Project Number 780251

D7.8 Integrated Platform - Final Version

Version 1.2
28 July 2020
Final

Public Distribution

CLMS

Project Partners: **Alpha Bank, ATB, Centrum Wiskunde & Informatica, CLMS, Edge Hill University, GMV, OTE, SWAT.Engineering, The Open Group, University of L'Aquila, University of Namur, University of York, Volkswagen**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the TYPHON Project Partners accept no liability for any error or omission in the same.

© 2020 Copyright in this document remains vested in the TYPHON Project Partners.

PROJECT PARTNER CONTACT INFORMATION

Alpha Bank Vasilis Kapordelis 40 Stadiou Street 102 52 Athens Greece Tel: +30 210 517 5974 E-mail: vasileios.kapordelis@alpha.gr	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de
Centrum Wiskunde & Informatica Tijs van der Storm Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 9333 E-mail: storm@cwi.nl	**CLMS** Antonis Mygiakis Mavrommataion 39 104 34 Athens Greece Tel: +30 210 619 9058 E-mail: a.mygiakis@clmsuk.com
Edge Hill University Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk	**GMV Aerospace and Defence** Almudena Sánchez González Calle Isaac Newton 11 28760 Tres Cantos Spain Tel: +34 91 807 2100 E-mail: asanchez@gmv.com
OTE Theodoros E. Mavroeidakos 99 Kifissias Avenue 151 24 Athens Greece Tel: +30 697 814 7618 E-mail: tmavroeid@ote.gr	**SWAT.Engineering** Davy Landman Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 633754110 E-mail: davy.landman@swat.engineering
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org	**University of L'Aquila** Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it
University of Namur Anthony Cleve Rue de Bruxelles 61 5000 Namur Belgium Tel: +32 8 172 4963 E-mail: anthony.cleve@unamur.be	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Volkswagen Behrang Monajemi Berliner Ring 2 38440 Wolfsburg Germany Tel: +49 5361 9-994313 E-mail: behrang.monajemi@volkswagen.de	

DOCUMENT CONTROL

Version	Status	Date
0.1	Updated Architecture	01 June 2020
0.2	API Services	05 June 2020
0.3	Initial User Guide	19 June 2020
0.4	Updated Workflows	21 June 2020
0.5	Content Corrections	24 June 2020
0.6	Updated Table of Figures, Table of Abbreviations	10 July 2020
0.7	Formatting Corrections	15 July 2020
0.8	Updated Continuous Integration & Deployment	20 July 2020
0.9	Added contributions from SWAT, University of L'Aquila, ATB, University of Namur for User Guide	24 July 2020
1.0	Finalize conclusions, add requirements	24 July 2020
1.1	Address internal review comments, update ML documentation, add Analytics documentation	27 July 2020
1.2	Address further review comments, finalisation for EC submission	28 July 2020

TABLE OF CONTENTS

1. Introduction **1**

2. Updated Platform Architecture **1**

2.1 Components Overview *1*
2.1.1 Typhon ML 1
2.1.2 Typhon DL 2
2.1.3 Polystore API 2
2.1.4 Polystore UI 2
2.1.5 Metadata Database 2
2.1.6 Evolution Toolset 2
2.1.7 Typhon QL Server 3
2.1.8 Analytics Service 3
2.1.9 Storage Databases 4

3. Workflows **4**

3.1 Design *5*

3.2 Deployment *5*

3.3 Runtime *6*
3.3.1 Query Execution 6
3.3.2 Evolution 7

4. Hybrid Polystore API & UI **8**

4.1 API Services *8*
4.1.1 User Services 8
4.1.2 Status Services 8
4.1.3 Model Services 8
4.1.4 Query Services 9
4.1.5 Backup/Restore Services 9

4.2 Polystore UI *9*

5. Continuous Integration & Deployment Process **10**

5.1 Overview *10*

5.2 Typhon Eclipse plugins *10*

5.3 Polystore API *11*

5.4 Polystore UI *11*

5.5 Evolution *12*

5.6 Typhon QL *12*

5.7 Archiva Repository *12*

5.8 Distribution Project *12*

5.9 GitHub Integration *13*

5.10 Slack Integration *13*

6. User Guide **14**

6.1 Installation *14*
6.1.1 Installing Eclipse 14
6.1.2 Installing Docker 16
6.1.3 Typhon Toolset 16
6.1.4 Optional Libraries 16

6.2 Usage *17*
6.2.1 Design Example 17
6.2.2 Runtime Example 26

6.3 Component Documentation *29*

6.3.1 Typhon Modelling Language (ML) ... 29
6.3.2 Typhon Deployment Language (DL) ... 39
6.3.3 Typhon Query Language (QL) Eclipse Plugin ... 54
6.3.4 Polystore API ... 61
6.3.5 Typhon Query Language (QL) ... 69
6.3.6 Evolution ... 76
6.3.7 Analytics ... 103

6.4 Known issues ... 115

7. Conclusions ... 116

TABLE OF FIGURES

Figure 1: Architecture of TYPHON Platform ... 5
Figure 2: Query Execution Workflow ... 6
Figure 3: Evolution Workflow ... 7
Figure 4: Typhon DL CI Workflow ... 11
Figure 5: Polystore API CI Workflow ... 11
Figure 6: Distribution CI Workflow ... 13
Figure 7: Downloading Eclipse Installer ... 14
Figure 8: Installing Eclipse for Java Developers ... 15
Figure 9: Eclipse Installation ... 15
Figure 10: Installing New Software to Eclipse IDE ... 16
Figure 11: Sirius Installation ... 17
Figure 12: Creating new Java Project ... 18
Figure 13: Creating Project ... 18
Figure 14: Creating TML file ... 19
Figure 15: Xtext project prompt ... 20
Figure 16: Simple ML example ... 21
Figure 17: Creating XMI file from TML ... 22
Figure 18: Creating Visual Model Representation ... 23
Figure 19: Visual Model Representation ... 23
Figure 20: Creating TDL file ... 24
Figure 21: Generating Deployment Scripts ... 25
Figure 22: Sample Project files after model XMI file generation ... 25
Figure 23: Polystore UI Login Page ... 26
Figure 24: Polystore UI Manage Users Page ... 27
Figure 25: Polystore UI Models Page ... 28
Figure 26: Query Menu ... 29
Figure 27: TyphonML overview ... 30
Figure 28: Custom data types ... 31
Figure 29: Conceptual entity ... 32
Figure 30: Relational DB ... 33
Figure 31: Document DB ... 33
Figure 32: Graph DB ... 34
Figure 33: Key-Value DB ... 34
Figure 34: Change operators ... 35
Figure 35: Graphical editor ... 37
Figure 36: OpenAPI generation ... 38
Figure 37: An instance of generated OpenAPI specification ... 39
Figure 38: TyphonDL DB Template preferences ... 42
Figure 39: TyphonDL Creation Wizard ... 44
Figure 40: TyphonDL Creation Wizard: Page one ... 45
Figure 41: TyphonDL Creation Wizard: Configuring the Analytics component Docker Compose vs. Kubernetes) ... 45
Figure 42: TyphonDL Creation Wizard: Choosing the DBMS for each database (Docker Compose vs. Kubernetes) ... 46
Figure 43: TyphonDL Creation Wizard: Further database configuration (MariaDB container vs. MongoDB container) 47

Figure 44: TyphonDL Creation Wizard: Further database configuration (MongoDB external database vs. MariaDB Galera Cluster) 48
Figure 45: TyphonDL textual editor with syntax highlighting and auto completion 52
Figure 46: Generate Deployment Scripts 53
Figure 47: Creating a new Project 55
Figure 48: New QL project 55
Figure 49: QL Setup 56
Figure 50: Typhon QL options 57
Figure 51: Dump Schema results 58
Figure 52: Reset Polystore prompt 59
Figure 53: Insert query example 60
Figure 54: Results of select query 61
Figure 55: Use Inject to model to generate XMI file 81
Figure 56: Architecture of the continuous evolution tool 86
Figure 57: Main page of the visual analytics tool 87
Figure 58: Overview of the current schema configuration of the Polystore 88
Figure 59: Overview of the current size of the Polystore entities 89
Figure 60: Evolution of the entity size over time 89
Figure 61: CRUD operation distribution 90
Figure 62: Proportion of queried entities 90
Figure 63: CRUD operation distribution over time 91
Figure 64: TyphonQL queries monitoring 91
Figure 65: Query execution monitoring 92
Figure 66: Schema evolution recommendations 93
Figure 67: TyphonDL Creation Wizard 94
Figure 68: Containers of the Polystore Continuous Evolution component 95
Figure 69: Data ingestion process 97
Figure 70: Example input relational schema with four tables 99
Figure 71: Input schema extracted as three entities 99

TABLE OF ABBREVIATIONS

Abbreviations	Detailed Factors
API	Application Programming Interface
CI	Continuous Integration
CI/CD	Continuous Integration/Continuous Deployment
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DL, Typhon DL	Definition Language, Typhon Definition Language
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
ML, Typhon ML	Modelling Language, Typhon Modelling Language
OCL	Object Constraint Language
QL, Typhon QL	Query Language, Typhon Query Language
SDK	Software Development Kit
UI	User Interface
URL	Uniform Resource Locator
WP	Work Package

EXECUTIVE SUMMARY

This deliverable provides the final working software prototype synthesizing relevant technical contributions and a technical report.

This report document describes the final implementation of the integrated platform and provides a detailed architecture and life-cycle management description. Furthermore, it discusses details on how users are allowed to design, deploy, query and evolve hybrid Polystores, via the platform's Application Programming Interface (API) and its Graphical User Interface (GUI).

Additionally, it provides a clear representation of the various workflows naturally occurring for the design, deployment and usage of the Polystore, along with the interactions between the various components responsible.

This document continues with an installation and usage guide which is accompanied by the appropriate screenshots and concludes with an outline of the work done and the requirements of the platform, as expressed by the project's Use Case partners.

1. INTRODUCTION

The goal of the Integrated Platform work package (WP7) is to deliver a working software prototype synthesizing relevant technical contributions from work packages 2 to 6, as well as a written report. The final implementation of the integrated platform is based on a modern and robust architecture. It fully supports the design, deployment, querying and evolution of hybrid Polystores, as described in the previous work packages.

The integrated platform enhances its interoperability characteristics by enabling other systems and applications to access its core functionality through an Application Programming Interface (API). Hybrid Polystore users alternatively may access the platform and perform available tasks via a Graphical User Interface (GUI), which is accessible by every device capable of running a web browser. Additionally, some of the functionalities offered from the Polystore are available for execution through relevant Eclipse plugins.

This report provides an architectural overview and outlines the implementation details of the platform's core components and modules (including the API and the GUI), their orchestration and deployment.

To support the integrated platform through its whole lifecycle, a Continuous Integration (CI) and Deployment environment infrastructure has been designed and set-up. This document offers detailed information on how each core component and the platform, as a whole, is handled by this CI infrastructure in order to deliver stable products to end users and development teams.

The remainder of this document contains a detailed User Guide that explains the user actions required to perform basic day-to-day tasks when using the Integrated Platform.

2. UPDATED PLATFORM ARCHITECTURE

2.1 COMPONENTS OVERVIEW

The Integrated Hybrid Polystore Platform consists of several independent components, which coexist and cooperate to serve the platform's purposes. These components are the following: the Polystore API, the Polystore UI, the Metadata Database, the Evolution components, the Query Engine contained in the QL Server, the Analytics service and the actual Storage Databases, as defined in the Typhon DL. Additionally, the Integrated Platform relies on the Modelling and Definition Language components, which are available through Eclipse.

2.1.1 Typhon ML

The starting point of each Polystore is Typhon ML. Through ML, a user can model their desired schemas and corresponding databases. This is done through an Eclipse installation with the corresponding plugins available. It supports all basic data types, and classical SQL-like relationships between entities. Additionally, ML offers the capabilities of producing visual schemas based on the entities and their relationships. Finally, ML can also produce basic CRUD service

implementations based on the entities modelled. Finally, ML supports a variety of database types, such as Relational, Document, Graph and Key-Value backends.

2.1.2 Typhon DL

Following up ML, the Typhon DL component is used to create deployment scripts for the entities and databases modelled. This component makes use of the ML file to generate additional configuration files for databases and components. Various options are available to the user through the DL wizard, such as making the databases public, changing the default credentials, an option to use analytics and evolution components etc. The final step for the user is to generate the deployment scripts based on the selections made previously, which will create deployment configurations for all selected databases and containers for the various Polystore components.

2.1.3 Polystore API

The Polystore API component is responsible both for providing to external systems a variety of endpoints serving the platform’s core functionalities and for orchestrating the other components within the Polystore accordingly. If the analytics component is used, it is responsible for populating appropriate queues with incoming queries. It is also responsible for orchestrating the initialization of the databases through the QL Server. It forwards authenticated Typhon QL queries to the QL Server and prepares the responses containing the unified query results. Simple CRUD operations for declared entities are also available through the API, using the QL Server. Moreover, it handles all Metadata Database operations required for user management, model versioning etc. It is implemented using the Java programming language and the Spring Enterprise Applications Framework.

2.1.4 Polystore UI

The most user-friendly way to access and use the Hybrid Polystore platform is through the Polystore User Interface. Technically, it is a web-based client for the Polystore API that allows users to interact with the platform in a familiar way, using any device capable of connecting to the Internet and running a web browser. The Polystore’s UI Backend part is coded in Java using both Spring Enterprise Applications Framework and the Polystore API component as well. For the Frontend part, well-known web technologies have been used, such as CSS 3, HTML 5, Typescript and Angular 8. The User Interface is used as an easy way to perform the various operations supported by the API.

2.1.5 Metadata Database

The Metadata Database is used to store data related to Model versions, Model update history and user management. For this purpose, the Hybrid Polystore Platform has adopted MongoDB, a lightweight, cross-platform, document-oriented database which uses JSON-like documents for data storage and offers functionality like indexing, multi-document ACID (Atomicity, Consistency, Isolation, Durability) transactions, replication and load balancing. The Polystore API component has access to the Metadata Database and handles all metadata related operations.

2.1.6 Evolution Toolset

Hybrid Polystore needs to evolve in response to changes to the business, technological or regulatory requirements. Users apply evolutionary changes to their models at design time via the Typhon ML and Typhon DL, which result in a new model version. The Evolution component is responsible for propagating these changes via produced change operators to all the affected components that

operate inside the Integrated Polystore Platform. A standalone Java Archive file (JAR) contains the Schema and data evolution/migration classes and resources, which can be used externally by a user by providing the relevant configuration information (change operators, API connection information etc.). When the tool receives new model changes via change operators, it uses the Query services of the API to perform the required schema and data migrations to the Polystore Databases.

An additional part of the Evolution component is the Query Evolution Tool. This tool is packaged as an Eclipse plugin, and similar to the schema/data evolution tool, it takes as input a model with change operators to be applied to the schema and a list of QL queries. The tool process then evolves these queries to be compatible with the new schema changes, while also classifying them as such. If this is not possible for any of the queries, they are also classified as broken with a relevant comment as to why they cannot be transformed.

Another tool offered by the Evolution toolset is the Evolution analytics tool, which makes use of the analytics component to find potential performance issues stemming from the schema and produces relevant recommendations (in the form of change operators) to the user. This analysis can be also viewed through a Graphical Interface, which is coupled with the Evolution analytics tool.

Finally, a data ingestion tool is also included in the toolset, allowing users to ingest data into the Polystore from relational databases. Details on installation, deployment and usage of this toolset can be found in Section 6.3.6 of this document.

2.1.7 Typhon QL Server

The Query Language Server is one of the most critical components of the Hybrid Polystore. It consists of a REST API service, which integrates the actual Query Engine, and is used by the Polystore API and the Evolution components. The decision to move the Query Engine to a dedicated REST API contained in a separate container was made in order to reduce interdependencies and complexity of development, as using the Query Engine library directly created a lot of overhead for the Polystore API. Additionally, by making the QL Server stateless, the Polystore user can scale up specifically the querying capabilities of the Hybrid Polystore, as it is a point that, under big data operations may come under heavy load.

The primary task of the QL Server (and the Query Engine) is to process query requests. Once a new query is received, it’s translated from the Hybrid Polystore query language, Typhon QL, into native queries suitable for the various Database types that users define for their models at design time. Then, the results derived from each different database backend are aggregated by the Query Engine, which generates a uniform result representation, which is then returned as a standard HTTP response.

2.1.8 Analytics Service

The Integrated Polystore Platform relies on the Analytics Service for authorization and query statistics collection. It is deployed within the platform as a Java service that includes a message broker based on Kafka[1] and a real-time processing engine based on Flink[2]. The Polystore API

[1] See https://kafka.apache.org/
[2] See https://flink.apache.org/flink-architecture.html

communicates with the Analytics service only via the message broker. New query requests are published to the “Pre” topic. After a series of validation and authorization checks are completed, they are flagged as “authorized” or “unauthorized” and subsequently re-published to the “Auth” topic, where the API is listening. After queries are executed, the Polystore API publishes the results to the “Post” topic for analysing and calculating statistics. The Analytics service may also request inverted queries, which then the Polystore API executes and attaches the result to the Post event object. The number and the nature of the query statistics depend on the actual Analytics Service implementation which may be different for each case of Integrated Polystore Platform deployment. Developers are able to implement deployment-specific metrics and statistics by creating their own Analytics service implementation. It must reference the base Analytics Java library that provides the core functionality and enables cross-component communication via the messaging broker. The usage of the analytics component is optional.

2.1.9 Storage Databases

Storage Databases are the places where actual data is persisted and retrieved. Their number, type (relational, document-based etc.) and specific properties are not fixed, since they depend on specifications set by the user, at design time, through Typhon DL. The instance types (Domain Classes) for which a Storage Database is responsible for handling are defined at design time too, using Typhon ML. Each storage Database is set-up inside an isolated container, which is generated during the build process. The only component that directly connects to the Polystore Storage Databases is the Query Engine. It is fully aware of all the characteristics defined at design-time and is able to perform all the required operations to retrieve or store data from each different database instance using the corresponding API. At the moment, Relational, Document, KeyValue and Graph databases are supported by all the components of the Integrated Platform. Provisions and placeholders also exist for Hive databases, but the underlying implementation in various components is not yet completed.

3. WORKFLOWS

This section presents a high-level analysis of how the integrated components function as parts of the underlying workflows that serve the platform’s main tasks: Design, deployment and usage of the Polystore. Figure 1 below shows the finalized interacting components and their position in the overall architecture of the Polystore.

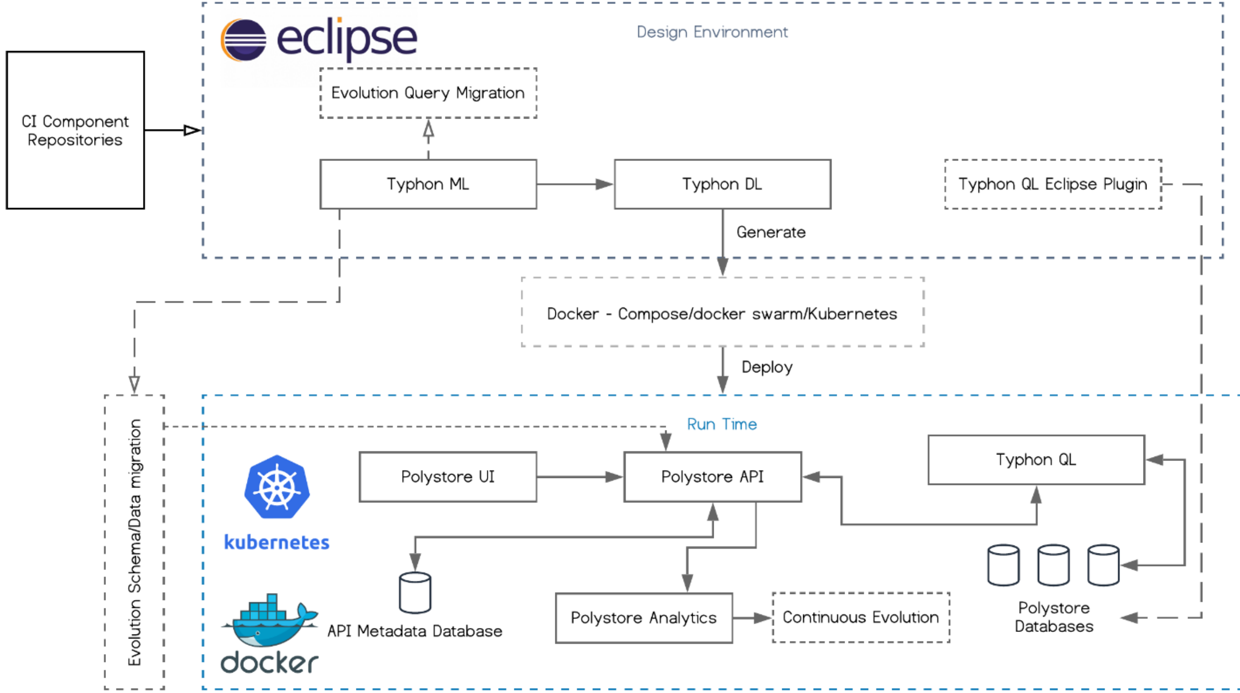


Figure 1: Architecture of TYPHON Platform

3.1 DESIGN

The design task is not done on the Integrated Platform itself, but it is performed on an Eclipse installation containing Typhon ML & DL plugins. The ML plugin is used to design the model to be used and to indicate on which databases the classes created should be stored. The DL plugin is responsible for deploying all necessary Polystore software, based on the ML model. Optional software, such as the analytics component, is also included if the user wishes so. The end result of the DL plugin operations are deployment scripts for the user's chosen platform.

3.2 DEPLOYMENT

The deployment process for the Polystore is initiated using the deployment scripts generated by Typhon DL. These can include analytics and evolution components if the user indicated so during the design phase. By default, Docker is used as virtualization tool. The resulting containers can be deployed using Docker Compose, Docker Swarm or Kubernetes, which can be used to enable horizontal scaling for stateless components, such as the API and the QL Server as well as the replication of databases. Additionally, if the user indicates that they would like the deployed backends to be accessible externally, the deployment process makes the databases public. Apart from the backends, during the deployment process the various components needed by the Polystore, such us the API, the UI, the metadata database and the QL Rest Server are also deployed (user can optionally also configure deployment for analytics & evolution components). Furthermore,

additional configurations can be made through the DL Creation Wizard to explicitly set allowed resource usage. The deployment process is finalized as soon as all the containers are up & running.

3.3 RUNTIME

3.3.1 Query Execution

The main functionality offered by the runtime environment of the Polystore is query execution. A Polystore query can be submitted in various ways. It may be directly submitted to the corresponding Polystore API endpoint as a regular Http POST request containing relevant inputs (as seen in Section 3.4.6), or through the Polystore UI, which undertakes the task to create the appropriate API request and present the results. Additionally, a Swagger interface is available through the API, which can also be used.

In all cases, the Polystore API receives the Query request and, if the analytics component is available, forwards the Query to the analytics service for authorization (as seen in Section 2.1.8). If authorization checks succeed, the API executes the query using the Typhon QL Rest server. Queries, which are always written in Typhon QL, are processed by the Query Engine and results are returned as a uniform Polystore response.

At this point, the Polystore API publishes the results to the “Post” topic of the Analytics Component, so that an array of metrics and statistics is extracted, according to the needs of each individual Polystore deployment.

Query results, except for being sent to the Analytics Component, are “packaged” in JSON format and returned into a regular Http response, which is sent back to the initiator client to be consumed accordingly. In case the initial query was submitted by the Polystore UI, it is received, parsed and displayed to the user. If the Analytics component is not used, the received query is forwarded to the QL server and executed regardless of authorization. Figure 2 below presents the full workflow of a query originating on the Polystore UI and going through the Analytics component.

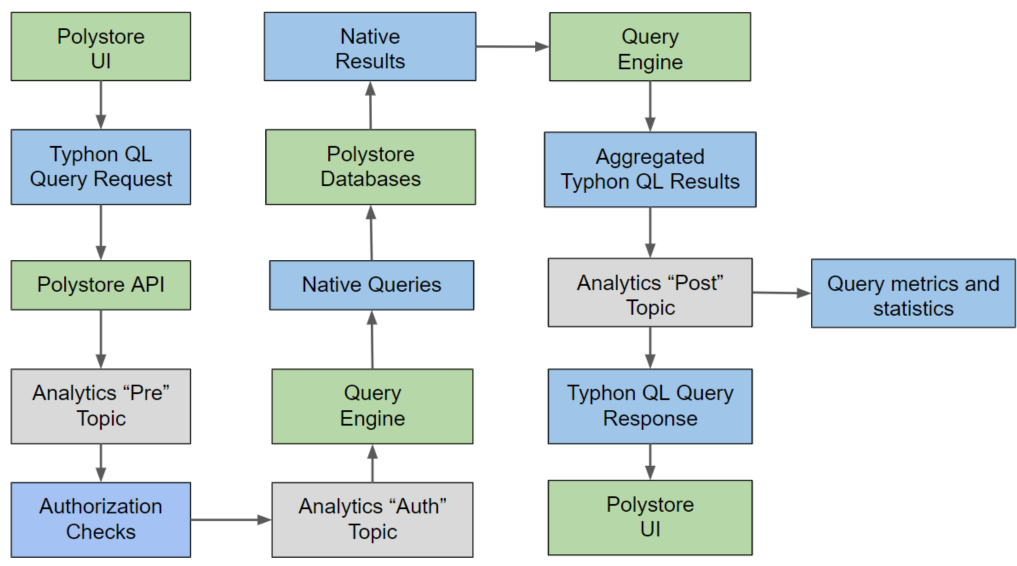


Figure 2: Query Execution Workflow

3.3.2 Evolution

Developers apply changes to their models in the Typhon Workbench, using Typhon ML and Typhon DL and re-build their Typhon project. The latest ML model, containing the change operators, and latest DL model, may then be used with the data/schema evolution tool for the Evolution process to begin.

In case there are new Databases to be deployed at the platform, at the current state of the implementation, an administrator user must manually execute the deployment scripts, as generated based on the DL model, for the new Database containers/instances to be deployed.

For schema changes, the ML model containing the relevant change operators are processed by the data/schema migration tool. It consumes the submitted change operators and performs all the required changes to databases, schemas and data, using QL Queries through the relevant API services. On each individual change, the resulting schema is uploaded automatically to the API. In case of a migration failure, the latest attempted change is either rolled back or discarded. The user can check the status of the evolution in the console where the tool is executed.

When the changes (migration) are completed, the final schema is uploaded to the Polystore API and inserted to the metadata database. Additionally, the user can make changes to commonly used queries to be in line with the newly evolved schema, using the query migration tool through Eclipse.

Finally, using the Continuous Evolution tool, the user is able to get helpful performance statistics about the Polystore schema and relevant queries, as well as recommendations about potential changes that will improve performance. The workflow of the schema/data migration operations can be found in Figure 3 below.

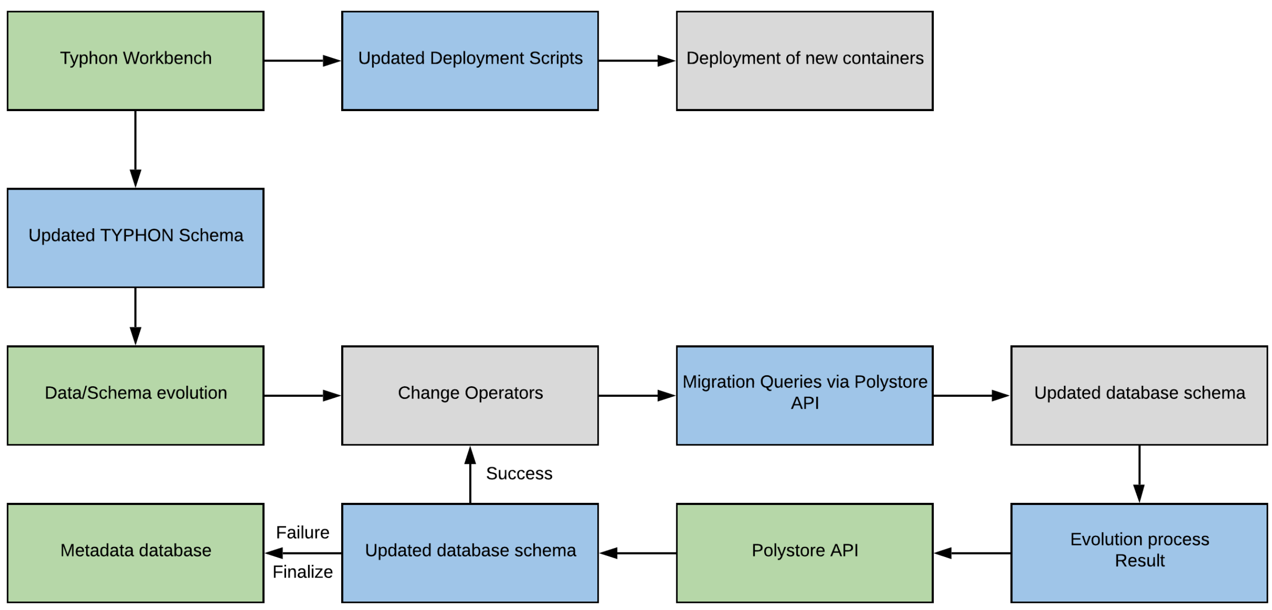


Figure 3: Evolution Workflow

4. HYBRID POLYSTORE API & UI

The Hybrid Polystore Application Programming Interface (API) is one of the core components of the Integrated Platform. The API provides centralized access to all user accessible functions of the Polystore and acts as an orchestrator, essentially being the central connector between the interconnected components.

