NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

LENINO MANUEL LIMA DIAS
Bachelor in Computer Science

OUTSYSTEMS LOGIC PREVIEWER

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

DEPARTMENT OF
COMPUTER SCIENCE

OUTSYSTEMS LOGIC PREVIEWER

LENINO MANUEL LIMA DIAS
Bachelor in Computer Science

Adviser: Bernardo Parente Coutinho Fernandes Toninho
Assistant Professor, NOVA University of Lisbon

Co-adviser: Vasco Andrade e Silva
Product Designer, OutSystems

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

OutSystems Logic Previewer

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.7.1) [17].

To my family.

Acknowledgements

I would first like to thank my advisers, Vasco Silva, from OutSystems and Bernardo Toninho, from Nova School of Science and Technology. Thank you, Vasco, for always being available to answer all my questions and for pushing me to produce a great work. Also, for all your advice and patience during the development of this dissertation. Thank you so much, Professor Bernardo, for your support during the writing of this dissertation, for your patience to review this entire report and for all your orientation to help develop this work. To both of you, I appreciated the weekly follow up that was crucial to achieve the goals of this work.

Furthermore, a special thanks to Nova School of Science and Technology - Nova University of Lisbon, especially to the Computer Science Department for all the knowledge that I acquired for the last six years, preparing me with the fundamental tools that will allow me to start my professional career.

I would like to acknowledge OutSystems for providing me a scholarship for this dissertation and for the great environment where I was inserted to produce this work. A thank you to everyone at OutSystems for all the support and help, especially to the model service team. I am so grateful for the time spent at OutSystems, for the connections established with the amazing people that work in this company and for the opportunity to experience this great environment.

In addiction, I would like to thank my friends and colleagues, especially my colleague Christophe Cruz for all the support and help during this course, for all projects we developed together and for the snacks we shared. A special thank you to Laura Dantas for helping review this report and for all the support.

Finally, and most important, I want to thank my parents. Thank you so much for the efforts you made to give me the opportunity to study abroad. You made all of this possible and I will be forever grateful for everything you have done for me. Thank you for all your help, support, inspiration, and for building the person that I am today. Your lessons will never be forgotten. Thank you so much to my brothers for all the support and for taking care of our parents while I am away.

"You cannot teach a man anything; you can only help him discover it in himself." (Galileo)

Abstract

Low-Code Platforms have been increasingly adopted by companies to develop their software products, since visual programming languages provide a faster development process by removing various user concerns about implementation details. OutSystems allows its users to create software in an agile form in order to improve their productivity, as it is one of the key aspects to fast software delivery.

OutSystems development processes are very fast and allow the creation of state-of-the-art products in a very short time when compared to more traditional approaches. However, the feedback loop received by users at OutSystems during development is very long because the existing methods for checking application behavior, such as using the application UI (or a dummy one) to test its behavior, or even using testing tools (such as *BDDFramework*), do not have an immediate feedback. Using tests to check application behavior in OutSystems is also a time-consuming process because of the required *publishes* (compilation and deployment to server) of the tests and the code. This problem in the OutSystems development significantly decreases users productivity.

Therefore, the main goal of this work is to transform the OutSystems applications logic to allow developers at OutSystems to preview the effects/side-effects of the logic being built. We also aim to provide to developers all the information related to the logic dependencies to the database (during development), to help them control and visualize the dependencies. The final goal is to reach a logic previewer that represents a visualization of the logic effects/side-effects in a fast feedback loop. By achieving this, we aim to increase developer productivity in the OutSystems development process.

Because logic previewing is a very large feature, we will focus this work on transforming all Server Actions within an application Modules being built, with the purpose of making them return all their side-effects to the database. Along with this, we will also provide to the developer all the possible reads and writes that each Server action can perform to the database (direct and indirectly).

Keywords: OutSystems, Logic Preview, Software Testing, Feedback Time, Visual Programming Language, OutSystems Development Process, Side-effects to Database.

Resumo

OutSystems é uma plataforma *Low-Code* que permite aos seus utilizadores criar as suas aplicações seguindo o desenvolvimento *agile* de forma a melhorar a produtividade, sendo que permite uma rápida produção de software.

Os processos de desenvolvimento em OutSystems são muito rápidos e permitem a criação de aplicações modernas num curto período de tempo. No entanto, o ciclo de *feedback* recebido pelos utilizadores durante o processo de desenvolvimento em OutSystems (na criação ou alteração de código) é muito longo, porque os métodos existentes para testar o comportamento das aplicações, tais como testar a aplicação com a sua interface (ou uma fictícia) ou mesmo usar frameworks de testes como o *BDDFramework*, não têm um *feedback* imediato. O uso de testes para verificar a lógica das aplicações em OutSystems é um processo que gasta muito tempo, uma vez que requer que ambos os testes e o código sejam compilados, publicados, executados e que os efeitos/efeitos secundários da lógica sejam manualmente e indiretamente verificados (ir à base de dados ver os registos, por exemplo). Este processo faz com que a produtividade dos utilizadores em OutSystems diminua consideravelmente.

Portanto, o objetivo final é construir uma ferramenta que permita pré-visualizar os efeitos/efeitos secundários da lógica, permitindo um ciclo de *feedback* mais rápido. Uma vez que esta funcionalidade é bastante ampla, o foco deste trabalho consiste em transformar todas as *Server Actions* dentro dos módulos da aplicação sendo desenvolvida, de forma a fazê-los retornar todas as suas leituras e escritas à base de dados. Para além disso, pretendemos permitir ao desenvolvedor visualizar todas as possíveis leituras e escritas que podem ser feitas à base de dados por parte de cada *Server Action* (direto e indiretamente), para os ajudar a visualizar e controlar as dependências. Cumprindo estes objetivos, espera-se que a produtividade durante o processo de desenvolvimento em OutSystems aumente consideravelmente.

Palavras-chave: OutSystems, Visualização de Lógica, Testagem de Software, Tempo de *feedback*, Linguagens de Programação Visual, Operações à Base de Dados.

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
CRUD	Create, Read, Update and Delete
DLL	Dynamic Link Library
IDE	Integrated Development Environment
JIT	Just-in-time
OLP	OutSystems Logic Previewer
PoC	Proof of Concept
REPL	Read-Eval-Print-Loop
SBE	Specification By Example
TDD	Test-Driven Development
UI	User Interface
VPL	Visual Programming Language

1 Introduction

OutSystems is a software company that is the leader in the Magic Quadrant[1] for Low-Code Application Platforms (LCAP) 2019 [37]. The OutSystems Platform is a powerful, feature-packed low-code development platform for large enterprises or developers looking to publish straight to consumer application stores [18]. By using visual, model-driven development and AI-powered tools, the OutSystems platform improves the entire application lifecycle; which combine with a cloud-native platform so that users can quickly build, deploy and manage their software [49]. The Visual Programming Languages (VPL) allow users to create software products by focusing on the intended functionality without stressing with implementation details.

One of OutSystems goals is to get people to produce state-of-the-art applications in a very short period of time. The fast development process in OutSystems environment is due to the efficiency provided by the usage of a VPL by the platform, the excellent architecture, among other important aspects.

1.1 Motivation

OutSystems users, while developing their application, put together the components (*widgets*, *screens*, *actions*, *etc.*) and make the necessary modifications that are needed to obtain the expected functionalities of the application. During this development process, which is generally faster than more traditional approaches, users are continuously writing the application logic in order to obtain the desired application behavior.

However, there is no way to visualize the effects/behavior of the defined logic without a complete application. During the development process of writing and changing the application logic, users have no feedback about what they are implementing. Nevertheless, users need to know the effects of their code and its behavior, and to accomplish this they usually follow some approaches such as: testing the application using its UI or a dummy UI (for testing purposes), testing the application using testing frameworks, or checking application misbehavior with the debugger of the platform.

[1]https://www.gartner.com/en/research/magic-quadrant

These approaches are not suitable to interactively visualize application logic, since the feedback loop they provide to users (developer at OutSystems) is very long, causing losses of productivity (these approaches have their own limitations that are covered in more detail in the chapter 2).

1.2 Goals

The final (OutSystems) goal is to enable the preview of logic in OutSystems development processes. By doing this, we aim to reduce the feedback loop that developers experience during the development process (creating and changing code) in OutSystems environment.

Because this is a very large feature, we focused our work in previewing the applications logic side-effects (reads and writes) to the database. We aim to allow the OutSystems developers to preview all the reads and writes that are performed to the database by the logic being built and all the possible reads and writes that can be performed to the database.

This work introduces a new concept in OutSystems development processes: **Logic Previewer**. We want to allow the developers to create/change their code and be able to preview its effects/side-effects (as soon as possible), thus improving the feedback cycle that they have about the code logic being developed and, consequently, their productivity.

With this work we aim to built a solution that transforms the logic being written within an application to enable the developers to visualize its effects (or part of the effects) while it is being written. That allows the developers to preview and control the side-effects, as well as the dependencies existing between the application logic and the database.

1.3 Key Contributions

Reaching the goals of this work, we get a running prototype tool capable of transforming the application logic, with the main purpose of providing the developer a preview of the logic effects and the dependencies between them. That improves the developing process at OutSystems, as our solution allows the logic visualization of the application being developed.

We provide developers at OutSystems with a PoC tool designed with an API capable of manipulating the OutSystems model (the Model API) and thus, can be added into the OutSystems development process (Service Studio). This tool can also be used aside with other testing techniques to reinforce the confidence level of the applications logic.

This work brings another great contribution that are the analyses performed to the real-world OutSystems code. These analyses allow us to understand several information about the OutSystems applications code, such as the distribution of the Actions, logic, aspects related to the application modules, among several others that help to understand many development patterns and project improvements.

These performed analyses are very important for the OutSystems product managers to analyse and improve their product, based on the obtained results.

1.4 Document Structure

This dissertation is organized as follows:

- **Chapter** 1 - **Introduction**: this chapter gives an overview of OutSystems enterprise and its platform. It also introduces the problem to address and the goals to achieve with this work.
- **Chapter** 2 - **Background**: describes the OutSystems Platform, the theoretical concepts related with this dissertation such as Testing, and how this concepts apply to the OutSystems environment.
- **Chapter** 3 - **Related Work**: this chapter presents some techniques and approaches related to the context of this work, essentially the logic visualization.
- **Chapter** 4 - **Proposed Work**: presents an overview of the problem addressed in the context of this dissertation, along with some data analyses used to refine the problem and the proposed approach to be followed to build a solution.
- **Chapter** 5 - **Implementation**: this chapter explains all the details of the prototype implementation and its use.
- **Chapter** 6 - **Evaluation**: describes the evaluation performed to the implemented tool and the results that we achieved.
- **Chapter** 7 - **Conclusions**: concludes all the work done, highlights the main contribution of this dissertation, as well as the future work that needs to be done.

2 Background

The OutSystems platform consists of several tools and cloud solutions for developers, administrators, and operators, allowing rapid application development with advanced capabilities for enterprise mobile and web applications [39].

2.1 OutSystems Platform

The OutSystems platform allows its users to create both web and mobile applications supported by a visual language, focusing mainly on assembling visual components to obtain the desired products, by reducing significantly the users concerns about implementation details. This makes the development experience in OutSystems very fast and scalable. All these approaches combined make the OutSystems platform much faster than more traditional development, with more quality and requiring less coding efforts from its developers.

The platform empowers its users (developers) with high-productivity, connected, AI-assisted tools for faster development of a full range of applications [49]. The following sections will explain in more detail the main concepts of the OutSystems platform architecture, run-time and development experience.

2.1.1 Architecture

The OutSystems platform architecture can be visualized in Figure 2.1, that demonstrates the main components that together compose the platform. It can be divided into these main components: *Service Studio*, *Integration Studio* and *Code Generator and Optimizer (Platform Server)* [41].

2.1.1.1 Service studio

Service Studio is the environment available to developers to create all parts of the application stack: *the data model*, *application logic*, *UI*, *business process flows*, *integrations*, and *security policies* [38]. This IDE allows the user to drag and drop visual elements to create UIs, business processes, business logic and data models for their application [38]. After

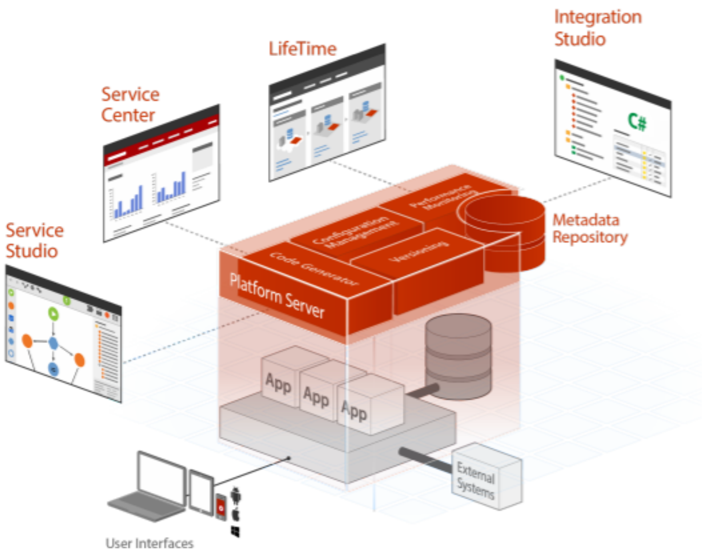


Figure 2.1: OutSystems platform: the Architecture Overview [41].

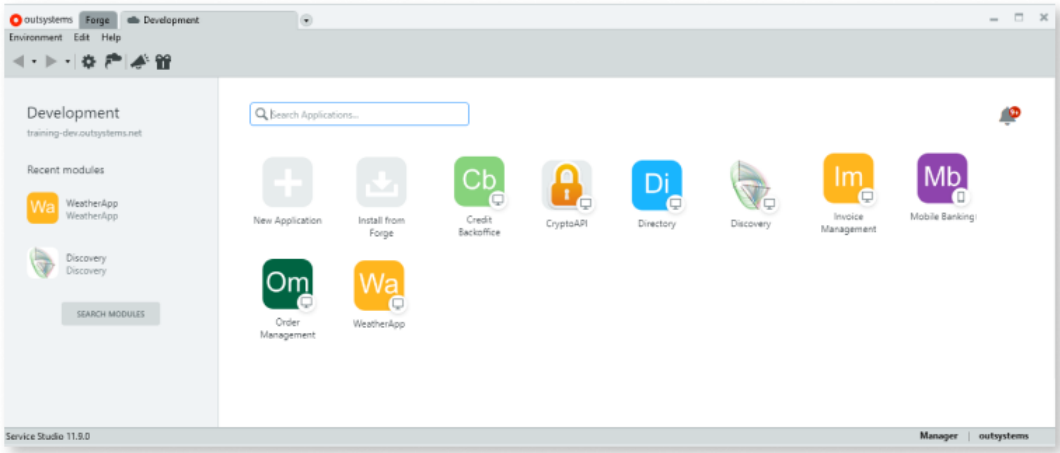


Figure 2.2: Service Studio Workspace [44].

opening Service Studio and connecting to the OutSystems environment in the cloud (or an on-premise server in user's data) the user is able to see the list of applications held on the server or, additionally, create a new one [44]. Figure 2.2 shows the Service Studio interface.

During the development of an application, the IDE will present the users with four main development areas: the *Interface* tab, the *Data* tab, the *Logic* tab and the *Processes* tab, as shown in Figure 2.3.

Interface tab: This layer is used to define the UI flows of the applications. It is composed by a group of **widgets** (building blocks of a screen) that can be dragged and dropped to the canvas to compose the desired screen interface. Also, in this layer, it can

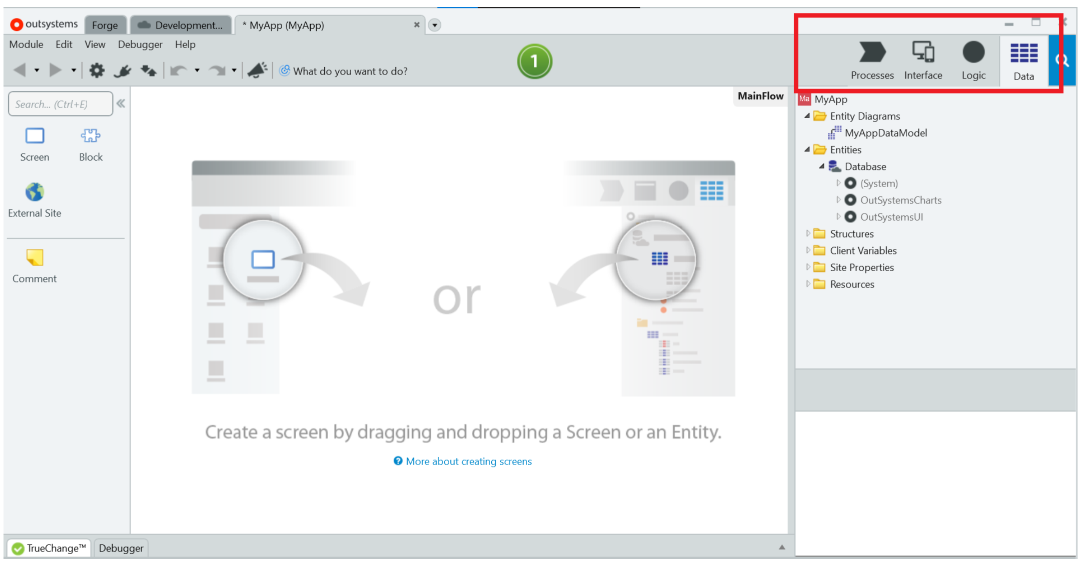


Figure 2.3: Service Studio: Example of the 4 Different Layers.

be applied different **styles** to the components as desired. It is also available a **widget tree** to help the developers with the complexity of the widgets numbers [45].

Logic tab: Similar to the UI, this layer has a canvas in the middle of the workspace with visual representation of the application logic. There is also a **toolbox**, identical to the UI, but instead of showing widgets, it shows the elements that can be used inside the logic flows [45]. The Logic layer is divided into **Client Actions** (that runs on the device) and **Server Actions** (that runs on the server) [26]. It also has logic elements that allow the user to integrate with external systems, such as *SOAP Web Services*, *RESTful Services* and *SAP*.

Data tab: This is the layer where the user is able to define all the **entities** in the database or in the local storage [45]. There is the possibility for users to create **entity diagrams** to see the visual representation of the data (the data model). In the Data layer it is also held **Structures** (in-memory representation of the data), **Client Variables** (user-specific data in the client side), **Site Properties** (cross application data on the server) and **Resources** (other types of data) [26].

2.1.1.2 Integration Studio

Integration Studio is the OutSystems environment where the users can create components that extend the OutSystems platform and integrate with third-party systems [38]. One of the key-points of the OutSystems platform is that once the components are deployed, they can be reused by all applications built with OutSystems. Integration Studio allows the creation of **Extensions**, which are sets of actions, structures and entities that increment OutSystems and allows the integration with external systems [32].

2.1.1.3 Platform Server

The platform Server is the server component and the core of OutSystems platform. It encompasses all the steps required to generate, build, package and deploy applications. These steps are more detailed below [40]:

- *Code Generator*: The code generator service is responsible for receiving the application model and generate all the native application components, ready to be deployed to an application server. This process includes checking for external dependencies, applying optimizations, generating native code for all layers, among other key tasks.
- *Deployment Services*: These services deploy the generated application components to an application server and ensure that the application is consistently installed on each front-end server of an organization's server farm (factory that uses OutSystems developing technologies). The deployment service deploys a .NET [19] application on a specific front-end server.
- *Application* Services: These are services for managing the applications at runtime. It can be divided into *Scheduler Service* (the Scheduler Service manages the execution of scheduled tasks) and *Log Service* (applications are automatically instrumented to create error, audit, and performance logs).

2.1.2 OutSystems Language - A visual language

The aim of this section is to explain in more detail how the OutSystems language works and what are the key elements of the language. As mentioned before, OutSystems is a VPL dedicated to answer the challenges of digital transformation, mobile and faster delivery cycles [52].

OutSystems uses a notion of *modular programming* when developing applications, called OutSystems *Modules* [36]. Modules encapsulate everything to execute one particular aspect of functionality. In OutSystems a module is where a UI and business logic code are developed. To have a complete overview of the language it is important to briefly describe the most important elements/components of the language such as *Screens*, *Actions*, *Entities*, *Nodes* and some others that are relevant for the scope of this dissertation.

2.1.2.1 Screens

A screen is an interface element composed by other elements that the user can interact with. Screens can be used to create web pages and mobile screens in the user's applications [42]. During the development, when users want to create a new Screen, they can create an *empty screen* or use a *Screen Template* [43]. By creating an *empty screen*, the user has to choose the *layout*, *widgets*, *components*, *styles* and *logic*. Unlike the *empty screen*, *screen templates* have predefined values that can increase productivity. An example of how screens can be created in OutSystems is shown in Figure 2.4

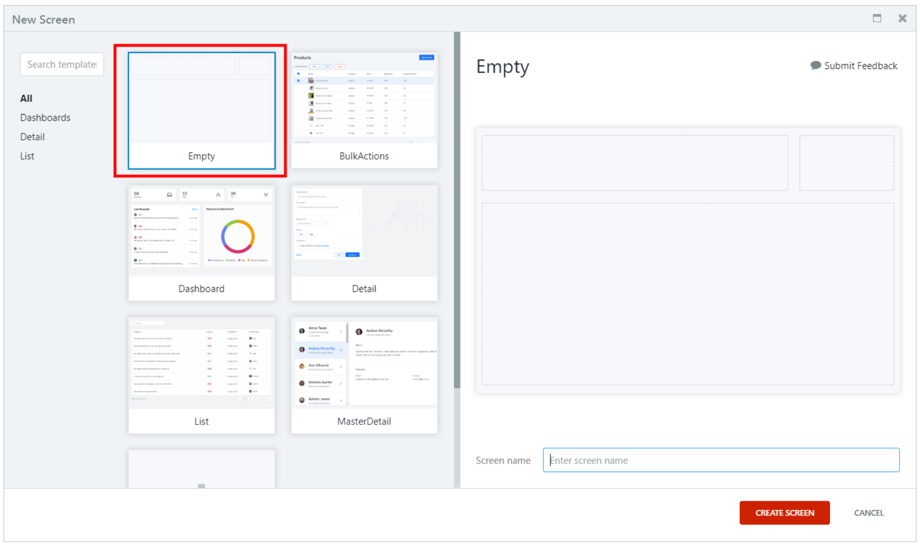


Figure 2.4: Service Studio: Create a New Screen [42].

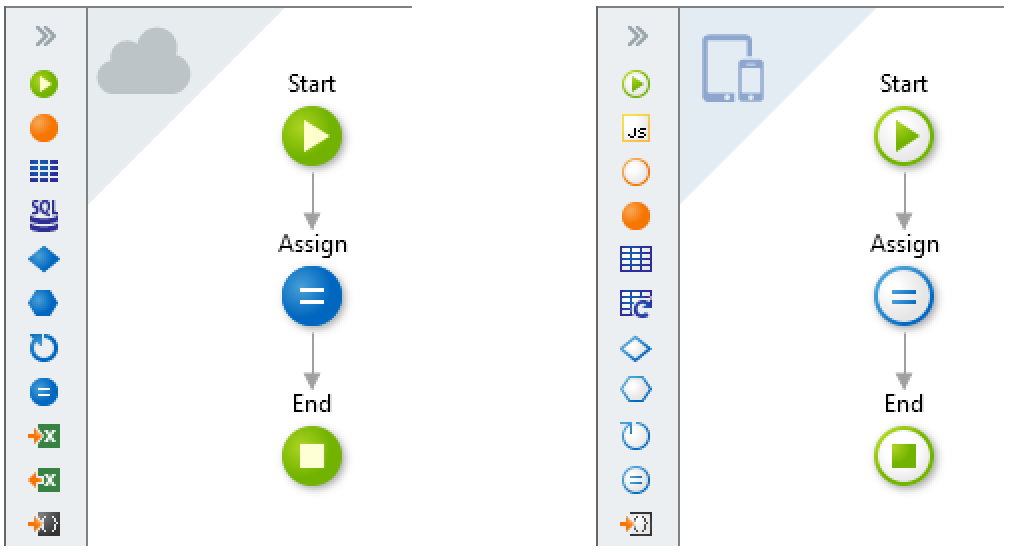


Figure 2.5: Service Studio: Representation of Actions [23].

