar with the ASP.NET and its underlying technologies, namely C#, as well as with a DBMS. Once again, the learning curve is much steeper for inexperienced developers and more work needs to be done, since a lot of the tasks explained here are abstracted to the developer in OutSystems.

However, with .NET Hot Reload enabled, we were able to change the backend code without having to relaunch the server. Once again, this contrasts with the experience in OutSystems. In OutSystems, for the changes to the application's backend code to take effect, the developer has to publish the app first. In this case, reloading the app in the web browser is not required, so the experience is not as cumbersome as in the previous example. Nonetheless, publishing the app is still time-consuming. Also, since the server is restarted, it may take a non-negligible amount of time to be available again, depending on the app's complexity.

3.2 Persistent Data Structures

The term Persistent Data Structures was introduced by Driscoll, Sarnak, Sleator and Tarjan in their 1986 article [5]. In this article, the authors characterize ordinary data structures as ephemeral, since when a change is made to a structure, the old version ceases to exist and only the new version is available for use. On the other hand, persistent structures allow

any version to be accessed at any time. They also make a distinction between partially persistent structures and fully persistent structures. In the former, all versions can be accessed but only the newest version can be modified, whereas, in the latter, all versions can be both accessed and modified.

Persistent Data Structures are relevant for our work, as we will see later. In essence, applying their principles to the OutSystems application model allows us to have, in a single instance, multiple versions of a model. We will explain in more detail how this is useful when proposing our solutions.

4

OutSystems Platform

The OutSystems Platform is a low-code software development tool that allows developers to use a model-driven, visual language with a drag-and-drop interface to build apps, along with pre-built components and templates. It enables the creation of complete applications with modern user interfaces, integrations, data and logic faster than traditional development, by delivering visual development capabilities and automation. The problems it aims to solve are the lack of agility of traditional development, the rapid increase in Software as a Service (SaaS) demand, and the difficulty in hiring developer talent [45, 35].

With OutSystems, development teams can quickly create a working prototype and, from there, gather business feedback and adjust the product accordingly. Through abstraction and automation, repetitive and tedious tasks are eliminated. Customer expectations can be met faster and employees can benefit from a boost in efficiency and motivation. Changing and maintaining applications becomes quicker and easier. The tool is accessible even to inexperienced developers, lowering the barrier to entry, and developer retention is also increased [39].

In this chapter we aim to describe the aspects of the platform that are relevant to our work.

4.1 Architecture of the OutSystems Platform

The OutSystems Platform has two main components: the Service Studio, the IDE where developers create and develop applications, and the Platform Server, where the applications are compiled and deployed. The compilation and deployment processes are aggregated in a single process called Publication. The Publication of an application is executed in the Platform Server and the process will be described in a subsequent section.

The Platform Server is composed of many components that have different roles in the Publication Process. Figure 4.1 shows these components and the relations between them. One of these components is Service Center, who acts as an interface between Service Studio and the other components of the Platform Server. In the Publication process, Service Studio communicates only with Service Center, which orchestrates most of the

process [9].

It can be observed that the Platform Server's components run in three servers: the Front End Server, the Database Server and the Deployment Controller Server.

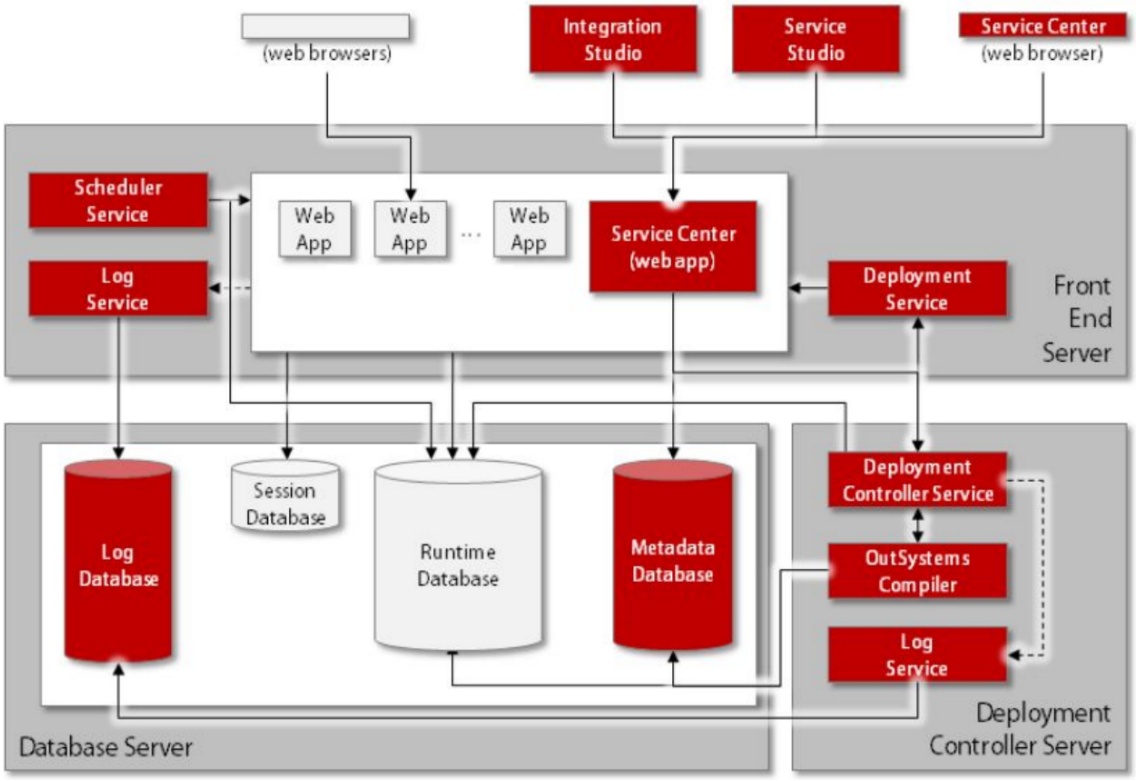

Figure 4.1: OutSystems Platform Server's Architecture

4.2 OutSystems Application

An OutSystems application can be defined in two levels: the IDE level and the runtime level:

- IDE - During development in the IDE, an application consists of a model of its elements, that is loaded in memory.
- Runtime - During runtime (i.e. when an application is published and available to the users) an application has a client/server architecture. It consists of files for the client and server sides, that are stored in the Front End Server.

4.3 OutSystems 11 Publication Process

The Publication of an application is a process that consists of transforming the application model into an ASP.NET application and deploying it to the Web Application Server [9],

which is part of the Front End Server. It is now possible to understand why the OutSystems Platform has a compiler. Apps cannot simply be interpreted, because it is necessary to produce artifacts from the model, as we will see in this chapter.

4.3.1 Compilation Units

In an OutSystems application, Compilation Units are the model elements that produce files, and they define the compilation granularity, since they serve as atomic elements in the compilation process. This means that, either the whole Compilation Unit is compiled, including all its elements, or none of them are. It is not possible to compile individual elements of a Compilation Unit, if they are not Compilation Units themselves. All elements of an application have a unique key - Global Unique Identifier (GUID) - and Compilation Units have Compilation Hashes as well. The Compilation Hash of a Compilation Unit is a hash value that is calculated based on its contents. They allow for quickly determining if a unit changed, by comparing the hashes of the two versions of that unit, instead of having to do a deep (and costly) comparison of their properties and child elements.

Elements that are Compilation Units include: eSpace, Screens, Entities, UI Flows, Server Actions and Client Actions.

4.3.2 Publication Steps

The process of publishing an OutSystems application consists of several steps (serialization, transportation, metadata extraction, compilation and deployment), which are presented subsequently. Publication is triggered when the application model is in a valid state and the developer clicks the 1-Click Publish (1CP) button, in Service Studio. This is the process used by version 11 of the OutSystems platform. The OutSystems Developer Cloud (ODC) version [40] has a different process, but the main steps are the same.

4.3.2.1 Serialization

In the first step (represented by 1 in figure 4.2), Service Studio performs a Compare and Merge operation and serializes the in-memory application model to an XML binary file.

The Compare and Merge operation first checks if the base version of the application in the IDE (before the latest changes) is the same as the latest published version. If this is true, the new version can be safely published without any additional operations. Otherwise, the new version has to be merged with the latest published version. In the best-case scenario, there are no conflicts between the two models, and the merge can be performed automatically. If conflicts exist, developer action is required to decide how to solve these conflicts.

The XML file stores a tree of Compilation Units, as presented in an example file in listing 4.1. This example application has an eSpace with one UI Flow named "Employees" and a Client Variable named "Username". The UI Flow has two Screens. The first Screen

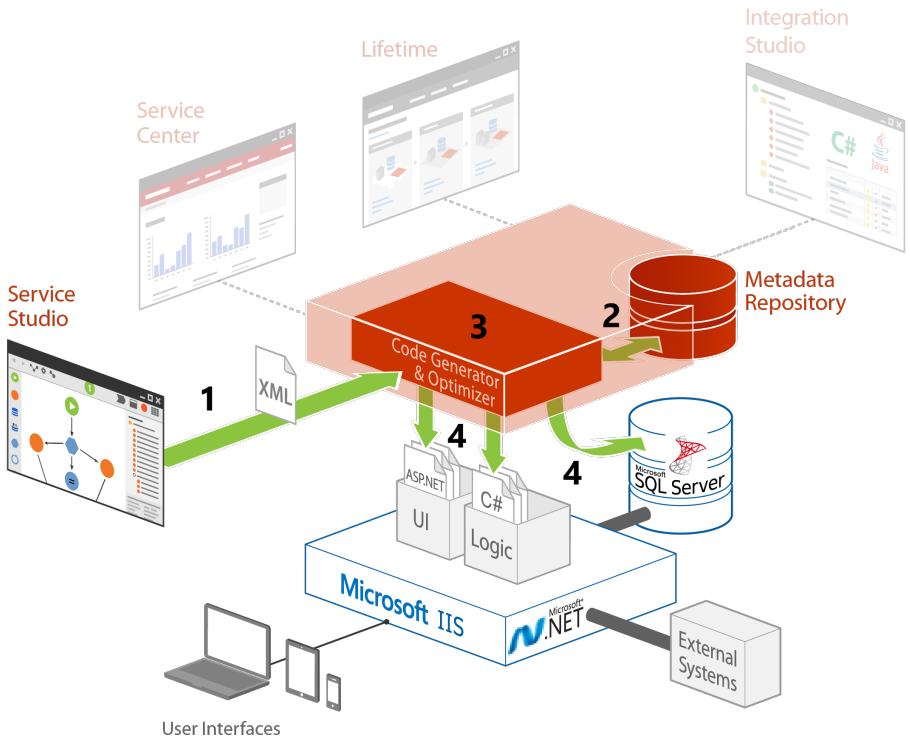

Figure 4.2: OutSystems Publication Process

is named "EmployeeList" and has an Aggregate named "EmployeeAggregate" and a Text Widget named "EmployeeListWidget". The second Screen is named "EmployeeDetails" and has a Text Widget named "EmployeeDetailsWidget". In this phase, the Compilation Hashes for each Compilation Unit are calculated as well. The eSpace contains the paths in the tree for each Compilation Unit and its corresponding Compilation Hash (starting in line 29). This information is redundant but is used to speed up the compilation process.

The binary file is then transported to Service Center. Service Center runs as any other OutSystems web application in the Front End Server, as can be seen in figure 4.1. It drives the Deployment Controller Service throughout the Publication process, dispatching the different phases as the feedback it receives from the Deployment Controller Service is positive [9].

Listing 4.1: Example application model

```
1 <oml>
2   <espace key="6e19b1f5-aede-4c7f-8b89-ee76ec49c7ba" compilationHash="b4d399cd
        ↪ -3268-4475-865c-ddae4cbfe8d6">
3     <uiFlows>
```

```
4       <uiFlow name="Employees" key="251ee3e2-6626-4ed7-a115-ee17cc11d5e2" compilationHash="
            ↪ d8f4d721-36f6-45a1-9887-7eeb795778a6">
5         <nodes>
6           <screen name="EmployeeList" key="9a640ed5-cba0-4218-b6ee-f4ff0e55c05f"
                ↪ compilationHash="b69f1aab-7a87-4d8e-b67c-645f95698a77">
7             <aggregates>
8               <aggregate name="EmployeeAggregate" key="37e0ad08-4eed-435d-965c-7295f92c5b31"
                    ↪ >
9             </aggregates>
10
11            <widgets>
12              <textWidget name="EmployeeListWidget" key="474f61e2-6094-4e38-9e1c-908
                    ↪ a26dab283" />
13            </widgets>
14          </screen>
15
16          <screen name="EmployeeDetails" key="83d3b7b4-d090-468a-bb18-4d655243071c"
                ↪ compilationHash="f3a05522-152e-44cd-a0c2-06c6cd836e4d">
17            <widgets>
18              <textWidget name="EmployeeDetailsWidget" key="f62258b6-4698-4086-a22b-4
                    ↪ e98686d4b94" />
19            </widgets>
20          </screen>
21        </nodes>
22      </uiFlow>
23    </uiFlows>
24
25    <clientVariables>
26      <clientVariable name="Username" key="e02c03ba-0b42-4158-b1b0-0573ac47cc47" />
27    </clientVariables>
28
29    <compilationUnits>
30      <compilationUnit path="/" compilationHash="b4d399cd-3268-4475-865c-ddae4cbfe8d6" />
31      <compilationUnit path="/uiFlows/251ee3e2-6626-4ed7-a115-ee17cc11d5e2" compilationHash
            ↪ ="d8f4d721-36f6-45a1-9887-7eeb795778a6" />
32      <compilationUnit path="/uiFlows/251ee3e2-6626-4ed7-a115-ee17cc11d5e2/screens/9a640ed5
            ↪ -cba0-4218-b6ee-f4ff0e55c05f" compilationHash="b69f1aab-7a87-4d8e-b67c-645
            ↪ f95698a77" />
33      <compilationUnit path="/uiFlows/251ee3e2-6626-4ed7-a115-ee17cc11d5e2/screens/83d3b7b4
            ↪ -d090-468a-bb18-4d655243071c" compilationHash="f3a05522-152e-44cd-a0c2-06
            ↪ c6cd836e4d" />
34    </compilationUnit>
35  </espace>
36 </oml>
```

4.3.2.2 Metadata Extraction

In the second step (represented by 2 in figure 4.2), Service Center is responsible for extracting some metadata from the application and sending it to the Metadata Database

where it is stored. It also sends the XML file to the OutSystems Compiler, which is running in the Deployment Controller Server, as it is possible to see in figure 4.1.

4.3.2.3 Compilation

The third step is the compilation process (represented by 3 in figure 4.2), which is responsible for producing the necessary artifacts. The Compiler will produce, for each Compilation Unit, its corresponding artifacts. These artifacts comprise code in different languages and file formats. They include: C# files to define the server logic of the application, stylesheets, JavaScript scripts and HTML files for the client, and SQL scripts to update the database schema and data. An example can be seen in the diagram of figure 4.3, which uses the example application presented in listing 4.1.

In fact, we are simplifying the process, since the compilation consists of two phases [9]. The first phase is the Code Generation phase, which produces C# source code files, among others. The second phase is where the source code files are compiled with the C# compiler into Dynamic-link library (DLL) assemblies. However, for the purpose of this work, we can ignore the second phase and assume that the compilation comprises only the Code Generation phase.

To aid in explaining the compilation process, we will present snippets of a simplified version of the Compiler. The complete code of this abstraction can be found in listing A.1 of Appendix A.

The Compiler keeps a Compilation Summary of the previous published version of the application. This summary contains the Compilation Units of the previous version, their Compilation Hashes and which artifacts were produced for each. The Compiler uses the information about the Compilation Units in the XML file and the Compilation Summary to check which actions it has to perform, as follows.

If the Compilation Summary does not exist, it means that it is the first time the application is being compiled. Thus, a new Compilation Summary is created and all Compilation Units will be compiled (listing 4.2).

Listing 4.2: Compilation Summary does not exist

```
1 if(summary == null) {
2   summary = new CompilationSummary(currentCompilerVersion);
3 }
```

If the previous version was compiled with a different version of the Compiler, the application will have to be recompiled. All the previously created artifacts are deleted, a new Compilation Summary is created and all the Compilation Units will be compiled (listing 4.3).

Listing 4.3: Compiler version changed

```
1 else if(!summary.getCompilerVersion().equals(currentCompilerVersion)) {
2   summary.deleteAllArtifacts();
```

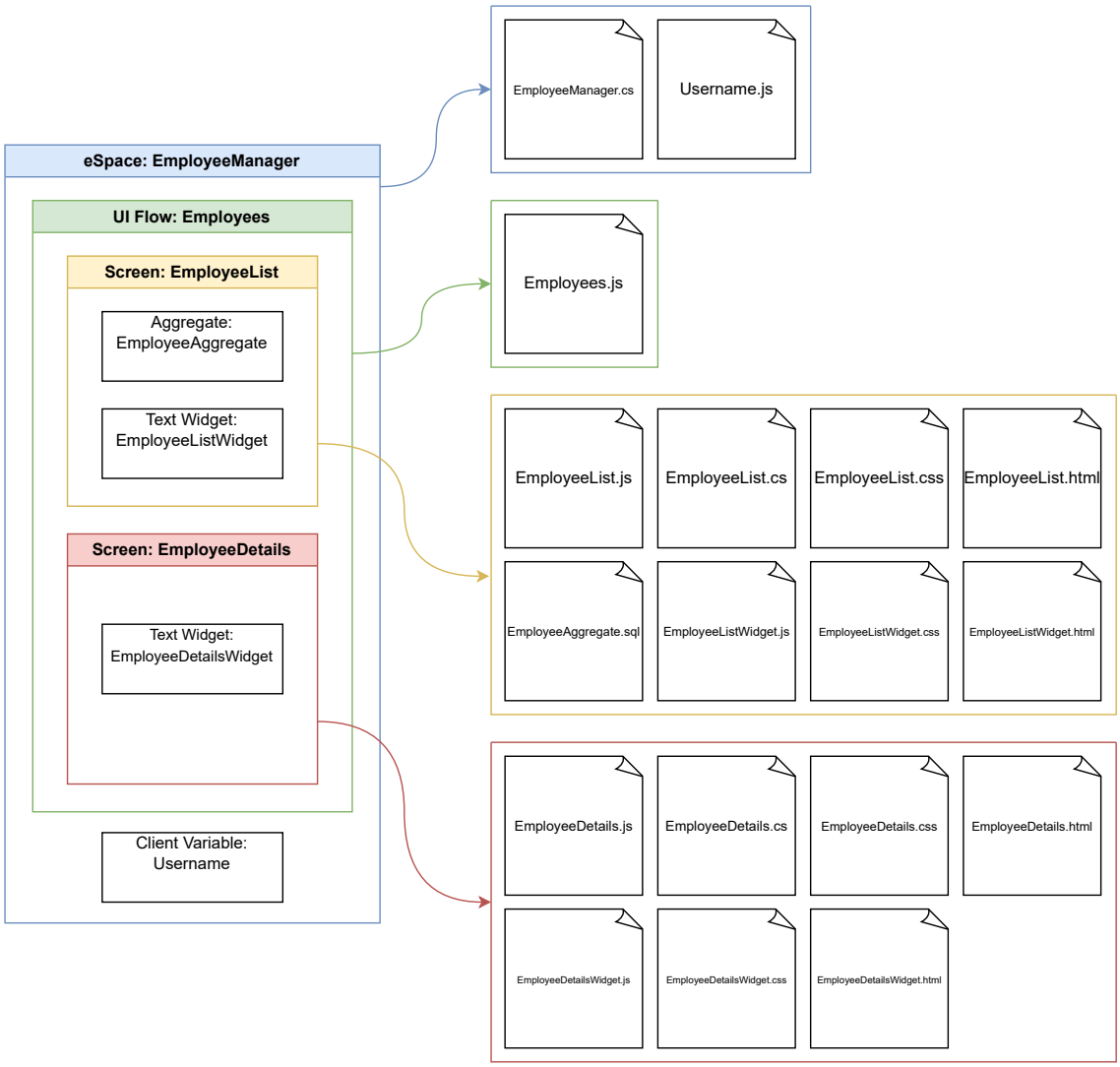

Figure 4.3: Artifacts produced by Compilation Units

```
3     summary = new CompilationSummary(currentCompilerVersion);
4 }
```

Otherwise, before proceeding with the compilation of the Compilation Units, a cleanup has to be performed. For each Compilation Unit in the Summary, the Compiler checks whether it is present in the file and, if it is, compares the previous and current hashes. If the Compilation Unit is not present in the file, it means that it has been removed. Thus, all of its previously produced artifacts are deleted. The same happens if the hashes are different. Since it is not possible to know which artifacts will remain necessary when a Compilation Unit is modified, all of its previously produced artifacts have to be deleted (listings 4.4 and 4.5).

Listing 4.4: Compilation Summary exists and Compiler version is the same

```
1 else {
2   summary.cleanUp(espace.getCompilationUnitsInfo());
3 }
```

Listing 4.5: Cleanup

```
1  void cleanUp(List<CompilationUnitInfoInModel> list) {
2    Iterator<Entry<Key, CompilationUnitInfoInSummary>> it = unitInfos.entrySet().iterator();
3    while(it.hasNext()) {
4      CompilationUnitInfoInSummary infoInSummary = it.next().getValue();
5      boolean found = false, deleteArtifacts = true;
6      for(CompilationUnitInfoInModel infoInModel : list) {
7        if(infoInSummary.getKey().equals(infoInModel.getKey())) {
8          found = true;
9          if(infoInSummary.getCompilationHash().equals(infoInModel.getCompilationHash())) {
10           deleteArtifacts = false;
11         }
12         break;
13       }
14     }
15     if(!found) {
16       it.remove();
17     }
18     if(deleteArtifacts) {
19       infoInSummary.deleteArtifacts();
20     }
21   }
22 }
```

Finally, the Compiler traverses the Compilation Unit tree in the file and, for each, it compares the previous and current hashes. If they are equal, the Compilation Unit has not been modified, so the compilation for that unit is skipped. If they are different, or if the key of the Compilation Unit is not present in the Compilation Summary, the unit is compiled. The former case indicates that the Compilation Unit has been modified while the latter indicates that it is a newly added unit.

To better illustrate this process, we can use the example application in listing 4.1 and a second version of this application, with some modifications, presented in listing 4.6. In the second version, the EmployeeDetails Screen was removed. This Screen is a child of the Employees UI Flow so, as expected, its removal caused the parent's Compilation Hash to change (see line 4, listing 4.1 and line 4, listing 4.6). Also, a new DepartmentList Screen was added within a new Departments UI Flow. With these changes, the eSpace had one of its children changed and a new one added. Once more, as expected, this caused its Compilation Hash to change (see line 2, listing 4.1 and line 2, listing 4.6). Since the EmployeeList was not changed, its Compilation Hash remained the same (see line 6, listing 4.1 and line 6, listing 4.6).

If the second version of the app is published, after the first version has been published, the Compiler will start by doing a cleanup, since a previous Compilation Summary is

present (and assuming the Compiler version has not changed). In this example, the Compiler detects that the Compilation Hash for the eSpace and for the UI Flow has changed, so it deletes all the previously created artifacts for these units. It also detects that the EmployeeDetails Screen has been removed, so it deletes all its artifacts as well. Then, the compilation proceeds. The Compiler recursively traverses the Compilation Unit tree, starting by the eSpace. Since its Compilation Hash has changed, it has to be recompiled. Its own artifacts are created, as well as the artifacts for its Username Client Variable. The Compiler continues to its children, and detects that the Employees UI Flow Compilation Hash has changed, so it is recompiled. Its own artifacts are produced, and then the Compiler proceeds to its child, the EmployeeList Screen. This time, the Compilation Hash remained unchanged, so compilation for this unit is skipped. Coming back to the eSpace's children, the Compiler detects a new Compilation Unit, the Departments UI Flow, which is compiled. Its own artifacts are created, and then the Compiler proceeds to its child, the DepartmentList Screen. It detects that this is also a new Compilation Unit and compiles it. Artifacts for this Screen are produced, as well as for its Text Widget, EmployeeDetailsWidget. After this last step, compilation is over.

Listing 4.6: Example application model with modifications

```
<oml>
  <espace key="6e19b1f5-aede-4c7f-8b89-ee76ec49c7ba" compilationHash="30a46a0c-0ff3-4f61-
      ↪ a801-7512d3ea7a4a">
    <uiFlows>
      <uiFlow name="Employees" key="251ee3e2-6626-4ed7-a115-ee17cc11d5e2" compilationHash="
          ↪ 0f125b27-2995-4f4d-b02d-4156cf37317c">
        <nodes>
          <screen name="EmployeeList" key="9a640ed5-cba0-4218-b6ee-f4ff0e55c05f"
              ↪ compilationHash="b69f1aab-7a87-4d8e-b67c-645f95698a77">
            <aggregates>
              <aggregate name="EmployeeAggregate" key="37e0ad08-4eed-435d-965c-7295f92c5b31"
                  ↪ >
            </aggregates>

            <widgets>
              <textWidget name="EmployeeListWidget" key="474f61e2-6094-4e38-9e1c-908
                  ↪ a26dab283" />
            </widgets>
          </screen>
        </nodes>
      </uiFlow>

      <uiFlow name="Departments" key="ab3367b1-8a87-47f6-80a3-8c36f6172d4e" compilationHash
          ↪ ="de89100d-f08f-475c-850b-184bc022663d">
        <nodes>
          <screen name="DepartmentList" key="54d37874-1957-422f-a1f3-a6fdc368d787"
              ↪ compilationHash="bb34e5a8-bd6c-42ef-866e-8d4c591446fd">
            <widgets>
```

```
22              <textWidget name="DepartmentListWidget" key="1584a57c-a397-4853-bf81-49490
                    ↪ a1574d0" />
23            </widgets>
24          </screen>
25        </nodes>
26      </uiFlow>
27    </uiFlows>
28
29    <clientVariables>
30      <clientVariable name="Username" key="e02c03ba-0b42-4158-b1b0-0573ac47cc47" />
31    </clientVariables>
32
33    <compilationUnits>
34      <compilationUnit path="/" compilationHash="30a46a0c-0ff3-4f61-a801-7512d3ea7a4a" />
35      <compilationUnit path="/uiFlows/251ee3e2-6626-4ed7-a115-ee17cc11d5e2" compilationHash
            ↪ ="0f125b27-2995-4f4d-b02d-4156cf37317c" />
36      <compilationUnit path="/uiFlows/ab3367b1-8a87-47f6-80a3-8c36f6172d4e" compilationHash
            ↪ ="de89100d-f08f-475c-850b-184bc022663d" />
37      <compilationUnit path="/uiFlows/251ee3e2-6626-4ed7-a115-ee17cc11d5e2/screens/9a640ed5
            ↪ -cba0-4218-b6ee-f4ff0e55c05f" compilationHash="b69f1aab-7a87-4d8e-b67c-645
            ↪ f95698a77" />
38      <compilationUnit path="/uiFlows/ab3367b1-8a87-47f6-80a3-8c36f6172d4e/screens/54d37874
            ↪ -1957-422f-a1f3-a6fdc368d787" compilationHash="bb34e5a8-bd6c-42ef-866e-8
            ↪ d4c591446fd" />
39    </compilationUnit>
40  </espace>
41 </oml>
```

4.3.2.4 Deployment

The fourth step (represented by 4 in figure 4.2) is the Deployment. The Deployment Controller Service produces an archive containing all the deployable files and sends it to the Deployment Service. The Deployment Service is then responsible for transporting the compiled application to the Web Application Server. The compiled application consists of the files produced by the compiler. In the case of DLLs, this process is optimized so that copies of DLLs are only sent if they do not already exist in the destination. It is important to note that applications may consume other applications, which means that some applications might need to receive copies of some files from other applications, namely DLLs. DLLs are, in fact, stored in a shared directory, and applications only contain hard links for the DLLs that they use.