The User Interface (UI) of the Hybrid Polystore was developed in order to provide easy access to the Hybrid Polystore functionalities to users without them having to evoke any API calls. The UI development is in tandem with the API development, ensuring that additional services and capabilities added in the API are readily available through the UI.

The finalized versions of the Polystore API & UI are detailed in the sections below. Usage of the functionalities mentioned is presented with examples on the User Guide in Section 6.

4.1 API SERVICES

4.1.1 User Services

Apart from the Polystore services, the API contains user services in order to enable authentication across the Polystore. The users will be able to authenticate against the API and then have access to the rest of the Polystore functionalities. The current endpoints that the API offers are:

- Register: register a new user
- List: get all the users of the Polystore
- Get: get specific user via username and list their respective attributes
- Delete: delete the specified user via username

4.1.2 Status Services

The API offers services designed to inform the end-user about the current status of the platform. By querying the Status service, the response will indicate whether the Polystore is up or down. Depending on the state of the Polystore, some functionalities could be temporarily unavailable. The Polystore goes down when the manual Down service has been called externally by an authorized user. In most cases, this would mean an authorized component calling the down service to perform critical operations, such as the evolution component. Similarly, the Polystore will automatically return to the Up state as soon as the critical operation finalizes, or if no operations are pending, when an external user evokes the Up service. As a note, the actual container that the Polystore API resides in is not affected, but rather functionalities are not allowed when the status is Down. In short, the endpoints available for the Status services are the following:

- Up: brings the Polystore Up if no critical operations are pending
- Down: brings the Polystore down
- Status: returns the state of the Polystore

4.1.3 Model Services

In order for the API to act as an orchestrator, there is a need for it to manage the ML and DL models, two of the most important building blocks of the Polystore, which are used by other components to get connection details for databases in the case of DL, or get the schema details, in

the case of ML. The Model Services allow authorized users to download and upload both of these models. As mentioned above, these models are stored in a metadata database. The API has a versioning system in place for both of the models. These models are used by the QL Server for database initialization. The endpoints for the Model Services are the following:

- Get DL: gets the latest DL model or the model matching the version input
- Set DL: uploads a new DL model
- Get ML: gets the latest ML model or the model matching the version input
- Set ML: uploads a new ML model

4.1.4 Query Services

The query service enables authorized users to send Typhon QL queries to the API for execution. After a request has been made, the query is validated by the analytics component and if the query passes authorization, it is forwarded to the Typhon Query Engine for execution. If the analytics service is not available, all queries passed to the API will be executed. If the query is successful, the API then returns the results to the end-user. The query endpoints are the following:

- Query: executes “select” queries
- Update: executes “update” and “insert” queries
- Prepared Update: executes batch “update” and “insert” queries

4.1.5 Backup/Restore Services

The backup and restore services are used by authorized users to manage data across all Typhon Polystore databases. The backup service takes as input the database name and a desired backup name and then uses native database clients to run the backup process on the selected database. The resulting file is stored on the API server and the filename is returned to the user. The restore service works in the same principle, as the input is the database name and the backup file name. After making sure that the database/filename exists, the API evokes a native client for the specific database type and runs a restore process based on the backup file. Additionally, the user can use the download endpoint to download directly the backup file. In conclusion, the available endpoints for the backup/restore services are:

- Backup: connects and backups database using native client
- Restore: connects and restores database using native client
- Download: used by user to download backup files

4.2 POLYSTORE UI

The Polystore UI has been designed for ease of use and supports all the functionality of the API listed above. Users will be authorized against the User database of the API and after a successful login, they will be able to access all relevant functions and status updates depending on their role.

The UI comes pre-packaged with the Hybrid Polystore as one of the main components. It was built using popular web technologies and as such, is easily accessible from any device that supports a

web browser. Implementations for all API operations have been made on the UI, and as such, it can be used to perform any of the functions listed above through its graphical interface.

5. Continuous Integration & Deployment Process

To ensure fast, reliable and continuous delivery of the latest updates to teams involved in testing and development of the Typhon service, a versatile and modern continuous integration and deployment process based on Jenkins[3] was designed and implemented.

During the second half of the project, the infrastructure was moved to a dedicated server in order to be able to meet the computational and storage needs of the various components, while the complexity, size and amount of build iterations were increased.

5.1 Overview

The whole procedure consists of 5 stages. Initially, a developer pushes changes to the master branch of a Typhon component project. Our Continuous Integration Infrastructure detects these changes and triggers a validation process, in order to confirm that everything works as expected without any issues. When validation finishes, all appropriate users receive an automated email that informs them about the validation result. The next action undertaken by the Continuous Integration Infrastructure is to create new versions for each plugin, by building them using the updated source code. Finally, the contents of the Eclipse update site are refreshed, so that the latest distributions become available for download.

During the last half of the project, significant effort was made by technical partners to unify their CI processes regarding Eclipse plugins & update sites. The result was a homogenous deployment process and a unified Eclipse plugin repository, which significantly simplifies the installation process (as seen in Section 6.1.3). As such, the Continuous Integration process for ML, DL, QL and Evolution Eclipse plugins is very similar.

5.2 Typhon Eclipse plugins

When Continuous Integration infrastructure detects newly pushed code to the master branch of any of the components of Typhon project, it pulls it immediately from the corresponding GitHub repository. These projects are then built using Maven & Tycho[4]. Usually, this entails of building a mavenized java parent project, which uses features bundles and modules to build their subcomponents. One of these subcomponents is a p2 update site for the plugin. Proper functionality is guaranteed by running a series of unit tests for each generated project. On a successful build, p2 update sites are deployed to the distribution folder.

[3] See https://jenkins.io/
[4] https://www.eclipse.org/tycho/

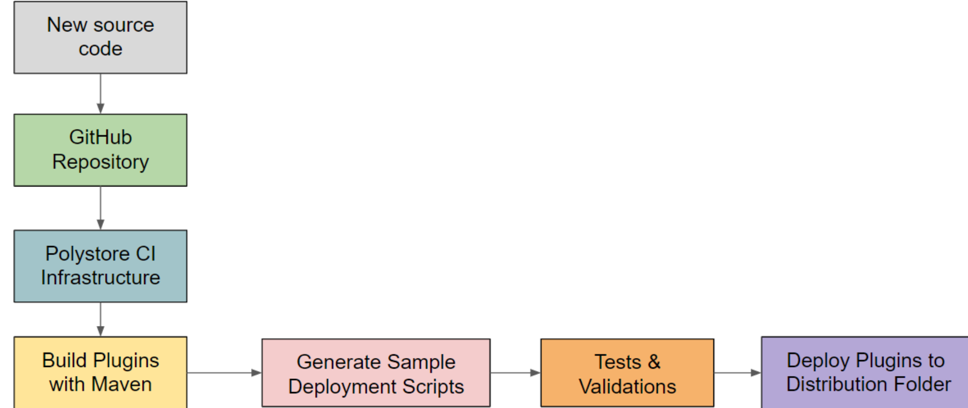


Figure 4: Typhon DL CI Workflow

5.3 POLYSTORE API

A similar process takes place for the continuous integration purposes of the Polystore API component. As soon as the CI infrastructure is notified for source code updates, it pulls it from the repository and attempts to build the Polystore API through Maven. The API was modified to use Google's Jib[5], which allows Maven to also build and publish a Docker image based on the API code and a custom built base Docker image, which contains an orchestration script (wait-for-it.sh[6]). This orchestration script allows the API to wait for the Polystore Metadata database to be functional before trying to connect to it, thus reducing errors and retry attempts.

In a successful Maven build, the "fresh" Docker image is pushed to its corresponding Docker Hub repository. The Typhon DL deployment script is able to take delivery of the latest stable API Docker image just by downloading it using the shared repository URL. In the previous release of the Polystore API, it would also include and build the Polystore UI every time a code change was detected. As this was adding additional complexity and load on the CI infrastructure, the UI was removed from the build process of the API and was moved to a standalone repository with its own CI/CD project, configuration and deployment process.

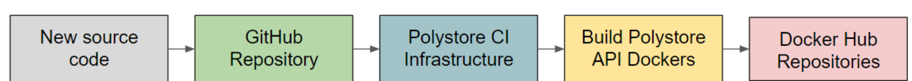


Figure 5: Polystore API CI Workflow

5.4 POLYSTORE UI

As mentioned previously, for the final release of the Integrated Platform, the Polystore UI build and deployment process was separated from the API. The deployment process of the UI is based on a Dockerfile, which the CI/CD infrastructure makes use of to create the corresponding image. The Dockerfile contains instructions for the installation of the dependencies of the UI and the commands required to execute the UI process. Changes were made to the base image and the

[5] https://cloud.google.com/java/getting-started/jib
[6] https://github.com/vishnubob/wait-for-it

dependencies used to make the image more lightweight, resulting in a size reduction of almost 800 MB.

5.5 EVOLUTION

The evolution toolset also makes heavy use of the CI/CD infrastructure to build its individual tools. On a detected code change, the Jenkins Evolution project has been configured to build both standalone tools (data/schema migration & data ingestion tools) as .jar files. On a successful build, these standalone tools are deployed to a static folder residing in the main project repository, publicly available for download.

Additionally, the build process also builds the Query evolution Eclipse plugin in a similar manner to what was described in section 5.2, while also building Docker images for the Continuous evolution tool The result of this process is, assuming a successful build, three Docker images published to the public Docker repository. These images are (1) the Kafka consumer which captures and analyses the QL queries on-the-fly, (2) the Typhon Evolution Analytics Client and (3) the Typhon Evolution Analytics Backend, which can then be retrieved by the DL deployment scripts for use in the Polystore.

5.6 TYPHON QL

The Typhon QL component, apart from the Eclipse plugin process mentioned earlier, also utilizes Jib to build a Docker image for its REST server. As such, each time a code change is detected, Jenkins is notified and begins the process as configured in a special file called Jenkinsfile. This file allows for more advanced configuration via code rather than via the preset options allowed in the GUI of Jenkins' configuration page. For the case of this component, the pipeline process checks out the latest code from Github, builds the QL Eclipse plugin and the QL server and deploys them to the unified update site and to the public Docker registry, respectively.

5.7 ARCHIVA REPOSITORY

An Apache Archiva[7] repository was also created and hosted on the CI/CD infrastructure. This repository is used to store various build artifacts and libraries stemming from the various components. These artifacts are then used as Maven dependencies where needed. Additionally, some commonly used dependencies across components are also hosted in this repository, allowing us to have a centralized place for dependency resolution and versioning of component libraries.

5.8 DISTRIBUTION PROJECT

When the continuous integration and deployment process for one of the Typhon project components – as mentioned above – finishes successfully, the Distribution Jenkins project is triggered. Its goal is to build and deploy an updated Eclipse distribution ready to be downloaded via the unified Eclipse Update Site. To achieve this goal, a new Eclipse Update Site, containing all the generated plugins, is automatically created using a custom node.js script.

[7] https://archiva.apache.org/

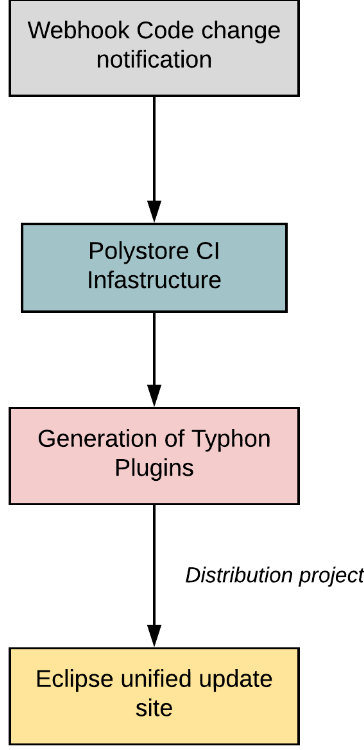


Figure 6: Distribution CI Workflow

5.9 GitHub Integration

Projects for Typhon components use GitHub repositories for the storage and versioning purposes of their source code. Our Continuous Integration infrastructure must be able to initiate the integration and deployment processes for each project, as soon as any new source code is pushed to the master branch of a project repository. This requirement is fulfilled thanks to the GitHub web-hooks. They allow external services to be notified when certain events, related to a GitHub project, happen. In this case, when a new commit is pushed to the master branch, our Continuous Integration infrastructure receives a call to a specific endpoint and triggers the appropriate process. As soon as this process finishes, the corresponding project readme files are updated so that they display the latest build status of the project.

5.10 Slack Integration

While not a critical part of the CI/CD infrastructure by any means, a Slack workspace was set up for the needs of the project. This proved to be very useful, as it allowed us to have real time discussions on integration, interdependency and other critical issues. Additionally, every build process described in the Sections above notify specific Slack channels (for each component) of the status of the build. These real time notifications were immensely useful for quick bug & problem resolution. Finally, we also integrated GitHub issues for all component repositories with Slack as well, which also enabled real time notifications on issue creation and comments.

For the remaining months of the project, use case partners were also given access to the Slack workspace, allowing for more direct communication that will hopefully allow the technical partners to quickly rectify any issues that might occur going forward.

6. USER GUIDE

6.1 INSTALLATION

In the previous version of this guide, the installation of an Eclipse IDE Distribution, containing the latest version of the required tools for Typhon development, could be done in two ways: either adding a standalone distribution from the Typhon Eclipse Distribution Site or by downloading the official Eclipse installer and manually installing the required tools. As various modules and dependencies changed and became deprecated, for the latest working version it is safer to start from a clean install of Eclipse and download all needed dependencies manually, as this process seems to produce less errors overall across users who tested it. As such, the standalone guidelines have been removed from this version of the guide.

6.1.1 Installing Eclipse

The official Eclipse installer may be used to create a fresh Eclipse IDE installation. Typhon plugins must be manually installed as described in this section. The benefit of this option is that plugin updates are automatically detected and easily applied, without the need to reinstall a new standalone distribution. If there is already an active Eclipse installation, Typhon plugins may be installed through their corresponding Eclipse Update Site.

To download the installer visit the official download site (https://www.eclipse.org/downloads/)

Download Eclipse Technology
that is right for you

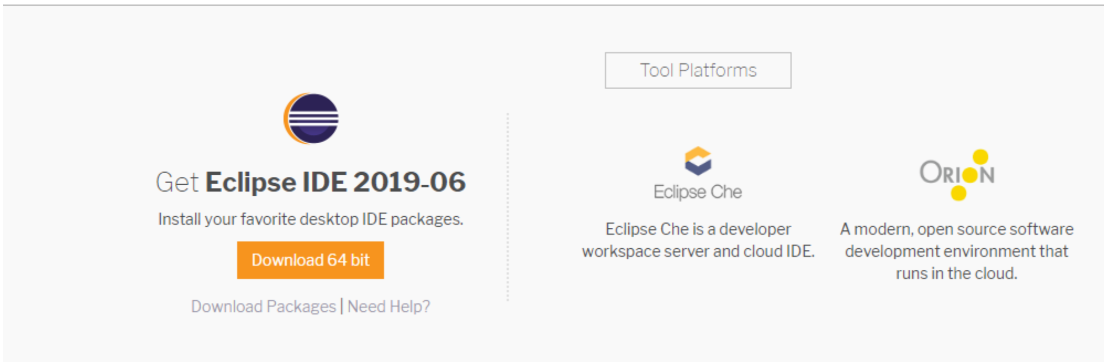


Figure 7: Downloading Eclipse Installer

Once the download finishes, run the installer executable file and choose to install Eclipse IDE for Java Developers.

Figure 8: Installing Eclipse for Java Developers

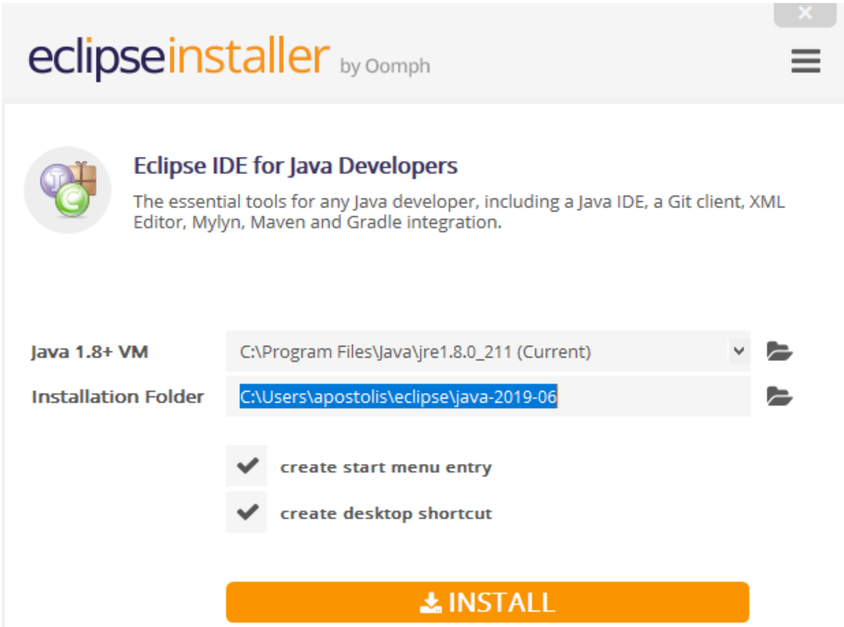


Figure 9: Eclipse Installation

The next step is to install all the plugins required to start Typhon Development, through the *Install New Software* option:

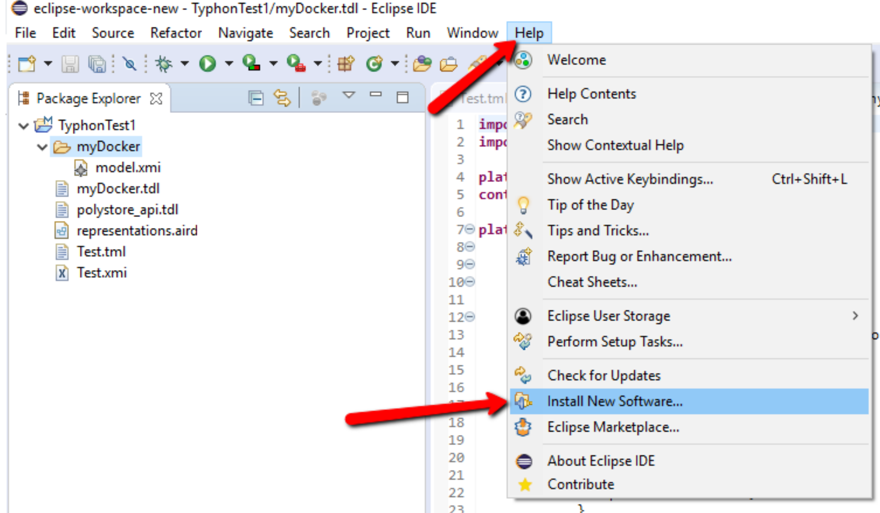


Figure 10: Installing New Software to Eclipse IDE

6.1.2 Installing Docker

To deploy the Polystore to your machine, you will need to have Docker installed. You can get Docker from here:

https://docs.docker.com/get-docker/

This installation will also include Docker Compose and the Docker Swarm mode. Tools for orchestrating a Kubernetes deployment of the Polystore with kubectl can be downloaded here:

https://kubernetes.io/docs/tasks/tools/install-kubectl/

Tools to run a Kubernetes cluster on your machine can be found here:

https://kubernetes.io/docs/tasks/tools/

6.1.3 Typhon Toolset

Install **ML**, **DL, QL and Evolution** Typhon plugins as shown in Figure 10 using the URL:

http://typhon.clmsuk.com

The tools will automatically resolve and download any needed dependencies.

6.1.4 Optional Libraries

In order to use the graphical representation for the schema offed by ML, you will have to add Sirius to your Eclipse installation. This can be done by using the update site URL below:

http://download.eclipse.org/sirius/updates/releases/6.3.2/2019-06

Install

Available Software

Check the items that you wish to install.

Work with: http://download.eclipse.org/sirius/updates/releases/6.3.2/2019-06 Add... Manage...

type filter text Select All

Name	Version
☑ Sirius	
☑ Sirius Dependencies	
☑ Sirius SDK	
☐ Uncategorized	

Deselect All

68 items selected

Details

☑ Show only the latest versions of available software

☑ Group items by category

☐ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

☐ Hide items that are already installed

What is already installed?

< Back Next > Finish Cancel

Figure 11: Sirius Installation

6.2 Usage

The following section describes the usage of the TYPHON toolset. It is assumed that user has already downloaded and installed the various tools and plugins needed as described in the previous section.

6.2.1 Design Example

To start developing for Typhon, open the Eclipse IDE and **Create a new project**

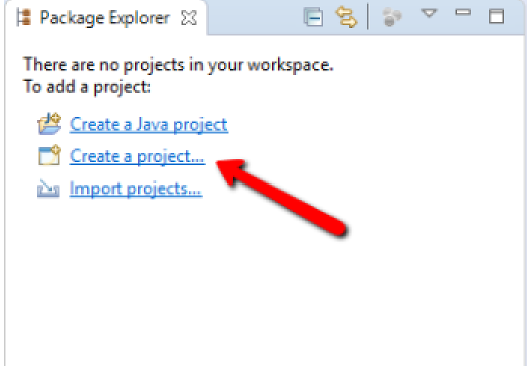


Figure 12: Creating new Java Project

Select **Project** under General wizard category and a project will be created. **Alternatively,** if you want to use the graphical representation of the schema you will need to create a **Sirius** project (you will have to install Sirius as per Section 6.1.4

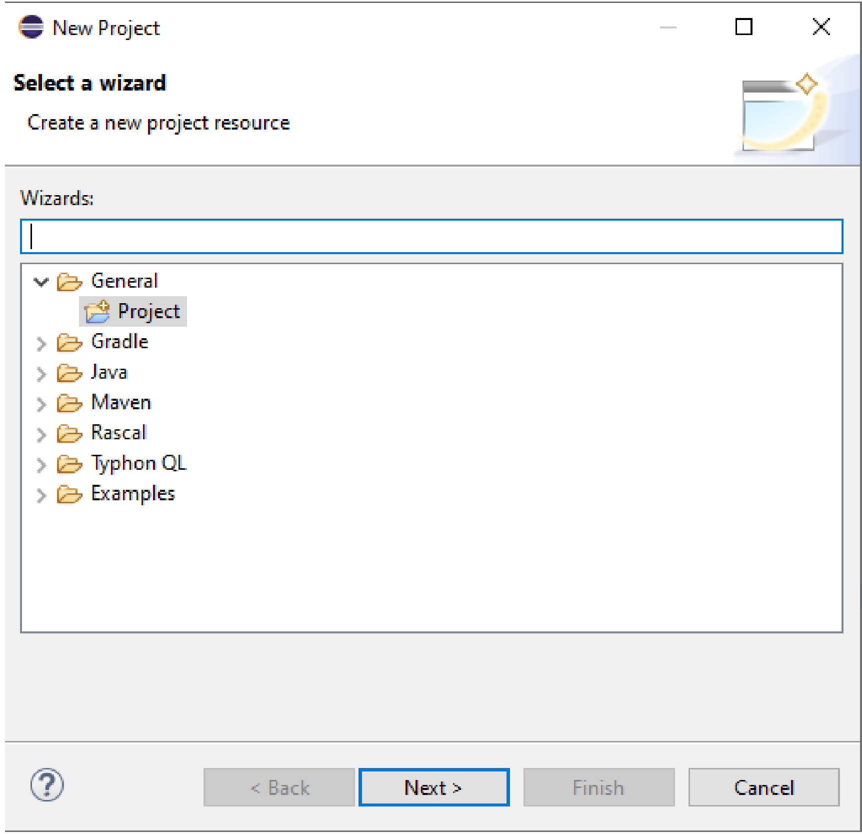


Figure 13: Creating Project

Create a new Typhon ML file, give it a name and add the **.tml extension**.

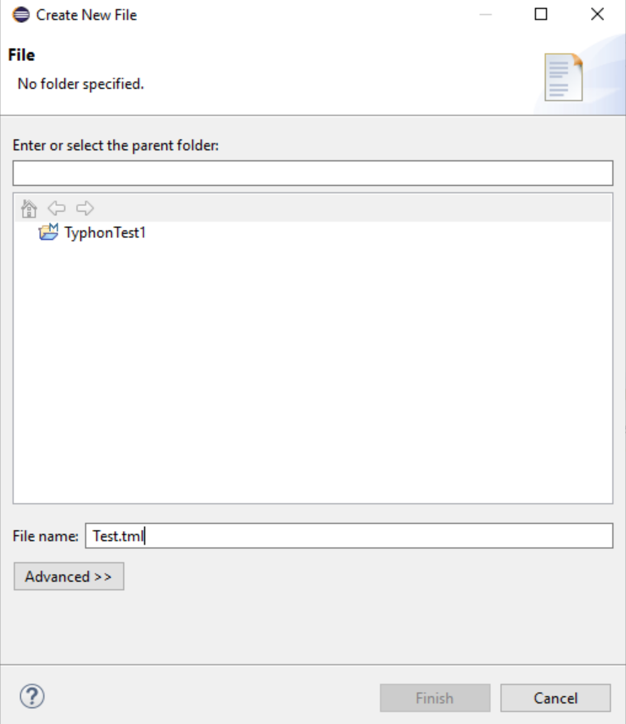


Figure 14: Creating TML file

After clicking Finish, you will be prompted to convert the file into an Xtext file. You will need to click **YES.**

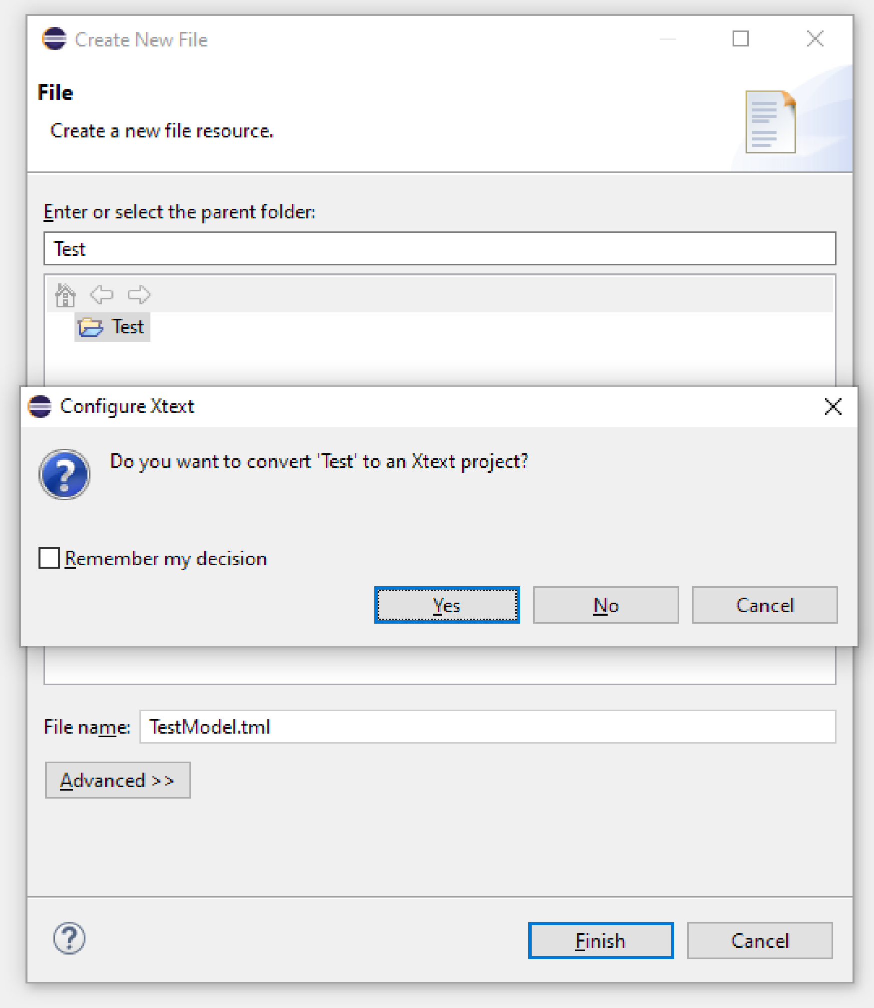


Figure 15: Xtext project prompt

6.2.1.1 Simple Example of Usage

A very simple model, consisting of two classes, will be used for the rest of this Quick Start Guide. There is a *Product* class with id and name properties that has a zero to many relationship with the *Review* class. A relational database stores product records and a document database stores the reviews. To create this model, edit the contents of the .tml file you have just created and enter the following code:

```
entity Review{
    id: string[20]
    product -> Product [1]
}

entity Product{
    id: string[20]
    name: string[50]
    description: string[255]
    review :-> Review."Review.product"[0..*]
}


relationaldb Reviews {
    tables {
        table {
            Review : Review
        }
    }
}


documentdb Products {
    collections {
        Products : Product
    }
}
```

Figure 16: Simple ML example

To transform this ML code into a model, right click on the .tml file, and select **Typhon -> Inject to model**. This will generate a visual representation of the model classes and their relationships, creating an xmi file containing model data.

