Model-based Data Synchronization Framework for Data-driven Serverless Applications

Author

Fatima Samea

FALL 2016-MS-16(CSE) 00000171206

Supervisor

Dr. Farooque Azam

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

ISLAMABAD

JANUARY 2019

Model-based Data Synchronization Framework for Data-driven Serverless Applications

Author

Fatima Samea

FALL 2016-MS-16(CSE) 00000171206

A thesis submitted in partial fulfillment of the requirements for the degree of

MS SOFTWARE Engineering

Thesis Supervisor:

Dr. Farooque Azam

Thesis Supervisor's Signature: _____

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY,

ISLAMABAD

JANUARY 2019

Declaration

I certify that this research work titled "*Model-based Data Synchronization Framework for Data-driven Serverless Applications"* is my own work. The work has not been presented elsewhere for assessment. The material that has been used from other sources it has been properly acknowledged / referred.

_____

Signature of Student

Fatima Samea

FALL 2016-MS-16(CSE) 00000171206

LANGUAGE CORRECTNESS CERTIFICATE

This thesis is free of typing, syntax, semantic, grammatical and spelling mistakes. Thesis is also according to the format given by the University for MS thesis work.

_____

Signature of Student

Fatima Samea

FALL 2016-MS-16(CSE) 00000171206

_____

Signature of Supervisor

Copyright Statement

Acknowledgements

I am thankful to my Creator Allah Subhana-Watala to have guided me throughout this work at every step and for every new thought which You setup in my mind to improve it. Indeed, I could have done nothing without Your priceless help and guidance. Whosoever helped me throughout the course of my thesis, whether my parents or any other individual was Your will, so indeed none be worthy of praise but You.

I am profusely thankful to my beloved parents who raised me when I was not capable of walking and continued to support me throughout in every department of my life.

I would also like to express special thanks to my supervisor **Dr. Farooque Azam** and **Dr. Wasi Haider** for their help throughout my thesis and also for Software Development and Architecture (SDA) and Model Driven Software Engineering (MDSE) courses which they have taught me. I can safely say that I haven't learned any other engineering subject in such depth than the ones which he has taught.

I would like to pay special thanks to **Muhammad Waseem Anwar** for his tremendous support and cooperation. Each time I got stuck in something, he came up with the solution. Without his help I wouldn't have been able to complete my thesis. I appreciate his patience and guidance throughout the whole thesis.

I would also like to thank **Dr. Arslan Shaukat**, **Dr. Urooj Fatima** for being on my thesis guidance and evaluation committee. I am also thankful to **Mehreen Khan** for her support and cooperation.

Finally, I would like to express my gratitude to all the individuals who have rendered valuable assistance to my study.

Dedicated to my husband and children whose extraordinary support and cooperation always remained a source of motivation for me in accomplishing this tremendous achievement.

Abstract

Serverless Computing is a new approach to cloud computing in which the cloud provider dynamically manages the allocation of machine resources. The recent trend in writing serverless applications mainly addresses the web as well as other data-driven and event-driven distributed applications which frees developers from maintaining a server. Developers just focus on the application logic in developing applications. Data management is the foundation of high-quality data-driven serverless applications in use today. Such data-driven applications are used on multiple devices, over a variety of connections and it is essential to offer a consistent user experience across different connection types. Particularly, the growing number of these serverless applications that allow activities such as messaging, commenting and collaboration present users with updated information using real time ability. Consequently, design and implementation of behavior of such serverless applications is very complex especially when data is shared among multiple users. To target this issue, this research introduces UMSDA (Unified Modeling Language Profile for Serverless Data-driven Applications) which adapts the concept of UML Class Diagram, Object Diagram and State Machine Diagram to model the frontend and backend requirements for data-driven serverless applications at high abstraction level. To resolve the complexity of application behavior, backend requirements containing data store and sync concepts are modelled in UMSDA. For integration of data with the view layer, UMSDA covers the frontend requirements including the user interface and data binding. As a part of research, a complete transformation engine is developed using Model-to-Text approach to automatically generate frontend and backend low level implementations of Angular 2 and GraphQL respectively from high level source UMSDA models. Finally, the validation of this research work is presented through two case studies: 1) Real-time Chat Application and 2) Weather Forecast Application, deployed on AWS (Amazon Web Services) Serverless platform. The outcomes prove that the proposed framework allows the modeling of both frontend as well as backend requirements of data-driven serverless applications with simplicity. Furthermore, a transformation engine is capable to automatically generate the deployable Angular 2 and GraphQL code. Finally, it has been concluded that the proposed framework greatly simplifies the design and implementation complexity of data-driven serverless applications to achieve certain business objectives like productivity and time to market.

Key Words: *UMSDA, Data-driven, Data Synchronization, MDA, Serverless Applications*

Table of Contents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

CHAPTER 1: INTRODUCTION

This chapter delivers a comprehensive introduction of the research which is categorized in different sections. Section 1.1 presents background study. Section 1.2 is problem statement. Section 1.3 is proposed methodology. Section 1.4 is research contribution and Section 1.5 represents thesis organization.

1.1 Background Study

The background study introduces the concepts being used in this research which are; 1) Serverless Computing, 2) Data-driven Applications and 3) Model Driven Architecture. The details of the following are given in subsequent sections.

1.1.1 Serverless Computing

Serverless is a new approach to cloud computing where the cloud provider dynamically manages the allocation and provisioning of servers. It allows us to build applications without managing the complex infrastructure such as containers, virtual machines etc. As compared to traditional applications, a serverless application runs in stateless compute containers that are short-lived, event-triggered and fully managed by the cloud provider. Pricing is based on the consumption of resources, rather than on pre-purchased compute capacity. Applications can be written to be purely serverless and use no provisioned servers at all i.e. *"Focus on application, not the **infrastructure**"*.

This is a substantial change from the application hosting platform-as-a-service providers. Instead of servers running continuously, functions are deployed that operate as event handlers, and the cost occurs only for CPU time based on execution of these functions [1]. The developers write concise, stateless functions which can be triggered through events produced from different sensors as well as services / users or middleware [2]. Amazon was the first one to introduce serverless paradigm in year 2014 by introducing Lambda. Nowadays almost every cloud provider e.g Google, Apache, Microsoft etc offers serverless platform which appeals to many developers. The developers just focus on the application logic while shifting infrastructure concern to platform. In response for writing stateless short-running functions, the platform ensures secure and timely execution of these functions. The function implements a business logic, depending on the desired result of the application [3]. These functions are typical piece of code in well-known programming languages that perform execution in a stateless manner. The developers use special cloud functions such as AWS

Lambda [4], IBM OpenWhisk functions [5], Azure Functions [6] and Google Cloud Functions [7] for executing custom application codes. The vendor is responsible to run the infrastructure efficiently as well as optimize resources for satisfying execution of functions owing to changes due to user demand [8]. Various applications are developed using serverless cloud computing. Data-driven apps are one amongst them. Developing these applications is a tedious and error prone task. The details are in the next section.

1.1.2 Data-driven Applications

Data-driven applications have become a major growth engine for the worldwide software market. The application flow is governed by data it processes. Input data set can change the behavior of your application. Programming logic will remain same, but it will be coded such that input changes the way application behaves. Designing these applications still requires technical and programming skills. Another important aspect is real time data synchronization which is a complex process to be handled in synchronizing data among multiple devices and automatically catering changes back and forth. Most modern applications are written to take benefit of RESTful web services for accessing data that uses HTTP requests to GET, PUT, POST and DELETE data. For applications using REST (Representational State Transfer) for serverless backend, a set of Lambda functions behind an API gateway are used to access the data.

1.1.3 Model Driven Architecture

MDA (Model-driven architecture) is a renowned software design methodology to abstract the complexity of software development. It simplifies the design and development of software applications [9]. Therefore, it is generally applied in diverse domains like embedded systems [10] etc. and is used in complex application development [11]. Models are used as a set of guidelines to structure design specifications. In MDA, business and application logic is separated from the underlying platform technology [12]. The functionality of the system is defined without having any technology-specific implementation information i.e. as a platform-independent model. Transformation techniques convert platform-independent models into platform-specific models. The MDA model is related to multiple standards. Among them UML (Unified Modelling Language) is one of the most powerful language which has been used by many software engineers. UML provides a standard means for visualizing the design of the system.

UML diagrams are divided into two groups: Structural and Behavioral. Structural diagrams represent the structure of the system whereas Behavioral diagram describes the functionality of the software system. UML conceptual models can be customized and extended using a UML profile diagram. A profile is a lightweight extension mechanism to the UML standard [13].

1.2 Problem Statement

Data driven serverless applications are the most widely used applications in software industry. The demand of these applications is increasing day by day. However, implementing these applications on cloud is a complex process. Firstly, managing identity to authenticate and authorize users is a difficult process. Secondly, there are so many devices and a lot of variations in platform. Moreover, data is very complex to manage, specially synchronization. Different storage systems are used to get data. Millions of users using the applications would require real time data whenever any change happens on the server database. The data is then distributed and sent down to the concerned client among the millions of clients. However, aggregating all data and displaying data in the application without introducing any complexity is a difficult task.

The development complexity increases as data requirements vary across devices and becomes harder when multiple users share data. The practical applications in industry are providing solution to these applications, still its behavior is complex and complicated to develop on cloud. In response to this problem, this research provides a solution that simplifies the design and low-level implementation complexity by introducing Unified Modeling Language Profile which provides a high level of abstraction and automatically generate target code.

1.3 Proposed Methodology

Research is carried out in a systematic way. Figure 1 presents the step by step flow of research. Firstly, we identify the problem and proposed a solution for the identified problem. A comprehensive and detailed literature review has been carried out to find the ideal solution for the problem. Also, the related work regarding the proposed solution is analyzed.

Furthermore, the proposed solution comprises a model-based approach for data synchronization in serverless data driven applications at platform independent level. A UML Profile is defined to support the modelling of both frontend and backend implementations. It

also provides tool for model to text transformation. The transformation engine is based on mapping rules which transforms the source model into target low level implementation code. The transformation has been verified by two case studies.

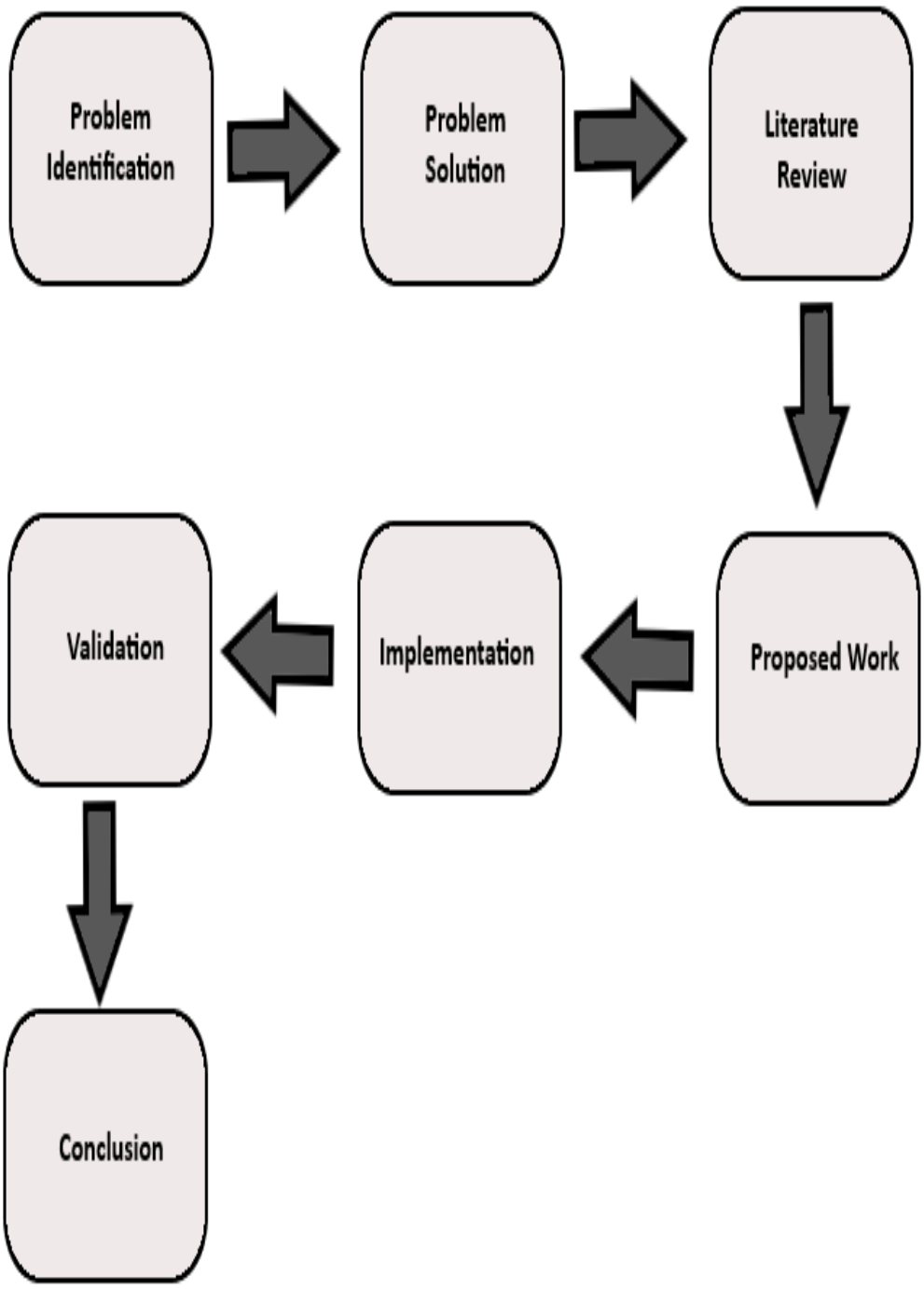


Figure 1**:** Research Flow

1.4 Research Contribution

The contribution made in this research work are as follows:

- Modeling frontend of the data driven applications at platform independent level including User Interface (UI), Data Sync Client (which communicates with the backend and bind data to UI).
- Modeling backend at platform independent level to manage real time data and present the updated information to users.
- Transformation engine to generate frontend and backend implementation by transforming higher level model to low level implementation code in Angular and GraphQL. The transformation engine is developed using Acceleo.
- Validation of the proposed work by deploying it to AWS Serverless platform.

1.5 Thesis Organization

Figure 2 represent the thesis organization**. Chapter 1:** Deals with introduction consisting of background study about the concepts used in research, problem statement, research contribution and thesis organization. **Chapter 2: C**omprises of literature review and deliver the description of the work done in serverless computing and data driven applications. Research gaps are also highlighted. **Chapter 3:** Describes the details of proposed methodology used for identified problem. It presents UMSDA (Unified Modeling Language Profile for Serverless Data driven Applications) for building data-driven applications. **Chapter 4:** Covers the detailed implementation regarding the transformation rules for the proposed UML profile and transformation engine along-with its architecture. **Chapter 5:** Presents the validation of the proposed methodology using two case studies. The case studies used are Real-time Chat application and Weather Forecast Application. The applications are verified by deploying it to AWS Serverless platform to validate our proposed approach. **Chapter 6:** Narrates a brief analysis of our proposed work with the previous research works. **Chapter 7:** A brief discussion on the work done is presented in this chapter. It also contains the limitations in our research. **Chapter 8:** Concludes the research and recommends future work.

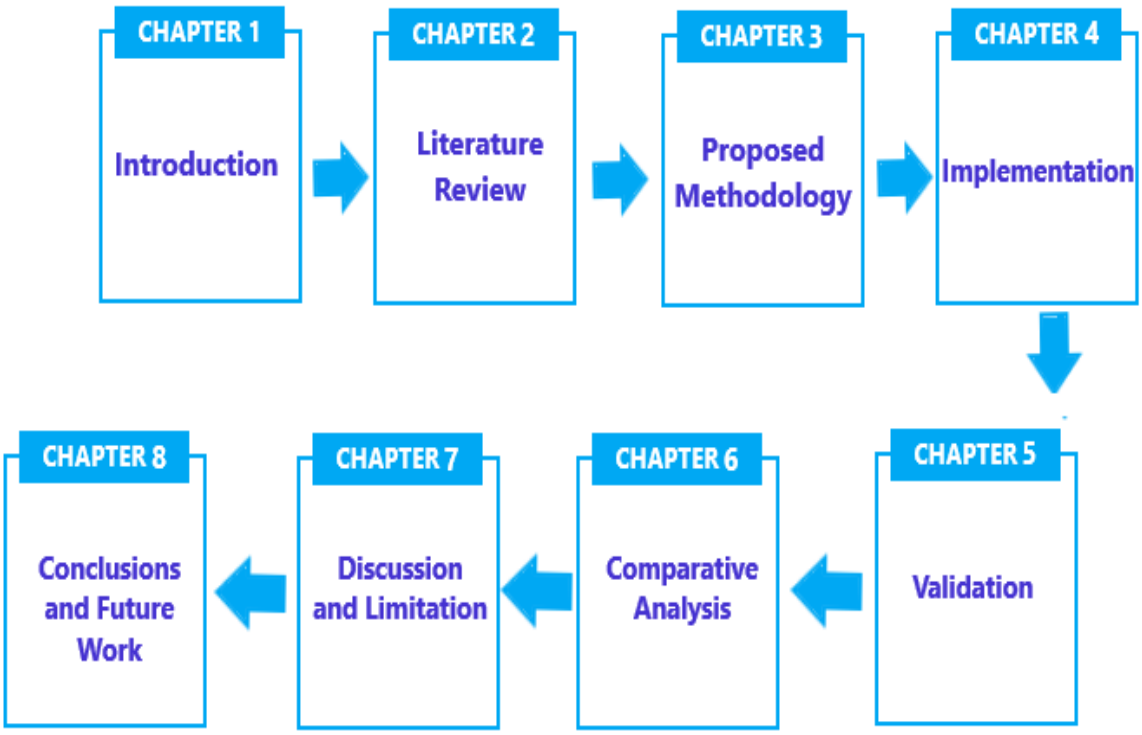


Figure 2: Thesis Outline

Chapter 2

Literature Review

CHAPTER 2: LITERATURE REVIEW

This chapter presents research work conducted in cloud computing for developing applications. After a brief literature review of work conducted in this area we highlighted the research gaps that we found in previous works.

2.1 Literature Review

Nowadays, an IT industry is moving into a wide array of cloud platforms. Various researches related to **cloud application development** has been found in literature. Xiuwei Zhang et al. [14] proposes an on-demand Service Oriented Architecture based approach for enterprise and mashup applications development. Their proposed MDA approach uses Role and Goal, Process-Service meta-model through which they achieve the model transformation of Computation Independent Model, Platform Independent Model and Platform Specific Model. They present a prototype application of their proposed framework. However, they develop an initial prototype model of their proposed work.

Wei-Tek Tsai et al. [15] propose a SaaS (Software as a service) design strategies for applications build using Google App Engine as platform as a service. They use a model-based approach for multitenancy, customization, scalability etc. to automatically generate code for Google App Engine based SaaS application. Their proposed model driven approach saves time and effort in developing a large scalable software system.

Wei-Tek Tsai et al. [16] propose a SaaS Application development framework. Their proposed framework decreases the tenant workload, provides a layered customization model based on keyword search engine and also provides testing support in the SaaS application development. Their proposed framework provides two ways for building SaaS application. In the first way, the tenants publish the specifications of the application with their requirements, while in the second way, the tenants use the templates provided in their proposed framework for composing the application.

Assylbek Jumagaliyev, Jon Whittle [17] analyzes variability challenges for the development of multitenant SaaS applications. They propose an idea of model driven engineering as a potential solution to achieve the variability in software as a service application. They consider the surveys application and discuss the challenges faced during its development. For instance, in multitenant SaaS application if one instance of an application fails, it affects all the tenants sharing that instance.

Xiaoyan Jiang et al. [18] propose a model-based approach for SaaS application platform that focuses on three aspects such as multitenancy, integration and customization over the platform. However, their proposed work is partially complete and did not cover security and scalability issues of the platform.

Xiyong Zhu [19] describes the design and implementation of an XML based service template markup language to provide a comprehensive solution for customizing SaaS applications for distinct users. Dapeng Chen et al. [20] propose a process customization framework for SaaS environment that configures and customizes processes keeping the tenancy isolation. Each tenant administrator customizes the process to meet the demand. Tenancy user information is used to guarantee the isolation between tenants.

G Fylaktopoulos et al. [21] propose a CIRANO platform with Model driven development for rapid development of advanced cloud applications. However, as an application platform as a service, it still has some shortcomings in terms of development such as change in programming language by developers, their methodology and the set of tools they used in development. Ioana Baldini et al. [22] demonstrates the usage of OpenWhisk in mobile applications development which allows the Application programming interface customization for mobiles and simplifies the architecture of mobile applications.

Moez Essaidi [23] presents a platform that provides on-demand business intelligence services. Their proposed platform supports current business intelligence and data ware housing problems. They use model driven approach to design the data ware house framework through a 2-track unified process. Their proposed approach offers a mix-driven approach including a combination of user, data and the goal-driven approaches with semantic-driven, model-driven approaches.

The core of **data-driven applications** is visualization and manipulation of data. Richard Fujimoto et al. [24] propose a dynamic data driven system to track a vehicle. Their proposed approach tracks a vehicle's movements form the live video and image data to predict future locations and data dissemination over wireless network. They use image processing algorithms to capture the vehicle from the real time image data. The proposed framework selects a suitable prediction model dynamically based upon currently available data.

Biplab Deka [25] presents the different data-driven approaches for mobile application design including interaction mining and dynamic components of application design. They present

two approaches for interaction mining existing Android Applications. One of the application capture data seamlessly in the background while uses human interaction for exploring an application. The second application uses automated agent leveraging human interaction. Also, three interaction mining mobile applications are presented by the resulting data.

Jiaju Wu et al. [26] designed a data driven based universal data editing framework which takes the data models as input and only corresponding data models need to be adapted on data alteration. They used model driven approach in designing their framework. Two model transformations are used in their approach. Firstly, they transform the data model into java code and then they transform the data model into relational database schema which provides an HTML preview and two editing patterns. However, their framework has a limitation that they only focus on five components which ultimately leads to incorrect working of some their data models.

Joao Seixas [27] proposes a model driven approach namely XIS-Web technology for the development of responsive web applications. Their approach comprises of two main components which are XISWeb modeling language and XIS-Web framework. The first one is implemented as UMLProfile while the other focuses on the software tools that support their approach. Their framework consists of four main characteristics: 1) splits web application modeling in 6 views to promote separation of concerns for managing complexity. 2) generates the interaction spaces and navigation among the views to relieve the complex task from the user. 3) allowing flexible development of responsive web applications using modern web technologies such as JavaScript, CSS and HTML5. 4) allows the creation of (PIM) platform-independent models.

Kapil Kumar et al. [28] provide a dashboard framework which amalgamates data from various analytics sources i.e. Flurry, JSON, Google Analytics and Excel files to create a customizable user interface. They use two configuration files for dashboard configuration which are generic information and individual services. Also, they develop a prototype for their proposed framework. Voon Yang Nen, Ong Chin Ann [29] develop a prototype tool named pigeon-table for developing data-driven web application to reduce the development time and effort. Their proposed tool retrieve data from MySQL database. The retrieved data is then rendered into an interactive table in a web application and also responsive for mobile devices.