2.1.2.2 Actions

In OutSystems, Actions are logic that runs on the **server** and logic that runs on the **client** device (like tablet or smartphone). The user can create the following Actions: **Data Actions**, **Client Actions** and **Server Actions** [23]:

Data Actions: are actions that run on the server. The user can create Data Actions to fetch complex data from a database, when the retrieval cannot be achieved using a single server aggregate [25] or to fetch data from an external system, such as REST API[1].

Client Actions: The user can set a Client Action as a function and use it directly in expressions (Action flow) of the client-side logic [23]. Client Actions run logic on the user

[1]https://restfulapi.net/

device. The user is able to create Client Actions in two different scopes: The **Screen** and the **Client**. The difference is that in the scope of a Screen the user is allowed to run logic when the user interacts with the Screen, whereas in the client logic the user is allowed to encapsulate logic to reuse in several Screens. On the right side of Figure 2.5, it can be seen how the client logic is represented in the Service Studio.

Server Action: The users also have the ability to create Server Actions in the application to encapsulate the logic and be able to reuse it in other Actions, such as other Server Actions, Data Actions or Client Actions. These kinds of Actions run logic in the server and are represented in Service Studio on the left side of Figure 2.5.

2.1.2.3 Data Model - Entities

Entities are elements that allow the user to persist information in the database and to implement the database model [31]. Entities are similar to tables or views in a relational database[2]. Entities are composed by:

- **Primary key**: In OutSystems, a primary key is called *Entity Identifier*. It is created automatically as an attribute called Id and added to the Entity.
- **Sequential Attributes**: are normally useful for Entity Identifier attributes. It is an easy way to ensure that each record has a unique primary key.
- **Indexes**: Similar to relational databases, OutSystems provides indexes for faster access to data in the entity.

Apart from that, in OutSystems, the user can employ **Aggregates** to fetch data using optimized queries. Aggregates can load data from the server of the local database, and they support the combination of several Entities and advanced filtering [25]. Aggregates can be divided into *Client-side aggregates* - run in the client logic; and *Server-side aggregates* - the ones that are in the logic flows.

2.1.2.4 Nodes

Figure 2.6 shows an Action Flow built in Service Studio, where different types of nodes such as *Start*, *Aggregate* and *End* can be seen. The nodes are combined in a graph representation of functions, procedures, methods, etc. We now detail some of the OutSystems platform nodes that are relevant for this work:

- **Aggregates**: as already mentioned, aggregates are used to fetch data using an optimized query. They can load data from the server of the local database, and they support several Entities and advanced filtering [25].
- **Assign**: used to assign values to variables in OutSystems [27].

[2]https://aws.amazon.com/relational-database/

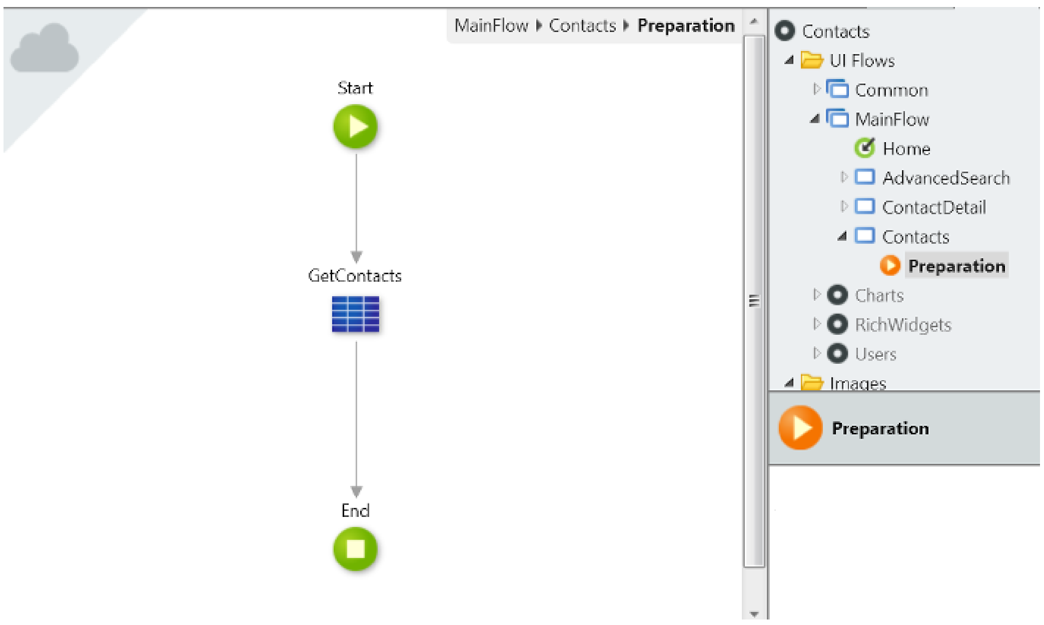


Figure 2.6: Service Studio: Nodes in an Action Flow [24].

- **If**: the user can use the If Tool to execute a branch of the action flow if the condition is evaluated as *True*, and another branch if the condition is evaluated to *False* [35].
- **For Each**: repeats the execution of an action path for each entry in the Record List [33].
- **Switch**: the Switch Tool splits the action flow into two or more paths, where the first action path whose connector evaluated as true will run [47].
- **End**: used when designing the process flow of the user process, the user must end the flow paths [30].
- **Start**: this node indicates where a flow starts executing [46].

These are just some of the components (nodes) that exist in the OutSystems platform environment. There are several others that are used by the the developers to achieve their goals when developing applications with OutSystems platform, such as *Refresh Data*, *Attach File*, *Download*, *Destination*, etc. A complete description can be found in the OutSystems platform Documentation [51].

2.2 Software Testing: Overview

Testing plays an important role on the software life cycle and it is required from the early stages of software development, such as requirements specification. Testing refers to many different approaches that intend to validate a piece of software. It is a challenging activity involving demanding tasks, such as deriving adequate suite of tests, according to a feasible and cost/effective selection technique, the ability to run tests (the environment), deciding if the tests outcome are acceptable or not, judging whether testing is enough or not, among others. These tasks provide significant challenges to developers and testers, where skills and expertise remain of high importance [3].

2.2.1 Software Testing Techniques

Testing refers to a full range of techniques that are split into two categories: *Static Techniques* and *Dynamic Techniques* [3]:

- **Static Techniques**: Static techniques are based solely on the examination of the software code, documentation or information about requirements specifications and design. This characteristic allows the static techniques to be employed at any stage during the development, despite being highly desired at the early stages. Traditional static techniques include:

  - *Software inspection*: the analysis of the software documents produced, considering a compiled checklist of common and historical defects.
  - *Software reviews*: in this process, all the aspects related to the software product are presented to managers, users, customers or other different stakeholder for approval and comment.
  - *Code reading*: the analysis of the code (on a screen) to discover typing errors not related to style or syntax.
  - *Algorithm analysis and tracing*: the analysis of employed algorithms to study their worst-case, average-case and other probabilistic analyses evaluations.

  All these techniques are error-prone, time-consuming and done manually. To minimize these issues, there are some static analyses techniques proposed by researchers relying on the use of formal methods. The goal is also to automate as much as possible the verification of the properties of the requirements and design.

- **Dynamic Techniques**: the main difference between these techniques and the static ones depend whether the code is executed or not. These techniques obtain information of the software by observing some executions. *Testing* in a literal sense is a dynamic technique, based on the execution of the code on specific input values. The amount of inputs to be tested are infinite, meaning that the testers/developers must wisely choose interesting ones that can coverage the software at most and run at a reasonable time. In other words, the programs are tested to observe some samples of the program's behavior. Therefore, tests strategies must be adopted to find a trade-off between number of inputs chosen and effort dedicated to testing purposes.

2.2.2 Test Classification

Many people classify tests into different categories based on their purposes and scope. Martin Fowler [10] classifies test into the following categories:

- **Unit Tests**: There are several definitions of unit testing. Despite the variations, there are some common elements, such as the notion that unit tests are low-level (focusing on small parts of the systems). Unit tests are nowadays written by programmers themselves using their regular tools (sometimes the difference being the framework) and they are expected to be much faster than other kinds of tests. Besides the common elements, there are also differences among the concepts, for example, what people consider to be a *unit* (Figure 2.7). Object-oriented designs usually treat a class as a unit, while functional or procedural approaches consider a single function as a unit. It is actually a matter of situation, depending on what makes sense for a given system and its testing. The unit tests can be *Sociable Tests*, meaning they interact with other units to fulfill their behavior, or *Solitary Tests*, meaning that the tested unit is isolated. The key quality of the unit tests is their **speed**, allowing the programmers to join them into a *compile suite* (a suite of unit tests that programmers run whenever they think of compiling), and a *commit suite* (a suite of unit tests that programmers run before committing new versions of software). To have a good coverage of their code, programmers must build complete test suites, but they should keep in mind that the test suites should run fast enough that they are not discouraged from running them frequently enough [12].

- **Integration Tests**: determine if independently developed software units work correctly when they are connected to each other. Unit testing should be the first one to be performed to test a module on its own. Once that is complete, we can now do integration testing to test the connections among the various modules into the entire system or even sub-systems. The point of this kind of testing is essentially to test if independent modules (developed separately) work together correctly. In integration testing, it is usual to use *Test Doubles*[3] to test interaction behavior without activate a third full component instance. If the third component is another service, then this technique brings many advantages as the external service requires its own build tools, environments, and network connections (Figure 2.8). *Contract Tests*[4] can be used to check if doubles are truly faithful [8].

- **Broad Stack Tests**: are the tests that exercise the most parts of a large application. They are also know as **end-to-end tests** or **full-stack tests** and they contrast with *Component Tests*, which only exercise a part of the system. Broad-stack tests often manipulate the entire systems, going through the UI to the lower levels of the system. On this kind of tests it is often a good idea to use test doubles, as calling

[3]https://martinfowler.com/bliki/TestDouble.html
[4]https://martinfowler.com/bliki/ContractTest.html

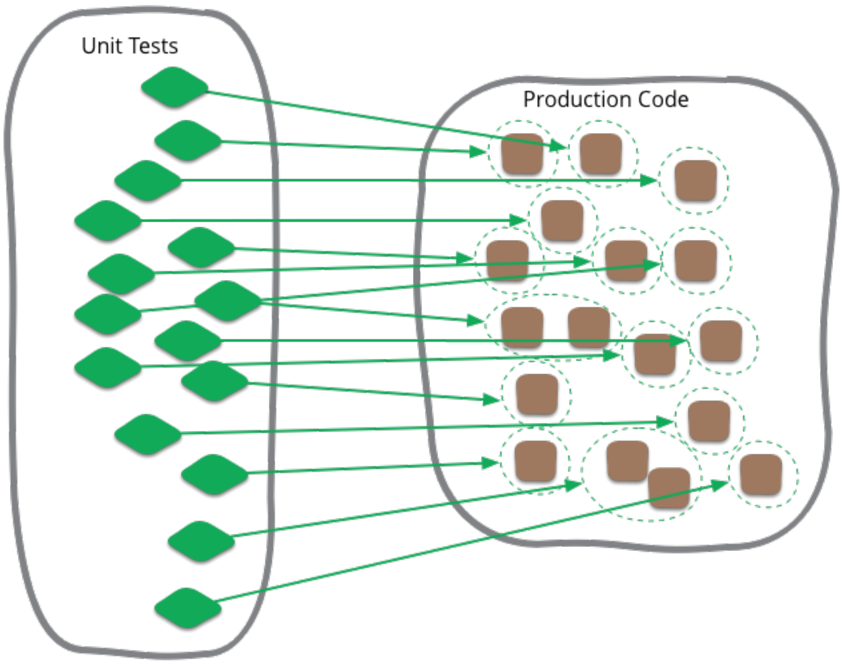


Figure 2.7: Unit Testing Sketching [12].

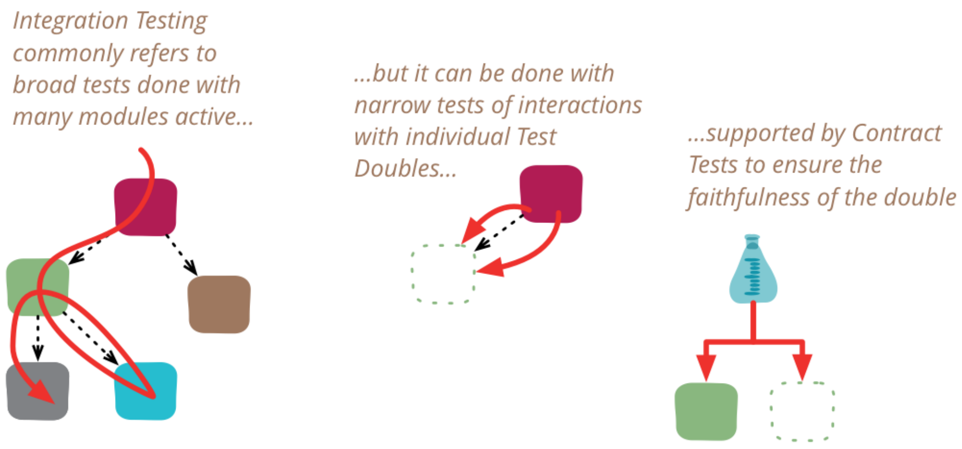

Integration Testing commonly refers to broad tests done with many modules active...
...but it can be done with narrow tests of interactions with individual Test Doubles...
...supported by Contract Tests to ensure the faithfulness of the double

Figure 2.8: Integration Testing Sketching [8].

remote systems is unnecessarily slow and brittle. Although the broad-stack tests exercise the whole system, we should use fewer of these as they are hard to maintain and much slower than component tests [5].

Listed above, there are some tests classifications that are relevant for the context of this dissertation, but there are many others, such *Story Tests*, *Contract Tests*, *Components Tests*, described by Martin Fowler [10].

2.2.3 Test Design Techniques

There are two main techniques frequently used to find errors, and the main difference between them stands on the working (source) code [22]:

- **Black-box testing**: is a testing technique employed without any knowledge of the application source code. It uses the main aspects or functionalities of the system for examination, not having any relevance with the internal logical structure of the software. The system under test will be treated as a "black box", but the tester must know the software architecture and its behavior (no source code details).

- **White-box testing**: it investigates the internal logic and the structure of the source code. This technique requires the testers to have complete knowledge of the source code, allowing them to find out a lot of implementation errors. One of the most significant advantages of this technique is to provide a maximum coverage of the system.

For large code segments, Black-box testing is much more efficient than White-box testing, however, Black-box testing provides a limited coverage of the system's behaviors and it requires clear specifications to be employed.

2.2.4 Testing in OutSystems Platform

Given the abstraction provided by visual languages like OutSystems, in association with continuous integrity validation that is built into the OutSystems platform, users will notice that the number of bugs introduced into code is much lower compared to other technologies, requiring fewer testing and fix cycles [48]. This accelerates the development and application delivery.

However, testing is still a necessary step during the application development. The OutSystems approach is to keep the OutSystems environment open so it is compatible with tools typically used such as *Ghost Inspector*, *Katalon*, *JMeter*, among others [28]. This way, testing is integrated in the continuous delivery cycle so there are no losses in productivity [34]. Another important aspect to highlight is the OutSystems *Impact analysis and self-healing*. OutSystems tracks global dependencies and pinpoints the impacts of a change on all layers of an application, assuring nothing breaks when the application goes live, even if major changes are made on the application (data model, APIs and architecture). These capabilities automatically correct problems or inform developers of any corrections they must handle. At a broader level, OutSystems does *Impact analysis* even for multi-applications, for example, preventing deployment from the test environment if it is missing a dependency in production [34]. The following tools are available to test applications in OutSystems [28]:

- **Unit Testing Framework**: Unit Testing Framework allows the user to develop and run unit tests for OutSystems platform projects. This tool can be applied in a number of ways, depending on the scale and the complexity of the applications and its architecture. Unit Testing can be particularly effective for calculation engines and for business service components. Having a good set of unit tests for the system can help greatly when the user comes to change or refactor a system [50].

- **Behavior Driven Development Framework**: the primary purpose of Behavior Driven Development framework (BDD framework) is to support Behavior-Driven Development, where all technical (e.g., developers) and non-technical (e.g., business analyst) participants in a software project collaborate to define a common understanding of how the software should behave.

  The BDD framework is a component that facilitates test automation where tests are specified using *Gherkin*[5] syntax. Gherkin is a human-readable language for structuring and describing application's expected behaviors. This scenario can be used to build BDD test automation for user's applications [53]. The main focus is testing the logic of user's modules, by exercising the critical actions that support the application's use cases. It also provides a set of tools for easily creating BDD/TDD (Test Driven Development) styles tests for OutSystems applications. This component creates test *scenarios* and *steps* that are conformant to the principles of BDD, Enables TDD, enhance test maintenance among other key-points [29]. Figure 2.9 shows an example of a Gherkin scenario:

  – ***Scenario***: describes the specific scenario that illustrates a business rule.

  – ***Given***: describes the initial context of the scenario. All the required pre-conditions that needs to hold before conducting the action or event to be tested.

  – ***When***: describes the specific action or event. In many scenarios there should only be one such step. If more is needed, the user should consider to break up the scenario into two or more.

  – ***Then***: describes the expected outcomes of conducting the action or event in the system. These steps commonly contain various assertions that verify everything we want to check as a result of this test.

[5] https://cucumber.io/docs/gherkin/

```
Scenario: Adding a product to the cart
Given:
That I have a cart
And there is a product called "Prosecco Armani DOC"
When:
I add the product to the cart
Then:
The operation should be successful
And the cart should have been correctly updated
```

Figure 2.9: Gherkin Scenario Example [53].

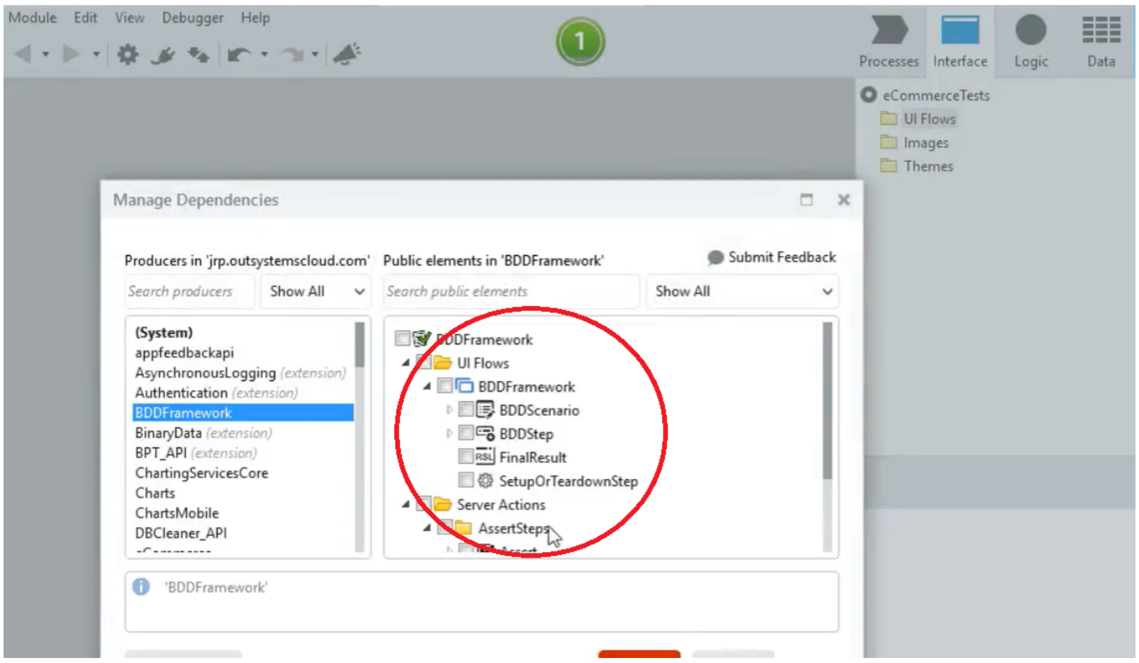


Figure 2.10: Service Studio: Web Blocks in *BDDFramework* [53].

This syntax requires the scenario to be clear to anyone who reads it, whether they are technical or non-technical participants. After the creation of the scenario, the user can start creating an automated BDD test using *BDDFramework*. The BDD testing framework includes four web blocks (Figure 2.10) that can be used to built tests [53]:

– ***BDDScenario***: each scenario is represented by a BDDScenario web block.

– ***BDDStep***: each group of steps is represented by a BDDStep web block.

– ***FinalResult***: returns stats about all scenarios run on the web screen. It should always be included at the end.

– ***SetupOrTeardownStep***: is a special kind of step that can be included in scenarios to perform setup or cleanup operations of data that is outside of the scope of the scenario from a functional or business perspective.

Finally, this framework can be used for Traditional Web and Service Applications, the server component of Reactive Web or Mobile Applications, and REST and SOAP APIs [53].

All industry testing practices can be used for testing OutSystems projects/applications, although some approaches produce better results than others. That is why OutSystems has a set of guidelines available for its users to follow:

- *Component or Unit Testing*: allows customers to independently validate small parts of an application. This is particularly effective for business service components.
- *Integration and API Testing*: focus on validating parts of an application that work together. This approach is particularly effective for complex systems.
- *Functional Testing*: is the practice of validating all components included in an application against its functional requirements, most frequently captured inside user stories[6].

Although it is very useful tool for OutSystems platform environment, BDDFramework has its own limitations, for example, it does not support the creation of tests using Reactive Web or Mobile module, it does neither support tests being run in parallel over the same OutSystems environment, among other limitations [29].

These limitations are barriers against the use of BDDFramework as a logic visualization tool, because the feedback that the developer needs must be acceptable (as little as possible), which is not the outcome of this framework. With BDDFramework we have to write the logic, then switch to BDDFramework environment, write some tests and run, and finally see the result of our logic. That is why it is not a suitable solution for the context of logic visualization.

[6]https://www.atlassian.com/agile/project-management/user-stories

3 Related Work

This chapter aims to present some works and techniques used to address the generic problem of logic previewing as well as the specific one at OutSystems development process. The goal of this section is to present some mechanisms introduced by some authors that are related to the topics of this work along with a summary of the mentioned techniques.

3.1 Logic Visualization

During the development of any application, the developers need feedback over the work that has been done. Developers aim for this feedback to be as immediate as possible, so they can visualize the effect of what they have been building and also to check if it is exactly as intended. When the subject to visualize is, for example, *graphic components* (UI components, *Widgets*), the OutSystems Platform has available a preview of the layout so the developers can check if the layout is exactly as expected.

On the other hand, the subject to visualize could be the *logic*, which is exactly what this section addresses. In order to visualize logic, most of the programmers use some approaches that are strongly related to *Software Testing*. Software Testing allows programmers and users to be confident about the software developed. Not only to be confident about the software being developed, but also to spot unexpected behavior from the software or to find out bugs, the developers use the Quality Assurance aspect of the tests.

The following subsections cover some testing methods that are used by the developers in order to obtain feedback of the logic while developing an application [10].

3.1.1 Exploratory Testing

Exploratory testing explores the characteristics of the software, raising discoveries that are classified depending on the behavior that can be considered either reasonable or a failure. Exploratory testing is a style that emphasizes a rapid cycle of learning, test design and execution, rather than trying to verify if the software conforms to a pre-written test script. Exploratory testing is the opposite of *scripted testing*, since as in scripted testing, test designers create a script of tests that are executed everytime the developing code is manipulated. These scripts can be executed by different users (not necessarily the person who wrote them) and if any test shows different behavior from what is expected, it is considered a *failure*. Before the influence of *Extreme Programming*[1], the scripted test was executed by testers, following the script and checking the result. Extreme Programming allowed the automation of the previous method, delivering a faster execution and elimination of the human error involved in evaluating expected behavior. However, even the most exhaustive automated test suite has its own limitations. Scripted testing can only verify what is in the script, referring only the conditions that are known about. It is a very good technique to spot bugs that try to go through it, but there is no guaranty that it covers all it should.