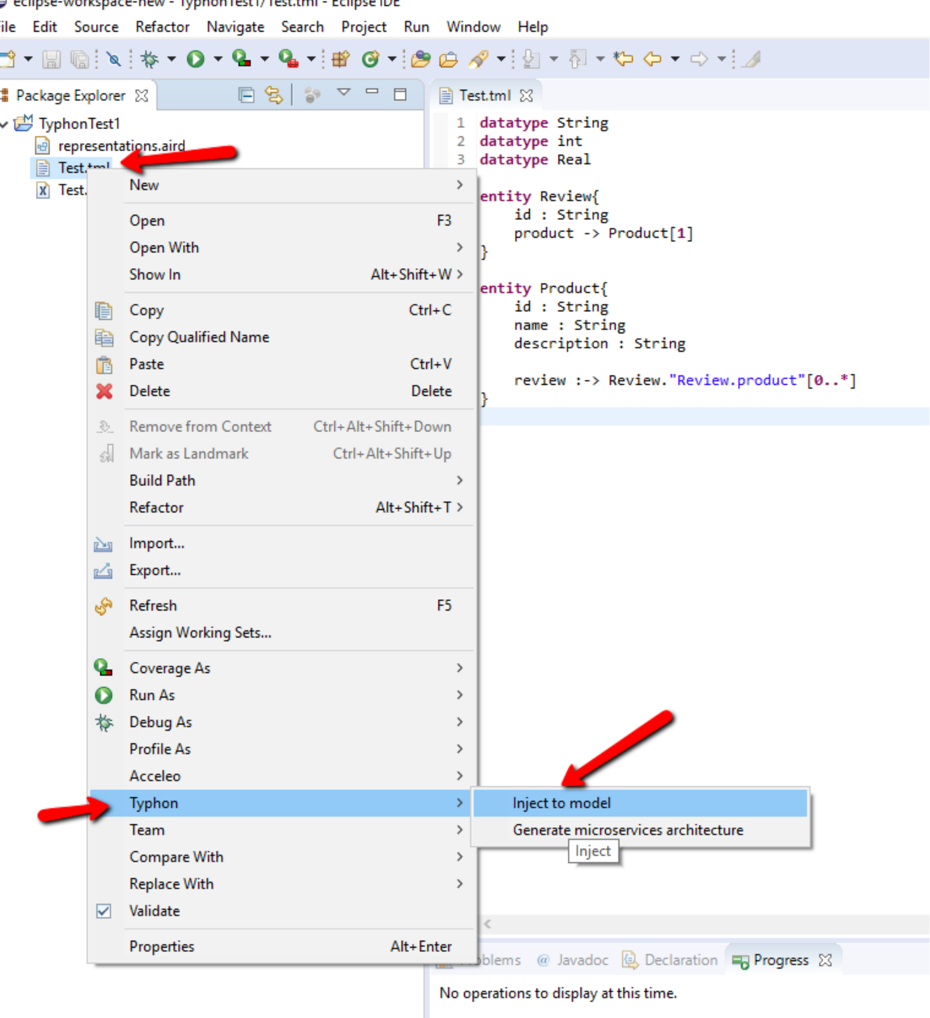


Figure 17: Creating XMI file from TML

If you opted to create a Sirius project, to visually inspect the model, open the generated **representations.aird** file, select **typhonML** as representation and click **New…**

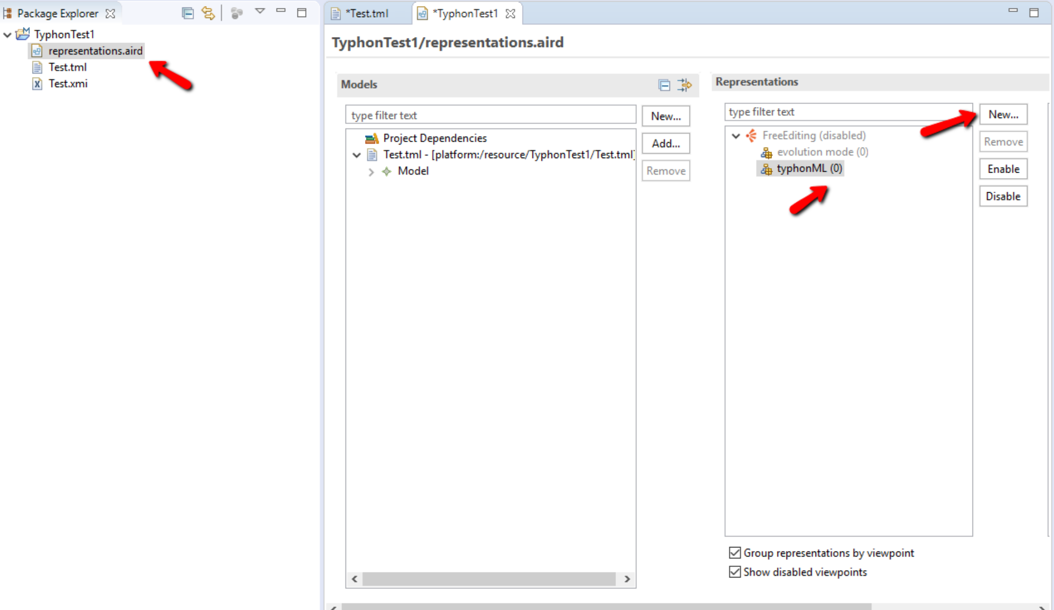


Figure 18: Creating Visual Model Representation

The visual model representation should look like this:

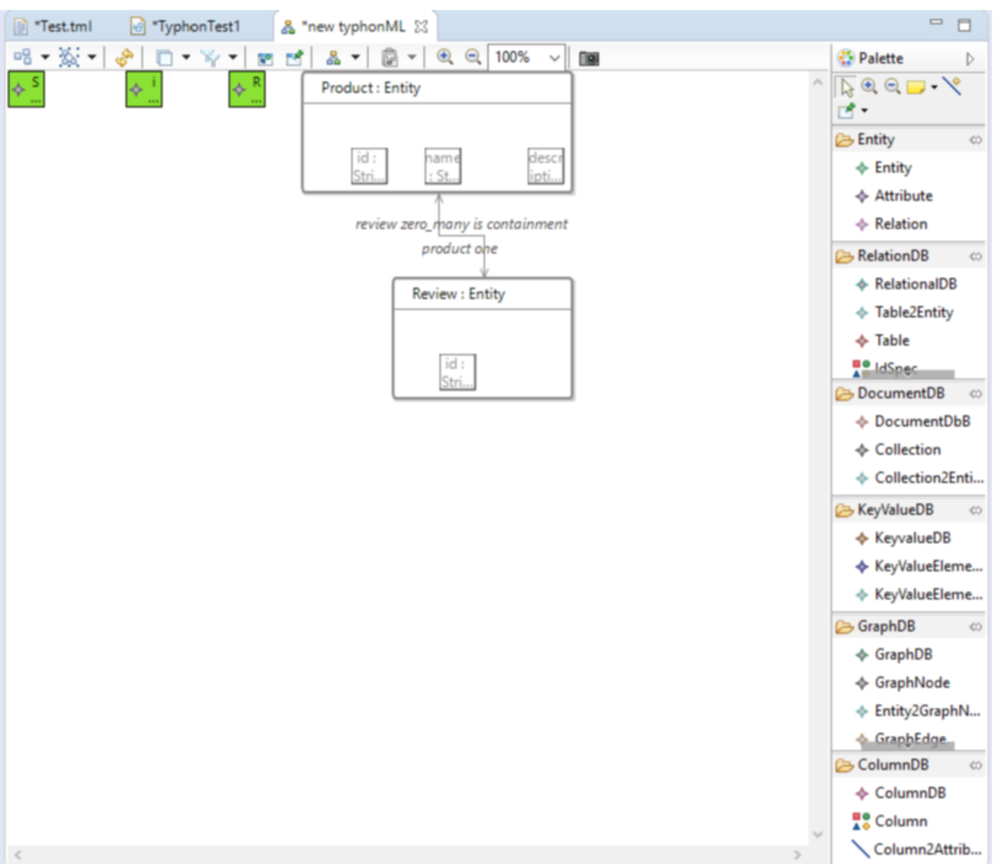


Figure 19: Visual Model Representation

At this point, the class design has finished and you may proceed to create the Typhon DL model. To do so, right click on the generated .xmi file and select **TyphonDL -> Create Typhon DL Model**.

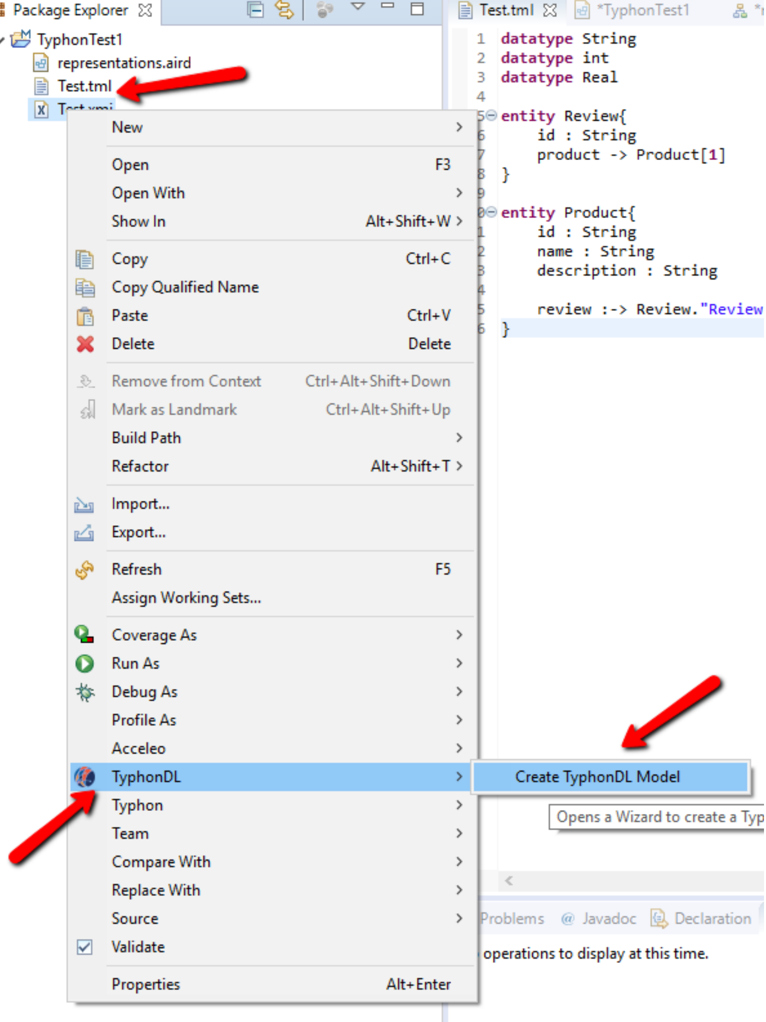


Figure 20: Creating TDL file

You can then follow the instructions on the DL wizard to complete the deployment script generation. For the sake of simplicity, for this example we chose **docker-compose** as our deployment method.

After the DL wizard is finalized, the next step is the creation the deployment scripts based on the Typhon DL. Right click on TyphonDL model file (.tdl) with the name that was given in the wizard file and Select **Typhon DL -> Generate Deployment Scripts**.

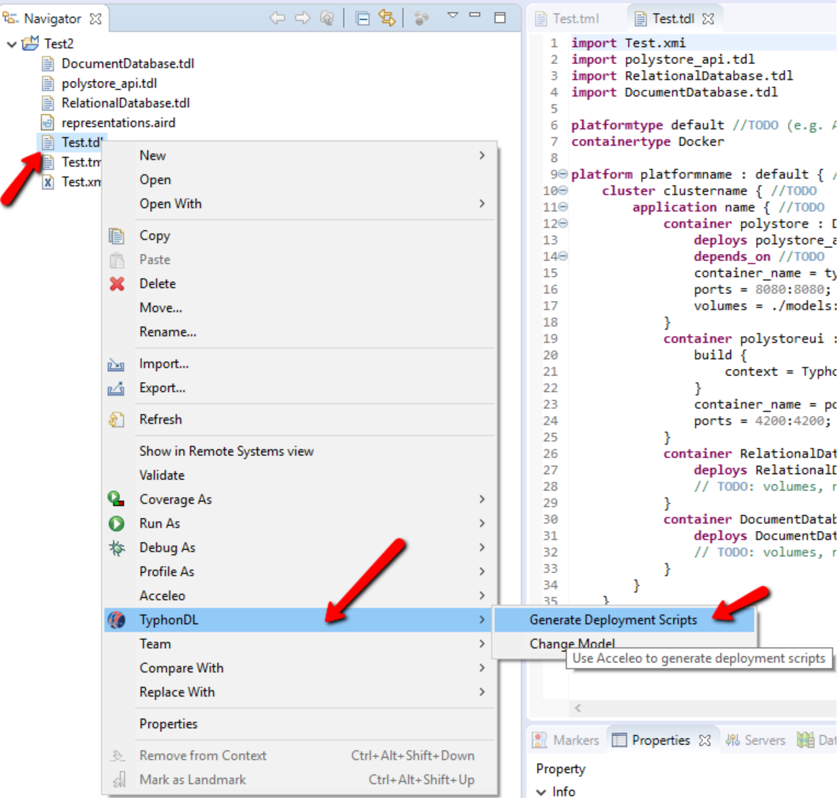


Figure 21: Generating Deployment Scripts

This action will add a folder containing a **model.xmi** and a docker-compose yaml.

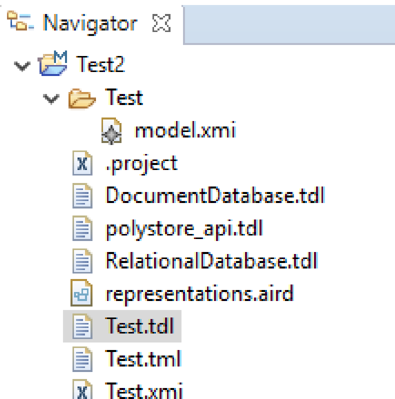


Figure 22: Sample Project files after model XMI file generation

For details concerning the use and capabilities of the DL module, you can read Section 6.3.2.

6.2.1.2 Launching of Polystore

To finalize this example, the docker compose file must be built and deployed. To do so, go to the folder where it has been generated, open a terminal/powershell window and type:

```
docker-compose up --build
```

to start it. You can stop the running Polystore instance by pressing Ctrl + C. After this, to safely remove everything you can do

```
docker-compose down
```

and

```
docker-compose rm -v
```

This will remove all containers and volumes allowing the Polystore to be correctly recreated the next time. For different deployment methods, you can read Section 6.3.2.5.

6.2.2 Runtime Example

6.2.2.1 Polystore UI

A short while after performing the aforementioned Docker commands, the Polystore will be available. You will now be able to open a browser and navigate to the Polystore UI web application, which is locally available at http://localhost:4200

You can login with the default credentials, which are:

Username: **admin**
Password: **admin1@**

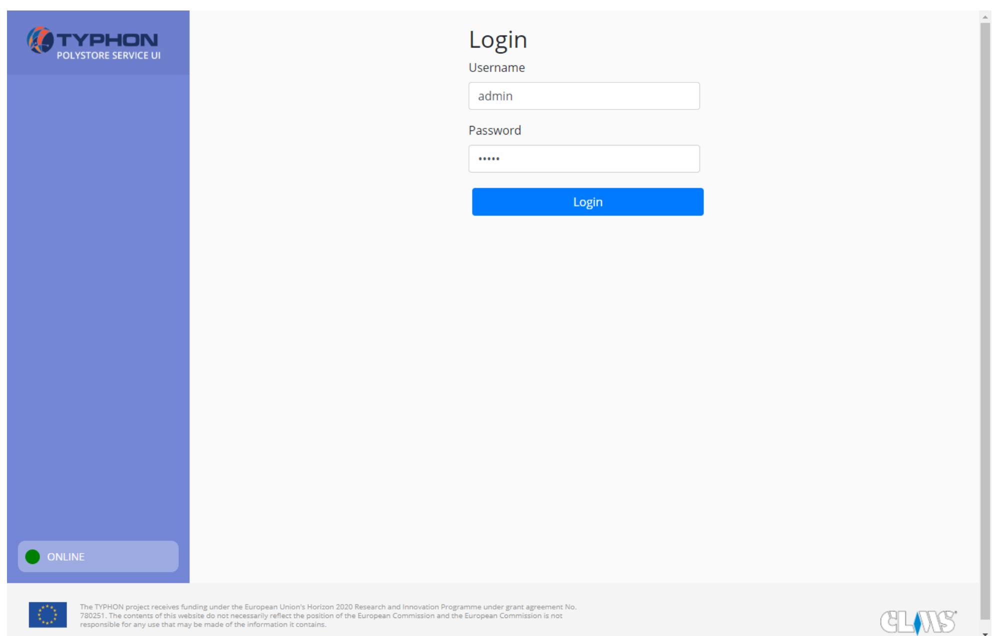


Figure 23: Polystore UI Login Page

Polystore users may be added or removed through the **Manage users** page.

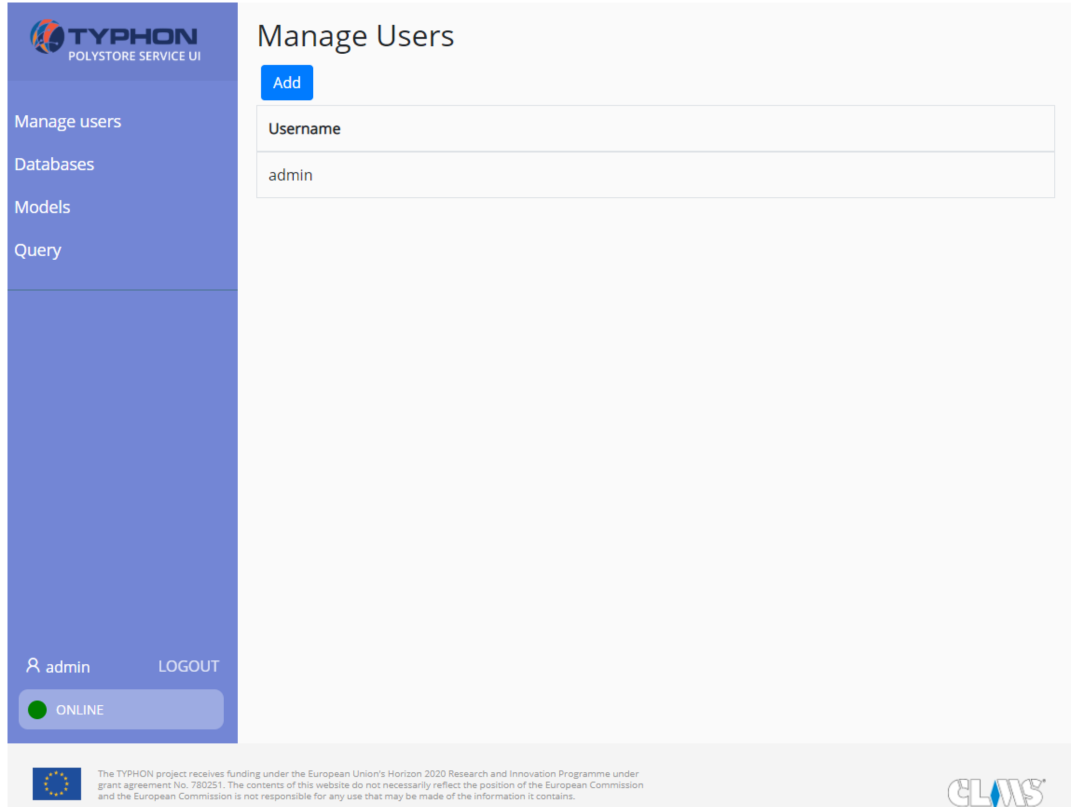


Figure 24: Polystore UI Manage Users Page

To upload the generated Typhon DL and Typhon ML xmi files, go to the **Models** page and click the **Upload new version** button under each section. By default, the first version of both DL & ML .xmi models are uploaded during initialization of the containers.

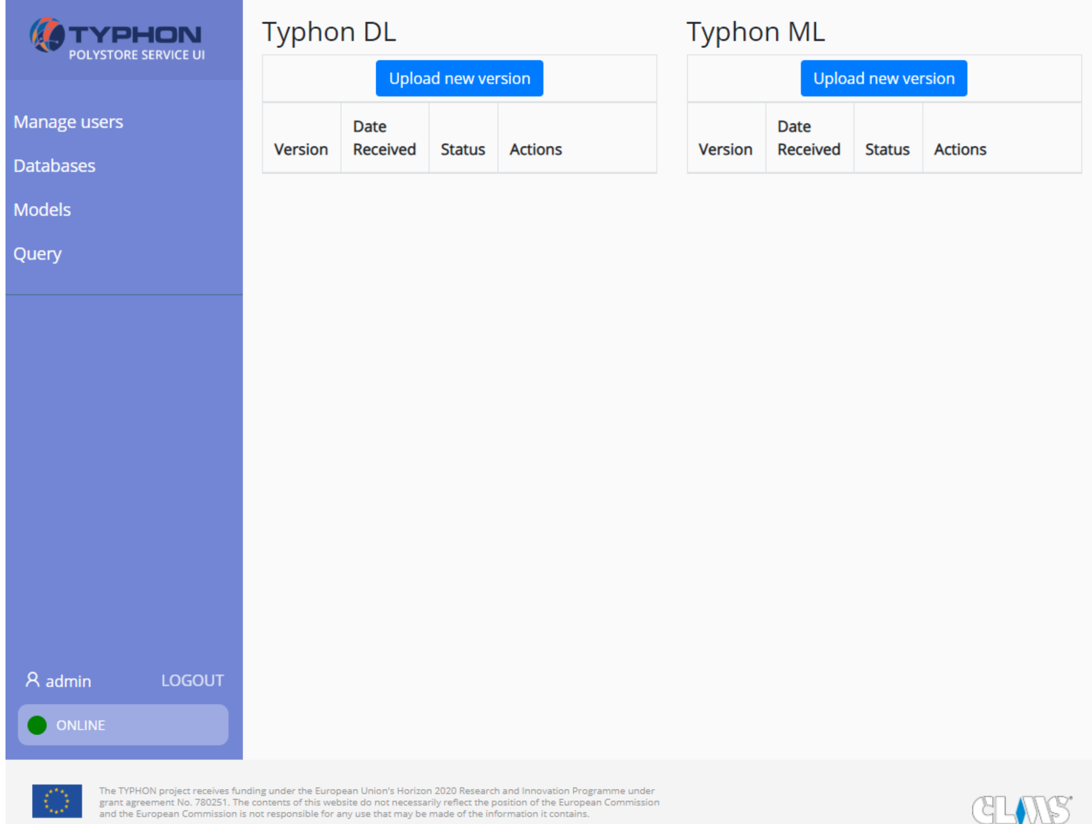


Figure 25: Polystore UI Models Page

Once xmi files have been successfully uploaded, new model version entries will be displayed.
The next step for Polystore usage is to initialize the databases and perform queries. To do this, you either use the UI (next section) or you can skip ahead to Section 6.3.4.6 to perform these operations through the API, or if you wish to perform these through Eclipse you can skip to Section 6.3.3.

The first step to querying the Polystore is to initialize the databases. This can be done through the Query menu.

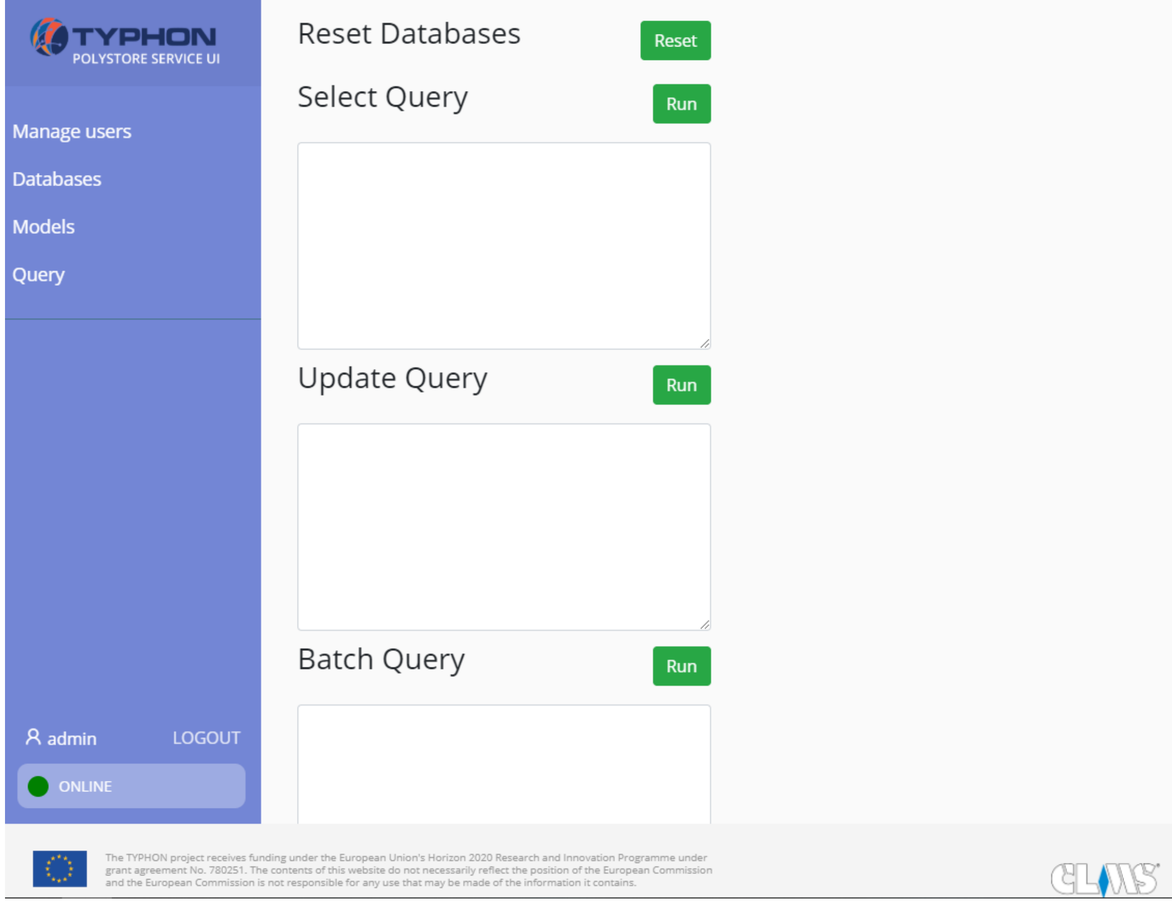


Figure 26: Query Menu

To initialize the databases, you can press the Reset Databases button. After a short while, you will get a 200 OK message, indicating that the databases have been initialized and are ready to use.
You can then use the Select, Update/Insert, or Batch windows to execute corresponding queries. For more information about the inputs needed, you can read Section 6.3.4.6 and section 6.3.5.

6.3 Component Documentation

The extensive documentation in this section will be updated per component online in their respective README pages.

6.3.1 Typhon Modelling Language (ML)

To start a TyphonML model specification, we need to create a new modelling project within a .tml file. The TyphonML editor supports the modeller with the following facilities:

- Syntax Highlighting,
- Background Validation,
- Error Markers,
- Content Assist,

- Hyperlinking, and
- Quick fixes.

In the rest of this section we describe each part of the TyphonML language. In particular, the TyphonML specification consists of 4 main blocks as depicted in Figure 27:

- Custom data types,
- Entities,
- Databases and
- Change operators.

```
//Custom data types
customdatatype address {…
customdatatype zip {…
//Entities
entity Review {…
entity Product {…
entity Category {…
entity Item {…
entity Tag {…
entity User {…
entity Biography {…
entity Concordance {…
entity Wish {…
//Databases
relationaldb Inventory {…
keyvaluedb Stuff {…
documentdb Reviews {…
relationaldb Inventory {…
graphdb MoreStuff {…
// Change operators
changeOperators [
    add attribute star : int to Review,
    add entity Order {…

]
```

Figure 27: TyphonML overview

6.3.1.1 Custom data type

Custom data types allow the modeler to define their data types.

It extends the abstract DataType metaclass in order to enable the specification of custom data types. To this end, each CustomDataType instance consists of different elements, which overall contribute the definition of the new data type being defined. For instance, in order to represent geographical points of interest, users can define an *address* type consisting of four DataTypeItem elements, i.e., street, city, location, and zipcode. Such data items would be of primitive types (e.g., *string*) or custom data types (e.g., *zip*). A custom data type consists of primitive and custom type elements as depicted in Figure 28 where *address* includes of 3 primitive types, i.e., *street*, *city*, and *location*, and the *zipcode* custom datatype. Each element is represented by the name, colons and type.

```
customdatatype address {
    street: string[256],
    city: string[256],
    zipcode: zip,
    location: point
}
customdatatype zip {
    nums: string[4],
    letters: string[2]
}
```

Figure 28: Custom data types

6.3.1.2 Entity

The conceptual entities have a name and consist of attributes and relations. Attribute is a named element, which is defined in terms of the type of elements to be represented. An attribute can be typed as a primitive, or an custom type. The *attributes* are defined by the name, colons and type that can be primitive or custom data types. We report the complete list of primitive data types in the following:

- Int,
- bigint,
- string: this datatype allows to specify the strin max size as parameter (see line 21 of Figure 29),
- text,
- point,
- polygon,
- bool,
- float,
- blob,
- date,
- datetime, and
- text: this primitive data type can be decorated with NLP tasks (see line 22 of Figure 29). The complete list of NLP tasks is reported in the following:
    - ParagraphSegmentation,
    - SentenceSegmentation,
    - Tokenisation,
    - PhraseExtractor,

- NGramExtractor,
- POSTagging,
- Lemmatisation,
- Stemming,
- DependencyParsing,
- Chunking,
- SentimentAnalysis,
- TextClassification,
- TopicModelling,
- TermExtraction,
- NamedEntityRecognition,
- RelationExtraction, and
- CoreferenceResolution.