The generated database scripts that update the database schema and data are also executed in the Runtime Database, which is running in the Database Server.

Finally, Service Center sends feedback to Service Studio, so that the developer can open the application or be informed if any error occurred during the Publication process.

4.4 Lazy Loading

It is worth noting that, in the OutSystems Platform, the whole application model is not loaded at once. Instead, Lazy Loading is used, a mechanism that loads only parts of the model, as needed. This mechanism is used by both Service Studio and the Compiler. In fact, the application model is not stored in a single file. Instead, the model is divided in different parts called fragments and each fragment is stored in a different file. The files are loaded when the fragment that they store is needed. There are two types of fragments, the root fragment, which is the eSpace, and collection fragments. The root fragment has collection fragments as children. Collection fragments contain a collection of objects. The parent of a collection fragment stores a boolean attribute that indicates whether it has elements or not. This is useful because if a collection fragment does not have elements, it is not necessary to load it. By convention, the names of the files of collection fragments are in the format "name_of_collection.key_of_parent_object.xml".

Taking the model in listing 4.6 as an example, it would be stored as follows. The eSpace is a root fragment, so it would be stored in a file with name espace.xml and its contents can be seen in listing 4.7.

Listing 4.7: espace.xml

```
<oml>
  <espace key="6e19b1f5-aede-4c7f-8b89-ee76ec49c7ba" compilationHash="30a46a0c-0ff3-4f61-
      ↪ a801-7512d3ea7a4a">
    <uiFlows hasChildren="true" />

    <clientVariables hasChildren="true" />

    <compilationUnits hasChildren="true" />
  </espace>
</oml>
```

As can be seen, uiFlows, clientVariables and compilationUnits are collection fragments and their elements are stored in different files with names uiFlows.xml, clientVariables.xml and compilationUnits.xml, respectively. In these cases, the name of the parent object is not included in the name of the file, because the parent object is the eSpace. Their contents can be seen in listings 4.8, 4.9 and 4.10.

Listing 4.8: uiFlows.xml

```
<oml>
  <uiFlows>
    <uiFlow name="Employees" key="251ee3e2-6626-4ed7-a115-ee17cc11d5e2" compilationHash="0
        ↪ f125b27-2995-4f4d-b02d-4156cf37317c">
      <nodes hasChildren="true" />
    </uiFlow>

    <uiFlow name="Departments" key="ab3367b1-8a87-47f6-80a3-8c36f6172d4e" compilationHash=
        ↪ "de89100d-f08f-475c-850b-184bc022663d">
```

```
      <nodes hasChildren="true" />
    </uiFlow>
  </uiFlows>
</oml>
```

Listing 4.9: clientVariables.xml

```
<oml>
  <clientVariables>
    <clientVariable name="Username" key="e02c03ba-0b42-4158-b1b0-0573ac47cc47" />
  </clientVariables>
</oml>
```

Listing 4.10: compilationUnits.xml

```
<oml>
  <compilationUnits>
    <compilationUnit path="/" compilationHash="30a46a0c-0ff3-4f61-a801-7512d3ea7a4a" />
    <compilationUnit path="/uiFlows/251ee3e2-6626-4ed7-a115-ee17cc11d5e2" compilationHash=
        ↪ "0f125b27-2995-4f4d-b02d-4156cf37317c" />
    <compilationUnit path="/uiFlows/ab3367b1-8a87-47f6-80a3-8c36f6172d4e" compilationHash=
        ↪ "de89100d-f08f-475c-850b-184bc022663d" />
    <compilationUnit path="/uiFlows/251ee3e2-6626-4ed7-a115-ee17cc11d5e2/screens/9a640ed5-
        ↪ cba0-4218-b6ee-f4ff0e55c05f" compilationHash="b69f1aab-7a87-4d8e-b67c-645
        ↪ f95698a77" />
    <compilationUnit path="/uiFlows/ab3367b1-8a87-47f6-80a3-8c36f6172d4e/screens/54d37874
        ↪ -1957-422f-a1f3-a6fdc368d787" compilationHash="bb34e5a8-bd6c-42ef-866e-8
        ↪ d4c591446fd" />
  </compilationUnit>
</oml>
```

Now each UIFlow has a nodes collection fragment and each is stored in a different file. Thus, there will be two files, nodes.251ee3e2-6626-4ed7-a115-ee17cc11d5e2.xml and nodes.ab3367b1-8a87-47f6-80a3-8c36f6172d4e.xml, that can be seen in listings 4.11 and 4.12.

Listing 4.11: nodes.251ee3e2-6626-4ed7-a115-ee17cc11d5e2.xml

```
<oml>
  <nodes>
    <screen name="EmployeeList" key="9a640ed5-cba0-4218-b6ee-f4ff0e55c05f" compilationHash
        ↪ ="b69f1aab-7a87-4d8e-b67c-645f95698a77">
      <aggregates>
        <aggregate name="EmployeeAggregate" key="37e0ad08-4eed-435d-965c-7295f92c5b31">
      </aggregates>

      <widgets hasChildren="true" />
    </screen>
  </nodes>
</oml>
```

Listing 4.12: nodes.ab3367b1-8a87-47f6-80a3-8c36f6172d4e.xml

```
1 <oml>
2   <nodes>
3     <screen name="DepartmentList" key="54d37874-1957-422f-a1f3-a6fdc368d787"
          ↪ compilationHash="bb34e5a8-bd6c-42ef-866e-8d4c591446fd">
4       <widgets hasChildren="true" />
5     </screen>
6   </nodes>
7 </oml>
```

Finally, both these nodes collection fragments contain a single screen and both screens have a widgets collection fragment. Therefore, they will be stored in files widgets.9a640ed5-cba0-4218-b6ee-f4ff0e55c05f.xml and widgets.54d37874-1957-422f-a1f3-a6fdc368d787.xml, whose contents are presented in listings 4.13 and 4.14.

Listing 4.13: widgets.9a640ed5-cba0-4218-b6ee-f4ff0e55c05f.xml

```
1 <oml>
2   <widgets>
3     <textWidget name="EmployeeListWidget" key="474f61e2-6094-4e38-9e1c-908a26dab283" />
4   </widgets>
5 </oml>
```

Listing 4.14: widgets.54d37874-1957-422f-a1f3-a6fdc368d787.xml

```
1 <oml>
2   <widgets>
3     <textWidget name="DepartmentListWidget" key="1584a57c-a397-4853-bf81-49490a1574d0" />
4   </widgets>
5 </oml>
```

It is also worth noting that the aggregates collection in the EmployeeList screen is not a fragment. It is stored within the nodes fragment and is loaded with it (as can be seen in lines 4 to 6 of listing 4.11). When deciding which elements should be fragments, the goal was to achieve a balance between the amount of fragments and the dimension of each fragment. These design choices were also based on the requirements of Service Studio. When editing a screen, for example, the developer can see all the screens in the same UI Flow and their respective aggregates, in the element tree. Thus, it makes sense that these elements should be loaded together. This is why they are part of the same fragment.

One might ask why the fragments are stored in separate files. There could be a single XML file and the goal of only loading the needed parts of the model into memory could still be achieved. However, this would require to have the entire XML file in memory, and the file could be significantly large. This is why each fragment is stored in its separate XML file.

4.5 Developer Workflow

As pointed out in Miguel Pires' thesis [9], a typical OutSystems developer's workflow can be described as a change-publish-validate cycle, as depicted in figure 4.4. The developer makes changes to the application model using Service Studio, publishes the application, and validates the results by testing the deployed application. This process is used during the development and maintenance phases, which constitutes the entire application's lifetime.

During a development session, Service Studio continuously validates the changes applied to the model and informs the developer, with error and warning messages in real-time, if there are errors in the application (i.e. the model is not valid). The developer can only trigger the Publication of the application when the model is valid.

As previously mentioned in chapter 3, for large applications, the publishing phase of this cycle can take a significant amount of time and interfere in the developer's workflow. Every time the developer wants to test and validate the results of their changes, they have to stop and wait for the Publication process to complete. Furthermore, since the application has to be relaunched, all state is lost. This severely impacts the experience when a developer is testing a feature that depends on state, since they have to recreate the state every time the application is published.

In conclusion, reducing the time between the moment a change is made and the moment the developer can see and test the results is important and is the problem we are aiming to solve. This time gap consists mainly of rebuilding and executing the application (encompassed in the Publication process) and guiding the application to the place where the changes can be observed. In this work we will focus on making the transportation, loading and compilation of the application model more efficient.

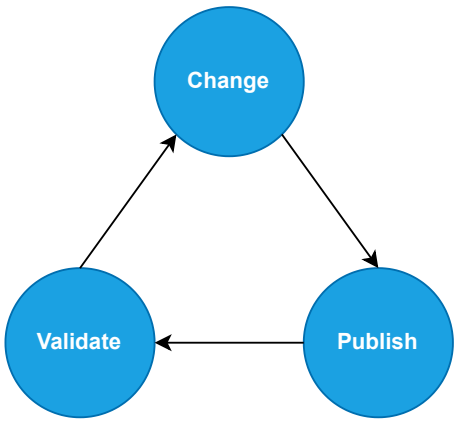

Figure 4.4: Developer workflow

5

Proposed Solutions

In this chapter, we will propose two different approaches to solve the identified problem and present a description of each. Both solutions focus on well-defined part of the compilation process, the transportation and loading of the model.

5.1 Shared Model

This is the most disruptive approach, in relation with the current architecture of the OutSystems Platform, explained in chapter 4. Instead of the IDE sending the application model to the Compiler, the model would be shared by both, and a service would expose an API to access and make changes to the model. Some functionalities available in the IDE require many changes to the model, which, with this approach, translates to many calls to the API. Therefore, it would no longer be viable to have the IDE and the Compiler running in different processes. This requires a new IDE whose backend runs in the OutSystems Platform Server, like the Compiler, so both are close to the shared model. The developer would access the IDE via a web frontend, instead of it being a local application like Service Studio. A diagram of this architecture can be seen in figure 5.1. It is worth mentioning that a new IDE with an architecture similar to this is currently being developed, under the name of Project Morpheus [41], that is focused on Artificial Intelligence (AI)-powered assistance. This poses a problem because, in the current state of the OutSystems Platform, changing the model and compilation cannot happen simultaneously (i.e. they block each other). This is because if changes to the model are made while the Compiler is traversing the model, it could lead to errors and inconsistencies.

To tackle this, a versioning of the model is required. Independent versions of the same model need to be able to exist. This way, if the Compiler is in the process of compiling a version of the model and changes are made to the model, these changes are not immediately seen by the Compiler. Instead, they are stored in a new version, which the Compiler will only be able to see and compile after it finishes the current compilation.

The undo/redo mechanism already keeps implicit information about previous versions. However, this is not sufficient since, as explained before, explicit independent versions of

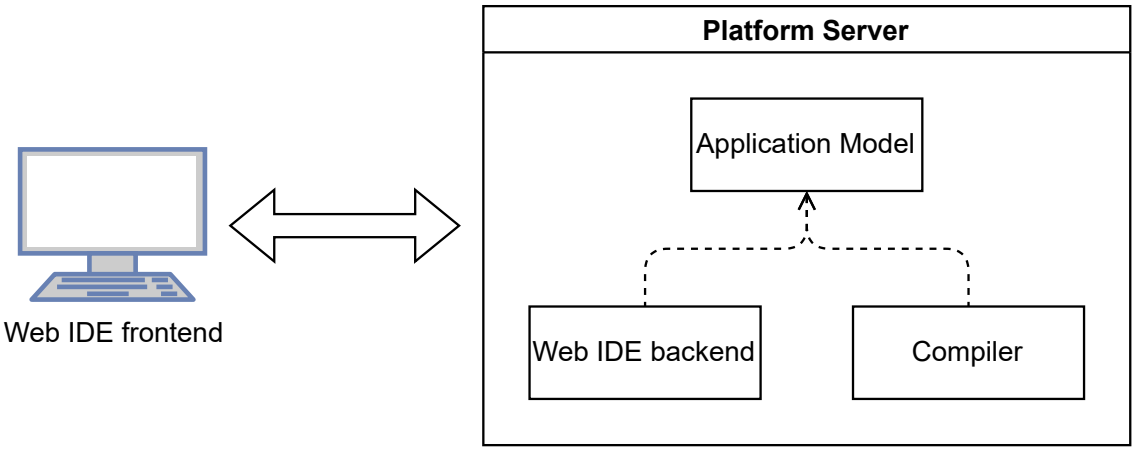

Figure 5.1: Shared Model approach

the same model are needed.

One way of achieving this is to simply load multiple versions of the model. This solution, however, is not very efficient, because every version would contain all the elements of the model, even the ones that were not modified between versions. This would create duplication of information, using unnecessary memory.

A better solution is already being developed within OutSystems and we can extend it for our work. This solution makes use of Persistent Data Structures mechanisms, previously mentioned in chapter 3. A single shared instance of the model exists, but with added information to support multiple versions. When elements from a collection are added or removed, instead of modifying the collection, a new one is created that reflects these changes. Also, information is kept about the attributes of each element that were changed, along with their new values. To better illustrate how this mechanism works, we can use an example application model in figures 5.2 and 5.3. In the first version of the application, version v0, its name is "MyApp" and it has a set of screens (l1), containing just one screen, named "Employees".

Then, the name of the application was changed to "Directory" and a new screen was created, named "Details". These changes were committed and belong to a new version of the application, v1. As can be seen in figure 5.3 highlighted in green, the new screen is created, as well as a new set of screens (l2) that contains both screens, while the original set is kept unchanged. Furthermore, the model contains information about new values for some attributes, namely the new name and the new set of screens of the application element a. The original name and set are also kept unchanged. Finally, the name of the screen s1 was changed to "List" and this change was committed, creating version v2. In this case, the only necessary information to add to the model about v2 is the new name of element s1 (highlighted in yellow in figure 5.3).

This way, information about all versions of the model is kept in a single instance and

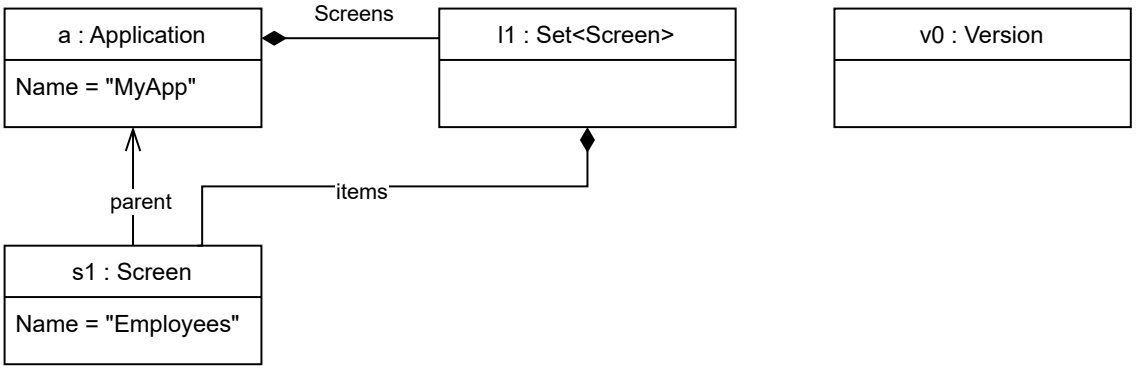

Figure 5.2: Example application model with one version

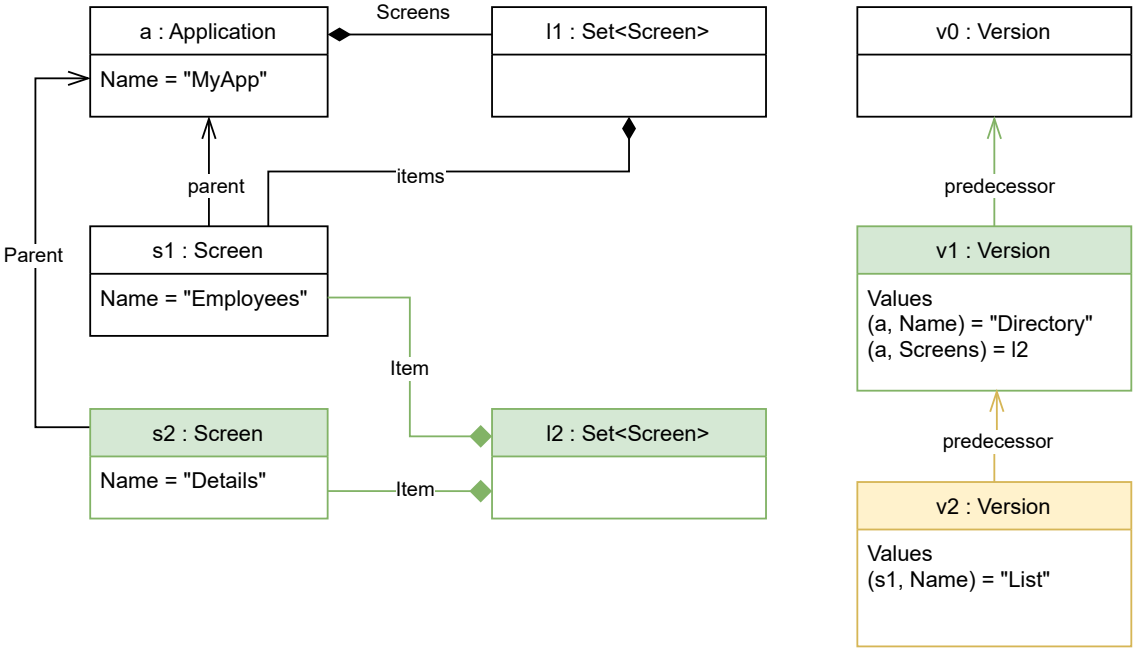

Figure 5.3: Example application model with multiple versions

the IDE and the Compiler can work on different versions of the model simultaneously. The Compiler will work with the latest committed version. To get the value for an attribute, they just have to do a lookup on the version nodes, up to the version they are working on, to check if a new value for that attribute is available. Changes to attributes in following versions are ignored.

This approach eliminates the need for transporting the model from the IDE to the Compiler and for the Compiler to load it.

We were addressing a concurrency problem with one writer (the IDE) and one reader (the Compiler). However, concurrency problems with multiple writers can also arise when there is collaboration (i.e. more than one instance of the IDE frontend are making

changes to the application). This problem is out of the scope of this work, however, since it is not part of the responsibilities of the Compiler. There needs to be a way of merging the changes made by the multiple writers, but the Compiler would already receive the result of this merge.

5.2 Keeping the Compiler running

This approach is the closest to the current architecture of the OutSystems Platform. In the current state of the platform, when an application is published, a new instance of the Compiler is created. After the compilation is finished, this instance is terminated.

In this approach, we propose that the instance is kept running for some amount of time. The optimal amount of time will have to be determined based on factors like the time expected between consecutive changes to an application and ensuring that too many resources from the Platform Server are not consumed. Empirical tests will also have to be performed, to ensure that the chosen amount of time meets the necessary requirements.

Another main aspect of this solution that we propose to implement is that, instead of sending and loading the entire model, only the changed parts of the model would be sent. This requires developing a way to calculate the changes made to a model, i.e. a Δ (delta). The versioning mechanism introduced in the previous approach could also be used here, to send only the parts of the model that were changed. Service Studio would then be able, if the developer so chooses, to send the Δ in the background, every time changes to the application are made and the application is in a valid state.

This solution reduces the amount of data sent in the transportation of the model, as well as the amount of data that the Compiler has to load. We will take the diagram in figure 5.3 as an example. If the model originally loaded by the Compiler is version v0, after the two changes, it is only necessary to transport the elements corresponding to versions v1 and v2 (highlighted in green and yellow). With this information, the Compiler does not need to reload the entire model. It only needs to create the new elements in memory and add v1 and v2 to the versions list. The Compiler will then use v2 as the active version. As a result, it will be able to see the new version of the model.

6

Prototype Implementation

In this chapter, we present our prototype implementation process, emphasizing the relevant technical details. The development of our prototype can be divided into four phases. Firstly, it was necessary to implement a simplified OutSystems meta-model. The reasons behind this are that not only is the OutSystems meta-model (the implementation used by Service Studio) very complex, which would hinder the prototyping of our solutions, but we also needed to use the versioning solution mentioned in the previous chapter, which is also being prototyped within OutSystems. Since this versioning solution is not yet ready to be integrated with the OutSystems meta-model, we had to create our meta-model. It is important to note that we did not develop the versioning solution. We simply use it to solve some of the challenges posed by our solutions. Secondly, we implemented an abstraction of the OutSystems compiler. We started by converting the abstraction we had previously implemented in Java (presented in Chapter 4) to the .NET framework and then developed it further. We developed our prototype in the .NET framework because, as already mentioned, we wanted to integrate it with the versioning solution, which is also being developed in .NET. The third and fourth phases correspond to the implementation of the two solutions previously discussed.

6.1 Meta-model

For our simplified meta-model, we decided to include the following elements:

1. eSpace
2. UI Flow
3. Screen
4. Text Widget
5. Image Widget
6. Link Widget

We created a meta-model library with interfaces and classes that represent these elements, and that exposes an API for creating and manipulating an application's model. A class diagram representing these interfaces and classes, their properties, and the relations between them is shown in Figure 6.1 (with interfaces highlighted in blue and classes in green).

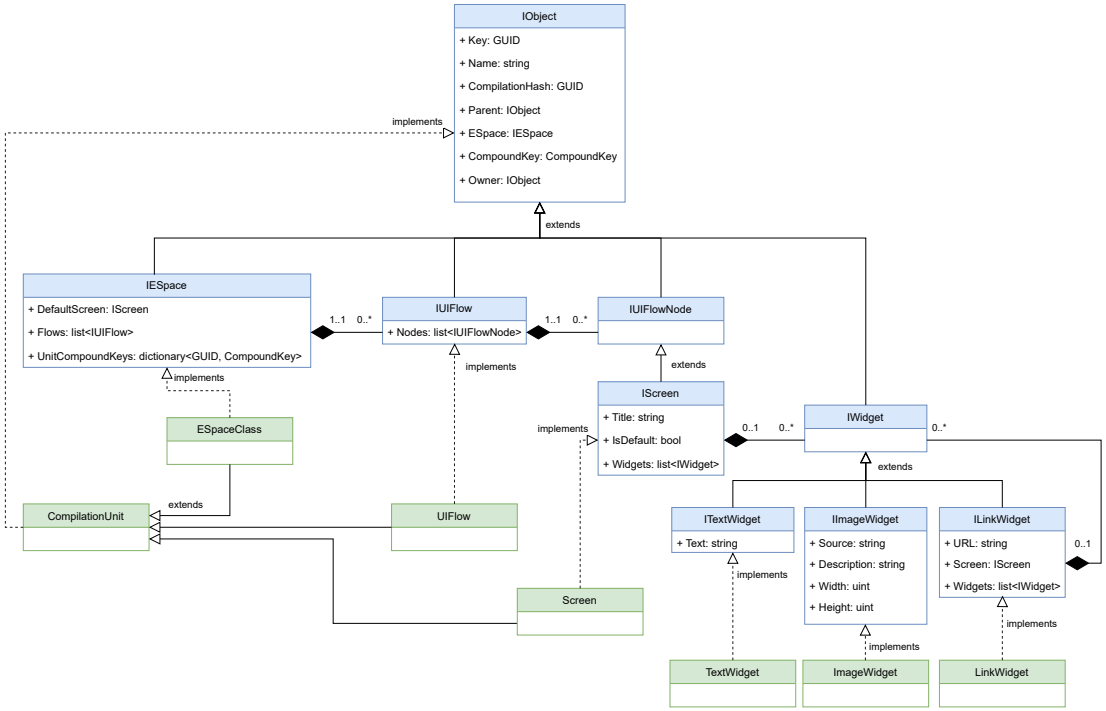

Figure 6.1: Meta-model class diagram

We created the interface IObject, which all elements from our meta-model extend. This interface contains the properties that every element must have: a unique key (which is a GUID), a name, a compilation hash (also a GUID), its parent object, the eSpace it belongs to and a Compound Key. The Compound Key contains the object's key, preceded by the keys of all the objects that form the object's path within the meta-model tree. By knowing the Compound Key of an object, it is possible to find that object, starting at the root of the application (i.e., the eSpace). We will explain its use later in this section. The IObject interface also has a property called Owner. For every object, it is the Compilation Unit it belongs to, and if the object is a Compilation Unit, then its Owner is itself. When producing the artifacts for an object, these artifacts will be associated with the object's Owner. This way, it is possible to keep track of the artifacts produced for each Compilation Unit and its children that are not Compilation Units. We will explain this in more detail when discussing the implementation of the compiler.

We also created the abstract class Compilation Unit, which contains some common logic to all elements that are Compilation Units. In our meta-model, the ESpaceClass, UIFlow, and Screen classes are its subclasses.

We implemented saving and loading operations to and from disk for our model. We did it by serializing and deserializing the model to a JSON file. We chose JSON for its simplicity and for being easy for humans to read, which helped us inspect and debug the results. Listing 6.1 shows a JSON file containing the serialized version of an example application using our meta-model. In our meta-model, an application must have one and only one eSpace, which is the root of the application, as we can observe in the example application. In the example, the eSpace has one UI Flow. The UI Flow has one Screen, named screen1, with three Widgets: one Text Widget, one Image Widget, and one Link Widget. The Link Widget contains another Text Widget.