Exploratory testing emerges as a technique that seeks to test the boundaries of the scripted test suite, finding new behaviors not covered by the script. The failures found by this technique can often be added to the scripts. Although Exploratory testing is a much more informal process than the scripted testing, it still requires a very disciplined way to be done well. A very good way to do it is in time-boxed sessions, where the sessions focus on a particular aspect of the software. Exploratory testing involves trying things, learning more about what the software does and using the learning to generate questions and hypotheses, so it can be generated new tests in the moment to gather new information. This a technique that requires skilled and curious testers, who are comfortable with learning about the software and generating new test designs during the session. It also requires for the testers to be observant, and to look up to any behavior that might seem odd. It also is an activity to be done regularly during the software development [7].

There are some particular ways of Exploratory Testing that are used by developers in order to visualize logic while developing software:

- **Debugger**: is a tool that offers a closer look into the execution as it allows the programmer to work through the code line-by-line to find out what is going wrong or unexpected, where and why. With the debugger the programmer is able to interact with the flow of the execution, changing the code flow whilst running, stopping whenever needed and controlling all the execution. Some debuggers require that the code to be debugged must be compiled with the debugging information inserted. This information is provided by *debugging symbols* included by the compiler in the binaries and they describe where functions and variables are stored in memory. These symbols normally make the executable run a bit slower, but still runs as a normal program.

  Debuggers have an important feature which is the possibility to set *breakpoints* that allow the developer to stop the program execution on demand: the program runs normally until the execution is at the same point as the breakpoint address. After the breakpoint is reached, the execution drops into the debugger to look at the variables or even to continue the execution. The breakpoints can be set to

[1]`https://martinfowler.com/bliki/ExtremeProgramming.html`

beginning of functions, at specific addresses or specific lines numbers. After the execution stops as consequence from a breakpoint, the debugger can execute the next program line *stepping over* any functions calls in the line as a single instruction. On the opposite side, the debugger can *step into* a function call and execute all its code, line-by-line. There is also available in the debuggers the possibility to set *watchpoints*, which are a particular type of breakpoint that stop the execution whenever a variable changes (even if the current line does not refer to the variable explicitly). Differently from breakpoints, watchpoints look at the memory address and notify the programmer if something is written to it [1].

Errors that are made during the implementation of algorithms are reasonably easy to track with debuggers, and that is one of the key points for the use of debugger to visualize logic. Programmers use debugger to spot bugs and wrong behavior on their software which also allows them to visualize part of the software logic. Although debuggers are normally used to find out bugs in algorithms it is unusual to use them to certify that some logic is correct. Also, debugging is a process that can be very exhaustive because it requires human involvement and a large amount of time and resources [21].

- **Manual UI Testing: Real Application**: when the application is already finished or it has one of its first versions implemented, including the UI, there is something that is usually done by its developers: use the brand new application functionalities in order to look at the behavior and check if it is exactly as expected. This seems to be a very simple task, but it can actually be a good technique to verify if the product meets its goals. If done very carefully and by experienced testers it can be very useful to spot errors. Experienced testers hold a lot of knowledge acquired from past projects that can be used to test critical aspects of the applications. This technique is a kind of high level testing method that allows the developers to visualize the logic behind their application.

  However, this is a method that requires the software to be finished or having one implemented version (including the UI) in order for the developer to use the application to test the behavior. This aspect discourages the developers to adopt this method as the main technique to visualize the application logic as it requires the application to be implemented. The required time to get feedback of the logic is too big, not allowing this technique to succeed as one of the main approach.

- **Manual UI Testing: Dummy UI**: this method is very similar to the previous one. The key point of this approach is also to test the application functionalities to check if they are in conformance with the expected. Nevertheless it has a significant difference: this method does not require the application to be fully implemented, neither to have a version of the application with the UI implemented. The developer can create a *dummy* UI just for the purpose of testing the functionalities. This notably reduces the feedback time in comparison with the real application testing.

These properties bring good advantages in terms of time and productivity as it allows the developers to have an early feedback of the logic, but it is not the best solution because the feedback still takes some time. The developer needs to spend time creating a dummy component to visualize the logic/behavior of the application, and sometimes even the dummy component requires some aspects that are not related to the logic that the developer wants to visualize.

3.1.2 Logging and Monitoring

Logging is a concept that refers to the printing of messages that normally is an aid to debugging. Logging is the automatic recording of information messages or events, with the intent of monitoring the status of a program and to diagnose problems. Some systems use this approach because the failure can affect the correctness of the whole system. It is also used to evaluate products about their reliability [1].

Logging brings a concept of "*Observability*", that is, for example, the tracking of things like memory, CPU utilization, network and disk I/O, thread counts. It can also track things related to business or domain, as session duration and payment failure rate, for example. Besides, product-oriented metrics are more valuable to observe because they closely reflect that the systems is performing in conformance to its business goals [6]. Logging is a very powerful way of gathering data about a system. They are no longer just text files to look up in case something goes wrong. As mentioned, it is not restricted to technical data, it can also log valuable data. There are some tools available like *Splunk*[2] that can build indexes based on the kind of logs (*ERROR*, *WARN* and *INFO*) and offer optimised search, aggregation and visualisations. When the developers look at the log files, they should be able to filter the results to see just errors, for example, in order to ensure that nothing has gone wrong [59].

During the development of a software product, developers take advantage of this approach to visualize software logic and to identify the errors that were made. This technique, if well used, allows the access to precious and structured information in case of error or even information about the business logic. Despite very useful, Observability requires hand-rolled instrumentation logic and the instrumentation code can be very noisy, easily leading to a distracting mess [6]. These characteristics make this method an unreasonable strategy to visualize logic in OutSystems Platform.

3.1.3 Testing Support for the OutSystems Agile Platform

The Testing support for the OutSystems Agile Platform is a Master of Science dissertation work which main goal is to add automated testing capabilities to OutSystems Platform,

[2]https://www.splunk.com/

in order to make testing a reality for the majority of developers that use this platform [13]. With this new feature, the developers would feel more confident on doing heavy refactoring on their code, and it would ensure that the produced application had a high degree of quality.

The Testing support for the OutSystems Agile Platform started by introducing a series of new elements in the OutSystems language to allow the creation of tests: **Test Case Flow, Assert tool, Data Assert tool, Execute Query and Execute Advanced Query tools** [13].

This approach was created in such a way that each *ESpace*[3] acts as test suite, where it could be chosen to execute one, a selection (manually selected), or all the tests of the module. The tests are executed at the server in the user's personal area, one by one. When the execution is finished, the user receives a visual notification of the tests results in the development environment (in Service Studio).

The work also includes a feature that is the most relevant aspect for the logic previewer, the **Test Action**. The Test Action works by updating the personal area at the server, without deploying it, in order to be faster. The values of input used to test the actions are not stored in the module, but in the user's settings, to avoid updating the personal area. Test Action had to stub a few ASP.NET [19] objects because the Actions code is expected to run under a web request [13].

Although very promising, the results of this work can not be well measured as the proposed solution was not integrated into the OutSystems development environment (Service Studio).

3.2 Summary

The related works presented above, despite the fact that they are not completely applicable to the problem, represented the closest existing solutions prior to our work. The tests solutions are important approaches to ensure a good level of software correctness, along with quality assurance. However, the tests solutions are not the best approaches to preview logic, either because the approach is not perfectly suitable for the OutSystems development processes or they are not enough by themselves. In some of the testing approaches, the cycle of feedback is still way longer than the desired, and that is because most of those techniques are used for ensuring software correctness and others are performed at the end of the development process. Others are just not commonly adopted by the developers, either because of the extra time required (during development phase) or because of their complexities.

Others presented techniques include Manual UI Testing that is suitable to be performed at the end of an application development with the purpose of final testing, but not as main choice for previewing logic. It is not suitable because, as mentioned previously, it takes to much time to test the application using its UI and this also requires an

[3]a module where an application is developed. Where the data, logic, screens, among other things relevant to an application are created.

existing UI. Belonging to the same approach as the Manual UI Testing, another technique presented was the Debugger. As seen in Section 3.1.1, it is a good solution for tracking execution details and to spot bugs on algorithms, but not so applicable for previewing logic because of its excessive use of time and resources.

There are some other techniques that can be used to check and ensure applications logic correctness, but are not strongly related with logic visualization itself, such as Specification by Example [11], Test-Driven Development [2, 9, 56], Read-Eval-Print-Loop [58, 55], among others.

Several of the mechanisms presented above in this section can be used to partially preview the effects of logic in OutSystems. However, they require too much time and resources, and therefore they are not the best or the obvious option. Also, most of these related techniques that try to preview the logic in OutSystems, even the ones that get closer to the goal, do not fulfill the requirements as they are not as immediate as desired or they do not preview the main effects of the applications logic. Nevertheless, these related works and approaches are extremely important for our solution projection, as we can learn from their best aspects and their own limitations.

4 Proposed Work

This chapter presents the details of the problem to address, the methods thought to address it and also the goals of this work.

As software development increases day by day, the goal of the companies that build software products is to increase productivity as much as possible in order to respond to high demand. OutSystems is improving everyday its methods to satisfy its clients with excellent products and short delivery time, greatly due to improvement of the software development environment. VPLs accelerate the development process by reducing the concerns of the users with implementation details, consequently improving their productivity. Despite the software development process in OutSystems Platform being fast, there is an important problem that needs to be solved: *The feedback loop of logic visualization*. How can we improve the cycle of feedback received by developers when writing their logic in OutSystems Platform? This is the question that we tried to answer during the implementation of this work.

4.1 The Problem: Overview

Each amount of time that can be reduced during software development is a significant asset for productivity, even small reductions may present significant value. Therefore, all processes (during development process and not only) should be carefully and continuously analysed in order to obtain some improvements. Developers in OutSystems, when writing their code (such as OutSystems Action logic) do not have an immediate nor short-term feedback of the logic's behavior. This is the key problem to address in this work: During development, developers do not know immediately, what are the effects of the logic being written, which can significantly impact the user's productivity.

But how can logic be visualized in OutSystems? Developers in OutSystems have some approaches to visualize the behavior of the logic they are writing, which can be shown below:

- **By testing**: developers in OutSystems (and not only) use testing properties to check the behavior of applications and to be able to spot bugs introduced during development. Section 3.1 presented different strategies to visualize logic and developers use many of them in OutSystems. Starting by **Manual UI testing using real application**, which is an approach used to check the application behavior by using the finished (or a released version) of the product UI. Although with this strategy the developer is able to spot some misbehavior and have a clear notion of the functionalities usage, it is not a suitable approach to visualize logic during development, as it requires the software product to be implemented or at least have a released version. Similarly, we have the **Manual UI testing with a dummy application** which concept is almost the same as the real app testing but it does not require a finished version of the UI, hence reducing some feedback time. However, this approach also requires the implementation of a dummy UI for testing proposes, delaying the feedback time to the developer. In addition, both approaches mentioned are within Exploratory Testing and, as mentioned, it is a technique that requires high levels of experience and expertise from application developers.

Still on testing, developers in OutSystems also use testing frameworks to test their software and to check the application's behavior. **BDDFramework** and **Unit Testing Framework** (presented in Section 2.2.4) are the most used in OutSystems platform for testing applications. We saw that both are good tools within the testing context, but when it comes to visualizing the logic they are nowhere near a reasonable solution because of their previously mentioned limitations (in Section 2.2.4), such as the incompatibility of using BDDFramework with the OutSystems environment at the same time, not allowing the developer to visualize the effects of the logic being developed.

- **With the debugger**: the debugger is a very useful tool in the OutSystems platform as it allows to have closer look into the applications execution, which is used to spot misbehavior and bugs in the software. Nonetheless, debugging requires the software to be published, then a browser or a mobile device to check the execution flow. Also, as mentioned in the previous chapter, debugging is a process that requires large amount of time and resources and its not suitable for checking software correctness.

In summary, all the approaches used in OutSystems for visualizing logic described above have their particular limitation and all of them have one main problem in the context of logic visualization: **The feedback cycle to visualize logic during development is extremely long**.

4.2 The problem: Data Analysis

Throughout this section we aim to analyse the problem explained previously in Section 4.1, and retrieve information and insights that help to understand where to focus our approach and how we can solve it. With these analyses, we aim to retrieve information concerning the OutSystems applications logic, to understand which are the most used logic components in the applications, how they are composed, what are their dependencies to the database and to each other, among several other information related to the applications logic. Along with this information, we intend to gather knowledge about the applications modules, such as the distribution of logic within them, the dependencies between its logic components, among other important details. The point of these analyses is to gather the data required to understand the problem and to help us design a solution for it.

In order to obtain the intended results, we used two *datasets* of real-world Outsystems code obtained from several farms[1]. Along with these two *datasets*, we used tools like *Microsoft PowerBI* [20] so we could extract relevant information in order to clarify and refine the problem. All the information gathered by analysing these data were directly used to choose the most suitable solution that brings more value for the OutSystems customers (anyone who develops code using the OutSystems platform).

4.2.1 Datasets

The first *dataset* is called **actionflows** and holds information about several Outsystems factories, containing 5.74GB of data. The data is split into 933 compacted *JSON.gz [14]* files (one per farm), and each one has fields like: *omlKeyGUID*, *flowKey*, *flowPath*, *flowName*, *isPublic*, *inputVars*, *localVars*, *graph*, *flowKind*, among others. Each line of the *JSON.gz* represents an OutSystems action[23] and contains the information specified by the fields just mentioned above.

The second *dataset*, named **moduleDependencies**, is a larger one containing 27.7GB of data, distributed into 1054 *.jsonlines* files. Each file represents one factory and it contains information about the dependencies between all of its modules. The *.jsonlines* files have fields like: *Key*, *Name*, *Path*, *ActivationCode*, *RuntimeKind*, among others.

4.2.2 Tools and Processes

The OutSystems tool named *Query Grabber*[2] was used in order to built the *datasets* mentioned above. By creating a query (specificaly for Query Grabber) and running it in the farms, it was possible to create the *datasets*, using OutSystems real-world code.

After putting together all the needed data, we had to process it in a way that we could gather relevant information and insides related to this project. To accomplish that, we used the *Python* language [54] to process the data (*.jsonlines* files) and produce more precise (related to the project's scope) *datasets*. Regarding the first *dataset*, the process consisted in going through all the files (*.jsonlines*) and capture all the relevant data about

[1]factories that use Outsystems developing technologies.

[2]An OutSystems tool that retrieves applications information from OutSystems factories given some queries.

the actions. A *.csv* file was created to save the metrics gathered from the *dataset*. The *.csv* file had all the important information about each triple <*activationCode*, *moduleId*, *actionId*>, such as: *Name*, *Kind*, *IsPublic*, *Number of Inputs and Outputs of each type*, *Number of OutSystems Nodes of each type* and others.

The *.csv* file produced, named **actionsCount.csv**, had slightly more than 1GB and it was more suitable to obtain the desired information about the applications (about their modules, logic, among others), as this *dataset* was built including only the relevant information for this work. In addition, the structure of this new *dataset* was created to facilitate the search for information relate to applications logic, as each line represents an OutSystems Action and its related data.

Similar to the process used in the first *dataset* (**actionflows**), the second *dataset* (**moduleDependencies**) was also used as source to produce another *dataset* in the format of a *.csv* file. This *.csv* file named **moduleDeps.csv** also had the triple <*activationCode*, *moduleId*, *actionId*> followed by fields such as *actionKind*, *ImpactedModuleId*, *ImpactedActionId* and others. All the *.csv* files created using both *datasets* were built with the main goal of creating simpler and yet more precise *datasets*, mainly by adding the relevant fields for this work and simple counts.

After creating the two new *datasets* (*.csv* files), the following step was to get our intended metrics. To achieve that, we had to take advantage of a powerful tool mentioned above, the *Microsoft PowerBI*. So, the new challenge is now to load the amount of data into *PowerBI* and starting creating our first queries. After loading the **actionsCount.csv** we had to clean the data, create some queries and some new tables in order to obtain more insides about the OutSystems actions. The same process was applied to the **moduleDeps.csv**, but this time we had some limitations caused by the size of this *dataset*, that was 49.3GB. In the next section, we present some of the data and metrics that were collected during this process, that was extremely important to precisely target our solution.

4.2.3 Results

As seen in Section 2.1.2.2, Actions are logic that runs in the server and on the client device, meaning that logic in OutSystems appears in the form of **Data Action, Client Action and Server Action**[23]. As the main goal of this work is to preview the effects of logic in OutSystems, we must search where the biggest slices of logic in OutSystems applications are located. To do so, the first question that we must answer is: **what are the Actions that appear more frequently in the OutSystems applications?** By answering this question, it is possible to know where is located the greatest amount of logic on OutSystems applications and therefore what are the most used Actions during OutSystems applications development. By using our first built *dataset*, the actionsCount.csv, and with the help of Microsoft PowerBI, the desired answer was obtained, as seen in Figure 4.1.

By analysing Figure 4.1, it was clear that **Server Actions** represents the majority of the flow nodes in OutSystems Applications, 65.5% of all nodes in the *dataset*. This first

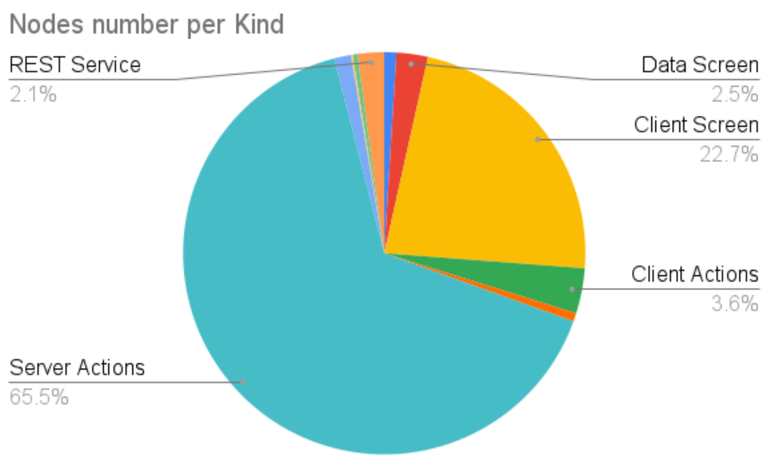


Figure 4.1: Distribution of nodes in OutSystems applications (obtained from actionsCount.csv *dataset*).

indicator was very important to refine the problem, as well as the solution. At this point, it is known that the Server Actions are the main target for the logic previewer, directing the second step of the analyses to them. After acquiring this knowledge from the data, it was necessary to extract more information about the flow nodes, but now putting the Server Actions as the base of the analyses.

Knowing that the Server Actions are the first/main target to attack the problem, the next steps of the analyses focused exactly on them. Therefore, some metrics related to the Server Actions **number of nodes, inputs, outputs**, among some others were also collected. The metrics related to inputs and outputs are very useful to address aspects related to the integration of the solution in Service Studio. Figure 4.2 shows that a solution (logic previewer) integrated in Service Studio starting with only simple data types[3] would encompass 38% of the Server Actions used on the OutSystems applications and 88% of the Server Actions if types such as *Entities* and *Entities Identifiers* were added. Finally, the Server Actions would be complete when the *Structures* and the *Lists* were added to the remaining data types. Similar to the inputs, Figure 4.3 shows how the increment would be in relation to the outputs.

Several metrics were collected to precisely highlight the problem, as well as the solution. Along with the presented data above, we also did an analysis about the complexity of the Server Actions, in order to understand the complexity-to-be of the solution. Figure 4.4 shows the results obtained around the Server Actions complexity. It shows that about 95% of the Server Actions contain less than 20 nodes, and this metric is very important to help us analyse the performance of the solution implemented in the next chapter.

The data and the metrics presented above were all gathered from the actionsCount.csv *dataset* and they had a very important role in the refinement of the problem, together

[3]Types such as Text, Integer, Email, Boolean, among others.

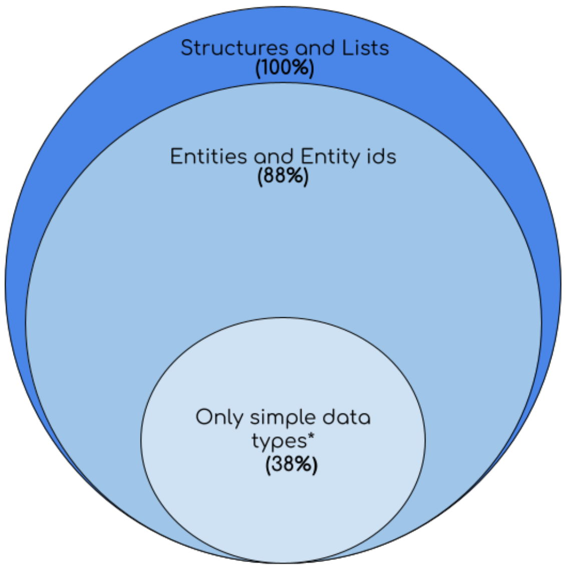


Figure 4.2: Server Actions input types (obtained from actionsCount.csv *dataset*).

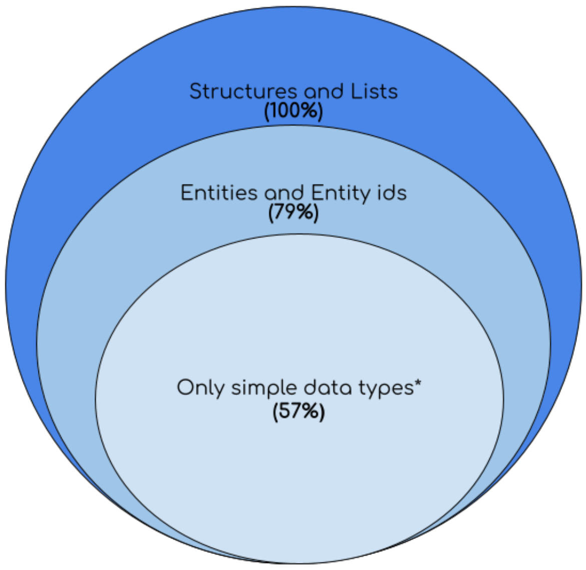


Figure 4.3: Server Actions output types (obtained from actionsCount.csv *dataset*).

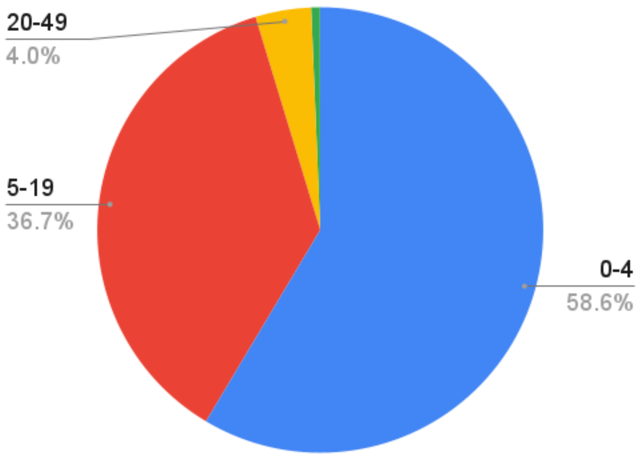


Figure 4.4: Server Actions complexity - number of nodes (obtained from actionsCount.csv *dataset*).

with the projection of the solution. Nevertheless, we needed more information about the problem, more precisely about the Server Actions. To extract more metrics from the OutSystems applications code, we used the second *dataset*, the moduleDeps.csv. At this point, it was known that the Server Actions were the holders of most of the logic present in OutSystems applications code, and now it was necessary to understand their dependencies (between modules). Figure 4.5 reveals the dependencies of the Server Actions in relation to the number of modules required to achieve their purposes. It is also clear that most of the Server Actions (65% of all Server Actions from the moduleDeps.csv *dataset*) requires just **one module** for their definitions. This metric was one of the key points of our analyses, as so far we knew that the **Server Actions** hold the majority of the logic in the OutSystems applications, and they mostly depend on **one single module** (meaning that everything they need is in the same module as they are defined).