Relation is a named element, which permits to specify relationships between different entities. In particular, the structural features of such modeling constructs are the following:

- **type**: it permits to define the type of the relationship being specified;
- **cardinality**: entities can be involved in relationships of different cardinalities, which can be singular or multiple;
- **opposite**: when creating a reference from one entity (e.g., named e1) to a second entity (e.g., named e2) it is possible to specify the opposite reference from e2 to e1 in order to define a bidirectional relation instead of two different unidirectional ones.
- **isContainment**: it is a boolean attribute, which permits to specify if the target entity is contained (e.g., to trigger cascade-deletion) or not in the entity being modeled.

The *relations* are represented by the name, the keyword ->, the linked entity, and the cardinality, which can be either *0..1*, *1*, *0..**, and * (see line 27 of Figure 29). A *containment* relation is represented as colons before the -> keyword (see line 28 of Figure 29), whereas a b*idirectional* relation is defined with its opposite relation (see line 26 of Figure 29). Containment and bidirectional relations can occur together as shown in line 30 of Figure 29.

```
20  entity Product {
21      name : string[256]
22      description : freetext [POSTagging [eng_pol], SentimentAnalysis[eng_pol]]
23      price : int
24      productionDate : date
25      availabilityRegion: polygon
26      reviews -> Review."Review.product"[0..*]
27      tags -> Tag[0..*]
28      inventory :-> Item[0..*]
29      category -> Category[1]
30      wish :-> Wish."Wish.product"[1]
31  }
```

Figure 29: Conceptual entity

6.3.1.3 Databases

Currently, TyphonML supports four kinds of database systems, i.e., relational DB, graph DB, documental DB, and key-value DB. We report the syntax of each supported DB system in the following.

Figure 30 depicts an instance of relational DB. It consists of a name (line 70) and tables (lines 71-80). A table maps a conceptual entity (line 73) and allows specifying indexes (lines 74-76) and ids (line 79). The editor accepts and suggests only attributes specified in the mapped entity (line 75) as possible indexes ones or id (line 77).

```
70  relationaldb Inventory {
71      tables {
72          table {
73              UserDB : User
74              index UserNameIndex {
75                  attributes (name, location)
76              }
77              idSpec (name)
78          }
79          table { ItemDB: Item }
80      }
81  }
```

Figure 30: Relational DB

Figure 31 depicts an instance of document DB that consists of a name (line 87) and a list of collections (lines 88-92), where each one maps a conceptual entity (line 89).

```
87  documentdb Reviews {
88      collections {
89          Review : Review
90          Biography : Biography
91          Category: Category
92      }
93  }
```

Figure 31: Document DB

An instance of Graph DB specification is depicted in lines 70-82 of Figure 32, where two entities i.e., *Wish* and *Concordance*, are mapped as graph edges. It worth noting that only entities with more than two *one-to-one* relations can be used as edge, and only a *one-to-one* relationship can be used as a source or a target. In this way, the editor can detect possible edge mapping errors. For instance, lines 73-76 in Figure 32 notify an error because the edge points to the *Wish* entity that consists of a single *one-to-one* relation.

```
60 entity Wish {
61     intensity: int
62     user -> User[1]
63     product -> Product[0..*]
64 }
65 entity Concordance {
66     weight: int
67     source -> Product[1]
68     target -> Product[1]
69 }
70 //Databases
71 graphdb MoreStuff {
72     edges {
73         edge Wish {
74             from user
75             to product
76         },
77         edge Concordance {
78             from source
79             to target
80         }
81     }
82 }
```

Figure 32: Graph DB

Figure 33 depicts an instance of Key-value DB that consists of a name (line 82) and a list of elements (lines 88-92), where each one represents a key-value structure. Each element consists of a name (*User* in line 84), the key name (*userKey*) and a list of entities' attributes (*User.photoURL* and *User.avatarURL*) that the key-value structure maps.

```
82 keyvaluedb Stuff {
83     elements {
84         User { userKey -> ("User.photoURL", "User.avatarURL") }
85     }
86 }
```

Figure 33: Key-Value DB

6.3.1.4 Change operators by example

This section presents the syntax of change operators to support the evolution workpackage. A change operator can be applied on entities, relations, attributes, databases and custom data types. For more details see Section 6.3.6.1. We recap all the operators in the following. Moreover, Figure 34 depicts all the change operators at work.

```
121  // Change operators
122  changeOperators [
123      //Entity
124      rename entity Product as NewProduct,
125      remove entity Category,
126      split entity vertical User to UserData attributes:["User.name","User.address" ],
127      split entity horizontal Biography to NewBiography where "Biography.content" value "empty",
128      migrate Product to Neo4j,
129      merge entities Category Product as Product_Category,
130      add entity Order {
131          attributes [
132              attr : string[255]
133          ]
134          relations [
135              add relation product to Order -> Product
136          ]
137      },
138
139      // Reference
140      add relation newRelation to Biography -> Item,
141      rename relation "User.biography" as bio,
142      remove relation "Biography.user",
143      EnableRelationContainment { relation "User.biography"},
144      DisableRelationContainment { relation "User.biography"},
145      change cardinality "Product.category" as 0..*,
146
147      //Attribute
148      add attribute title : string [255] to Biography,
149      add attribute address : address to User,
150      remove attribute "User.avatarURL",
151      rename attribute "Wish.intensity" as level,
152      change attribute "User.location" : point,
153      change attribute "User.address" : zip,
154
155      //Database
156      rename table "Inventory.TagDB" as TagDataBase,
157      AddIndex {table "Inventory.TagDB" attributes ("Tag.name")},
158      DropIndex { table "Inventory.ItemDB" },
159      extends tableindex "Inventory.ItemDB" { "Item.shelf"},
160      reduce tableindex "Inventory.ItemDB" { "Item.picture"},
161      rename collection "Reviews.Biography" as BiographyExtended,
162      AddCollectionIndex { collection "Reviews.Biography" attributes ( "Biography.content")},
163      DropCollectionIndex { collection "Reviews.Biography" },
164
165      //Custom Data type
166      AddCustomDataType newCustomDataType { first: string[255], second: int}
167  ]
```

Figure 34: Change operators

Within entity we can apply the following change operators:

- Rename Entity (line 124),
- Remove Entity (line 125),
- Split Entity Vertical (line 126),
- Split Entity Horizontal (line 127),
- Migrate Entity (line 128),
- Merge Entity (lines 129), and
- Add Entity (lines 130-137).

Within entity we can apply the following change operators:

- Add Relation (line 140),
- Rename Relation (line 141)

- Remove Relation (line 142),
- Enable Relation Containment (line 143),
- Disable Relation Containment (line 144), and
- Change Relation Cardinality (line 145).

Within attribute we can apply the following change operators:

- Add Custom Data Type Attribute (line 148),
- Add Primitive Data Type Attribute (line 149),
- Remove Attribute (line 150),
- Rename Attribute (line 151),
- Change Primitive Data Type Attribute (line 152), and
- Change Custom Data Type Attribute (line 153).

Within database elements we can apply the following change operators:

- Rename Table (line 156),
- Add Index (line 157),
- Drop Index (line 158),
- Add Attributes to Index (line 159),
- Remove Attributes to Index (line 160),
- Rename Collection (line 161),
- Add Collection Index (line 162), and
- Drop Collection Index (line 163).

Finally we provide a construct to add Custom Data Type (line 166).

6.3.1.5 Graphical editor

The TyphonML graphical editor has been developed by means of Sirius[8]. Sirius is an Eclipse project that enables the development of graphical modelling environments by leveraging well-

[8] https://www.eclipse.org/sirius/

established technologies. Starting from a metamodel, it allows a model-based specification of visual concrete syntax organized in viewpoints(pointed with3), i.e., models that can be authored by means of different notations that suit the needs of various stakeholders. Figure 35 depicts an instance of TyphonML model defined by the graphical editor. The palette, in the right of Figure 35, allows the modeller to add the TyphonML elements to the canvas in the left part.

Section 6.2.1 describes how to open a TyphonML models by the graphical editor. Thanks to the Sirius and Xtext[9] integration, graphical and textual editors are both synchronized. In this way, stakeholders with different skills can define TyphonML models using different syntaxes.

Figure 35: Graphical editor

6.3.1.6 Generation of the OpenAPI specification

In this section, we describe how the OpenAPI[10] specification is generated from the TyphonML model. In particular, it is a specification for describing, consuming, and visualizing RESTful web

[9] https://www.eclipse.org/Xtext/
[10] https://www.openapis.org/

services which allows both humans and machines to discover and understand the provided services. An OpenAPI definition can be used for many purposes, e.g., documentation, generation of clients in various programming languages, displaying APIs as a web UI, testing, and many other use cases. Once the TyphonML specification is completed, a synthesis tool is applied to generate the corresponding OpenAPI specification by a set of coordinated Acceleo-based model-to-code transformations[11].

In The contextual menu (see

Figure 36) allows the modeler to produce the OpenAPI specification of a given TyphonML model. Then, she can use it to directly generate clients in various programming languages that programmatically interact with the Polystore resources. In Figure 37, we report an excerpt of the OpenAPI specification generated from a simple eCommerce TyphonML model.

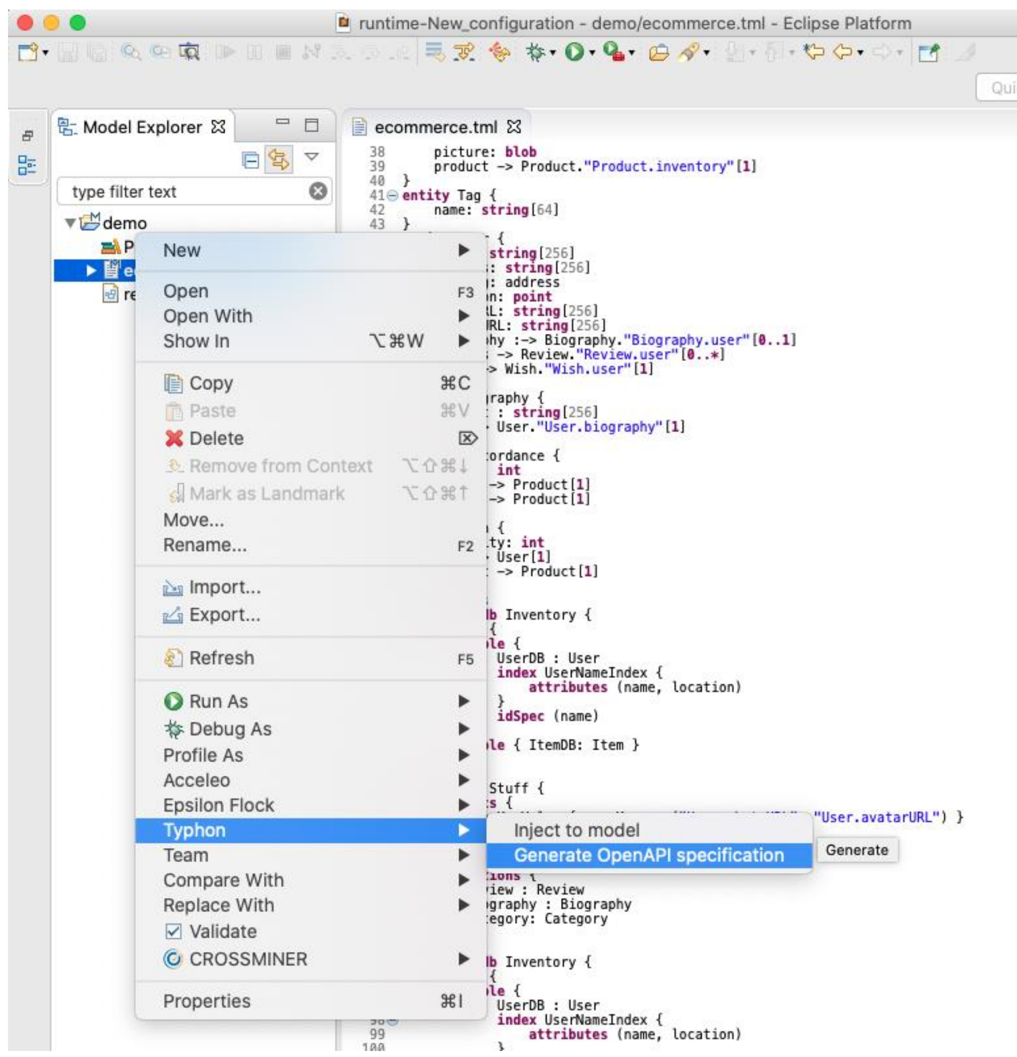


Figure 36: OpenAPI generation

[11] https://www.eclipse.org/acceleo/

Model Explorer

type filter text

- demo
  - Project Dependencies
  - ecommerce
    - gen-src
      - api-docs.json
    - ecommerce.tml
    - representations.aird

ecommerce.tml | api-docs.json

```json
{
    "swagger": "2.0",
    "info": {
        "description": "Api Documentation",
        "version": "1.0",
        "title": "Api Documentation",
        "termsOfService": "urn:tos",
        "contact": { },
        "license": {
            "name": "Apache 2.0",
            "url": "http://www.apache.org/licenses/LICENSE-2.0"
        }
    },
    "basePath": "/",
    "tags":[
        {
            "name": "Review-rest-controller",
            "description": "Review Rest Controller"
        },  {
            "name": "Product-rest-controller",
            "description": "Product Rest Controller"
        },  {
            "name": "Category-rest-controller",
            "description": "Category Rest Controller"
        },  {
            "name": "Item-rest-controller",
            "description": "Item Rest Controller"
        },  {
            "name": "Tag-rest-controller",
            "description": "Tag Rest Controller"
        },  {
            "name": "User-rest-controller",
            "description": "User Rest Controller"
        },  {
            "name": "Biography-rest-controller",
            "description": "Biography Rest Controller"
        },  {
            "name": "Concordance-rest-controller",
            "description": "Concordance Rest Controller"
        },  {
            "name": "Wish-rest-controller",
            "description": "Wish Rest Controller"
        }
    ],
    "paths":{
        "/Review": {
            "post": {
                "tags": [
                    "Review-rest-controller"
                ],
                "summary": "storeReview",
                "operationId": "storeReviewUsingPOST",
                "consumes": [
                    "application/json"
                ],
                "produces": [
                    "application/json"
                ],
                "parameters": [
                    {
                        "in": "body",
                        "name": "review",
                        "description": "review",
                        "required": true,
                        "schema": {
                            "$ref": "#/definitions/Review"
                        }
                    }
                ],
                "responses": {
                    "200": {
                        "description": "OK",
                        "schema": {
                            "type": "boolean"
                        }
                    },
                    "201": {
                        "description": "Created"
                    },
                    "401": {
                        "description": "Unauthorized"
                    },
                    "403": {
                        "description": "Forbidden"
```

Figure 37: An instance of generated OpenAPI specification

6.3.2 Typhon Deployment Language (DL)

In this section the usage of the TyphonDL modelling tools including script generation is presented. After creating a TyphonML model with the help of the TyphonML modelling tools a TyphonDL model can be created with the help of the TyphonDL Wizard (see 3.2.3) from the ML model. The wizard uses the previously defined (default or use-case specific) templates (see 6.3.2.2) and creates a TyphonDL model file and additional model files for every database that can be edited with the textual and/or graphical editor (see 6.3.2.4). When the DL model is ready, the TyphonDL Script

TL;DR
1. Create TyphonDL model (right click *on MLmodel.xmi -> Create TyphonDL Model*)
2. Create Deployment Scripts (right click on *DLmodel.tdl -> Generate Deployment Scripts*)
3. Run polystore

Generator can be used to generate technology dependent deployment scripts (see 6.3.2.5). Before the tools are explained, an overview over the Typhon Deployment Language is given in 6.3.2.1.

6.3.2.1 Typhon Deployment Language

This section introduces the metamodel that formalises the concepts that constitute the language primitives of TyphonDL. Meta-classes described below are presented using the font as in `font`:

`DeploymentModel` represents the root container of each TyphonDL specification and consists of two distinct elements:

- `MetaModel`: It represents the set of operators on TyphonDL models.
- `Model`: It represents the set of concepts that will be used in a TyphonDL model.

These elements are further defined as follows:

`MetaModel` consists of the import operation that allows a TyphonDL model to include the contents of another TyphonDL model.

`Model` consists of the following classes that categorise the components of a TyphonDL model:

- `Type`: It represents the collection of all types that are used in a TyphonDL model.
- `Services` represents the collection of deployable software services.
- `Platform` represents the logical units in a deployment environment.

`Type` consists of the following types:

- `PlatformType` represents the set of different types of platforms that can be used in a deployment task in the cloud. Example platform types are the Amazon Web Services cloud services platform, Microsoft Azure, Google Cloud etc.

- `ClusterType` represents the set of different types of schemes to govern over a cluster of containers.

- `ContainerType` represents the set of different types of containerisation software that can be used in a deployment task. Example container types are Docker, rkt, VirtualBox, VMWare, etc.

- `DBType` represents the collection of different database management systems such as MariaDB, MongoDB, Neo4j, Cassandra, etc.

`Services` distinguish between of database services `DB` and all other software services `Software`.

- Database services `DB` are named elements typed by a database type defined by `DBType`.
- `Software` is a named element and consists of a list of configuration parameters including image, URI, environment and properties.

`Platform` is a named element and typed by a platform type defined by `PlatformType`. It permits to model an individual platform space on a specific platform provider. It consists of a list of cluster declarations.

`Cluster` is a named element and typed by a cluster type defined by *ClusterType*. It consists of a list of application declarations.

`Application` is a named element that represents a software-based application that is possibly composed of several smaller software components that are deployed in individual containers.

`Container` is a named element that is typed by a container type defined by `ContainerType`. It represents a container or a virtual machine and consists of a list of configuration elements that are part of the TyhonDL metamodel or other container specific properties that are defined by `Property`.

Configuration elements consist of a set of pre-selected standard deployment configuration parameters. These parameters are the following ones.

- `Image:` The image that contains a set of instructions for creating a container.
- `HelmList:` The list of specifications to use Helm charts[12] to define the setup configuration of database deployments. In particular, the name of the Helm chart, the repository name and the repository address of the respective Helm chart are specified.
- `Environment:` The environment parameters used in the setup configuration of database deployments.
- `Credentials:` The credentials to be defined in the setup configuration of database deployments.
- `URI:` The URI for a database or a container through which they are accessed by Typhon
- `deploys:` The link between a service specification and the respective container it is deployed in.
- `depends_on:` The dependency relation between two containers.
- `Networks:` The network parameters to which a container is part of are specified.
- `Ports:` The parameters that publish a container to be reachable outside of a Polystore network are specified. These parameters are typically a target port for

[12] www.helm.sh/docs/topics/charts/

the container, and a published port that makes the container available outside of the Polystore.

- `Resources`: The parameters that control or limit the resources allocated to a container, such as CPU and memory.
- `Replication`: The parameters to define replicated instances of a container based on a specific replication mode including multi-primary, replica set with a primary/source and *n* replicas, and stateless replication.
- `Volumes`: The mount parameters for the directories in a container to save data or share data between containers. The parameters are the volume name, the mount path, the volume type and any other technology specific parameters for a volume.

`Property` is a set of three different kinds of configuration declarations in the form of:

- Key-value pairs (`Key_Values`),
- key and array of values (`Key_ValueArray`),
- list of key-value pairs (`Key_KeyValueList`)

and permit to represent any other configuration properties that are specific to individual containerisation technologies.

6.3.2.2 TyphonDL Templates

The TyphonDL plugin comes with a set of default `DB` and `DBType` templates, that can be viewed, imported, exported and edited in *Eclipse → Window → Preferences → TyphonDL → Templates* (see Figure 38). Here, additional templates can be added, or company specific `DB` settings can be defined and used for creating a new Polystore deployment.

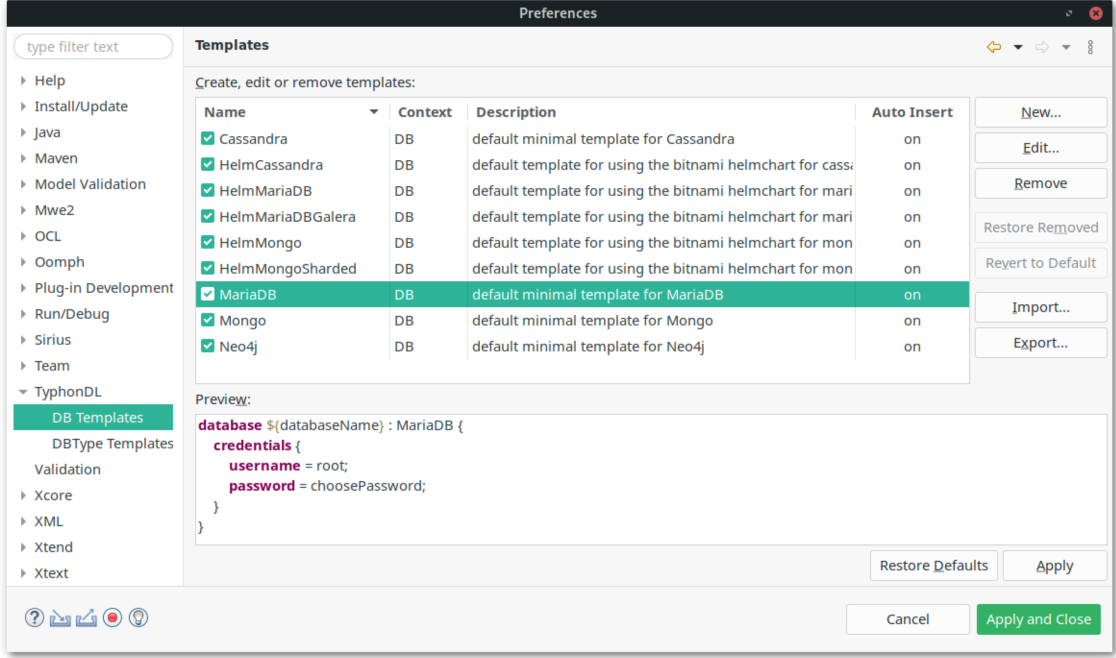


Figure 38: TyphonDL DB Template preferences

The default DB Templates include:

- MariaDB with `DBType` MariaDB[13] containing `Credentials` with *username = root* and a *password* to be set by the user.
- Mongo with `DBType` Mongo[14] containing `Credentials` with *username* and *password* to be set by the user.
- Cassandra with `DBType` Cassandra[15] containing an `Environment` to set the *maximum heap size* and the *amount of heap memory allocated to newer objects*[16].
- Neo4j with `DBType` Neo4j[17] containing `Credentials` with *username = neo4j* and a *password* to be set by the user.
- HelmMariaDB with `DBType` MariaDB containing a `HelmList` using bitnami/mariadb[18] and `Credentials` with *username = root* and a *password* to be set by the user.
- HelmMariaDBGalera with DBType mariadbgalera containing a `HelmList` using bitnami/mariadb-galera[19] and `Credentials` with *username = root* and a *password* to be set by the user.
- HelmMongo with `DBType` Mongo containing a `HelmList` using bitnami/mongodb[20] and `Credentials` with *username=root* and a *password* to be set by the user.
- HelmMongoSharded with `DBType` mongoshareded containing a `HelmList` using bitnami/mongodb-sharded[21] and `Credentials` with *username=root* and a *password* to be set by the user.
- HelmCassandra containing a `HelmList` using bitnami/cassandra[22] and `Credentials` with *username* and *password* to be set by the user.
- HelmNeo4j containing a `HelmList` using neo4j-helm[23] and `Credentials` with *username=neo4j* and a *password* to be set by the user.

[13] https://hub.docker.com/_/mariadb
[14] https://hub.docker.com/_/mongo
[15] https://hub.docker.com/_/cassandra
[16] https://docs.datastax.com/en/ddac/doc/datastax_enterprise/operations/opsConHeapSize.html
[17] https://hub.docker.com/_/neo4j
[18] https://github.com/bitnami/charts/tree/master/bitnami/mariadb
[19] https://hub.helm.sh/charts/bitnami/mariadb-galera
[20] https://github.com/bitnami/charts/tree/master/bitnami/mongodb
[21] https://hub.helm.sh/charts/bitnami/mongodb-sharded
[22] https://hub.helm.sh/charts/bitnami/cassandra

6.3.2.3 *TyphonDL Wizard*

To create a TyphonDL model from a TyphonML model the TyphonDL Wizard has to be started by selecting the given ML model and selecting *Create TyphonDL model* in the Typhon context menu (see Figure 39: TyphonDL Creation Wizard).

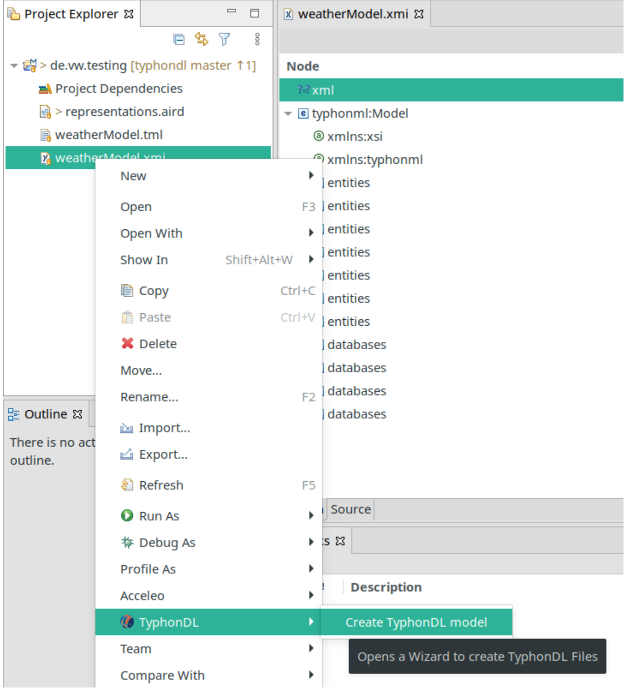


Figure 39: TyphonDL Creation Wizard

On the first page of the wizard (see Figure 40) the name for the TyphonDL model has to be entered and a deployment technology such as Docker Compose, or Kubernetes has to be chosen from a dropdown menu. The selected technology will be included in the model in the form of `Clustertype` which is used when defining a `Cluster`:

```
clustertype DockerCompose

cluster clusterName: DockerCompose …
```

The Analytics component (see 2.1.8) can be activated and deployment scripts can be created to either run it alongside the other Polystore components, or to run it on a different machine. If run alongside the other Polystore components, the Typhon Continuous Evolution component can also be activated. An already running Analytics component can also be added to the model by giving its URI. The URI for reaching the Analytics container can only be defined here. If it were to change at a later point, the DL model would have to be recreated with the Creation Wizard. For the UI to be reachable by the API, the API URI (consisting of host and port) has to be given to the Wizard. If Swarm Mode or Kubernetes is used, it is possible to scale the stateless parts of the Polystore, i.e. the API and the QL server.

---

[23] https://github.com/neo4j-contrib/neo4j-helm

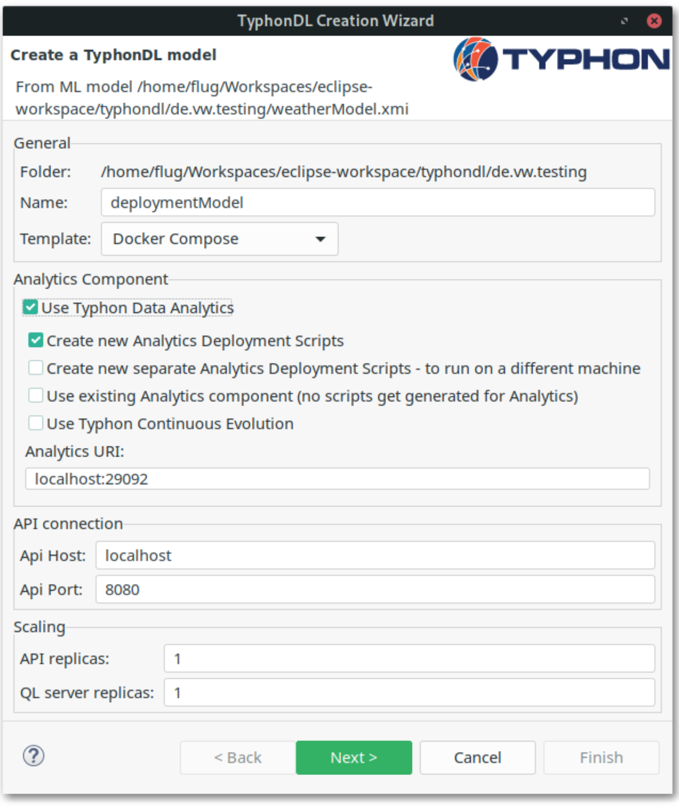


Figure 40: TyphonDL Creation Wizard: Page one

If the Analytics component is to be generated, an optional page (see Figure 41) appears after the first one. Here, the Analytics component can be configured.