Steffen Vaupel et al. [30] propose a general architecture for context aware data and transaction management in mobile applications. Their proposed framework support changing network states so that users can use applications in both online and off line conditions. They also implement a prototype for Android and used simulation experiments to validate their approach.

Janis Kampars, Janis Grabis [31] addresses the problem of data integration and processing for designing data driven applications with the introduction of auto scaling and adjustment platform for cloud based systems using model driven approach which supports configuration, scalability, context processing algorithms etc.

With the paradigm shift to **serverless computing**, a few research studies also mention application development with serverless computing. For instance, Mengting Yan et al. [32] present a chatbot prototype using serverless platform. Their proposed architecture is based on function sequences which coordinates with the cognitive services in the Watson Developer cloud so that the chatbot can interact with the outside services. Nirmal K Mukhi [33] present a practical application of a personalized tutoring system using serverless technology. They implement the Watson Tutor orchestration logic using OpenWhisk. Their designed system is multimodal (provides a tutoring understanding comprising of various learning activities), stateful and interactive dialog.

Josep Sampe et al. [34] propose a data-driven middleware for object storage using serverless computing. The proposed data driven functions captures and operate over objects as they are read or write to an object store. They use OpenStack Swift to implement a prototype of their serverless framework which store huge volume of data through RESTful API. They evaluate their approach on swift by running benchmarks to observe the behavior of the functions. The results show that their functions runtime is five times less than AWS lambda runtime.

Li Weiping [35] analyze challenges in the workflow system of SaaS applications and provide a solution to the identified problems based on these characteristics: workflow model, wokflow engine and time management. Garrett McGrath et al. [36] discuss the current state of cloud event services and highlighted the difference between application programming paradigms with cloud events and software designs with more traditional infrastructure by presenting two real world applications. Also, they highlight challenges in cloud events.

Jaewook Kim et al. [37] propose a GPU-enabled serverless computing framework. Their proposed framework deploys services efficiently as compared to current serverless computing framework. They implement an Application Programming Interface that integrates the open source serverless framework to the NVIDIA-Docker and commands used for enabling GPU Programming. They measure the performance of their approach through experiments.

Theo Lynn et al. [38] analyze the seven enterprise serverless computing platforms. Their results suggest that several use cases which are currently unaddressed by the academic community can be targeted to serverless cloud computing. Furthermore, they also suggest the issues which needs to be addressed in two major areas i.e System level challenges, programming model and Dev-Ops challenges.

Markus Ast et al. [39] propose an approach for enabling self-contained web components in serverless computing. Usually, web components functionality consists of presentation and business logic. Their proposed approach deploys the business logic of web component as cloud hosted functions which can easily be integrated into existing websites, thereby reducing cost and time. However, their proposed solution cannot be merged into current cloud providers due to user privileges for security and privacy requirements. Vendor lock in is another challenge due to non-compatibility of APIs.

2.2 Research Gaps

This section discusses the research gaps from the literature review. After analyzing the above literature, it has been observed that the increasing use of multiple devices (e.g laptops, mobile, tablets, watches etc.) has increased the importance of data driven serverless applications. Researchers propose different type of frameworks for data driven applications for both web and mobile. A few of them uses model driven engineering [27, 29] but does not cover serverless architecture. Furthermore, few studies support real time data synchronization capabilities on different devices. Developing such serverless data driven applications is a complex process. Particularly, data management is difficult for applications to be developed on cloud because of accessing backend resources and changing network states. Furthermore, low level implementation of application behavior is very complex especially when data is to be distributed among several users and on different devices. Although there are practical applications developed in industry for real time and offline data synchronizations, but

implementation complexity increases for developers. Therefore, the need arises for a solution to simplify the low-level implementation for serverless data driven applications.

Chapter 3

Proposed Methodology

CHAPTER 3: PROPOSED METHODOLOGY

As mentioned earlier, software market is evolving towards the data-driven applications. However, developing such applications is a time consuming and tedious task. This chapter discusses our proposed UML profile for modelling Data-driven Serverless Applications at platform independent level.

In UML, a profile provides a light-weight generic extension mechanism for customizing UML models for specific application domains and platforms which are based on various elements: Stereotypes (allow you to increase UML vocabulary) , Tag values(used to extend the UML properties so that we can add additional information in the specification of a model element) and Constraints(to specify conditions that must be held true at all time). These all are applied to specific model elements such as Classes, Operations, Attributes, Activities etc.

3.1 UMSDA Description

The proposed UMSDA is developed in Unified Modeling Language tool Papyrus based on Eclipse. Our proposed profile provides stereotypes to model both the frontend and backend of the data driven serverless application including the application behavior. These stereotypes are the extensions of UML meta-classes and it provide support to modeling using multiple UML diagrams. Figure 4 shows a complete picture of the proposed profile. Numerous concepts are defined for frontend modelling. Various stereotypes related to Application Structure are defined. Different stereotypes for User Interface are defined. For modelling the backend, stereotypes for data management tasks such as data store, data synchronization and updated real time data are defined. Some stereotypes related to data fetching from the server are defined and modelled in a profile. Each of these stereotypes consists of tagged values. The details of each of the stereotype along with its functionality is discussed in the sub section. Transformation rules are defined for model to text transformation.

3.1.1 Structural Concepts of Application

Figure 3 shows the stereotypes used to model the frontend application structure. It consists of four stereotypes and two base classes.

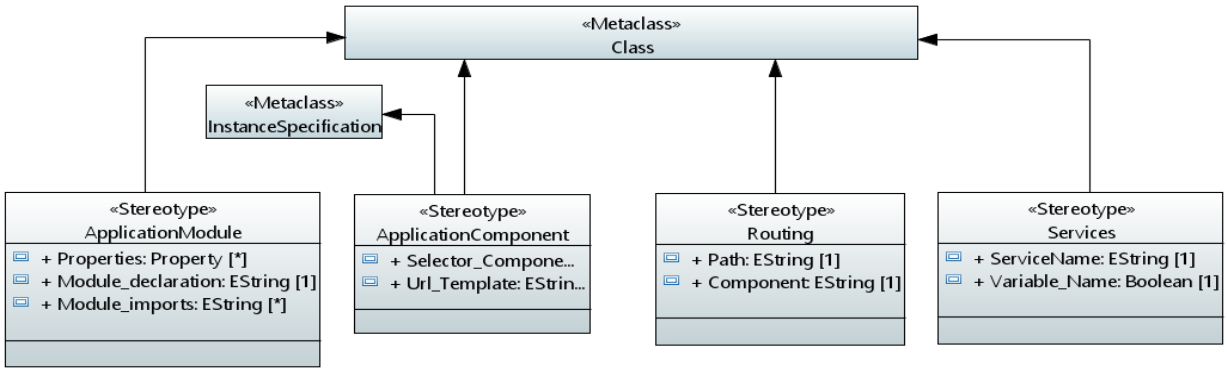


Figure 3: Structural Concepts of Application

1) Application Module Stereotype

Description: Stereotype "*Application Module*" is used for launching an application.

Base Class: Class

Tag Values: It consists of three tag values: Properties of type Property, Module_declaration of type String and Module_imports of type String. *Properties* defines the service providers that become accessible in all parts of the application. *Module_declaration* is used to define the components that belong to this Application module. *Module_imports* defines the exported declarations of other modules which are available in the current module.

2) Application Component Stereotype

Description: Stereotype "*ApplicationComponent*" is defined to separate the functionality.

Base Class: Class, Instance Specification

Tag Values: It comprises of two tag values: Selector_Component and Url_Template. The tag value *Selector_Component* create and insert an instance of the component. *Url_Template* is the address of the components template which needs to be rendered in the application.

3) Routing Stereotype

Description: Stereotype "*Routing*" is responsible for navigation.

Base Class: Class

Tag Values: It has two tag values: Path and Component. *Path* is a string that matches the URL in the browser address bar. *Component* defines the component that the router should create when navigating to this route.

4) Services Stereotype

Description: Stereotype "*Services*" gets the data and configuration from the server.

Base Class: Class

Tag Values: The tag value Name is the name of service used in the application. The tag value *VariableName* enables/disable the offline setting option.

3.1.2 User Interface Concepts

The user interface concepts modelled are defined in **Figure 4**.

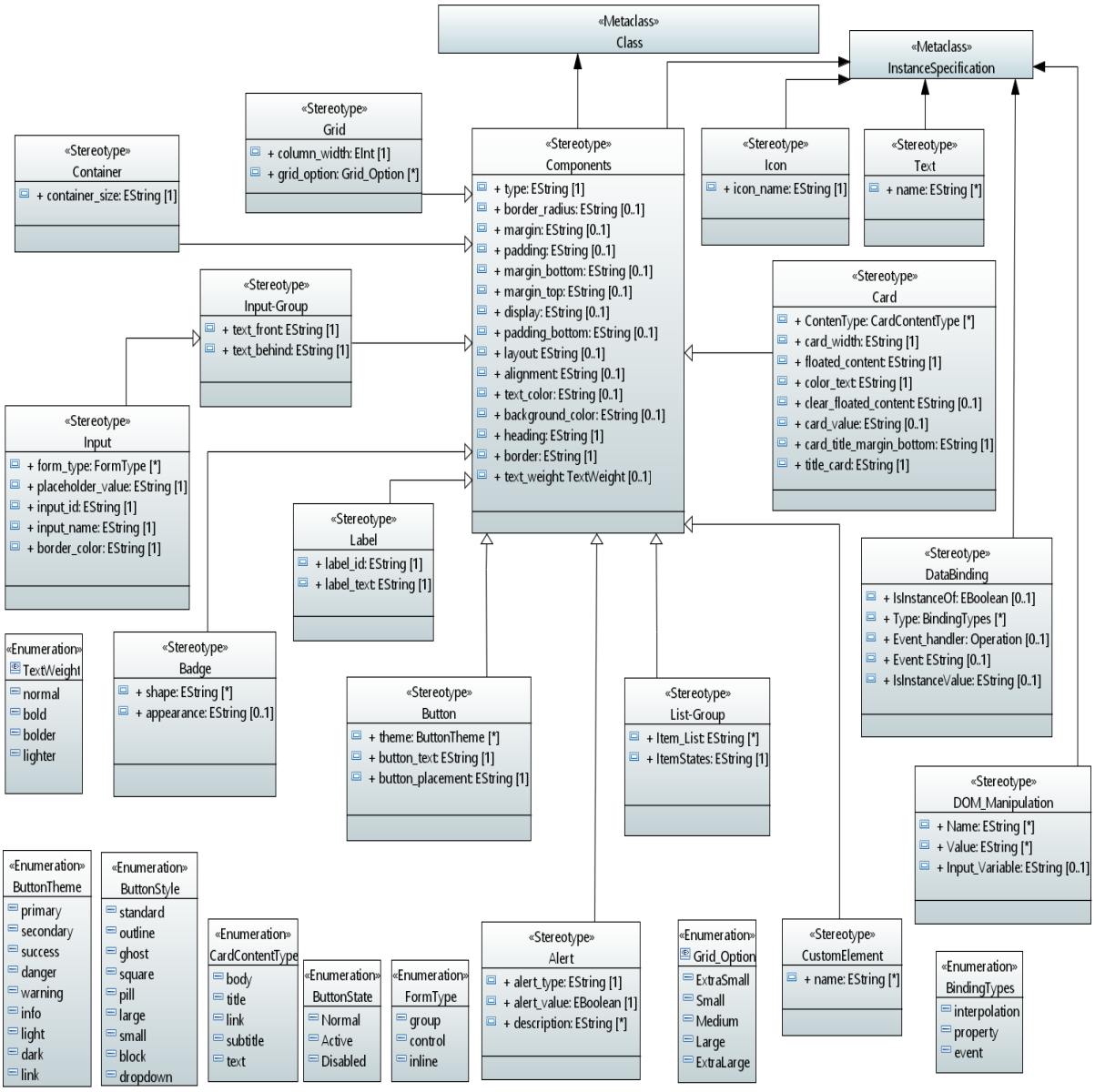


Figure 4: User Interface Concepts

5) Stereotype Components

Description: Stereotype *"Components*" defines the user interface related concepts. This stereotype sets the common properties which are present in all the components.

Base Class: Class, Instance Specification

Tag Values: The tag value *type* defines the type of component. Tag value *border_radius* specify the radius of the elements corner. The property *margin* and *padding* set the spacing of elements. The tag value *margin-top* and *margin-bottom* specifies the margin for top and bottom sides of an element. *Icon_font* property defines the glyphs used on the element. The tag value *display* defines the elements display property. Similarly, other tag values such as *alignment* sets the horizontal and vertical alignment of the element, *text_color* defines the color of the text, *background_color* specifies the background color, *heading* defines the heading and *text_weight* are also defined to specify the font weight. An enumeration class *TextWeight* is defined that contains three literal values which are normal, bold, bolder and lighter. The component specific values are defined in their corresponding stereotypes.

6) Stereotype Container

Description: Stereotype "*Container*" is used to center and horizontally pad the contents of the site.

Base Class: Class, Instance Specification

Tag Values: The tag value *container_size* defines the width of the viewport and is of type String.

7) Stereotype Grid

Description: Stereotype "*Grid*" defines the layout and align contents which is placed inside the column.

Base Class: Class

Tag Values: The tag value *column_width* defines the width of the column in percentages. Tag value *grid_option* specifies the sizes in pixels. Enumeration class Grid_Option contains the screen sizes which are Small, Medium, Large, Extral Small and Extra Large.

8) Stereotype Card

Description: Stereotype "*Card*" defines an element that is used to display content.

Base Class: Class, Instance Specification

Tag Values: The tag value *Cardtype* defines the content type supported in the card including text, images, links etc. The enumeration class *CardContentType* defines several enumeration literals. The enumeration literal body is the basic building block of card. Further, there are

other card content types such as titles, links, text etc. The tag value *card_width* defines the width of the card. Tag value *floated_content* defines the positioning and formatting content. *Clear_floated_content* property is used to clear the floated content. *Color_text* specifies the text color on card. *Card_title_margin_bottom* define the bottom margin. The tag value *title_card* defines the card title.

9) Stereotype InputGroup

Description: Stereotype "*Input-Group*" adds text, button or button groups by extending form controls.

Base Class: Class, Instance Specification

Tag Values: It contains two tag values: text_front and text_behind. The tag value *text_front* is used to add the help text in front of input while *text_behind* specifies the text behind the input.

10) Stereotype Input

Description: Stereotype "*Input*" is used to specify a form control for input.

Base Class: Class, Instance Specification

Tag Values: It encompasses following tag values: form_type, placeholder_value, input_id, input_name, border_color. The tag value *form_type* is of enumeration type so enumeration class FormType is defined which contains three literal values such as group, control and inline. The literal value group add structure to forms, literal value control add style to the form elements. The literal value inline defines the layout of the form. *Input_id* defines the id for the input element. *Input_name* specifies the name of input element. We can set the color of the border using *border_color* property.

11) Stereotype Button

Description: Stereotype "*Button*" specifies the button element of the user interface.

Base Class: Class, Instance Specification

Tag Values: It comprises of several tag values that are used to define the style of the button. The tag values *theme* specifies the button theme which is of enumeration type, *button_text* defines the text to be placed on the button, *button_placement* defines the button position. Three enumeration classes for button are defined. Enumeration class *ButtonTheme* contains the button styles for actions in forms, dialogs etc. *ButtonStates* defines the state of the button which can be in one of three states: Normal, Active and Disabled.

12) Stereotype Badge

Description: Stereotype "*Badge*" defines a link or button to provide a counter.

Base Class: Class, Instance Specification

Tag Values: It contains two tag values: shape and appearance. The tag value *shape* defines the shape of the badge e.g rounded etc. The tag value *appearance* specifies the appearance.

13) Stereotype Alert

Description: Stereotype "*Alert*" is used to define feedback messages for typical user actions.

Base Class: Class, Instance Specification

Tag Values: It consists of three tag values namely alert_type, alert_value, description. *Alert_type* defines the type of alert. An enumeration AlertType is defined having literal values primary, success, danger, warning, info, light and dark. The tag value *alert_value* defines the value to be true or false for an alert and description defines the description.

14) Stereotype ListGroup

Description: Stereotype "List-Group" define the series of contents.

Base Class: Class, Instance Specification

Tag Values: The tag value *Items_list* defines the items to be added in a list. Tag value *ItemStates* defines the item states. An enumeration ItemStates is defined that consist of three literal values for list items namely Active and Disabled. Active is used to indicate current active selection. Disable makes the item appear disabled.

15) Stereotype CustomElement

Description: Stereotype "*CustomElement*" defines a new fully feature DOM element. The tag value name specifies the custom element name.

16) Stereotype DataBinding

Description: Stereotype "DataBinding" controls the data flow in the application between a component and its view template.

Base Class: Instance Specification

Tag Values: The tag values defined in this stereotype presents binding type with the tag value of enumeration data type. IsInstanceOf is a flag that is set to bind the html control to

application data. Event_handler defines an operation on which data is to be bind. Event is the occurrence of certain action.

17) Stereotype DOM_Manipulation

Description: Stereotype "DOM_Manipulation" is used to bind application data to the attributes of HTML DOM elements.

Base Class: Instance Specification

Tag Values: It has three tag values Name, Value and Input Variable which are used to change the layout of the application or change the behavior of the DOM element.

3.1.3 Serverless Schema Concepts

Figure 5 shows the concepts defined for serverless schema modelling. For schema, type and fields on those types are defined for data and then functions for each field on each type is provided.

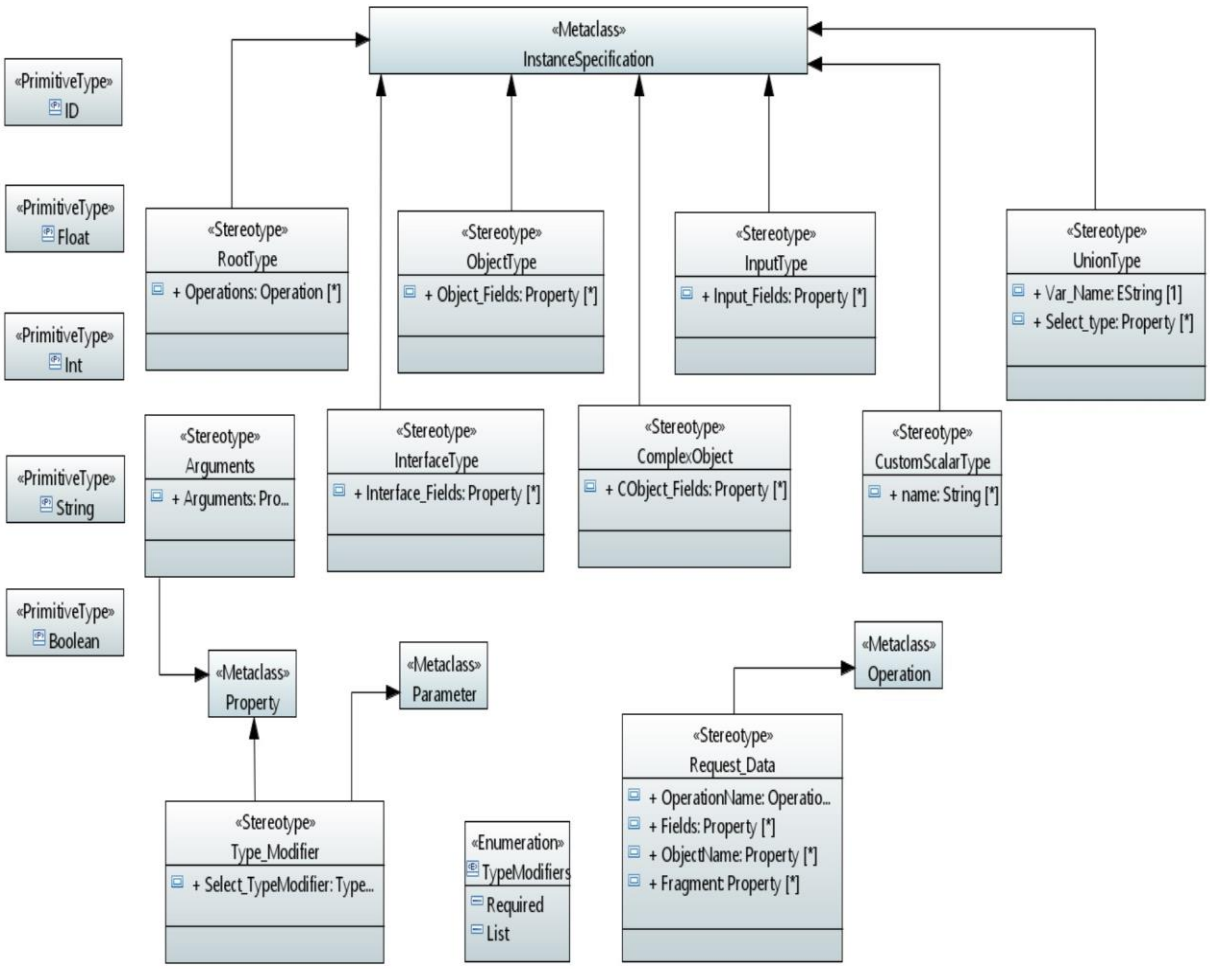


Figure 5: Serverless Schema Concepts

18) Stereotype RootType

Description: Stereotype "*RootType*" is specified to define entry point of an application.

Base Class: Instance Specification

Tag Value: The tag value consists of Operations which are used for data fetching. Root operations such as read, write or subscribe are taken through this stereotype.

19) Stereotype ObjectType

Description: Stereotype "ObjectType" defines an object we can fetch from our defined service.

Base Class: Instance Specification

Tag Value: The tag value Object_Fields define the fields on that object through property type.

20) Stereotype InputType

Description: Stereotype "*InputType*" is defined having tag value Input_Fields that specify the fields for the input type.

Base Class: Instance Specification

21) Stereotype InterfaceType

Description: Stereotype "*InterfaceType*" is an abstract type that includes a certain set of fields which a type must include to implement the interface.

Base Class: Instance Specification

Tag Value: The fields in the interface type are defined with the tag value Interface_Fields through property type.

22) Stereotype UnionType

Description: Stereotype "UnionType" does not specify any fields.

Base Class: Instance Specification

Tag Value: It comprises of two tag values. The tag value Var_name specified the name for the union type. Select_type defines the concrete object types.

23) Stereotype CustomScalarType

Description: Stereotype "*CustomScalarType*" specifies the custom scalar type.

Base Class: Instance Specification

Tag Value: The tag value name defines the custom scalar type name.

24) Stereotype TypeModifier

Description: Stereotype "*Type_Modifier*" defines the special group of types.

Base Class: Property, Parameter

Tag Value: It has one tag value *Select_TypeModifier* of type enumeration. This attribute selects the desired modifier type from the enumeration. An enumeration class *TypeModifier* is defined having two literal values: Required and List. Required presents a type as non-null by adding an exclamation mark after the type name. List indicates that the field will return an array of that type. This stereotype extends meta-class property and meta-class parameter.

25) Stereotype Arguments

Description: Stereotype "Arguments" contain the arguments within the fields or operations.