Up to now, after the analyses that were performed and presented above, the problem had been well defined and it was known what was the main/starting point to address the problem: **Server Actions that depend only on the module itself**. Thus, after the data that was gathered from the *datasets*, we turn to the following question: **How to preview the effects of the Server Actions that depends only on the module itself?** In order to answer this question, we performed some analyses on the moduleDeps.csv *dataset*. We highlight an extremely important metric about the Server Actions Side-effects, as shown in Figure 4.6, which demonstrates that almost 60% of all Server Actions in OutSystems applications code (presented in the *datasets*) **read and/or write to the database**.

In summary, the first key point of the whole analysis was established: **Server Actions** are the main holder of logic in OutSystems applications. The second key point of this study is that most **Server Actions** depend only on the **module** they are defined in. Finally, another important aspect was about the preview of the effects of logic itself: almost

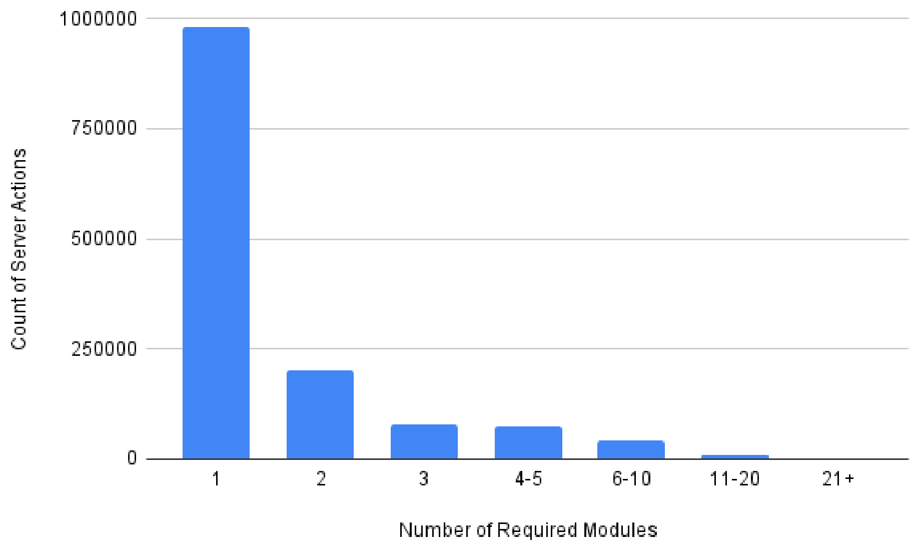


Figure 4.5: Server Actions module dependencies (obtained from moduleDeps.csv *dataset*).

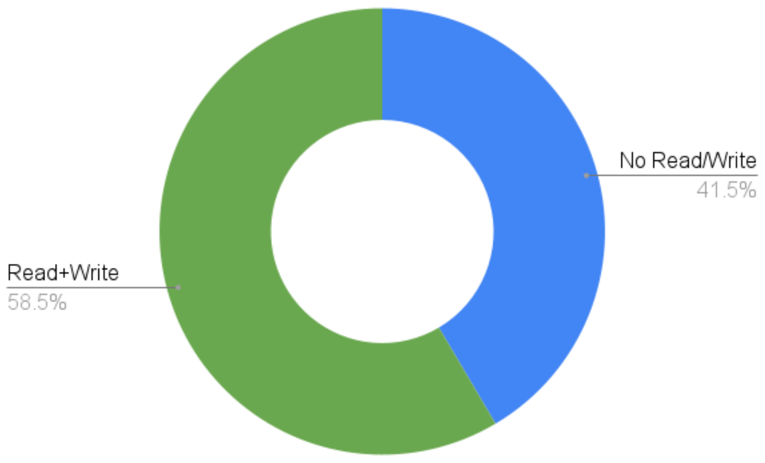


Figure 4.6: Server Actions reads and writes to the database (obtained from moduleDeps.csv *dataset*)

60% of the Server Actions **read and/or write to the application database**. The summary presented above was crucial to better understand and narrow down the problem, enabling the solution explained in the next chapter.

4.3 Proposed Approach

This section presents a description of the approach developed in this work, in order to address the problems highlighted above. This approach aims to improve the experience of OutSystems users when developing software, by allowing them to visualize part of their application behavior and reducing the feedback loop they have while creating and changing code. The process of creating software, compiling it, publishing into OutSystems server, and finally testing to check its behavior is time-consuming and hampers users productivity.

Currently in OutSystems, when users want to test an application behavior, they need to follow some approaches described previously in this document, such as testing the application directly with its UI, testing with a dummy UI, among others. In case of using testing frameworks (which is one of the fastest approaches among the existing ones), one of their limitations is that they require the code (being developed by user) and the tests to be published (compiled and deployed). Most of the time, even when using the best available testing techniques, the OutSystems developers do not have a clear visualization of the logic being written, especially when it comes to side-effects to the database.

After performing the deep analysis presented in the previous section, this work focuses on tackling what the data showed to be the main problem of previewing logic in OutSystems applications development: **preview and control the Server Actions**[4] **side-effects (reads and writes to the database)**.

Therefore, this proposed work aims to change the status quo of logic visualization, by putting together the following stages:

1. Transform the Server Actions to make them **return all of their side-effects** to the database.

2. Show to the OutSystems developers **all possible direct/indirect, reads and writes** to the database.

The first stage of the proposed work (**stage 1**) begins by tackling the question about the writes to the database performed by the Server Actions. As said before, it is very useful for the OutSystems developer to visualize the effects of the logic being developed. Enabling a preview of the applications logic, especially during implementation, would be a game changer in the OutSystems development environment as it would allow the developers to have an almost immediate feedback of the logic being written. This idea of previewing logic in OutSystems development environment (and others programming

[4]Server Actions that depend only on the module where they are defined.

environments) is usually achieved via some approaches mentioned in Section 3.1. These approaches have limitations, and even the ones that present better results are not enough for visualizing logic, especially during development and when the applications produce side-effects to database.

For that reason, at this first stage, our solution aims to address the writes to the database that are performed by all the Server Actions within a module. To accomplish that, we had to analyse all the Server Actions in the module (where the application is being built) and apply some transformations, with the main purpose of making them collect and return all their side-effects to the database. By doing that, we allow the developer to preview the side-effects to the database of his logic during the development.

In **stage 2**, after transforming the Server Actions to make them return all their side-effects to the database, we also aim to enable the OutSystems developer, not just to visualize the side-effects to the database, but also to control the possible, direct and indirect, reads and writes of each Server Action being built in the application. To do that, we intend to analyse all Server Action within the module, and for each one of them, present to the developer its possible reads/writes directly and indirectly.

All the decisions made to narrow down the problem, such as addressing the **Server Actions** and their **Reads and Writes** to the database, are strongly driven by the data analyses presented in Section 4.2.3.

5 Implementation

In this chapter, we detail the implementation of our prototype, the algorithms used to produce a prototype tool, along with the challenges faced during this work. We start by presenting an overview of the implemented solution to address the problem and highlighting the desired goals. Following this overview, we also present the implementation details, the choices that were made, and the results achieved from this implemented work.

5.1 Overview

Aiming to solve the problem explained in Section 4.1, this section gives an overview of the implemented solution. This work tackled a sub-problem of the exposed problem, driven by the knowledge gathered from the performed analysis to the real-world OutSystems applications code, as explained in Section 4.2. After narrowing down the problem (Section 4), the goal of the solution was to find a way of previewing and controlling the Server Actions side-effects (reads and writes to application database), in order to reduce the feedback cycle that developers have during OutSystems application development.

To accomplish the desired goal, we implemented a PoC tool, that receives an OutSystems program (*.oml* file) as input, creates a cloned file from the input program and transforms all its Server Actions in order to preview and control all of their side-effects (writes) to the application database. In addition, the tool tracks all the possible reads of each Server Action of the OutSystems program received as input that can be used for dependencies control and visualization. Therefore, this allows to preview all side-effects to the database (along with the reads) and anticipates the visualization of the Server Actions logic (behavior).

5.1.1 The Server Actions Input/Output Parameters

The Server Actions in the OutSystems language may contain two types of parameters: **Input Parameter** and **Output Parameter**. Along with the parameters, a Server Action may

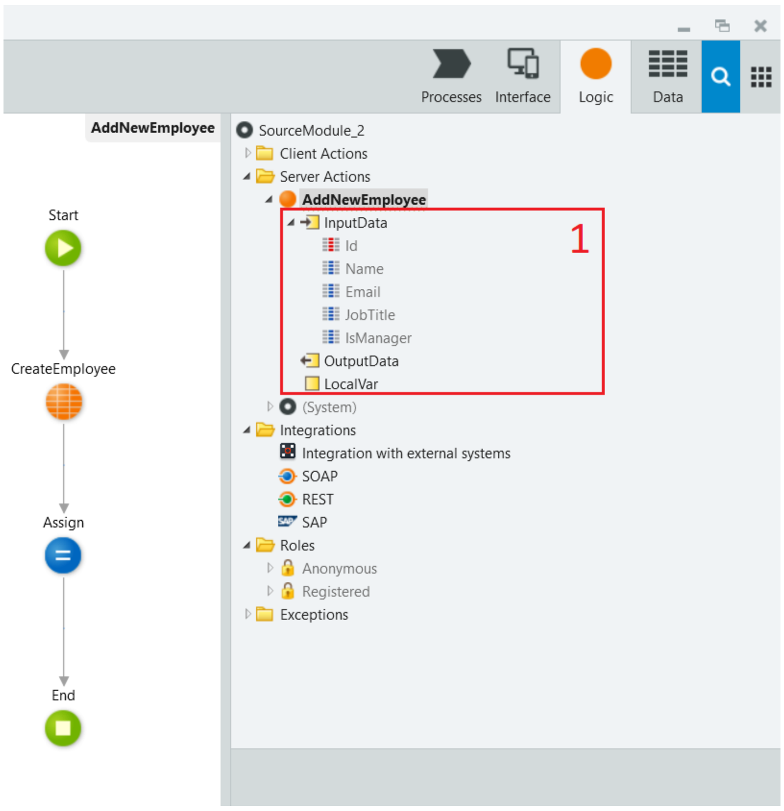


Figure 5.1: Service Studio: Example of a Server Action (*AddNewEmployee*). It inserts an *employee* (received as input) into the database (*Employee* entity), and returns the tuple Id where it was inserted.

also contain local variables. Figure 5.1 (box 1) gives an example of an action (*AddNewEmployee*) that has an input parameter (InputData), output parameter (OutputData) and a local variable (LocalVar).

Our tool takes the Server Actions of the *.oml* file received as input/source (an OutSystems program) and makes several transformations to them, with the main purpose of enabling them to return all of their side-effects to the application database. In order to accomplish that, an extra output parameter is added to each Server Action to track their side-effects. Along with the database writes, the tool produces a data structure (a file) with all entity names that each Server Action reads and/or writes directly and/or indirectly. All these transformations performed on the Server Actions allow the developers to preview their effects/side-effects during development phase and thus reducing the feedback loop they have when creating and changing them.

5.1.2 The Model API

An OutSystems model is represented in memory as an object graph whose entry-point (the main object) is an instance of class *ESpace*, and is persisted as binary XML file. A model's XML representation is a serialization of the objects graph [15].

OutSystems's costumers use Service Studio, the platform IDE, to design in a single place all the aspects of their applications, including the user interfaces, business logic, database models, and integration with external systems [16].

The Model API is an API (internal to OutSystems) used to manipulate the OutSystems Model. It consists of a set of .Net Framework DLLs that provide a low level API to read, create and change OutSystems Solutions, Applications and modules. The Model API allows the automation of several operations that would otherwise be manually performed in Service Studio and to manipulate an existing OutSystems applications code. This was one of the reasons we chose the Model API to develop our tool.

Besides, the model API is a stable approach to develop the OLP tool[1], as it works for all versions of the Outsystems product. Other approaches that could be chosen to achieve the goals of this work would involve interpretation or compilation techniques, by trying to make the compilation/interpretation of the applications logic faster that it is actually.

However, choosing an interpretation or compilation approach would have its own limitations, such as the dependency on the OutSystems product version, meaning that the solution would have to change whenever modification were made to the OutSystems interpreter/compiler.

5.1.3 Introductory Example

As explained above, the goal of this PoC tool (prototype) is to transform all Server Actions of a given OutSystems module (.oml file), in order to preview and control their side-effects[2] and possible reads to the database[3], by making them produce an extra output parameter containing all the related information (about the side-effects). Along with the extra parameter (per Server Action), a data structure is also created (per module) and it lists all Server Actions of the module, in addition with the possible reads and writes that they perform, directly and indirectly to the database.

As example, Figure 5.2 shows the result of transforming a Server Action, shown in Figure 5.1. By looking at Figure 5.2, it shows how the nodes of the original Server Action are changed (box 1) in order to capture its side-effects to the database and the addition of the extra output parameter (box 2), named *Ret_lp* (Ret is short for Return and lp is short for logic previewer), to propagate the Server Action's side-effects to the database. Figure 5.3 shows the generated data structure (JSON format) to capture the possible direct and indirect reads/writes to the database for all Server Actions present in the source module.

To better understand our solution, Figure 5.4 shows an example of the output resulting from the *AddNewEmployee* call with the input shown in box 3 from Figure 5.4, after it is transformed by our prototype. In case the execution of the *AddNewEmployee* ends with no exceptions, Figure 5.4 (box 1) shows an example of the Server Action output, while

[1]The name we refer to our prototype tool.
[2]All Server Action's side-effects performed directly on the database and calls to other Server Actions.
[3]All Server Action's possible reads performed directly on the database and calls to other Server Actions.

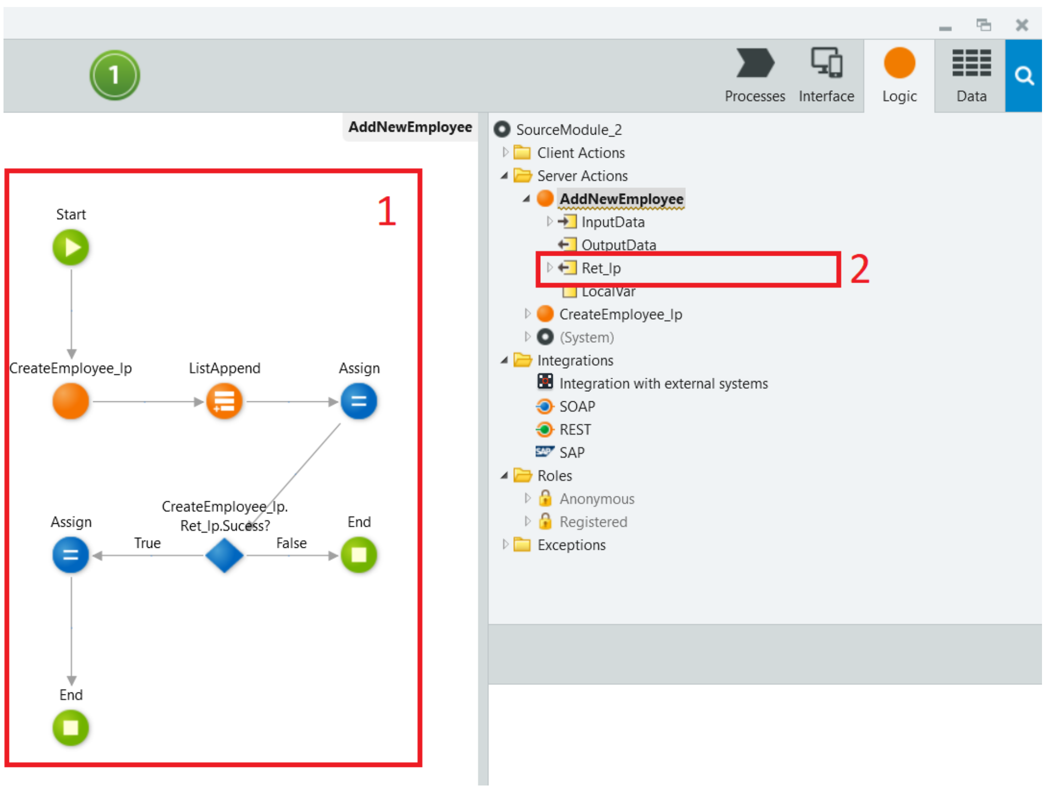
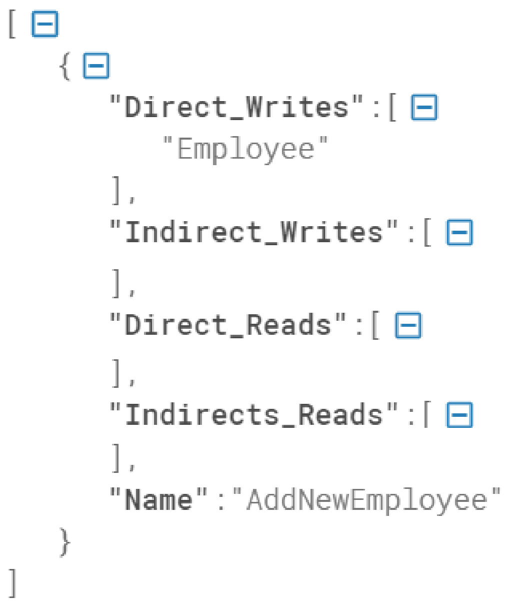


Figure 5.2: Service Studio: Example of a Server Action (*AddNewEmployee*, presented in Figure 5.1) after the transformations performed by the OLP tool.



```
[
    {
        "Direct_Writes":[
            "Employee"
        ],
        "Indirect_Writes":[
        ],
        "Direct_Reads":[
        ],
        "Indirects_Reads":[
        ],
        "Name":"AddNewEmployee"
    }
]
```

Figure 5.3: Data structure produced by the OLP tool, applied to a module containing just the *AddNewEmployee* Server Action (presented in Figure 5.1).

the box 2 shows us what would be a possible output if, for example, an exception like a Database Exception happens. A very important aspect of our solution is that, in case of exception, all the side-effects and reads to the application database that took place before the exception are returned, along with the exception message and the exception type. Notice that, if we execute the original *AddNewEmployee* with the same input as shown in Figure 5.4 (box 3), the result would **only** be the *tuple Id* where the *employee* was inserted. We now describe all the details of the implementation in the next sections of this chapter.

5.2 Implementation Details

Throughout this section, we aim to accurately describe the implementation details of the PoC tool, the OutSystems Logic Previewer (OLP).

5.2.1 Cloning the Source Module

The process starts by cloning the module (.oml file) received as input. The cloning module method (*CloneESpace*) is offered by the Model API and it creates a new module that is an exact copy of the one received. Since this operation copies everything from the source *.oml* file, this makes the operation to be one of the heaviest of the entire process[4]. All the next operations of the process are performed on the new module created by the *CloneESpace* operation.

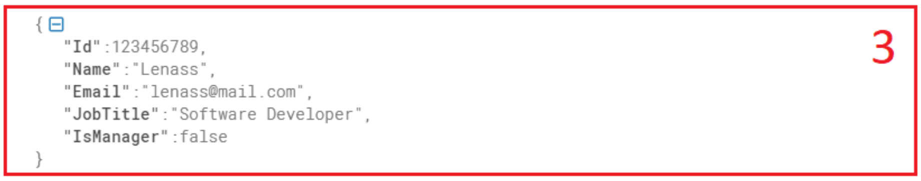
All elements in the source module are copied to the new module (including UI elements, Client Actions, etc) to enable the developer to test all the functionalities provided by the module, along with the ones provided by our tool.

5.2.2 Analysing Each Server Action

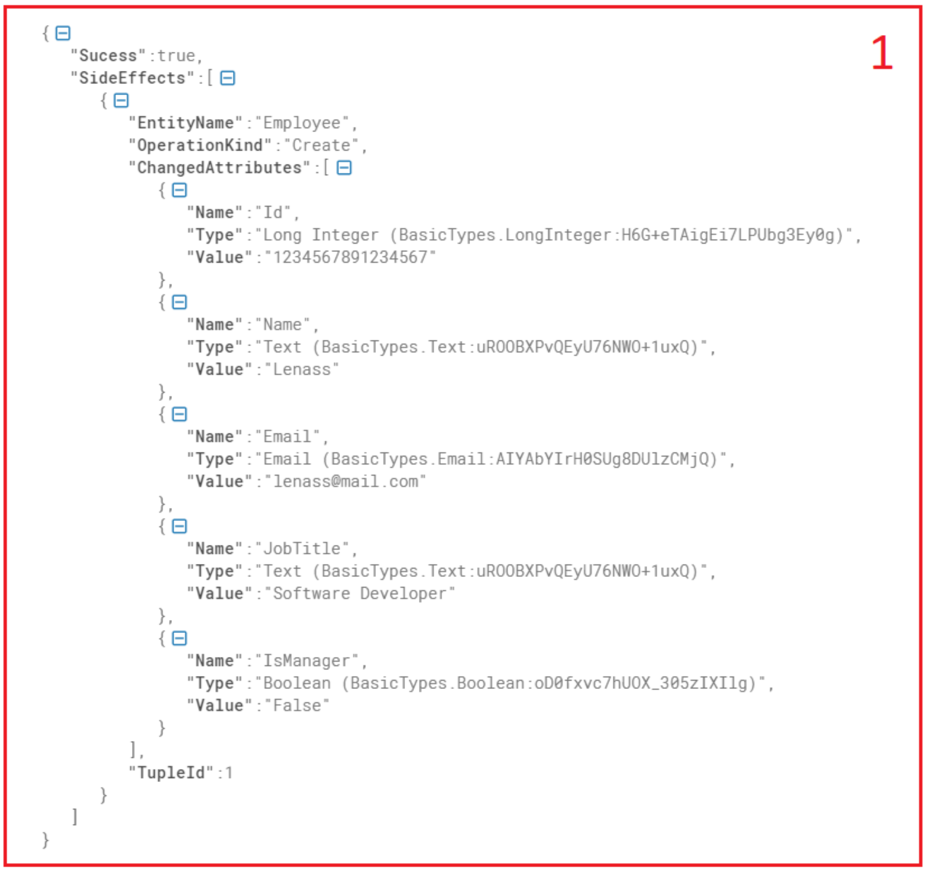
After cloning the module and thus creating the new one (a copy of the input *.oml* file), we start transforming the Server Actions of the new module (all next operations, are performed in the new created/cloned module).