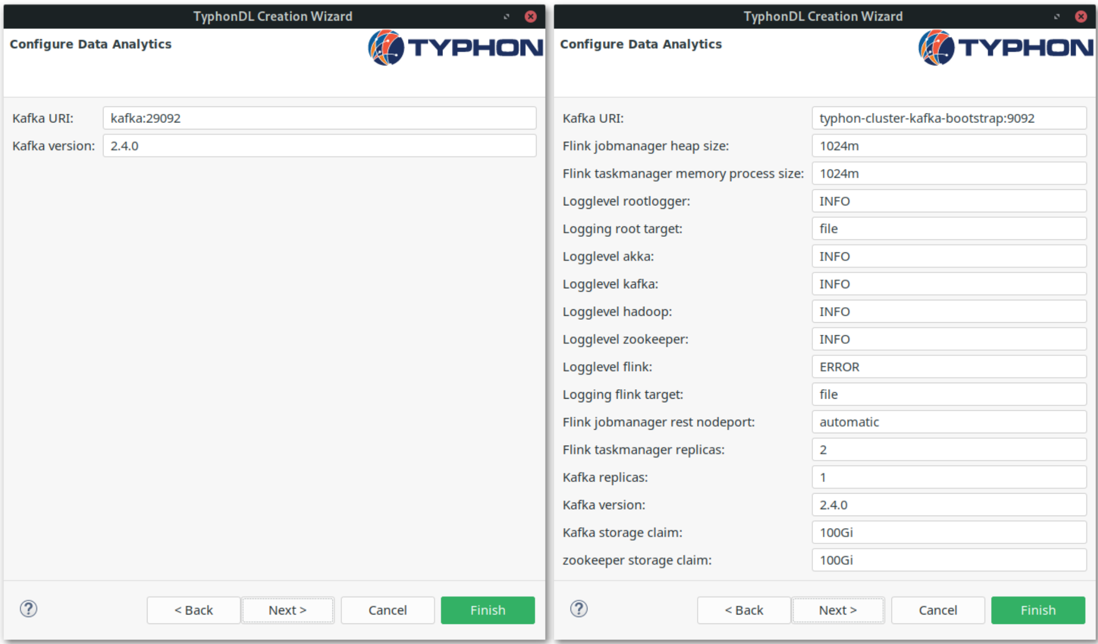


Figure 41: TyphonDL Creation Wizard: Configuring the Analytics component Docker Compose vs. Kubernetes)

TyphonML provides an XMI representation of the ML model that is parsed by the TyphonDL Wizard and that filters out the databases to be deployed by TyphonDL. For each database the second page of the wizard (see Figure 42) provides the possibility to choose one of the following options:

1. Use a pre-existing `DB` model file[24] if a file with the name *<databasename>.tdl* exists in the project folder.
2. Create a new `DB` model object by choosing a template (shown in 6.3.2.2) from the drop down menu.
3. Use an existing externally running database. A `DB` model object with the flag `external`, an `URI` and the `DBType` of the selected template is created.
4. If Kubernetes is chosen on the first page, the option to use a Helm Chart[25] is added. Here, one of the templates already containing a `HelmList` should be chosen, their names all start with "Helm". Otherwise a new default `HelmList` using bitnami[26] as *Helm Repo* is created.

In each of the above cases, the resulting `DB` model object is cached in the Creation Wizard for further configuration on the next pages.

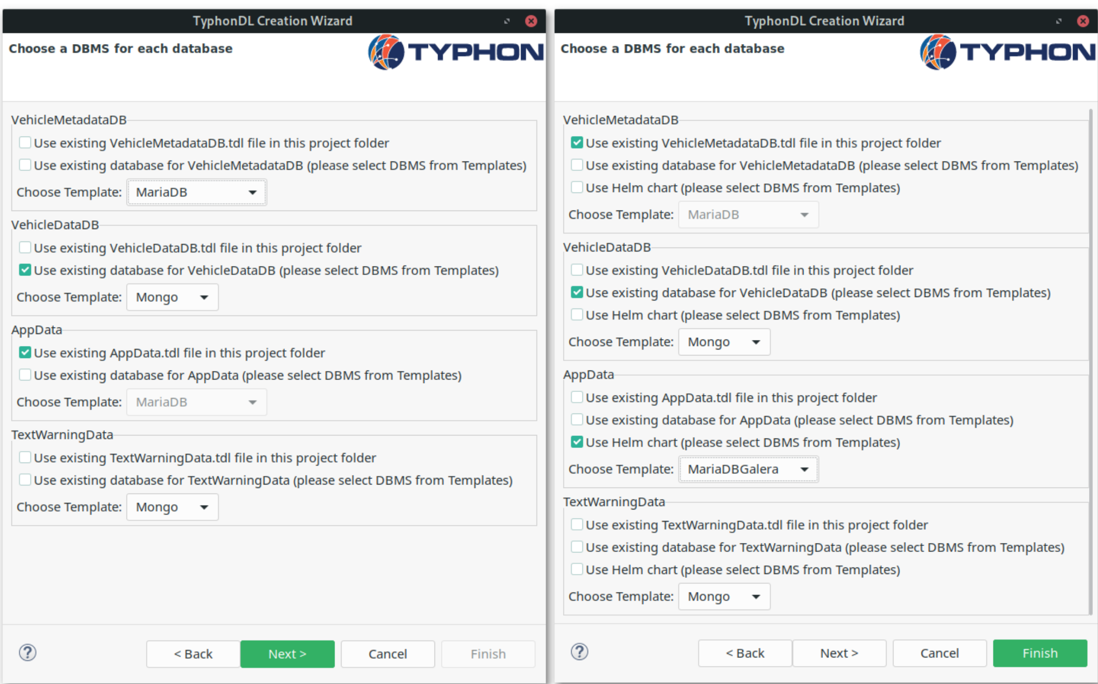


Figure 42: TyphonDL Creation Wizard: Choosing the DBMS for each database (Docker Compose vs. Kubernetes)

---

[24] (examples in Listing 3, Listing 4 and Listing 5)
[25] https://hub.helm.sh/
[26] https://bitnami.com/stacks/helm

In Figure 43 two example `DB` configurations are shown. If the `DB` is not set to external, a `Container` model object for each database is created and cached together with the `DB` object in the Wizard. The `Container` gets an `URI` object with the value *<containerName>:<containerPort>*. This `URI` is parsed by the API to know where to reach each database.

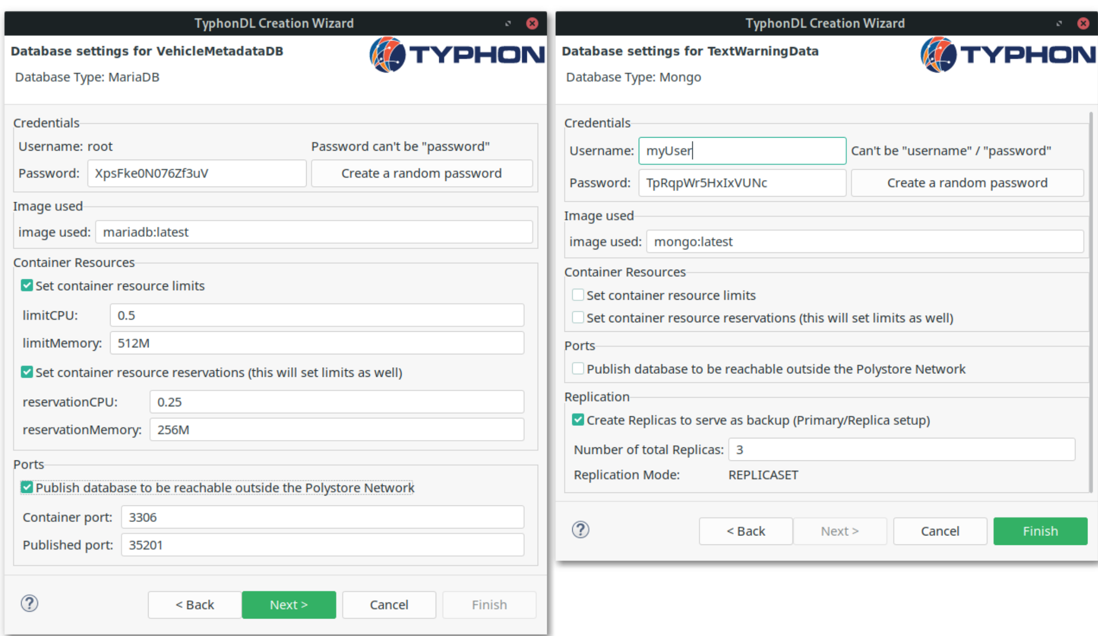


Figure 43: TyphonDL Creation Wizard: Further database configuration (MariaDB container vs. MongoDB container)

On the left side of Figure 43 database settings for VehicleMetadataDB are presented. The previously chosen `DBType` is shown on the top of the page – here MariaDB (compare to Figure 42). The template (see Figure 38) has a given username (*root*) and only allows to choose the password. The Wizard provides the possibility to generate a 16 digit password containing small and capital letters and numbers. If a different image version should be used, it can be defined in the "Image used" group. Next, container resources can be defined by checking the respective checkboxes. This will add a `Resources` object to the `Container`. CPU is measured in CPU units, given as the fragment of available processing time (0.2 = 20%). Memory is measured in bytes and is expressed as integer using one of these suffixes: T, G, M, K. It's possible - though not recommended in production - to publish a database container with a given "Published Port" in the "Ports" group. This will add a `Ports` object to the `Container`.

On the left side of Figure 43, the MongoDB TextWarningData can be configured. Here, both username and password can be chosen. Additionally, to the options above, it's allowed to replicate the MongoDB[27] if Docker Compose is used. If the Primary/Secondary option is chosen, a `Replication` object is added to the `Container`. The number of total Replicas denotes the number of additionally created containers.

On the left side of Figure 44, the database settings for VehicleDataDB, an `external` MongoDB (compare with the checkbox in Figure 42:left) are presented. Additionally, to setting the `Credentials`, the user has to give an `URI` pointing to the database in the "Database Address" group.

An example for using Helm charts in the `DB` AppData (compare with the checkbox in Figure 42:right) is given on the right side of Figure 44. The template for MariaDB Galera (see 6.3.2.2) already contains the repository settings. The user can specify the use of a custom *values* file. If the valuesFile field contains the repository name (here "bitnami"), the default values provided by the chart are taken[28].

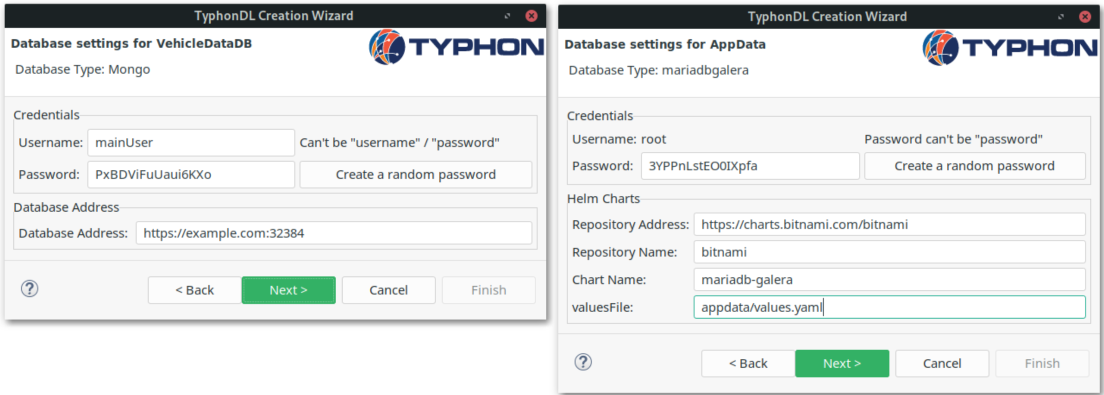


Figure 44: TyphonDL Creation Wizard: Further database configuration (MongoDB external database vs. MariaDB Galera Cluster)

When the wizard is finished, the following TyphonDL files get added to the project:

- TyphonDL model file with the name that was given in the wizard (examples in Listing 1 using Docker Compose and Listing 2 using Kubernetes).
- Properties file needed to generate deployment scripts.
- One model file for each database (examples in Listing 3, Listing 4 and Listing 5).

[27] https://docs.mongodb.com/manual/replication/
[28] E.g. https://github.com/bitnami/charts/blob/master/bitnami/mariadb-galera/values.yaml

- One model file containing the DBTypes (example in Listing 6).

```
import weatherModel.xmi
import VehicleMetadataDB.tdl
import AppData.tdl
import TextWarningData.tdl
import VehicleDataDB.tdl
import dbTypes.tdl
containertype Docker
clustertype DockerCompose
platformtype localhost
platform platformName : localhost {
	cluster clusterName : DockerCompose {
		application Polystore {
			container vehiclemetadatadb : Docker {
				deploys VehicleMetadataDB
				ports {
					target = 3306 ;
					published = 35201 ;
				}
				resources {
					limitCPU = 0.5 ;
					limitMemory = 512M ;
					reservationCPU = 0.25 ;
					reservationMemory = 256M ;
				}
				uri = vehiclemetadatadb:3306 ;
			}
			container appdata : Docker {
				deploys AppData
				uri = appdata:3306 ;
			}
			container textwarningdata : Docker {
				deploys TextWarningData
				uri = textwarningdata:27017 ;
				replication {
					replicas = 3 ;
					mode = replicaSet ;
				}
			}
		}
	}
}
```

Listing 1: Main model file deploymentModel.tdl generated by the TyphonDL Creation Wizard using Docker Compose

```
import weatherModel.xmi
import AppData.tdl
import TextWarningData.tdl
import VehicleMetadataDB.tdl
import VehicleDataDB.tdl
import dbTypes.tdl
containertype Docker
clustertype Kubernetes
platformtype minikube
platform platformName : minikube {
	cluster clusterName : Kubernetes {
		application Polystore {
			container appdata : Docker {
				deploys AppData
				uri = appdata:3306 ;
			}
			container textwarningdata : Docker {
				deploys TextWarningData
				uri = textwarningdata:27017 ;
			}
			container vehiclemetadatadb : Docker {
				deploys VehicleMetadataDB
				ports {
					target = 3306 ;
					published = 3306 ;
				}
				resources {
					limitCPU = 0.5 ;
					limitMemory = 512M ;
					reservationCPU = 0.25 ;
					reservationMemory = 256M ;
				}
				uri = vehiclemetadatadb:3306 ;
			}
		}
	}
}
```

Listing 2: Main model file deploymentModel.tdl generated by the TyphonDL Creation Wizard using Kubernetes

```
database AppData : MariaDB {
	credentials {
		username = root ;
		password = zRcUgpmgcBmZuSSI ;
	}
}
```

Listing 3: AppData.tdl containing the password created in the Wizard

```
external database VehicleDataDB : Mongo {
	uri = https://example.com:32384 ;
	credentials {
		username = mainUser ;
		password = yG7w4djhIg1F2ZI3 ;
	}
}
```

Listing 4: VehicleDataDB.tdl is an external database which is not deployed by a container in the main model file

```
database AppData : mariadbgalera {
	helm {
		repoName = bitnami ;
		repoAddress = https://charts.bitnami.com/bitnami ;
		chartName = mariadb-galera ;
		valuesFile = appdata/values.yaml ;
	}
	credentials {
		username = root ;
		password = ell8qy43MvnwxFEa ;
	}
}
```

Listing 5: AppData.tdl when using a Helm Chart and giving a custom values file

```
dbtype MariaDB {
	default image = mariadb:latest;
}
dbtype Mongo {
	default image = mongo:latest;
}
dbtype mariadbgalera {
	default image = bitnami/mariadb-galera;
}
```

Listing 6: dbtypes.tdl

6.3.2.4 TyphonDL Editor

Xtext provides a textual editor with syntax highlighting, auto completion and an outline view. If the project that includes the models does not hold an Xtext nature, the TyphonDL Creation Wizard automatically adds it to the project. Linking between files (shown in Figure 45) is provided by Xtext.

The TyphonDL Creation Wizard already creates a valid TyphonDL model, comprehensive enough to generate Polystore deployment scripts, but the user can still add additional information. When Kubernetes is chosen, the `Platformtype` is automatically set to "minikube[29]", a testing environment. A different Platform Type can easily be used by changing the value of `Platformtype` and adding a "kubeconfig" Key_`Values` to the Cluster. The "kubeconfig" file can be downloaded from the cluster provider. An example for using AWS is shown in Listing 7.

```
platformtype AWS
platform platformName : AWS {
	cluster clusterName : Kubernetes {
	    kubeconfig = /path/to/downloaded/kubeconfig.yaml;
```

Listing 7: Changing the Platformtype and providing a kubeconfig file

[29] https://kubernetes.io/docs/setup/learning-environment/minikube/

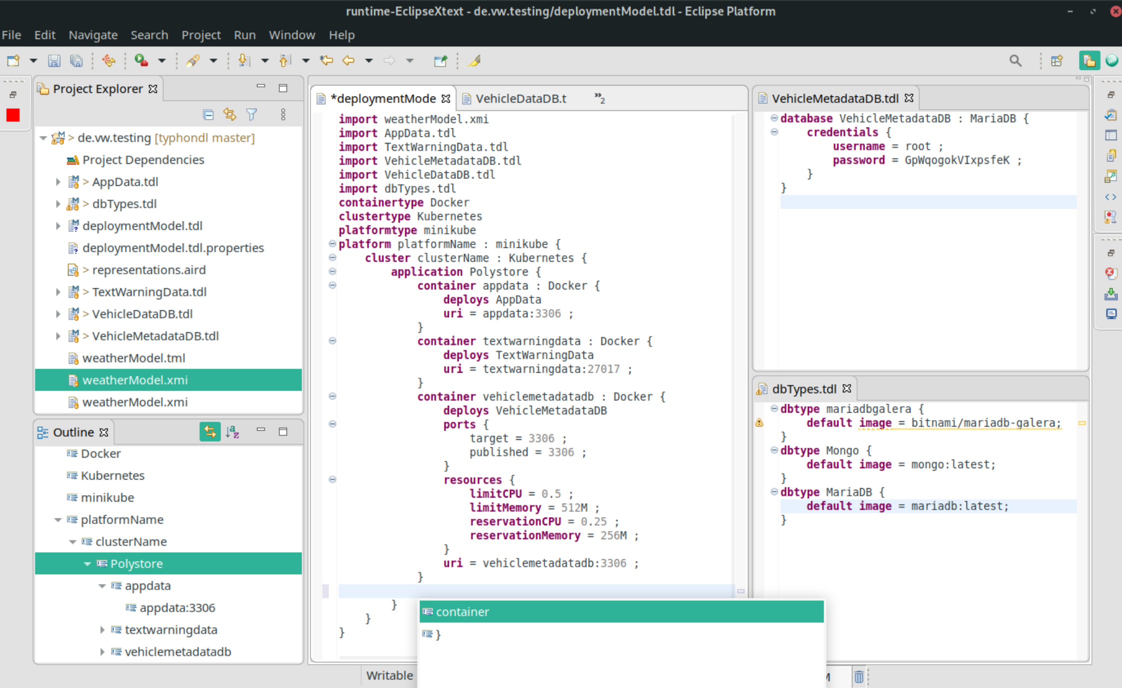


Figure 45: TyphonDL textual editor with syntax highlighting and auto completion

Using the text editor it is possible make further modifications in a generated TyphonDL model or to add additional specifications. Features such as volumes or network configurations can be added as shown in **Listing 8**.

```
platform platformName : localhost {
      cluster clusterName : DockerCompose {
            networks weatherwarningapp
            application Polystore {
                  container vehicledatadb : Docker {
                        deploys VehicleDataDB
                        networks weatherwarningapp
                        uri = vehicledatadb:27017;
                        volumes {
                              volumeName = vehicledata;
                              mountPath = /vehicledata;
                              volumeType = volume ;
                              volume {
                                    nocopy = true;
                              }
                        }
                  }
            volumes {
                  vehicledata
            }
            }
      }
}
```

Listing 9: Adding volumes and networks in for a Docker-Compose

6.3.2.5 TyphonDL Script Generation and running the Polystore

To create deployment scripts the TyphonDL Script Generator has to be started by selecting the created and completed DL model (main model file) and choosing *Generate Deployment Scripts* in the TyphonDL context menu (see Figure 46). .

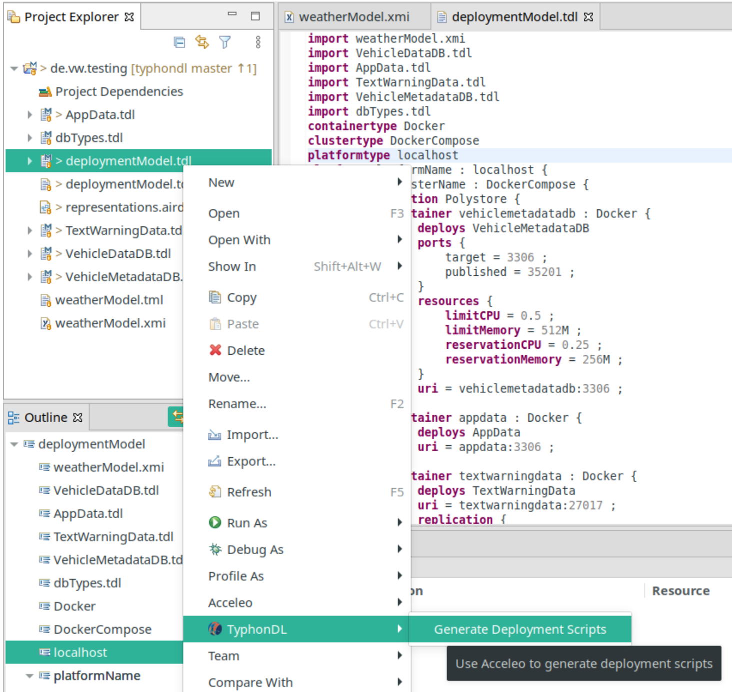


Figure 46: Generate Deployment Scripts

A folder with the name of the DL model is generated. It contains all files necessary to run the Polystore deployment.

1. If Docker Compose was chosen, a Service is created for every database and the Polystore can be started by running:

```
$ docker-compose up -d
```

Show all running containers:

```
$ docker-compose ps
```

Show logs of a specific service (e.g. the API):

```
$ docker-compose logs typhon-polystore-service
```

Stop and remove the Polystore (including volumes):

```
$ docker-compose down -v
```

2. If the DL model contains `Resources` or the replication of stateless Polystore parts (i.e. API, QL server and Analytics.Kafka), the Polystore has to be started by running

```
$ docker stack deploy --compose-file docker-compose.yaml typhon
```

with Docker running in Swarm Mode. Otherwise, the resource definition is ignored. The user can also setup Docker in Swarm Mode using multiple worker nodes and deploy the Polystore as a stack30.

Show all running containers:

```
$ docker stack services typhon
```

Stop and delete all Polystore containers:

```
$ docker stack rm typhon
```

3. If Kubernetes was chosen, a Deployment and a Service to connect to the Pod(s) created by the Deployment is created for every database and the Polystore can be started by executing:

```
$ sh deploy.sh
```

Stop and delete the Polystore deployment:

```
$ kubectl delete namespaces typhon
```

If the analytics component was started and also should be stopped and removed:

```
$ kubectl delete namespaces kafka
```

6.3.3 Typhon Query Language (QL) Eclipse Plugin

6.3.3.1 Initialize TyphonQL

The TyphonQL IDE can be used to develop new QL queries and inspect the Polystore. To start out, we need to create a new project. Open the Eclipse IDE and go to Create a project. (or go to File → New project)

[30] https://docs.docker.com/engine/swarm/stack-deploy/

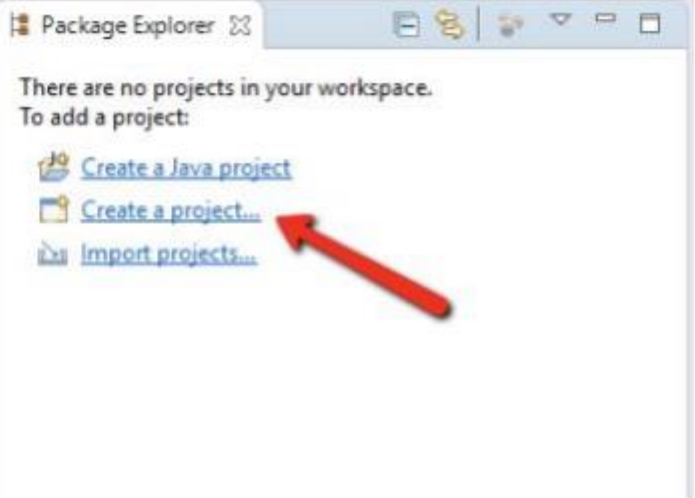


Figure 47: Creating a new Project

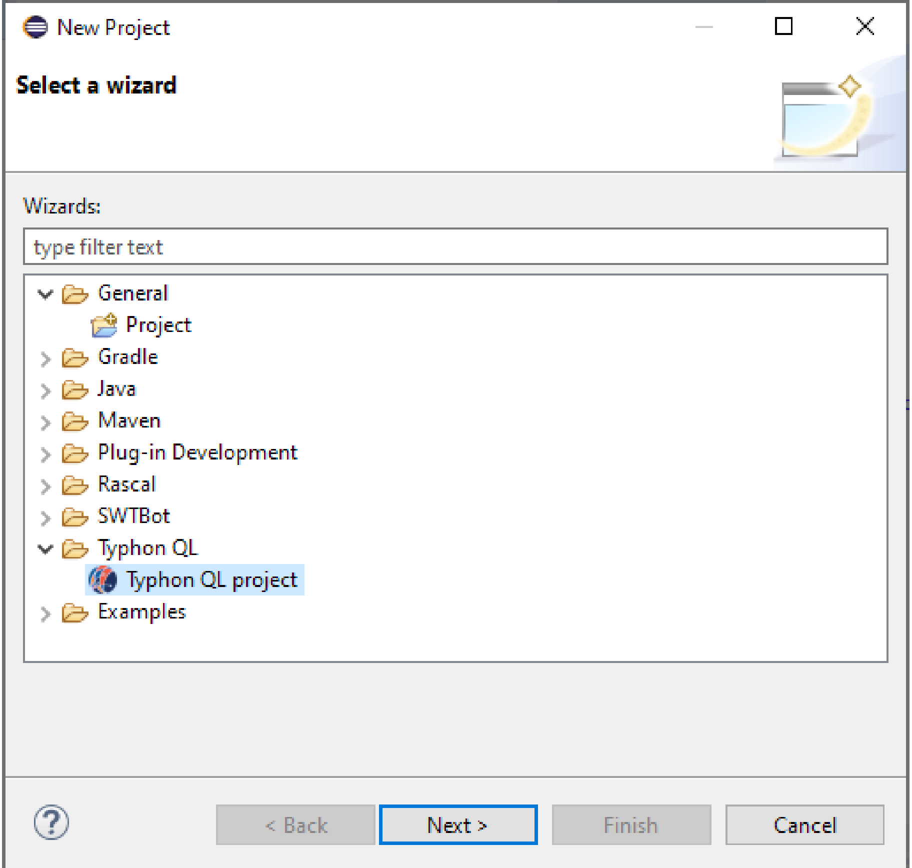


Figure 48: New QL project

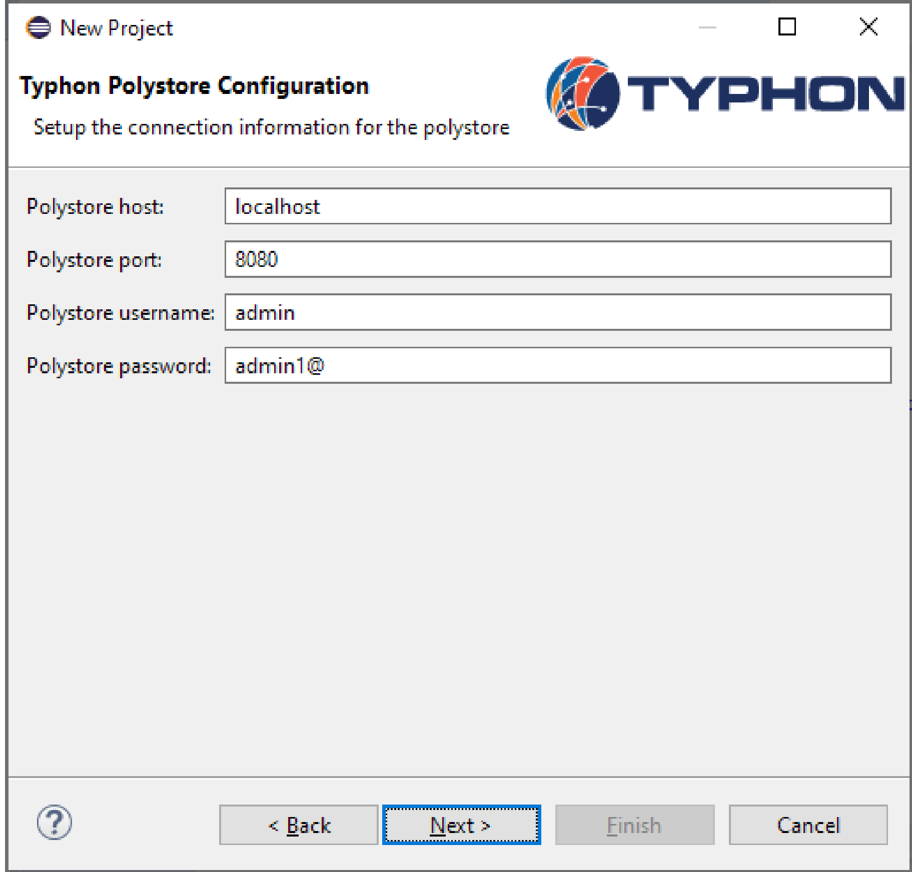
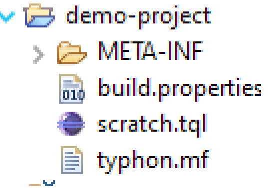


Figure 49: QL Setup

Select Typhon QL project under TyphonQL wizard category (see Figure 48) and configure the project with connection details of the Polystore (see Figure 49: QL Setup).
This is the structure of the newly created TyphonQL project:



It contains a scratch file where you can try queries, and a typhon.mf file that contains the connection details supplied in the wizard. We assume the Polystore is running and both a Typhon ML and DL model had been uploaded. Only given such circumstances we can execute queries.