Listing 6.1: example-app.json

```
{
  "key": "314c6722-eafc-5db1-3ac4-fb665ad6cc0f",
  "typeName": "ESpaceClass",
  "name": "example-app",
  "ownerKey": "314c6722-eafc-5db1-3ac4-fb665ad6cc0f",
  "ownerType": "ESpaceClass",
  "defaultScreen": "c02a48a6-af38-49bf-1439-5a57dcc7f33c",
  "flows": [
    {
      "key": "b865ee99-001b-2814-a312-16c532721767",
      "typeName": "UIFlow",
      "name": "",
      "ownerKey": "b865ee99-001b-2814-a312-16c532721767",
      "ownerType": "UIFlow",
      "nodes": [
        {
          "key": "c02a48a6-af38-49bf-1439-5a57dcc7f33c",
          "typeName": "Screen",
          "name": "screen1",
          "ownerKey": "c02a48a6-af38-49bf-1439-5a57dcc7f33c",
          "ownerType": "Screen",
          "title": "screen1",
          "isDefault": true,
          "widgets": [
            {
              "key": "2757e2b8-3a2e-f04c-a7ba-e35269d3cd3a",
              "typeName": "TextWidget",
              "name": "",
              "ownerKey": "c02a48a6-af38-49bf-1439-5a57dcc7f33c",
              "ownerType": "Screen",
              "text": "Hello, World!"
            },
            {
              "key": "a32ede7e-58cc-2f7b-9fae-53d7e1cabcf8",
              "typeName": "ImageWidget",
              "name": "",
              "ownerKey": "c02a48a6-af38-49bf-1439-5a57dcc7f33c",
```

```
            "ownerType": "Screen",
            "source": "https://www.someimageurl.com",
            "description": "Image",
            "width": 640,
            "height": 360
          },
          {
            "key": "ddf4e3ce-4f66-6ec2-ad7c-e8e490722463",
            "typeName": "LinkWidget",
            "name": "",
            "ownerKey": "c02a48a6-af38-49bf-1439-5a57dcc7f33c",
            "ownerType": "Screen",
            "url": "http://www.example.com",
            "screen": "",
            "widgets": [
              {
                "key": "9e2239d8-d544-0b73-06bf-22884d3efd9f",
                "typeName": "TextWidget",
                "name": "",
                "ownerKey": "c02a48a6-af38-49bf-1439-5a57dcc7f33c",
                "ownerType": "Screen",
                "text": "Link"
              }
            ]
          }
        ]
      }
    ]
  },
  "compilationUnitCompoundKeys": [
    {
      "key": "314c6722-eafc-5db1-3ac4-fb665ad6cc0f",
      "compoundKey": "/314c6722-eafc-5db1-3ac4-fb665ad6cc0f"
    },
    {
      "key": "b865ee99-001b-2814-a312-16c532721767",
      "compoundKey": "/314c6722-eafc-5db1-3ac4-fb665ad6cc0f/b865ee99-001b-2814-a312-16
          ↪ c532721767"
    },
    {
      "key": "c02a48a6-af38-49bf-1439-5a57dcc7f33c",
      "compoundKey": "/314c6722-eafc-5db1-3ac4-fb665ad6cc0f/b865ee99-001b-2814-a312-16
          ↪ c532721767/c02a48a6-af38-49bf-1439-5a57dcc7f33c"
    }
  ]
}
```


It is important to note that when implementing the serialization and deserialization of our meta-model, the performance and efficiency of these operations were not our primary

concerns. Our main goal was to have a working prototype. This could impact the duration of these operations, which we will present in detail in the next Chapter. Improving their efficiency could be a challenge for future work.

Figure 6.2 shows what this application looks like after compiling it, running it, and accessing it in a web browser.

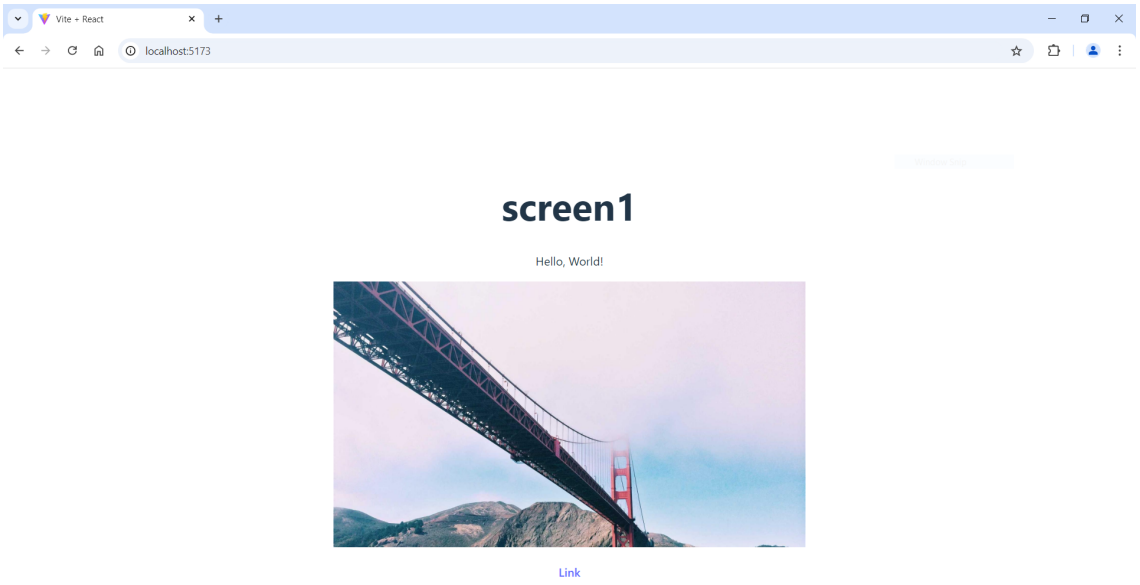

Figure 6.2: Example app created with our prototype

6.1.1 IESpace and ESpaceClass

The eSpace element is represented by the IESpace interface. This interface has the following properties: the dictionary UnitCompoundKeys, which keeps the Compound Keys of all the Compilation Units that belong to the eSpace (the keys of the Compilation Units are stored as keys of the dictionary, and their Compound Keys as values); Flows, which is the collection of UI Flows of the eSpace; and DefaultScreen, which, as the name suggests, stores the application's default or home screen. The IESpace interface is implemented by the class ESpaceClass.

If one wants to access an application's element only by knowing its key, one approach would be to start at the application's root and traverse all its elements until the element is found. However, this search operation could be very costly in large applications. This operation is also frequently performed to access Compilation Units (during the cleanup process, for example). The purpose of the UnitCompoundKeys dictionary is to make it possible to retrieve a Compilation Unit's Compound Key by knowing its key. As previously

explained, the Compound Key is an object's path in the application model. Therefore, knowing it can significantly reduce the search cost.

6.1.2 IUIFlow and UIFlow

The UI Flow element is represented by the IUIFlow interface. Its property Nodes is the collection of UI Flow Nodes that belong to the UI Flow. Its implementing class is the UIFlow class.

6.1.3 IUIFlowNode, IScreen and Screen

In OutSystems, UI Flows contain UI Flow Nodes, which can be of several different types. In our simplified model, we created the IUIFlowNode interface to capture this aspect of the OutSystems meta-model. However, the only type of UI Flow Node that we decided to include is the Screen. It is represented by the IScreen interface, which extends IUIFlowNode. This interface has a property Title, which is a string representing the screen's title, and a property IsDefault, which is a boolean indicating whether the Screen is the application's home screen. It also has the property Widgets, which is the collection of the Widgets belonging to the Screen. It is implemented by the Screen class.

6.1.4 IWidget

Similar to UI Flow Nodes, several types of Widgets exist in the OutSystems meta-model. To represent this concept, we created the IWidget interface. We decided to include in our model three types of Widgets, represented by the interfaces presented subsequently, which all extend IWidget.

6.1.4.1 ITextWidget and TextWidget

Text Widgets are represented by the ITextWidget interface. It has a property called Text, a string corresponding to the widget's text. The class that implements it is TextWidget.

6.1.4.2 IImageWidget and ImageWidget

Image Widgets are represented by the IImageWidget interface. Its properties correspond to the source of the image, its description, and its dimensions, width, and height. ImageWidget is the class that implements it.

6.1.4.3 ILinkWidget and LinkWidget

Link Widgets are represented by the ILinkWidget interface. In our implementation, a Link Widget can be a link to another Screen in the application or an external URL. Therefore, this interface has a URL property, which is a string, and a Screen property, which is an object of type IScreen. Link Widgets can also contain other Widgets, so the interface has

a property called Widgets, similar to IScreen, which is the collection of its Widgets. The ILinkWidget interface is implemented by the LinkWidget class.

6.2 Compiler

As we previously stated, the Compiler's implementation in .NET was based on the version developed in Java to help explain the OutSystems 11 compilation process. Therefore, it works similarly to what was already explained in Chapter 4. We created a Compiler library with a Compiler static class that has a method responsible for compiling an application, the CompileApp method. This method starts by checking the existence of a Compilation Summary. If one does not exist, it means the application is being compiled for the first time, and a new Summary will be created. If one exists but the Compiler's version has changed, all previously created artifacts are deleted, and the application will be recompiled. A cleanup is performed if a Summary exists and the Compiler's version has not changed. Then, the model tree will be traversed, compiling and producing the artifacts for each new or modified Compilation Unit.

6.2.1 Compilation Summary

To implement the Compilation Summary, we created the ICompilationSummary interface and the CompilationSummary class, and the ICompilationUnitInfoInSummary interface and CompilationUnitInfoInSummary class. A CompilationUnitInfoInSummary object keeps information about a Compilation Unit's key, Compilation Hash, and a collection of the artifacts created for it. The Compilation Summary is essentially a collection of CompilationUnitInfoInSummary objects. When producing the artifacts for an application element, information about them must be added to the Compilation Summary, associating them with the Compilation Unit responsible for their creation. This is the case where an object's Owner property will be used. To recapitulate, if an object is a Compilation Unit, its Owner is itself. Otherwise, it is the Compilation Unit that owns it. This way, by associating the artifacts created by an object with its Owner, we can guarantee that they are being associated with the responsible Compilation Unit.

6.2.2 React and Vite

In our prototype, an application model is compiled to a React application. We made this decision so we could take advantage of React's components. Each application element will generate a React component, in its own JavaScript file. In React, components can refer to other components inside them, so a component generated by an application element can refer to the components corresponding to its child elements. Furthermore, React applications can be easily set up, and we can also take advantage of HMR to make changes to an application and see the effects in real-time. We used Vite to create, set up, and run React applications. Vite is a build tool with two major parts [47] [26]. The first is a local

development server, with support for HMR. The second one is a build command that bundles code. We use Vite's scaffolding feature to create a React application's directory and necessary assets. To create a new Vite project using a React/JavaScript template, we run the Node.js command shown in 6.2, where "app-name" is the application's name (i.e., the eSpace's name, in our model).

Listing 6.2: Vite's create React app command

```
$ npm create vite@latest app-name -- --template react
```

All artifacts (JavaScript files) generated by an application's elements will be created inside the directory "app-root/src", where "app-root" is the application's root directory, created by Vite. To run an application, we must run the command shown in 6.3 (again, "app-root" should be replaced by the application's root directory).

Listing 6.3: Vite's run app command

```
$ npm --prefix app-root run dev
```

This will start Vite's development server, and then the application can be accessed in a web browser.

6.2.3 Artifact creation

We took advantage of .NET's extension method feature to create the artifacts for each element type. This feature allows one to add new methods to existing types without modifying the original types. This enabled us to decouple the compilation logic from the meta-model logic. We created the Compile extension method for the IObject type, as showed in listing 6.4. This way, all element types will have this additional method (since they all extend IObject), which then uses Dynamic Dispatching to call the appropriate compilation method for each specific type. By casting the obj variable to a dynamic variable, method resolution is deferred until runtime.

Listing 6.4: "Compile" extension method

```
public static void Compile(this IObject obj, ICompilationSummary summary, string
    ↪ artifactsDir = "")
  {
    CompileObject((dynamic) obj, summary, artifactsDir);
  }
```

We then had to implement the CompileObject method for all element types of our meta-model that are Compilation Units. This method first compares its current Compilation Hash with the one in the Compilation Summary. If the hashes are equal, the compilation of the element is skipped, and the method returns. Compilation proceeds if they are different or if the element was not present in the Summary, and the Summary is updated. The method creates the artifacts for the element and calls the Compile method on all its child elements that are Compilation Units.

For the Text Widget, Image Widget, and Link Widget, we create generic, parameterizable React components when compiling the eSpace. This way, a component for each Widget instance will not be created. Instead, a single component for each Widget type will be generated, and each instance will be created by calling the component with its properties as arguments, as shown subsequently. To simplify this process, components for the three Widget types will always be generated, even if the application does not use them. In the real scenario, this kind of reusable artifacts would be in a library and would not be generated by applications.

6.2.3.1 eSpace

The artifact generated for the eSpace will always be named "App.jsx" since Vite assumes this is the application's main component, acting as the entry point for all other React components, which is the desired behavior. The eSpace of the example application shown in Listing 6.1 and Figure 6.2 would generate a file similar to that shown in Listing 6.5.

Listing 6.5: App.jsx

```
1  import React from 'react';
2  import { useState } from 'react';
3  import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
4  import './App.css';
5  import Fb865ee99001b2814a31216c532721767 from './b865ee99-001b-2814-a312-16c532721767';
6
7  function App() {
8    return (
9        <div>
10           <Fb865ee99001b2814a31216c532721767 />
11       </div>
12   );
13 }
14
15 export default App;
```

The code starts by importing the necessary modules and components. It must import the components corresponding to its child elements, the UI Flows, which in this example is only one component named "Fb865ee99001b2814a31216c532721767". We decided to use the object's key as the name of the component instead of the object's name because our prototype does not ensure that names are unique, but keys are guaranteed to be unique. The "App" component is defined by the function with the same name, and all it has to do is return a "div" element containing all the components that correspond to its child elements.

6.2.3.2 UI Flow

The artifacts generated for UI Flows and all other Compilation Unit types will have the key of their corresponding elements as their names, so we can guarantee that each file's

name is unique, as explained previously. The UI Flow of the example application would generate a file similar to that shown in Listing 6.6.

Listing 6.6: b865ee99-001b-2814-a312-16c532721767.jsx

```jsx
import React from 'react';
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
import './App.css';
import Fc02a48a6af3849bf14395a57dcc7f33c from './c02a48a6-af38-49bf-1439-5a57dcc7f33c';

function Fb865ee99001b2814a31216c532721767() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Fc02a48a6af3849bf14395a57dcc7f33c/>} />
        <Route path="/screen1" element={<Fc02a48a6af3849bf14395a57dcc7f33c/>} />
      </Routes>
    </BrowserRouter>
  );
}

export default Fb865ee99001b2814a31216c532721767;
```

Like what happens with the eSpace's component, the code starts by importing the necessary components, including those corresponding to its Screens. UI Flows are mapped to "BrowserRouter" components from the "react-router-dom" library. These components are used to handle routing in React applications. They provide the functionality to manage navigation and URL-based state within a single-page application. We did this to enable navigation between different Screens of the same UI Flow. As it is possible to see in 6.6, The "BrowsweRouter" component contains different "Route" components for each of its Screens, with their corresponding URL paths. In this specific case, the UI Flow has only one Screen. However, since this Screen is the application's home screen, it can be accessed via two paths. One path is "/", effectively making this Screen the home screen. The other is "/screen1", which uses the Screen's name for the path.

It is important to note that an application with multiple UI Flows will compile to a React application with multiple "BrowserRouter" components. Although a React application with multiple "BrowserRouter" components is technically correct and runs without compilation errors, it is generally not recommended and can lead to unintended behavior. The "BrowserRouter" component is designed to manage the history and URL synchronization of the React app, and it should typically wrap the entirety of the application or, at the very least, the top-level routing structure. However, as previously stated, our goal with this prototype is not to generate entirely correct React applications. We aimed to develop a prototype that could generate visible results, which we could then test and demonstrate, and create a proof of concept.

6.2.3.3 Screen

The Screen of the example application generates a file like the one shown in Listing 6.7.

Listing 6.7: c02a48a6-af38-49bf-1439-5a57dcc7f33c.jsx

```jsx
import React from 'react';
import './App.css';
import TextWidget from './TextWidget';
import ImageWidget from './ImageWidget';
import LinkWidget from './LinkWidget';

function Fc02a48a6af3849bf14395a57dcc7f33c() {
  return (
    <div>
      <h1>screen1</h1>
      <TextWidget text="Hello, World!" />
      <ImageWidget source="https://www.someimageurl.com" description="Image" height="360"
          ↪ width="640" />
      <LinkWidget url="http://www.example.com" isExternal={true} >
         <TextWidget text="Link" />
      </LinkWidget>
    </div>
  );
}

export default Fc02a48a6af3849bf14395a57dcc7f33c;
```

It would be possible to perform a pre-processing of the Screen's widgets to know which types need to be imported. However, we decided that, for our prototype, it would be sufficient to always import the three components corresponding to the three Widget types.

The component returns a "div" element containing a heading element, whose text is the Screen's name, as it is possible to see in the page heading shown in Figure 6.2. The "div" element also contains the components corresponding to the Screen's Widgets. As it is possible to see in this example, the generic component for each Widget's type is being called, and its properties are being passed as arguments. One of the arguments of the Link Widget is a Text Widget.

6.2.3.4 Text Widget

The generic component generated for Text Widgets is shown in Listing 6.8.

Listing 6.8: TextWidget.jsx

```jsx
import React from 'react';

function TextWidget(props) {
  return (
    <p>{props.text}</p>
```

```
  );
}

export default TextWidget;
```

This component returns a paragraph element with the Text Widget's text, retrieved from the "props" parameter.

6.2.3.5 Image Widget

The generic component generated for Image Widgets is shown in Listing 6.9.

Listing 6.9: ImageWidget.jsx

```
import React from 'react';

function ImageWidget(props) {
  return (
    <img src={props.source} alt={props.description} width={props.width} height={props.
        ↪ height}/>
  );
}

export default ImageWidget;
```

The component returns an image element with the Image Widget's source URL, description, and dimensions, retrieved from the "props" parameter.

6.2.3.6 Link Widget

The generic component generated for Link Widgets is shown in Listing 6.10.

Listing 6.10: LinkWidget.jsx

```
import React from 'react';
import { Link } from 'react-router-dom';
import './App.css';

function LinkWidget(props) {
  if (props.isExternal) {
    return (
      <a href={props.url}>
        {props.children}
      </a>
    );
  } else {
    return (
      <Link to={props.url}>
        {props.children}
      </Link>
    );
```

```
18    }
19  }
20
21  export default LinkWidget;
```

What is returned by the Link Widget component depends on whether it is a link to an external URL or another Screen of the application. In the former case, an anchor element is returned. The anchor element has a "href" attribute with the Link Widget's destination URL and contains the components corresponding to its child Widgets, retrieved from the "props" parameter. A Link component from the "react-router-dom" library is returned in the latter case. This component has a "to" attribute, which would contain the URL path of the destination Screen using that Screen's name, and also contains the components corresponding to its child Widgets.

6.3 Shared model

As explained in Chapter 5, one of the challenges we needed to overcome with this approach was the concurrency between changing and compiling the model. We wanted these two operations to be able to occur simultaneously, without blocking each other. To achieve this, we first had to integrate the versioning solution already being developed within OutSystems with our meta-model. We also developed a simple console application to serve as an IDE and had to overcome challenges regarding the concurrent use of the compiler.

6.3.1 Versioning solution integration

As was also presented in Chapter 5, the purpose of the versioning solution is to keep information about the values of an object's attributes as they are changed over time by storing information about different versions. When changes are made, a new version is created to store the new values instead of overwriting the original values. When reading an attribute's value, it must first be looked up in the list of versions and, if not found, in the object itself. To this end, this solution uses the concept of Versioned Objects and Transactions. A Versioned Object is an object that has attributes and/or collections that are versioned. Versioned Objects have a Transaction Manager responsible for managing the object's versions. Every read or write operation performed on an object's versioned attributes must be done inside the context of a Transaction through the object's Transaction Manager. Read operations can be performed in a Read-only Transaction. This kind of transaction does not create new versions of the object and accesses the latest committed version. Read-only transactions can also be performed concurrently with other Read-Only Transactions and regular transactions (which support write operations). Regular transactions create new versions of the object that store the new values of the changed attributes.

By integrating this solution with our meta-model, we can make use of the transactions. Since compiling the model accesses the values of its attributes but does not change them, this operation can be performed with a read-only transaction. In contrast, the IDE must use regular transactions to be able to change the model. These two kinds of transactions can be performed concurrently, and the Compiler can only access the values of the latest committed version at the start of the read-only transaction, so it will not be affected by changes being made by the IDE. Thus, this solves the concurrency problem.

To integrate the versioning solution with our meta-model, we had to change the IObject interface to extend the IVersionedObject interface, exposed by the versioning solution library. This way, all our meta-model elements will be Versioned Objects since they all extend IObject. Then, we had to change the implementation of the elements' properties that must be versioned, in their implementing classes. Before, we used simple getters and setters. Whereas now, accessing and changing a versioned property must be done through the object's Transaction Manager. We can use the Screen class as an example to clarify this. The Screen's Title property was initially implemented using the default getters and setters, as seen in Listing 6.11.

Listing 6.11: Original Title property implementation

```
1 public string Title { get; set; }
```

Since this property must be versioned, it is now implemented as shown in Listing 6.12.

Listing 6.12: New Title property implementation

```
1  private string _title;
2  private static readonly Field<Screen, string> TitleField = new(s => s._title, (s, v) => s.
       ↪ _title = v);
3  public string Title
4  {
5    get => TransactionsManager.GetValue(this, TitleField);
6    set
7    {
8      TransactionsManager.SetValue(this, TitleField, value);
9      InvalidateCompilationHash();
10   }
11 }
```

We will not discuss the implementation details of the versioning solution since it is outside this work's scope. However, it is possible to see that now access to the value of the Title property (for reading or writing) is done via the Transactions Manager of the Screen object. We also had to change the implementation of the collections of our model's elements. We replaced the .NET default collections' implementations, mainly List and Dictionary, with the ones provided by the versioning solution library, VersionedList and VersionedDictionary, respectively. These collections are Versioned Objects themselves.

6.3.2 IDE console app

The Shared Model approach has two components, the compiler and the IDE, that share the application model. Having already created a compiler implementation for our prototype, we created an IDE for this approach. We developed a simple IDE console app, shown in the screenshot of Figure 6.3.

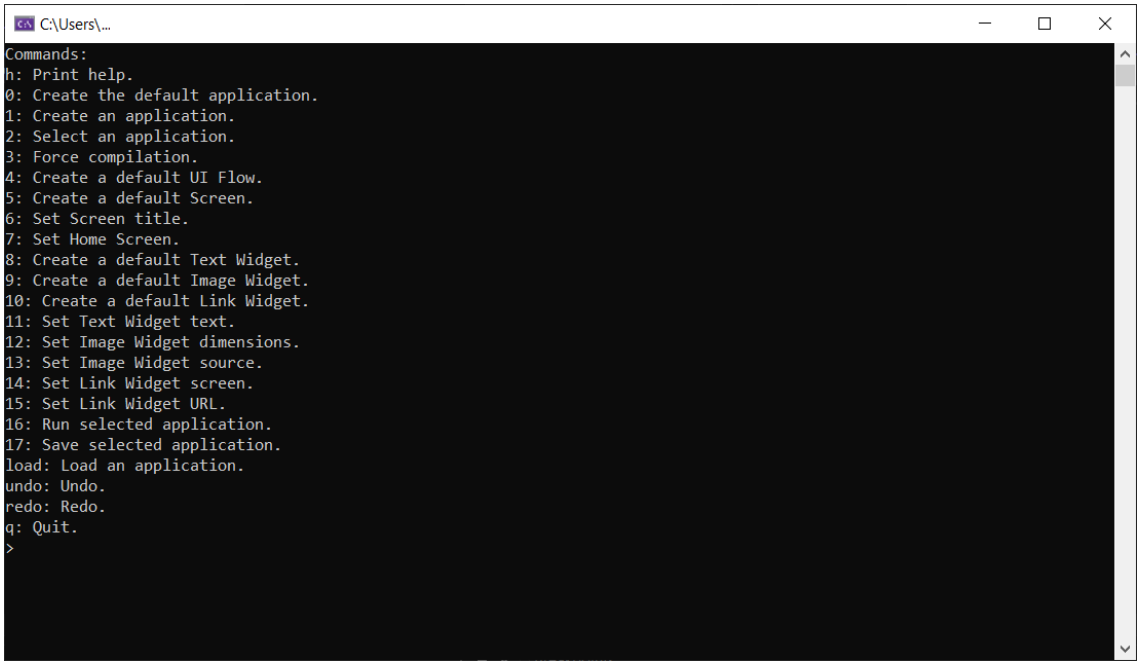

Figure 6.3: Shared model IDE console app screenshot

We did not create a full-fledged IDE since that would be out of the scope of this work. We intended to have a way of simulating the editing of a model concurrently with its compilation. Since we are not dealing with the actual OutSystem meta-model, we cannot use Service Studio, so we created this abstraction. It supports many operations, including: The creation of applications; Adding UI Flows, Screens, Text Widgets, Image Widgets, and Link Widgets; Modifying the properties of Widgets; Running applications; Saving and loading applications.

We wanted the Compiler to react automatically to changes made to an application model. To achieve this, we used Events. In the .NET framework, an Event is a message sent by an object to signal the occurrence of an action [20]. The versioning solution's Transactions Manager already raises Events for each executed step of a Transaction. In our Compiler class, we created a method for the Compiler to subscribe to these Events, as shown in Listing 6.13.

Listing 6.13: SubscribeToTransactionStepEvent method (Compiler class)

```
1 public static void SubscribeToTransactionStepEvent(IESpace eSpace)
2 {
```

```
  eSpace.TransactionsManager.ExecutedStep += TransactionStepExecuted;
}
```

This method takes an eSpace (i.e., the root of an application) as a parameter and subscribes the Compiler to the "ExecutedStep" event of the eSpace's Transactions Manager. Effectively, the "TransactionStepExecuted" method will be called whenever the "ExecutedStep" event is raised. The IDE has to call this method of the Compiler static class when it needs to subscribe the Compiler to an application's events, as shown in Listing 6.14.