Base Class: Property

Tag Value: The tag value *Arguments* take the arguments used in the fields.

26) Stereotype Request_Data

Description: Stereotype "*Request_Data*" defines the requested data operation in the form of data structure which is used to communicate with backend.

Base Class: Operation

Tag Value: It encompasses four tag values. *OperationName* specifies the name of the operation. Fields define the data which must be fetched. *ObjectName* defines the nested data and Fragment can construct set of fields.

Five **primitive types** have been defined which represents scalar type. These are ID, Int, Float, String and Boolean.

3.1.4 Behavioral Concepts of Application

Figure 6 shows the modelling of data management concepts in a state machine such as data access and data synchronization.

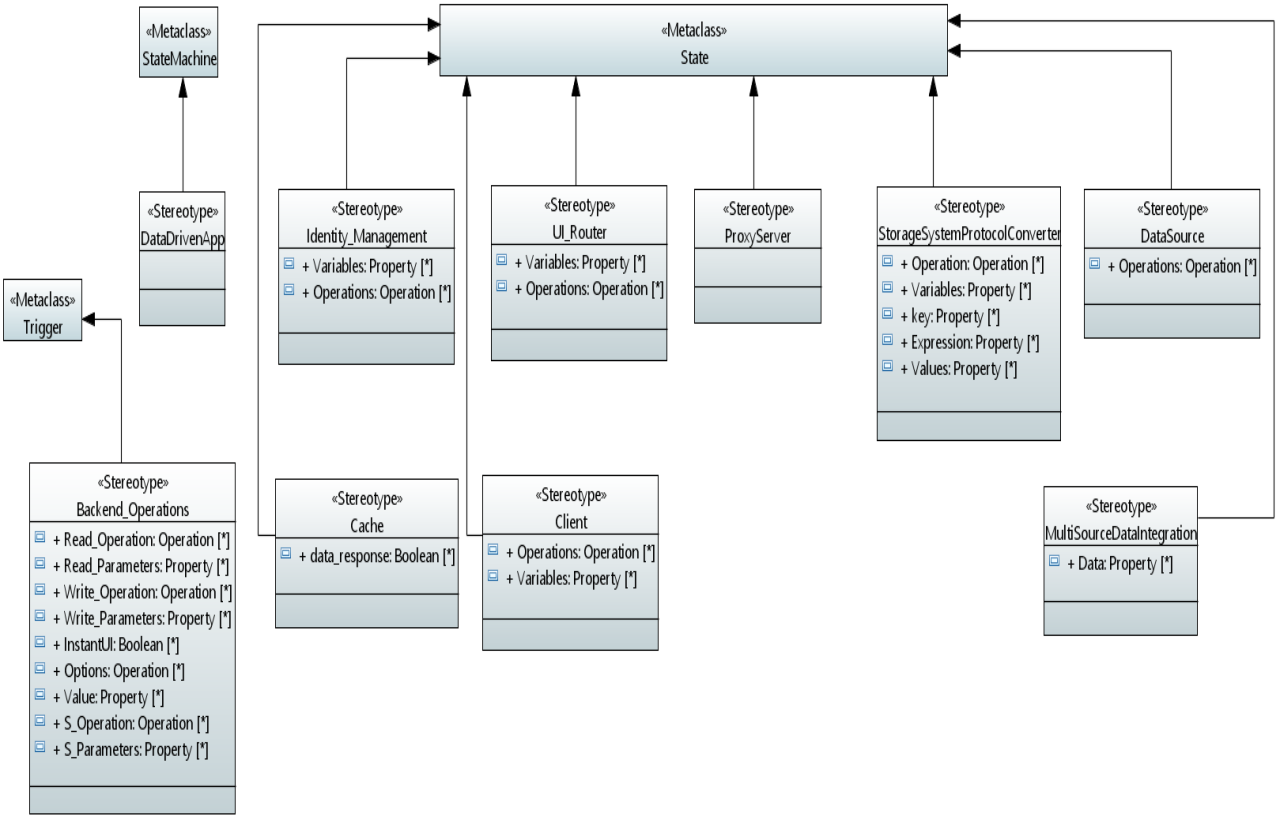


Figure 6: Data Management Concepts

27) Stereotype Identity Management

Description: Stereotype "IdentityManagement" is responsible for the authentication. This stereotype authenticates users to secure and protect data.

Base Class: State

Tag Value: To manage the identities tag value *Variables* and *Operations* are used to define the different variables and operations used for providing functionality for managing identity.

28) Stereotype UI_Router

Description: Stereotype "UI_Router" manages the transitions between states of an application.

Base Class: State

Tag Value: The tag value Variable defines the attributes of an operation through property type and the tag value Operations specify the operation.

29) Stereotype Client

Description: Stereotype "*Client*" is defined to communicate with the backend data sources and binding data to the frontend user interface element.

Base Class: State

Tag Value: The tag values Variables and Operations define the logic for data communication.

30) Stereotype Cache

Description: Stereotype "Cache" stores the data and updates the cache on arrival of new data.

Base Class: State

Tag Value: The tag value data_response is defined that is true if data is found in the cache.

31) Stereotype Proxy Server

Description: Stereotype "*Proxy Server*" handles client request.

Base Class: State

Tag Value: The tag value endpoint is defined on which all requests should be directed.

32) Stereotype StorageSystemProtocolConverter

Description: Stereotype "*StorageSystemProtocolConverter*" defines the resolver function which converts the payload data to the underlying storage system protocol.

Base Class: State

Tag Value: It includes the following tag values: Operation, Variables, key, Expression and Values. *Operation* specifies the NoSQL database operation to be performed. *Variables* define the remaining attributes of the item to be put into NoSQL database. *Key* denotes the key of the item in the NoSQL database. *Expression* defines the query expression. Values defines the substitution for expression attribute value. The key matches to a value placeholder used in the expression. The value must be a typed value.

33) Stereotype MultiSourceDataIntegration

Description: Stereotype "*MultiSourceDataIntegration*" combines data from multiple data sources.

Base Class: State

34) Stereotype Backend_Operations

Description: **S**tereotype "*Backend_Operations*" is defined which handles the backend operations.

Base Class: Trigger

Tag Value: It have the following tag values: *Read_Operation* of type Operation, *Read_Parameters* of type Property, Write_Operation of type Operation, Write_Parameters of type Property, InstantUI of type Boolean, Options of type Operation, Value of type Property, S_Operation of type Operation and S_Parameters of type Property. Write_Operation specifies the write request followed by a fetch. Write_Parameters are the parameters used in the write request. Read_Operation specifies the read only fetch. Read_Parameters define the parameters for read request. InstantUI is a Boolean variable that gives an instant response in low connectivity or offline mode. Options specify the operation for fetching data. Value defines the data to be fetched. S_Operations is the long-lived connection for receiving data. S_Parameters describe parameters for real time update.

Chapter 4

Implementation

CHAPTER 4: IMPLEMENTATION

4.1 Transformation Rules

This section discusses the transformation rules which are used to map UMSDA into frontend and backend code. These rules produce a target frontend and backend code for data synchronization in a client application through transformation engine. For UMSDA models, Papyrus modeling editor of Eclipse is used. For validation, the code is deployed on AWS Serverless platform using AWS SAM (Serverless Application Model).

4.1.1 Transformation Rules for Application Structure

The transformation rules for transforming UMSDA Concept into Angular Code are shown in Table 1. Rules for main application module, application component and routing are provided in this section.

Table 1: Transformation rules for Application Structure Code

Sr.No	UMSDA Concept	Corresponding Angular Concept	Description
1.	**ApplicationModule**	RootModule	ApplicationModule→RootModule Module_declaration→declarations Module_imports→imports
2.	**ApplicationComponent**	Component Decorator	ApplicationComponent→Component Decorator Selector_Component→selector Url_Template→templateUrl
3.	**Routing**	Routing Module	Routing→Routing Module Path→path Component→component

4.1.2 Transformation Rules for User Interfaces

Table 2 highlights the transformation rules for transforming UMSDA Concept into corresponding Angular Html Template concept.

Table 2: Transformation rules for User Interface Code

Sr.No	UMSDA Concept	Corresponding Html Concept	Description
1.	**Container**	viewport	Container→viewport
2.	**Grid**	grid rows column	Grid→grid row→row column→column col_width→width grid_option→grid breakpoints
3.	**Input-Group** **Input**	input-group input	Input-Group→input-group Input→input text_front→input-group-prepend text_behind→input-group-append form_type→form-control placeholder_value→required placeholder input_id→id input_name→name border_color→borders
4.	**Button**	button	Button→button theme→button styles button_text→text
5.	**List-Group**	list-group	List-Group→list-group Items_List→list-group-item ItemStates→list-group-item→Active Items
6.	**Card**	card	Card→card ContentType→Content types Card_width→card width title_card→ card title floated_content→Float clear_floated_content→Clearfix border_color→border color color_text→Text color

7.	**Badge**	badge	Badge→badge shape→Pill badges

4.1.3 Transformation Rules for Serverless Schema

The transformation rules for transforming UMSDA concepts into corresponding GraphQL Code is summarized in Table 3. Only RootType and ObjectType rules are discussed in this section along with the type modifiers and Request data operations.

Table 3: Transformation rules for Serverless Schema

Sr.No	UMSDA Concept	Corresponding GraphQL Code	Description
1.	**RootType**	Code contains the type name and its operations for the stereotype RootType.	RootType→ type Operation→Operation→name →owned parameter
2.	**RootType TypeModifier**	Code contains the type modifiers on parameters for the stereotype TypeModifier.	RootType→Operation→owned parameter→type modifier
3.	**ObjectType**	Code contains the type name operations for the stereotype ObjectType.	ObjectType→type
5.	**ObjectType Property**	Code contains fields for Object type.	ObjectType→Property→ Fields→name and datatype
6.	**ObjectType TypeModifier Property**	Code contains the type modifiers on fields for the stereotype TypeModifier	ObjectType→ Property→type modifier.

7.	**Arguments Property**	Code contains the arguments in the property for the stereotype Arguments	ObjectType→Property→Arguments
8.	**Request_Data Operation**	Code contains the operations which defines the data structure for the stereotype Request_Data.	Request_Data→Operation

4.1.4 Transformation Rules for Application Behavior

The transformation rules for transforming UMSDA concepts into corresponding Angular 2 and VTL (Velocity Template Language) code is shown in Table 4.

Table 4: Transformation rules for Application Behavior Code

Sr.No	UMSDA Concept	Corresponding Angular and VTL (Velocity Template Language) Code	Description
1.	State UI_Router	Code contains the component decorators for the stereotype UI_Router	UI_Router→Component
2.	State IdentityManagement	Code contains the Authentication for the stereotype IdentityManagement	IdentityManagement→CognitoUser Pool for token-based authentication.
3.	State Client	Code contains the state operations for the stereotype Client	Client → StateOperations (Client mapped to the state operations)
4.	State	Code contains the	StorageSystemProtocolConverter→

	StorageSystemProtocolConverter	resolver functions for the stereotype StorageSystemProtocol Converter	Resolver function with data conversion for requested data
5.	State DataSource	Code contains the operation to be performed for stereotype DataSource	DataSource → Database Table
6.	Transition Trigger Backend_Operations	Code contains the data operations for the stereotype Backend_Operations	Backend_Operations→Operations for the client operations to communicate.

4.2 Transformation Engine Architecture

After defining the transformation rules, we developed a transformation engine by implementing transformation rules to generate Angular 2 and GraphQL code from the models. The transformation engine architecture is shown in Figure 7. The tool used for model to text transformation is Acceleo. The transformation engine comprises of two key components: 1) Application Launcher and 2) Code Generator.

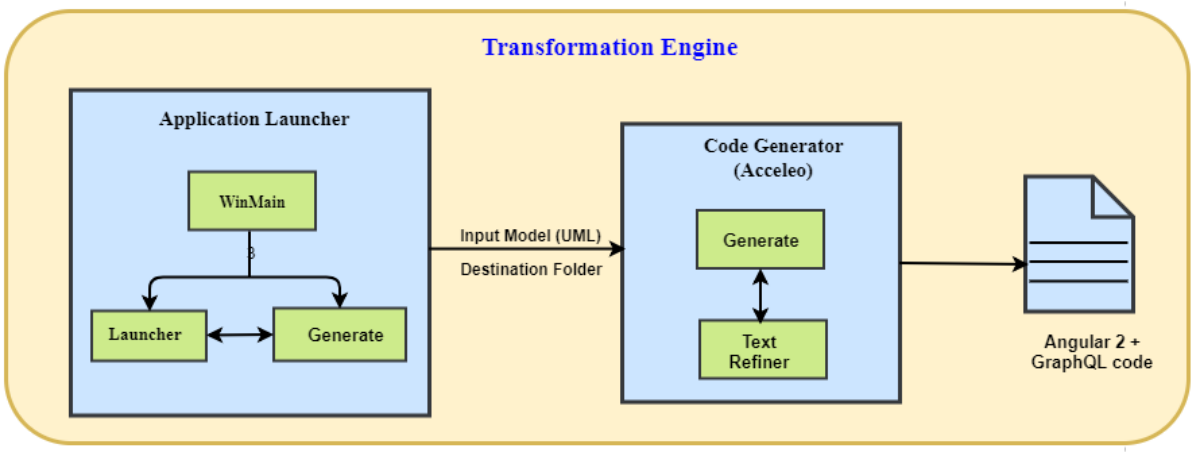


Figure 7: Transformation Engine Architecture

Application Launcher: The application launcher of the transformation engine consists of three main sub-components which are WinMain, Launcher and Generate. Particularly,

WinMain provides the transformation engine user interface implementation. Moreover, various settings required to execute transformation engine are implemented in Launcher sub-component. Additionally, Generate sub-component registers the profile addresses.

Code Generator: This component is responsible for generating Angular 2 and GraphQL code. The input model receives UML model and output folder receives the destination folder address (where the code files are to be generated) from the application launcher component. The Generate sub-component (Generate.mtl) extracts the desired element from the model. *Text Refiner* is a sub-component that deals with the formatting issues of the generated files. The interface for transformation engine is provide in **Figure 8**. Transformation Engine takes the UML model and path of output folder for generating the code from model using the browse button. Generate button is provided to generate the required outputs. Console shows the progress of the transformation process. A *Reset* is provided to clear all fields i.e. input model path, output folder path and console. *Close* button is provided to closes the interface from the screen.

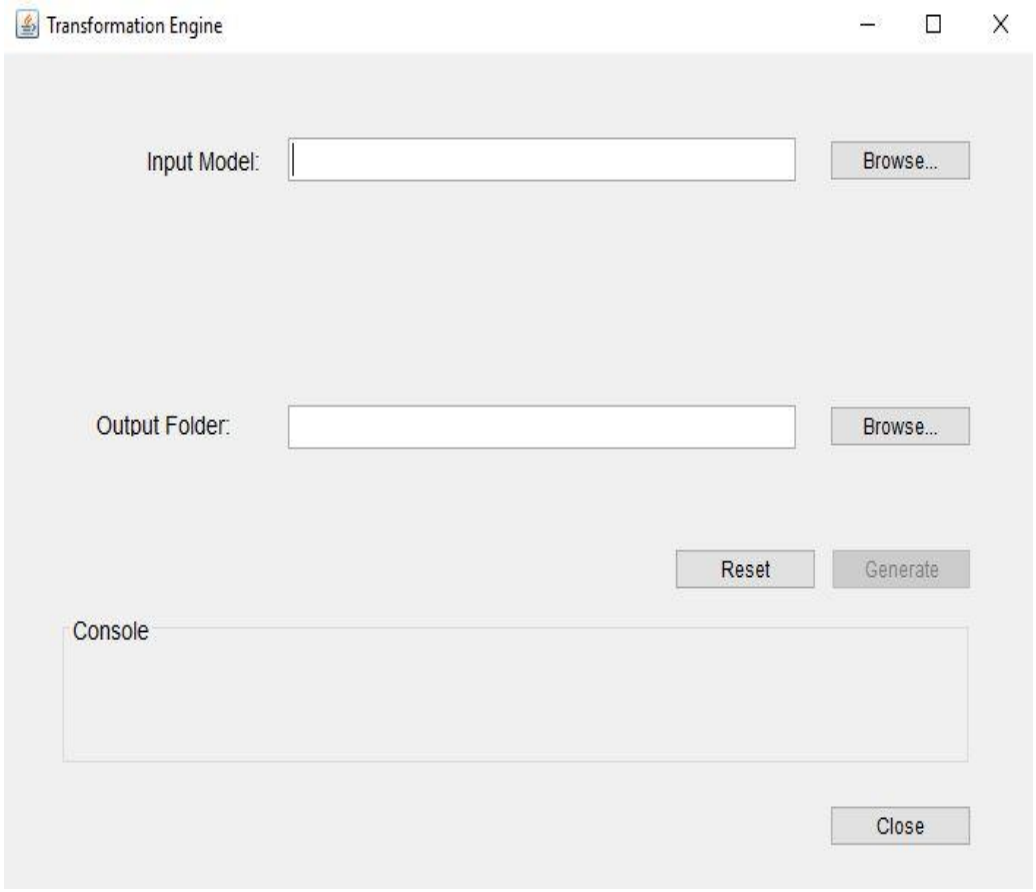


Figure 8: Transformation Engine Input Model Interface

Chapter 5

Validation

CHAPTER 5: VALIDATION

In this section, the applicability of our proposed framework is validated with the help of case studies. The case studies are discussed and documented in descriptive form. Two case studies are presented to validate our proposed work. The first one Real Time Chat Application is discussed and validated in **Section 5.1** and second one Weather Forecast Application is offered in **Section 5.2.**

5.1 Real Time Chat Application Case Study

This case study is divided into four sections. Firstly, the requirements of the real time chat application are discussed in **Section 5.1.1**. Secondly, **Section 5.1.2** contains the UML class diagram and state machine diagram with applied profile to present the system architecture of the required system. Furthermore, **Section 5.1.3** shows the transformation results in the form of generated code. And finally, **Section 5.1.4** contains validation of the system.

5.1.1 Requirements

For real time chat application, the high-level representation of the data requirements is shown in graph in Figure 9.

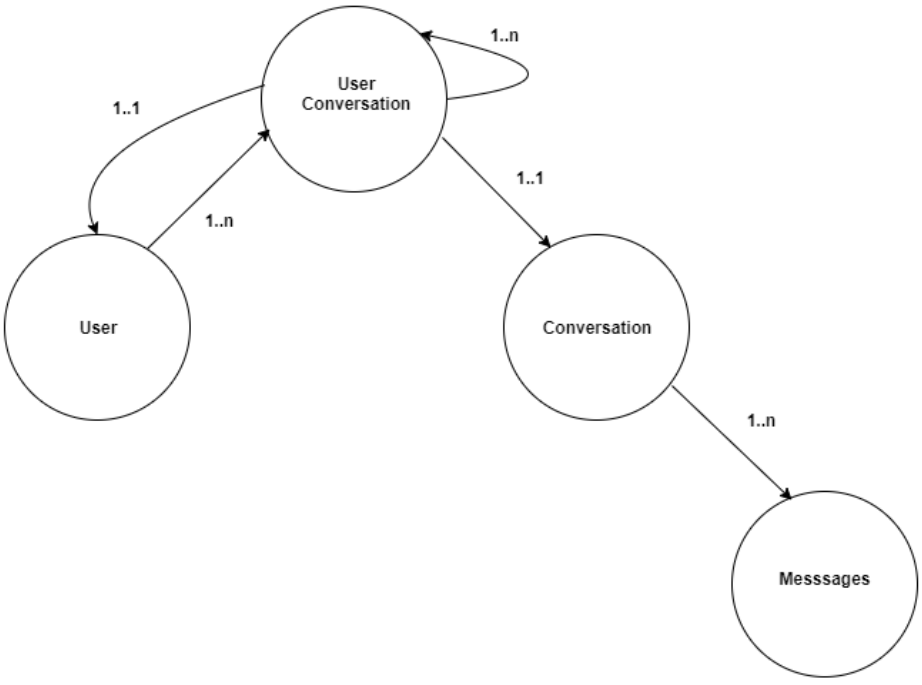


Figure 9: Data Requirements in Real-time Chat Application

The graph comprises of edges and nodes. For each connected set of nodes cardinality information is presented along the edges. The nodes represent the data that we are storing, and the edges represent associations between the data. The data is stored in DynamoDb (NoSQL) database of Amazon.

Users can see a list of all other users and can start conversations with any of them. However, they can only see conversations that they initiated or are invited to. To make it all work, the backend requirements includes the application behavior:

- **Authentication**

  The primary requirement of every application is authentication that allows the user/tenant to secure and protect data using API keys, Identity and access management or through cognito user pools. Users register their account through one of these options and then sign in to authenticate.
- **Authorization**

  After successful authentication, a JWT token is returned to the application that is used to identify the user and authorize access to backend sources.
- **Client**

  The client rapidly builds a UI that fetches data from the cloud. It is used with one of the view layer integrations. The client performs appropriate authorization wrapping of request statements before submitting to the server for synchronizing. Responses are persisted in an offline store and write requests are made in a write-through pattern. The client submits the operation request to the server along with an identity context and credentials.
- **Operations**
  - Read Requests consists of fetching list of users, related conversations, and messages
  - Write Requests create users, messages, conversations, and relations between users and conversations
  - Subscribe Requests automatically retrieve new messages in a conversation as soon as they are received in the backend
- **Schema**

  The schema consists of object types and root types which defines the data shape that flows and the operations that can be performed. This schema validates all the data operations.

- **Server Synchronization**

  The server processes the received requests and mapped them to logical functions for data operations or triggers. Then it passes this request to the Storage system protocol converter.

- **Storage System Protocol Converter**

  The storage system converter maps and executes the request payload against data source.

- **Data Source**

  A persistent storage system along with credentials for accessing that system. The state of the application is managed by the system or trigger defined in a data source.

The frontend requirements consist of:

- **ChatApplication Module**

  Chat application module is composed of several components such as User list, Conversation list, Messages, MessageView and Input component. Container and Grid element is used for layout.

- **Chat Application Components**

  The chat application components include:

  - User List Component

    User list component encompasses List Group element, Data binding and DOM manipulation.

  - Conversation List Component

    Conversation list component includes List Group, Data binding and DOM manipulation

  - Messages Component

    The messages component includes card and icon elements.

  - MessageView Component

    Message View component contains card element, data binding and DOM manipulation

  - Input Component

    Input component encompasses input group. Input group contains input and data binding. Button element is attached to the input element.

.

5.1.2 Modeling

A complete model including frontend and backend for Real-time Chat Application is provided. The model of Application frontend is shown in Figure 10.