The following image shows the scratch tql file. You can right-click inside the text region of the editor, select the TyphonQL menu, and execute some useful actions on the Polystore. If the TyphonQL menu is not shown, it could be that the environment needs more time to setup the TyphonQL language. If that is the case, close the file and reopen it a minute later.

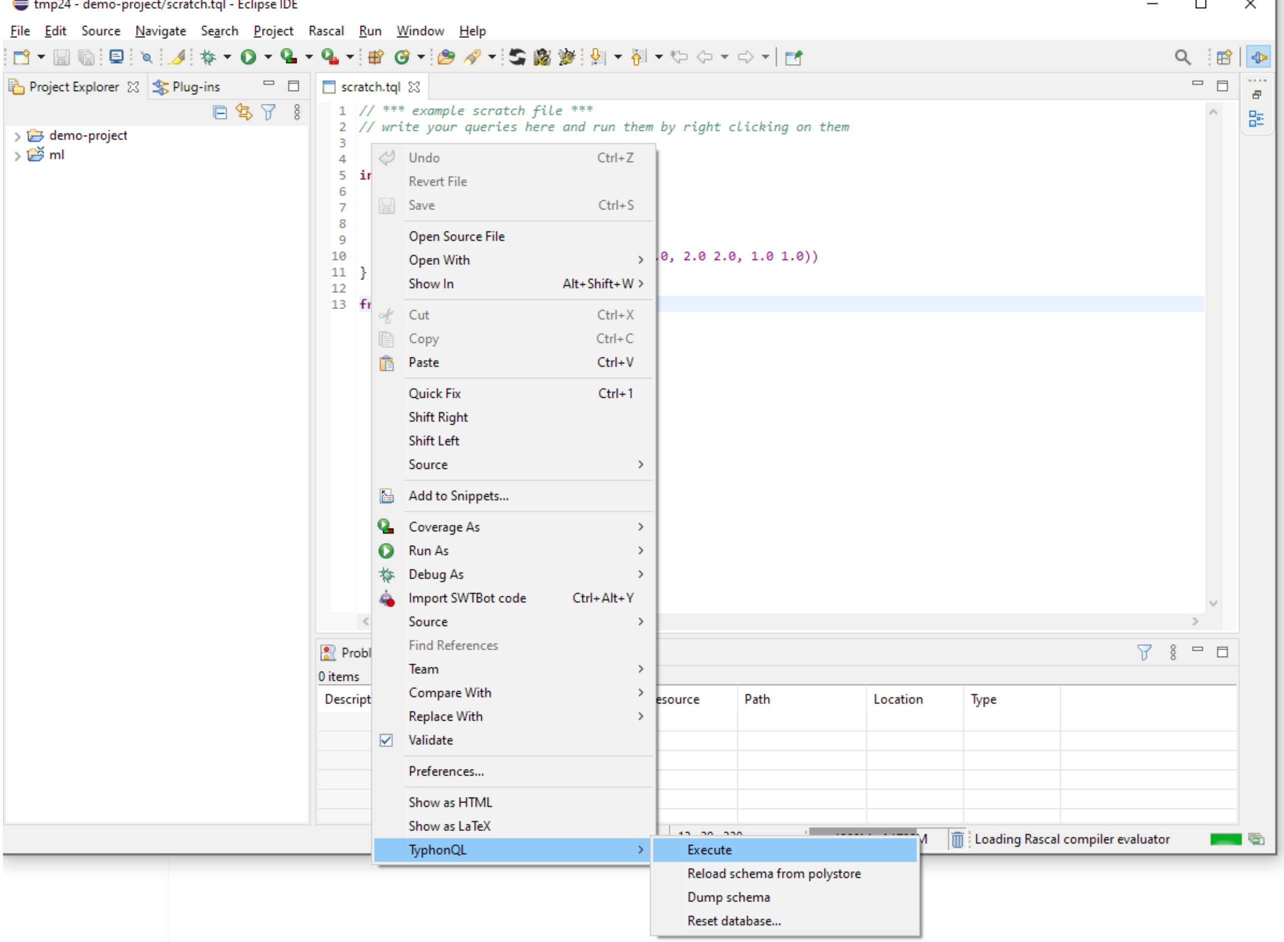


Figure 50: Typhon QL options

As you can see, there are 4 possible actions available on the TyphonQL menu. From them, the last 3 are global operations on the Polystore, that act independently of whether there are queries written on the tql file.

6.3.3.2 TyphonQL Global Operations

Reload schema from Polystore

This operation updates the schema that the IDE uses, using the properties specified in the typhon.mf file to query the polystore again in order to get the schema. This action is needed if the Polystore properties are changed in the configuration file or if new versions of the DL/ML models are updated in the poly store.

Dump Schema

This operation allows us to see a simple representation of the schema that is being used by the IDE.

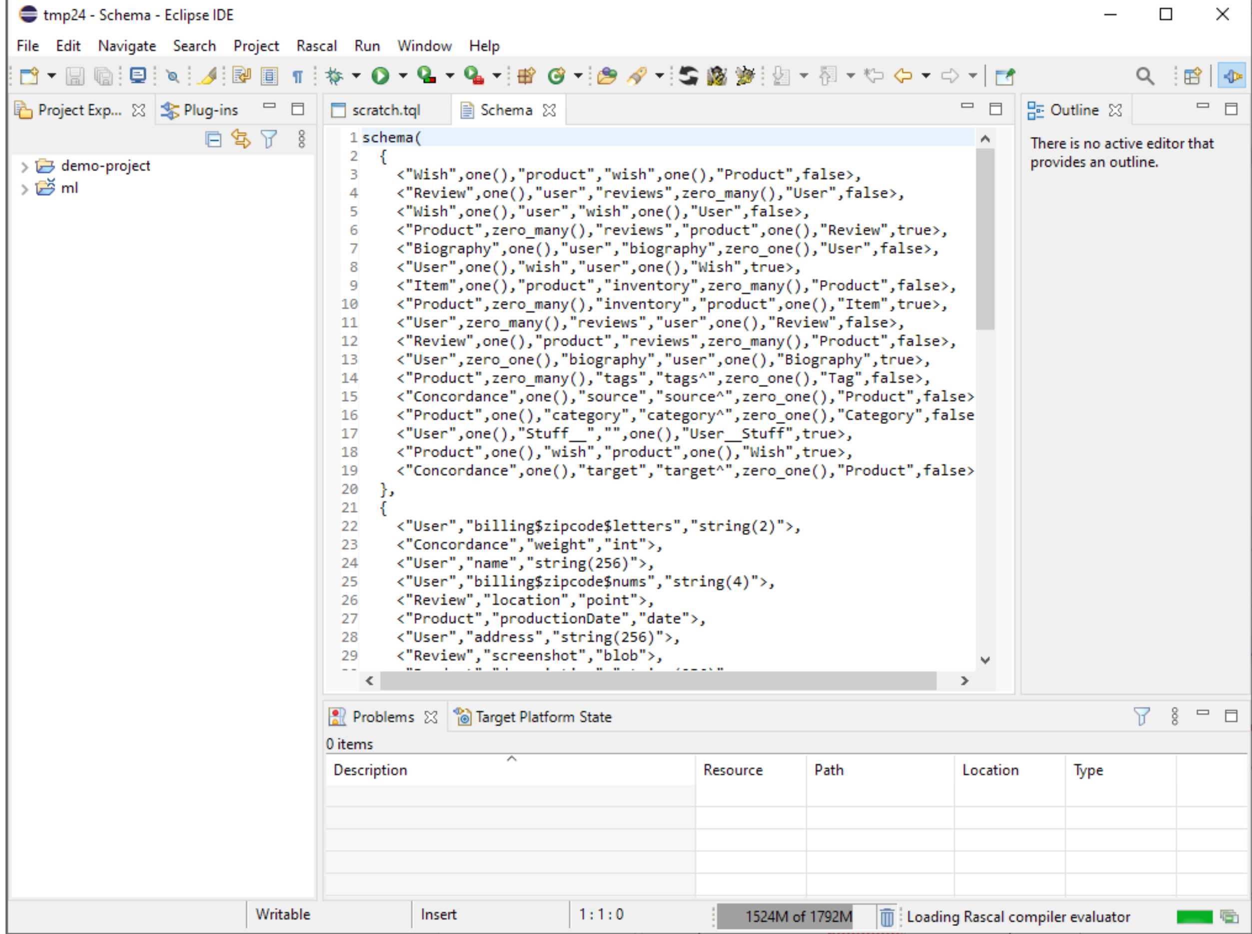


Figure 51: Dump Schema results

Reset Database

This action is needed in two cases: if the Polystore databases have not yet been created, and if they exist, but the user wants to reset them. Notice that this step is fundamental before any querying takes place. If there is a running Polystore in which both a Typhon ML and DL models have been uploaded, this means that the infrastructure has been set up (the different kinds of native databases). However, the logical databases in each database have not been created. To do so, we need to execute this action.

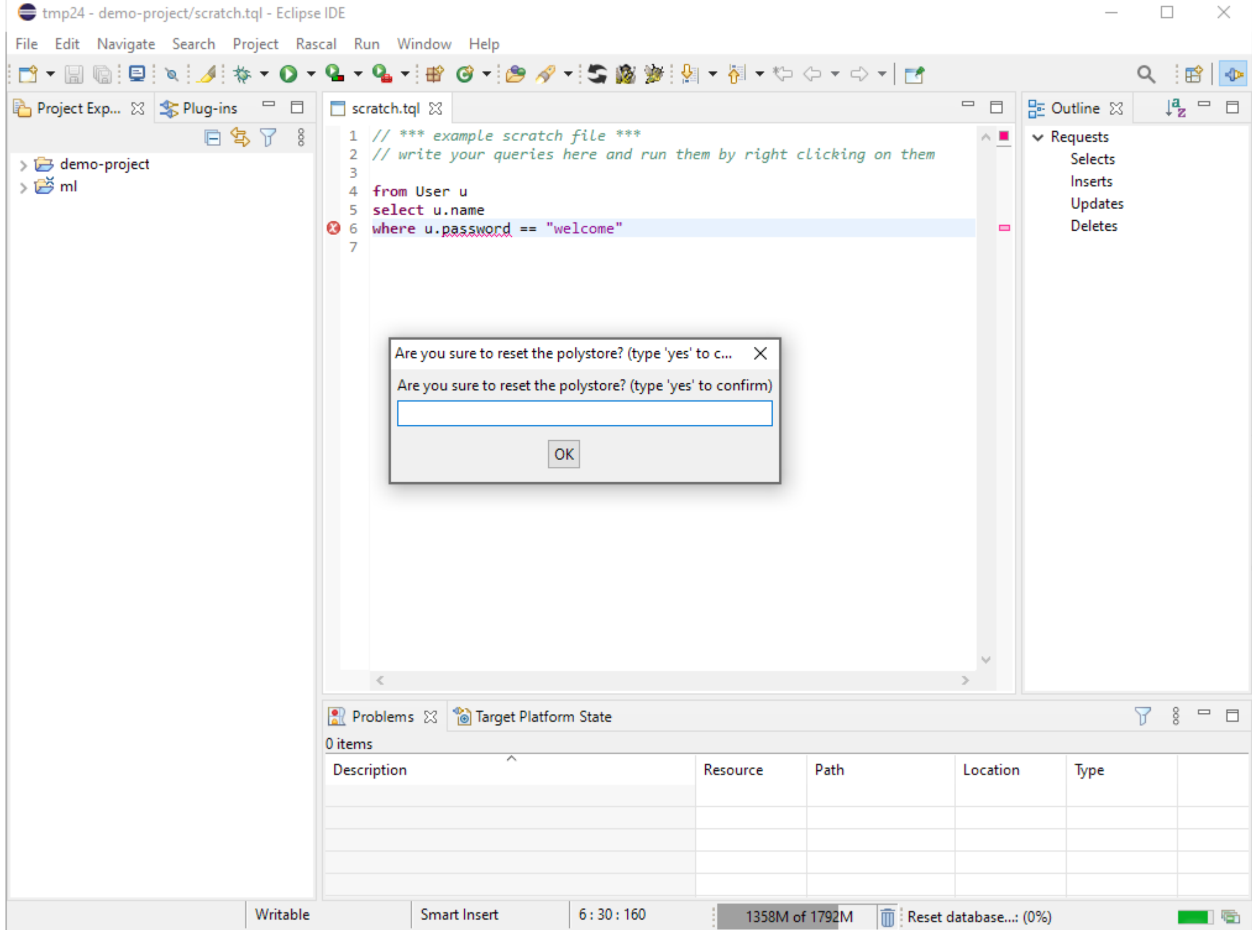


Figure 52: Reset Polystore prompt

Executing Queries

Once the databases have been set up (see 2.1.3), we can query the Polystore. For this User Guide we assume a Typhon model of Orders, Products, Users, Comments, Reviews, etc.
In this section we present a simple insertion and a selection query. For a reference on the syntax and semantics of the TyphonQL language, refer to Section 6.3.5.
To query the Polystore, we start writing queries in the scratch.tql file, for example:

```
insert Product {
       name: "Raspberry Pi",
       description: "Small arm board",
       price: 100,
       productionDate: $2020-01-01$,
       availabilityRegion: #polygon((1.0 1.0, 2.0 2.0, 1.0 1.0))
}
```

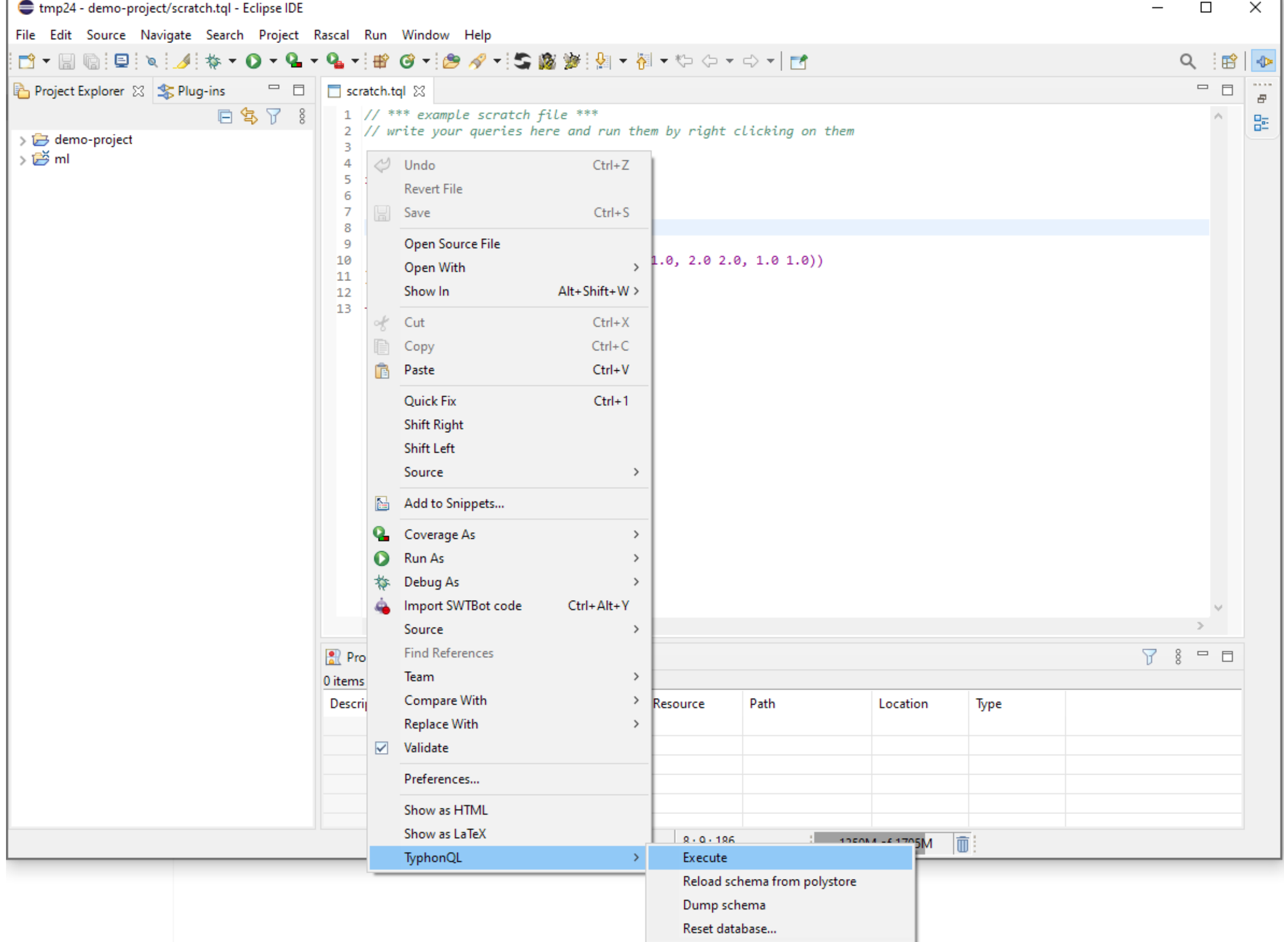


Figure 53: Insert query example

The previous query is an insertion operation. In order to execute it, we need to position the cursor inside the region of the query text, and right-click on it. Then, from the TyphonQL menu, select Execute.

We can also write a "selection" query to retrieve the orders, for instance:

```
from Product p select p.name
```

If we right-click on our new query and select TyphonQL→ Execute, this is the result, as expected:

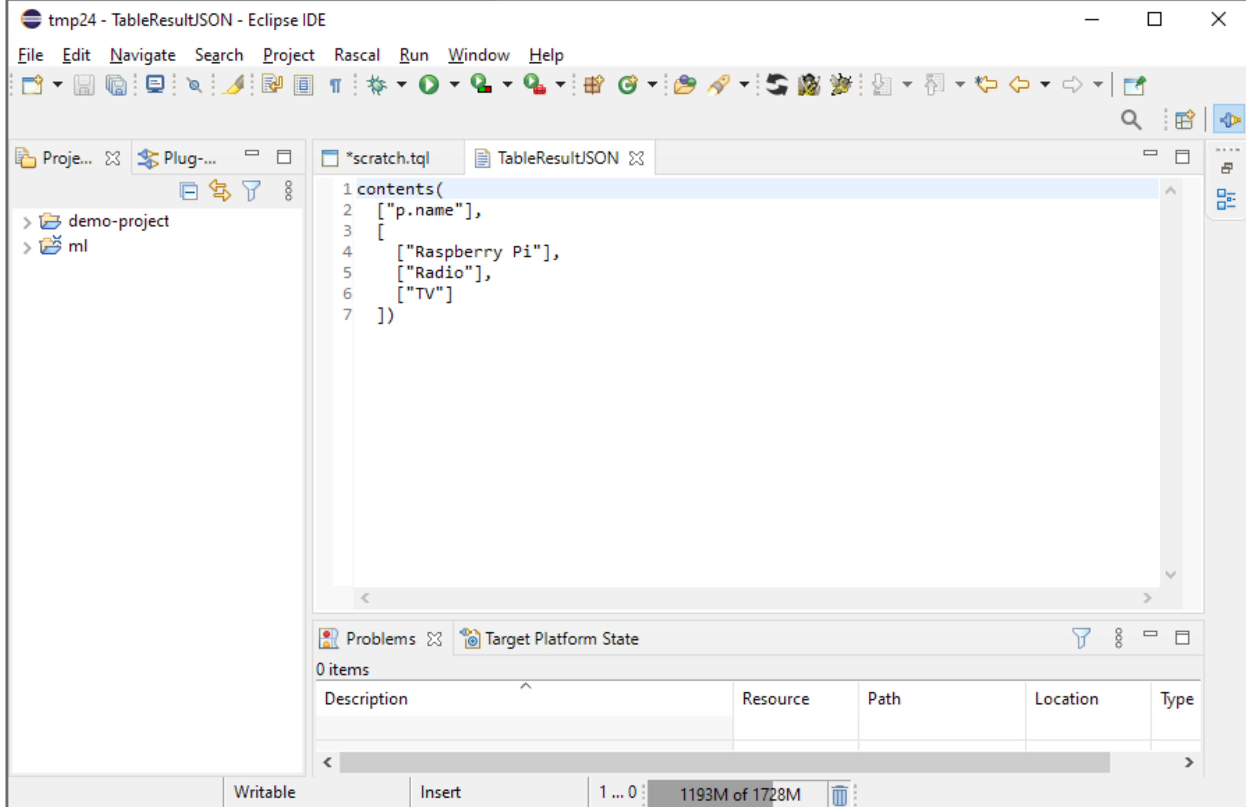


Figure 54: Results of select query

6.3.4 Polystore API

The next step for using the Polystore is to perform queries. These can be performed either through the API or directly through the Eclipse QL plugin. The first case, along with other API operations will be described in this section.
All endpoints use Basic authentication, and as such the relevant header will need to be included in all requests. Most of these services are also available on the Swagger page of the API. Additionally, an up-to-date Postman collection can be found at

6.3.4.1 User Services

The API provides user management services. This allows already registered users to create, modify, get or delete users. The relevant endpoints, example inputs & outputs can be found in the table below:

Description	**Endpoint**	**Body**	**Output**
Register a new user	POST /user/register	`{"u":{ "username":"username" , "password":"password" } }`	200 OK
Get all users	GET /users	N/A	`[ {`

			"username": "admin"
			,
			"pass-
			word": "admin1@"
			},
			{
			"username": "5f117d
			57b3ec643e55a4cec7",
			"password": null
			}
			]
			
			200 OK

Update a user	POST /user/{username}	{"u":{"username":"5f1
		17d57b3ec643e55a4cec7
		","password":"passwor
		d"}}
			{
			"username": ":"5f117d57
			b3ec643e55a4cec7",
			"password": "password"
			}
			200 OK

Delete a user	DELETE /user/{username}	N/A	200 OK

6.3.4.2 Backup/Restore Services

The API also offers backup and restore capabilities for some types of Polystore databases. These requests can be found in the table below:

Description	**Endpoint**	**Input**	**Output**

Backup a database	POST /api/backup	{
		"db_name":"maria",
		"type":"mariadb",
		"host":"maria",
		"port":"3306",
		"username":"root",
		"password":"choosePassword",
		"backup_name":"test_bkup"
		}
			{
			"filename":
			"test_bkupmaria_mari
			a_17072020.sql"
			}
			200 OK

Restore a database	POST /api/restore	{ "db_name":"maria", "type":"mariadb", "host":"maria", "port":"3306", "username":"root", "password":"choosePassword", "back- up_name":"test_bkupmaria_maria_1 7072020.sql"	200 OK

		}	
Download a backup	GET /api/download/{filename}	N/A	File download 200 Ok

6.3.4.3 *Service/Database information Services*

The result of the DL parsing, which occurs in the API and by consequence the connection information regarding databases and services that are used by other components and the API can be retrieved for debug/validation purposes. The requests that can be used for these retrievals are the following:

Description	**Endpoint**	**Input**	**Output**
Get Databases	GET /api/databases	N/A	[{ "name": "Reviews", "status": "ONLINE", "internalHost": "reviews", "externalHost": "reviews", "internalPort": 27017, "externalPort": 27017, "username": "chooseUsername", "password": "choosePassword", "dbType": "MongoDb", "serviceType": "Database", "engineType": "Document", "external": false }, { "name": "Inventory", "status": " ONLINE", "internalHost": "inventory", "externalHost": "inventory", "internalPort": 3306, "externalPort": 3306, "username": "root", "password": "choosePassword", "dbType": "MariaDb", "serviceType": "Database", "engineType": "Relational", "external": false }, { "name": "Stuff", "status": " ONLINE", "internalHost": "stuff", "externalHost": "stuff", "internalPort": 9042, "externalPort": 9042,

			"username": "admin",
			"password": "password",
			"dbType": "cassandra",
			"serviceType": "Database",
			"engineType": "KeyValue",
			"external": false
			},
			{
			"name": "MoreStuff",
			"status": " ONLINE",
			"internalHost": "morestuff",
			"externalHost": "morestuff",
			"internalPort": 7687,
			"externalPort": 7687,
			"username": "neo4j",
			"password": "choosePassword",
			"dbType": "neo4j",
			"serviceType": "Database",
			"engineType": "Graph",
			"external": false
			},
			{
			"name": "polystore_db",
			"status": " ONLINE",
			"internalHost": "polystore-mongo",
			"externalHost": "polystore-mongo",
			"internalPort": 27017,
			"externalPort": 27017,
			"username": "admin",
			"password": "admin",
			"dbType": "MongoDb",
			"serviceType": "Database",
			"engineType": "Document",
			"external": false
			}
			]

Get Databases & Services	GET /api/services	N/A	[{ "name": "Reviews", "status": " ONLINE", "internalHost": "reviews", "externalHost": "reviews", "internalPort": 27017, "externalPort": 27017, "username": "chooseUsername", "password": "choosePassword", "dbType": "MongoDb", "serviceType": "Database", "engineType": "Document", "external": false }, {

			"name": "Inventory", "status": " ONLINE", "internalHost": "inventory", "externalHost": "inventory", "internalPort": 3306, "externalPort": 3306, "username": "root", "password": "choosePassword", "dbType": "MariaDb", "serviceType": "Database", "engineType": "Relational", "external": false }, { "name": "Stuff", "status": " ONLINE", "internalHost": "stuff", "externalHost": "stuff", "internalPort": 9042, "externalPort": 9042, "username": "admin", "password": "password", "dbType": "cassandra", "serviceType": "Database", "engineType": "KeyValue", "external": false }, { "name": "MoreStuff", "status": " ONLINE", "internalHost": "morestuff", "externalHost": "morestuff", "internalPort": 7687, "externalPort": 7687, "username": "neo4j", "password": "choosePassword", "dbType": "neo4j", "serviceType": "Database", "engineType": "Graph", "external": false }, { "name": "polystore_db", "status": " ONLINE", "internalHost": "polystore- mongo", "externalHost": "polystore- mongo", "internalPort": 27017, "externalPort": 27017, "username": "admin", "password": "admin", "dbType": "MongoDb", "serviceType": "Database", "engineType": "Document", "external": false

```
    },
    {
        "name": "polystore_api",
        "status": "ONLINE",
        "internalHost": "typhon-
polystore-service",
        "externalHost": "localhost",
        "internalPort": 8080,
        "externalPort": 8080,
        "username": null,
        "password": null,
        "dbType": null,
        "serviceType": "Software",
        "engineType": null,
        "external": null
    },
    {
        "name": "polystore_ui",
        "status": "ONLINE",
        "internalHost": "polystore-
ui",
        "externalHost": "localhost",
        "internalPort": 4200,
        "externalPort": 4200,
        "username": null,
        "password": null,
        "dbType": null,
        "serviceType": "Software",
        "engineType": null,
        "external": null
    },
    {
        "name": "polystore_ql",
        "status": "ONLINE",
        "internalHost": "typhonql-
server",
        "externalHost": "typhonql-
server",
        "internalPort": 7000,
        "externalPort": 7000,
        "username": null,
        "password": null,
        "dbType": null,
        "serviceType": "Software",
        "engineType": null,
        "external": null
    }
]
```

6.3.4.4 ML/DL model Services

Additionally, the ML and DL models that are used by QL are managed in the API. As such, you can use the API to upload new versions of these models or download latest or oldest versions already uploaded to the API. The corresponding endpoints are the following:

Description	Endpoint	Input	Output

Get ML
 	
Get DL
	GET /api/model/ml
	
	GET /api/model/dl
		N/A	`[`
			`{`
			`"id": "b8859d8f-3e69-4737-b910-d2e88e587c00",`
			`"version": 1,`
			`"initializedData-bases": false,`
			`"initializedConnec-tions": false,`
			`"contents": "...xml con-tents..."`
			`"type": "ML/DL",`
			`"dateReceived": "2020-07-17T09:44:11.212+0000"`
			`}`
			200 OK

Update ML
 	
Update DL
	POST /api/model/ml
	
	POST /api/model/dl
		`{`
		`"name":"name",`
		`"con-tents": "xml contents"`
		`}`
			200 OK

Get specific version of a model	GET /api/model/{type}/{version}	N/A	File 200 OK

6.3.4.5 CRUD Services

The Polystore API, through the QL server also supports explicit CRUD, operations on entities. Example operations can be seen in the table below.

Description	Endpoint	Input	Output
Insert Entity	POST /crud/{entity}	For e.g. /crud/User `{ "name": "\"Patrick\"", "age": "39" }`	200 OK `{ "@id": "#b58f8848" }`
Get Entity	GET /crud/{entity}/{id}	For e.g. /crud/User/ b58f8848	`{ "@id": "#b58f8848", "name": "\"Patrick\"", "age" : "39" }`
Update Entity	PATCH /crud/{entity}/{id}	For e.g. /crud/User/ b58f8848 `{ "age": "40" }`	200 OK
Delete Entity	DELETE /crud/{entity}/{id}	For e.g. /crud/User/ b58f8848	200 OK

6.3.4.6 QL Services

Queries on the Polystore can also be made through the use of the API. The relevant inputs and outputs are shown in the table below.