Listing 6.14: Subscribing the Compiler to an application's events

```
Compiler.SubscribeToTransactionStepEvent(eSpace);
```

The "TransactionStepExecuted" method is responsible for triggering the compilation of the application. However, it starts by doing some verifications, as shown in Listing 6.15.

Listing 6.15: TransactionStepExecuted method (Compiler class)

```
public static void TransactionStepExecuted(TransactionsManager sender,
    ↪ TransactionEventArgs e)
{
  if (e.IsReadOnly == true)
    return;

  if (e.Step.Equals(TransactionStep.InsideTransaction))
    return;
  .
  .
  .
```

Since we do not want to compile the application after read-only transactions, the method returns if the transaction is of this kind (the "IsReadOnly" property is compared with "true" because it is of a nullable type). The method also returns if the Transaction is not finished because we also do not want to trigger the compilation in this case.

6.3.3 Compiler concurrency

At this point in the development, we faced a concurrency challenge. Different IDEs might modify different applications' models simultaneously, resulting in a concurrent usage of the Compiler. If only one instance of the Compiler is launched, each compilation request would have to wait for the previous one to finish before it starts. This is not desirable since, in a real-world scenario, many developers are working simultaneously on their applications, and it would not be reasonable only to compile one application at a time. So, we decided we had to support running multiple instances of the Compiler concurrently. However, we also need to guarantee that the same application is not compiled concurrently by different instances of the Compiler if, for example, the same IDE makes several changes to a model in sequence. This is important because if more than one thread tries to compile

the same application simultaneously, it could lead to inconsistencies and errors since they may access different versions of the model or try to write to the same files concurrently.

To implement this, we had to create an auxiliary data structure, the CompilationQueue, shown in Listing 6.16.

Listing 6.16: CompilationQueue

```
internal class CompilationQueue
{
  private readonly List<IESpace> queue;
  private readonly HashSet<Guid> keySet;

  public CompilationQueue()
  {
    this.queue = [];
    this.keySet = [];
  }

  public int Count() { return queue.Count; }
  public bool Contains(Guid key) { return keySet.Contains(key); }

  public bool Add(IESpace eSpace)
  {
    if (Contains(eSpace.Key)) { return false; }

    queue.Add(eSpace);
    keySet.Add(eSpace.Key);

    return true;
  }

  public IESpace Get(int i) { return queue[i]; }

  public bool RemoveAt(int i)
  {
    if (i < 0 || i >= Count()) { return false; }

    IESpace e = queue[i];
    queue.RemoveAt(i);
    keySet.Remove(e.Key);

    return true;
  }
}
```

This data structure essentially has a list of applications and a set of their names. This makes it more efficient to check if an application already exists in the collection and ensures that it does not contain repeated elements while maintaining their order.

The "TransactionStepExecuted", introduced in 6.15, tries to add an application to the Compilation Queue whenever a Transaction is executed, as shown in Listing 6.17. This

will only be possible if the application is not already in the queue. If an application is in the Compilation Queue, its compilation has not started. When it starts, the Compiler will access the most recent version of the application model. Even if several transactions have been executed, the Compiler will not see the intermediate versions of the model. Thus, adding an application to the queue would be redundant and inefficient if it is already present.

Listing 6.17: TransactionStepExecuted method continuation

```
6   .
7   .
8   .
9   if (sender.Owner is not IESpace eSpace)
10    return;
11
12  Log.Information($"Transaction executed for {eSpace.Name}.");
13  try
14  {
15    Monitor.Enter(queue);
16    if (queue.Add(eSpace))
17    {
18      Log.Information($"{eSpace.Name} added to compilation queue.");
19    }
20    else
21    {
22      Log.Information($"{eSpace.Name} already in compilation queue.");
23    }
24    Monitor.Pulse(queue);
25    Monitor.Exit(queue);
26  } catch (Exception) {}
27  finally
28  {
29    if (Monitor.IsEntered(queue))
30    {
31      Monitor.Exit(queue);
32    }
33  }
34}
```

Notice that a Monitor is being used to restrict access to the section of code that manipulates the Compilation Queue. The Monitor class provides a mechanism that synchronizes access to objects and regions of code by taking and releasing a lock on a particular object [22]. In this case, the "Monitor.Enter" method is used to acquire a lock, pausing execution until a lock can be acquired. The "Monitor.Exit" method is used to release a lock. While a thread owns the lock for an object, no other thread can acquire that lock [22]. This way, access and modification of the Compilation Queue in this code section is thread-safe. Before the lock is released, the "Monitor.Pulse" method is called. This method notifies a thread in the waiting queue to acquire a lock of a change in the

locked object's state. The reason for this will be explained shortly in this section.

To manage the Compilation Queue and launch new threads to compile applications concurrently, we used the Compiler static class's constructor, as shown in Listing 6.18. This method is guaranteed to run automatically and only once. It is executed before any static members of the class are accessed or any static methods are called without the need to be explicitly invoked.

Listing 6.18: Managing the Compilation Queue and launching compilation threads

```
1  private static readonly CompilationQueue queue = new();
2  private static readonly ConcurrentDictionary<Guid, Object?> appsCompiling = [];
3
4  static Compiler()
5  {
6    Task.Run(() =>
7    {
8      while (true)
9      {
10       try
11       {
12         if (!Monitor.IsEntered(queue))
13         {
14           Monitor.Enter(queue);
15         }
16
17         IESpace? eSpace = GetNextAppToCompile();
18         if (eSpace != null)
19         {
20           appsCompiling.TryAdd(eSpace.Key, null);
21
22           Monitor.Exit(queue);
23
24           Task.Run(() =>
25           {
26             eSpace.TransactionsManager.ExecuteReadOnlyTransaction(() =>
27             {
28               CompileApp(eSpace);
29               Log.Information($"{eSpace.Name} compilation finished.");
30             });
31
32             appsCompiling.TryRemove(eSpace.Key, out Object? o);
33
34             try
35             {
36               Monitor.Enter(queue);
37               Monitor.Pulse(queue);
38               Monitor.Exit(queue);
39             } catch (Exception) { }
40             finally
41             {
```

```
42              if (Monitor.IsEntered(queue))
43              {
44                Monitor.Exit(queue);
45              }
46            }
47          });
48        }
49        else
50        {
51          Monitor.Wait(queue);
52        }
53      } catch (Exception) { }
54      finally
55      {
56        if (Monitor.IsEntered(queue))
57        {
58          Monitor.Exit(queue);
59        }
60      }
61    }
62  });
63}
64
65// Must be called with an acquired lock of the compilation queue
66private static IESpace? GetNextAppToCompile()
67{
68  for (int i = 0 ; i < queue.Count() ; i++)
69  {
70    var eSpace = queue.Get(i);
71    if (!appsCompiling.ContainsKey(eSpace.Key))
72    {
73      queue.RemoveAt(i);
74      return eSpace;
75    }
76  }
77
78  return null;
79}
```

Using the Task class, the constructor launches a new thread to execute an operation asynchronously. This operation is an infinite cycle that fetches applications from the Compilation Queue and launches new threads to compile them. We will refer to this thread as the scheduler thread. To achieve this, it starts by trying to acquire a lock over the queue object if it does not already own it, as shown in lines 12 to 14 of Listing 6.18. After the lock is acquired, the execution enters a critical section highlighted in red. This is necessary because this thread will access and potentially modify the Compilation Queue to retrieve an application, and, as seen in Listing 6.17, another thread may try to add applications to the queue concurrently.

In the critical section, the "GetNextAppToCompile" method is called to retrieve an application from the Compilation Queue. This method is also shown in Listing 6.18, starting in line 66. It essentially iterates over the Compilation Queue until it finds an application not currently being compiled. This is done because of the concurrency problem mentioned before. Compilation of one application can only start if no other instance is compiling it. When it finds one, it removes it from the Compilation Queue and returns it. If none is found, the method returns null.

To check whether an application is currently being compiled, we used a collection called "appsCompiling" that stores which applications are being compiled at all times. To implement this collection, we used the ConcurrentDictionary class. It effectively serves as a set since we only use the dictionary's keys to store the applications' keys. We used this class because we needed a thread-safe collection that can be accessed by multiple threads concurrently, and, as of this writing, .NET does not provide an implementation of a concurrent set.

After calling the "GetNextAppToCompile" method, if no application to compile was found, the "Monitor.Wait" method is called (line 51). This method releases the lock on the object and blocks the current thread until it reacquires the lock after being notified by a call to "Monitor.Pulse". One of the occasions when the lock can be reacquired is after the "Monitor.Pulse" call on Listing 6.17, which signifies that a new application has been added to the queue, so it is ready to be fetched.

On the other hand, if an application is found, it is added to the "appsCompiling" collection (line 20). Doing this before the lock is released guarantees the atomicity of removing an application from the Compilation Queue and adding it to the "appsCompiling" collection. After the lock is released, a new thread is launched, which starts a read-only transaction and calls the "CompileApp" method to compile the application (lines 24 to 30).

Between releasing the lock and starting the read-only transaction, new changes to the model can occur, which would cause the application to be added to the Compilation Queue since it has been removed from it. However, this would not cause any errors. When the transaction starts, it will access the most recent version of the application's model. The worst possible consequence would be that if no more changes are made to the application, the Compiler would try to recompile it but find the eSpace's Compilation Hash unchanged, so its compilation would be skipped.

After the application is finished compiling, it is removed from the "appsCompiling" collection (line 32). Then, a new critical section is entered, highlighted in green. This is done to call the "Monitor.Pulse" method. This way, if the application whose compilation finished is in the Compilation Queue, and the scheduler thread is waiting to be notified of changes to the queue, it will get a notification and be able to fetch this application from the queue to compile it once more.

This explanation intuitively demonstrates the correctness of this algorithm, which is thread-safe and allows for the concurrent compilation of multiple applications. This completes the implementation of the shared model approach in our prototype, and we

will move on to the second approach.

6.4 Version serialization

As discussed in Chapter 5, one aspect of the second approach was to send only the changed parts of the model, a delta, instead of the whole model in each Publication. We can take advantage of the versioning solution to achieve this. With this solution, the delta corresponds to the list of versions created since the last Publication. Thus, we had to implement the serialization and deserialization of versions so they could be transported, as the versioning solution did not already implement this. Subsequently, we will present the most relevant aspects of this implementation, starting with the serialization and then moving to the deserialization.

6.4.1 Serialization

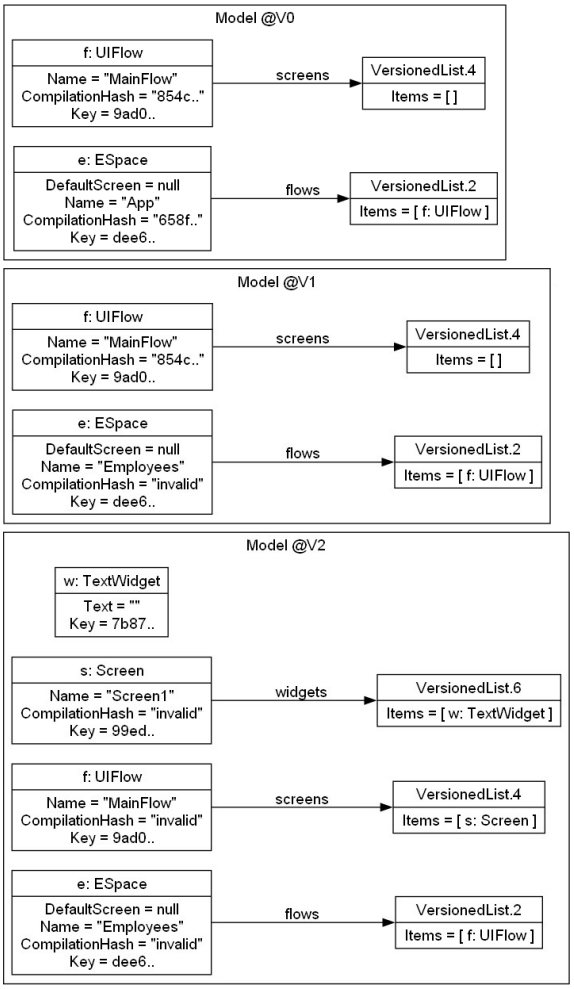

Figure 6.4: Example application with different versions

To help with this explanation, we will use an example application, represented in Figure 6.4.[1] Initially, the application is in version 0. It is named "App" and has a UI Flow without Screens. Then, a Transaction that changes the application's name to "Employees" is performed, creating version 1. Finally, a Transaction is done to add a new Screen to the application's UI Flow, called "Screen1", creating version 2. This new Screen also contains a Text Widget, with an empty string as its text.

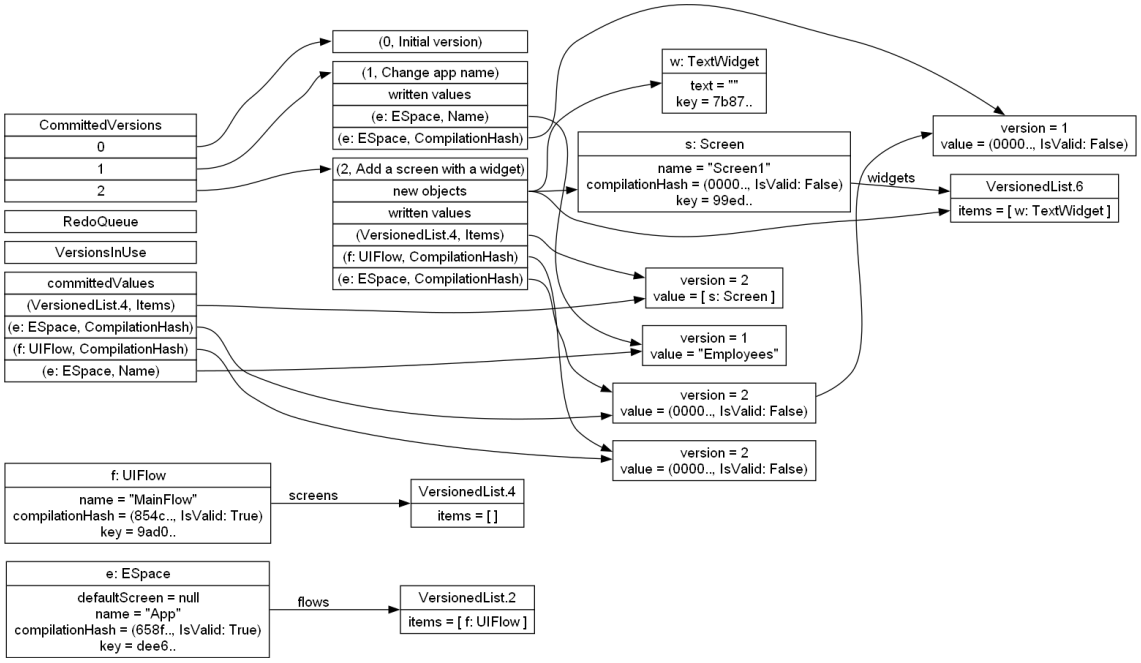

Figure 6.5: Versioning solution's data structures

The diagram in Figure 6.5 represents the data structures the versioning solution uses.[1] Notice that the application object and its UI Flow (with its collection of Screens) remain unmodified. The versioning solution has a collection of committed versions, the first being the initial version. Each of the following versions is created after a Transaction is executed. Each version has a collection of new objects containing the objects created in that version and a collection of written values that contains the modified fields and points to their corresponding new values. Versions also have a deleted objects collection containing the objects deleted in that version. However, in this example, no objects are deleted in any version, so this collection is not represented. In version 1, no new objects are created, so its new objects collection is empty and not in the diagram. On the other hand, the written values collection contains the "Name" field of the application, which points to its new value, "Employees". It also contains the "CompilationHash" field. This is an example of a Calculated Field. A Calculated Field works like a cached field. Its value is stored temporarily (i.e., until it is invalidated) for quick access, avoiding the need to recompute

[1] These diagrams were generated automatically using the Graphviz software [16] and are based on this presentation [31].

it every time. In this case, since the eSpace's name was changed, its Compilation Hash was invalidated. The version represents this information by pointing the field to a value whose "IsValid" flag is set to false.

Version 2's new objects collection contains the new Screen, its Widget collection and the new Text Widget. Its written values collection contains the UI Flow's Screen collection field since this collection was modified to include the newly created Screen. It also contains the "CompilationHash" fields of the eSpace and the UI Flow. Since both these Compilation Units were modified, their hashes were invalidated and that information is represented here.

Listings 6.19 and 6.20 show the equivalent serialized versions 1 and 2, respectively.

Listing 6.19: Serialized version 1

```
{
  "sequentialId": 1,
  "id": 1,
  "name": "Change app name.",
  "isUndoVersion": false,
  "newObjects": [],
  "deletedObjects": [],
  "writtenValues": [
    {
      "objectId": "dee6741a-dc75-b2f4-28e2-4d137bd5b675",
      "field": "Name",
      "type": "OutSystems.Model.Versioning.ValueNode`1[System.String]",
      "valueType": "System.String",
      "value": "Employees"
    },
    {
      "objectId": "dee6741a-dc75-b2f4-28e2-4d137bd5b675",
      "field": "Generation",
      "type": "OutSystems.Model.Versioning.GenerationValueNode",
      "valueType": "System.Int32",
      "value": 1
    },
    {
      "objectId": "dee6741a-dc75-b2f4-28e2-4d137bd5b675",
      "field": "Hash",
      "type": "OutSystems.Model.Versioning.ValueNode`1[OutSystems.Model.Versioning.
          ↪ CalculatedFieldValue`1[System.Guid]]",
      "calculatedFieldValue": {
        "valueType": "System.Guid",
        "isValid": false,
        "value": null
      }
    }
  ]
}
```

Listing 6.20: Serialized version 2


```
{
  "sequentialId": 2,
  "id": 2,
  "name": "Add a screen with a widget.",
  "isUndoVersion": false,
  "newObjects": [
    {
      "id": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a",
      "type": "MetaModelLibrary.Screen",
      "parentId": "9ad0a9c2-7f73-7351-1391-81f48322e85d",
      "parentType": "MetaModelLibrary.UIFlow",
      "object": {
        "generation": 2,
        "name": "Screen1",
        "ownerId": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a",
        "ownerType": "MetaModelLibrary.Screen",
        "title": "",
        "isDefault": false,
        "widgets": "44bad0b2-4a49-ae17-6699-9de8d00eddf0"
      }
    },
    {
      "id": "44bad0b2-4a49-ae17-6699-9de8d00eddf0",
      "type": "OutSystems.Model.Versioning.Collections.VersionedList`1[MetaModelLibrary.
          ↪ IWidget]",
      "parentId": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a",
      "parentType": "MetaModelLibrary.Screen",
      "object": {
        "generation": 2,
        "elements": [
          {
            "element": "7b87f2a5-b4d0-4408-e74b-f17436903b8c",
            "elementType": "MetaModelLibrary.TextWidget"
          }
        ]
      }
    },
    {
      "id": "7b87f2a5-b4d0-4408-e74b-f17436903b8c",
      "type": "MetaModelLibrary.TextWidget",
      "parentId": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a",
      "parentType": "MetaModelLibrary.Screen",
      "object": {
        "generation": 2,
        "name": "Widget1",
        "ownerId": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a",
        "ownerType": "MetaModelLibrary.Screen",
        "text": ""
      }
```

```
    }
  ],
  "deletedObjects": [],
  "writtenValues": [
    {
      "objectId": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a",
      "field": "Generation",
      "type": "OutSystems.Model.Versioning.GenerationValueNode",
      "valueType": "System.Int32",
      "value": 2
    },
    {
      "objectId": "919f87f8-9210-2c38-3280-a8df1d381b39",
      "field": "Items",
      "type": "OutSystems.Model.Versioning.ValueNode`1[System.Collections.Generic.
          ↪ IDictionary`2[System.Guid,MetaModelLibrary.CompoundKey]]",
      "valueType": "System.Collections.Generic.Dictionary`2[System.Guid,MetaModelLibrary.
          ↪ CompoundKey]",
      "value": [
        {
          "keyType": "System.Guid",
          "key": "dee6741a-dc75-b2f4-28e2-4d137bd5b675",
          "valueType": "MetaModelLibrary.CompoundKey",
          "value": {
            "key": "dee6741a-dc75-b2f4-28e2-4d137bd5b675",
            "compoundKey": "/dee6741a-dc75-b2f4-28e2-4d137bd5b675"
          }
        },
        {
          "keyType": "System.Guid",
          "key": "9ad0a9c2-7f73-7351-1391-81f48322e85d",
          "valueType": "MetaModelLibrary.CompoundKey",
          "value": {
            "key": "9ad0a9c2-7f73-7351-1391-81f48322e85d",
            "compoundKey": "/dee6741a-dc75-b2f4-28e2-4d137bd5b675/9ad0a9c2-7f73
                ↪ -7351-1391-81f48322e85d"
          }
        },
        {
          "keyType": "System.Guid",
          "key": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a",
          "valueType": "MetaModelLibrary.CompoundKey",
          "value": {
            "key": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a",
            "compoundKey": "/dee6741a-dc75-b2f4-28e2-4d137bd5b675/9ad0a9c2-7f73
                ↪ -7351-1391-81f48322e85d/99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a"
          }
        }
      ]
    },
```

```
    {
      "objectId": "919f87f8-9210-2c38-3280-a8df1d381b39",
      "field": "Generation",
      "type": "OutSystems.Model.Versioning.GenerationValueNode",
      "valueType": "System.Int32",
      "value": 2
    },
    {
      "objectId": "dee6741a-dc75-b2f4-28e2-4d137bd5b675",
      "field": "Generation",
      "type": "OutSystems.Model.Versioning.GenerationValueNode",
      "valueType": "System.Int32",
      "value": 2
    },
    {
      "objectId": "44bad0b2-4a49-ae17-6699-9de8d00eddf0",
      "field": "Generation",
      "type": "OutSystems.Model.Versioning.GenerationValueNode",
      "valueType": "System.Int32",
      "value": 2
    },
    {
      "objectId": "5bac5cef-5f63-9bd2-be94-1a5eb55cbb6e",
      "field": "Items",
      "type": "OutSystems.Model.Versioning.ValueNode`1[System.Collections.Generic.IList`1[
          ↪ MetaModelLibrary.IUIFlowNode]]",
      "valueType": "System.Collections.Generic.List`1[MetaModelLibrary.IUIFlowNode]",
      "value": [
        {
          "elementType": "MetaModelLibrary.Screen",
          "element": "99ed2ebd-42e5-bed3-ee9e-8d5a14d5116a"
        }
      ]
    },
    {
      "objectId": "5bac5cef-5f63-9bd2-be94-1a5eb55cbb6e",
      "field": "Generation",
      "type": "OutSystems.Model.Versioning.GenerationValueNode",
      "valueType": "System.Int32",
      "value": 2
    },
    {
      "objectId": "9ad0a9c2-7f73-7351-1391-81f48322e85d",
      "field": "Generation",
      "type": "OutSystems.Model.Versioning.GenerationValueNode",
      "valueType": "System.Int32",
      "value": 2
    },
    {
      "objectId": "9ad0a9c2-7f73-7351-1391-81f48322e85d",
```

```
144       "field": "Hash",
145       "type": "OutSystems.Model.Versioning.ValueNode`1[OutSystems.Model.Versioning.
              ↪ CalculatedFieldValue`1[System.Guid]]",
146       "calculatedFieldValue": {
147         "valueType": "System.Guid",
148         "isValid": false,
149         "value": null
150       }
151     },
152     {
153       "objectId": "dee6741a-dc75-b2f4-28e2-4d137bd5b675",
154       "field": "Hash",
155       "type": "OutSystems.Model.Versioning.ValueNode`1[OutSystems.Model.Versioning.
              ↪ CalculatedFieldValue`1[System.Guid]]",
156       "calculatedFieldValue": {
157         "valueType": "System.Guid",
158         "isValid": false,
159         "value": null
160       }
161     },
162     {
163       "objectId": "7b87f2a5-b4d0-4408-e74b-f17436903b8c",
164       "field": "Generation",
165       "type": "OutSystems.Model.Versioning.GenerationValueNode",
166       "valueType": "System.Int32",
167       "value": 2
168     }
169   ]
170 }
```

Considering this example, we will discuss how the serialization was implemented. We created the "Serialize" extension method for the Version class of the versioning solution, inside our own VersionSerializer class. We decided to do this to keep this code separate from the versioning solution's code, as that solution is not part of this thesis's work. However, the method could be directly implemented in the Version class. We also decided to serialize the versions to JSON files for simplicity and to facilitate inspecting the results for debugging purposes. The serialization includes some of the Version's properties: "SequentialID", "Id", "Name" and "IsUndoVersion", a boolean value.

The new objects collection is serialized into a JSON array, as shown in Listing 6.21. The "writer" variable is an object of the Utf8JsonWriter class.

Listing 6.21: NewObjects collection serialization

```
1 writer.WritePropertyName("newObjects");
2 writer.WriteStartArray();
3 foreach (var v in version.NewObjects)
4 {
5   writer.WriteStartObject();
6   writer.WriteString("id", v.Id);
```

```
7    writer.WriteString("type", v.GetType().ToString());
8    writer.WriteString("parentId", v.Parent?.Id.ToString());
9    writer.WriteString("parentType", v.Parent?.GetType().ToString());
10   writer.WritePropertyName("object");
11   v.SerializeFlat(writer);
12   writer.WriteEndObject();
13 }
14 writer.WriteEndArray();
```

Each element of the collection is a Versioned Object, and we serialize its ID, type, parent's ID, and parent's type. A Versioned Object's ID is a unique identifier for that object regarding the Transactions Manager where it was loaded or created. In our meta-model's elements, their Key property maps to their ID. Serializing this information is necessary for reasons that will become clear when discussing the deserialization. After that, the "SerializeFlat" method is called on the element. This is a method implemented by every Versioned Object. In the case of the elements of our meta-model, this method serializes the values of their properties. However, if a property is also a model element (the eSpace's home screen, for example), only its ID is serialized. This is because these properties can also be new objects, in which case their data is serialized in the "newObjects" array, and it would be redundant to serialize them here. Or they are objects that existed before the new version, in which case serializing them is unnecessary. The same is true for their collections of child elements; only their IDs are serialized. In this case, they are guaranteed to be new objects, too.

To illustrate this, we will use the serialized version 2, shown in Listing 6.20. In this version, a new Screen, "Screen1", with a Text Widget was created. The "newObjects" array has three elements: the new Screen (lines 7 to 21), the Screen's Widget collection (lines 22 to 36), and the new Text Widget (lines 37 to 49). Note that the value of the "widgets" key of the Screen object (in line 19) is the ID of the collection object, not the serialized collection.