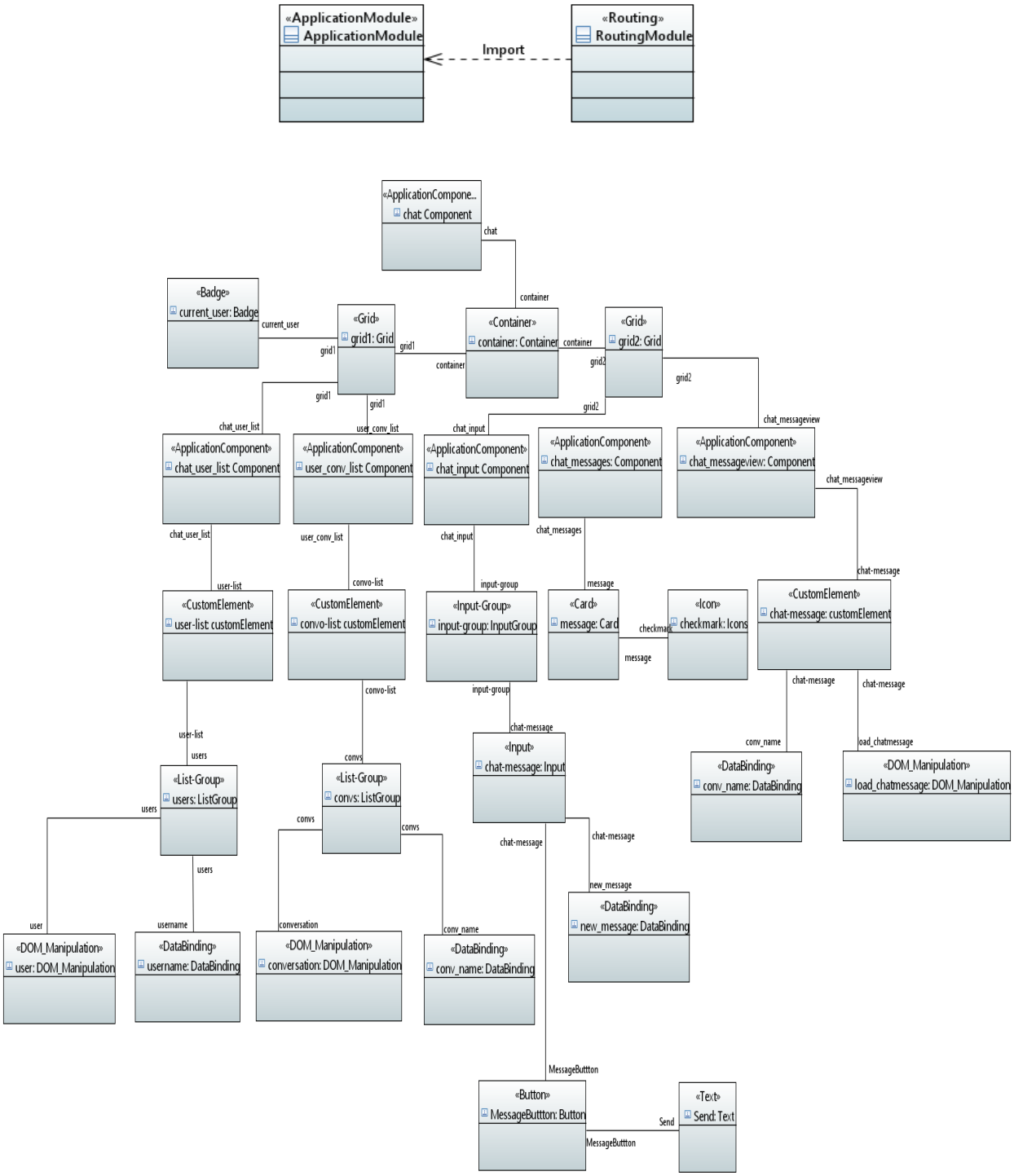


Figure 10: Application Frontend Model

In this model, <<ApplicationModule>> stereotype is applied to ChatAppModule which loads other parts of our application. <<Routing>> stereotype is mapped to ChatAppRoutingModule to tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar. <<ApplicationComponent>> stereotype is mapped to different components of the chat application which contains template, business logic and metadata. <<Container>> stereotype is mapped to container which provide a basic layout element. <<Grid>> is mapped to grid for layout and content alignment using containers, columns and rows. <<Badge>> is mapped to the badge element which create small labelling component. <<List-Group>> stereotype is mapped to the ListGroup which displays a series of contents. <<Input-Group>> is applied to InputGroup to provide extension to form controls by adding buttons, button groups or text on either side of custom selects, textual inputs etc. <<Card>> is mapped to card element for providing extensible content container with multiple variations and selections. <<Input>> is mapped to input which extend form control by adding text. <<Button>> stereotype is mapped to button which extend form control by adding button. For data binding <<DataBinding>> stereotype is used which handles communication between component and the DOM (Document Object Model). To change the DOM layout or behavior of an element <<DOM_Manipulation>> stereotype is used.

Figure 11 shows a model of an instance specification for serverless schema which is the backend of an application.

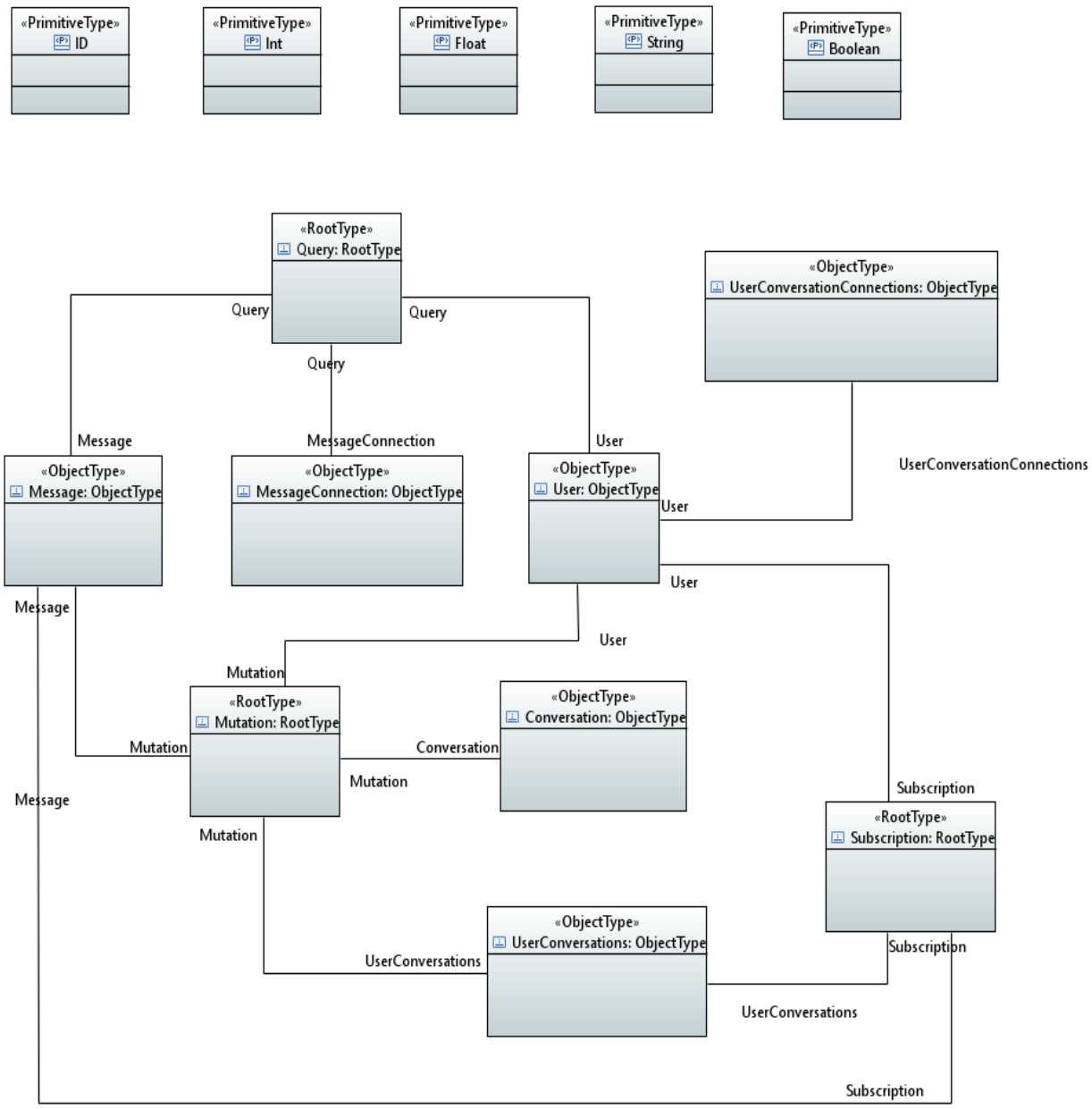


Figure 11: Model of Serverless Schema

In this model, <<RootType>> stereotype is applied to Root Types which defines the Root Operations (Read, Write and Subscribe). <<ObjectType>> is applied to Object Type which define fields of the object type. The type modifiers on property and parameters are also applied through <<TypeModifier>> stereotype. <<Request_Data>> stereotype is applied to Root Operations that define the shape of the data that flows.

Figure 12 shows a model of state machine diagram for data sync client in data-driven application.

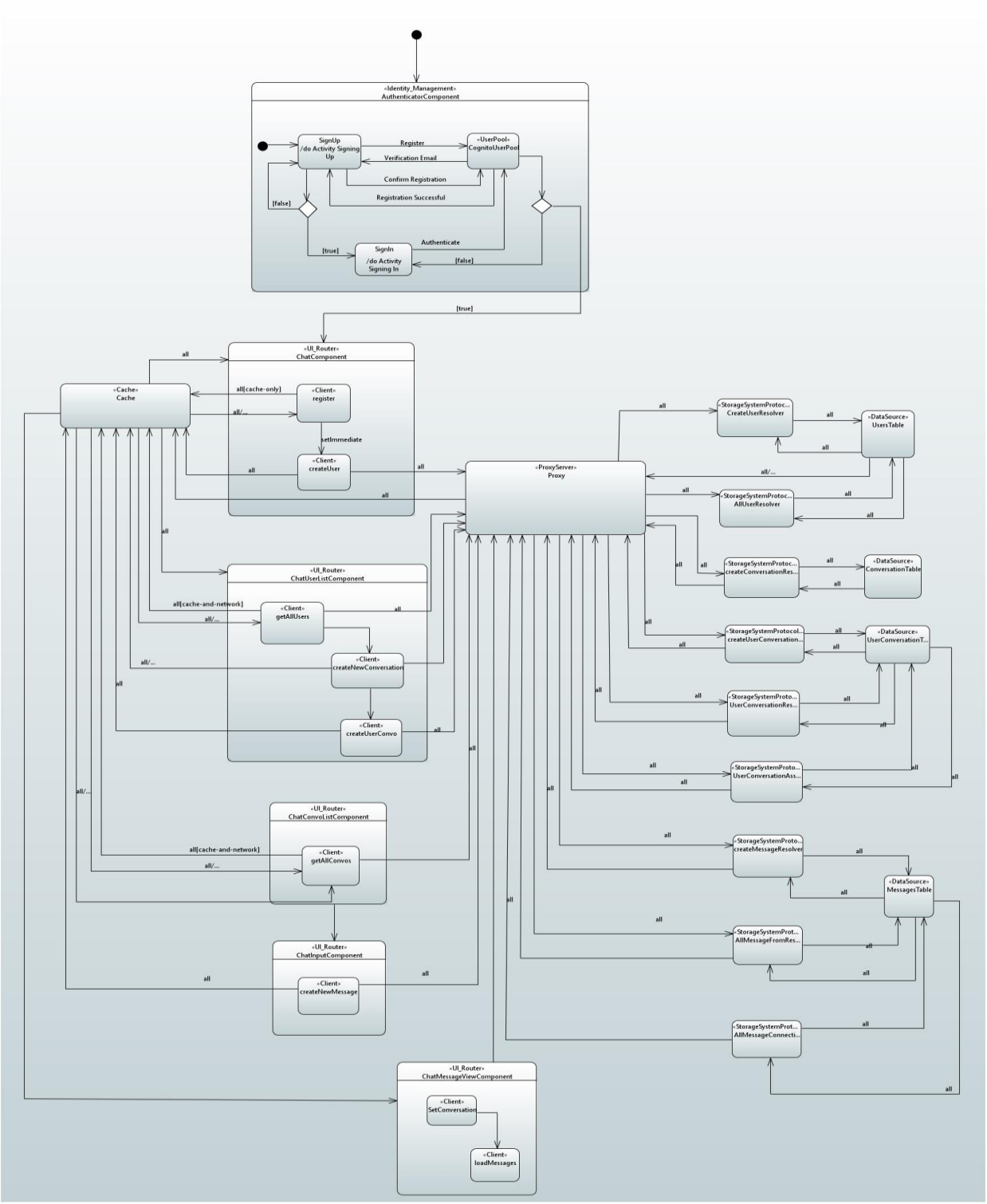


Figure 12: Model for Application Behavior (Data Synchronization)

The model shows the state machine behavior for the whole application. <<IdentityManagement>> is applied to AuthenticatorComponent which authenticates the user with Amazon Cognito User Pool. Users register their account and signs in. After

successful authentication, a JWT token is returned by Cognito to the application that is used to identify the user and authorize access to the application programming interface. <<UI_Router>> is mapped to the ChatComponent that manages the transition between application states. The ChatComponent has two states: register and createUser. <<Client>> stereotype is applied to both the states. It performs appropriate authorization wrapping of request statements before submitting to the server for synchronizing. Responses are persisted in an offline store (cache) and write requests are made in a write-through pattern. In the register state the transition is triggered on a get request to fetch the registered user from the cache. Therefore, cache-only condition is set to true. The cache checks for the registered user. If data is found, it subscribes to that data and updates the user interface. Then it enters the createUser state. A write request is made to the server along with an identity context and credentials. The request is passed to server when a transition is triggered. The server processes the received requests and mapped them to logical functions for data operations. It calls the createUser Resolver function that adds User to the data source by using <<StorageTransitionProtocolConverter>> stereotype. Once the user is created in a UsersTable, an event-based action is performed on the server that synchronizes the new user to the users list of all the subscribers connected to this application. In case a network is low or in offline mode the user gets updated data on network connectivity. If the user is using application in offline mode, instant UI is created but it does not communicate with the backend data sources. The request is saved and when the network goes online the request is automatically sent to the server. In ChatUserListComponent state, <<UI_Router>> stereotype is mapped. This state goes through three sub-states such as getAllUsers, createNewConversation, createUserConv. In getAllUsers state <<Client>> stereotype is applied, the transition is triggered on read request with cache and network condition equal to true. The simultaneous requests are made to cache and to the backend. If data is found in cache, a Boolean condition for data is true. When the request is sent to server. The server calls the getAllUsers Resolver using <<StorageTransitionProtocolConverter>> stereotype. It scans the database and response is sent to user. The cache is then updated by any new data received from the network. Similarly, for createNewConversation and createUserConv the same process is applied as for createUser request. When the user is invited from user list, the conversation connection is created between the two users and messages are exchanged between them. For each request the server calls the corresponding resolver that maps and execute the request payload against data source. In ChatInputComponent state, new messages

are created in createNewMessage state using <<Client>> stereotype. The transition is triggered on write request to server. The server calls the createMessageResolver that puts messages in the data source. New messages are subscribed, and our message list is updated. The same read request as dicussed earlier is applied to messageViewComponent.

5.1.3 Code Generation

This section highlights the code generation process from our proposed transformation engine. The model of data synchronization is given as input to our proposed transformation engine whose interface is shown in **Figure 13**. The transformation engine uses transformation rules for mapping and transforms the model into code. UML model with .uml extension is selected as input model and target folder on desktop is provided as output folder for generated code files. On clicking the generate button following outputs are generated in the target folder.

1. The code for frontend Application in Angular 2 as shown in **Figure 14, 15**.
2. The backend code in GraphQL (Graph Query Language) as shown in **Figure 16, 17**.

Console shows the progress of transformation process.