Description	Endpoint	Input	Output
Reset databases	GET /api/resetdatabases	N/A	200 OK true/false
Query Databases (select queries)	POST /api/query	`from Review o select o`	200 OK `{"columnNames": ["o.content","o.location" ,"o.screenshot","o.product","o.user"], "values":[]}`
Update Databases (update/insert	POST /api/update	`insert User{name:"test", age:30}`	200 OK `{"affectedEntities":-1,"createdUuids":{"uuid":"c6a7d4c7-b1b1-4943-`

queries)			98e6-8316e13d9f24"}}
Batch update databases (batch in-sert/update queries)	POST /api/preparedupdate	{"command": "insert User{name:??name, age:??age}", "parameterNames": ["name,age"], "boundRows": [["john,jack"],["30","29"]]}	200 OK [{"affectedEntities":-1,"createdUuids":{"uuid":"3bbd0bc7-f98d-4719-8e39-3d8867dbe673"}},{"affectedEntities":-1,"createdUuids":{"uuid":"3f82286c-fc50-4261-bc4c-fc1910b1601f]

6.3.5 Typhon Query Language (QL)

6.3.5.1 Introduction

TyphonQL[31] is a query language and data-manipulation language (DML) to access polystores (federations of different kinds of database back-ends, relational, document, key-value etc.) while at the same time abstracting as much as possible from how the data is actually stored.
Executing TyphonQL queries is parameterized by a TyphonML model, which provides the logical data schema in the form of an object-oriented information model. A TyphonML model declares entities with primitively-typed attributes, and bi-directional (many-valued) relations (which can be containment/ownership) relations.
TyphonQL is designed to allow the query writer to think at the level of TyphonML entities as much as possible. With TyphonQL one does not manipulate tables, graphs, documents, or key-value pairs, but sets of objects which may have relations to each other, and which conform to the entity types declared in the TyphonML model.
The present document aims to describe the TyphonQL in sufficient detail for end-users of the language. Thus, it is not a formal reference document, but rather a short overview, touching upon the most common and most quirky features in equal amount.
The next section presents an abstract overview of the language, and after we present the language using numerous examples.

6.3.5.2 The Language

This section provides a cursory overview of the language.

[31] The latest version of this Section of the guide can be found on: https://github.com/typhon-project/typhonql/blob/master/typhonql/doc/typhonql.md

Literal expressions

TyphonQL supports the following literal (constant) expressions:

- Booleans: `true`, `false`
- Integer numbers: `123`, `-34934`
- Strings: `"this is a string value"`
- Floating point numbers: `0.123`, `3.14`, `-0.123e10`, `2324.3434e-23`
- Dates: `$2020-03-31$`
- Date and time values: `$2020-03-31T18:08:28.477+00:00$`
- Geographical points: `#point(23.4 343.34)`
- Polygons: `#polygon((23.4 343.34), (2.0 0.0));`
- Null (indicating absence of a reference or value): `null`
- Blob-pointers: `#blob:2ed99a8e-5259-4efd-8cb4-66748d52e8a1`

Furthermore, TyphonQL supports syntax for dealing with objects (instances of entity types):

- Object literals (tagged with the entity type, in this case Person): Person {name: "Pablo", age: 30, reviews: [#879b4559-f590-48ea-968c-ff3b69ec5363, #23275eec-4746-4f23-a854-660160cafed2]}
- Reference values (pointers), represented as UUIDs: #879b4559-f590-48ea-968c-ff3b69ec5363
- Collections of pointers to objects: [#8bc3f0a0-5cf4-42e5-a664-0617feb2d400, #23275eec-4746-4f23-a854-660160cafed2, #879b4559-f590-48ea-968c-ff3b69ec5363]

Object literals are used as argument to insert statements, and (lists of) references are used in both insert and update statements to create links/relations between objects. In the future we might support nesting of object literals and within-insert symbolic cross referencing to manipulate complete object graphs all at once.

Other expressions

Select queries as well as update and delete statements use expressions to filter results and find objects to operate on respectively. For instance, a from-select query specifies a number of result expressions and conditions in the where-clause. Update and delete find the object(s) to be update resp. deleted using similar conditions in a where-clause.
TyphonQL supports the following non-literal expressions:

- Attribute or relation access: `entity.field`
- Accessing the identity of an object: `entity.@id`
- Boolean operators: `!exp` (negation), `exp1 && exp2` (conjunction), `exp1 || exp2` (disjunction)

- Arithmetic operators: `exp1 * exp2`, `exp1 / exp2`, `exp1 + exp2`, `exp1 - exp2`
- Comparison operators: `exp1 == exp2`, `exp1 != exp1`, `exp1 > exp2`, `exp1 >= exp2`, etc.

The prefix and infix operators follow the precedence levels of Java-like languages.
To be implemented:

- member operator: `exp1 in exp2`
- textual match operator: `exp1 like exp2`

Geographical expressions

```
pt1 = point(1.3,2.5)
pt2 = point(3.5,4.6)
pg1 = polygon([
  [point(0,0), pt1],
  [pt1, point(1,1)],
  [point(1,1), pt2],
  [pt2, point(0,0)]
])
pg2 = polygon([
  [point(3,0), pt1],
  [pt1, point(2,2)],
  [point(2,2), pt2],
  [pt2, point(3,2)]
])
```

distance in meters:

- two points: `distance(pt1, pt2)`
- one point and closest edge of polygon: `distance(pt1, pg2)`

containment:

- point inside a polygon: `pt1 in pg2`
- polygon fully inside another polygon: `pg1 in pg2`

overlap:

- polygon partially overlaps another polygon: `pg1 & pg2`

note: on MongoDB backends distance is limited to the where query and only in presence of a comparison operator.

Blobs

Blobs are handled in a special way, during insertion/update you have to send them as a pointer to a blob: `#blob:UUID` (and pass along the contents of the blob to the API in a separate field). While selecting them, you get a base64 encoded version of the blob. It is not possible to do any operations on them, they are opaque.

Queries

Queries follow the tradition of SQL queries, except that the select and from parts are swapped. A basic query thus has the form of "*from* bindings *select* results *where* conditions". Bindings consist of a list of "Entity Variable" pairs, separated by comma, which introduce the scope of the query. Results is a list of expressions (separated by commas) that will make up the final result of the query. The where-clause is optional, but if present it consists of a list of expressions (separated by commas) filtering the result set.
For now, in results the only allowed expressions are `x` (an entity variable introduced in the bindings), `x.@id`, and `x.f` (attribute or relation access).

DML

The general form of the insert statement is "*insert* Entity { assignments }". The entity is the type of the object to be inserted as defined in the TyphonML statement. The assignments are bindings of the form "attrOrRelation: expression". The TyphonQL type checker will check that all assignments are correctly typed according the TyphonML model, including multiplicity constraints.
Update and delete statements specify the objects to work on via where-clauses. For instance, update has the form "*update* Entity x *where* conditions *set* { assignments }". The assignments are the same as in insert, except that for many-valued relations, they can specify additions ("relation +: expression") and removals ("relation -: expression").
Delete has the form "*delete* Entity x *where* conditions", which will delete all entities of type Entity satisfying the conditions in the where-clause.
All three DML statements ensure (as much as possible) that relational integrity is preserved, even across database back-ends. In particular this means:

- creating resp. breaking a relation between entities entail creating resp. breaking the inverse link as well (if so declared in the TyphonML model)
- deleting an object will delete all objects "owned" by it via containment relations (cascading delete).

Cascading delete of contained object is currently limited to one hop across database boundaries. In other words, if a sequence of containment relations alternatingly cross multiple database back-ends the cascade is only performed for the first relation.

6.3.5.3 TyphonQL by Example

Introduction

In this section we will illustrate TyphonQL using numerous examples. The example queries and DML statements should be understood in the context of an example TyphonML, which is shown below.

```
entity Product {
        name : string[256]
        description : string[256]
        price : int
        productionDate : date
        reviews :-> Review."Review.product"[0..*]
        wish :-> Wish."Wish.product"[1]
```

```
}

entity Review {
        content: text
        product -> Product[1]
        user -> User[1]
}

entity User {
        name : string[256]
        address: string[256]
        biography :-> Biography[0..1]
        reviews -> Review."Review.user"[0..*]
        wish :-> Wish."Wish.user"[1]
}

entity Biography{
        content : string[256]
        user -> User[1]
}

entity Wish {
        intensity: int
        user -> User[1]
        product -> Product[1]
}

relationaldb Inventory {
        tables{
          table { UserDB : User }
      table { ProductDB : Product }
        }
}

documentdb Reviews {
        collections{
          Review : Review
          Biography : Biography
        }
}

graphdb Wishes {
        edges {
                edge Wish {
                        from "Wish.user"
                        to "Wish.product"
                }
        }
}
```

Entities Product and User are deployed to an SQL database (MariaDB), called Inventory; the Review and Biography entities are stored on a (MongoDB) document-store called Reviews; and the Wish entity is stored on a (Neo4J) graph database.

Products own a number of Reviews ("deleting a product will delete associated reviews as well") via the relation `reviews`. The ownership link can be traversed from the `product` reference in Reviews because of the opposite declaration on `reviews`.

Reviews are also authored by users, which is modeled by the `reviews` relation on the User entity. This relation is not a containment relation, because an entity can only be owned by a single entity at one point in time. User biographies however are owned by User entities via the `biography` relation.
A Wish relates one user to one product, holding a value for the "intensity" of this relation. Entities that are stored in graph databases have a number of constraints, as they represent edges in this kind of backends. Wish must have exactly two related entities with cardinality 1, and the opposite relation might be declared in the related entities, as long as they represent containment and have cardinality one (see `wish` relation in Product and User). In other words, removing any of the entities that correspond to the vertices should also remove the "edge" entity. The directionality of the relation is established in the database mapping, particularly, in the graphdb section, where we see which relation represent the source and which one the target inside the graph database.

Well-formedness of TyphonML models

TyphonQL assumes TyphonML models are well-formed in the following ways:

- all entities are are placed on a database back-end
- containment is uni-directional (e.g. inverses of containment cannot be containment)
- containment is not many-to-many (i.e. tree shaped)
- containment is uniquely rooted: every owned entity can be reached from a unique path starting from an entity that is not owned

Realizing references

TyphonML references support bidirectional navigation over relations between entities through inverses (AKA "opposites"). In other words, it is possible to navigate across a *single* relation in two ways. In order to support this in the implementation of TyphonQL, such bidirectional relations are realized in the back-ends in both directions. TyphonQL ensures that updates to a relation are always mirrored in the other direction according to the opposite declaration(s). This means that how you navigate across a relation (from which direction) may have different consequences at the level of the implementation.
The only exceptions to this rule are:

- a containment relation within SQL is always modeled using a *single* foreign key from child to parent
- a cross-reference relation within SQL is modeled using a *single* junction table (representing both directions).

Querying

Selecting all users:
```
from User u select u
```
This will return the identities of all users.
Selecting specific attributes of users:
```
from User u select u.name
```
This will return the identities of the users paired with their name.

Selecting a specific relation:

```
from User u select u.reviews
```

This will return pairs of user identity and review identity. If a user has no reviews, its identity will be paired with `null`.

Filtering on a specific attribute:

```
from User u select u where u.name == "Pablo"
```

This will return the identities of the users with name = “Pablo”

A complex query across database boundaries: find all user and product name pairs for which a user has written a review containing the word "bad".

```
from User u, Product p, Review r select u.name, p.name
 where u.reviews == r, p.reviews == r, r.text like "bad"
```

Note the use of "==" even for many-valued references.

Manipulating Data

With insert: custom data type value must be fully specified, but in updates, you can partially update sub-fields.

```
insert User { name: "John Smith", age: 30 }
insert User { name: "John Smith", age: 30, cards: [#a129feec-4b92-4ab2-9ef5-
d276a7566f56] }
insert CreditCard { number: "1762376287", expires:  $2020-02-21T14:03:45.274+00:00$ }
```

The following is not allowed, because owner is an inverse.

```
insert CreditCard {
  number: "1762376287",
  expires:  $2020-02-21T14:03:45.274+00:00$,
  owner: #ff704edc-5d85-470b-9ed4-fb8761bbe93a
}
```

Alternative is:

```
insert CreditCard {
  number: "1762376287",
  expires:  $2020-02-21T14:03:45.274+00:00$
}
```

and then:

```
update User u where u.@id == #ff704edc-5d85-470b-9ed4-fb8761bbe93a
set { cards +: [#the-id-of-the-new-creditcard] }
```

Or, (better), inserting into owner directly:

Update

Well-formedness of Update

- you cannot update @id fields
- no nested object literals

Updating simple-valued attributes

```
update User u where u.name == "John Smith" set { age: 30 }
```

Setting a relation:

```
update Review r where r.@id == #13245f43-634f-46bf-a73d-6bd30865f5d4
  set { author: #a129feec-4b92-4ab2-9ef5-d276a7566f56 }
```

This is equivalent to:

```
update User u where u.@id == #a129feec-4b92-4ab2-9ef5-d276a7566f56
  set { reviews +: [#13245f43-634f-46bf-a73d-6bd30865f5d4] }
```

Setting a many-valued relation:

```
update User u where u.name == "John Smith"
  set { cards: [#a129feec-4b92-4ab2-9ef5-d276a7566f56] }
```

Adding:

```
update User u where u.name == "John Smith"
  set { cards +: [#a129feec-4b92-4ab2-9ef5-d276a7566f56] }
```

Removing

```
update User u where u.name == "John Smith"
  set { cards -: [#a129feec-4b92-4ab2-9ef5-d276a7566f56] }
```

Delete

cascade to owned things, but only one hop across database boundaries (so from DB A to DB B, but not continuing from DB B to DB c) .

Placeholders

```
update User u where u.@id == ?
  set { cards +: [#a129feec-4b92-4ab2-9ef5-d276a7566f56] }
```

Named placeholders:

```
update User u where u.@id == ??param
  set { cards +: [#a129feec-4b92-4ab2-9ef5-d276a7566f56] }
```

6.3.6 Evolution

6.3.6.1 Schema and data evolution/migration (D6.3)

This command-line application permits to evolve/migrate automatically the deployed TyphonML polystore schema, the corresponding physical structures and data from one polystore schema version to another, using evolution operators defined in the Typhon ML grammar.

To execute the application:

1. Download the jar from: http://typhon.clmsuk.com/static/evolution-tool.jar

2. For the first usage:

   - Run the jar with the command: java -jar evolution-tool.jar

   - Open the automatically created "application.properties" file and check the configuration. The properties should contain the input and result XMI files paths for the evolution operators you would like to execute. The following properties are the most important to check/fill ("RESULT_FILE" corresponding file can be empty, and will be populated with the evolution tool):

     INPUT_XMI=**addAttribute.xmi**

     POLYSTORE_API_USER_PASSWORD=**admin\:admin1@**

RESULT_FILE=**addAttribute_result.xmi**

POLYSTORE_API_URL=**http\://localhost\:8080/**

- Run again the command: java -jar evolution-tool.jar

3. For next usages:

- Open the “application.properties” file and replace the input and result XMI files paths for the evolution operators you would like to execute. For example:

    INPUT_XMI=**removeAttribute.xmi**

    RESULT_FILE=**removeAttribute_result.xmi**

- Run the command: java -jar evolution-tool.jar

To create the input XMI file, you should use the TyphonML Eclipse-based textual or graphical editors on the TML schema file. Regarding the use of the graphical editor, we refer to the TyphonML documentation. Below, you can find the TML textual syntax for the evolution operators supported by the polystore schema and data evolution/migration tool:

• **Add attribute:**

```
changeOperators [
        AddPrimitiveDataTypeAttribute newPrimitiveAttribute {
                ownerEntity Test type bigint
        },
        AddCustomDataTypeAttribute newCustomAttribute {
                ownerEntity Test type customType
        }
]
```

• **Add entity:**

```
relationaldb RelationalDatabase {
    tables {
        table {
            Test2Table : Test2
        }
    }
}
changeOperators [
        add entity Test2 {
                attributes [
                        testDate: date,
                        testBool: bool
                ]
        }
]
```

• **Add relation:**

```
changeOperators [
        add relation newRelation to Test -> Test2
]
```

• **Rename attribute:**

```
changeOperators [
        rename attribute 'Test.id' as 'identifier'
]
```

• **Rename entity:**

```
changeOperators [
        rename entity Test as 'TestNewName'
]
```

• **Rename relation:**

```
changeOperators [
        rename relation newRelation as 'newRelationName'
]
```

• **Remove attribute**:

```
changeOperators [
        remove attribute 'Test.id'
]
```

• **Remove entity**:

```
changeOperators [
        remove entity Test2
]
```

• **Remove relation**:

```
changeOperators [
        remove relation newRelation
]
```

• **Migrate entity**: migrate an entity from one database to another. Hypothesis: the incoming relations (relations pointing to the migrated entity) will be lost.

```
changeOperators [
        migrate Test to DocumentDatabase
]
```

• **Merge entities**: merge 2 entities using the "as" for the relation name. Hypothesis: the relation between the 2 entities is 1-1.

```
changeOperators [
        merge entities Test Test2 as 'Test2.relationName'
]
```

• **Split entity vertical**: split an entity into 2 entities. The attributes specified as input will be moved from the first (existing) entity to the second (new) entity.

```
changeOperators [
        split entity vertical Test2 to Test3 attributes: [ "Test2.testBool" ]
]
```

• **Split entity horizontal**: split an entity into 2 entities. The where clause contains the attribute and the value on which the split is applied. The result data of the where clause will be contained in the second (new) entity. Both entities will have the same attributes.

```
changeOperators [
        split entity horizontal Test2 to Test3 where "Test2.testBool" value "true"
]
```

- Enable relation containment:

```
changeOperators [
	EnableRelationContainment { relation newRelation }
]
```

- Disable relation containment:

```
changeOperators [
	DisableRelationContainment { relation newRelation }
]
```

- Enable relation opposite:

```
changeOperators [
	EnableBidirectionalRelation { relation newRelation }
]
```

- Disable relation opposite:

```
changeOperators [
	DisableBidirectionalRelation { relation newRelation }
]
```

- Change attribute type:

```
changeOperators [
	ChangePrimitiveDataTypeAttribute { attributeToChange "Test.id" newType bigint },
	ChangeCustomDataTypeAttribute { attributeToChange "Test.id" newType customType }
]
```

Once the desired evolution operators have been specified using the Eclipse-based editors (textual or graphical) you can generate the corresponding XMI file by using the Eclipse TML plugin (right-click on the TML file, then “Typhon” and “Inject to model” buttons).

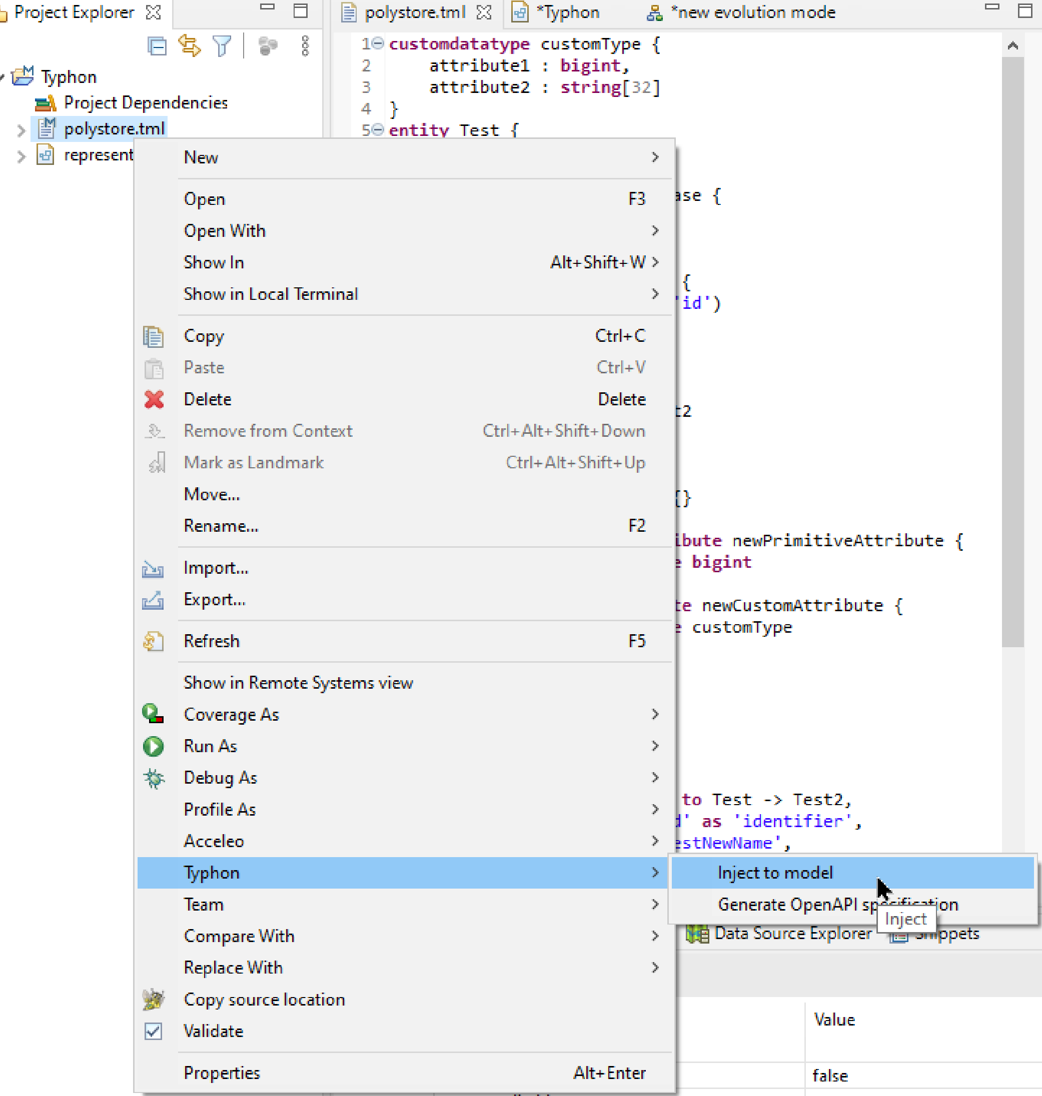


Figure 55: Use Inject to model to generate XMI file

Once the full XMI file is generated, you can use it as input of the schema and data evolution/migration tool, by providing the path to the XMI file in the “application.properties” file.

Log messages are provided in the console where you run the schema/data evolution tool. The output of the evolution tool is the XMI file referenced in the “application.properties” file, under the “RESULT_FILE” property. In addition to the target Polystore schema produced as output, the Polystore data structures and database contents have been modified according to the desired evolution operators.

6.3.6.2 Query evolution (D6.4)

The Query Evolution Tool is a plugin for Eclipse helping the developers to identify TyphonQL queries impacted by a change in the Polystore schema, and to automatically adapt those queries to the target Polystore schema, when possible. This section will cover the installation and the usage of the query evolution plugin.

4. Installation

The eclipse repository for this sub-project is : http://typhon.clmsuk.com:8082/typhon-evolution/repository

5. Overview

The plugin introduces a new kind of file with the extension '.qevo'. An evolution file is made of two-part :

- **Schema to Apply:** A line referencing the schema used to evolve the Polystore.
- **Query List**: The TyphonQL queries to evolve separated by a comma.

Here is a minimal example of .qevo file :

```
apply ./src/xmi/addRelationChangeOperator.xmi;
from Order o select o,
from Order o select o.id,
delete User u where u.id == 42
```

Applying the Query Evolution process to a .qevo file will assign a status to each output TyphonQL query. There are 4 different statuses :

- **UNCHANGED**: the input query has not been changed since it remains valid with respect to the target schema;
- **MODIFIED**: the input query has been transformed into an equivalent output query, expressed on top of the target schema;
- **WARNING**: the output query (be it unchanged or modified) is valid with respect to the target schema, but it may return a different result set;
- **BROKEN**: the input query has become invalid, but it cannot be transformed into an equivalent query expressed on top of the target schema.

Queries with the status **WARNING** or **BROKEN** are also accompanied by a comment explaining which change operator causes them to take that status and why.

Here is an example of .qevo file after the query evolution process:

apply ./src/xmi/addRelationChangeOperator.xmi;

WARNING

#@ Attribute users added to Order. Result of the query may have changed @#

from Order o select o,

WARNING

#@ Attribute users added to Order. Result of the query may have changed @#

from Order o select o.id,

UNCHANGED

delete User u where u.id == 42

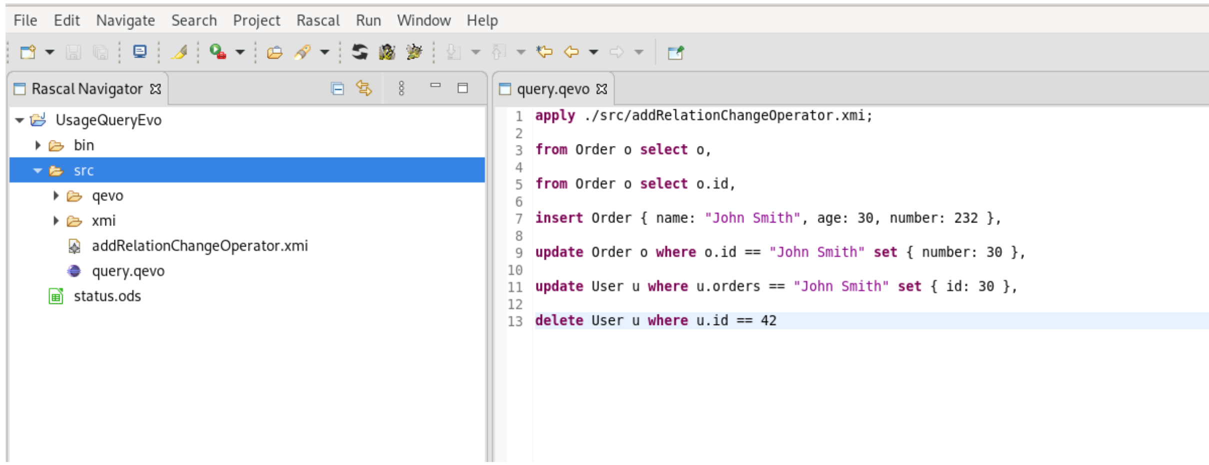
Usage

Once the plugin is installed, create a new text file with the extension '.qevo' in your project. The first line of this file should be the path of the xmi file containing the schema and the change operator (the same file expected by the schema evolution tool). The syntax of this line is :

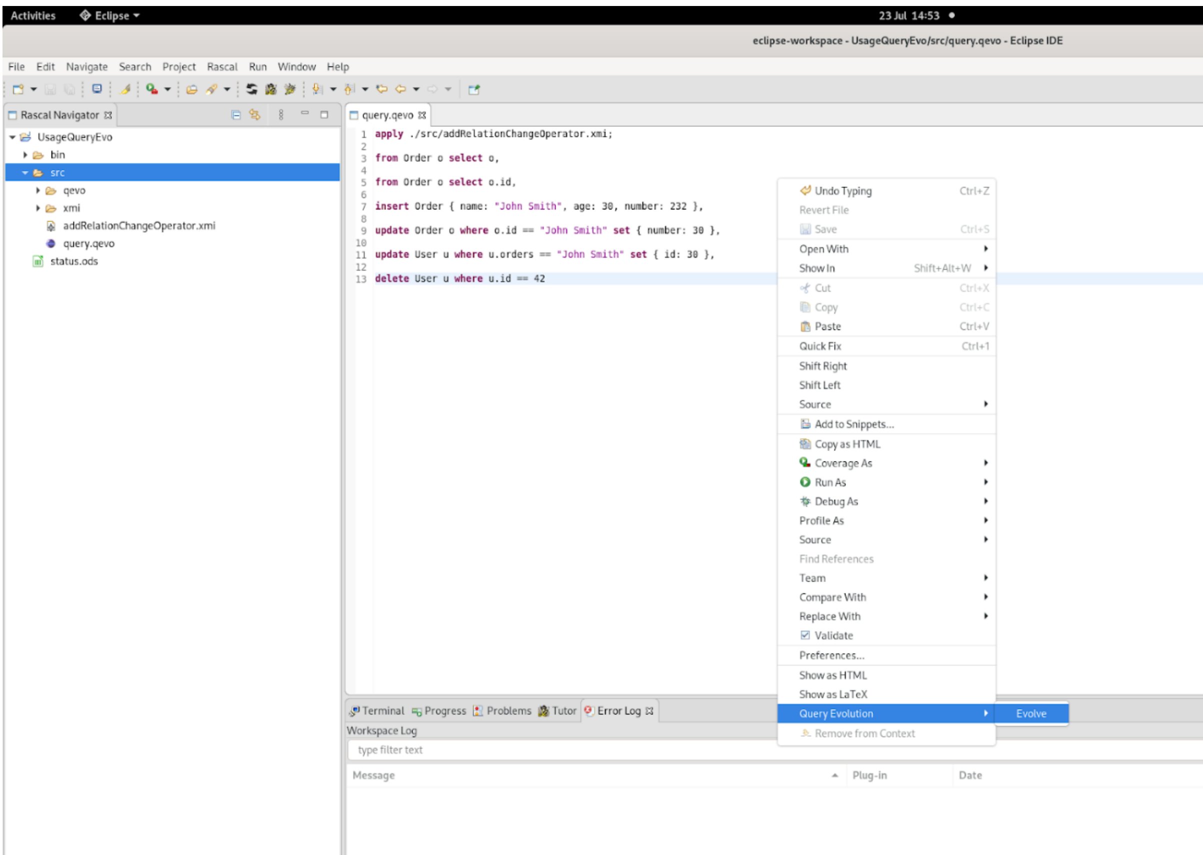
apply *<path>*;

where the path is relative to the root of your eclipse project.