Having available the new module, we start by analysing each Server Action contained in the module. For each Server Action, we first see if the Server Action has any side-effects to the application database (writes to database). To know if a Server Action writes to database, all its nodes with datatype *IExecuteServerActionNode* (see Figure 5.5) must be analysed until it is found one that writes directly or indirectly to the database. If the Server Action does not contain any of these nodes, it does not write to the database. If it is the case that the Server Action has any of these nodes, the first thing we have to ask is: is this node's Action a Server Action? In case it is, this process is applied recursively. If the node's Action is not a Server Action, then we have to see if it has an Entity Action signature. If it has an Entity Action signature, we have to confirm if the name of the

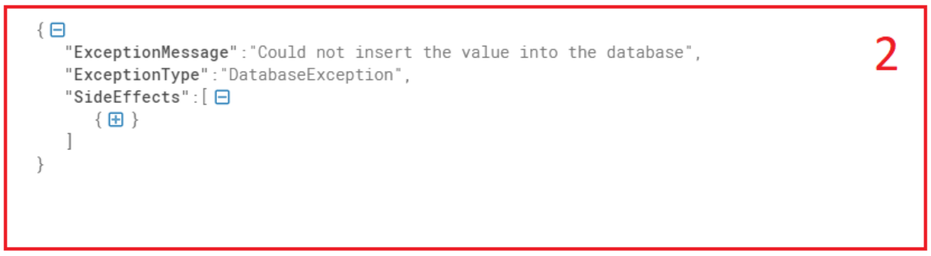
[4]Some optimizations are presented later on this work.



```
{⊟
   "Id":123456789,
   "Name":"Lenass",
   "Email":"lenass@mail.com",
   "JobTitle":"Software Developer",
   "IsManager":false
}
```

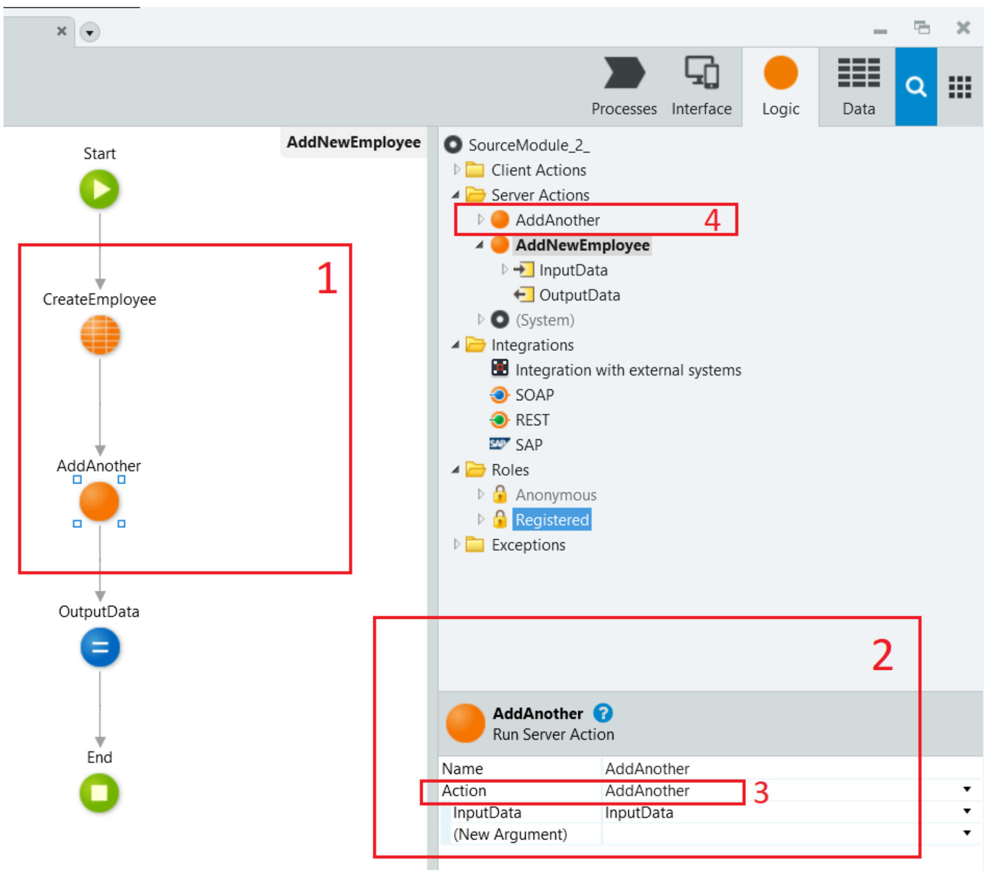


```
{⊟
   "Sucess":true,
   "SideEffects":[⊟
      {⊟
         "EntityName":"Employee",
         "OperationKind":"Create",
         "ChangedAttributes":[⊟
            {⊟
               "Name":"Id",
               "Type":"Long Integer (BasicTypes.LongInteger:H6G+eTAigEi7LPUbg3Ey0g)",
               "Value":"1234567891234567"
            },
            {⊟
               "Name":"Name",
               "Type":"Text (BasicTypes.Text:uROOBXPvQEyU76NWO+1uxQ)",
               "Value":"Lenass"
            },
            {⊟
               "Name":"Email",
               "Type":"Email (BasicTypes.Email:AIYAbYIrH0SUg8DUlzCMjQ)",
               "Value":"lenass@mail.com"
            },
            {⊟
               "Name":"JobTitle",
               "Type":"Text (BasicTypes.Text:uROOBXPvQEyU76NWO+1uxQ)",
               "Value":"Software Developer"
            },
            {⊟
               "Name":"IsManager",
               "Type":"Boolean (BasicTypes.Boolean:oD0fxvc7hUOX_305zIXIlg)",
               "Value":"False"
            }
         ],
         "TupleId":1
      }
   ]
}
```



```
{⊟
   "ExceptionMessage":"Could not insert the value into the database",
   "ExceptionType":"DatabaseException",
   "SideEffects":[⊟
      {⊞}
   ]
}
```

Figure 5.4: JSONs produced (boxes 1 and 2) by the execution of the *AddNewEmployee* Server Action (using a REST API call with the input shown in box 3) after the transformations performed by the OLP tool (presented in Figure 5.2).



Name	AddAnother
Action	AddAnother
InputData	InputData
(New Argument)	

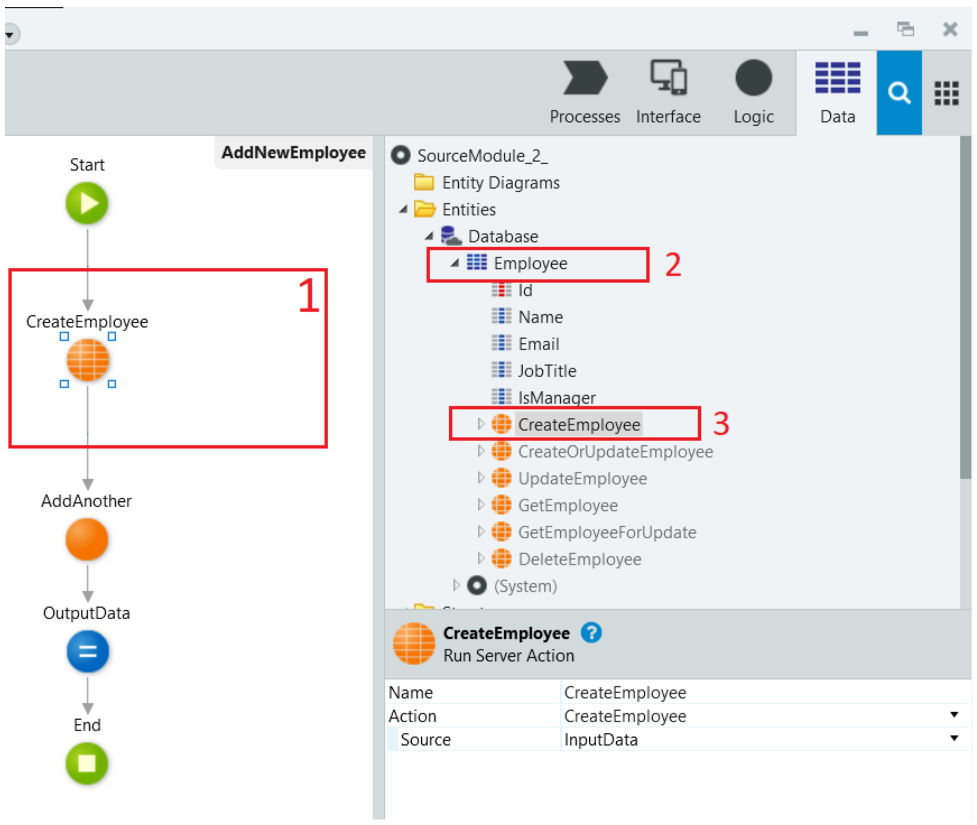
Figure 5.5: Service Studio: Example of Execute Server Action nodes (box 1). A kind of OutSystems node that executes an OutSystems Server Action contained in its property *Action* (box 3).

node's Action starts with *Delete/Create/Update*. If the node's Action does not start with any of these words or if it has not an Entity Action signature (Figure 5.6 shows an example of an Entity Action), it means the node does not write to the database. Finally, if no Server Action nodes write to the database, the Server Action does not have side-effects to the database.

Continuing the process of transforming each Server Action in the module, as seen above, we have to see if the Server Action has side-effects to the database. Our solution takes two different approaches depending on whether the Server Action writes to database or not.

5.2.3 Server Actions With Side-effects to the Database

If the Server Action performs any write to the database, we have to apply several transformations to it, in order to make it return all its direct or indirect side-effects to the database.



Name	CreateEmployee
Action	CreateEmployee
Source	InputData

Figure 5.6: Service Studio: Example of an Execute Server Action node (box 1) that has an Entity Action signature. Entity Action is a kind of OutSystems Action that is a child of (belongs to) an OutSystems Entity (box 2).

5.2.3.1 Extra Output Parameter

The process starts by adding an extra output parameter named *Ret_lp* to the Server Action, as seen before. The *Ret_lp* is an OutSystems *Structure* (is a custom datatype that can be used in a module) containing the following attributes (Figure 5.7):

- **Success** - a *Boolean* datatype attribute that returns *true* if the Server Action finished its execution without errors (Exceptions) or *false* if any Exception is raised by the Server Action.

- **ExceptionMessage** - a *Text* datatype attribute that in case any Exception is thrown, it returns the Exception message. If no Exception occurs, this attribute is empty.

- **ExceptionType** - a *Text* datatype attribute and it is used to return the Exception type (as text) in case any Exception is thrown. If no Exception occurs then it is empty.

- **SideEffects** - a *Record List* datatype attribute that is responsible for returning all side-effects to the database of the Server Action. The Record contains the following attributes:

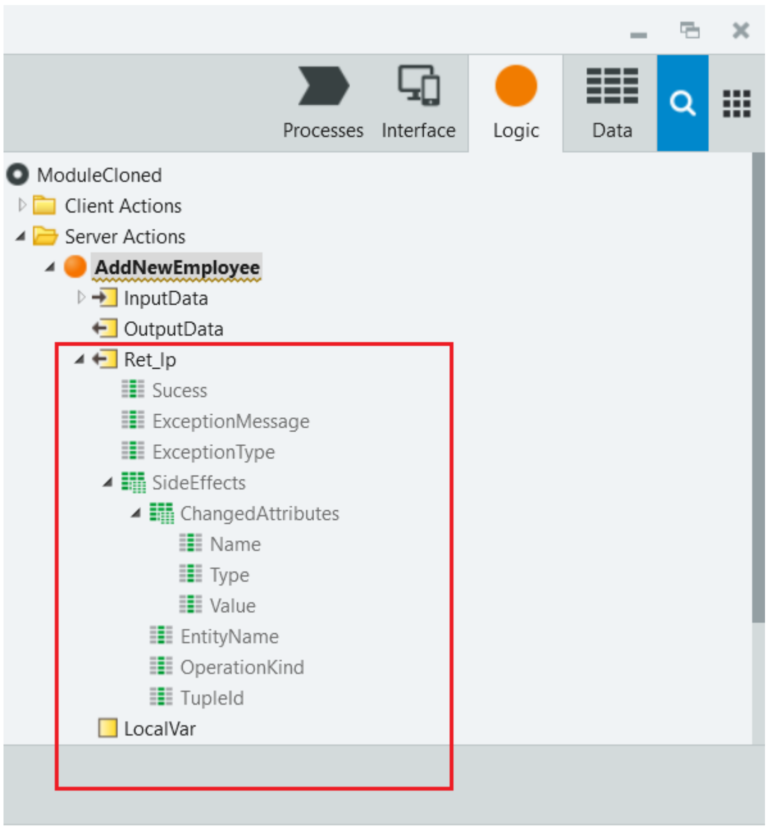


Figure 5.7: Service Studio: Example of the *Ret_lp* output parameter. This parameter has a Structure datatype and all its attributes can be seen in this image.

- **EntityName** - a *Text* datatype attribute containing the name of the entity used by the Server Action operation (that writes to database).
- **OperationKind** - a *Text* datatype attribute that returns the kind of operation performed on the database, which can be: *Create, Update or Delete*.
- **TupleId** - a *Long Integer* datatype that stores the tuple id in the entity (database) used by the Server Action operation.
- **ChangedAttributes** - a *Record List* attribute responsible for returning all the attributes changed (inserted or updated) by the Insert and Update operations to the database. For each changed attribute, this list contains:
  - **Name** - a *Text* datatype attribute containing the name of the changed attribute.
  - **Type** - a *Text* datatype attribute containing the type of the changed attribute (as text).
  - **Value** - it is also a *Text* datatype attribute that contains the value of the changed attribute (after the insertion or update).

Instead of creating the *Ret_lp* Structure everytime, we first see if it already exists in the new module, thus avoiding the repeated creation every and each time this process runs.

5.2.3.2 Execute Server Action Nodes

After adding the *Ret_lp* output parameter, we have to deal with the Server Action nodes. The first kind of nodes that this solution starts tackling are the Execute Server Action nodes. As seen in Figure 5.5, the Execute Server Action nodes are a kind of OutSystems node that execute a Server Action specified in their properties *Action*. To track all the Server Action side-effects to the database, all its Execute Server Action nodes must be properly handled. Therefore, this brings us to one of the main aspects of the prototype implementation: **the transformations of the Actions held by the Execute Server Action nodes within the Server Action**.

For each Action held by these nodes inside the Server Action being transformed, we must see if **this Action has side-effects to the database**. At this point, we already have the information about whether an Action has side-effects or not (using the algorithm explained in Section 5.2.2) and we just have to query the structure to obtain the result. Two different approaches take place, depending on whether the node writes to database or not.

Thus, if the Execute Server Action node (its Action) **has side-effects** to the database, then we have to check if the Action held by the node (in its property named *Action*) is an **Entity Action** or a **Server Action** (these are the only two types of OutSystems Actions that can simultaneously be held on the *Action* property of the Execute Server Action node and write to the database). In case the Action held by the node is a Server Action, then this algorithm is applied recursively and it returns the transformed Server Action. When this algorithm returns, the transformed Server Action becomes the new Action held on the property *Action* of the node (meaning that the node will now call the transformed Action).

On other hand, if the Action held by the node is an Entity Action, we have to **create a new Server Action** that is a *wrapper*[5] of the Entity Action.

5.2.3.3 Wrapper Creation

As seen above, if the Action held by the Execute Server Action node has side-effects to the database and it is an Entity Action, we then have to create a new Server Action in order to get and return all the side-effects that the Action performs to the database. In case of **insert** or **update** operations to the database, if it is, for example, the *Employee* entity, the Entity Actions may be: *CreateEmployee*, *UpdateEmployee*, *CreateOrUpdateEmployee*, and others which names start with *Create* or *Update*.

[5]Is the name that we call a Server Action (in this process) created for addressing the side-effects to the database of an Entity Action.

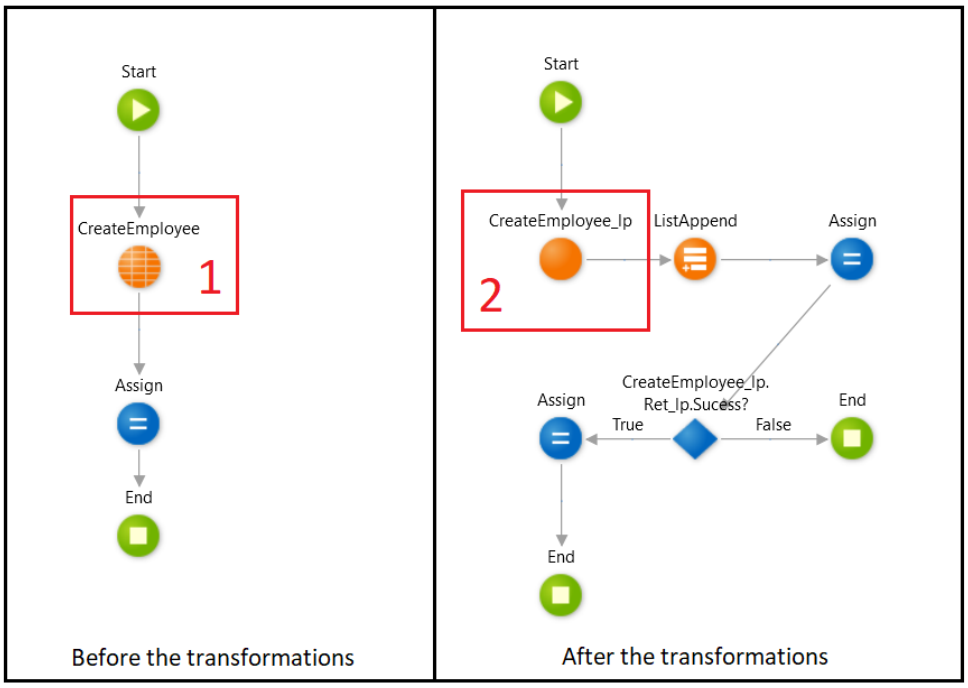


Figure 5.8: Example of the transformations performed on an Execute Server Action node by the OLP tool, by changing the Entity Action held on the node's Action (box 1) to its wrapper (box 2). The *CreateEmployee* Entity Action, receives an *employee* as input and inserts it into the database (*Employee* entity).

To better understand this transformation, Figure 5.8 presents an example of the change of the node's Action from Entity Action (box 1) to its wrapper. To create the Entity Action wrapper, we first create a Server Action with the same name as the Entity Action and add the word *_lp* to the end of the name (as also seen in Figure 5.8).

Figure 5.9 shows the created parameters, in the particular case of the example of the *CreateEmployee* Entity Action. All the existing inputs in the Entity Action are also created in the wrapper. The *CreateEmployee* receives an *employee* and adds it to the database (*Employee* entity). Therefore, the *employee* is also received as input by the wrapper (box 1). Next, the same applies to the outputs, as can also be seen in box 2 of Figure. *CreateEmployee* returns the Id where it inserted the *employee*, so, the wrapper also produce an Id as output parameter.

The next parameter is very similar to one that we discussed above in this chapter, and it has the same name, the *Ret_lp* (box 3). It has the same goals as the one used in the transformed Server Action, but instead of having a Record list (*SideEffects*) to return the side-effects to the database, it has a single Record also named *SideEffects*. This is because an Entity Action can only write to a single entity. The final one is a local variable named *ChangeSingleAttr* (box 4) that is used to address the aspects of the changed attributes in the insert and update operations.

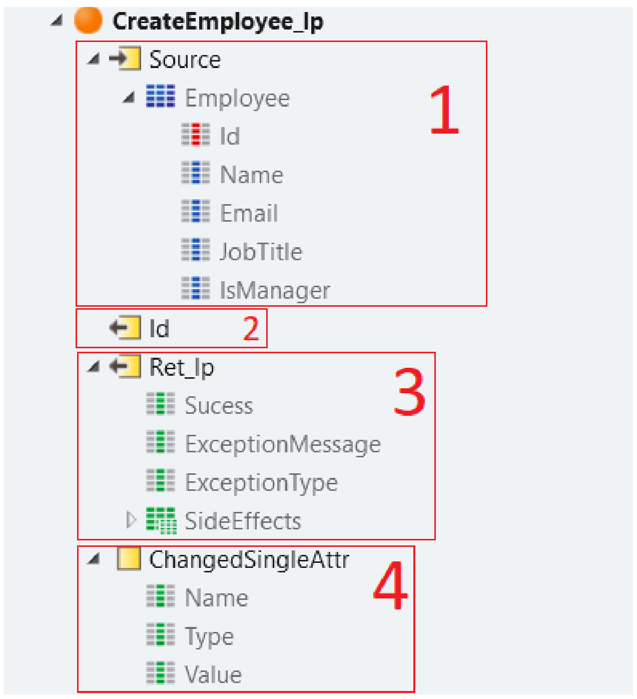


Figure 5.9: Service Studio: The parameters of the *CreateEmployee* wrapper (*CreateEmployee_lp*).

5.2.3.4 Wrapper Logic

Once the wrapper and all its parameters are created, we have to build its logic by adding the necessary nodes. Figure 5.10 shows an example of how the content of the wrapper is created for the Entity Action *CreateEmployee*. First, a *Start* node is added to the wrapper to initialize the process as shown in box 1 of Figure 5.10. Connected to the *Start* node, an Execute Server Action node is also created. The property *Action* of the node is set to the same Entity Action being wrapped (the Entity Action that the wrapper is being built for) as shown in box 2 of Figure 5.10. Then, connected to the Execute Server Action node, an *Assign* node (box 3) is created to assign:

- The outputs of the Entity Action to the outputs of the wrapper.
- The value of *Success* (within the *Ret_lp* parameter) to *True*.
- The *EntityName* to the name of the entity being written. (*Employee*, in the particular example shown in Figure 5.10)
- The *OperationKind* to the kind of the operation being performed on the Entity Action (*Create*, also in the particular example shown in Figure 5.10).
- The *TupleId* to the Id returned by the Entity Action (shown in box 2 of Figure 5.10).

After the Creation of the *Assign* node above, this process might add more nodes to the wrapper, depending on the operation being performed by the wrapped Entity Action. In

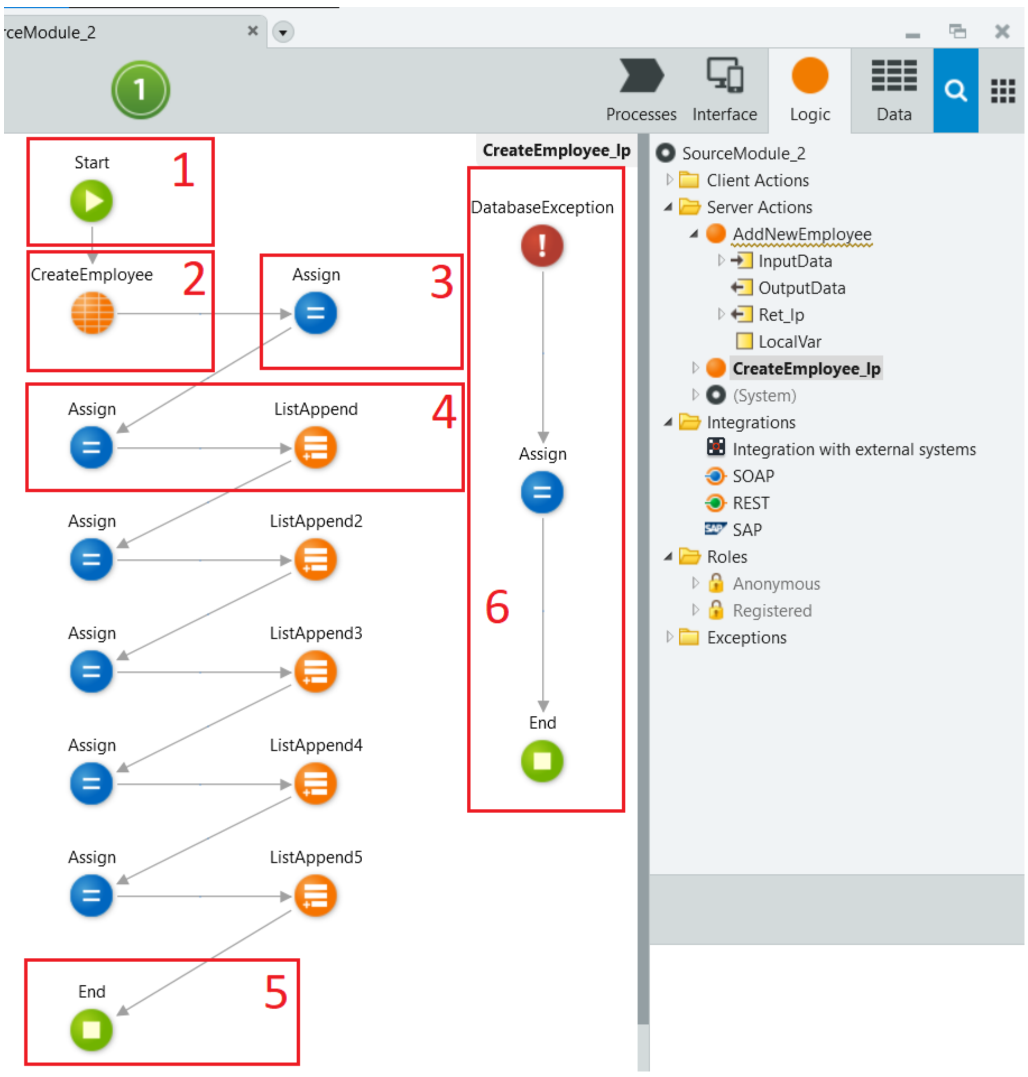


Figure 5.10: Service Studio: The wrapper (*CreateEmployee_lp*) for the Entity Action *CreateEmployee*.

case of a *Delete* operation, there is no need of adding more nodes to the wrapper but the *End* node (see example in box 5 of Figure 5.10).