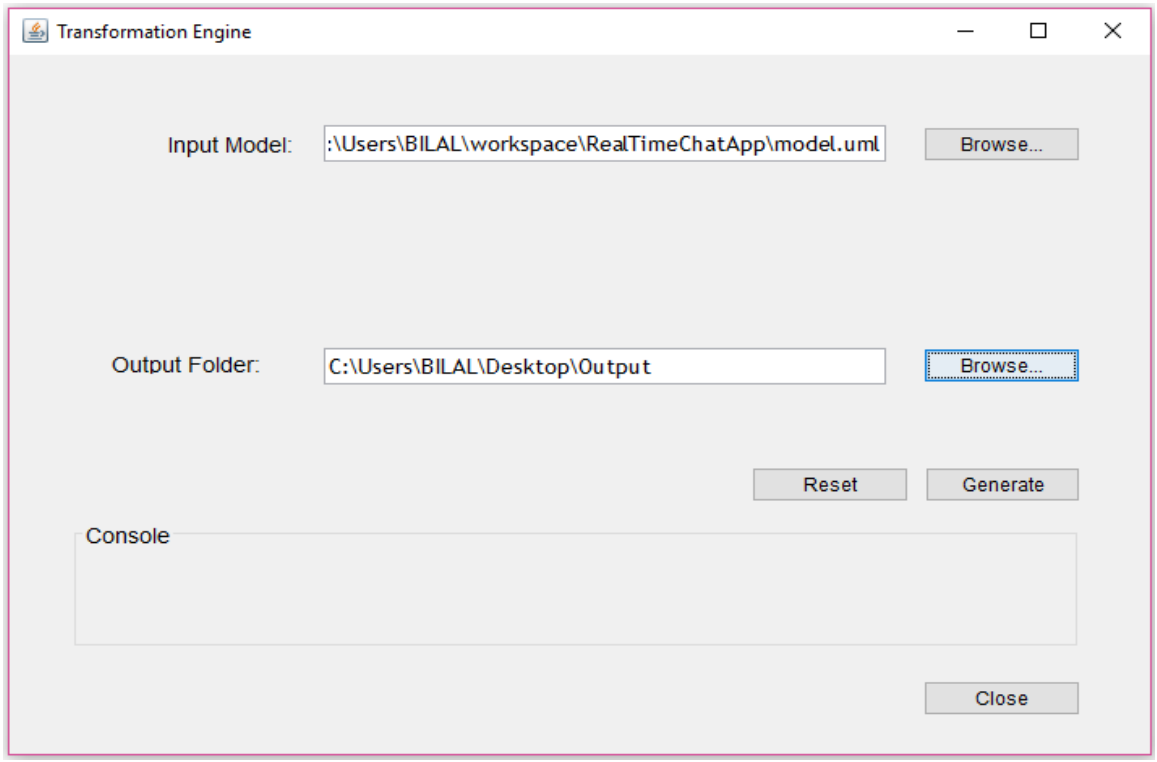


Figure 13: Transformation for Real-time Chat Application Code

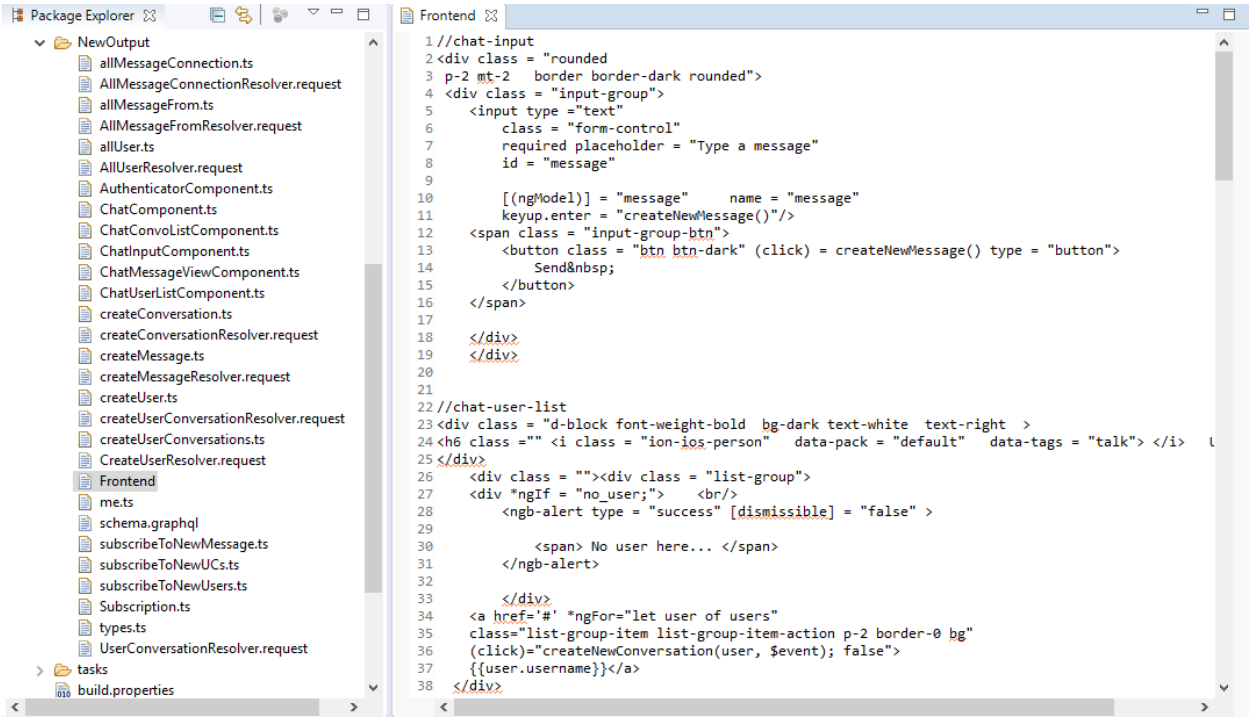


Figure 14: Generated Code for FrontEnd User Interface

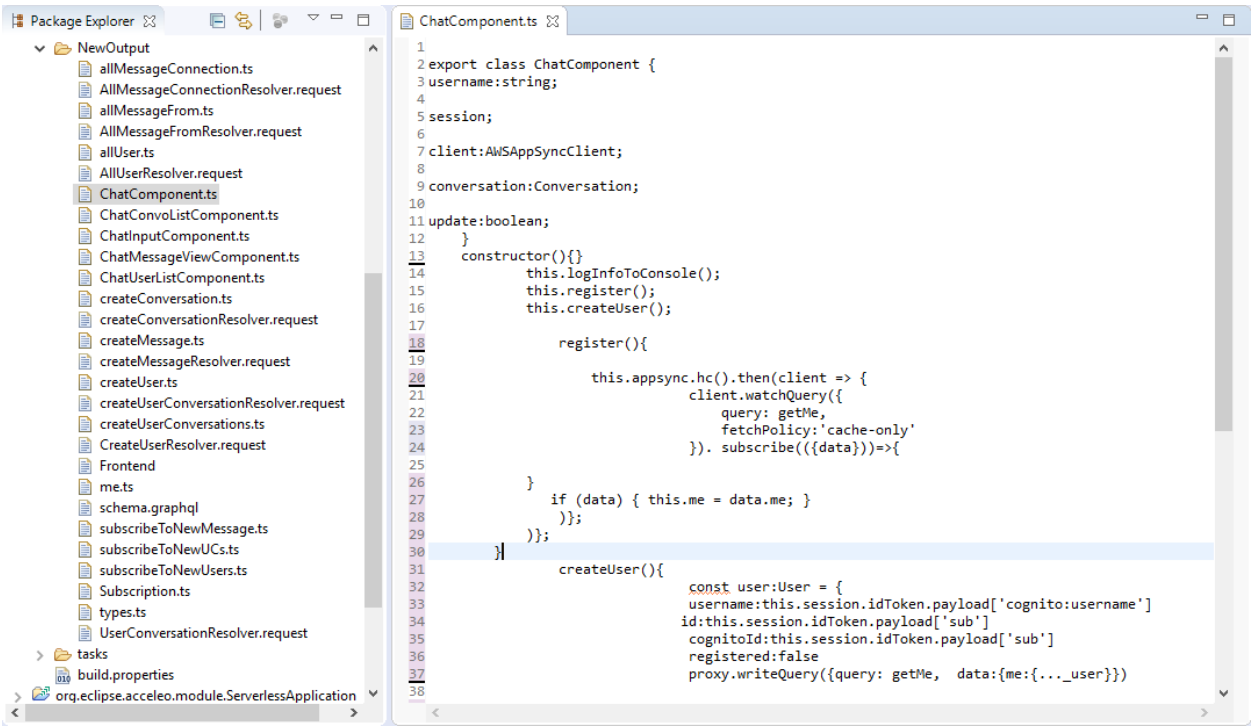


Figure 15: Generated Code for Data Synchronization Client

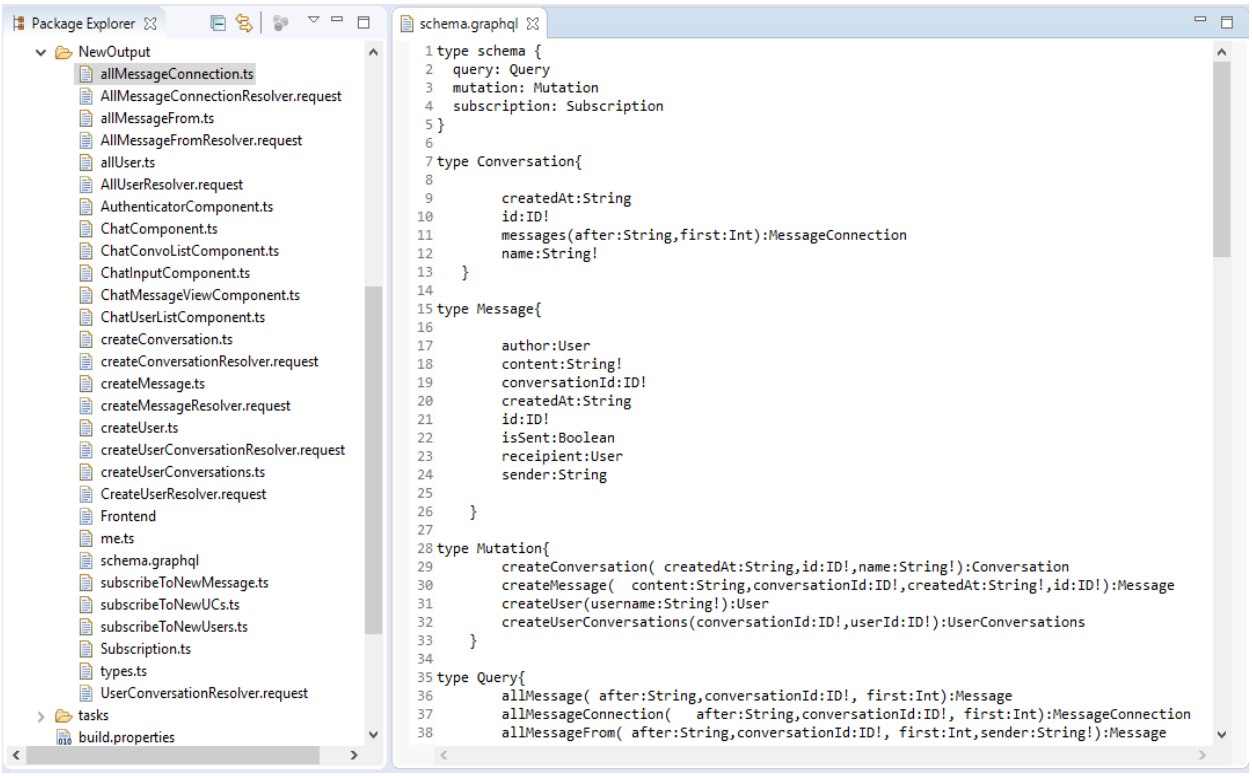


Figure 16: Generated Code for Serverless Schema

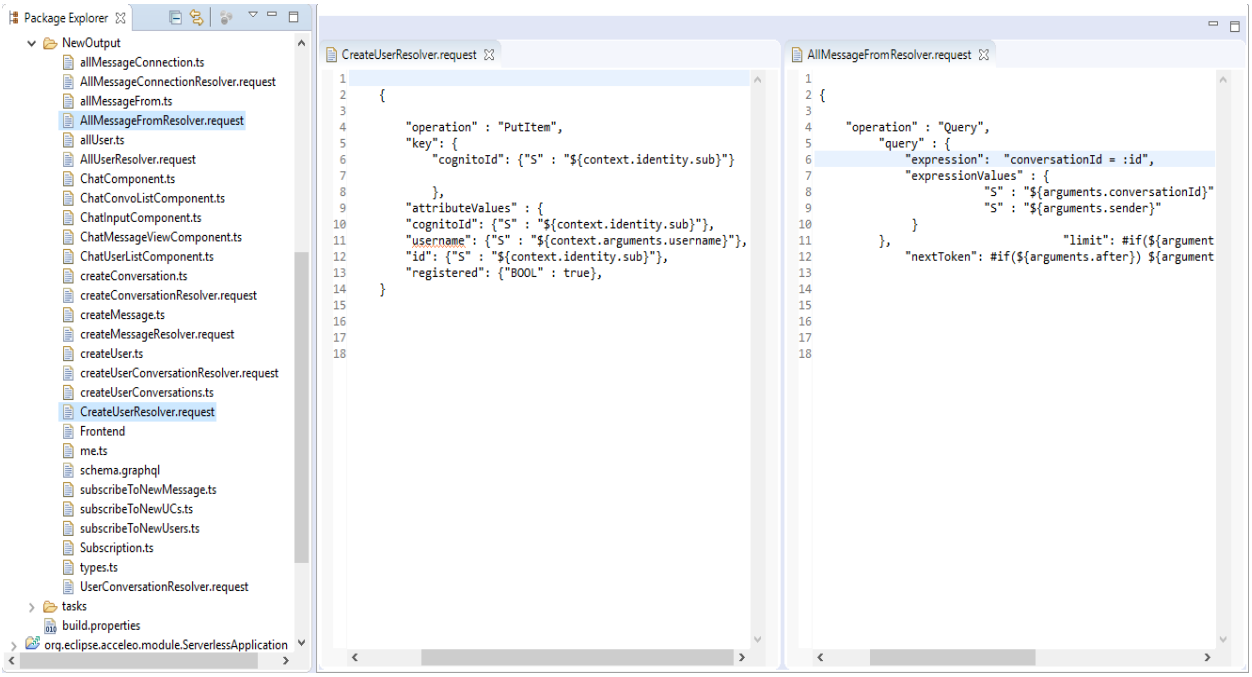


Figure 17: Generated Code for Data Resolvers

5.1.4 Verification

For verification of generated code, its compilation and execution are necessary. Therefore, we deploy this code in angular application. We created an angular application and pasted our generated code accordingly as shown in **Figure 18,19, 20 and 21.**

chat-input.component.html

```
<div class="rounded p-2 mt-2 bg border border-dark rounded">
  <div class="input-group">
    <input type="text" class="form-control no-focus"
    required placeholder="Type a Message"
    id="message" [(ngModel)]="message" name="message"
    (keyup.enter)="createNewMessage()" />
    <span class="input-group-btn">
      <button class="btn btn-dark" (click)="createNewMessage()" type="button">
        Send 
      </button>
    </span>
  </div>
</div>

```

Figure 18: Code Deployed for Real Time Chat Application (User Interface)

chat.component.ts

```
  createUser() {
    const user: User = {
      username: this.session.idToken.payload['cognito:username'],
      id: this.session.idToken.payload['sub'],
      cognitoId: this.session.idToken.payload['sub'],
      registered: false
    };
    console.log('creating user', user);
    this.appsync.hc().then(client => {
      client.mutate({
        mutation: createUser,
        variables: {username: user.username},

        optimisticResponse: () => ({
          createUser: {
            ...user,
            __typename: 'User'
          }
        }),

        update: (proxy, {data: { createUser: _user }}) => {
          // console.log('createUser update with:', _user);
          proxy.writeQuery({query: getMe, data: {me: {..._user}}});
        }
      }).catch(err => console.log('Error registering user', err));
    });
  }

  register() {
    this.appsync.hc().then(client => {
      client.watchQuery({
        query: getMe,
        fetchPolicy: 'cache-only'
```

Figure 19: Code Deployed for Real Time Chat Application (Data Sync Client)

schema.graphql

```
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

type Conversation {

  createdAt: String
  id: ID!
  messages(after: String, first: Int): MessageConnection
  name: String!
}

type Message {

  author: User
  content: String!
  conversationId: ID!
  createdAt: String
  id: ID!
  isSent: Boolean
  recipient: User
  sender: String
}

type MessageConnection {
  messages: [Message]
  nextToken: String
}

type Mutation {
  createConversation(createdAt: String, id: ID!, name: String!): Conversation
```

Figure 20: Code Deployed for Real Time Chat Application (Serverless Schema)