We also had to implement the "SerializeFlat" method for the classes of the collections provided by the versioning solution (VersionedList, VersionedSet, and VersionedDictionary). The elements of these collections must also be serialized in a flat manner. However, these collections are generic, so they are not guaranteed to contain only Versioned Objects. So, we had to create a "SerializeFlat" extension method for the System.Object type (the root of the type hierarchy), which uses dynamic dispatching to call the appropriate flat serialize method for each type. We then had to create a specific flat serialize method for each type used by our meta-model: IVersionedObject, string, int, uint, bool, Guid, and CompoundKey. In the case of IVersionedObject, the "SerialiazeFlat" method already provided by this type is called. For the remaining types, their string representations are written. Note that, in Listing 6.20, the object corresponding to the Screen's Widget collection has an array of elements. Its only element, the Text Widget, is also represented only by its ID, not the serialized object (line 31).

We did not have the time to implement the serialization of the "DeletedObjects"

collection in our prototype, so this approach will not work with changes that delete elements from a model.

The "WrittenValues" collection is also serialized into a JSON array. The code of this serialization is shown in Listing 6.22.

Listing 6.22: WrittenValues collection serialization

```
writer.WritePropertyName("writtenValues");
writer.WriteStartArray();
foreach (var v in version.WrittenValues)
{
  writer.WriteStartObject();
  writer.WriteString("objectId", v.Key.Object.Id);
  writer.WriteString("field", v.Key.Field.Name);
  writer.WriteString("type", v.Value.GetType().ToString());
  var value = v.Value.Value;
  if (value is ICalculatedFieldValue calculatedFieldValue)
  {
    writer.WritePropertyName("calculatedFieldValue");
    calculatedFieldValue.WriteJsonRepresentation(writer);
  }
  else
  {
    writer.WriteString("valueType", value?.GetType().ToString());
    writer.WritePropertyName("value");
    value?.WriteJsonRepresentation(writer);
  }
  writer.WriteEndObject();
}
writer.WriteEndArray();
```

Each element of this collection is a key-value pair, where the key contains the modified field and the object it belongs to, and the value is of type IValueNode, an interface of the versioning solution, and contains the new value of that field. The object's ID, the field's name, and the IValueNode object's concrete type are serialized. The reason for serializing this information will be explained when discussing the deserialization implementation.

Then, the new value's type is serialized, as well as the value itself by calling the "WriteJsonRepresentation" method on that object. This is another extension method we implemented for the System.Object type that uses dynamic dispatching to call the appropriate method for each type. The "SerializeFlat" method that was previously mentioned cannot be used here. If a new value is an element of the model (if the eSpace's home screen is changed, for example), it is guaranteed to be an object that existed before the new version, or if it is new, it was already serialized in the "newObjects" array. So, the object should not be serialized here; it is only necessary to serialize its ID. We then had to implement a specific "WriteJsonRepresentation" method for each type used in fields of our meta-model: IVersionedObject, string, int, uint, bool, Guid, CompoundKey, generic lists, and generic dictionaries. In the case of IVersionedObject, the object's ID is written, as was

already mentioned. For the generic collections, the type of each element is serialized, and then the "WriteJsonRepresentation" method is called on the element. For CompoundKey, its "Serialize" method is called. For the remaining types, their string representations are written.

Furthermore, the modified field could be a Calculated Field. If this is the case, the type of the new value will be ICalculatedFieldValue, an interface of the versioning solution. So, we also had to implement the "WriteJsonRepresentation" method for this type, specifically for the CalculatedFieldValue generic class that implements ICalculatedFieldValue. This class contains the new value itself and a boolean, indicating whether it is valid. The value's type and the "IsValid" property are serialized, and if the value is valid, the "WriteJsonRepresentation" method is called on the value's object. Otherwise, the null value is written.

We will use Listing 6.19 to clarify this explanation. Remember that, in this version, the eSpace's name was modified. The first element of the "writtenValues" array (lines 9 to 15) represents this change. The "objectId" key corresponds to the eSpace's ID, and the "field" key corresponds to the name of the modified field. In this case, the modified field is the eSpace's name, and it is called "Name". Then comes the concrete type of the IValueNode object, which is the generic class ValueNode, whose type parameter is String. Then, the value's type, String, in this case. And finally, the new value itself, "Employees".

The third element of the array (lines 23 to 32) corresponds to the new hash of the eSpace. Since its name was modified, its hash changed as a consequence, and that information is represented here. As previously stated, the Compilation Hash of our meta-model's elements is a Calculated Field, which can be observed here. Notice that this element is serialized as a Calculated Field Value. The concrete type of the IValueNode object is the generic class ValueNode. Its type parameter is the generic class CalculatedFieldValue, whose type parameter is Guid. Then comes the value's type, Guid, followed by the "isValid" key, whose value is false in this case. And finally the value itself, which is null since the value is not valid. The value is invalid because it was never accessed and thus never calculated. If it had been calculated before, the "isValid" key would have the value true, and the value would be the newly calculated hash.

Considering Listing 6.20 now, it is worth mentioning the second element of its "writtenValues" array (lines 60 to 94). It corresponds to the eSpace's dictionary of Compound Keys. It is in the array because a Compilation Unit, the new Screen, was added to the model, so this collection had to be modified to include its Compound Key. The sixth element of the array (lines 117 to 127) corresponds to the UI Flow's collection of UI Flow Nodes. Again, this collection is in the array because it was modified. Before, there were no Screens, and it was empty. Now it contains the new Screen.

6.4.2 Deserialization

We implemented the "Deserialize" static method in the Version class, which returns a Version object corresponding to the deserialized version, as shown in Listing 6.23.

Listing 6.23: Version deserialization

```
internal static Version Deserialize(JsonNode node, TransactionsManager owner)
{
  return owner.DoDuringLoad(() =>
  {
    int sequentialId = node["sequentialId"]!.GetValue<int>();
    int id = node["id"]!.GetValue<int>();
    string name = node["name"]!.GetValue<string>();
    bool isUndoVersion = node["isUndoVersion"]!.GetValue<bool>();

    Version version = new(id, sequentialId, name, owner, isUndoVersion: isUndoVersion);

    JsonNode newObjectsNode = node["newObjects"]!;
    foreach (var newObjectNode in newObjectsNode.AsArray())
    {
      var newObject = ITransactionsManager.GetOrCreate<VersionedObject>(newObjectNode!,
          ↪ owner);
      var parent = ITransactionsManager.GetOrCreate<VersionedObject>(newObjectNode!["
          ↪ parentId"]!.GetValue<string>(), newObjectNode!["parentType"]!.GetValue<string
          ↪ >(), owner);
      newObject.DeserializeFlat(newObjectNode!["object"]!);
      newObject.Parent = parent;
      version.NewObjects.Add(newObject);
    }
.
.
.
```

The parameters of this method are a JsonNode object and a Transactions Manager. The JsonNode class represents a single node within a JSON document; in this case, it must be the JSON document's root node corresponding to the serialized version. The Transactions Manager must be the owner of the version. The method starts by retrieving the version's properties from the JSON node: sequential ID, ID, name, and the isUndoVersion boolean. Then, a new Version object is instantiated with these properties. Finally, the "NewObjects" and "WrittenValues" collections must be filled.

For the following explanation, it is important to note that the Transactions Manager has a dictionary with all its objects, which, when used in our prototype, corresponds to all the objects of a model. The dictionary's keys are the objects' IDs, and the values are the objects themselves.

To deserialize the "NewObjects" collection, the "newObjects" array of the JSON document is traversed. Each element is another JSON node object that contains the new object's ID and type, its parent's ID and type, and the remaining object's data that was serialized,

as previously explained. An instance of this new object must be obtained (line 15), so its properties can be retrived from the JSON and set, and the object can be added to the "NewObjects" collection. To achieve this, it is necessary to know the object's ID and type, as will be shown shortly. This is the reason why this information needs to be serialized. We use the "GetOrCreate" static method from the ITransactionsManager interface, which we created, as shown in Listing 6.24.

Listing 6.24: GetOrCreate method

```
1 static T GetOrCreate<T>(Guid id, string typeName, TransactionsManager transactionsManager)
      ↪ where T : IVersionedObject
2 {
3   if (!transactionsManager.Objects.TryGetValue(id, out var obj))
4   {
5     obj = VersionedObjectFactory.Create(id, typeName, transactionsManager);
6   }
7
8   return (T)obj;
9 }
```

This method has several overloads. Its purpose is to try to find the object in the Transactions Manager's dictionary that contains all objects using the object's ID. If it finds it, it means that this object has already been created, and it is returned. If not, it must be created. A new object is instantiated the first time it is encountered. This could be in the "newObjects" array, in which case its data is deserialized after its instantiation. Or it could be as another object's property or inside a collection. In that case, the object is only instantiated since only its ID and type are known, and deserialization of its data is not done until it is encountered in the "newObjects" array, where the data is stored.

However, this method cannot instantiate the object because it would require the ITransactionsManager interface to depend on the classes of objects it needs to instantiate, which is undesirable. We want to create a more general solution that decouples the creation logic from the specific types. To achieve this, we created the VersionedObjectFactory class that manages the creation of Versioned Objects based on their type names, as shown in Listing 6.25.

Listing 6.25: VersionedObjectFactory class

```
1 public static class VersionedObjectFactory
2 {
3   public delegate IVersionedObject CreateVersionedObjectDelegate(Guid id,
        ↪ TransactionsManager transactionsManager);
4   public delegate T CreateVersionedObjectDelegate<T>(Guid id, TransactionsManager
        ↪ transactionsManager) where T : IVersionedObject;
5   static readonly IDictionary<string, CreateVersionedObjectDelegate>
        ↪ versionedObjDelegateByTypeName = new Dictionary<string,
        ↪ CreateVersionedObjectDelegate>();
6
```

```
7   public static void Register<T>(CreateVersionedObjectDelegate<T> del) where T :
        ↪ IVersionedObject
8   {
9     versionedObjDelegateByTypeName.Add(typeof(T).ToString(), (id, transactionsManager) =>
          ↪ del(id, transactionsManager));
10  }
11
12  public static IVersionedObject Create(Guid id, string typeName, TransactionsManager
        ↪ transactionsManager)
13  {
14    if(versionedObjDelegateByTypeName.TryGetValue(typeName, out var del))
15    {
16      return del(id, transactionsManager);
17    }
18
19    throw new ArgumentException($"Unknown type name {typeName}.");
20  }
21 }
```

This class's "Create" method uses delegates to create objects of each type. In .NET, a delegate is a type that represents references to methods with a particular parameter list and return type. When a delegate is instantiated, the instance can be associated with any method with a compatible signature and return type [19].

This class declares two delegates. "CreateVersionedObjectDelegate" (line 3) defines the method signature for creating a Versioned Object. It takes a Guid (ID) and a Transactions Manager and returns an IVersionedObject. And a generic version of this delegate, "CreateVersionedObjectDelegate<T>", where T is constrained to IVersionedObject (line 4). A dictionary is used to store delegates, "versionedObjDelegateByTypeName" (line 5). It stores "CreateVersionedObjectDelegate" instances, keyed by a type name (string).

The "Register" method registers a delegate for creating a specific type of Versioned Object (T) in the dictionary. The type is registered using its name as the key. Notice that it takes as arguments delegates of the generic type to get the type's name, but delegates of the non-generic type are stored.

The "Create" method creates a Versioned Object using the type name and the object's ID. It looks up the corresponding delegate in "versionedObjDelegateByTypeName" and invokes it. If no delegate is found for the type name, an ArgumentException is thrown.

With this approach, delegates for each type specifying how objects of that type should be created must be registered before any deserialization of versions is performed. We decided to register these delegates in the static constructor of the ModelServices class that is part of our meta-model library. Therefore, this class must be accessed before executing any operations that rely on these delegates.

Going back to the code in Listing 6.23, after the new object's instance is obtained, a similar process is done for its parent (line 16) since it is necessary to set the new object's parent to its corresponding instance. This is why the ID and type of the object's parent

are serialized. Then, the "DeserializeFlat" method is called on the new object (line 17), passing the object's JSON node. This is a method that every Versioned Object implements. In the case of the elements of our meta-model, this method deserializes their properties from the JSON node and instantiates and sets their collections of child elements.

We also had to implement it for the classes of the versioned collections. Remember that, as explained previously, when flat serializing these collections, only the types of their elements are serialized, as well as their IDs, if the elements are Versioned Objects, or their string representations if not. So, the "DeserializeFlat" method for versioned collections must traverse the element array retrieved from the JSON node to deserialize each element and add it to the collection. If an element is a Versioned Object, its type and ID are retrieved from the JSON node and used to get an instance of the object by calling the "GetOrCreate" method. If not, it is only necessary to obtain its value from the JSON node.

Finally, the deserialized object is added to the "NewObjects" collection. Next, the "WrittenValues" collection must be deserialized. This code is shown in Listing 6.26.

Listing 6.26: Version deserialization (continuation)

```
19 .
20 .
21 .
22     JsonNode writtenValuesNode = node["writtenValues"]!;
23     foreach (var writtenValueNode in writtenValuesNode.AsArray())
24     {
25       string objectId = writtenValueNode!["objectId"]!.GetValue<string>();
26       if (owner.Objects.TryGetValue(new Guid(objectId), out var obj))
27       {
28         string fieldName = writtenValueNode!["field"]!.GetValue<string>();
29         IField? field = FindField(fieldName, obj.GetVersionedFields());
30         if (field == null)
31         {
32           throw new Exception("Error deserializaing written value. Object ID: " + objectId
                 ↪ + ". Field not found. Field name: " + fieldName);
33         }
34
35         IValueNode value = DeserializeValue(writtenValueNode, owner, version);
36
37         version.WrittenValues.Add((obj, field), value);
38
39         ((VersionedObject) obj).IsModified = true;
40       }
41       else
42       {
43         throw new Exception("Error deserializaing written value. Object ID: " + objectId +
               ↪ ". Object not found.");
44       }
45     }
46
47     return version;
```

```
48   });
49 }
```

The "writtenValues" array from the JSON document must be traversed. Remember that each element of this array is a JSON node containing the name of the changed field, the ID of the object it belongs to, and the serialized new value. Thus, the first step is to try to get an instance of the object from the Transactions Manager's dictionary that contains all objects using the object's ID. If the object is not found, an exception is thrown. Otherwise, the next step is to get an instance of the field. The "GetVersionedFields" method from the versioning solution returns the collection of versioned fields of an object. We can use it to get the instance of the field by traversing the collection until we find a field with the given name (this is what the "FindField" method does). If the field is not found, an exception is thrown. The next step is to deserialize the new value. The value can be of any type, and once again, we want to decouple the deserialization logic from specific types. Thus, we used an object factory, as we had done previously, to create the new value object, using its type name and serialized representation (done in the "DeserializeValue" method). In this case, the new values are not guaranteed to be Versioned Objects, so we are not using the delegates shown previously, but the approach is similar. Then, it is necessary to add the new element to the "WrittenValues" collection, as shown in line 37 (a key-value pair where the key is also a pair containing the object and the field, and the value is the new value). Finally, the object's "IsModified" property must be set to true.

Notice that the entire code of the "Deserialize" method shown in Listings 6.23 and 6.26 is wrapped in a lambda passed to the "DoDuringLoad" method of the Transactions Manager. This is required to ensure correct state management. In the deserialization of a version, new objects might be created, and objects' properties might be changed, which typically must be done inside a Transaction. However, we do not want to start a new Transaction because it would create new versions.

6.4.3 Transporting versions

After implementing the serialization and deserialization of versions, it is necessary to implement their transportation from the IDE to the Compiler. However, we did not have time to implement the transportation of versions. To test our serialization implementation, we used the file system to save and load the serialized versions. We created the "AddCommittedVersions" method in the TransactionsManager class that receives a collection of file names as a parameter, as shown in Listing 6.27.

Listing 6.27: AddCommittedVersions method (TransactionsManager class)

```
1 public IEnumerable<Version> AddCommittedVersions(IEnumerable<string> fileNames)
2 {
3   var result = new List<Version>();
4
5   foreach (var fileName in fileNames)
```

```
6   {
7     using var stream = File.OpenRead(fileName);
8     JsonNode? root = JsonNode.Parse(stream);
9     if (root == null)
10    {
11      throw new Exception("Error parsing JSON file.");
12    }
13
14    var version = Version.Deserialize(root, this);
15    result.Add(version);
16    stream.Close();
17
18    Commit(version);
19  }
20
21  return result;
22 }
```

The file names correspond to the files of the serialized versions. This method opens the files, calls the "Deserialize" method to deserialize the versions and commits them. This concludes the discussion of our prototype implementation.

7

Results

In this chapter, we present and discuss the results of our measurements of our solutions' impact on the publication process of an OutSystems application.

Figure 7.1 shows the sequence of publication steps between when the developer makes a change and when it can be seen (explained in more depth in Chapter 4). It starts when the last change is made, followed by the time the developer takes to click the 1CP button, which effectively triggers the publication process. After that, the Compare and Merge process takes place. Then, the model is saved, uploaded, and loaded by the Compiler. Finally, the compilation and deployment processes take place. It should be noted that the figure is not to scale. Therefore, the length of the lines does not accurately represent the actual duration of these processes.

The dashed line represents the processes affected by our solutions, which can contribute to reducing the total publication time.

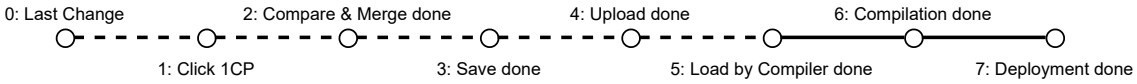

Figure 7.1: Publication steps

We will present measurements for the duration of saving and loading of models and provide estimates for uploading times. However, it is important to note that we do not include measurements for the time a developer takes to click the 1CP button, as this is entirely user-dependent and cannot be quantified objectively. Similarly, we have excluded measurements for the duration of the Compare and Merge process. As previously explained, in the best-case scenario, the Compare and Merge process identifies no conflicts and requires no further action. Conversely, if conflicts are present, developer intervention is necessary. Due to the variability in these scenarios, we do not have a method to measure these times objectively.

7.1 First approach

As previously explained, in our first approach, the IDE and the Compiler share the application model, so the serialization, transportation, and loading of the model by the Compiler are entirely removed from the publication process. We also eliminate the need for the developer to click the 1CP button to trigger the publication, since every change is performed directly on the shared model. This approach also allows for concurrent changes by different instances of the IDE without the need for the Compare and Merge process, since they would all perform the changes on a single instance of the model. Mechanisms to support this concurrency would need to be implemented, but this is out of the scope of this thesis. Figure 7.2 shows the publication steps, with the steps that our solution eliminates struck-out.

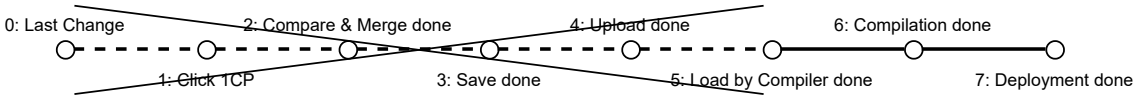

Figure 7.2: Publication steps with the first solution

We measured the saving and loading times for 100 real-world OutSystems 11 applications to draw conclusions about the impact of removing these operations from the Publication process. We also want to measure the impact of several variables on the duration of these operations: the size of the model, the number of screens of the model, and the number of changes made to the model before saving and loading. Although models might still contain other types of Compilation Units, we decided to focus only on screens since we also implemented them in our prototype's meta-model. This will facilitate making comparisons between the OutSystems 11 meta-model and our meta-model.

To make these measurements, we used the BenchmarkDotNet library [30], a .NET library designed for code benchmarking. We also used the OutSystems Model API to be able to programatically load, save, and modify the models. The benchmarks were run on a laptop machine with the following specifications:

- Operating System (OS): Windows 10
- Central Processing Unit (CPU): 12th Gen Intel Core i7-12800H, 1 CPU, 20 logical and 14 physical cores
- Random-access memory (RAM): 32 GB
- Runtime: .NET 8.0

7.1.1 No changes

We started by measuring save and load times without changing the model to establish a baseline. The complete results are shown in Table B.1 of Appendix B. We also created

charts to visualize the impact of the previously mentioned variables. Figure 7.3 shows the relationship between the save time and the number of screens in a model. This is a scatter chart with a trendline, where each point corresponds to a model. The y-axis represents time durations in milliseconds, and the x-axis represents the number of screens. Notice that many models have 0 screens. However, as previously mentioned, these models might still contain other Compilation Units and other elements, so their save times show significant variation. Focusing on the models with screens, it is possible to conclude that the number of screens does not strongly correlate with the save time when no changes are made. Nonetheless, models with 125 screens or more have relatively high save times.

Figure 7.4 shows the relationship between the save time and the model's size. The y-axis also represents time durations in milliseconds, and the x-axis represents sizes in megabytes. Contrary to the previous case, this chart shows a strong relationship between save times and size when no changes are made, which is intuitive.

Figures 7.5 and 7.6 are analogous to the previous ones but concern loading times. They show that loading times are only slightly related to the number of screens in a model and not related to its size. This is to be expected because of Lazy Loading. As discussed in Chapter 4, the entire model is not loaded at once in the OutSystems Platform. The collections of elements are loaded as they are needed. Thus, regardless of the model's size or the number of screens it contains, the loading process will always load only a small part of it.

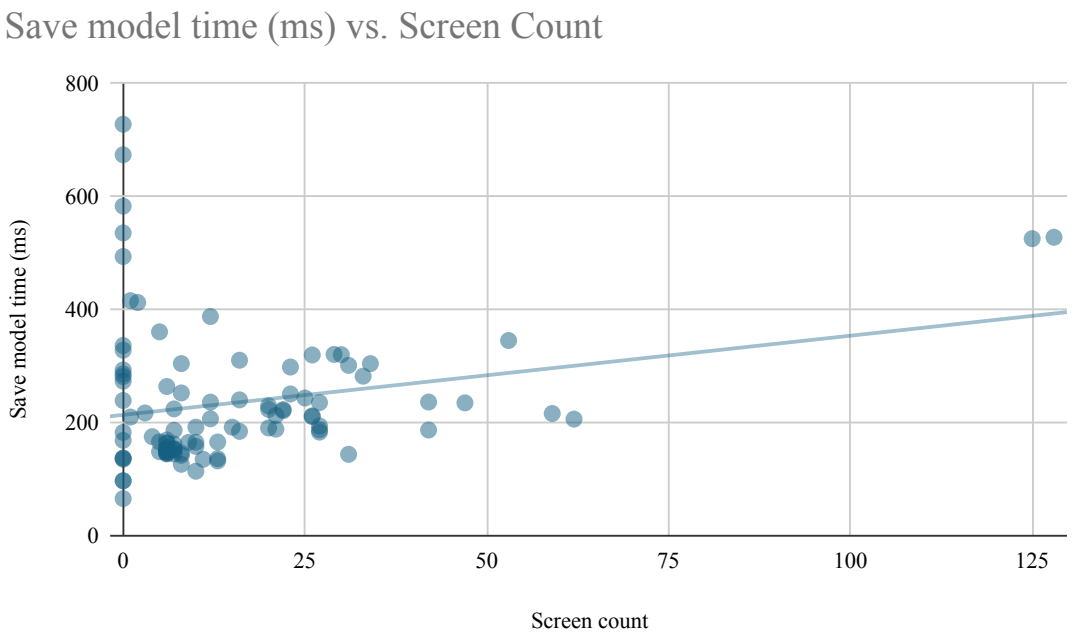

Figure 7.3: Save time vs. Screen count (no changes)

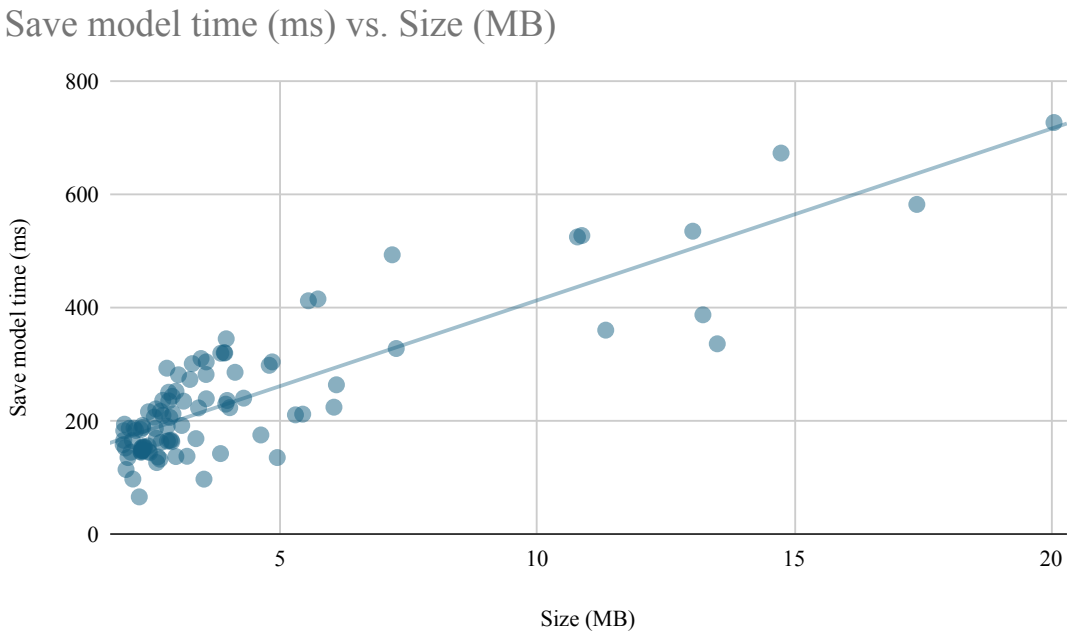

Figure 7.4: Save time vs. Size (no changes)