Thus, in case of the example presented in Figure 5.10, if it was the *DeleteEmployee* Entity Action instead of the one shown in box 2, then this process of creating a wrapper for the Entity Action would be finished (by connecting the node in box 3 to the one in box 5 and not adding the nodes between these two boxes). For better understanding, Figure 5.11 shows a wrapper for the *DeleteEmployee* Entity Action (box 2).

Nonetheless, if the operation performed by the Entity Action is an insert or an Update, that are some aspects related to the inputs to address. As said before, in case of the insert and update operations, we have to return the changed attributes resulting from the operation performed by the Entity Action to the database. That is why, in these cases, we also have to assign to the *ChangedAttributes Record List* (in *Ret_lp.Side-effects*), the

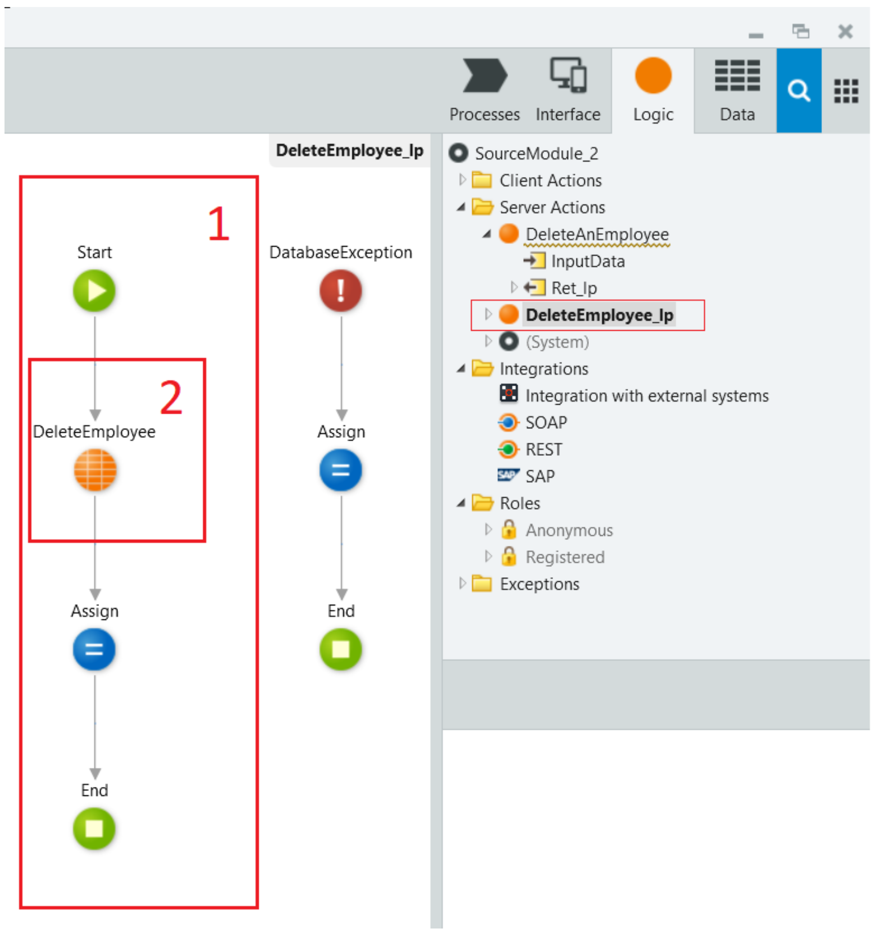


Figure 5.11: Service Studio: The wrapper for the *DeleteEmployee* Entity Action (box 2). The *DeleteEmployee* receives an *employee Id* and deletes it from the *Employee* entity.

properties of each attribute belonging to the input of the Entity Action being wrapped. For each attribute, the following properties must be assigned:

- The name of the attribute.
- The runtime type of the attribute.
- The value of the attribute (received as input).

If we see the example shown in Figure 5.10, we notice that there are five pairs of <*Assign*, *ListAppend*> (the first one in box 4 of the figure). Each pair represents the addition of the properties of each attribute to the *ChangedAttributes Record List* (five pairs for five attributes: box 1 of Figure 5.1 presents the attributes).

Therefore, this concludes the process of transforming an Action held by an Execute Server Action node. This process, is applied to each Execute Server Action node within the Server Action being processed. In the whole process of transforming each Server Action of the module, we are at the stage where we already treated all the Execute Server Action nodes of the Server Action (being transformed).

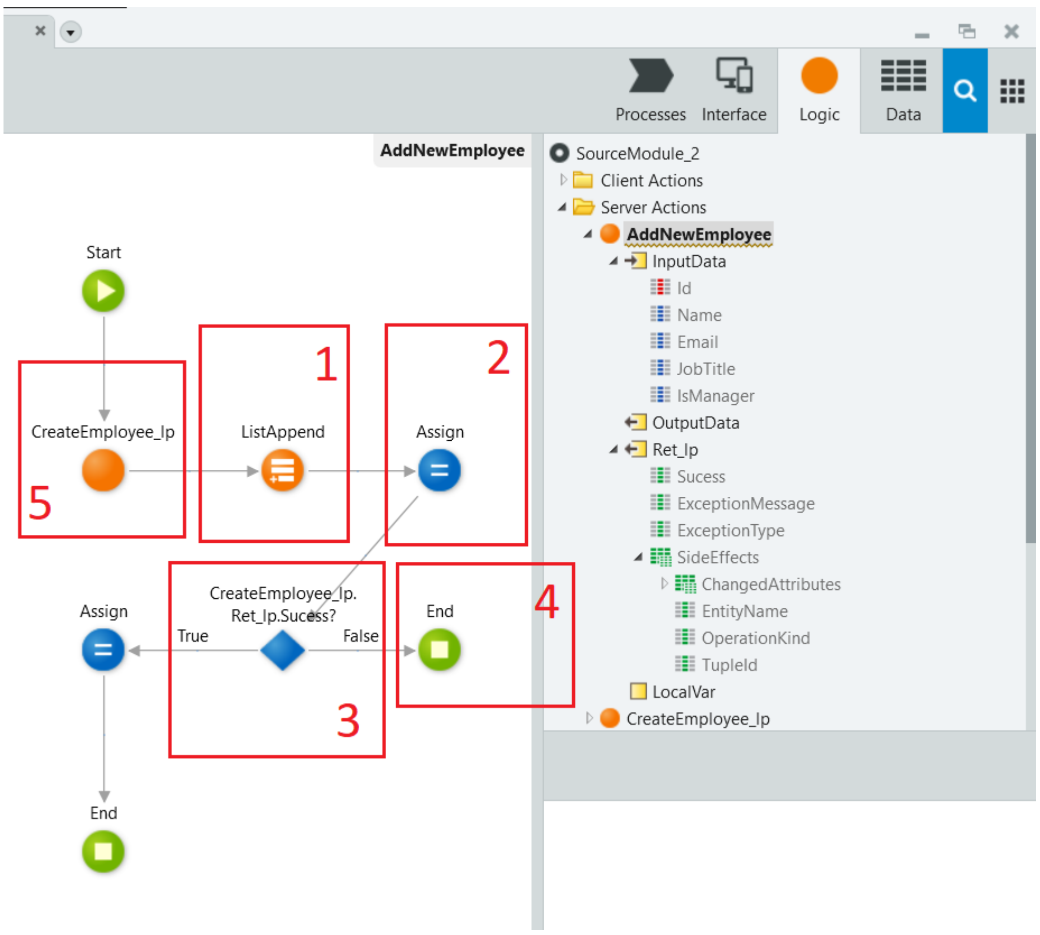


Figure 5.12: Service Studio: Example of a Server Action (*AddNewEmployee*, presented in Figure 5.1) after the transformations performed by the OLP tool.

5.2.3.5 Adding Extra Nodes

Continuing the process, when the algorithm that deals with the Execute Server Action nodes is applied, for each node it returns its transformed one. In case the Action held by the Execute Server Action node is a Server Action, then the whole algorithm is applied recursively and it returns the transformed Server Action. If it is an Entity Action, the algorithm returns its wrapper. The property *Action* of each Execute Server Action node within the Server Action being processed is set to its transformed one (an example is shown in box 5 of Figure 5.12, where the Action is changed from the *CreateEmployee* Entity Action to its wrapper). To propagate the side-effects to the database of each transformed Action (Inside the Execute Server Action nodes), this solution makes other additional changes to the original Server Action logic as seen in the example shown in Figure 5.12.

After changing the Action of each Execute Server Action node (the ones that have side-effects to the database) to the corresponding transformed one, an extra Execute Server Action node is created and connected to each node (box 1). The Action of this new node depends on the new transformed Action held by the previous Execute Server Action node.

If the new Action executed by the Execute Server Action node (after the transformations) is a wrapper (meaning that the previous Action was an Entity Action) then the Action of this new Execute Server Action node is a *ListAppend*[6]. It is used by this process to add the side-effects (a *Record*) received from the transformed Entity Action (an example in boxes 5 and 1, Figure 5.12) to the list of side-effects of the current Server Action being transformed. Now, if the new Action (after the transformations) is a transformed Server Action, then the Action of the new Execute Server Action is a *ListAppendAll*[7]. It is used to add all side-effects (a *Record List*) received from the transformed Action to the Server Action being transformed (see box 1 in the example shown in Figure 5.13).

After transforming the Actions of the Execute Server Action nodes that has side-effects to the database and adding the received side-effects from their executions to the Server Action being processed, the next step is to assign the information about the success of each executed Action to the Server Action (see example in box 2 of Figure 5.12).

Next, the process creates an *If* node to check the success of the executed Action (box 3 of Figure 5.12). If the executed Action successfully finished, then the flow continues as is in the original logic of the Server Action. If an Exception occurred, then we terminate the execution by adding an *End* node. All side-effects to the databases performed before the Exception are returned. The Server Action also returns the Exception message and type, in case it occurred in the executed Action.

5.2.3.6 Handling Exceptions in the Wrappers

One of the key points of this solution is that, in case of Exceptions during the execution of the transformed Server Actions, the execution is not interrupted with an error, and all side-effects performed on the application database until the Exception is thrown are returned.

The approach we followed to handle Exceptions in Server Actions is heavily inspired by functional error handling, more precisely the *Either* type. *Either* captures details about the outcome that has taken place, in the context of an operation with two possible outcomes. By convention, the two possible outcomes are indicated as *Left* and *Right*. The most common use of *Either* is to represent the outcome of an operation that may fail, in which case *Left* is used to indicate failure and *Right* to indicate success [4].

By looking at box 3 of Figure 5.12, we notice that the value of *Success* is being checked. If it is true (*Right* in the *Either* type) then the flow continues normally and if it is false (*Left* in the *Either* type) the flow is ended. The values associated with the possible outcomes of the *CreateEmployee_lp* are received by the *AddNewEmployee* Server Action in box 2 of the same figure. To accomplish this, some transformations must be applied to the logic of the Server Actions (only the ones that write to the database).

Starting by the wrapper for the Entity Actions, Figure 5.14 shows an example of the

[6]An OutSystems Server Action that is used to add an element to a list of that element type.

[7]An OutSystems Server Action that is used to add all elements from a source list to a target list with

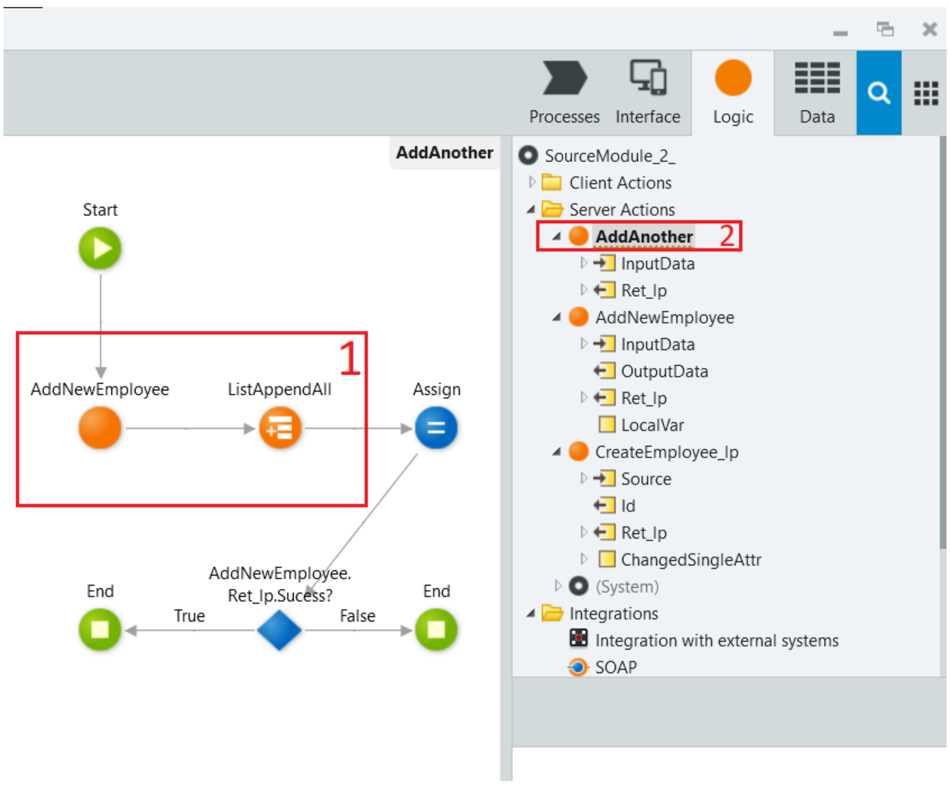


Figure 5.13: Service Studio: Example of the *ListAppendAll* Action to add all side-effects received from the execution of the *AddNewEmploye* Server Action (presented in Figure 5.1) to a transformed Action (*AddAnother*) by the OLP tool.

created Exception flow for all the wrappers. The flow consists of three nodes:

- The *ExceptionHandler* node - the **Database Exception**: this node represents the handler for the Database Exception. In the wrappers we assume that the relevant Exception that may be thrown is the Database Exception, because there is always an Entity Action that is executed. If something goes wrong during the execution of the Entity Action, the Exception is thrown and this node will certainly catch it.

- The **Assign** node: If a Database Exception is thrown, then this node is used to assign:

  – The value of *Success* to *false*.

  – The value of *ExceptionType* to *Database Exception*.

  – The value of *ExceptionMessage* to the message returned by the *ExceptionHandler* node.

- The **End** node - this node terminates the flow.

same datatype.

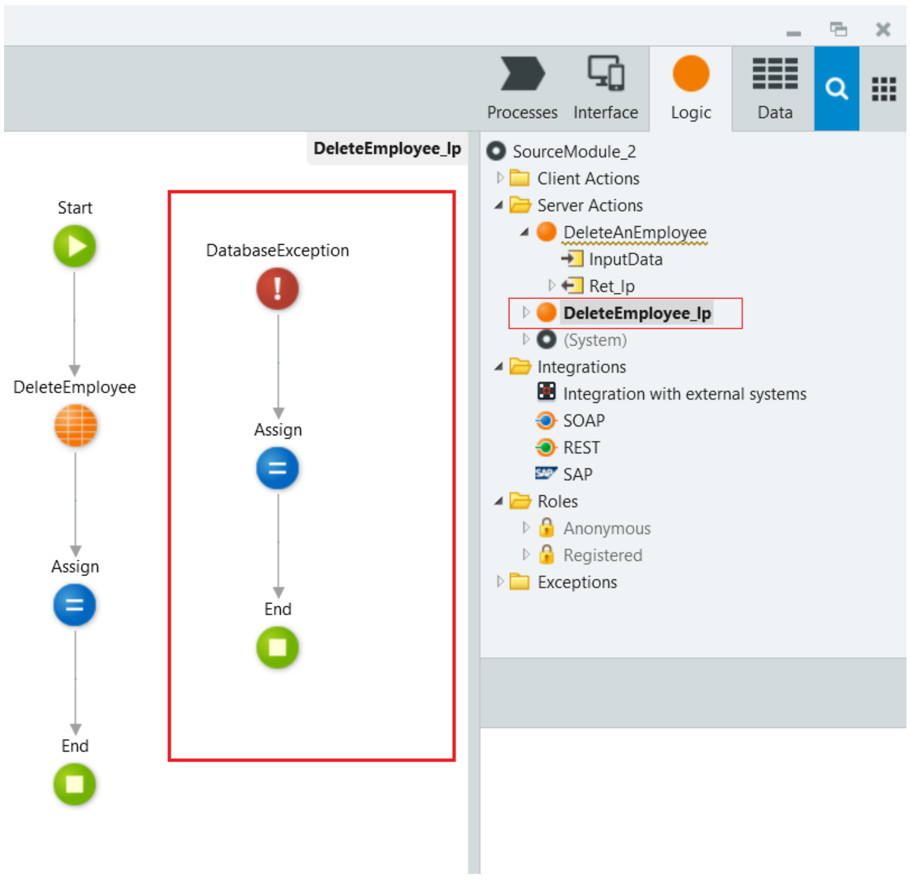


Figure 5.14: Service Studio: Exception flow inside the wrapper of the *DeleteEmployee* Entity Action.

5.2.3.7 Handling Exceptions in Server Actions

To address the aspects related to Exceptions during the execution of Server Actions, we need to consider two node kinds: the *Exception Handler* node and the *Raise Exception* node[8]. For each Raise Exception node in the Server Action being processed, this process searches if there is any Exception Handler node with the same Exception (meaning there is a *Catch* for the Exception that might be thrown by the Raise Exception node):

- If there is an Exception Handler node for the same Exception as the one thrown by the Raise Exception node, then our solution changes the existing Exception flow. An *Assign* node is added to the flow in order to make the *Ret_lp* return the aspects related to the thrown Exception (Exception name and type). Figure 5.15 shows an example of the transformations of a Server Action, where it can be seen the changes performed to its Exception flow (box 3).

- If there is no Exception Handler node for the exact same Exception, our solution searches for a parent Exception. In case an Exception Handler for a parent Exception

[8]An OutSystems node that throws an Exception held in its property named *Exception*.

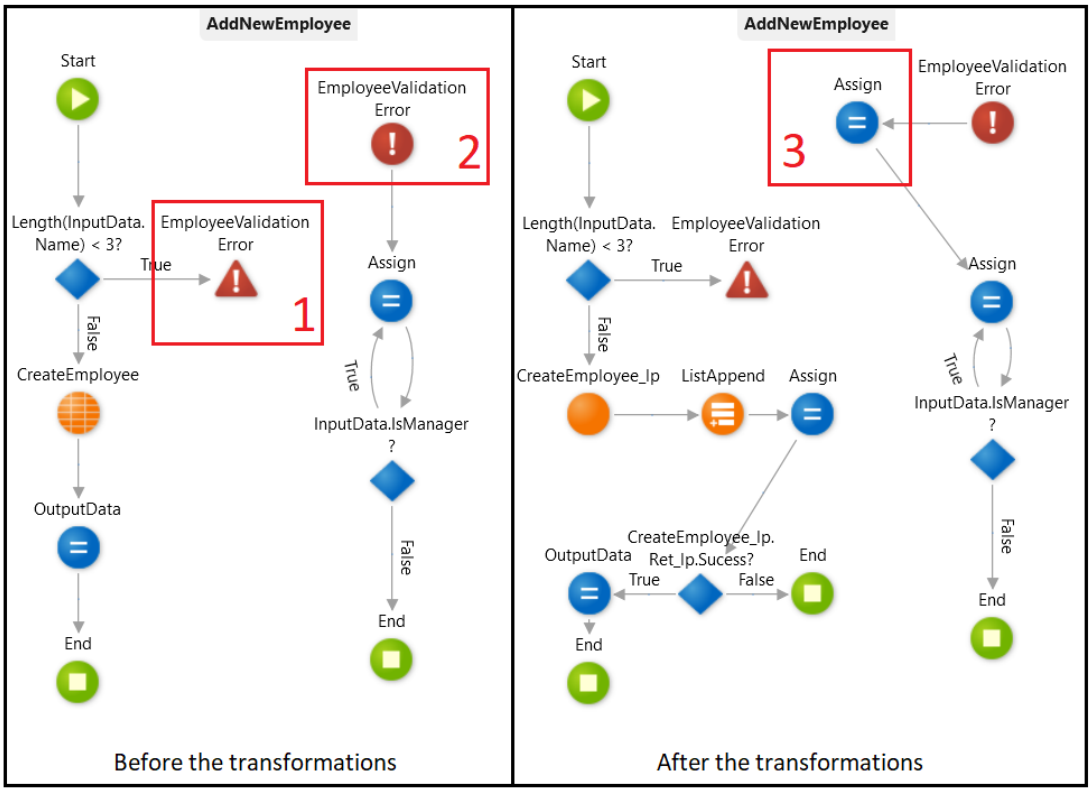


Figure 5.15: Example of changing an existing Exception flow (box 3) in the *AddNewEmployee* Server Action after the transformations performed by the OLP tool. In this example, the *AddNewEmployee* (before the transformations) does not just throw a User Exception (box 1), but also handles the Exception (box 2).

(of the Exception held by the Raise Exception node) exists, the algorithm explained above is applied. For example, in Figure 5.15, if the Exception handler in box 2 did not exist and if there was an Exception Handler for a User Exception, then the flow started by the User Exception would be changed following the algorithm explained above. This happens because the Exception held by the Exception Handler in box 2 is a child of User Exception.

- If there is neither an Exception Handler node for the exact same Exception nor a parent Exception Handler, then our solution creates a new Exception flow. This flow starts with an Exception Handler node for the raised Exception, followed by an *Assign* node to make the *Ret_lp* return the details of the Exception. Finally, an *End* node is added to terminate the process, as shown in Figure 5.16.
- If there is a Database Exception on the Server Action being transformed, then a different approach is followed. Since Entity Actions inside Server Actions are substituted by their wrappers, if there is a Database Exception inside the Server Action being transformed, all the wrappers of this Server Action will raise, in case of Exception, a specific Exception named *DatabaseException_lp* and the existing Database

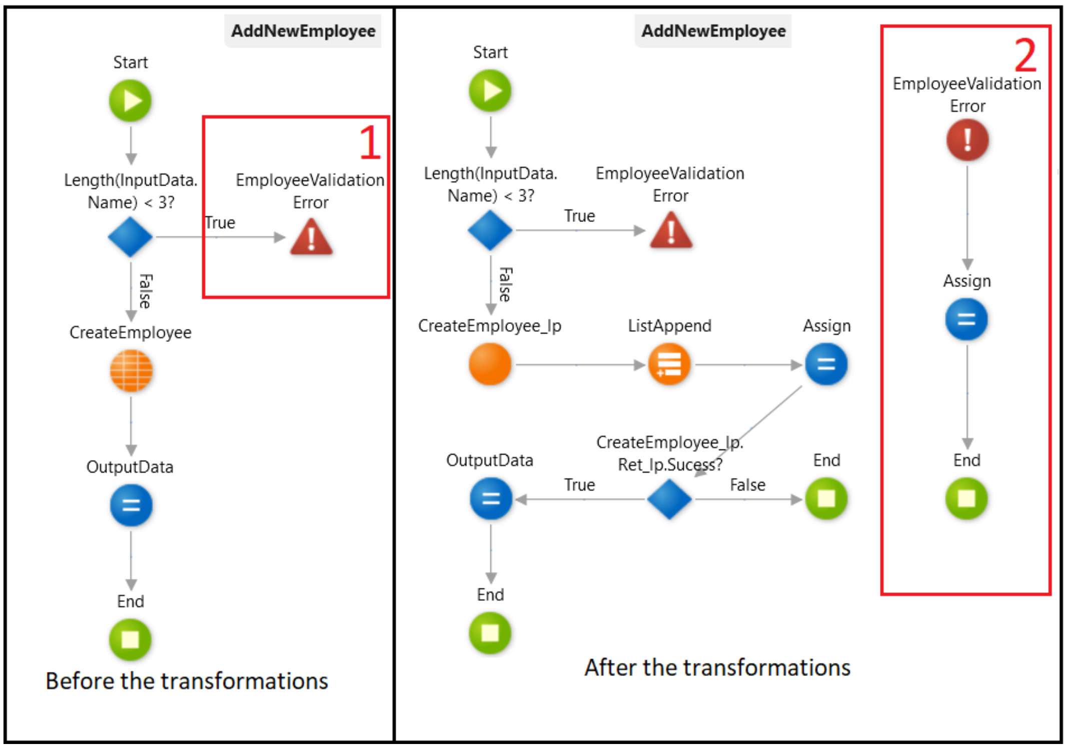


Figure 5.16: Example of creating a new Exception flow (box 2) in the *AddNewEmployee* Server Action after the transformations performed by the OLP tool. In this example, the *AddNewEmployee* (before the transformations) throws a User Exception (*EmployeeValidationError*) if the name of the received *employee* has less than 3 characters.