Mutation.createUser.request

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "cognitoId": { "S" : "${context.identity.sub}"}
    },
    "attributeValues" : {
        "cognitoId": {  "S": "${context.identity.sub}" },
            "username": {  "S": "${context.arguments.username}" },
        "id": {  "S": "${context.identity.sub}" },
        "registered": {  "BOOL": true }
    }
}
```

Figure 21: Code Deployed for Real Time Chat Application (Data Resolvers)

Validation of the application is done after deployment of code on AWS Serverless platform using local AWS command line interface as shown in Figure 22.

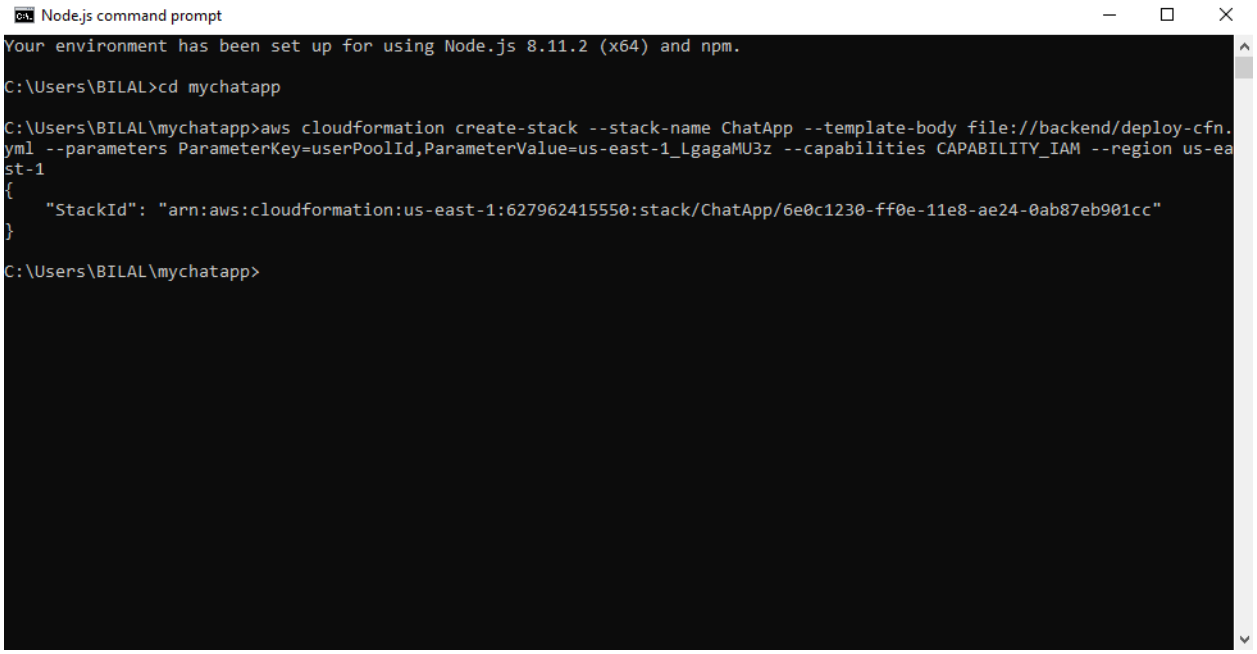


Figure 22: Deployment on AWS using Cloud Formation

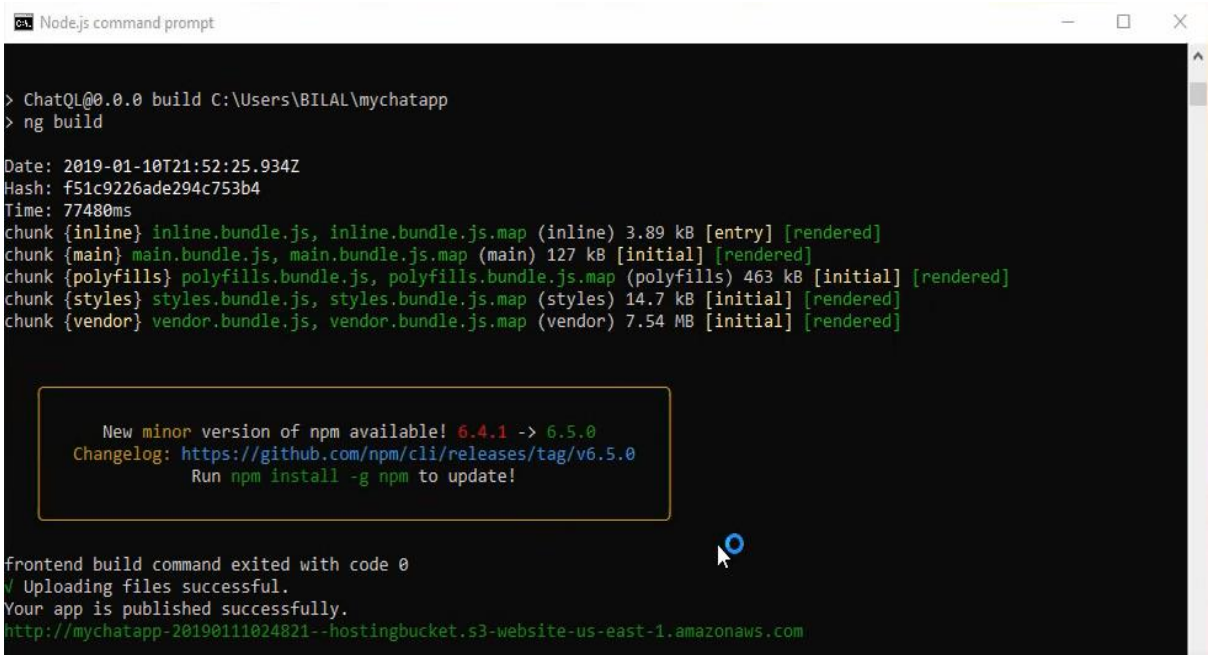


Figure 23: Client Application Deployed to cloud

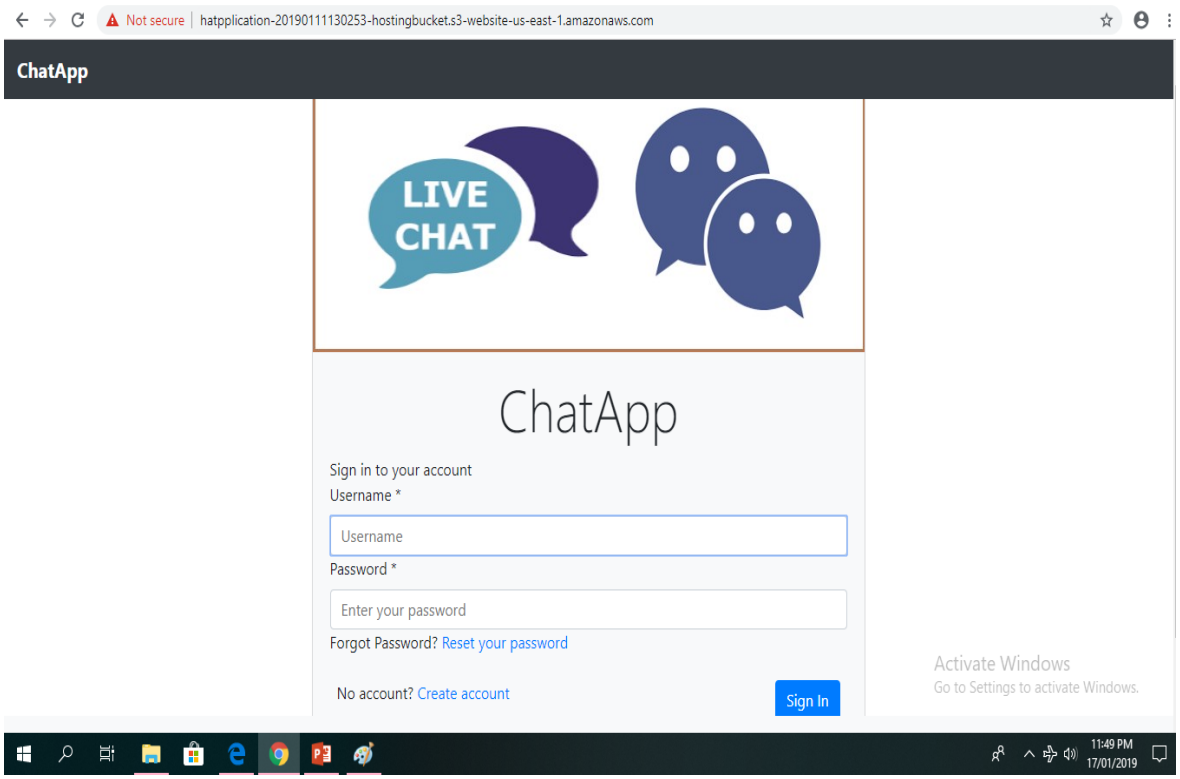


Figure 24: Transformation Result of Chat Application (Main Page)

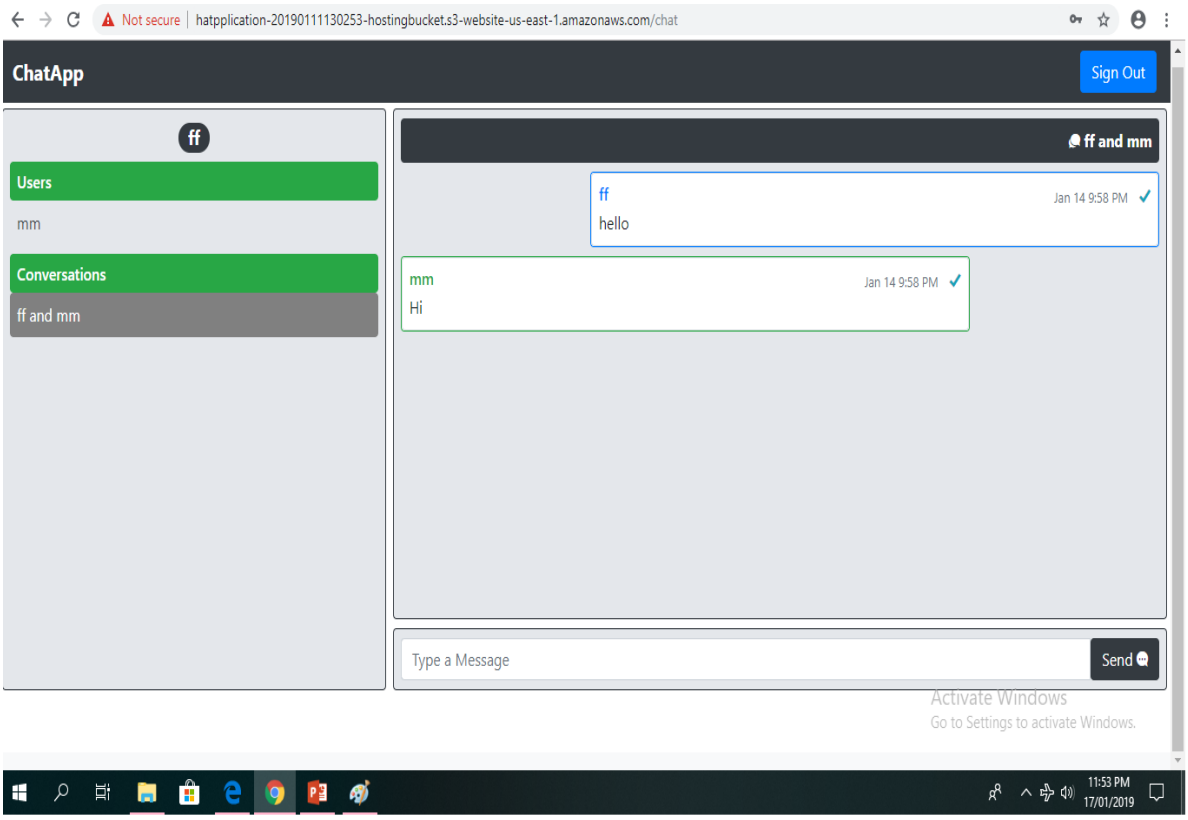


Figure 25: Chat Components with Behavior (User 1)

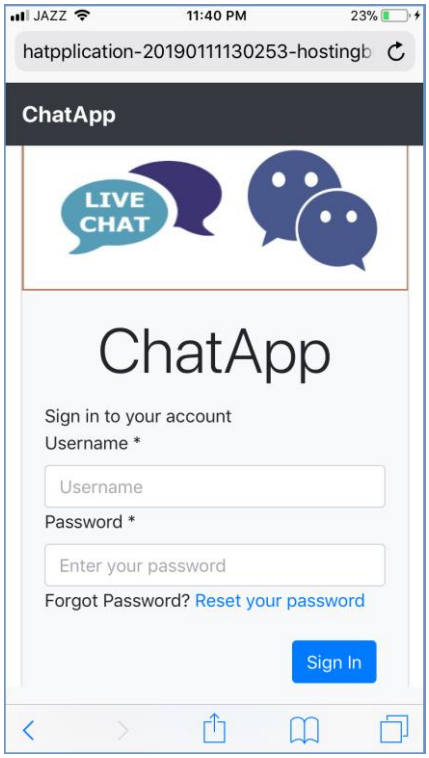


Figure 26: Transformation Result of Chat Application (Mobile Web)

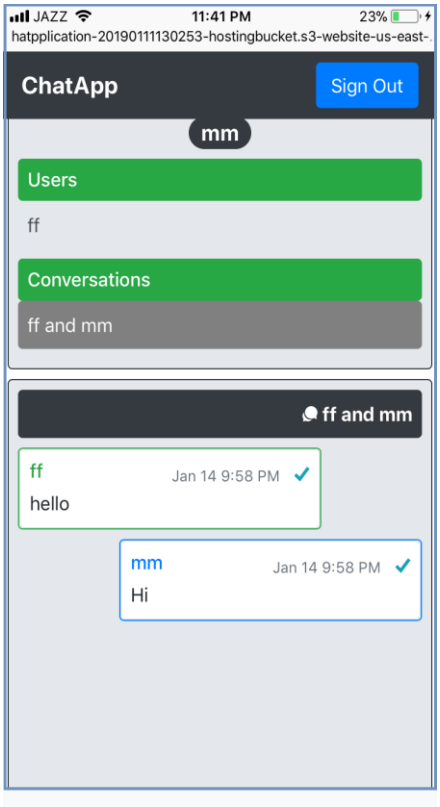


Figure 27: Chat Components with Behavior (User 2)

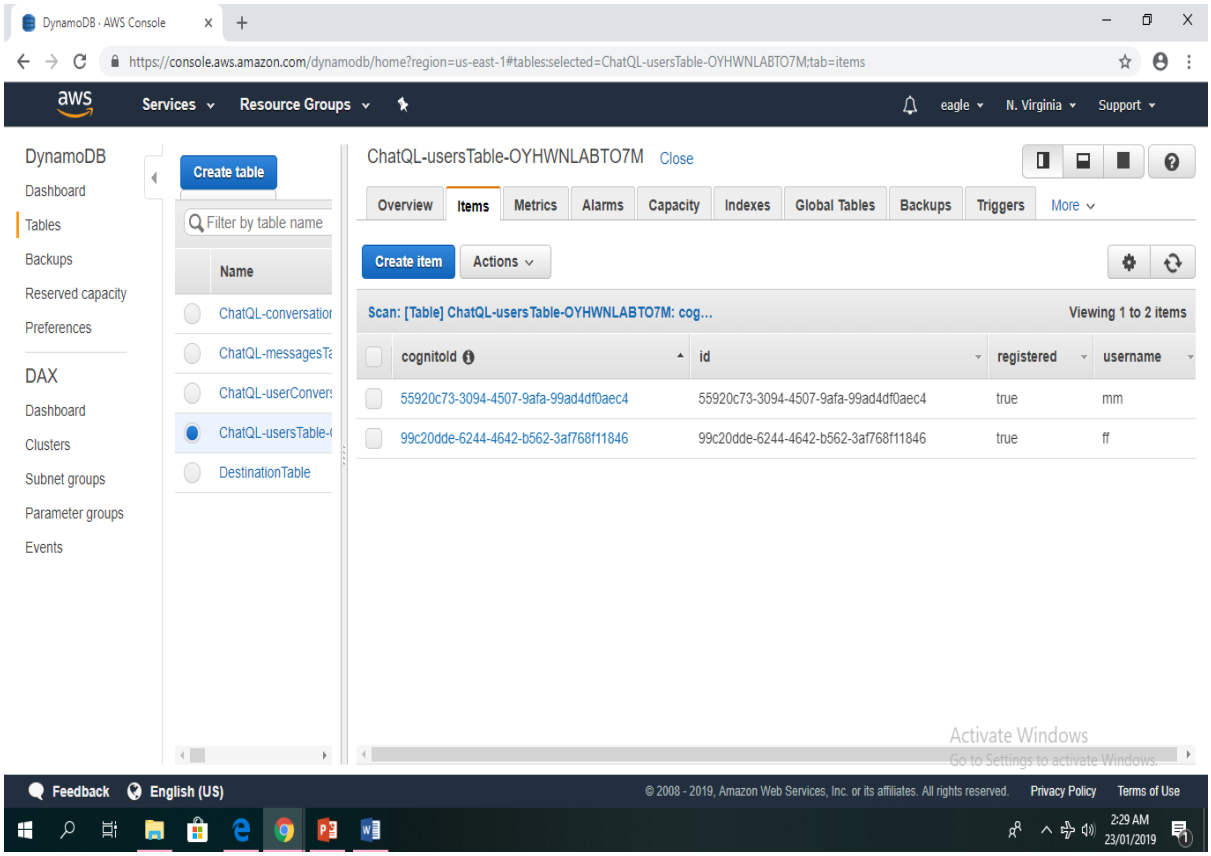


Figure 28: Users Table Amazon DynamoDb

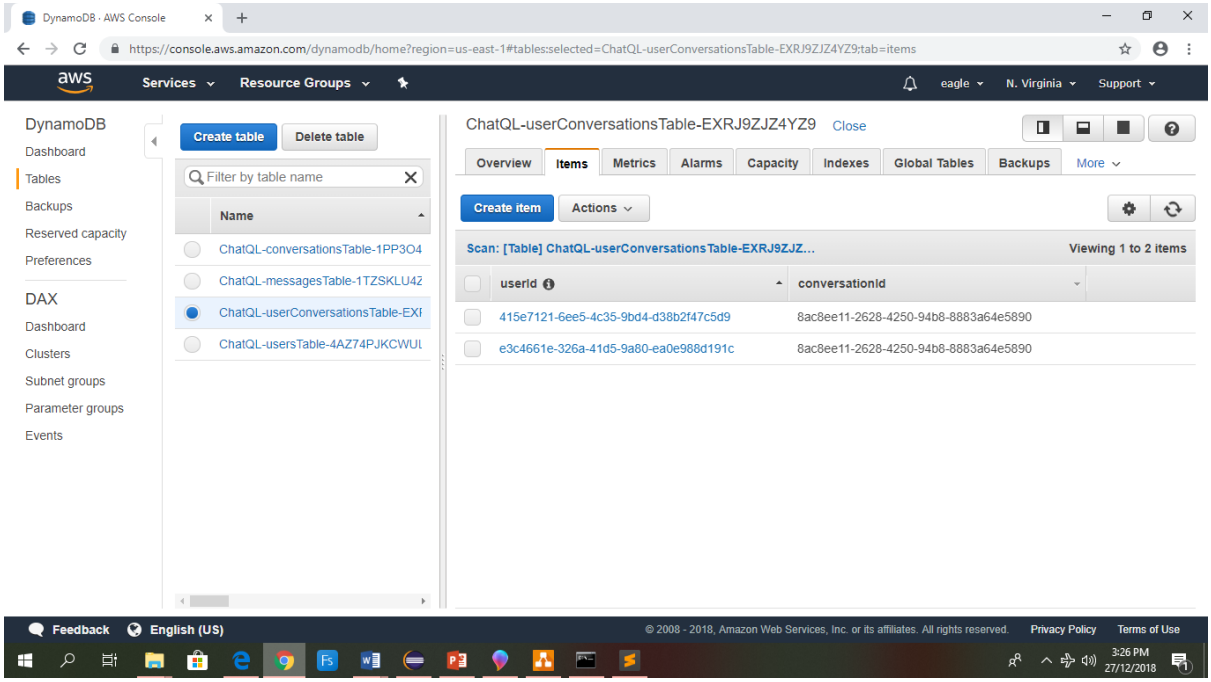


Figure 29: User Conversations Table

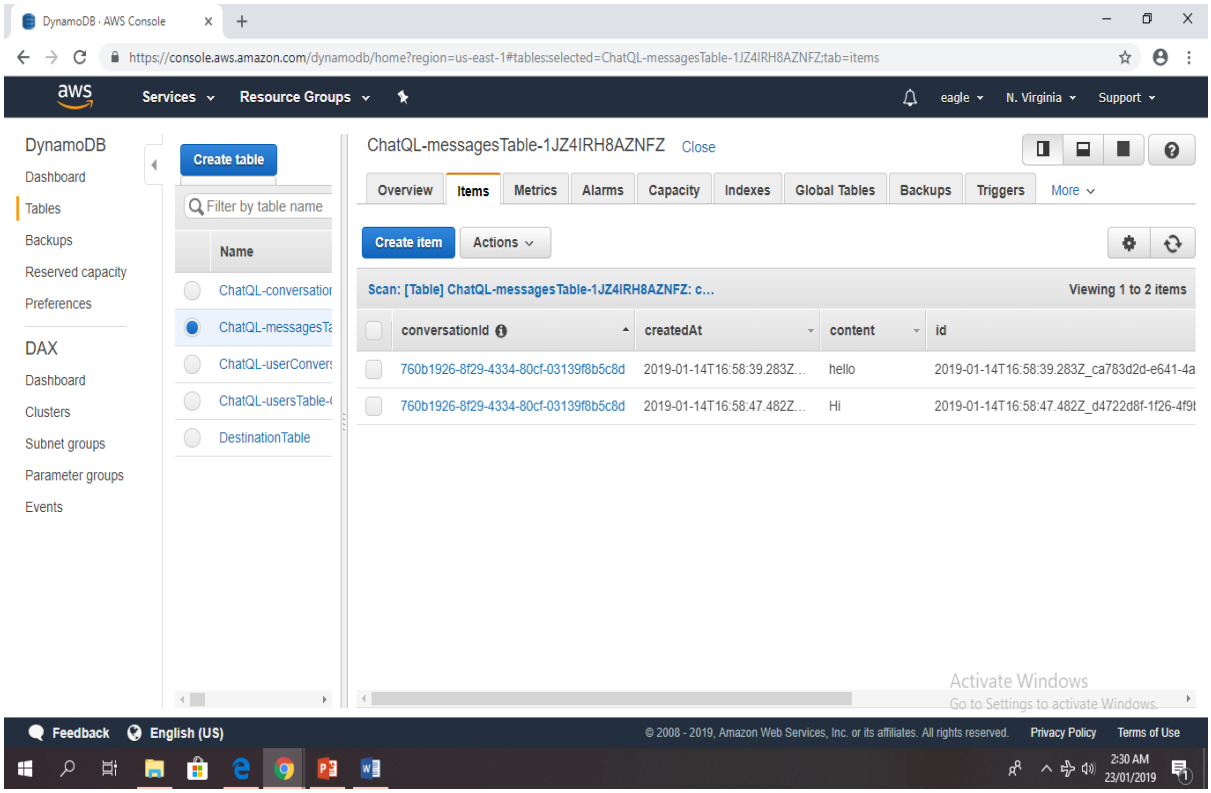


Figure 30: Messages Table

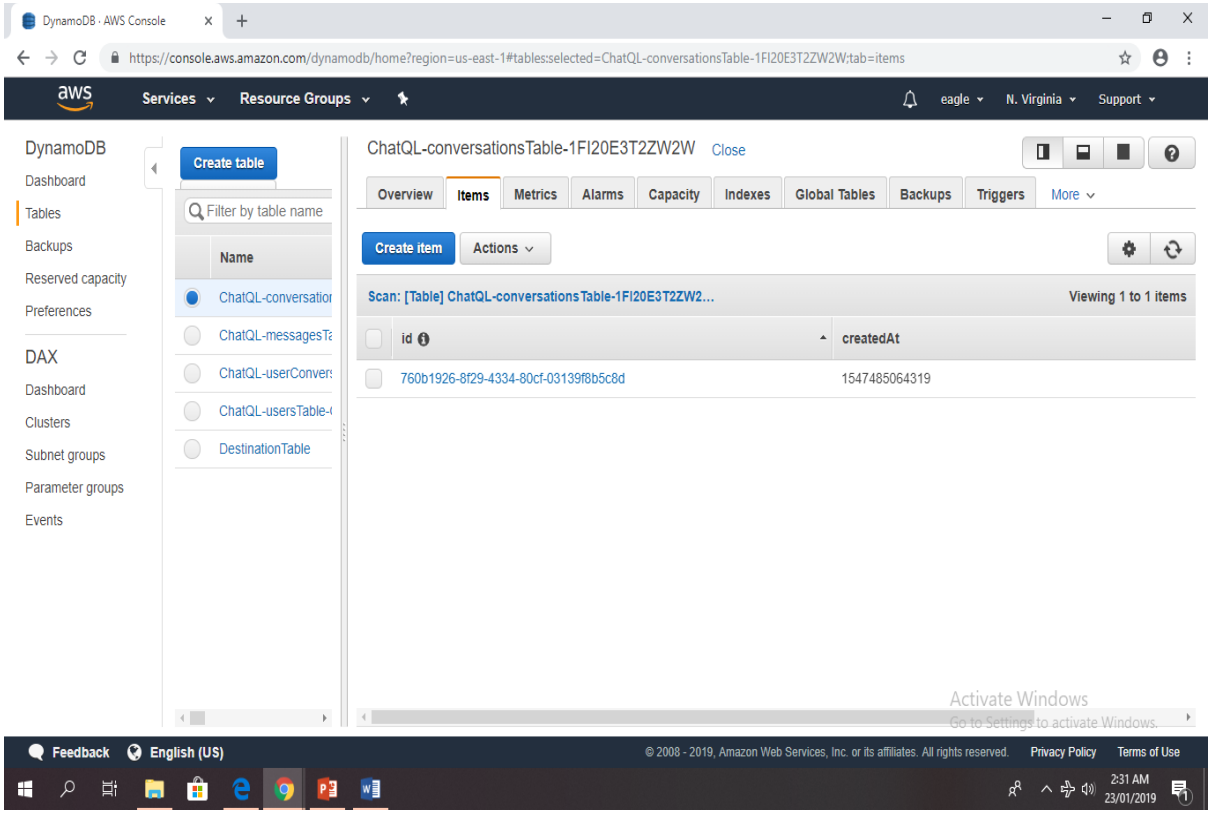


Figure 31: Conversations Table

5.2 Weather Forecast Application

This case study is divided into four sections. Firstly, the requirements of the weather application are discussed in **Section 5.2.1**. Secondly, **Section 5.2.2** contains the UML class diagram and state machine diagram with applied profile to present the system architecture of the required system. Furthermore, **Section 5.2.3** shows the transformation results in the form of generated code. And finally, **Section 5.2.4** contains validation of the system.

5.2.1 Requirements

For weather data application, user can search for popular tourist destinations. The application will provide real time weather analysis of the indexed destination. The data requirements include:

- **Authentication**
  User can authenticate with API key.
- **Client**
  The client fetches data from the cloud and rapidly builds UI. It is used with one of the view layer integrations.
- **Operations**
  - Read Requests consists of fetching list of travel destinations
  - Write Requests create destinations
  - Subscribe Requests automatically retrieve real time weather information as soon as it is received in the backend data sources
- **Schema**
  The schema comprises of object types and root types which defines the data shape that flows. It also defines the operations that can be performed.
- **Server Synchronization**
  The server processes the received requests and mapped them to logical functions for data operations or triggers. Then it passes this request to the Storage system protocol converter.
- **Storage System Protocol Converter**
  The storage system converter maps and executes the request payload against data source.

- **Data Source**

  A persistent storage system or a trigger, along with credentials for accessing that system or trigger.

The frontend requirements consist of:

- **WeatherApplication Module**

  Weather application module is composed of several components such as Destination Input Component and Destination View component. Container and Grid element is used for layout.

- **Weather Application Components**

  The chat application components include:

  - Destination Input Component

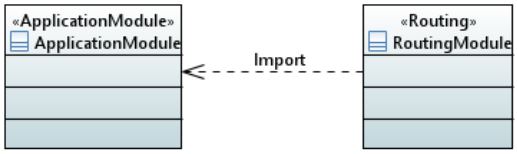
    Input component encompasses input group. Input group contains input and data binding. Button element is attached to the input element.

  - DestinationView Component

    Destination View component contains card element, data binding and DOM manipulation

5.2.2 Modeling

A complete model of data synchronization including frontend and backend for Weather Data Application has been provided. The model of Application User interface template is shown in Figure 32.



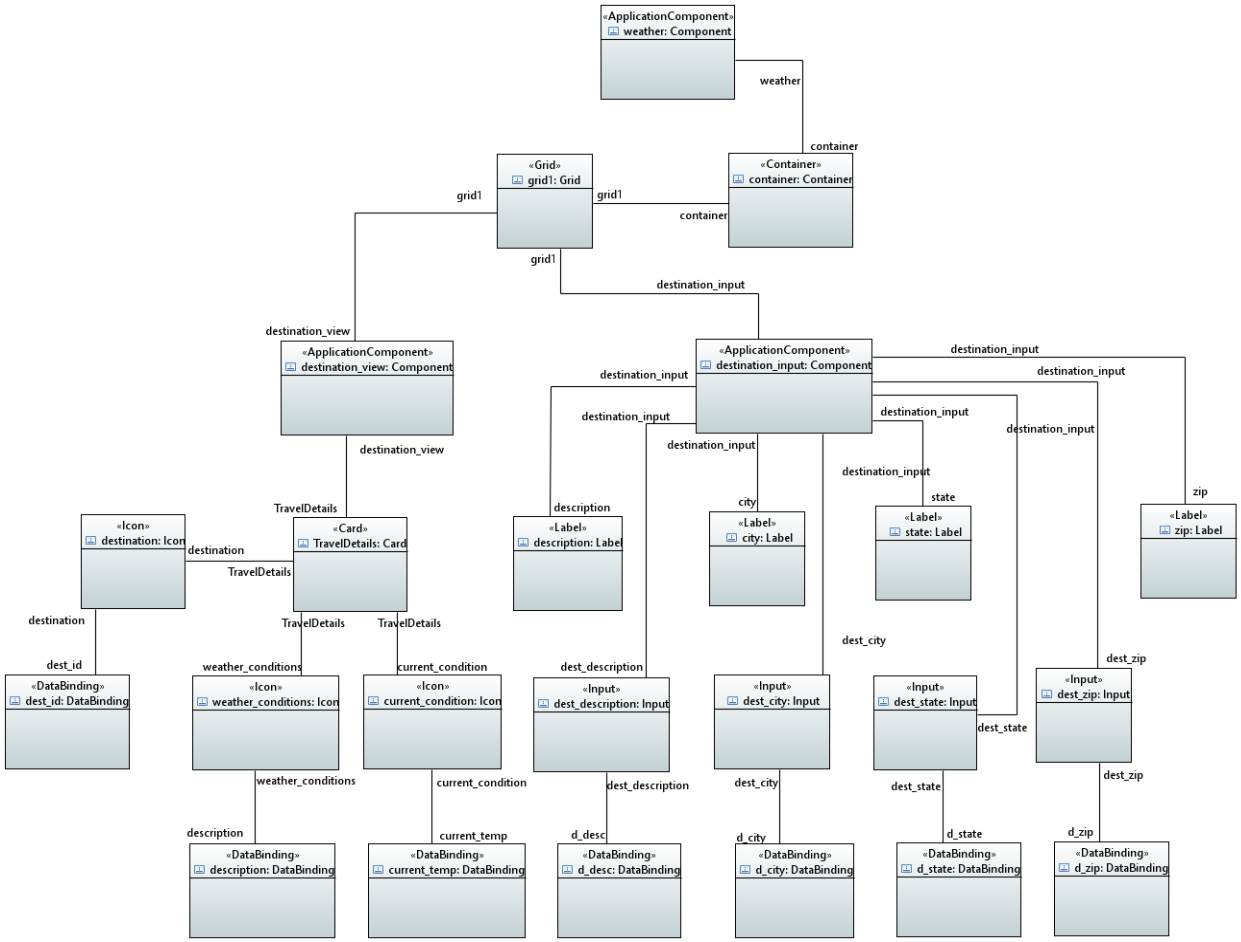


Figure 32: Model of Weather Forecast Application Frontend

In this model, <<ApplicationModule>> stereotype is applied to WeatherAppModule which loads other parts of our application. Also, this module imports components and modules within our application such as WeatherAppRoutingModule, WeatherAppComponent, DestinationViewComponent and DestinationInputComponent. <<Routing>> stereotype is mapped to WeatherAppRoutingModule to tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar. <<ApplicationComponent>> stereotype is mapped to different components of our weather application which contains template, business logic and metadata. The WeatherAppComponent contains container so <<Container>> stereotype is mapped to container which provide a basic layout element. The application uses grid so <<Grid>> is mapped to grid to layout and align content using containers, rows, and columns. <<Icon>> stereotype is mapped to the icon element which displays icons with the weather information. <<Label>> stereotype is mapped to label which provides additional information to the input. <<Input>> is applied to Input which creates a

destination by adding required values. <<Card>> is mapped to card element which delivers a flexible content container with numerous variations. <<Button>> stereotype is mapped to button which extend form control by adding button. <<DataBinding>> stereotype is mapped to the data binding instances which binds data to that instance.

Figure 33 shows a model of an instance specification for server type system which is the entry point of an application on server.

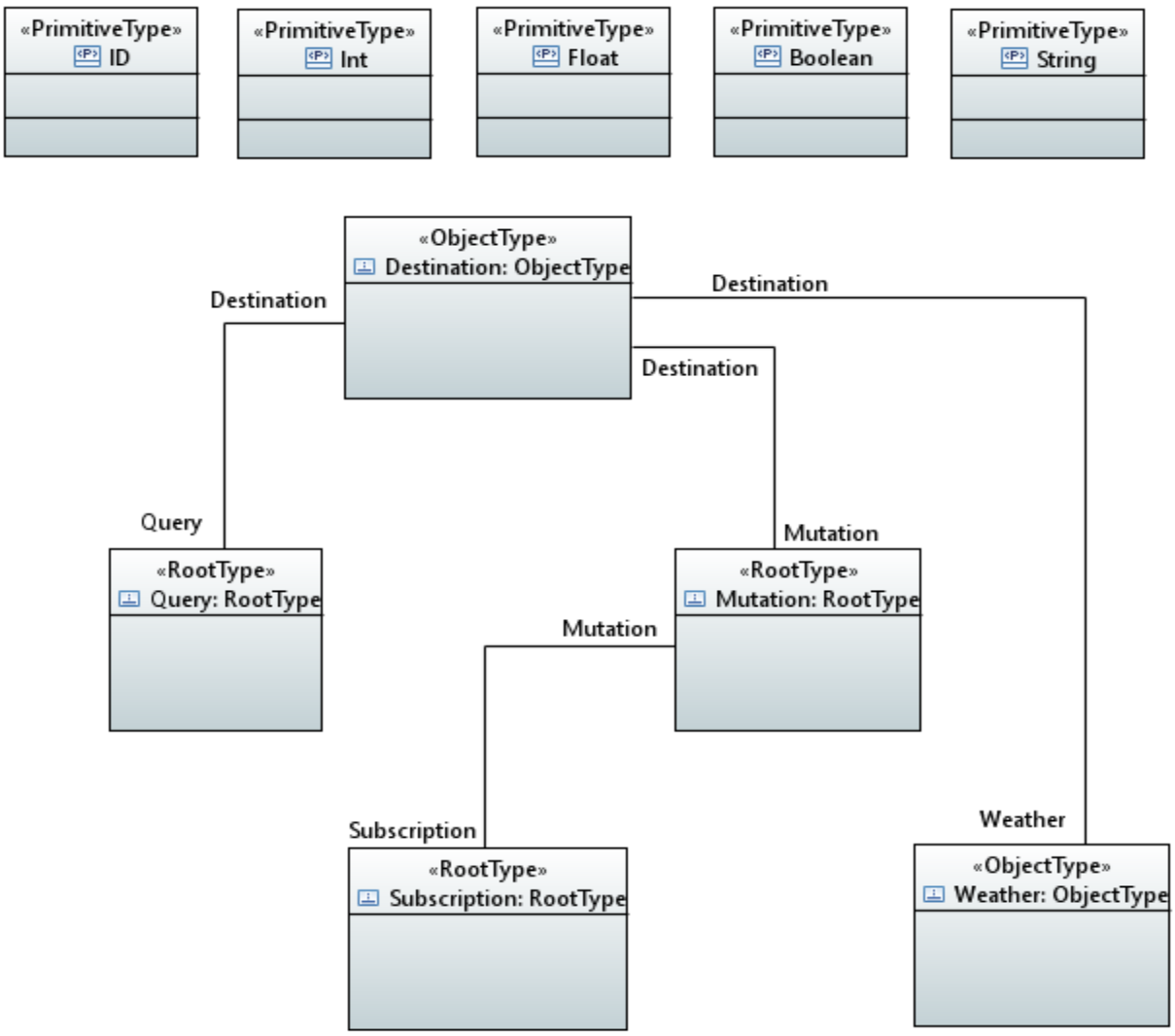


Figure 33: Model of Serverless Schema for Weather Forecast Application

In this model, <<RootType>> stereotype is applied to Root Types which defines the Root Operations (Read, Write and Subscribe). <<ObjectType>> is applied to Object Type which define fields of the object type. The type modifiers on property and parameters are also

applied through <<TypeModifier>> stereotype. <<Request_Data>> stereotype is applied to Root Operations that define the shape of the data that flows.

Figure 34 shows a model of state machine diagram for data sync client in data-driven application.

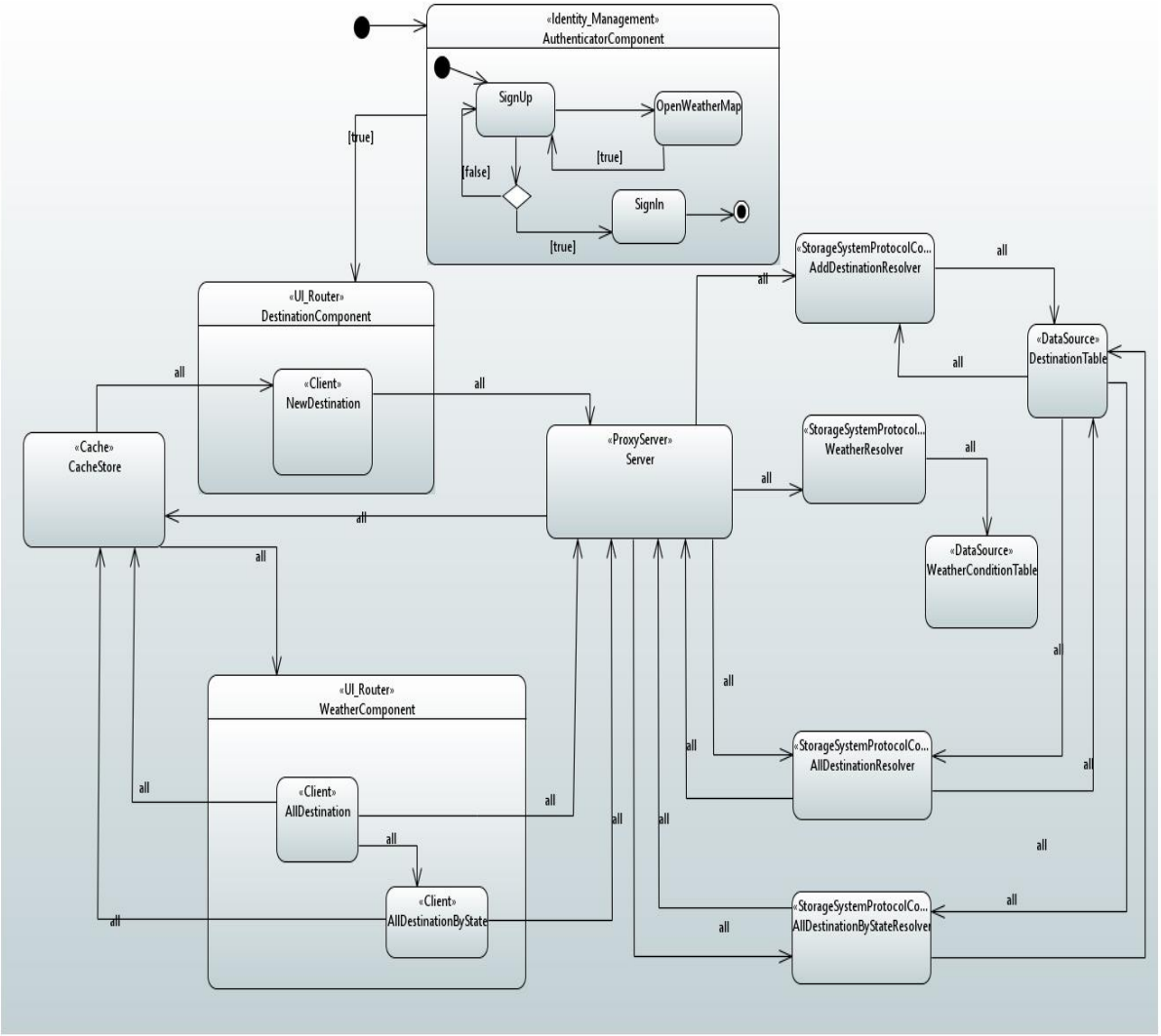


Figure 34: Model of Application Behavior for Weather Forecast Application

The model shows the state machine behavior for the whole application. For authentication user sign up for the OpenWeatherApp account to get the API key. The API key is used for authentication. After getting API key, user routes through the browser to the main application page. <<UI_Router>> is mapped to the DestinationComponent and WeatherComponent that manages the transition between application states. <<Client>> stereotype performs appropriate authorization wrapping of request statements before submitting to the server for

synchronizing. Responses are persisted in an offline store and write requests are made in a write-through pattern. To get the weather conditions, the server calls WeatherResolver that invokes the WeatherCondition data source to fetch the weather condition of the selected city. In the WeatherComponent, AllDestination state is triggered on a get request to fetch all the destinations from the cache through the <<Client>> stereotype. Therefore, cache-only condition is set to true. The cache checks for the fetched destinations. If data is found, it subscribes to that data and updates the User interface. The request is passed to server when a transition is triggered. The server processes the received requests and mapped them to logical functions for data operations. It calls the AllDestination Resolver function that adds Destination to the data source by using <<StorageTransitionProtocolConverter>> stereotype. The response is returned to the client. The cache is updated accordingly. Similarly, AllDestinationByState fetches the list of destinations by city State name. It calls the AllDestinationByStateResolver function that gets the updated destination through << StorageTransitionProtocolConverter >> stereotype and updates the cache store.

5.2.3 Code Generation

This section highlights the code generation process from our proposed transformation engine. The model of data synchronization is given as input to our proposed transformation engine whose interface is shown in **Figure 35**. The transformation engine uses transformation rules for mapping and transforms the model into code. UML model with .uml extension is selected as input model and target folder on desktop is provided as output folder for generated code files. On clicking the generate button following outputs are generated in the target folder.

1. The code for frontend Application in Angular 2 as shown in **Figure 36, 37**.
2. The backend code in GraphQL (Graph Query Language) as shown in **Figure 38, 39**.

Console shows the progress of transformation process.

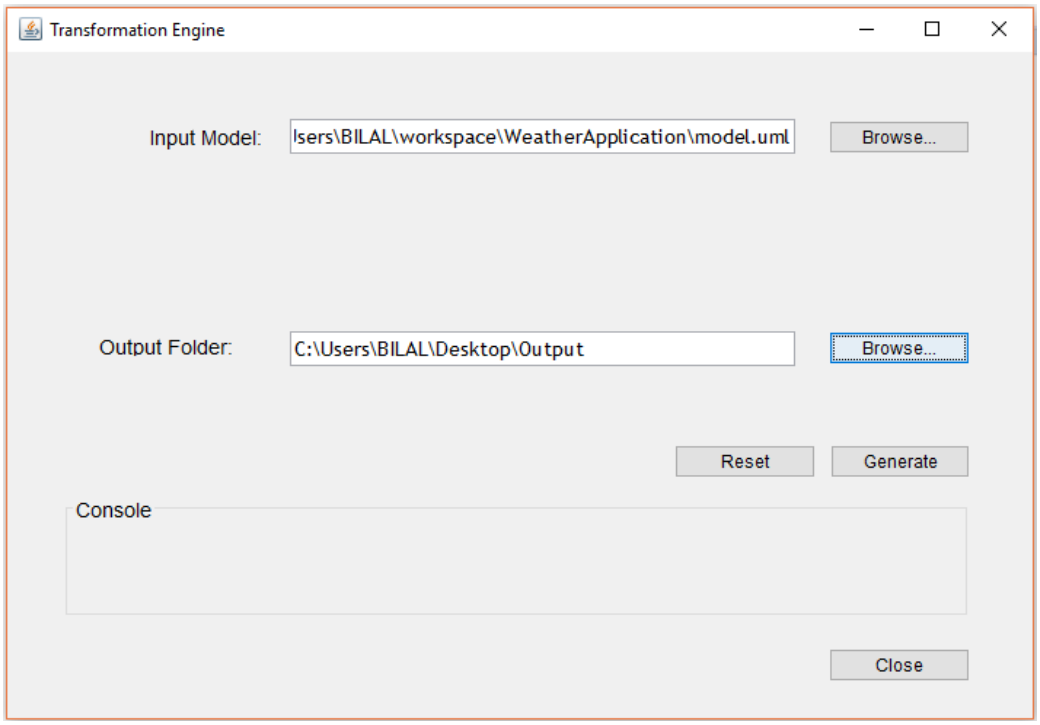


Figure 35: Transformation for Weather Forecast Application

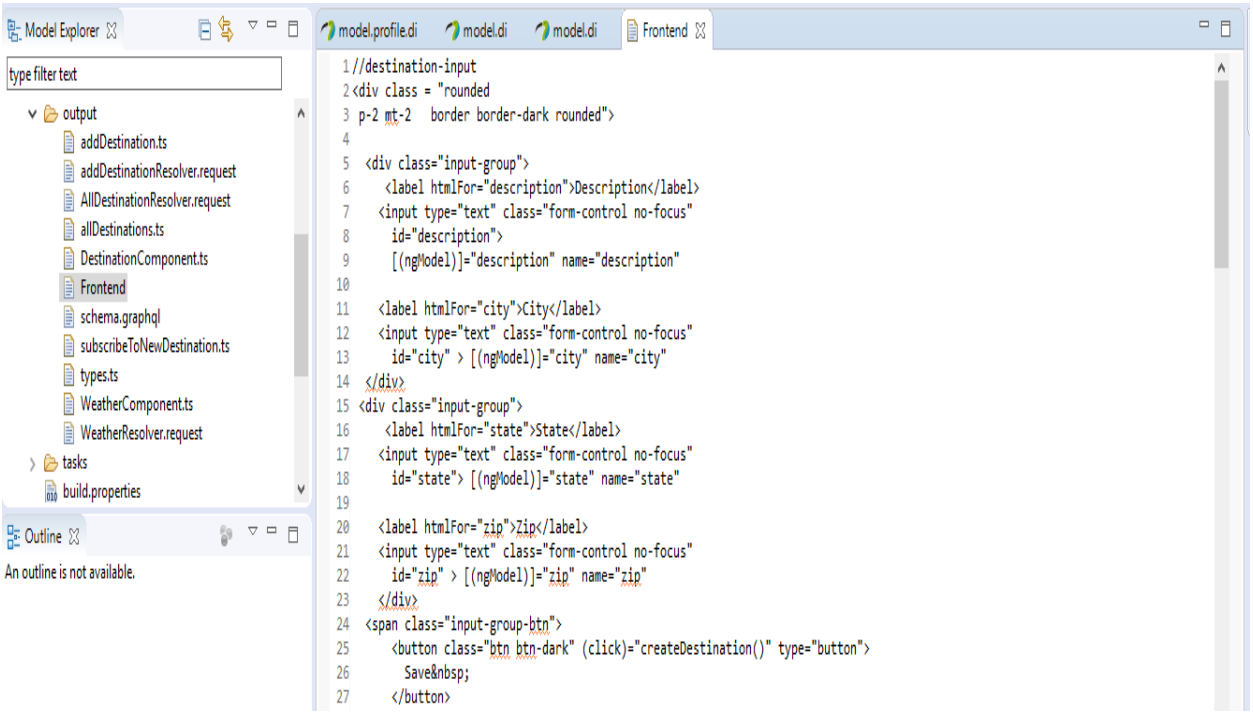


Figure 36: Generated Code of User Interfaces for Weather Forecast Application

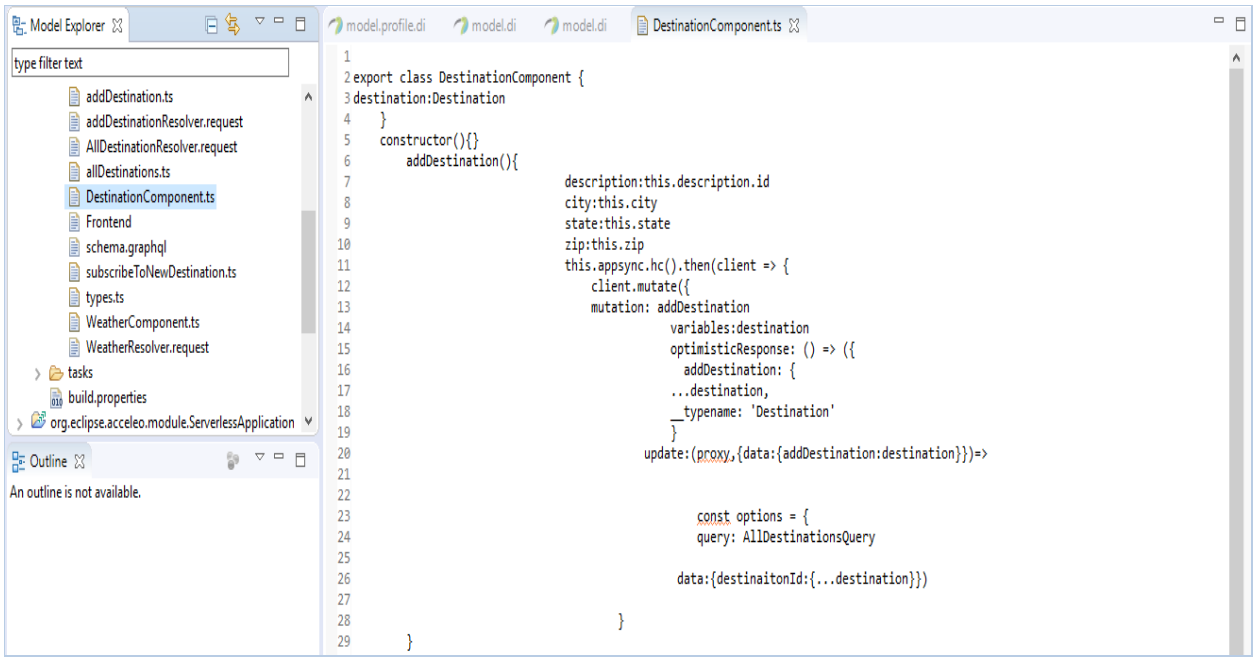


Figure 37: Generated Code of Data Synchronization Client for Weather Forecast Application

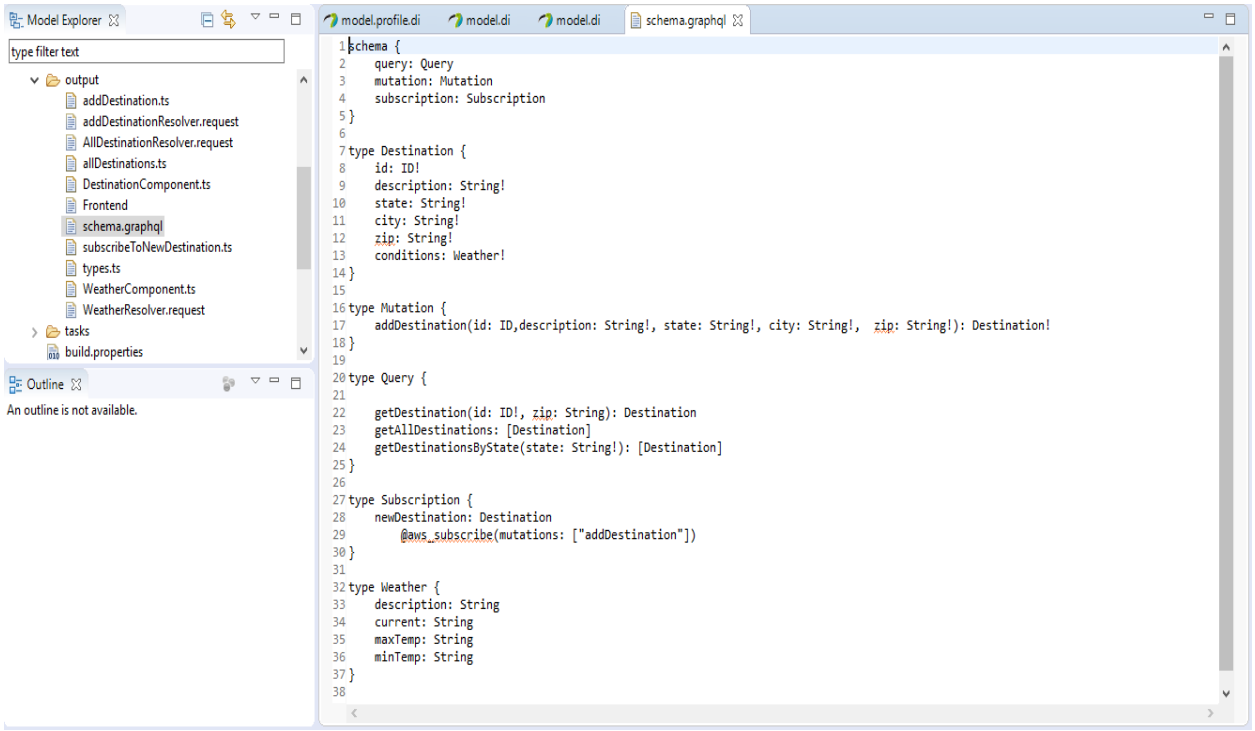


Figure 38: Generated Code of Serverless Schema for Weather Forecast Application

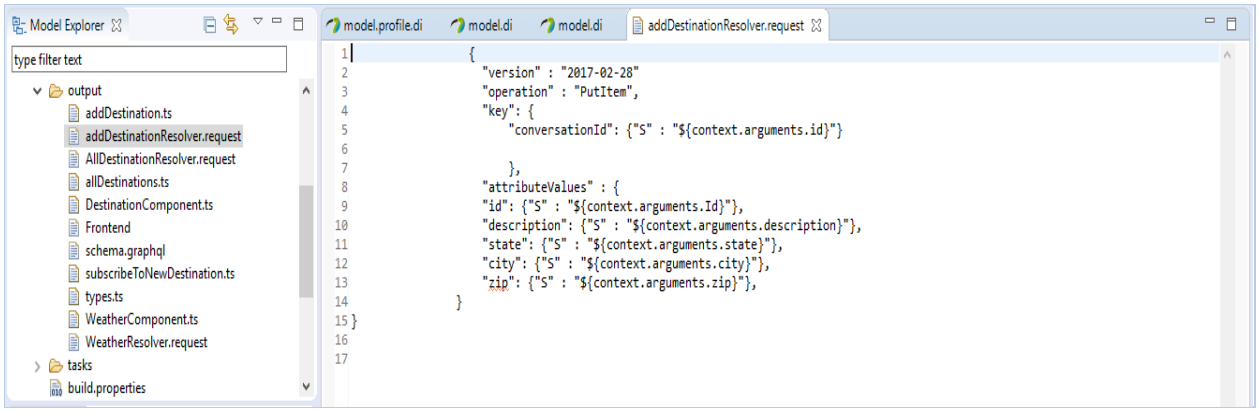


Figure 39: Generated Code of Data Resolvers for Weather Forecast Application

5.2.4 Verification

For verification of generated code, its compilation and execution are necessary. Therefore, we deploy the code in Angular Application. We created an angular application using angular cli and pasted our generated code accordingly as shown in **Figure 40, 41, 42 and 43.**