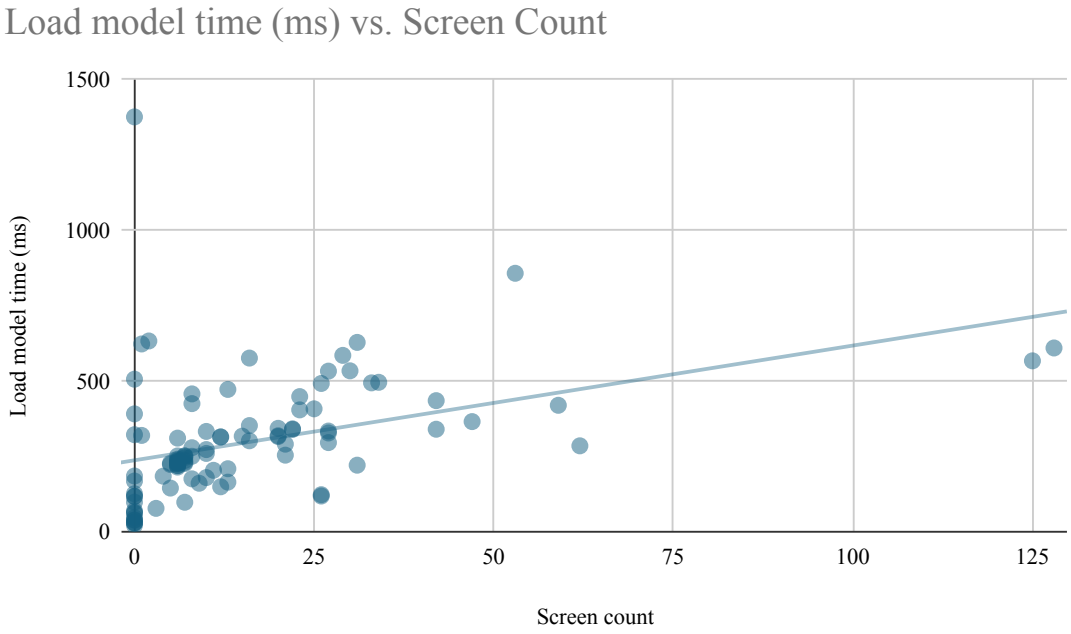

Figure 7.5: Load time vs. Screen count (no changes)

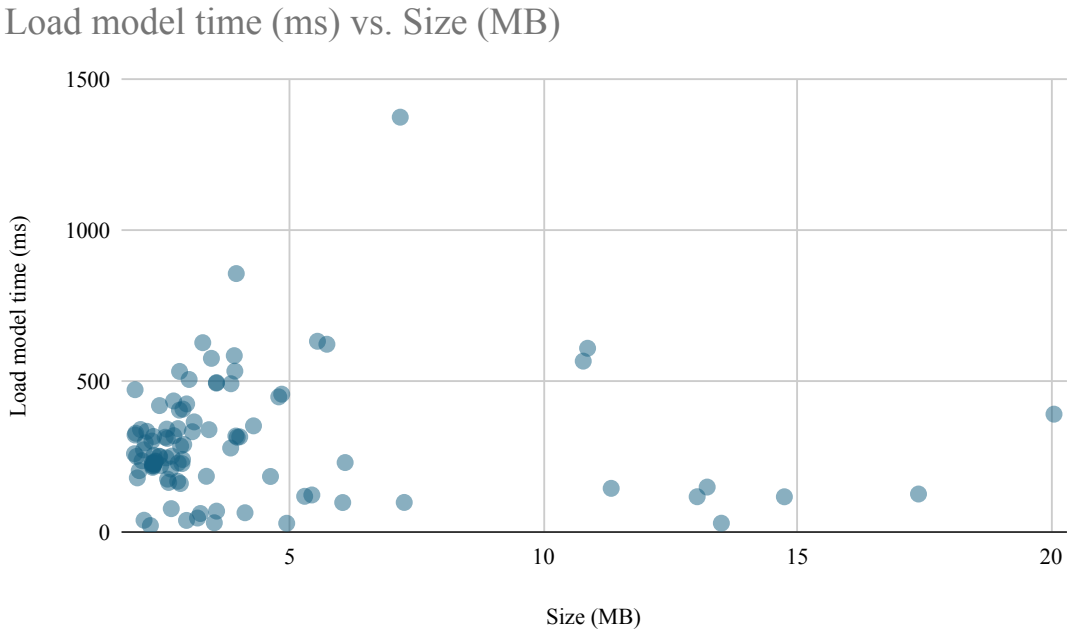

Figure 7.6: Load time vs. Size (no changes)

7.1.2 1 Change in 1 Screen

Afterward, we measured save times after making 1 change to 1 screen in each model (a widget of the screen is changed), and load times of the saved models. The results are shown in Table B.2 of Appendix B. This time, we only want to understand how the save and load times relate to the number of screens, so we excluded the models without screens. Figure 7.7 is the chart that shows the relationship between the save time and the model's number of screens in these conditions. Comparing it with Figure 7.3, it is possible to see that the times increased overall. When no changes were made, the lowest save time was around 65 ms and the highest under 800 ms, whereas now the lowest save time is around 161 ms and the highest almost 1300 ms. This is because after the model is modified, the save operation has to recalculate the fragments (mentioned in Chapter 4) to which the changed Compilation Units belong. This increased duration corresponds to the cost of the recalculation of the fragments.

It should be taken into account that since the models without screens are not being considered now, this will affect the trendline, creating a steeper slope. Even so, it is possible to see that the correlation between the two variables of the chart is stronger.

The relationship between the load time and the number of screens is shown in the chart of Figure 7.8. In these benchmarks, in addition to loading the model using the "LoadESpace" method from the OutSystems Model API, the widget collection of the changed screen is accessed. This way, we force the loading of the changed collection, simulating what would happen if the Compiler was loading the model, to recompile the changed Compilation Unit. When we compare this chart with the one in Figure 7.5, we can see a slight but not significant increase in the load times. The loading of the changed collection causes this increase. We can also observe that the correlation between the two variables is still present and not very strong. This is to be expected since only 1 screen was changed. In this case, models without screens were also not considered, which affected the trendline.

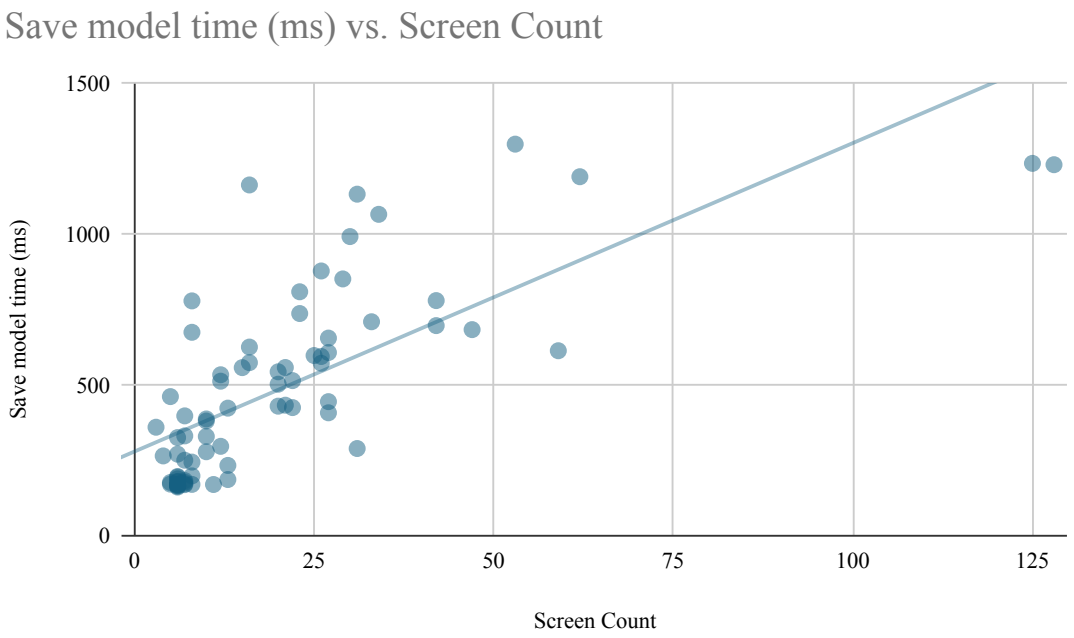

Figure 7.7: Save time vs. Screen count (1 change in 1 Screen)

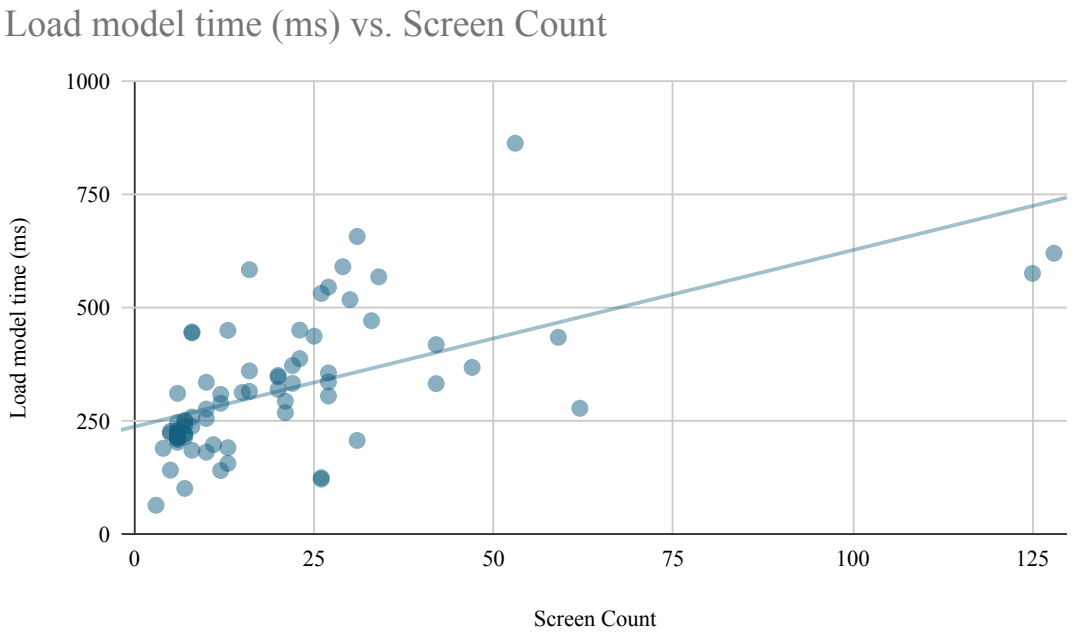

Figure 7.8: Load time vs. Screen count (1 change in 1 Screen)

7.1.3 Many changes in 1 Screen

Next, we measured save times after making many changes to 1 screen in each model (many widgets of the screen are changed), and load times of the saved models (and accessing the widget collection of the changed screen). The results are shown in Table B.3 of Appendix B. The chart in Figure 7.9 shows the relationship between the save time and the model's number of screens in these conditions. As expected, the results are identical to the ones in Figure 7.7. This is because the widgets' fragment of the modified screen must be recalculated in the save operation, whether the screen was subject to only one or many changes. Thus, the increased number of changes in the same Compilation Unit does not impact the duration of the save operation.

The chart in Figure 7.10 shows the relationship between the load time and the number of screens, and it also shows results identical to those in the chart in Figure 7.8. This is because a Compilation Unit must be recompiled regardless of whether one or many changes are made, and its collection of children (in this case, the screen's collection of widgets) has to be loaded in both cases, resulting in identical load times. Thus, the increased number of changes in the same Compilation Unit has no impact on the duration of the load operation.

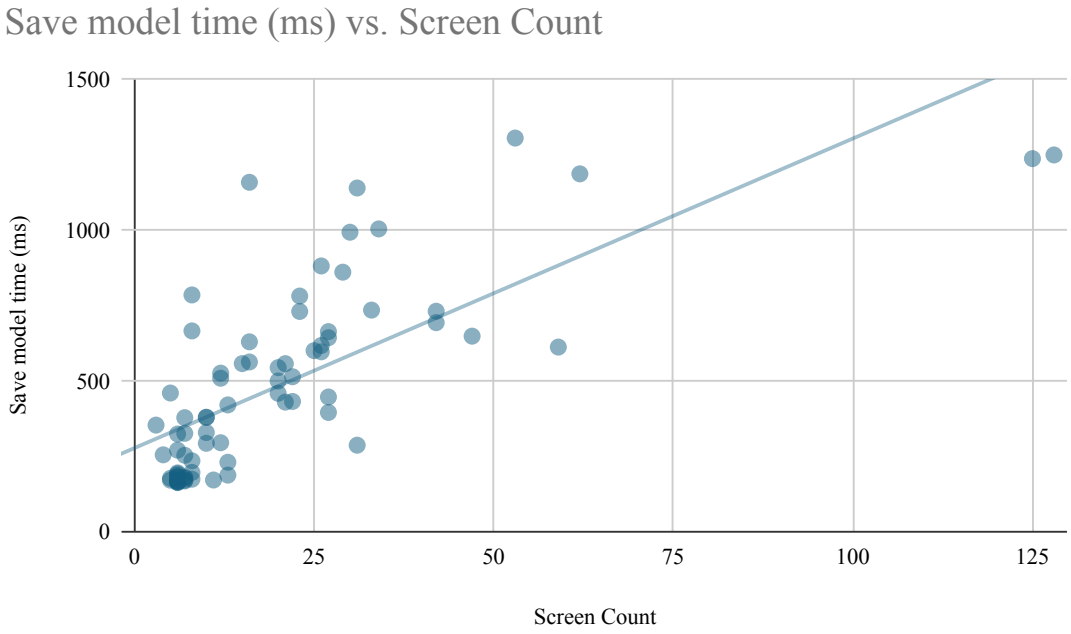

Figure 7.9: Save time vs. Screen count (Many changes in 1 Screen)

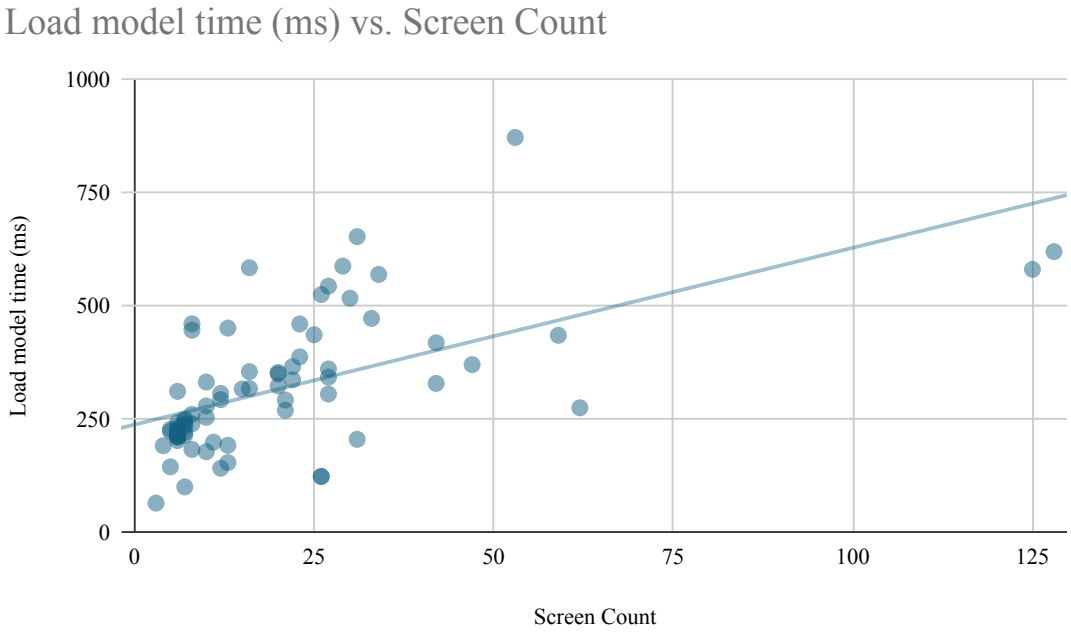

Figure 7.10: Load time vs. Screen count (Many changes in 1 Screen)

7.1.4 1 Change in all Screens

Then, we measured save times after making 1 change to every screen in each model (again, a widget of each screen is changed), and load times of the saved models (and accessing the widget collection of all changed screens). The results are shown in Table B.4 of Appendix B. Figure 7.11 corresponds to the chart showing the relationship between the save time and the number of screens in these conditions. If we compare it to the charts in Figures 7.9 and 7.7, it is possible to see that the results are very similar. There is only a slight increase in the higher times, but it is not significant. The same is true for the chart in Figure 7.12, which shows the relationship between the load time and the number of screens. This chart also shows results identical to those in Figures 7.10 and 7.8.

We expected a significant increase in save times since these changes affect all screens and require recalculating the fragments of their widget collections. We also expected a significant increase in load times since, in this case, the widget collections of all screens are being loaded. Nonetheless, the results do not corroborate this. One possible explanation is that the cost of saving and loading operations is dominated by compression and decompression, and Input/Output (I/O) operations. Thus, the effects of changing the screens of the model are not clearly visible in the results.

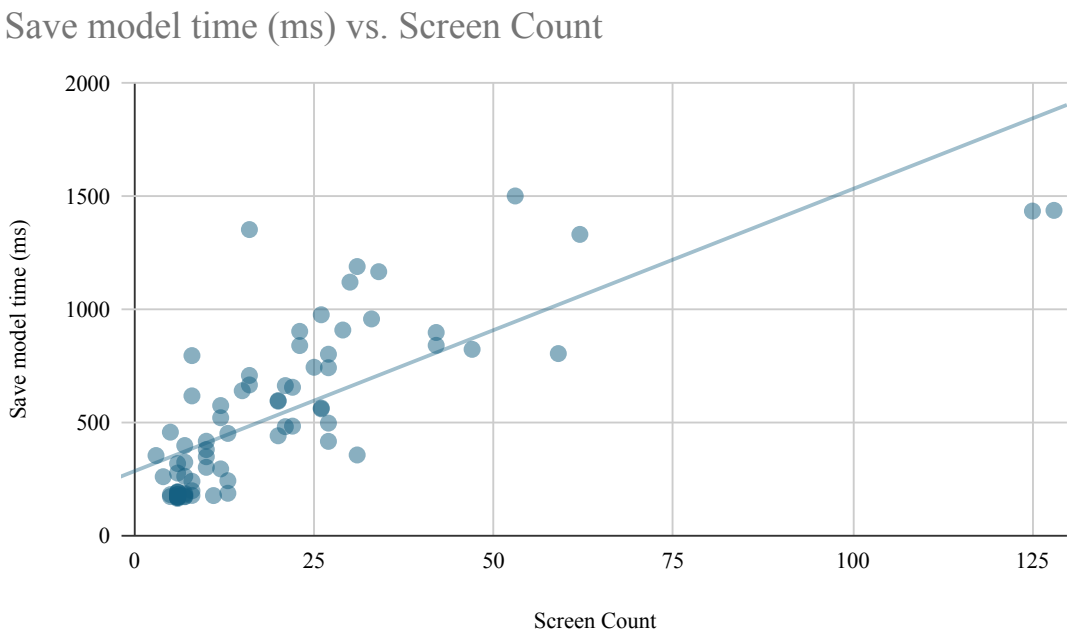

Figure 7.11: Save time vs. Screen count (1 change in all Screens)

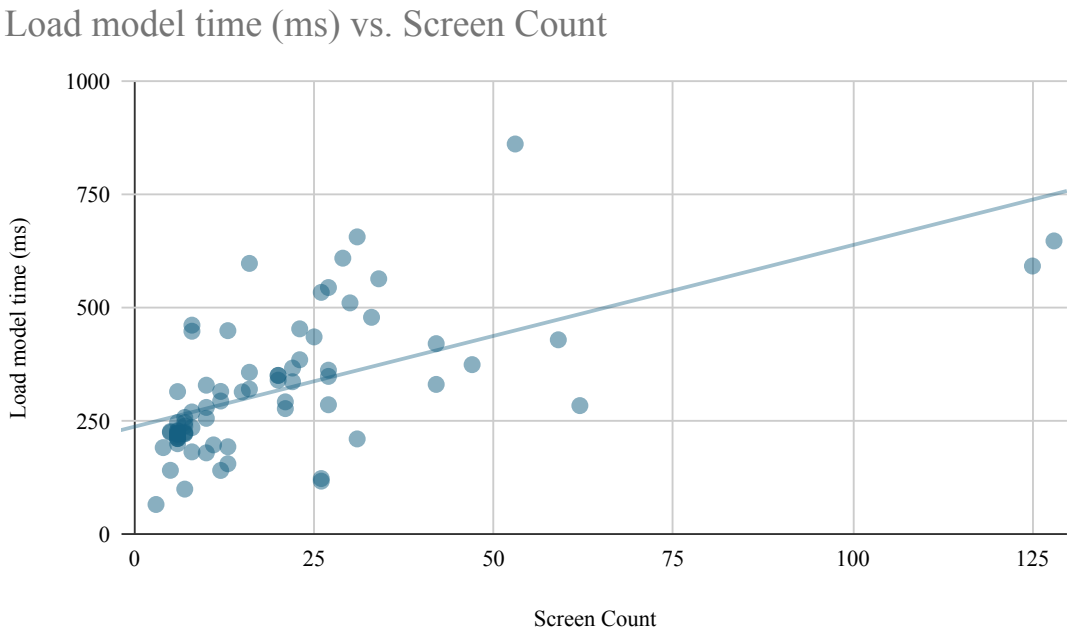

Figure 7.12: Load time vs. Screen count (1 change in all Screens)

7.1.5 Many changes in all Screens

Finally, we measured save times after making many changes to all screens in each model (many widgets of each screen are changed), as well as load times of the saved models (also accessing the widget collection of all changed screens). The results are shown in Table B.5 of Appendix B. Figure 7.13's chart shows the relationship between the save time and the number of screens in these conditions, and Figure 7.14's chart shows the relationship between the load time and the number of screens. As expected, these results are identical to those when only 1 widget is changed on each screen (Figures 7.11 and 7.12, respectively) for the same reasons previously explained. When saving the model, the fragments corresponding to the screens' collections of widgets must be recalculated whether one or many widgets were changed. And when loading the model, the widget collection of each screen is accessed, regardless of the number of widgets that were changed.

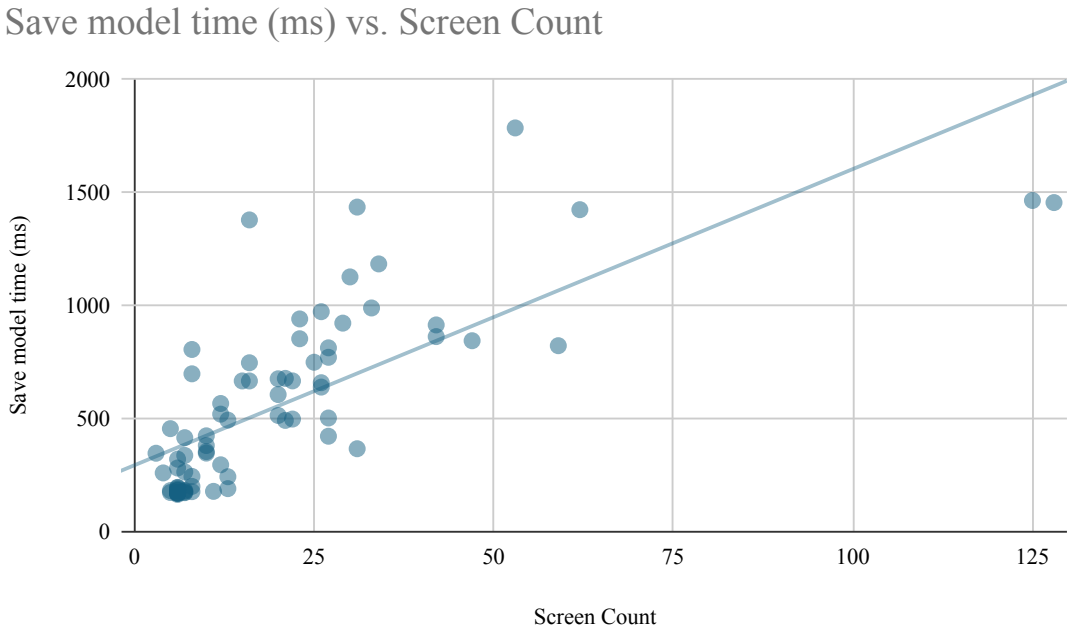

Figure 7.13: Save time vs. Screen count (Many changes in all Screens)

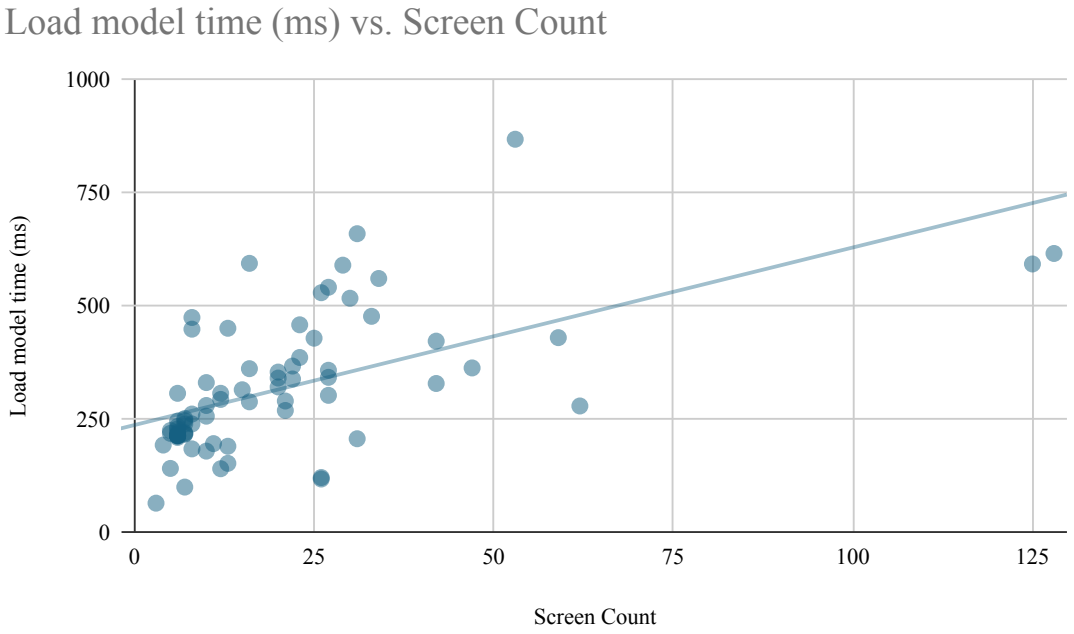

Figure 7.14: Load time vs. Screen count (Many changes in all Screens)

7.1.6 Uploading the model

The OutSystems Model API does not provide methods to upload models programmatically, so these measurements were conducted manually using a stopwatch. Measuring the upload duration of 100 models would not be feasible this way. So, we measured the upload duration of a relatively big model (around 20 MB) and a small model (around 2 MB). It should be noted that upload times are influenced by other highly variable factors, such as network speeds, and that these measurements have a high margin of error since they were performed manually. The purpose of these measurements is only to provide an estimate of the duration of this process. The upload duration ranged from 4 to 6 seconds for the big model and around 2 seconds for the small model.