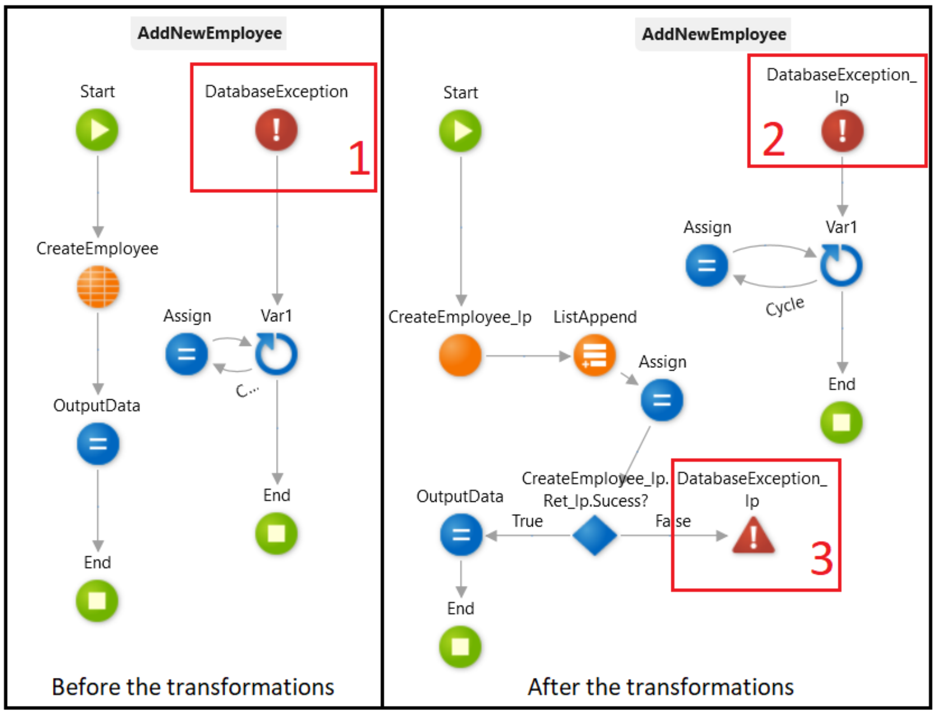


Figure 5.17: Example of changing an existing Database Exception flow in the *AddNewEmployee* Server Action after the transformations performed by the OLP tool.

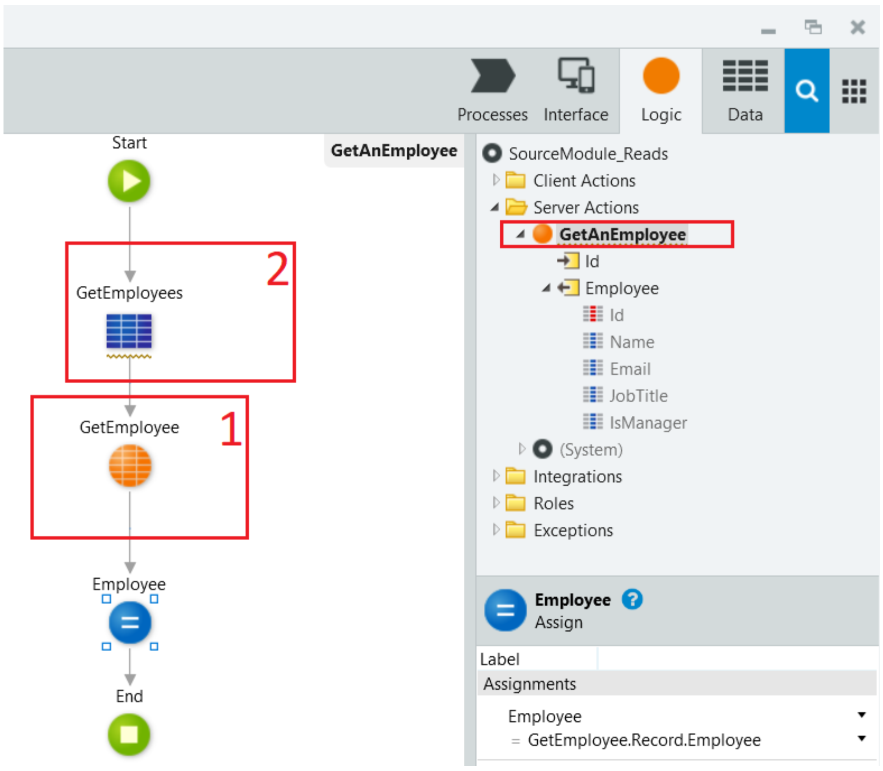


Figure 5.18: Service Studio: Example of an Entity Action (*GetEmployee*) that receives an *employee* identifier and fetches the *employee* from the database (*Employee* entity).

Exception will also be changed to the *DatabaseException_lp*. Basically, this means that the Database Exception is propagated to the wrappers. Figure 5.17 shows an example of the transformations made to an existing Database Exception inside a Server Action.

- *AllException* Handler node: If there is a Handler node for general Exceptions, then our solution modifies the Exception flow (starting by the *AllException* node) in order to propagate the side-effects to the database and the properties related to the thrown Exception (same approach as the first point presented above).

5.2.4 Server Actions Without Side-effects to the Database

If it is the case that the Server Action does not write to the database, then our process analyses its nodes to check about their possible direct and indirect reads. The algorithm to address these nodes (explained below), is also applied to the nodes that do not have side-effects, but are inside the Server Actions with side-effects to the database.

5.2.4.1 Execute Server Action Nodes

The process starts by analysing each Execute Server Action nodes within the Server Action. For each Action held by each node that does not has side-effects to the database, we must see if it is a Server Action or an Entity Action:

- In case it is a Server Action: This algorithm is applied recursively and all reads performed by the executed/called (Server) Action are added to the indirect reads of the Server Action being transformed by the OLP tool (are added to the JSON file).
- If it is an Entity Action: we see if the name of the Entity Action starts with *Get*, and if it does, we add the name of the entity (parent of the Entity Action) to the direct reads of the Server Action (see example in box 1 of Figure 5.18).

The same algorithm explained above for the reads is also applied to set the possible direct and indirect writes to the database for each Server Action.

5.2.4.2 Aggregate Nodes

Finally, to conclude the whole process of transforming the Server Actions in the module, we have to address *Aggregate* nodes. Figure 5.18 shows an example of an Aggregate node (box 2) that fetches some data from the *Employee* entity. Our solution searches for all the Aggregate nodes within the Server Actions, and in case any is found, the name of the entity being used by the node is also added to the direct reads of the Server Action.

5.2.5 Possible Reads and Writes

Besides the transformations of the Server Actions and their nodes, a JSON file is created to list all Server Actions in the original module and their direct and indirect reads/writes that might be performed to the database. The file is a JSON list containing JSON objects[9] with the following keys (see example in Figure 5.3):

- **Name**: the name of the Server Action.
- **Direct_Writes**: A list of entity names that the Server Action can write directly (the entities that are parents of the Entity Actions within the Server Action). If there is no Entity Action (that writes to database) in the Server Action, then this list is empty.
- **Indirect_Writes**: A list of entity names that the Server Action can write indirectly (entities written by other Server Actions that are executed inside the Server Action). If the Server Action does not execute/call any other Server Action (that writes to database), then this list is empty.
- **Direct_Reads**: A list of entity names that the Server Action can read directly.
- **Indirect_Reads**: A list of entity names that the Server Action can read indirectly.

[9]One JSON object per Server Action.

During the transformations of each Server Action, our solution adds to the JSON file, the name of the Server Action and its possible direct and indirect writes. This information within the JSON file is very important for the OutSystems developer to understand the dependencies between each Server Action and the database Entities. It also allows a static analysis of the possible reads and writes to the database that can be very useful when previewing and testing the Server Actions logic.

This concludes the implementation of the prototype for our solution, the OLP tool. It is very important to mention that the whole process deals with the relevant nodes just once (one iteration). Some data structures such as Dictionaries and Hash Sets were used, so each node is processed just once.

5.3 Example of Transformations Performed by the OLP Tool

To consolidate all aspects of the process explained in Section 5.2, in this section we present a simple example of the transformation of three Server Actions within a module by our OLP tool. The only entity in the module database is the *Employee* entity that has the following attributes:

- **Id**: a Long Integer attribute that is the entity primary key.
- **Name**: a *Text* attribute that contains the name of the employee.
- **Email**: a *Text* attribute to store the employee email.
- **JobTitle**: a *Text* attribute containing the job title of the employee.
- **IsManager** a Boolean attribute to store if the employee is a manager (true) or not (false).
- **CreatedAt**: a *Time* attribute to save the exact time the employee was added to the database.

Figure 5.19 presents the first Server Action named *GetAnEmployee*. This Server Action receives an *employee identifier* and fetches the corresponding employee in the database. Box 2 of the image shows the three Server Actions that are considered in this example. Within the module, we also have the *AddNewEmployee*, that receives an *employee* as input. If the name of the *employee* has more than 2 characters, then it inserts the employee into the database, fetches a list of employees from the database and finishes by returning the tuple id where the *employee* were inserted. If the name has less than 3 characters, a User Exception is thrown by the Server Action. At last, we have the *GetThenAdd* Server Action. This Server Action executes the *GetAnEmployee* Server Action, changes the *employee* returned by the executed Action and finally executes the *AddNewEmployee* Server Action to add the changed employee to the database.

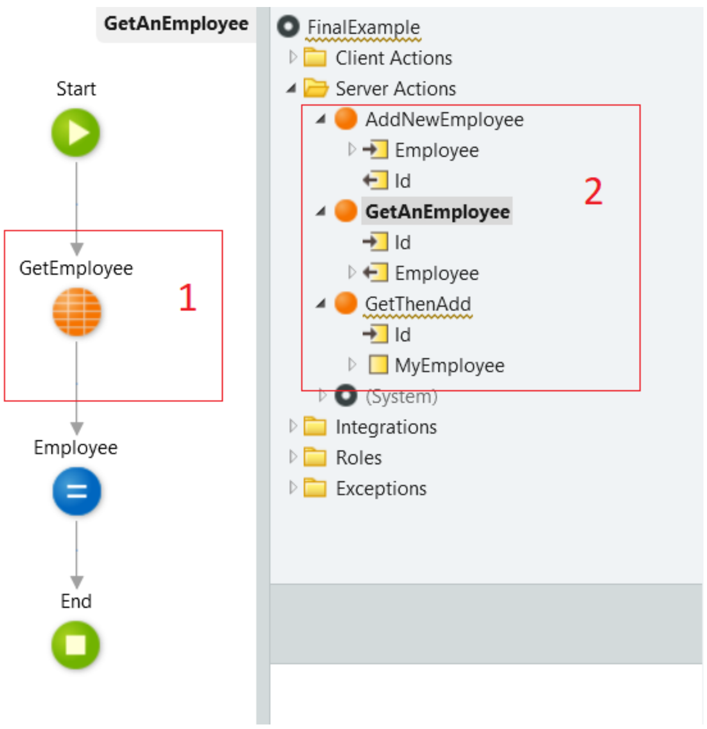


Figure 5.19: Service Studio: Example of a Server Action, the *GetAnEmployee*.

At this point, all the Server Actions in the module are presented and their logic are explained, so now we can see their transformations performed by our tool. The *GetAnEmployee* Server Action keeps the same (without transformation) as it does not have any side-effects to the database. The only Execute Server Action node it has is the *GetEmployee* Entity Action (presented in Figure 5.19, box 1) and it does not write to the database.

Figure 5.20 shows all the transformation performed to the *AddNewEmployee* Server Action by our tool: the transformation of the Execute Server Action node (box 2) to its wrapper (box 4), the extra nodes added to propagate the side-effects to the database (boxes 5, 7, 8 and 9) and the Exception flow added to the transformed Server Action (box 6) to address the Raise Exception node in the original Server Action (box 1). The *Ret_lp* is also added to the transformed Server Action to return all the writes to the database (see example in box 2 of Figure 5.21). Figure 5.10 shows the wrapper for the *CreateEmployee* Entity Action, named *CreateEmployee_lp*. The figure also shows the created nodes to address the side-effects to the database (boxes 2 and 3) and the Exception flow created to handle a Database Exception in case any is thrown (box 6).

To deal with the *GetThenAdd* Server Action, Figure 5.21 presents the transformations performed by the OLP tool. We can notice the additional nodes added to the Server Action to propagate the side-effects to the database (box 1). The Execute Server Action node named (*AddNewEmployee*) keeps with the same name after the transformation but the (Server) Action (within the node) is transformed by our tool. In box 2, it can be seen

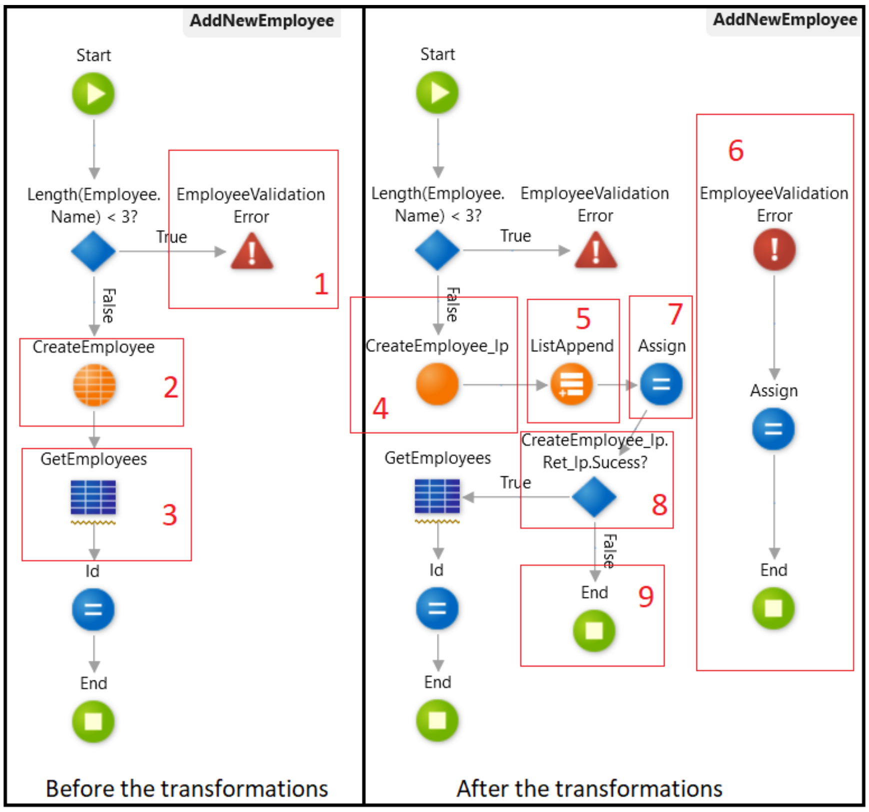


Figure 5.20: Service Studio: Example of the transformations performed to the *AddNewEmployee* by the OLP tool.

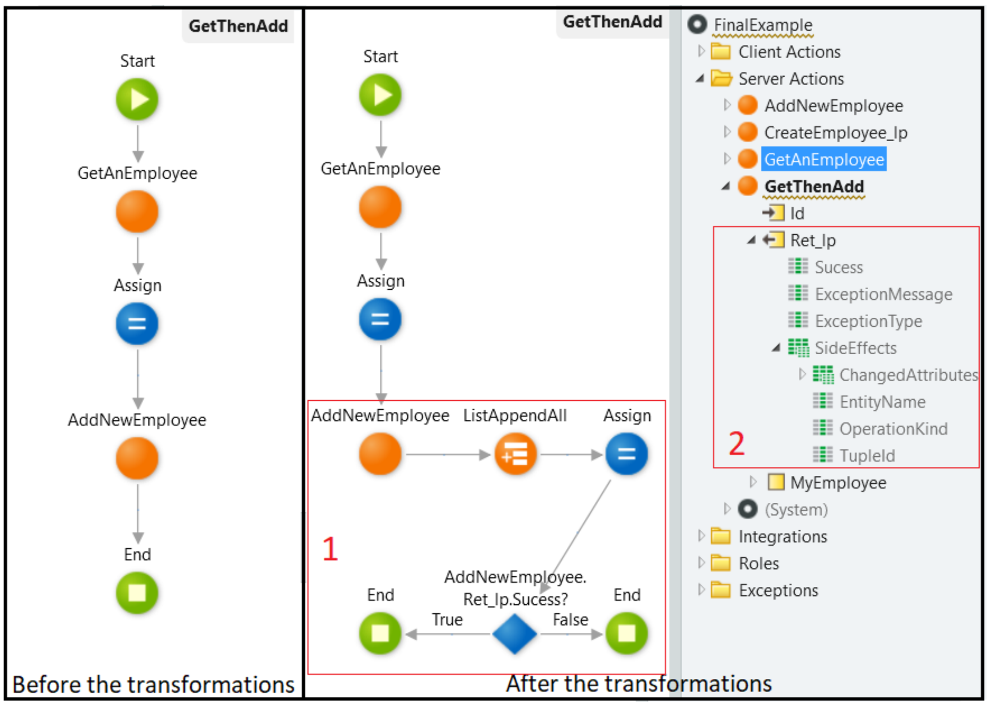
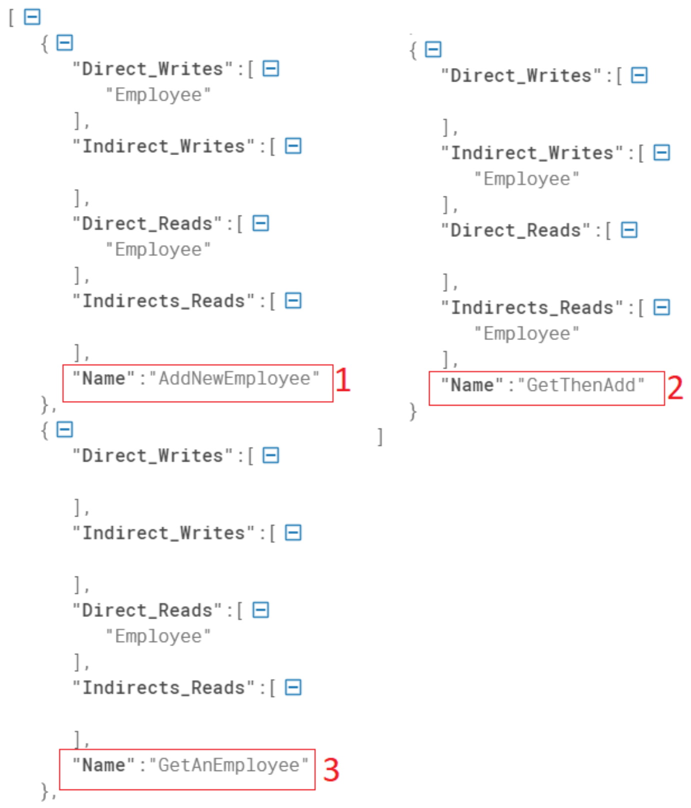


Figure 5.21: Service Studio: Example of the transformations performed to the *GetThenAdd* by the OLP tool.



```
[
   {
      "Direct_Writes":[
         "Employee"
      ],
      "Indirect_Writes":[

      ],
      "Direct_Reads":[
         "Employee"
      ],
      "Indirects_Reads":[

      ],
      "Name":"AddNewEmployee"
   },
   {
      "Direct_Writes":[

      ],
      "Indirect_Writes":[

      ],
      "Direct_Reads":[
         "Employee"
      ],
      "Indirects_Reads":[

      ],
      "Name":"GetAnEmployee"
   },
   {
      "Direct_Writes":[

      ],
      "Indirect_Writes":[
         "Employee"
      ],
      "Direct_Reads":[

      ],
      "Indirects_Reads":[
         "Employee"
      ],
      "Name":"GetThenAdd"
   }
]
```

Figure 5.22: Data structure produced by the OLP tool, applied to the module with the three Server Actions.

the *Ret_lp* added to the Server Action and its structure.

Last but no least, Figure 5.22 presents the data structure (JSON format) produced by the OLP tool, applied to the module. Box 1 shows that *AddNewEmployee* reads and writes the *Employee* entity directly. By looking at box 2, we see that *GetAnEmployee* only reads (directly) the *Employee* entity and finally, box 3 shows that *GetThenAdd* Server Action reads and writes the *Employee* entity indirectly (because this server Action executes/calls the other two Server Actions).

6 Evaluation

In this chapter, we aim to evaluate our approach considering its performance when applied to different modules. To better understand the performance of our tool when applied to real-world OutSystems code, we made some analyses using one of the *datasets* introduced in Section 4.2, the *actionsCount.csv*. The knowledge obtained from these analyses was extremely important to put together the samples used to evaluate the performance of our solution.

6.1 Analysis Overview

The OLP tool was built using the Model API, therefore, the performance of our tool also depends on the performance of some features provided by the Model API. For that reason, we need to perform an evaluation taking into account the features provided by the Model API, meaning that they also need to be evaluated.

Throughout this chapter, we analyse the results of a performance evaluation applied to the OLP tool, using some samples, precisely built after the knowledge obtained from the *actionsCount.csv datasets* analysis. At this point, it is known that the OLP tool transforms the Server Actions within an module, to make them return all their side-effects to the database and also analyse them to produce all their possible direct/indirect reads and writes.

However, before the transformations of the Server Actions, the module (that contains the Server Actions) must be cloned by the Model API, and all transformations are applied to the clone. Hence, we must evaluate several operations that compose the solution, to understand which parts are suitable for an excellent solution and which ones could be enhanced. In this chapter we present all the details of the execution applied to some samples, where we can compare the different parts of the process.

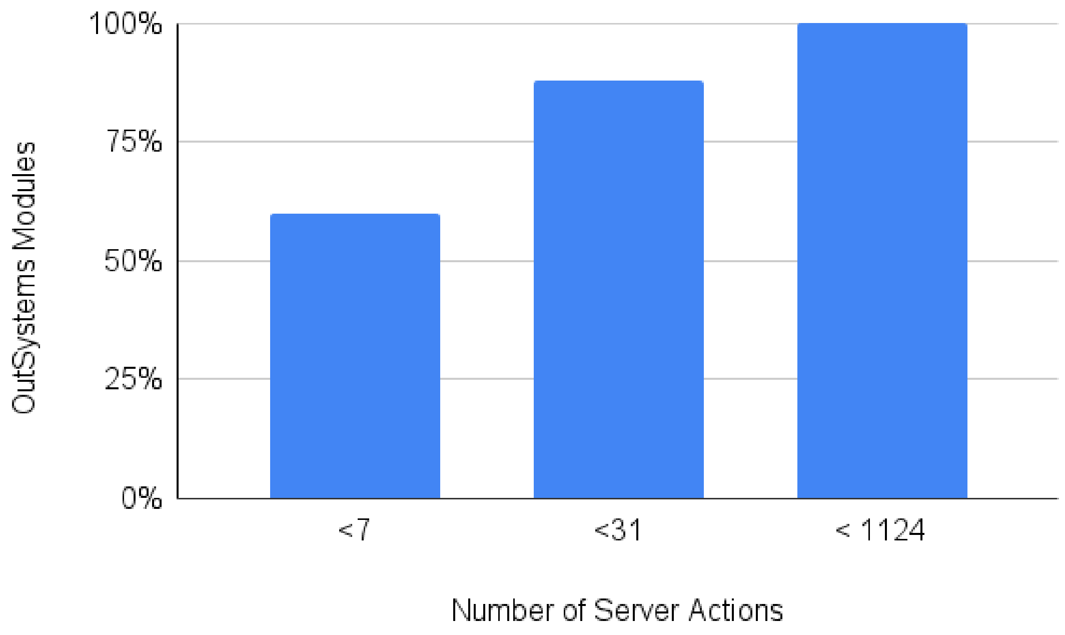


Figure 6.1: Distribution of Server Actions per module in OutSystems applications (obtained from *actionsCount.csv*).