```
app.component.html
<div class="rounded p-2 mt-2">

  <div class="input-group">
     <label htmlFor="description">Description</label>
    <input type="text" class="form-control no-focus"
      id="description">
      [(ngModel)]="description" name="description"

    <label htmlFor="city">City</label>
    <input type="text" class="form-control no-focus"
      id="city" > [(ngModel)]="city" name="city"
</div>
 <div class="input-group">
     <label htmlFor="state">State</label>
    <input type="text" class="form-control no-focus"
      id="state"> [(ngModel)]="state" name="state"

    <label htmlFor="zip">Zip</label>
    <input type="text" class="form-control no-focus"
      id="zip" > [(ngModel)]="zip" name="zip"
    </div>
<span class="input-group-btn">
      <button class="btn btn-dark" (click)="createDestination()" type="button">
        Save 
      </button>
    </span>
    </form>
</div>
```

Figure 40: Code Deployed for Weather Data Application (User Interface)

DestinationComponent.ts

```
export class DestinationComponent {
destination:Destination
  }
  constructor(){}
    addDestination(){
                description:this.description.id
                city:this.city
                state:this.state
                zip:this.zip
                  this.appsync.hc().then(client => {
                  client.mutate({
                  mutation: addDestination
                        variables:destination
                        optimisticResponse: () => ({
                          addDestination: {
                        ...destination,
                        __typename: 'Destination'
                        }
                      update:(proxy,{data:{addDestination:destination}})=>


                          const options = {
                          query: AllDestinationsQuery

                         data:{destinaitonId:{...destination}})

```

Figure 41: Code Deployed for Weather Data Application (Data Sync Client)

schema .graphql

```
schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}

type Destination {
    id: ID!
    description: String!
    state: String!
    city: String!
    zip: String!
    conditions: Weather!
}

type Mutation {
    addDestination(id: ID,description: String!, state: String!, city: String!,  zip: String!):
    Destination!
}

type Query {

    getDestination(id: ID!, zip: String): Destination
    getAllDestinations: [Destination]
    getDestinationsByState(state: String!): [Destination]
}
```

Figure 42: Code Deployed for Weather Data Application (Server Type System)

mutation.addDestination.request

```
{
        "version" : "2017-02-28"
        "operation" : "PutItem",
        "key": {
            "conversationId": {"S" : "${context.arguments.id}"}