7.1.7 Conclusions

In conclusion, our first approach shows the potential for reducing the Publication time considerably. The steps of clicking the 1CP button, the Compare and Merge process, saving, uploading, and loading the model are entirely removed from the Publication process. We cannot provide measurements for the duration of the first two steps, but it is not 0 in the current state of the OutSystems platform, so we can say that our solution results in an improvement concerning these steps. In the current state of the OutSystems platform, save times ranged from around 160 ms to around 1300 ms in our measurements for both 1 or many changes to 1 screen, which are the most common scenarios when a developer is working on an application. Load times ranged from around 60 ms to 870 ms in the same conditions. So our solution also provides a significant improvement regarding these steps. Finally, upload times ranged from 2 to 6 seconds in the current state of the OutSystems platform, so our approach also provides an improvement regarding this step.

7.2 Second approach

In our second approach, we propose saving, transporting, and loading only a delta instead of the entire model. To achieve this, we also propose to keep the Compiler running for some time. This way, the model is kept in memory, and only the delta must be loaded. This delta corresponds to the versions created by a versioning solution that is also being developed within OutSystems, after a model is changed. Another aspect of this solution is the possibility of having the IDE save and upload the versions in the background without requiring the developer to click the 1CP button. A Compare and Merge operation is still necessary with this approach since concurrent changes to the model are possible. However, we did not explore the challenges associated with this operation in our work. The Compare and Merge would occur automatically when a Publication is triggered and whenever another developer published other verions in the meantime. In this case, the IDE ceases to know the version in which the Compiler's model is. The Compare and Merge should result in having a local version with all the changes that exist in the Compiler side,

as well as the changes made locally that still have not been propagated to the Compiler. However it is not clear if sending this resulting delta is enough, and further exploration of this issue is needed. Figure 7.15 shows the publication steps with this solution.

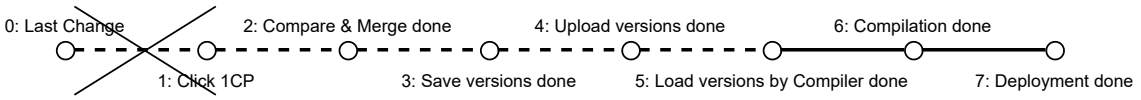

Figure 7.15: Publication steps with the second solution

It should be noted that the OutSystems Model API also provides methods for saving and loading deltas of a model. However, in this case, the deltas correspond to the fragments modified by the changes. We created two example models, using each of the meta-models (the OutSystems meta-model and our meta-model). One of the example models has 1 eSpace, with 1 UI Flow, 1 Screen, and 1 Text Widget. We will refer to it as the small model. The other has one eSpace, with 1 UI Flow and 10 Screens, each with 200 Text Widgets, 200 Image Widgets, and 200 Link Widgets. We will refer to it as the big model. We make changes to the models and then compare the duration of saving and loading a model with saving and loading the delta (in the case of the OutSystems meta-model), and saving and loading the model with saving and loading versions (in the case of our meta-model). This way, we can draw conclusions about the impact of this solution on the duration of the saving and loading operations , as well as understand if it has any benefits when compared to the saving and loading of deltas already implemented.

7.2.1 No changes

Like with the previous approach, we start by measuring the save and load times of the models, without making any changes, to establish a baseline. The results are shown in Table 7.1 for the small model, and in Table 7.2 for the big model. Our meta-model, being a simplification of the OutSystems meta-model, shows significantly lower times for the saving and loading operations.

Table 7.1: Small Model | No Changes

	O11 model (Small)	Our model (Small)
Save model time (ms)	6.689	0.26673
Load model time (ms)	20.703	0.09715

Table 7.2: Big Model | No Changes

	O11 model (Big)	Our model (Big)
Save model time (ms)	98.61	14.21

Load model time (ms)	78.61	22.98

7.2.2 1 Change in 1 Screen

Table 7.3 shows the duration of the saving operation for the small model in both meta-models after making 1 change to 1 screen. It also shows the duration of the save delta operation for the OutSystems meta-model and the save versions operation (in this case, only one new version is created) for our meta-model. The "Relative percentage" row corresponds to the ratio between the save delta time and save model time, in the case of the OutSystems meta-model, and the ratio between the save versions time and save model time for our meta-model, as a percentage. We can conclude that both saving the delta and saving the new version cost less time than saving the entire model. However, in this case, saving the delta costs roughly a quarter of the save model time, whereas saving the version costs almost 90% of the save model time.

Table 7.4 is analogous to the previous one but shows loading times. In this case, both loading the delta and loading the new version cost more than loading the model. However, loading the delta is less costly again, showing only a slight increase compared to the load model time, whereas loading the version costs almost three times the load model time.

Tables 7.5 and 7.6 are analogous to Tables 7.3 and 7.4, respectively, but show the results for the big model. First, considering Table 7.5, it is possible to see that saving the delta and saving the new version are again less costly than saving the entire model. However, now our solution shows a highly significant improvement, costing only less than 2% of the save model time, whereas saving the delta costs more or less 43% of the save model time. The same is true for loading times, shown in Table 7.6. In this case, loading the delta costs slightly more than loading the model, whereas loading the new versions costs less than 2% of the load model time.

Considering these results, it would be possible to conclude that, for small models and minor changes, our solution does not show the potential to provide any benefits. However, as previously explained, it should be noted that we did not implement the serialization and deserialization of versions with performance in mind. These operations have the potential to be optimized, which could result in a different outcome. Furthermore, making one change on only one screen in a relatively big model is probably one of the most common scenarios for an OutSystems developer. Under these conditions, our solution has the potential to reduce the cost of the saving and loading operations significantly.

Table 7.3: Saving | Small Model | 1 Change in 1 Screen

	O11 model (Small)	Our model (Small)
Save model time (ms)	9.846	0.26053
Save version time (ms)	-	0.23385
Save delta time (ms)	2.698	-
Relative percentage	27.40%	89.76%

Table 7.4: Loading | Small Model | 1 Change in 1 Screen

	O11 model (Small)	Our model (Small)
Load model time (ms)	20.57	0.0981
Load version time (ms)	-	0.29116
Load delta time (ms)	24.51	-
Relative percentage	119.15%	296.80%

Table 7.5: Saving | Big Model | 1 Change in 1 Screen

	O11 model (Big)	Our model (Big)
Save model time (ms)	135.44	14.1571
Save version time (ms)	-	0.2456
Save delta time (ms)	59	-
Relative percentage	43.56%	1.73%

Table 7.6: Loading | Big Model | 1 Change in 1 Screen

	O11 model (Big)	Our model (Big)
Load model time (ms)	78.37	22.8815
Load version time (ms)	-	0.4137
Load delta time (ms)	83.09	-
Relative percentage	106.02%	1.81%

7.2.3 Many Changes in 1 Screen

Tables 7.7, 7.8, 7.9, and 7.10 are analogous to the previous ones, but they show the measurements after making many changes to 1 screen. In this case, there are two options regarding the saving and loading of versions. Each change can be done in a different transaction. This creates a version for each change. The other option is that all changes

are done inside a single transaction, creating only one new version. We will refer to this method as making changes in bulk. We provide measurements for both options.

Looking at the results, it is clear that making each change in a different transaction is very costly. This is because serializing and deserializing many versions introduces a significant overhead. We will discard this option and focus only on the changes performed in bulk.

The results are similar to the ones in the previous subsection. For the small model, saving the new version costs less than saving the entire model, but saving the delta shows a higher improvement. In the case of the loading operation, neither loading the delta nor loading the version have an advantage over loading the entire model, and loading the version shows a higher relative cost.

On the other hand, for the big model, our solution shows the potential to significantly reduce the costs of saving and loading operations. Saving the new version costs only 9% of the save model time, whereas saving the delta costs more than 60% of the save model time. Loading the version costs 23% of the load model time, whereas loading the delta is more costly than loading the entire model.

Table 7.7: Saving | Small Model | Many Changes in 1 Screen

	O11 model (Small)	Our model (Small) - Many Transactions	Our model (Small) - Bulk
Save model time (ms)	9.029	0.26224	0.26224
Save versions time (ms)	-	143.95173	0.1709
Save delta time (ms)	2.523	-	-
Relative percentage	27.94%	54893.12%	65.17%

Table 7.8: Loading | Small Model | Many Changes in 1 Screen

	O11 model (Small)	Our model (Small) - Many Transactions	Our model (Small) - Bulk
Load model time (ms)	20.53	0.09872	0.09872
Load versions time (ms)	-	63.0048	0.2576
Load delta time (ms)	24.27	-	-
Relative percentage	118.22%	63821.72%	260.94%

Table 7.9: Saving | Big Model | Many Changes in 1 Screen

	O11 model (Big)	Our model (Big) - Many Transactions	Our model (Big) - Bulk
Save model time (ms)	151.88	14.69	14.6
Save versions time (ms)	-	141.65	1.317
Save delta time (ms)	95.2	-	-
Relative percentage	62.68%	964.26%	8.97%

Table 7.10: Loading | Big Model | Many Changes in 1 Screen

	O11 model (Big)	Our model (Big) - Many Transactions	Our model (Big) - Bulk

Load model time (ms)	79.73	23.85	23.85
Load versions time (ms)	-	64.29	5.508
Load delta time (ms)	86.06	-	-
Relative percentage	107.94%	269.56%	23.09%

7.2.4 1 Change in all Screens

Tables 7.11 and 7.12 show the measurements after making 1 change to all screens. In this case, we are not showing the results for the small model, as it only contains 1 screen, so the results would be the same as when doing 1 change to 1 screen.

Like previously, changes can be made in different transactions or in a single transaction in our meta-model. However, since the big model contains 10 screens and 1 change is made to each screen, if the changes are made in different transactions, only 10 new versions will be created, so the overhead is not as significant as before.

The results show that our solution provides significant improvements since saving the version that contains the changes in bulk costs only 2% of the save model time, compared to almost 80% in the case of saving the delta. Loading the version only costs around 3% of the load model time, whereas loading the delta costs more than loading the entire model.

Table 7.11: Saving | Big Model | 1 Change in all Screens

	O11 model (Big)	Our model (Big) - Many Transactions	Our model (Big) - Bulk
Save model time (ms)	198.1	14.045	14.045
Save versions time (ms)	-	2.321	0.2693
Save delta time (ms)	155.6	-	-
Relative percentage	78.55%	16.53%	1.92%

Table 7.12: Loading | Big Model | 1 Change in all Screens

	O11 model (Big)	Our model (Big) - Many Transactions	Our model (Big) - Bulk
Load model time (ms)	79.09	22.89	22.89
Load versions time (ms)	-	2.145	0.7433
Load delta time (ms)	91.82	-	-
Relative percentage	116.10%	9.37%	3.25%

7.2.5 Many Changes in all Screens

Finally, Tables 7.13 and 7.14 show the measurements after making many changes to all screens. We are also only providing measurements for the big model for the reasons explained previously.

If the changes are not made in bulk, there is significant overhead, and the costs of saving and loading the versions are very high. Therefore, we will focus on the changes made in bulk.

The results show that saving the new version costs around 67% of the save model time, whereas saving the delta costs around 90% of the save model time. However, loading the version costs significantly more than loading the entire model, whereas loading the delta has a similar cost to loading the entire model in these conditions. We do not know why this is the case, but we hypothesize that it is related to the poor optimization of the version deserialization operation.

Table 7.13: Saving | Big Model | Many Changes in all Screens

	O11 model (Big)	Our model (Big) - Many Transactions	Our model (Big) - Bulk
Save model time (ms)	400.2	20.07	20.07
Save versions time (ms)	-	1470.35	13.39
Save delta time (ms)	363.3	-	-
Relative percentage	90.78%	7326.11%	66.72%

Table 7.14: Loading | Big Model | Many Changes in all Screens

	O11 model (Big)	Our model (Big) - Many Transactions	Our model (Big) - Bulk
Load model time (ms)	78.79	31.64	31.64
Load versions time (ms)	-	678	78.18
Load delta time (ms)	80.21	-	-
Relative percentage	101.80%	2142.86%	247.09%

7.2.6 Conclusions

In conclusion, our proposed approach of saving and loading only the new versions of the model showed the potential to reduce the duration of the saving and loading operations in some cases but not all. Nonetheless, the serialization and deserialization of versions can be optimized, which would certainly yield better results. Furthermore, in the most common scenarios for an OutSystems developer, who typically performs one or many changes in one screen at a time in a model with several screens, each containing several widgets, our solution proved to be advantageous.

8

Conclusions

This work is motivated by the current workflow of an OutSystems developer, which has a significant time gap between making a change to an application and being able to see and test the results. This creates an experience that is not fluid and can be cumbersome. This is relevant because the OutSystems Platform aims to be an easy-to-use tool for creating software quickly. Furthermore, a good user experience for OutSystems developers is essential to ensure that they enjoy using the tool and that it boosts their productivity.

To mitigate this issue, we presented two approaches in this thesis, the purpose of which is to reduce the Publication times of OutSystems applications. In the first approach, by having the IDE and the Compiler share the application model, we can eliminate steps from the Publication process, most notably the saving, uploading, and loading of the model. The second approach does not remove these operations from the Publication process but aims to reduce their duration. To achieve this, it saves, transports, and loads only the modified parts of the model, a delta, instead of the entire model. To calculate this delta, we used a versioning solution that is also being developed within OutSystems. With this solution, changes to the model are made in the context of Transactions, and each Transaction creates a new version of the application. These versions contain the newly created objects, the deleted objects, and the values of the modified fields. The delta corresponds to the versions created since the last Publication.

We developed a prototype using an abstraction of the OutSystems meta-model to serve as a proof of concept. We explored whether these solutions were feasible, the challenges that they pose, and whether they can, in fact, contribute to reducing Publication time. We were successful in implementing the first approach in our prototype. And, although we did not fully implement the second approach, we were able to implement the serialization and deserialization of versions, which is needed for this approach and is not already implemented by the versioning solution. It is also worth noting that one of the biggest challenges we faced with the first approach was a concurrency challenge. We had to guarantee that the Compiler could compile many applications concurrently while ensuring that the same application is never compiled by different instances of the Compiler simultaneously. To solve this problem we developed an algorithm that schedules

the compilation of applications, launching new Compiler instances as needed, and that guarantees these conditions.

The combined duration of saving, uploading, and loading a model can take up to a few seconds, and since the first approach removes them from the Publication process, we can state that, in fact, it contributes to reducing its duration.

The results obtained for the second approach show that it has the potential to significantly reduce the costs of the saving and loading operations, in some cases. When making one change in one screen on a typical OutSystems model, saving and loading a version was shown to cost less than 2% of the cost of saving and loading the entire model. And when making many changes in one screen on a typical OutSystems model, saving versions was shown to cost around 9% of the cost of saving the entire model, and loading versions was shown to cost around 23% of the cost of loading the entire model. In other cases, this solution did not show an improvement, but these scenarios are less common. Furthermore, the performance of the serialization and deserialization of versions has room to be improved, which could yield better results.

8.1 Future Work

The work done in this dissertation can be continued in several ways, which we will present subsequently. Firstly, the second approach can be further developed. As already mentioned, the serialization and deserialization of versions were not implemented with performance concerns in mind, so they can be optimized. Further exploration is needed to understand how long the Compiler should be kept running, balancing a fluid developer experience with reasonable resource usage. Exploring the challenges associated with supporting concurrent model modifications with the second approach is also relevant. Furthermore, the results we obtained demonstrated the potential of our solutions to reduce the duration of the Publication process, which is an argument in favor of implementing them in the OutSystems Platform and meta-model. Finally, optimizing other steps of the Publication process, such as the compilation, are yet to be explored.

Bibliography

[1] J.J. Birchman and S.L. Tanimoto. "An implementation of the VIVA visual language on the NeXT computer". In: *Proceedings IEEE Workshop on Visual Languages*. 1992, pp. 177–183. DOI: 10.1109/WVL.1992.275767 (cit. on p. 6).

[2] Sebastian Burckhardt et al. "It's alive! continuous feedback in UI programming". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 95–104. DOI: 10.1145/2491956.2462170. URL: https://doi.org/10.1145/2491956.2462170 (cit. on p. 4).

[3] M.M. Burnett, J.W. Atwood, and Z.T. Welch. "Implementing level 4 liveness in declarative visual programming languages". In: *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No.98TB100254)*. 1998, pp. 126–133. DOI: 10.1109/VL.1998.706155 (cit. on p. 6).

[4] Miguel Domingues and João Costa Seco. "Type Safe Evolution of Live Systems". In: *Workshop on Reactive and Event-based Languages & Systems (REBLS'15)*. Pittsburgh, 2015. URL: https://docentes.fct.unl.pt/jrcs/files/rebls15.pdf (cit. on p. 6).

[5] James R. Driscoll et al. "Making Data Structures Persistent". In: *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*. Ed. by Juris Hartmanis. ACM, 1986, pp. 109–121. DOI: 10.1145/12130.12142. URL: https://doi.org/10.1145/12130.12142 (cit. on p. 17).

[6] Hugo Lourenço et al. "LUV is not the answer: continuous delivery of a model driven development platform". In: *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*. Ed. by Esther Guerra and Ludovico Iovino. ACM, 2020, 52:1–52:10. DOI: 10.1145/3417990.3419502. URL: https://doi.org/10.1145/3417990.3419502 (cit. on p. 1).

[7] João M. Lourenço. *The NOVAthesis Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[8] Richard G. McDaniel and Brad A. Myers. "Building Applications Using Only Demonstration". In: *Proceedings of the 3rd International Conference on Intelligent User Interfaces, IUI 1998, San Francisco, CA, USA, January 6-9, 1998*. Ed. by Loren G. Terveen, Peter Johnson, and Joe Mark. ACM, 1998, pp. 109–116. DOI: 10.1145/268389.268409. URL: https://doi.org/10.1145/268389.268409 (cit. on p. 6).

[9] Miguel Carvalho Pires. "Incremental compilation and deployment for OutSystems Platform". MA thesis. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2014-02. URL: http://hdl.handle.net/10362/15108 (cit. on pp. 7, 20, 22, 24, 32).

[10] Ivan E. Sutherland. "Sketchpad: a man-machine graphical communication system". In: *Proceedings of the 1963 spring joint computer conference, AFIPS 1963 (Spring), Detroit, Michigan, USA, May 21-23, 1963*. Ed. by E. Calvin Johnson. ACM, 1963, pp. 329–346. DOI: 10.1145/1461551.1461591. URL: https://doi.org/10.1145/1461551.1461591 (cit. on p. 6).

[11] Steven L. Tanimoto. "A perspective on the evolution of live programming". In: *Proceedings of the 1st International Workshop on Live Programming*. LIVE '13. San Francisco, California: IEEE Press, 2013, pp. 31–34. ISBN: 9781467362658 (cit. on p. 5).

[12] Steven L. Tanimoto. "Programming in a data factory". In: *2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003), 28-31 October 2003, Auckland, New Zealand*. IEEE Computer Society, 2003, pp. 100–107. DOI: 10.1109/HCC.2003.1260209. URL: https://doi.org/10.1109/HCC.2003.1260209 (cit. on p. 6).

[13] Steven L. Tanimoto. "VIVA: A visual language for image processing". In: *J. Vis. Lang. Comput.* 1.2 (1990-06), pp. 127–139. ISSN: 1045-926X. DOI: 10.1016/S1045-926X(05)80012-6. URL: https://doi.org/10.1016/S1045-926X(05)80012-6 (cit. on pp. 5, 6).

Webography

[14] Dan Abramov. *gaearon/react-hot-loader: Tweak React components in real time. (Deprecated: use Fast Refresh instead.)* URL: https://github.com/gaearon/react-hot-loader (visited on 2024-01-05) (cit. on p. 13).

[15] Dan Abramov. *Getting Started · React Hot Loader*. URL: https://gaearon.github.io/react-hot-loader/getstarted/ (visited on 2024-01-05) (cit. on p. 12).

[16] Graphviz Authors. *Graphviz*. 2021. URL: https://graphviz.org/ (visited on 2024-09-26) (cit. on p. 59).

[17] *Babel · Babel*. URL: https://babeljs.io/ (visited on 2024-01-10) (cit. on p. 12).

[18] React Collaborators. *react-refresh - npm*. URL: https://www.npmjs.com/package/react-refresh (visited on 2024-01-15) (cit. on p. 13).

[19] Dotnet Contributors. *Delegates - C# | Microsoft Learn*. 2022-09. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/ (visited on 2024-09-11) (cit. on p. 70).

[20] Dotnet Contributors. *Handling and Raising Events - .NET | Microsoft Learn*. 2022-04. URL: https://learn.microsoft.com/en-us/dotnet/standard/events/ (visited on 2024-08-28) (cit. on p. 51).

[21] Dotnet Contributors. *roslyn/docs/wiki/EnC-Supported-Edits.md at main · dotnet/roslyn*. 2023. URL: https://github.com/dotnet/roslyn/blob/main/docs/wiki/EnC-Supported-Edits.md (visited on 2024-01-15) (cit. on p. 16).

[22] Dotnet Contributors. *System.Threading.Monitor class - .NET | Microsoft Learn*. 2024-09. URL: https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-threading-monitor (visited on 2024-09-04) (cit. on p. 54).

[23] Microsoft Documentation Contributors. *.NET Hot Reload support for ASP.NET Core | Microsoft Learn*. 2023-04. URL: https://learn.microsoft.com/en-us/aspnet/core/test/hot-reload?view=aspnetcore-7.0 (visited on 2024-01-15) (cit. on p. 16).

[24] Microsoft Documentation Contributors. *Supported Code Changes (C# and Visual Basic) - Visual Studio (Windows) | Microsoft Learn*. 2023-12. URL: https://learn.microsoft.com/en-us/visualstudio/debugger/supported-code-changes-csharp?view=vs-2022 (visited on 2024-01-08) (cit. on p. 16).

[25] Webpack Documentation Contributors. *Hot Module Replacement | webpack*. URL: https://webpack.js.org/concepts/hot-module-replacement/ (visited on 2024-01-11) (cit. on p. 13).

[26] Wikipedia Contributors. *Vite (software) - Wikipedia*. 2024-08. URL: https://en.wikipedia.org/wiki/Vite_(software) (visited on 2024-08-17) (cit. on p. 43).

[27] Wikipedia Contributors. *Webpack - Wikipedia*. 2023-04. URL: https://en.wikipedia.org/wiki/Webpack (visited on 2024-01-10) (cit. on p. 12).

[28] Wikipedia contributors. *ASP.NET - Wikipedia*. 2023-11. URL: https://en.wikipedia.org/wiki/ASP.NET (visited on 2024-01-08) (cit. on p. 16).

[29] Wikipedia contributors. *React (software) - Wikipedia*. 2024-01. URL: https://en.wikipedia.org/wiki/React_(software) (visited on 2024-01-05) (cit. on p. 12).

[30] .NET Foundation and contributors. *Home | BenchmarkDotNet*. 2024. URL: https://benchmarkdotnet.org/ (visited on 2024-09-21) (cit. on p. 75).

[31] Hugo Lourenço, Carla Ferreira, and João Costa Seco. *PLF2024 - Google Slides*. 2024-09. URL: https://docs.google.com/presentation/d/1M3LcC2WfnW_8-xpYQj2etU5glayDwHFHDpPw1TXXJmc (visited on 2024-09-26) (cit. on p. 59).

[32] Dmitry Lyalin. *Introducing the .NET Hot Reload experience for editing code at runtime - .NET Blog*. 2021-05. URL: https://devblogs.microsoft.com/dotnet/introducing-net-hot-reload/ (visited on 2024-01-08) (cit. on p. 16).

[33] OutSystems. *Aggregate - OutSystems 11 Documentation*. 2024-01. URL: https://success.outsystems.com/documentation/11/reference/outsystems_language/data/handling_data/queries/aggregate/ (visited on 2024-01-31) (cit. on p. 8).

[34] OutSystems. *Client Variable - OutSystems 11 Documentation*. 2024-01. URL: https://success.outsystems.com/documentation/11/reference/outsystems_language/data/handling_data/client_variable/ (visited on 2024-01-31) (cit. on p. 8).

[35] OutSystems. *Defining a Low-Code Platform | OutSystems*. URL: https://www.outsystems.com/guide/low-code/defining-low-code-platform/ (visited on 2024-01-19) (cit. on p. 19).

[36] OutSystems. *Designing Screens - OutSystems 11 Documentation*. 2023-12. URL: https://success.outsystems.com/documentation/11/reference/outsystems_language/interfaces/designing_screens/ (visited on 2024-01-31) (cit. on p. 8).

[37] OutSystems. *Entities - OutSystems 11 Documentation*. 2023-12. URL: https://success.outsystems.com/documentation/11/developing_an_application/use_data/data_modeling/entities/ (visited on 2024-01-31) (cit. on p. 7).

[38] OutSystems. *Entity - OutSystems 11 Documentation*. 2024-01. URL: https://success.outsystems.com/documentation/11/reference/outsystems_language/data/modeling_data/entity/ (visited on 2024-01-31) (cit. on p. 7).

[39] OutSystems. *Low-Code Benefits | Low-Code Guide | OutSystems*. URL: https://www.outsystems.com/guide/low-code/benefits/ (visited on 2024-01-19) (cit. on p. 19).

[40] OutSystems. *OutSystems Developer Cloud ODC Low Code Platform | OutSystems*. URL: https://www.outsystems.com/low-code-platform/developer-cloud/ (visited on 2024-02-08) (cit. on p. 21).

[41] OutSystems. *Project Morpheus - Generative AI*. 2023. URL: https://www.outsystems.com/news/generative-ai-roadmap/ (visited on 2024-02-06) (cit. on p. 33).

[42] OutSystems. *Screen - OutSystems 11 Documentation*. 2023-12. URL: https://success.outsystems.com/documentation/11/developing_an_application/design_ui/screen/ (visited on 2024-01-31) (cit. on p. 8).

[43] OutSystems. *The Main IT Pressure Points of Today | OutSystems*. URL: https://www.outsystems.com/evaluation-guide/the-two-major-it-pressure-points-today/ (visited on 2024-02-06) (cit. on p. 1).

[44] OutSystems. *UI Flow - OutSystems 11 Documentation*. 2024-01. URL: https://success.outsystems.com/documentation/11/reference/outsystems_language/interfaces/designing_the_layout/ui_flow/ (visited on 2024-01-31) (cit. on p. 7).