6.2 The OLP Tool Evaluation

Before evaluating the performance of our solution, we first did some analyses using the *actionsCount.csv dataset* with the main purpose of understanding how the Server Actions are distributed in the OutSystems applications code. Knowing this, it would be very important when putting together the samples (modules) that are going to be used to evaluate the OLP tool.

Figure 6.1 shows the distribution of the Server Actions in the OutSystems application, according to the data in the *actionCount.csv dataset*. We can see that 60% of the OutSystems modules have less than 7 Server Actions, 88% have less than 31 Server Actions and only 12% have more than 30 Server Actions. We used the knowledge gathered by these data to built our samples and to test the performance of our solution. We also used the information gathered in Section 4.2, mainly the information about the Server Action nodes presented in Figure 4.4. These analyses were done with the main goal of creating samples that could represent real-word OutSystems code, therefore evaluating the OLP tool using these samples.

6.2.1 Performance Results

In this section we present and discuss the performance results of our tool. The results were obtained using a computer running Windows 10 operating system, equipped with an *Intel(R) Core(TM) i5-8350U - 1.70GHz* and 16GB of RAM.

	Module Size (KB)	Number of Server Actions	Number of Execution Server Action Nodes	Number of Exception Nodes
Module1	123	1	2	0
Module2	145	4	6	1
Module3	174	6	18	3
Module4	221	15	54	40
Module5	524	31	124	80

Table 6.1: The modules used to evaluate the OLP tool performance.

Table 6.1 presents a description of the modules used to test the OLP tool. Five different samples were used to benchmark the solution. All the modules are different in size, in the number of containing Server Actions and in the number of relevant nodes for our approach (such as Execute Server Action nodes, Exception Handler nodes and Raise Exception nodes). The modules also differ in the quantity of *screens*, *Client Actions*, among others, that help to justify the different sizes. They were added to the samples with the main goal of evaluating the *Clone* operation. All the samples were built taking into consideration the analysis performed to the *datasets* mentioned before.

6.2.1.1 Benchmarks

Table 6.2 shows the results obtained by executing our tool with the modules presented in Table 6.1. To obtain each execution time presented in this table, we calculated the average of thirty execution time values (and their corresponding standard deviations). Therefore, by looking at the average execution time of the model service loading (column 3), we notice that it is one of the longest process in our solution (the longest in these samples). The Model API needs, each time we run our solution, to load a service (model service), which gives us access to the OutSystems model. But it is a long process, that takes in average 5.38 seconds to get ready for use.

	Module Size Increase (%)	Service Load Execution Time (seconds)	Module Cloning Execution Time (seconds)	Transformation Execution Time (seconds)
Module1	3.25	5.38 ($\sigma = 0.22$)	1.72 ($\sigma = 0.33$)	0.78 ($\sigma = 0.18$)
Module2	6.90	5.38 ($\sigma = 0.22$)	2.09 ($\sigma = 0.10$)	0.80 ($\sigma = 0.07$)
Module3	16.67	5.38 ($\sigma = 0.22$)	2.33 ($\sigma = 0.13$)	1.23 ($\sigma = 0.06$)
Module4	23.53	5.38 ($\sigma = 0.22$)	2.51 ($\sigma = 0.15$)	2.20 ($\sigma = 0.11$)
Module5	12.21	5.38 ($\sigma = 0.22$)	4.12 ($\sigma = 0.27$)	2.80 ($\sigma = 0.16$)

Table 6.2: Detailed execution time of the OLP tool applied to the modules presented in Table 6.1.

Analysing the process of cloning each module (column 4), we see that, for modules that do not have much data to clone, the process can have a good average performance, such as the module 1. The average execution time for this module is not so high, but in order to add our solution to the OutSystems development environment, this process of cloning a module would have to be enhanced (or replaced). We also notice that, as the size of the modules get bigger, the average cloning time significantly increases. The average cloning time strongly depends on the size of the module being cloned, and for wide modules, this process would need optimization.

After cloning the module, we must analyse the most important part of our solution: the Server Actions transformations (column 5). By looking to the average time of the Server Actions transformations, we notice that, for modules with up to 7 Server Actions, the process is fast (maximum of 1.23 seconds on average). For modules with about 30 Server Actions, the time increases to an average of 2.80 seconds.

Nonetheless, as presented by Figure 6.1, **60%** of the OutSystems modules have **less than 7 Server Actions**, meaning that, in 60% of cases the average execution time of the transformations is close to values between 0.78 and 1.23 seconds. In approximately 28% of cases (modules that have between 7 and 30 Server Actions) the average execution time of the transformations is close to values between 1.23 and 2.8 seconds.

These results show that this part of the solution has the best performance in the whole process. If we take a closer look to the average execution time of the transformations applied to the module 4 and 5 (column 5 of Table 6.2), we see that the difference between their execution times is only 0.6 seconds, but the difference in the number of Server Actions is 16. This is accomplished because our solution, not only **does not make any alteration to the Server Actions with no side-effects to the database**, but also evaluates each Server Action once and the wrappers are reused for the same Entity Actions.

The publishing time has to be added to the total execution time of each module to allow developers to test their code. When the modules are transformed by the OLP tool, their sizes are increased, as can be seen in the second column of Table 6.2. We noticed that, in most cases, the increase is very small, meaning that the publish time of the transformed module is very similar to the original one, as it depends mainly on the module size.

Still on the growth rate of the modules (presented in Table 6.2), we can notice that module 4 has a larger growth rate than module 5, despite the latter having a larger size. This happens for two reasons:

- Our solution reuses wrappers that are already created for the same Entity Actions within the module.

- New Exceptions flows are only created into the transformed Server Actions if there are no Exception handlers already in the Server Actions (that match the raised Exceptions).

	Module Size Increase (%)	Service Load Execution Time (seconds)	Module Cloning Execution Time (seconds)	Transformation Execution Time (seconds)
Module1	3.57	5.38 ($\sigma = 0.22$)	1.52 ($\sigma = 0.07$)	0.66 ($\sigma = 0.02$)
Module2	6.09	5.38 ($\sigma = 0.22$)	1.55 ($\sigma = 0.03$)	0.79 ($\sigma = 0.04$)
Module3	23.20	5.38 ($\sigma = 0.22$)	1.73 ($\sigma = 0.03$)	1.31 ($\sigma = 0.05$)
Module4	35.29	5.38 ($\sigma = 0.22$)	1.77 ($\sigma = 0.03$)	2.22 ($\sigma = 0.05$)
Module5	35.27	5.38 ($\sigma = 0.22$)	1.98 ($\sigma = 0.13$)	2.63 ($\sigma = 0.05$)

Table 6.3: Detailed execution time of the OLP tool applied to the modules presented in Table 6.1, with only Server Actions within each module.

6.2.1.2 Modules With Only Server Actions

To understand the performance results of our tool when applied to modules containing only Server Actions, we put together five new samples resulting from deleting everything from the modules presented in 6.1 but their Server Actions. The new modules has the following sizes: **Module1**: 112 KB, **Module2**: 115 KB, **Module3**: 125 KB, **Module4**: 153 KB, **Module5**: 207 KB.

Table 6.3 shows the execution times obtained after running our tool using the new modules above. The size growth rate is now higher in the bigger modules (3, 4 and 5) because they have more Server Actions. However, the sizes of these modules are much smaller in comparison to the modules in Table 6.2.

The execution time of the Server Action transformations within these new modules (column 5) are very similar to the previous modules, as the Server Actions are targets for our tool. Therefore, we notice that the size of the modules does not affect these transformations.

In relation to the module cloning (column 4), we see that these values are much lower when comparing with the values from Table 6.2. This is because the modules used for this new analysis are much smaller as they only contain Server Actions. For that reason, if OutSystems developers use our tool on the early stages of the modules building or if they decide to first build the applications logic, they can preview their applications logic much faster, because the modules cloning will be faster.

All the performed analyses had great value to understand, not just the performance of our tool, but also to have better knowledge of how OutSystems applications code is built. Knowing the patterns of the real-world OutSystems code helped us design this solution and understand which parts are well designed and which could be enhanced.

6.3 Discussion

As seen in the Section 6.2.1, our solution has some processes that could be enhanced in order to present a much optimized version of the OLP tool and make the feedback loop during development even shorter. We aim to highlight some improvements that could be made to the solution, and were not implemented due to time constraints.

The first part identified was the model service, which takes an average of 5.38 seconds each time the process clones and transforms a module. This cost can be (almost) eliminated if, during the OutSystems development, this service remains active, instead of loading everytime. If this improvement was made, the developer would wait 5.38 seconds (on average) only when the process was run for the first time.

Regarding the module cloning, some improvements could also be made. To enhance this part of the process, we could take into account that, usually during the development process, developers are constantly making minor code changes and they want to preview the effects of the logic. This means that, in case the developer does not change anything in module but the Server Actions, we could just clone the Server Actions that were changed and use the previously cloned module. Thus, we would have to clone the whole module only at the first time we run the OLP tool and whenever the developer makes any changes to the other parts of the module apart from the Server Actions.

We can see the that the values for the module cloning shown in Table 6.3 are much lower when compared with the ones of Table 6.2. That happens because our tool is cloning modules containing only Server Actions. If we copied to the already cloned module (cloned initially) only Server Actions that were modified by the developer, we could save more time than shown in Table 6.3.

7 Conclusions

In this dissertation we presented the logic visualization in the OutSystems environment. They were explained in detail and their key points and flaws were highlighted. We saw that many approaches used during the OutSystems development phase to visualize logic are strongly related with testing techniques, but we also saw that they have several limitations that make the feedback loop that developers have when writing their logic too long.

To better understand and refine the problem, we used some *datasets* containing code used by the OutSystems factories, to understand how the logic is distributed through their applications. To accomplish that, we made some analyses to narrow down the problem and to project our solution.

Knowing where the problem was concentrated and what were the key aspects, we started projecting our PoC prototype, the OLP tool. Our solution was built with the main goal of allowing the developers at OutSystems to preview the effect/side-effects of their logic during development. To accomplish that, we implemented a prototype that is capable of transforming all Server Actions within a module to make them return all their side-effects to the database. Besides that, our tool gives the developer access to all possible reads and writes (directly and indirectly) that each Server Action in the module being built performs to the database. By building this tool, we implemented the bases to preview and control all the side-effects performed to the database, along with the database dependencies between Server Actions.

Finally, the implemented tool was submitted to an evaluation, aiming to obtain the results of its performance when applied to real-world application code. To achieve these results, some samples (modules) were built driven by the data obtained during the analyses performed to the *datasets*. This evaluation was crucial to understand the performance of the prototype and to spot some aspects that can be enhanced. The goals of this dissertation were achieved, as we enable the OutSystems developers to preview and control the side-effects to the database of the logic, thus reducing, most of the time, the feedback loop they have during development.

7.1 Outcomes

One of the key contributions produced by this work are the bases provided to preview and control the side-effects of most of the logic (the Server Actions) within the application being developed. This is achieved by the PoC tool implemented that is able to transform all the Server Actions within a module, making them return all their side-effects to the database and produce all the reads and writes dependencies to database of each Server Action.

A key aspect of the implemented solution is that it can also be used aside with other techniques to have clearer preview of the applications logic. For example, a testing technique can transform a module and then test it using its own approach. The advantage would be that, by testing the transformed module, the developer could visualize all side-effects of the logic being tested. This would reduce the number of required tests and would increase the confidence level of the logic correctness.

Another great contribution provided by this work is a broad and detailed analysis of the OutSystems applications code. The analyses were not only important to narrow down the problem and to project the solution, but they were also essential to understand several aspects of the OutSystems applications code, such as development patterns, logic distributions, application architecture, among others. These analyses were actively used during the first three quarters of 2021 by the *R&D* teams (OutSystems engineering team).

7.2 Future Work

Despite the fact that the main goal of this work was achieved, there are still aspects that need to be improved. We have the following points for a future work, that were not implemented due to the time allocated for this work:

- Integration in OutSystems development process (Service Studio): we consider this as the main aspect to accomplish in a future work. By adding our solution to Service Studio, we can for example, provide a feature where the developer inserts some inputs and the logic being written would be tested using the given inputs, and finally the results would be shown to the developer somewhere in the Service Studio. This would improve and finalize the use of our work during the Outsystems development phase.

- Server Actions depending on more than one module: as seen before, the solution was implemented to deal with Server Actions that only depend on a single module. Despite the fact that our solution represents most of the cases in the OutSystems application, it can be extended by dealing with the rest of the Server Actions.

- The Advanced SQL nodes: The Server Actions can have another type of node that can read or write to the database: the *Advanced SQL* nodes. This kind of OutSystems node performs an SQL [57] query to the application database. Although the CRUD operations are more often executed by the Entity Actions and Aggregates nodes, adding this node helps with the solution completeness.

- Enhance the model service loading and *Clone* operation: Loading the model service each time the Server Actions are about to be transformed is a heavy process, so this can be improved by keeping the model service active during development. Also, the cloning operation can take too much time, especially when the modules being cloned are relatively large.
- Support for Server Actions with database writes inside OutSystems Expressions.
- Wrappers for services, such as REST endpoints, SOAP, etc.
- Usability tests: After integrating the solution into Service Studio, it would be very important to evaluate the solution with some users, to check the details of the UI design, development process, among other relevant aspects.

Bibliography

[1] P. Adragna. “Software debugging techniques”. In: *Inverted CERN School of Computing, iCSC 2005 and iCSC 2006 - Proceedings* (2008), pp. 71–86 (cit. on pp. 20, 21).

[2] K. Beck. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530 (cit. on p. 23).

[3] A. Bertolino and E. Marchetti. “A Brief Essay on Software Testing”. In: *Area* (2003), pp. 1–14 (cit. on p. 11).

[4] E. Buonanno. *Functional Programming in C#: How to write better C# code*. Manning Publications Co., 2018. Chap. 6. ISBN: 9781617293955 (cit. on p. 49).

[5] M. Fowler. *BroadStackTest*. `https://martinfowler.com/bliki/BroadStackTest.html`. (Accessed on 02/18/2021) (cit. on p. 14).

[6] M. Fowler. *Domain-Oriented Observability*. `https://martinfowler.com/articles/domain-oriented-observability.html`. (Accessed on 02/12/2021) (cit. on p. 21).

[7] M. Fowler. *Exploratory Testing*. `https://martinfowler.com/bliki/ExploratoryTesting.html`. (Accessed on 02/11/2021) (cit. on p. 19).

[8] M. Fowler. *Integration Test*. `https://martinfowler.com/bliki/IntegrationTest.html`. (Accessed on 02/17/2021) (cit. on pp. 12, 13).

[9] M. Fowler. *Test Driven Development*. `https://martinfowler.com/bliki/TestDrivenDevelopment.html`. (Accessed on 02/19/2021) (cit. on p. 23).

[10] M. Fowler. *Testing Guide*. `https://martinfowler.com/testing/`. (Accessed on 02/11/2021). Dec. 2019 (cit. on pp. 12, 14, 18).

[11] M. Fowler. *The Practical Test Pyramid*. `https://martinfowler.com/articles/practical-test-pyramid.html?ref=hackernoon.com`. (Accessed on 02/16/2021) (cit. on p. 23).

[12] M. Fowler. *UnitTest*. `https://martinfowler.com/bliki/UnitTest.html`. (Accessed on 02/17/2021) (cit. on pp. 12, 13).

[13] G. Guerra. “Testing support for the OutSystems Agile Platform”. In: (2010) (cit. on p. 22).

[14] JSON. *JSON*. https://www.json.org/json-en.html. (Accessed on 10/20/2021) (cit. on p. 26).

[15] H. Lourenço and R. Eugénio. “TrueChange ™ under the hood : how we check the consistency of large models (almost) instantly”. In: (2019) (cit. on p. 35).

[16] H. Lourenço et al. “LUV is not the answer : Continuous delivery of a model driven development platform”. In: (2020) (cit. on p. 36).

[17] J. M. Lourenço. *The NOVAthesis LaTeX Template User’s Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[18] R. Marvin. *OutSystems Review | PCMag*. https://www.pcmag.com/reviews/outsystems. (Accessed on 02/22/2021) (cit. on p. 1).

[19] Microsoft. *ASP.NET | Open-source web framework for .NET*. https://dotnet.microsoft.com/apps/aspnet. (Accessed on 20/02/2021) (cit. on pp. 7, 22).

[20] Microsoft. *Data Visualisation | Microsoft Power BI*. https://powerbi.microsoft.com/en-au/. (Accessed on 10/20/2021) (cit. on p. 26).

[21] Minigranth. *Software Debugging | Software Testing Tutorial | Minigranth*. https://minigranth.in/software-testing-tutorial/software-debugging. (Accessed on 02/12/2021) (cit. on p. 20).

[22] K. Mohd. Ehmer and K. Farmeena. “A Comparative Study of White Box , Black Box and Grey Box Testing Techniques”. In: *International Journal of Advanced Computer Science and Applications* 3.6 (2012), pp. 12–15. ISSN: 1098-6596. arXiv: arXiv:1011.1669v3 (cit. on p. 14).

[23] OutSystems. *Actions in Reactive Web and Mobile Apps*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Implement_Application_Logic/Actions_in_Reactive_Web_and_Mobile_Apps. (Accessed on 02/05/2021) (cit. on pp. 8, 26, 27).

[24] OutSystems. *Actions in Web Applications*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Implement_Application_Logic/Actions_in_Web_Applications. (Accessed on 02/08/2021) (cit. on p. 10).

[25] OutSystems. *Aggregate*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Data/Handling_Data/Queries/Aggregate. (Accessed on 02/05/2021) (cit. on pp. 8, 9).

[26] OutSystems. *Application Layers < Service Studio Overview - Training*. https://www.outsystems.com/training/lesson/2186/application-layers. (Accessed on 02/04/2021) (cit. on p. 6).

[27] OutSystems. *Assign*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/Assign. (Accessed on 02/08/2021) (cit. on p. 9).

[28] OutSystems. *Automated Testing Tools*. https://success.outsystems.com/Documentation/Best_Practices/OutSystems_Testing_Guidelines/Automated_Testing_Tools?_gl=1*tch51t*_ga*MTAxOTI4MjgOMC4xNjA2NTg2NTgy*_ga_ZD4DTMHWR2*MTYxMjg4NDMwMy4zMi4xLjE2MTI4ODg5MzMuMjY.. (Accessed on 02/09/2021) (cit. on pp. 14, 15).

[29] OutSystems. *BDDFramework - Overview*. https://www.outsystems.com/forge/component-overview/1201/bddframework. (Accessed on 02/09/2021) (cit. on pp. 15, 17).

[30] OutSystems. *End*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/End. (Accessed on 02/08/2021) (cit. on p. 10).

[31] OutSystems. *Entities*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Data/Data_Modeling/Entities. (Accessed on 02/08/2021) (cit. on p. 9).

[32] OutSystems. *Extensions*. https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/Extend_Logic_with_Your_Own_Code/Extensions. (Accessed on 02/04/2021) (cit. on p. 6).

[33] OutSystems. *For Each*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/For_Each. (Accessed on 02/08/2021) (cit. on p. 10).

[34] OutSystems. *How does OutSystems support testing and quality assurance? | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/how-does-outsystems-support-testing-and-quality-assurance/?origin=d. (Accessed on 02/09/2021) (cit. on pp. 14, 15).

[35] OutSystems. *If*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/If. (Accessed on 02/08/2021) (cit. on p. 10).

[36] OutSystems. *Modular Programming < Intro to OutSystems Development - Training*. https://www.outsystems.com/training/lesson/2159/modular-programming. (Accessed on 02/04/2021) (cit. on p. 7).

[37] OutSystems. *OutSystems Again Named a Leader in Gartner's 2018 Magic Quadrant for Enterprise High-Productivity Application Platform as a Service*. https://www.outsystems.com/News/high-productivity-apaas-gartner-leader/. (Accessed on 02/03/2021) (cit. on p. 1).

[38] OutSystems. *OutSystems development and management tools | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/development-and-management-tools/. (Accessed on 02/03/2021) (cit. on pp. 4, 6).

[39] OutSystems. *OutSystems Evaluation Guide*. https://www.outsystems.com/evaluation-guide/. (Accessed on 02/03/2021) (cit. on p. 4).

[40] OutSystems. *OutSystems Platform Services | Evaluation Guide*. (Accessed on 02/05/2021). URL: https://www.outsystems.com/evaluation-guide/platform-services/#1 (cit. on p. 7).

[41] OutSystems. *Platform Runtime | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/platform-runtime/. (Accessed on 02/03/2021) (cit. on pp. 4, 5).

[42] OutSystems. *Screen*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Design_UI/Screen. (Accessed on 02/05/2021) (cit. on pp. 7, 8).

[43] OutSystems. *Screen Templates*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Design_UI/Screen_Templates. (Accessed on 02/05/2021) (cit. on p. 7).

[44] OutSystems. *Service Studio Overview*. https://success.outsystems.com/Documentation/11/Getting_started/Service_Studio_Overview. (Accessed on 02/03/2021) (cit. on p. 5).

[45] OutSystems. *Service Studio Walkthrough < Service Studio Overview - Training*. https://www.outsystems.com/training/lesson/2185/service-studio-walkthrough. (Accessed on 02/04/2021) (cit. on p. 6).

[46] OutSystems. *Start*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/Start. (Accessed on 02/08/2021) (cit. on p. 10).

[47] OutSystems. *Switch*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/Switch. (Accessed on 02/08/2021) (cit. on p. 10).

[48] OutSystems. *Testing OutSystems Applications | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/testing-outsystems-applications/. (Accessed on 02/09/2021) (cit. on p. 14).

[49] OutSystems. *The Modern Application Development Platform*. https://www.outsystems.com/platform/. (Accessed on 02/03/2021) (cit. on pp. 1, 4).

[50] OutSystems. *Unit Testing Framework - Overview*. https://www.outsystems.com/forge/component-overview/387/Unit+Testing+Framework/. (Accessed on 02/09/2021) (cit. on p. 15).

[51] OutSystems. *Web Logic Tools*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools. (Accessed on 02/08/2021) (cit. on p. 10).

[52] OutSystems. *Why OutSystems? | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/why-outsystems/. (Accessed on 02/04/2021) (cit. on p. 7).

[53] J. Proenca. *Your Complete Guide To BDD Testing In OutSystems*. https://www.outsystems.com/blog/posts/bdd-testing/. (Accessed on 02/09/2021) (cit. on pp. 15–17).

[54] Python. *Welcome to Python.org*. https://www.python.org/. (Accessed on 10/20/2021) (cit. on p. 26).

[55] J. C. Seco. "Interpretation and Compilation of Programming Languages Part 1 - Overview". In: (2014), pp. 1–11 (cit. on p. 23).

[56] J. Shore. *The Art of Agile Development: Test-Driven Development*. http://www.jamesshore.com/v2/books/aoad1/test_driven_development. (Accessed on 02/19/2021) (cit. on p. 23).

[57] W3schools. *SQL Tutorial*. https://www.w3schools.com/sql/. (Accessed on 11/01/2021) (cit. on p. 68).

[58] M. Wenzel. "READ-EVAL-PRINT in Parallel and Asynchronous Proof-checking". In: *Electronic Proceedings in Theoretical Computer Science* 118 (2013), pp. 57–71. ISSN: 2075-2180. DOI: 10.4204/eptcs.118.4 (cit. on p. 23).

[59] R. Wilsenach. *QA in Production*. https://martinfowler.com/articles/qa-in-production.html. (Accessed on 02/12/2021) (cit. on p. 21).