            },
        "attributeValues" : {
        "id": {"S" : "${context.arguments.Id}"},
        "description": {"S" : "${context.arguments.description}"},
        "state": {"S" : "${context.arguments.state}"},
        "city": {"S" : "${context.arguments.city}"},
        "zip": {"S" : "${context.arguments.zip}"},
    }
}

```

Figure 43: Code Deployed for Weather Data Application (Data Resolvers)

Validation of the application is done after deployment of code on AWS Serverless platform using local AWS command line interface as shown in Figure 44.

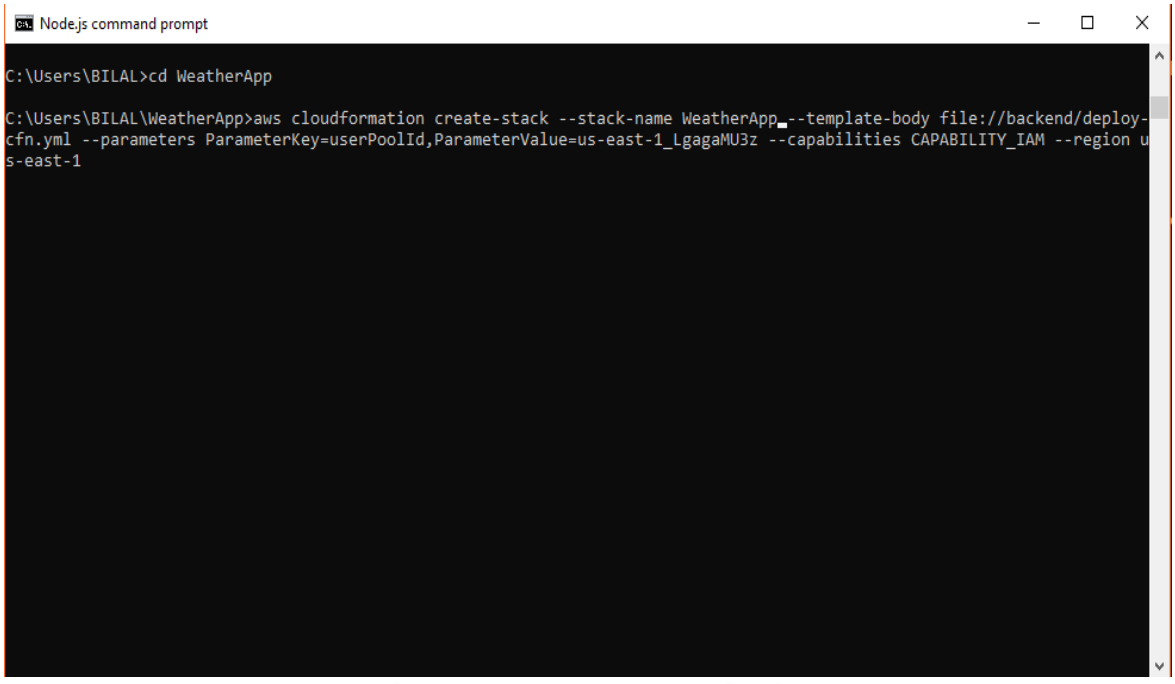


Figure 44: Deployment on AWS using Cloud Formation

Figure 45 shows the transformation result of the deployed application.

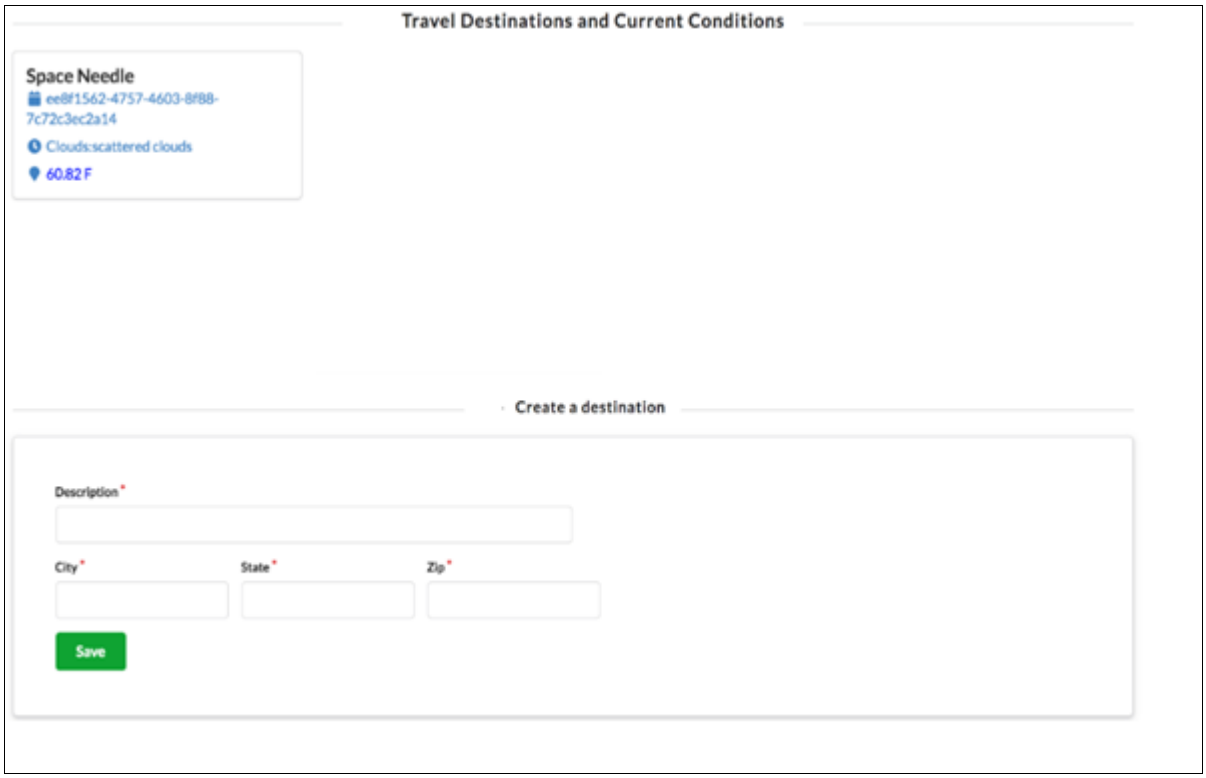


Figure 45: Transformation Result of Weather Forecast Application

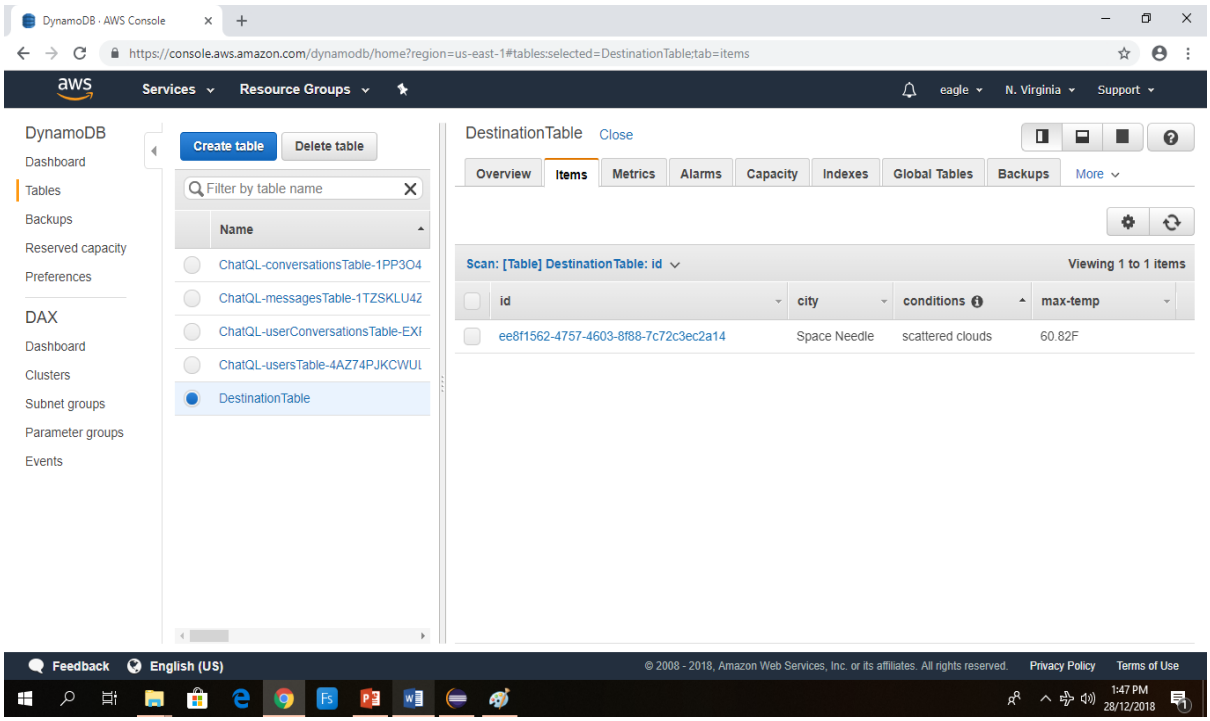


Figure 46: Destination Table

Chapter 6

Comparative Analysis

CHAPTER 6: COMPARATIVE ANALYSIS

The previous chapter deals with the implementation and validation aspects of the proposed work. The proposed framework presents the modeling of the concepts involved in implementation of a data synchronization for data driven serverless applications at higher level of abstraction. To model the concepts of system Unified Modeling Language (UML) profile is proposed and transformation of model into code has been carried out using transformation engine for implementation purpose. Our proposed approach provided a major contribution in the field of model driven engineering. This section discusses the comparison with our proposed solution.

6.1 Comparison

For comparison purpose, we have selected some researches from the previous studies and perform comparison as shown in **Table 5**.

Table 5: Comparison

Sr.No	Serverless Computing	Data-driven Applications	Frontend Prototype	Backend	Validation on Cloud	Modeling Approach	Tool Support
[26]	No	Yes	No	Yes	No	No	No
[28]	No	Yes	Yes	No	No	No	No
[29]	No	Yes	Yes	Yes	No	No	No
[33]	Yes	No	No	Yes	IBM Watson	No	No
[34]	Yes	Yes	No	Yes	OpenStack Swift	No	No
Our proposed Work	Yes	Yes	Yes	Yes	AWS	Yes	Yes

The above table provides a comparison between the previous literature studies and our proposed solution. The comparison is performed among the following parameters: Serverless computing, Data-driven Applications, Frontend Prototype, Backend, Modeling Approach and Tool support provided. The previous researches in literature shows that tool support is missing with data-driven serverless applications. A few solutions provided with data driven

applications target on the specific application domain. For instance, [23] provides a solution for data driven applications for vehicle tracking. A few uses simulation [34] to validate their approach. Some of the researches related to data driven applications has not been validated on cloud. Similarly, researches related to serverless does not include frontend of an application. Our proposed solution covers serverless data driven applications with tool support that generate both frontend and backend low-level implementation code for data store and synchronization. Moreover, we have validated our proposed approach by deploying it to AWS (Amazon Web Services). Furthermore, it contains a tool support for model to text transformation.

Chapter 7

Discussion and Limitation

CHAPTER 7: DISCUSSION AND LIMITATION

This section deals with a detailed discussion on the proposed research work as described in **Section 7.1.**

7.1 Discussion

Data Synchronization is a complex process in cloud-based application development especially, if data is shared among multiple users with varying devices. This research introduces a UML Profile to model the low-level implementation with regards to data synchronization concepts for serverless data-driven applications on cloud. Particularly, the behavior of the application is very complex, so we use model driven approach to simplify the design and implementation of an application. The proposed UMSDA provides backend concepts for data store and sync as well as frontend user interface and data binding concepts by adapting UML diagrams such as class diagram and state machine diagram. The application structural concepts are modelled in UML class diagram while state machine diagram is used to model the application behavior concepts. Consequently, our proposed tool provides numerous benefits in comparison with other state-of-the-art approaches (e.g. [17], [19] etc.).

The significant advantages of our proposed UMSDA are:

1) It simplifies the requirements for data synchronization for data-driven serverless applications.
2) It also simplifies the frontend integration with backend. The frontend requirements are also modelled for automatic code generation.
3) It supports the transformation engine that generates both frontend and backend code from high level source models.
4) It provides foundation to develop any serverless data driven application that requires real time data synchronization.

The models proposed in our solution are at high abstraction level and are easily understandable by various stakeholders. These models neglect the implementation details, thus simplifying the low-level implementation complexity. Furthermore, it reduces time and cost of development. Low level implementation code can be automatically generated from the source models. The generated code is deployed into its respective environment for verification. These models can be easily extendible for adding more features. Also, the

transformation engine is extendible. We can provide transformation rules for other user interface elements or new types in the schema.

The proposed framework has been validated on two case studies: Real time Chat Application and Weather Forecast Application. Both case studies require real time data synchronization and present the updated information to the users. It also handles the data in low network connectivity. The first case study uses single data source while the second one targets multiple data sources. The proposed solution provides automated code generation for data synchronization in data driven serverless applications, however, there exist some limitations. The frontend of an application does not cover all the user interface elements. Moreover, it does not completely provide full styling of UI elements. Furthermore, transformation rules for only Root types and object types are provided as they are used across the most applications.

Chapter 8

Conclusion and Future Work

CHAPTER 8: CONCLUSION AND FUTURE WORK

In this research, we propose UMSDA (Unified Modeling Language Profile for Serverless Data-driven Applications) to model the frontend and backend design and implementation requirements for data-driven serverless applications at high abstraction level. Particularly, due to increasing recognition of UML in industry, this research proposed the design and implementation of data driven serverless applications through UMSDA. The proposed UMSDA comprises several stereotypes related to frontend and backend concepts to model the data synchronization in serverless applications. These models generate code through the transformation engine. Verification of the generated code is done by deploying it to their respective implementation environments.

The research work is the first step to make the cloud application development process simpler. Particularly, UMSDA has been proposed which adapts the concept of UML Class Diagram and State Machine Diagram to model the design and implementation for both frontend as well as backend for data-driven serverless applications. Backend presents data store and synchronization requirements while frontend represents user interface requirements for integrating data with the view layer of serverless applications. A complete transformation engine is developed to transform the UMSDA source models into target low-level implementation of Angular 2 and GraphQL code respectively. The transformation engine implementation is carried out in Acceleo through Model to Text approach. We demonstrate the applicability of our proposed tool through two case studies deployed on AWS Serverless platform.

The results prove that the proposed framework allows the modeling of both frontend as well as backend requirements of data-driven serverless applications with simplicity. The transformation engine automatically generate code with high accuracy. The proposed framework greatly simplifies the design and implementation complexity of data-driven serverless applications to achieve certain business objectives like productivity and time to market.

In future, we may plan to further enhance the frontend to fully provide all the features by extending it with more elements. Furthermore, we also plan to enrich our application behavior with other javascript frontend frameworks such as React, React-Native etc. which are the most renowned framework used for frontend development these days.

REFERENCES

[1] Gojko Adzi, Robert Chatley 2017, “Serverless Computing: Economic and Architectural Impact”; ESEC/FSE’17 Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering Pages 884-889

[2] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, Olivier Tardieu 2017 " The Serverless Trilemma, Function Composition for Serverless Computing”, Onward!’17, Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software Pages 89-103

[3] Erwin Van Eyk, Simon Seif, Markus Thommes “The SPEC Cloud Group’s Research vision on FaaS and Serverless Architectures”, WoSC’17 Proceedings of the 2nd International Workshop on Serverless Computing Pages 1-4

[4] Amazon Web Services, Inc., “AWS Lambda: Developer Guide,” 2016.

[5] IBM Bluemix, “Welcome to Bluemix OpenWhisk.”: https://new- console.ng.bluemix.net/openwhisk/.

[6] Microsoft Azure, “Functions,” https://azure.microsoft.com/en-us/services/functions/.

[7] Google Cloud Platform, “Cloud Functions,” https://cloud.google.com/functions/.

[8] Mengting Yan, Paul Castro, Perry Cheng, Vatche Ishakian: Building a ChatBot with Serverless Computing: Proceedings of the 1st International Workshop on Mashup of Things and APIs (2016).

[9] Fatima Samea, Muhammad Waseem Anwar, Farooque Azam, Mehreen Khan, Muhammad Fahad Shinwari: An Introduction to UMLPDSV for Real-Time Dynamic Signature Verification: 24th International Conference on Information and Software Technologies (ICIST 2018), pages 388-398. doi: https://link.springer.com/chapter/10.1007%2F978-3-31999972-2_32.

[10] Muhammad Waseem Anwar, Muhammad Rashid, Farooque Azam and Muhammad Kashif Model-based design verification for embedded systems through SVOCL: an OCL extension for SystemVerilog Journal of Design Automation for Embedded Systems 2017, Vol 21, Issue 1, Pages 1-36

[11] Mehreen Khan, Muhammad Waseem Anwar, Farooque Azam, Fatima Samea and Muhammad Fahad Shinwari A Model-driven Approach for Access Control in Internet of Things (IoT) Applications – An Introduction to UMLOA The 24th

International Conference on Information and Software Technologies (ICIST 2018), *pages 198-209. DOI:* https://link.springer.com/chapter/10.1007%2F978-3-319-99972-2_16

[12] Model Driven Architecture [Online] Available: https://www.omg.org/mda/

[13] UML Profile [Online] Available: https://www.uml-diagrams.org/profile.html

[14] Xiuwei Zhang, Keqing He, Jian Wang, Jianxiao Liu, Chong Wang, Heng Lu: On-Demand Service-Oriented MDA Approach for SaaS and Enterprise Mashup Application Development. 2012 International Conference on Cloud Computing and Service Computing

[15] Wei-Tek Tsai, Wu Li, Babak Esmaeili, Wenjun Wu: Model-Driven Tenant Development for PaaS-Based SaaS. 2012 IEEE 4th International Conference on Cloud Computing Technology and Science

[16] Wei-Tek Tsai, Yu Huang, QiHong Shao, EasySaaS: A SaaS Development Framework. 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)

[17] Assylbek Jumagaliyev, Jon Whittle: Model-Driven Engineering for Multi-Tenant SaaS application development. CrossCloud '16 Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms

[18] Xiaoyan Jiang, Yong Zhang, Shijun Liu: A Well-designed SaaS Application Platform Based on Model-driven Approach. 2010 Ninth International Conference on Grid and Cloud Computing

[19] Xiyong Zhu, Shixiong Wang: Software Customization Based on Model-Driven Architecture Over SaaS Platforms. 2009 International Conference on Management and Service Science

[20] Dapeng Chen, Qingzhong Li, Lanju Kong: Process Customization Framework in SaaS Applications. 2013 10th Web Information System and Application Conference.

[21] G. Fylaktopoulos, M. Skolarikis, I. Papadopoulos, G. Goumas, A. Sotiropoulos, I. Maglogiannis: A distributed modular platform for the development of cloud-based applications. Future Generation Computer Systems

[22] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter: Cloud-Native, Event-Based Programming for Mobile Applications. 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems

[23] Moez Essaidi, ODBIS: Towards a Platform for On-Demand Business Intelligence Services. EDBT '10 Proceedings of the 2010 EDBT/ICDT Workshops

[24] Richard Fujimoto, Angshuman Guin, Michael Hunter, Haesun Park, Gaurav Kanitkar, Ramakrishnan Kannan, Michael Milholen, SaBra Neal, and Philip Pecher: A Dynamic Data Driven Application System for Vehicle Tracking, ICCS 2014. 14th International Conference on Computational Science

[25] Biplab Deka, Data-Driven Mobile App Design: UIST '16 Adjunct Proceedings of the 29th Annual Symposium on User Interface Software and Technology, Pages 21-24

[26] Jiaju Wu, Bin Ji, Xinglin Zhu, Zheng Cheng, Lirong Meng: Design and Implementation of Data-Driven based Universal Data Editing Framework, 2017 Chinese Automation Congress (CAC)

[27] Joao Seixas, A Model-Driven Development Approach for Responsive Web Applications: The XIS-Web Technology: https://pdfs.semanticscholar.org/f6f3/93dfafda18b6f398a601297546a5167b456a.pdf

[28] Kapil Kumar, Joy Bose, Sandeep Kumar Soni: A Generic Visualization Framework based on a Data Driven Approach for the Analytics data, 2017 14th IEEE India Council International Conference (INDICON)

[29] Voon Yang Nen, Ong Chin Ann: Pigeon-Table: A Quick Prototyping Tool using Twitter Bootstraps and AngularJS for Data-Driven Web Application Development, 2017 International Conference on Computer and Drone Applications (IConDA)

[30] Steffen Vaupel, Damian Wlochowitz, Gabriele Taentzer: A Generic Architecture Supporting Context-Aware Data and Transaction Management for Mobile Applications, 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems

[31] Janis Kampars, Janis Grabis: Near real-time big-data processing for data driven applications, 2017 International Conference on Big Data Innovations and Applications

[32] Mengting Yan, Paul Castro, Perry Cheng, Vatche Ishakian: Building a ChatBot with Serverless Computing: Proceedings of the 1st International Workshop on Mashup of Things and APIs (2016)

[33] Nirmal K Mukhi, Srijith Prabhu, Bruce Slawson: Using a serverless framework for implementing a cognitive tutor: experiences and issues. WoSC '17: Proceedings of the 2nd International Workshop on Serverless Computing (2017)

[34] Josep Sampe, Marc Sánchez-Artigas, Pedro García-Lopez, Gerard París: Data-Driven Serverless Functions for Object Storage, Middleware '17, Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Pages 121-133

[35] Li Weiping, An analysis of new features for workflow system in the SaaS software. ICIS '09 Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human

[36] Garrett McGrath, Brenden Judson, Paul Brenner, Jared Short, Stephen Ennis, Cloud Event Programming Paradigms Applications and Analysis: 2016 IEEE 9th International Conference on Cloud Computing

[37] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, Daeyoung Kim, GPU Enabled Serverless Computing Framework: 26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

[38] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, Vincent Emeakaroha, A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms: 2017 IEEE 9th International Conference on Cloud Computing Technology and Science

[39] Markus Ast, Martin Gaedke, Self-contained Web Components through Serverless Computing: WoSC '17 Proceedings of the 2nd International Workshop on Serverless Computing

[40] Guangyu Li, Qiang Shen, Yong Liu, Houwei Cao, Zifa Han, Feng Li, Jin Li: Data-driven Approaches to Edge Caching: Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies

[41] GraphQL [Online] Available: https://graphql.org/ [Accessed: 20-09-2018]

[42] Angular [Online] Available: https://angular.io/ [Accessed: 10-10-2018]

[43] Angular Guide [Online] Available: https://angular.io/guide [Accessed: 10-10-2018]

[44] Angular CLI [Online] Available: https://cli.angular.io/ [Accessed: 10-10-2018]

[45] Eclipse Acceleo [Online] Available: https://www.eclipse.org/acceleo/ [Accessed: 03-10-2018]

[46] Papyrus [Online] Available: https://wiki.eclipse.org/Papyrus_User_Guide [Accessed: 03-10-2018]

[47] Acceleo Documentation [Online] Available: https://www.eclipse.org/acceleo/documentation

[48] AWS Command Line Interface [Online] Available: https://aws.amazon.com/cli/ [Accessed: 17-09-2018]

[49] Apollo GraphQL [Online] Available: https://www.apollographql.com/ [Accessed: 22-09-2018]

[50] AWS Serverless Application Model [Online] Available: https://aws.amazon.com/serverless/sam/ [Accessed: 13-11-2018]

[51] Unified Modeling Language [Online] Available: https://sparxsystems.com.au/platforms/uml.html [Accessed: 19-08-2018]

[52] Papyrus UML [Online] Available: https://papyrusuml.wordpress.com/

[53] Serverless Web Application [Online] Available: https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/ [Accessed: 30-09-2018]

[54] Serverless AWS Documentation [Online] Available: https://serverless.com/framework/docs/providers/aws/ [Accessed: 30-09-2018]

[55] Serverless Architecture AWS [Online] Available: https://aws.amazon.com/lambda/serverless-architectures-learn-more/ [Accessed: 01-10-2018]

[56] AWS SAM Local [Online] Available: https://github.com/thoeni/aws-sam-local [Accessed: 14-10-2018]