[45] OutSystems. *What is Low-Code | Low-Code Guide | OutSystems*. URL: https://www.outsystems.com/guide/low-code/ (visited on 2024-01-19) (cit. on p. 19).

[46] Meta Open Source. *React*. 2024. URL: https://react.dev/ (visited on 2024-01-05) (cit. on p. 12).

[47] Evan You and Vite Contributors. *Vite | Next Generation Frontend Tooling*. 2024. URL: https://vitejs.dev/ (visited on 2024-08-17) (cit. on p. 43).

A

OutSystems Platform Compiler

The following source code is an abstraction/simplification of the OutSystems Compiler code. This was made to aid in explaining the compilation process in this thesis. It was also used as a starting point for the implementation of the prototype created during the development phase of this work. This code was written in Java for convenience and familiarity reasons. However, the prototype was implemented using the .NET framework, for the reasons discussed throughout this thesis.

Listing A.1: OutSystems Platform Compiler

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.util.*;
4  import java.util.Map.Entry;
5
6  public class Compiler {
7      CompilationSummary summary;
8      CompilerVersion currentCompilerVersion;
9
10     public Compiler(CompilationSummary summary, CompilerVersion version) {
11         this.summary = summary;
12         this.currentCompilerVersion = version;
13     }
14
15     public Compiler(CompilerVersion version) {
16         this.summary = null;
17         this.currentCompilerVersion = version;
18     }
19
20     public void changeVersion(CompilerVersion version) {
21         this.currentCompilerVersion = version;
22     }
23
24     public void compile(Application app) {
25         Espace espace = app.getEspace();
26
27         // No previous Summary, compiling for the first time
```

```
28          if(summary == null) {
29              summary = new CompilationSummary(currentCompilerVersion);
30          }
31          // Compiler version changed, needs to delete all previous artifacts and recompile
                ↪ app
32          else if(!summary.getCompilerVersion().equals(currentCompilerVersion)) {
33              summary.deleteAllArtifacts();
34              summary = new CompilationSummary(currentCompilerVersion);
35          }
36          // Previous Summary found, needs to perform cleanup
37          else {
38              summary.cleanUp(espace.getCompilationUnitsInfo());
39          }
40
41          espace.compile(summary);
42      }
43  }
44
45  abstract class AbstractObject {
46      Key key;
47      String name;
48      AbstractObject owner;
49
50      AbstractObject(Key key, String name) {
51          this.key = key;
52          this.name = name;
53          this.owner = this;
54      }
55
56      Key getKey() {
57          return this.key;
58      }
59
60      String getName() {
61          return this.name;
62      }
63
64      AbstractObject getOwner() {
65          return this.owner;
66      }
67
68      abstract void compile(CompilationSummary summary);
69  }
70
71  class Espace extends AbstractObject {
72      List<CompilationUnitInfoInModel> unitInfos;
73      List<UIFlow> flows;
74      List<ClientVariable> clientVariables;
75      CompilationHash hash;
76      Path path;
```

```
77
78      Espace(Key key, String name, CompilationHash hash, Path path, List<
            ↪ CompilationUnitInfoInModel> unitInfos, List<UIFlow> flows, List<ClientVariable>
            ↪  clientVariables) {
79          super(key, name);
80          this.unitInfos = unitInfos;
81          this.flows = flows;
82          this.clientVariables = clientVariables;
83          this.hash = hash;
84          this.path = path;
85      }
86
87      CompilationHash getCompilationHash() {
88          return this.hash;
89      }
90
91      Path getPath() {
92          return this.path;
93      }
94
95      List<CompilationUnitInfoInModel> getCompilationUnitsInfo() {
96          return this.unitInfos;
97      }
98
99      void compile(CompilationSummary summary) {
100         // Hash is the same, skip compilation
101         if( hash.equals(summary.getCompilationHash(this.getKey())) ) {
102             return;
103         }
104
105         // Hash changed, needs to compile Espace
106
107         summary.addOrUpdateUnit(key, path, hash);
108
109         for(UIFlow flow : flows) {
110             flow.compile(summary);
111         }
112
113         for(ClientVariable variable : clientVariables) {
114             variable.compile(summary);
115         }
116
117         // Produces a .cs file
118         File cs = new File(this.getKey() + ".cs");
119         try {
120             cs.createNewFile();
121             summary.addArtifact(key, cs);
122         } catch (IOException e) {
123             e.printStackTrace();
124         }
```

```
125     }
126 }
127
128 class ClientVariable extends AbstractObject {
129     ClientVariable(Key key, String name, AbstractObject owner) {
130         super(key, name);
131         this.owner = owner;
132     }
133
134     void compile(CompilationSummary summary) {
135         // Produces a .js file
136         File file = new File(this.getKey() + ".js");
137         try {
138             file.createNewFile();
139             summary.addArtifact(owner.getKey(), file);
140         } catch (IOException e) {
141             e.printStackTrace();
142         }
143     }
144 }
145
146 class UIFlow extends AbstractObject {
147     List<UIFlowNode> nodes;
148     CompilationHash hash;
149     Path path;
150
151     UIFlow(Key key, String name, CompilationHash hash, Path path, List<UIFlowNode> nodes)
        ↪ {
152         super(key, name);
153         this.nodes = nodes;
154         this.hash = hash;
155         this.path = path;
156     }
157
158     CompilationHash getCompilationHash() {
159         return this.hash;
160     }
161
162     Path getPath() {
163         return this.path;
164     }
165
166     List<UIFlowNode> getUIFlowNodes() {
167         return this.nodes;
168     }
169
170     void compile(CompilationSummary summary) {
171         // Hash is the same, skip compilation
172         if( hash.equals(summary.getCompilationHash(this.getKey())) ) {
173             return;
```

```
174         }
175
176         // Hash changed, needs to compile UIFlow
177
178         summary.addOrUpdateUnit(key, path, hash);
179
180         for(UIFlowNode node : nodes) {
181             node.compile(summary);
182         }
183
184         // Produces a .js file
185         File file = new File(this.getKey() + ".js");
186         try {
187             file.createNewFile();
188             summary.addArtifact(key, file);
189         } catch (IOException e) {
190             e.printStackTrace();
191         }
192     }
193 }
194
195 abstract class UIFlowNode extends AbstractObject {
196     CompilationHash hash;
197     Path path;
198
199     UIFlowNode(Key key, String name, CompilationHash hash, Path path) {
200         super(key, name);
201         this.hash = hash;
202         this.path = path;
203     }
204
205     CompilationHash getCompilationHash() {
206         return this.hash;
207     }
208
209     Path getPath() {
210         return this.path;
211     }
212 }
213
214 class Screen extends UIFlowNode {
215     List<Widget> widgets;
216     List<Aggregate> aggregates;
217
218     Screen(Key key, String name, CompilationHash hash, Path path, List<Widget> widgets,
        ↪ List<Aggregate> aggregates) {
219         super(key, name, hash, path);
220         this.widgets = widgets;
221         this.aggregates = aggregates;
222     }
```

```
223
224     List<Widget> getWidgets() {
225         return this.widgets;
226     }
227
228     List<Aggregate> getAggregates() {
229         return this.aggregates;
230     }
231
232     @Override
233     void compile(CompilationSummary summary) {
234         // Hash is the same, skip compilation
235         if( hash.equals(summary.getCompilationHash(this.getKey())) ) {
236             return;
237         }
238
239         // Hash changed, needs to compile Screen
240
241         summary.addOrUpdateUnit(key, path, hash);
242
243         for(Widget widget : widgets) {
244             widget.compile(summary);
245         }
246
247         for(Aggregate aggregate : aggregates) {
248             aggregate.compile(summary);
249         }
250
251         // Produces .js, .cs, .css and .html files
252         File js = new File(this.getKey() + ".js");
253         File cs = new File(this.getKey() + ".cs");
254         File css = new File(this.getKey() + ".css");
255         File html = new File(this.getKey() + ".html");
256         try {
257             js.createNewFile();
258             cs.createNewFile();
259             css.createNewFile();
260             html.createNewFile();
261             summary.addArtifact(key, js);
262             summary.addArtifact(key, cs);
263             summary.addArtifact(key, css);
264             summary.addArtifact(key, html);
265         } catch (IOException e) {
266             e.printStackTrace();
267         }
268     }
269 }
270
271 class Widget extends AbstractObject {
272     Widget(Key key, String name, AbstractObject owner) {
```

```
        super(key, name);
        this.owner = owner;
    }

    void compile(CompilationSummary summary) {
        // Produces .js, .css and .html files
        File js = new File(this.getKey() + ".js");
        File css = new File(this.getKey() + ".css");
        File html = new File(this.getKey() + ".html");
        try {
            js.createNewFile();
            css.createNewFile();
            html.createNewFile();
            summary.addArtifact(owner.getKey(), js);
            summary.addArtifact(owner.getKey(), css);
            summary.addArtifact(owner.getKey(), html);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class Aggregate extends AbstractObject {
    Aggregate(Key key, String name, AbstractObject owner) {
        super(key, name);
        this.owner = owner;
    }

    void compile(CompilationSummary summary) {
        // Produces a .sql file
        File file = new File(this.getKey() + ".sql");
        try {
            file.createNewFile();
            summary.addArtifact(owner.getKey(), file);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class CompilationSummary {
    CompilerVersion compilerVersion;
    Map<Key, CompilationUnitInfoInSummary> unitInfos;

    CompilationSummary(CompilerVersion version) {
        this.compilerVersion = version;
        this.unitInfos = new HashMap<>();
    }

    CompilerVersion getCompilerVersion() {
```

```
323         return this.compilerVersion;
324     }
325 
326     CompilationHash getCompilationHash(Key key) {
327         CompilationUnitInfoInSummary info = unitInfos.get(key);
328         if(info == null) {
329             return null;
330         }
331         else {
332             return info.getCompilationHash();
333         }
334     }
335 
336     void addOrUpdateUnit(Key key, Path path, CompilationHash hash) {
337         this.unitInfos.put(key, new CompilationUnitInfoInSummary(key, path, hash));
338     }
339 
340     void addArtifact(Key key, File artifact) {
341         CompilationUnitInfoInSummary infoInSummary = unitInfos.get(key);
342         if(infoInSummary == null) {
343             return;
344         }
345 
346         infoInSummary.addArtifact(artifact);
347     }
348 
349     void addArtifacts(Key key, List<File> artifacts) {
350         CompilationUnitInfoInSummary infoInSummary = unitInfos.get(key);
351         if(infoInSummary == null) {
352             return;
353         }
354 
355         infoInSummary.addArtifacts(artifacts);
356     }
357 
358     void cleanUp(List<CompilationUnitInfoInModel> list) {
359         Iterator<Entry<Key, CompilationUnitInfoInSummary>> it = unitInfos.entrySet().
             ↪ iterator();
360         while(it.hasNext()) {
361             CompilationUnitInfoInSummary infoInSummary = it.next().getValue();
362             boolean found = false, deleteArtifacts = true;
363             for(CompilationUnitInfoInModel infoInModel : list) {
364                 if(infoInSummary.getKey().equals(infoInModel.getKey())) {
365                     found = true;
366                     if(infoInSummary.getCompilationHash().equals(infoInModel.
                         ↪ getCompilationHash())) {
367                         deleteArtifacts = false;
368                     }
369                     break;
370                 }
```

```
371             }
372             if(!found) {
373                 it.remove();
374             }
375             if(deleteArtifacts) {
376                 infoInSummary.deleteArtifacts();
377             }
378         }
379     }
380 
381     void deleteAllArtifacts() {
382         for(Entry<Key, CompilationUnitInfoInSummary> e : unitInfos.entrySet()) {
383             CompilationUnitInfoInSummary info = e.getValue();
384             info.deleteArtifacts();
385         }
386     }
387 }
388 
389 class CompilationUnitInfoInModel {
390     Key key;
391     Path path;
392     CompilationHash hash;
393 
394     CompilationUnitInfoInModel(Key key, Path path, CompilationHash hash) {
395         this.key = key;
396         this.path = path;
397         this.hash = hash;
398     }
399 
400     CompilationHash getCompilationHash() {
401         return this.hash;
402     }
403 
404     Key getKey() {
405         return this.key;
406     }
407 
408     Path getPath() {
409         return this.path;
410     }
411 }
412 
413 class CompilationUnitInfoInSummary {
414     Key key;
415     Path path;
416     CompilationHash hash;
417     List<File> artifacts;
418 
419     public CompilationUnitInfoInSummary(Key key, Path path, CompilationHash hash) {
420         this.key = key;
```

```
421         this.path = path;
422         this.hash = hash;
423         this.artifacts = new LinkedList<File>();
424     }
425 
426     CompilationHash getCompilationHash() {
427         return this.hash;
428     }
429 
430     Key getKey() {
431         return this.key;
432     }
433 
434     Path getPath() {
435         return this.path;
436     }
437 
438     List<File> getArtifacts() {
439         return this.artifacts;
440     }
441 
442     void deleteArtifacts() {
443         for(File file : artifacts) {
444             file.delete();
445         }
446 
447         this.artifacts = new LinkedList<File>();
448     }
449 
450     void addArtifact(File artifact) {
451         this.artifacts.add(artifact);
452     }
453 
454     void addArtifacts(List<File> artifacts) {
455         for(File artifact : artifacts) {
456             this.artifacts.add(artifact);
457         }
458     }
459 }
```

B

Results tables

Table B.1: No changes

Model no.	Screen Count	Size (Bytes)	Save model time (ms)	Load model time (ms)
1	22	3409756	223.05	339.4
2	59	2435726	216.23	419.37
3	42	2062217	186.99	339.93
4	42	2713128	236.47	434.76
5	8	2594774	126.33	175.31
6	6	2803746	164.41	228.79
7	7	6043970	224.36	97.63
8	23	4784690	298.46	448.23
9	2	5546637	412.68	632.76
10	6	2296936	151.63	219.82
11	0	3514918	97.02	30.5
12	27	2153393	187.4	295.58
13	47	3118894	235.02	365.26
14	34	3558532	304.51	495.64
15	3	2667122	217.17	77.35
16	7	2568287	186.64	245.9
17	8	2970285	252.59	424.92
18	1	2716744	209.91	319.42
19	12	2547396	206.8	314.26
20	6	2293022	145.56	226.52
21	128	10866955	528.13	609.55
22	62	2851593	206.26	284.87
23	21	2912410	212.74	290.43
24	7	2318875	153.9	226.31
25	11	2036653	135.18	203.59
26	0	2131028	97.19	38.87

27	0	13025341	535.81	116.9
28	0	3184936	137.48	46.23
29	0	17384763	583.21	125.88
30	5	2321242	148.58	222.9
31	0	20053345	728.15	390.87
32	8	3836989	142.26	278.56
33	0	4119751	286.1	63.76
34	31	2460162	143.98	220.62
35	6	2292106	144.63	228.89
36	21	2324330	188.55	253.88
37	7	2092198	144.6	236.26
38	29	3908351	320.78	585.02
39	27	2186935	183.25	333.92
40	13	2616376	136.31	164.3
41	0	3559932	239.22	69.12
42	20	3939453	229.72	318.27
43	0	14742902	674.05	116.75
44	0	2967706	136.99	38.33
45	20	2799247	190.62	342.96
46	8	2438827	145.97	250.14
47	0	7177274	494.04	1376.36
48	16	3457909	310.47	576.08
49	26	3842871	319.83	491.8
50	0	13501970	336.53	29.01
51	16	4286928	240.35	352.1
52	6	2888902	163.35	240.54
53	27	1967913	194.24	327.51
54	33	3555776	282.24	493.8
55	10	2121972	165.32	272.43
56	31	3287564	301.43	628.07
57	0	3018792	281.65	506
58	27	2830881	235.51	532.72
59	125	10781443	525.64	566.86
60	13	2657025	132.35	208.88
61	6	2312106	147.63	218.82
62	6	2300681	147.46	214.33
63	0	4940907	135.21	28.81
64	15	2318426	191.97	316.92
65	23	2828094	250.79	404.16
66	30	3919518	320.28	533.77

67	25	2897617	243.59	407.61
68	6	2350630	154.34	235.49
69	0	2792953	293.38	168.12
70	12	3966395	236.26	314.31
71	7	2333769	152.43	230.23
72	12	13222589	387.86	148.96
73	6	2362382	153	233.93
74	6	2429247	155.7	251.25
75	7	2677841	162.01	251.61
76	16	2286518	184.58	301.61
77	9	2846619	165.21	160.72
78	10	1998843	114.08	179.44
79	0	1954405	182.75	321.79
80	53	3947589	345.39	857.4
81	13	1954605	165.8	472.09
82	7	1984847	152.14	250.71
83	0	3240725	273.49	61.24
84	6	2320434	147.74	225.84
85	5	2878349	165.93	226.89
86	20	4015787	223.47	315.64
87	5	11331944	360.71	144.71
88	6	2589457	169.38	310.42
89	10	3082171	192	332.63
90	4	4621694	175.19	184.11
91	6	6091810	263.98	230.3
92	26	5294594	210.8	118.28
93	10	1942623	157.99	259.61
94	1	5732696	415.93	622.89
95	22	2578514	221.26	341.28
96	0	2256444	65.61	21.15
97	0	3358725	168.68	184.68
98	0	7255647	328.19	97.96
99	8	4842664	304.5	457.2
100	26	5436112	212.03	122.71

Table B.2: One change in one Screen

Model no.	Screen Count	Save model time (ms)	Load model time (ms)
1	22	425.03	333.15
2	42	780.09	418.76

3	42	697.07	332.57
4	59	613.67	435.06
5	8	244.16	185.19
6	6	196.41	220.31
7	7	331.13	100.85
8	23	809.34	450.98
10	6	169.9	215.92
11	27	407.76	305.18
13	34	1066.28	568.73
14	47	683.73	368.46
15	3	359.84	63.35
16	7	397.53	248.13
17	8	674.87	444.74
19	12	533.93	308.65
20	6	161.28	211.68
21	128	1231.01	621.03
22	62	1191.13	278.05
23	21	432.94	293.77
24	7	169.34	214.45
25	11	169.79	197.57
30	5	171.08	222.59
32	8	170.19	258.67
34	31	289.2	206.87
35	6	165.98	214.43
36	21	558.06	268.17
37	7	177.2	220.23
38	29	851.97	591.17
39	27	444.76	336.35
40	13	186.32	156.1
42	20	543.78	347.31
44	20	502.62	319.83
46	8	198.44	237.98
48	16	1163.8	584.57
49	26	878.36	532.17
50	16	625.83	360.68
52	6	188.29	212.71
53	27	607.72	356.26
54	33	709.85	471.7
55	10	387.27	276.38
56	27	655.66	545.99

57	31	1133.2	658.13
59	125	1235.43	576.48
60	13	233.12	190.91
61	6	167.74	219.55
62	6	169.57	202.77
64	15	557.64	312.98
65	30	992.5	518.33
66	23	737.13	387.83
67	25	597.8	437.52
68	6	179.99	228.14
69	12	296.47	289.07
71	7	172.15	221.9
72	12	512.63	140.33
73	6	270.47	229.12
74	6	178.67	246.39
75	7	182.86	239.81
76	16	574.16	315.37
78	10	278.51	180.99
79	53	1299.21	864.31
81	13	423.19	450.35
82	7	250.32	251.22
84	5	176.56	226.94
85	20	429.75	350.61
86	6	164.1	208.95
87	5	461.5	141.15
88	6	194.05	311.02
89	10	380.45	335.56
90	4	264.65	189.4
91	6	325.54	221.1
92	26	594.56	124.52
93	10	329.23	255.54
95	22	514.53	372.43
99	8	779.11	446.74
100	26	571.99	121.24

Table B.3: Many changes in one Screen

Model no.	Screen Count	Save model time (ms)	Load model time (ms)
1	22	432.1	336.74
2	42	731.45	418.56

3	42	693.61	328.62
4	59	612.53	435.1
5	8	235.13	182.6
6	6	186.83	220.97
7	7	326.31	99.7
8	23	781.88	460.03
10	6	168.5	214.31
11	27	395.61	305.11
13	34	1004.62	569.7
14	47	648.83	370.18
15	3	353.66	63.78
16	7	378.27	249.55
17	8	666.16	460.46
19	12	526.08	306.5
20	6	162.78	210.8
21	128	1250.09	620.01
22	62	1187.38	274.65
23	21	429.89	291.93
24	7	168.38	214.03
25	11	171.42	198.63
30	5	170.86	222.26
32	8	174.2	260.08
34	31	286.96	204.93
35	6	164.94	215.14
36	21	557.58	269.1
37	7	177.15	232.27
38	29	861.11	588.09
39	27	446.85	342.57
40	13	187.63	153.43
42	20	544.33	349.83
44	20	499.94	323.16
46	8	196.61	239.31
48	16	1159.34	584.53
49	26	881.7	525.2
50	16	629.99	354.65
52	6	191.79	214.35
53	27	643.59	360.45
54	33	735.23	472.39
55	10	379.19	278.9
56	27	663.84	543.8

57	31	1140.42	653.34
59	125	1237.9	580.86
60	13	230.35	191.63
61	6	167.42	211.71
62	6	167.85	202.07
64	15	557.66	316.28
65	30	993.66	517.08
66	23	730.76	387.33
67	25	600.72	436.22
68	6	181.55	227.84
69	12	295.11	292.7
71	7	170.72	220.35
72	12	509.14	140.91
73	6	270.36	230.23
74	6	180.85	242.06
75	7	180.54	239.07
76	16	563.09	317.15
78	10	293.23	177.54
79	53	1306.12	872.63
81	13	420.6	451.06
82	7	253.47	246.37
84	5	177.16	227.01
85	20	459.18	352.77
86	6	164.02	209.47
87	5	460.07	144.04
88	6	195.02	311.08
89	10	379.69	331.69
90	4	255.02	190.73
91	6	324.06	223.95
92	26	617.85	123.07
93	10	329.05	253.52
95	22	514.12	365.69
99	8	785.53	446.33
100	26	596.77	122.45

Table B.4: One change in all Screens

Model no.	Screen Count	Save model time (ms)	Load model time (ms)
1	22	484.66	336.79
2	42	898.97	421.01

3	42	842.05	330.83
4	59	805.68	429.52
5	8	240.75	181.6
6	6	192.15	221
7	7	325.19	99.21
8	23	903.92	453.76
10	6	172.68	215.59
11	27	417.28	285.84
13	34	1168	564.48
14	47	824.4	374.57
15	3	354.61	65.2
16	7	398.87	257.76
17	8	618.46	461.93
19	12	575.68	315.28
20	6	164.55	223.68
21	128	1439.23	648.23
22	62	1332.97	283.96
23	21	482.28	292.22
24	7	173.52	224.65
25	11	177.46	197
30	5	172.95	224.54
32	8	178.13	269.86
34	31	357.06	210.25
35	6	168.87	221.19
36	21	664.02	277.3
37	7	180.66	221.53
38	29	909.86	610.09
39	27	497.96	348.56
40	13	187.6	155.26
42	20	594.79	350.73
44	20	597.59	340.26
46	8	198.68	235.18
48	16	1354.36	598.53
49	26	977.4	534.47
50	16	708.91	357.61
52	6	191.45	211.53
53	27	743.08	362.31
54	33	958.94	479.25
55	10	417.76	279.93
56	27	802.73	545.14

57	31	1190.73	657.19
59	125	1436.21	592.73
60	13	243.17	192.88
61	6	169.9	211.18
62	6	172.04	199.66
64	15	641.31	314.29
65	30	1121.71	511.17
66	23	841.17	385.38
67	25	745.11	435.93
68	6	182.53	226.96
69	12	295.36	294.1
71	7	172.32	222.4
72	12	521.77	140.5
73	6	277.06	228.57
74	6	181.67	246.39
75	7	186.9	238.52
76	16	666.95	320.46
78	10	302.36	179.4
79	53	1502.96	862.68
81	13	452.47	449.75
82	7	262.55	246.57
84	5	181.63	226.52
85	20	441.93	350.77
86	6	169.18	210.96
87	5	457.98	140.76
88	6	194.52	314.94
89	10	381.09	329.08
90	4	260.89	191.01
91	6	319.1	221.45
92	26	564.71	116.86
93	10	349.36	255.76
95	22	656.56	367.01
99	8	796.73	448.46
100	26	561	122.6

Table B.5: Many changes in all Screens

Model no.	Screen Count	Save model time (ms)	Load model time (ms)
1	22	498.25	338.39
2	42	914.72	422.13

3	42	863.11	328.43
4	59	822.82	429.88
5	8	244.49	183.8
6	6	194.58	217.31
7	7	337.31	99.27
8	23	941.24	458.25
10	6	170.34	213.88
11	27	422.31	302.23
13	34	1184.68	560.73
14	47	844.82	362.86
15	3	346.99	63.71
16	7	415.98	250.8
17	8	698.41	474.47
19	12	567.12	306.72
20	6	164.75	213.13
21	128	1456.08	616.07
22	62	1424.64	278.68
23	21	492.29	289.7
24	7	174.42	216.7
25	11	178.35	195.43
30	5	173.28	218.14
32	8	177.1	260.85
34	31	366.97	206.22
35	6	169.05	212.56
36	21	678.22	268.74
37	7	179.2	216.56
38	29	922.66	590.2
39	27	502.95	342.04
40	13	190.41	151.78
42	20	606.67	340.44
44	20	676.4	320.85
46	8	200.3	238.96
48	16	1379.34	594.22
49	26	973.29	529.27
50	16	747.33	361.24
52	6	188.74	213.15
53	27	771.51	357.68
54	33	989.75	477.02
55	10	424.05	279.8
56	27	813.43	541.24

57	31	1436.08	659.81
59	125	1465.3	592.82
60	13	243.11	189.58
61	6	172.83	213.45
62	6	170.33	209.1
64	15	666.92	314.56
65	30	1127.14	517.03
66	23	853.55	385.98
67	25	749.68	428.64
68	6	183.63	227.83
69	12	295.85	293.2
71	7	172.53	219.59
72	12	519.99	139.88
73	6	281.73	233.16
74	6	182.71	244.99
75	7	182.42	238.98
76	16	666.84	287.89
78	10	354.41	178.84
79	53	1786.35	868.77
81	13	493.64	450.62
82	7	264.3	246.26
84	5	181.73	224.38
85	20	514.23	353.67
86	6	170.96	211.94
87	5	455.79	140.53
88	6	195.96	306.69
89	10	381.18	330.46
90	4	259.93	192.25
91	6	320.8	220.76
92	26	659.06	117.14
93	10	347.61	256.13
95	22	667.1	367.01
99	8	806.11	448.62
100	26	639.72	120.41