After the 'apply' line you can copy paste the query you want to evolve separated by comma (see Figure below).



When all your queries are in the evolution file, you can apply the transformation by right clicking in the editors and select **query evolution -> evolve**.



The plugin will rewrite the *.qevo* file with the evolved queries:

```
query.qevo
1  apply ./src/addRelationChangeOperator.xmi;
2
3  WARNING
4  #@ Attribute users added to Order. Result of the query may have changed @#
5  from Order o select o,
6
7  WARNING
8  #@ Attribute users added to Order. Result of the query may have changed @#
9  from Order o select o.id,
10
11 BROKEN
12 #@ Attribute users added to Order. Insert is broken @#
13 insert Order { name: "John Smith", age: 30, number: 232 },
14
15 UNCHANGED
16 update Order o where o.id == "John Smith" set { number: 30 },
17
18 UNCHANGED
19 update User u where u.orders == "John Smith" set { id: 30 },
20
21 UNCHANGED
22 delete User u where u.id == 42
```

The updated queries could then be reused in the source code of the developers.

6.3.6.3 Polystore Continuous Evolution

The continuous evolution tool aims to monitor data usage performance in a Typhon Polystore in order to provide users with schema evolution recommendations, when relevant.

The tool communicates with several Polystore components, including the post-execution events queue (WP5) and, through the Polystore API, the Polystore TyphonML schema (WP2) and the Polystore databases (WP3 and WP4). This allows the continuous evolution to automatically retrieve useful information about the Polystore, including:

- the Polystore configuration, i.e., the TyphonML entities and their mapping to underlying native databases;
- the TyphonQL queries that are executed by the TyphonQL engine, and their duration;
- the (evolving) size of the TyphonML entities over time.

Overview

The general architecture of the continuous evolution tool is depicted in Figure 56 below. Each time a post-execution event is published to the post-event queue, a Java application wakes up and retrieves the event. If the event corresponds to a DML query execution, the Java application sends the corresponding TyphonQL query to a Rascal plugin. The latter parses, analyses and classifies the query and sends back the corresponding query information to the Java application. This information is stored in an internal MongoDB database, which is used as input by an interactive web application. The web application, relying on an Angular frontend and a Node.js backend, provides users with visual analytics of the Polystore data usage, as well as with performance-based schema reconfiguration recommendations. Each of these steps is described in further details in the remaining of this section.

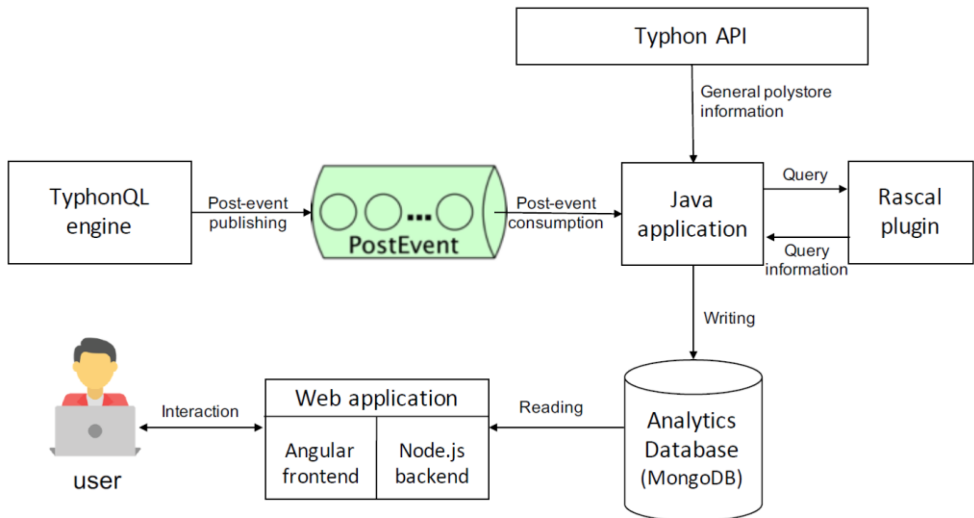


Figure 56: Architecture of the continuous evolution tool

Step 1. Capturing TyphonQL queries

The continuous evolution tool exploits the Polystore monitoring mechanisms developed in Work Package 5. Thanks to those mechanisms, the tool can capture at runtime the successive TyphonQL queries that are sent to the Polystore, and executed by the TyphonQL engine. To do so, the tool consumes and analyses the so-called post-execution events (PostEvent), generated and pushed by the TyphonQL engine to the analytics queue of the WP5 monitoring infrastructure. We refer to deliverable D5.3 for more details about this infrastructure.

Step 2. Parsing and classifying TyphonQL queries

The post-execution events captured at Step 1 include the TyphonQL queries that have been executed by the TyphonQL engine. The continuous evolution tool parses each of those queries, in order to extract relevant information to be used during the analytics and recommendation phases. Our tool focuses on post-execution events corresponding to DML queries, i.e., select, insert, delete, and update queries. It ignores other events such as, for instance, the execution of DDL queries (e.g., create entity, delete entity, etc.) sent by the schema evolution tool to the TyphonQL engine.

The tool parses each captured TyphonQL query in order to extract relevant information, including:

- the type of query (select, insert, delete, update);
- the accessed TyphonML entities;
- the join conditions, if any;
- the query execution time, expressed in ms.

The query parsing and extraction step is implemented using Rascal, based on TyphonQL syntax.

Once the query is parsed and analyzed, the tool also classifies it. This classification aims to group together all TyphonQL queries of the same form. A group of TyphonQL queries is called a query category. The queries belonging to the same query category are queries that would become the same query after replacing all input values with placeholders.

In addition to parsing, analyzing and classifying the queries executed by the TyphonQL engine, the continuous evolution tool also extracts - at regular time intervals - information about the Typhon Polystore, with a particular focus on TyphonML entities. This includes, in particular, the size of each TyphonML entity, expressed in terms of number of records, e.g., number of rows for a relational table or number of documents for a MongoDB collection.

The extracted information is stored in an internal MongoDB database.

This MongoDB database populated during Step 2 constitutes the main input of the next three steps, which respectively aim at:

- providing users with interactive visual analytics of the Polystore data usage (Step 3);
- providing users with Polystore reconfiguration recommendations for those query categories suffering from poor performance (Step 4);
- applying the reconfiguration recommendations selected by the user (Step 5).

Step 3. Visual analytics of Polystore data usage

The main page of the visual analytics tool is depicted in Figure 57. This page provides the user with a general overview (1) of the Polystore configuration and (2) of the Polystore data usage at a coarse-grained level.

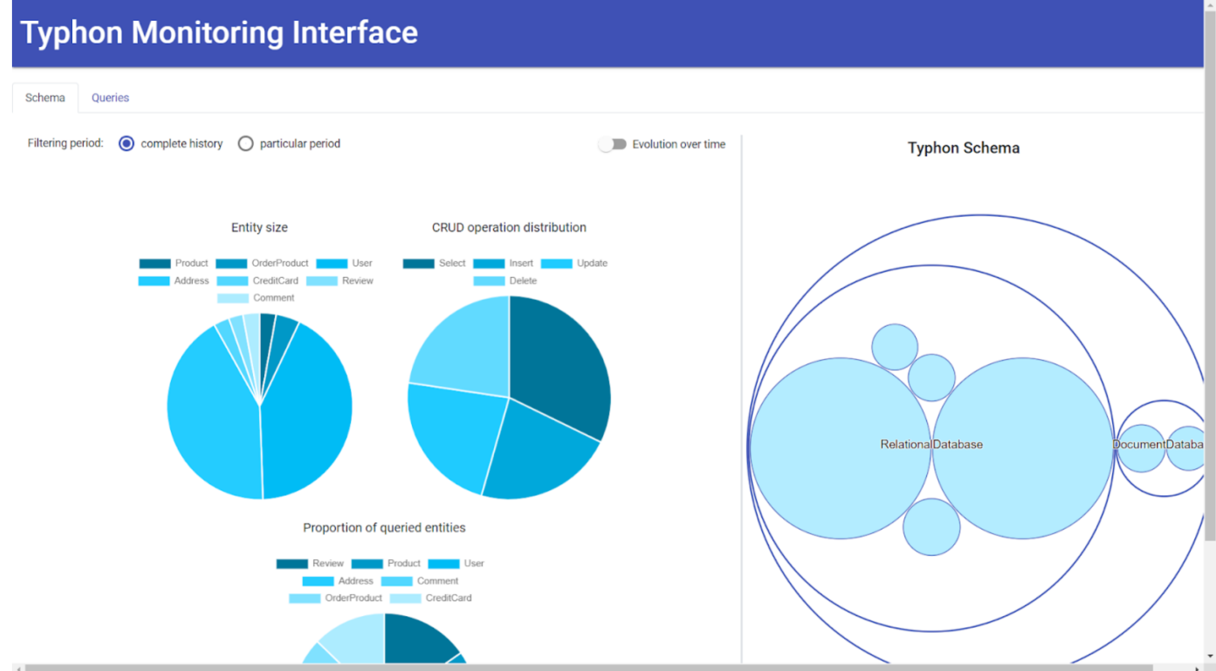


Figure 57: Main page of the visual analytics tool

- **Polystore schema view**: the tool provides the user with a global overview of the current schema configuration of the Polystore, as shown in Figure 58.

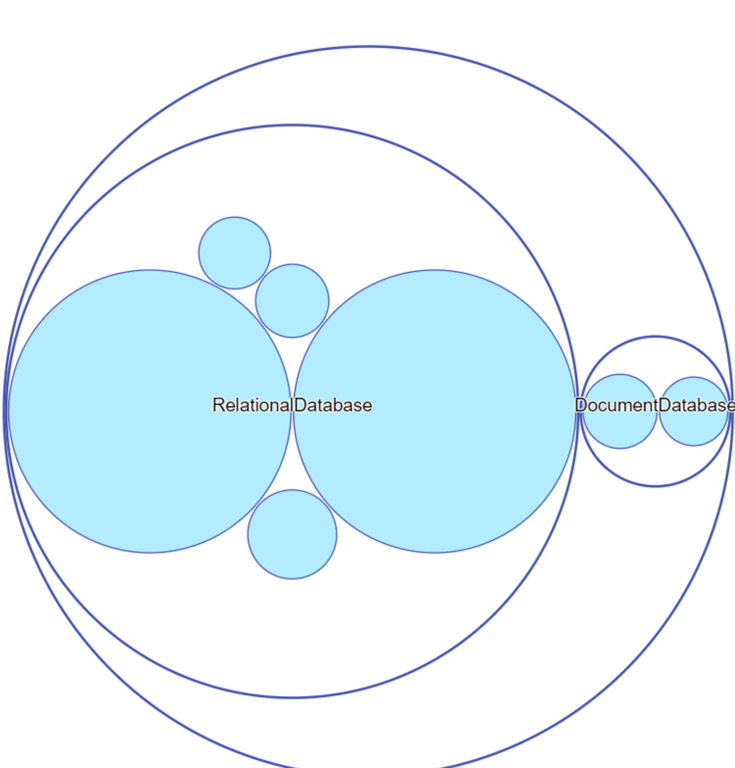


Figure 58: Overview of the current schema configuration of the Polystore

- **Polystore entities view**: the tool provides the user with a global overview of the current size of the Polystore entities, as shown in Figure 59.

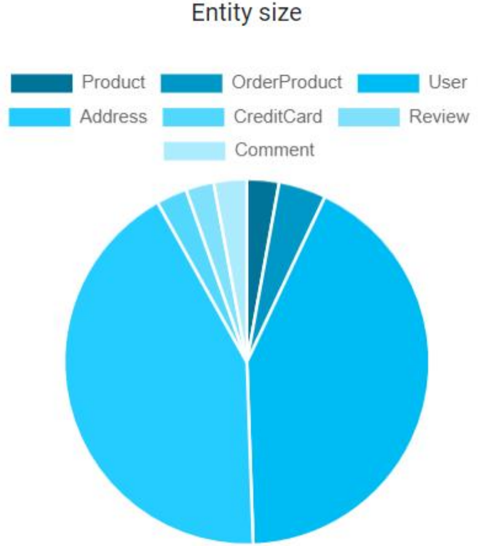


Figure 59: Overview of the current size of the Polystore entities

The evolution of the entity size over time is also provided, as shown in Figure 60.

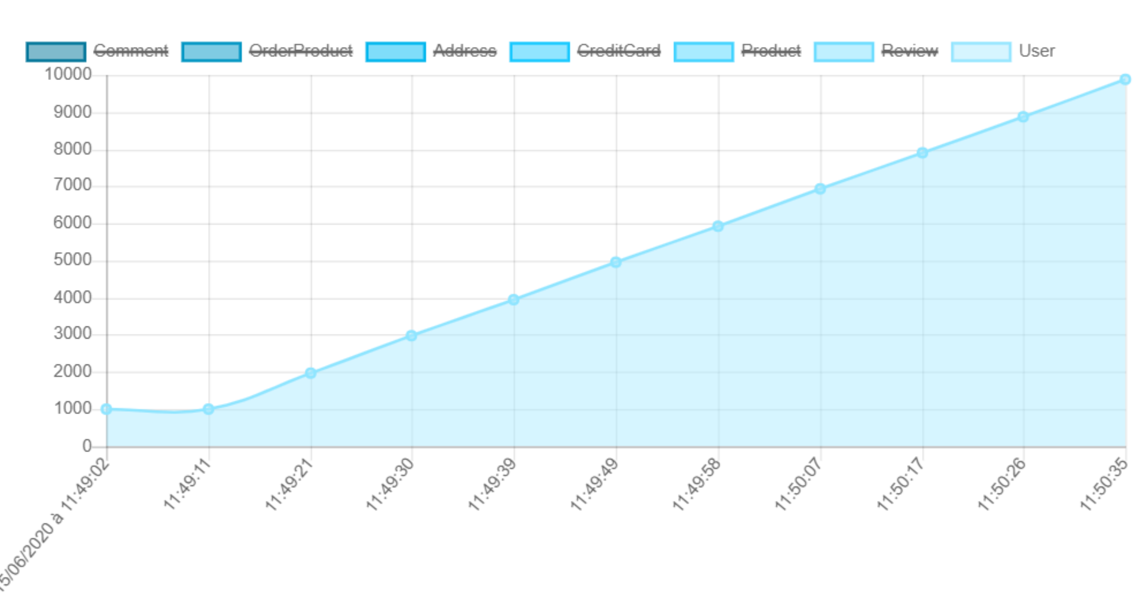


Figure 60: Evolution of the entity size over time

- **Polystore CRUD operations view**: a similar metric is provided for the distribution CRUD operations by TyphonML entity, as shown in Figure 61 and Figure 62.

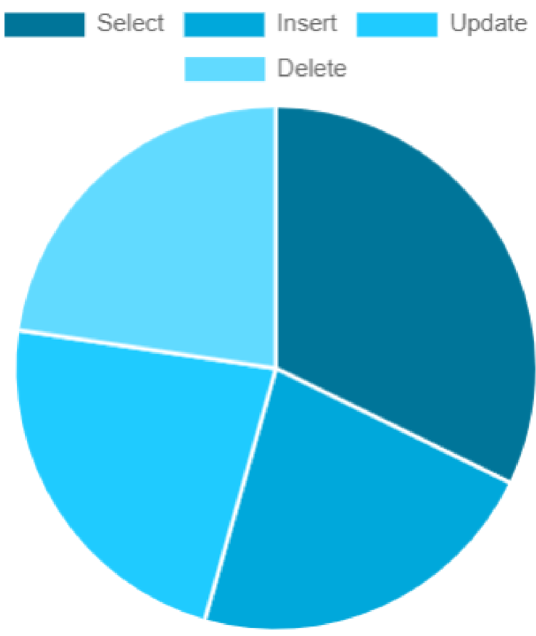


Figure 61: CRUD operation distribution

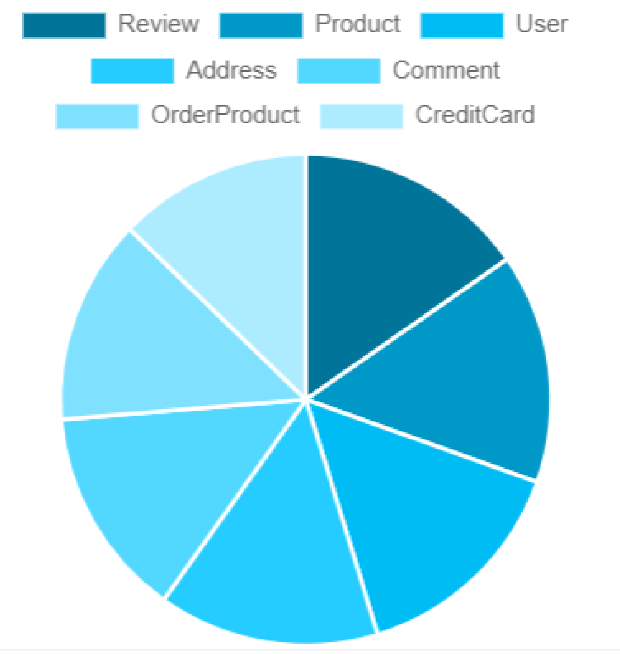


Figure 62: Proportion of queried entities

The user can also look at the evolution of the number of CRUD operations executed over time, at the level of the entire Polystore, as depicted in Figure below:

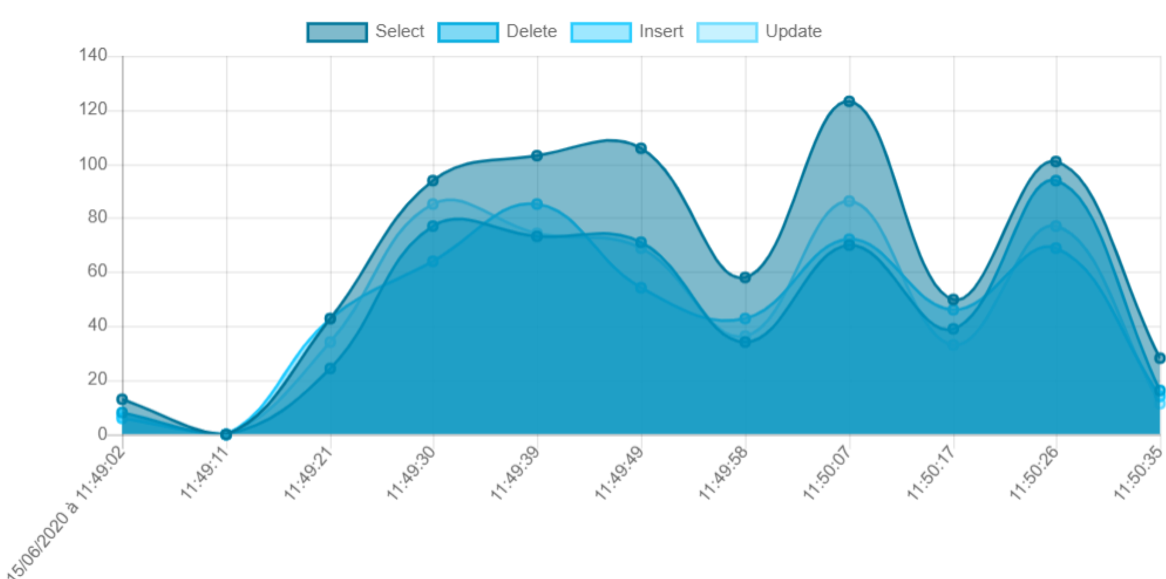


Figure 63: CRUD operation distribution over time

- **Polystore queries view**: the user can then have a finer-grained look at the TyphonQL queries executed by the TyphonQL engine on the Polystore. In the query view, the tool provides the user with two searchable lists, i.e., (1) the list of the most frequent query categories, in decreasing order of number of occurrences, and (2) the list of slowest queries, in decreasing order of execution time, as shown in Figure 64.

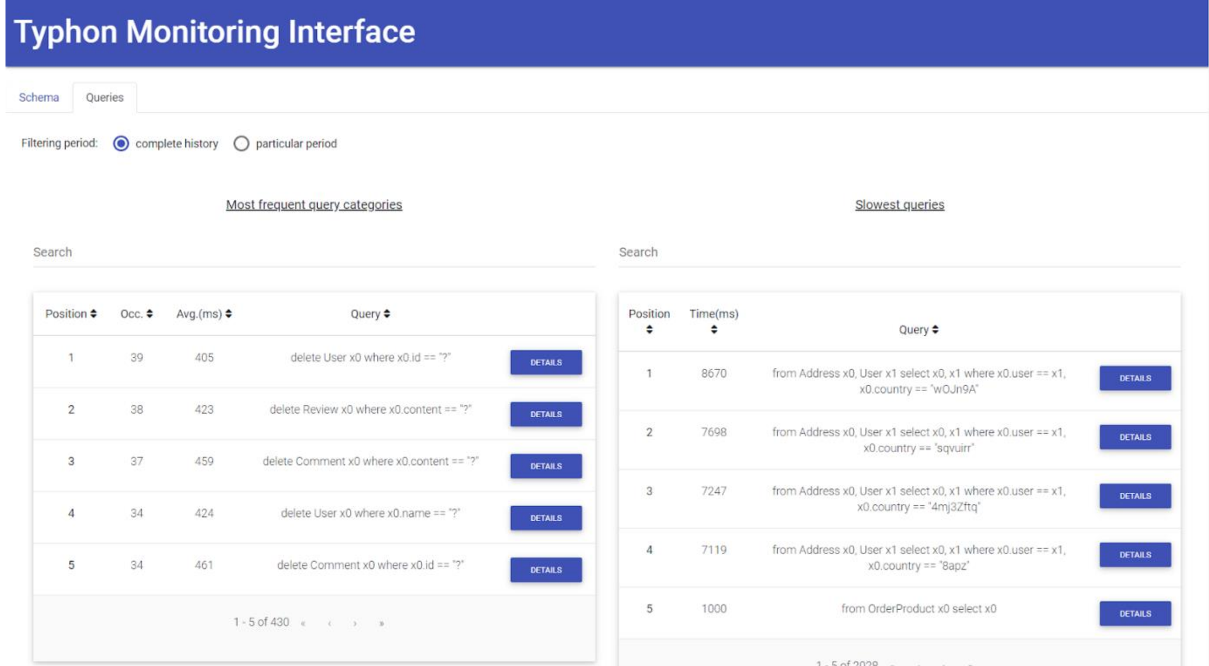


Figure 64: TyphonQL queries monitoring

Step 4. Recommending Polystore schema reconfigurations

When inspecting a particular (slow) query, the user can ask the tool for recommendations on how to improve the execution time of the query. When possible, the tool then recommends Polystore schema reconfigurations, in the form of a menu with clickable options, including one or several recommendations. Some of the provided recommendations may be mutually-exclusive, which means that they cannot be selected together in the menu.

For example, let us suppose that the user has identified the following slow query:

from Address x0, User x1 select x0, x1 where x0.user == x1, x0.country == “?”

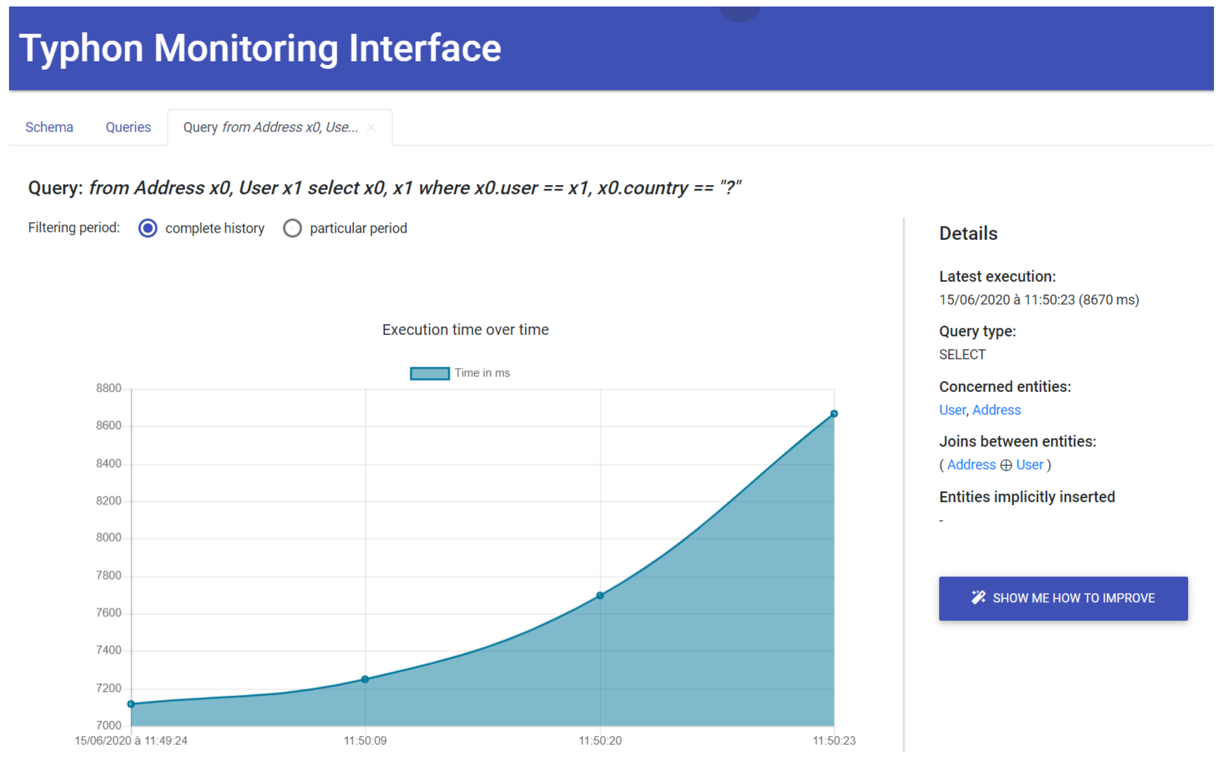


Figure 65: Query execution monitoring

This slow query involves a join between entities User and Address and a selection operator based on the value of the *Address.country* attribute.

In this case, the continuous evolution tool recommends two possible, non-exclusive schema reconfigurations (as shown in Figure 66) that respectively consist in:

1. defining an index on column AddressDB.country, which maps with attribute *Address.country*;
2. merging entity *Address* into entity *User*, via the one-to-one relation "*Address.user*" that holds between them.

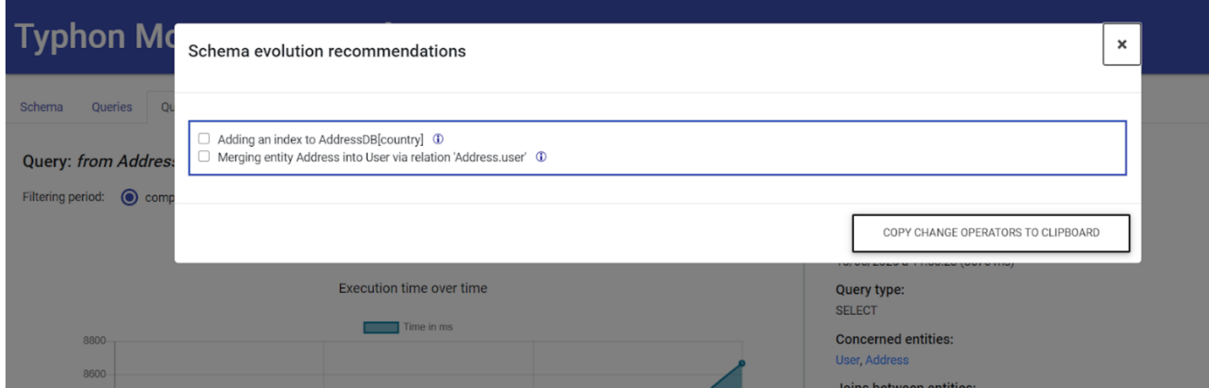


Figure 66: Schema evolution recommendations

By positioning the mouse pointer on the information icon, the user can get further information about the expected positive impact of the recommended schema change on the execution time of the query. Adding an index on a column *c* is a well-known technique to speed up a query including an equality condition on *c* in its where clause.

In the particular case of the considered query, attribute *Address.country* is used in an equality condition. As there is no index defined on table *AddressDB* (mapped with entity *Address*), the recommendation to define an index on column *AddressDB.country* is proposed.

Merging two entities into a single entity constitutes another recommendation that allows avoiding a costly join condition in a slow query. In our example, the recommendation to merge entity *Address* into entity *User* is motivated by the fact that the two entities are linked together via a one-to-one relationship (thus have the same number of records), and that both entities rapidly grow in terms of size, making the join condition slower and slower.

Step 5. Applying the selected recommendations

Using the option menu, the user may then choose which evolution recommendation(s) (s)he wants to actually follow by selecting the desired option(s). Once this selection has been done by the user, the user can click on the copy change operators to clipboard button. The tool will then automatically generate the list of schema evolution operators corresponding to the selected recommendations. These operators are expressed according to the TyphonML textual syntax (TML). So the user can simply paste the operators from the clipboard to the TML file of his TML schema, and then invoke the schema evolution tool via the Typhon API by passing the modified TML file as input.

```
changeOperators  [
      AddIndex {table 'AddressDB' attributes ('Address.country') }
     merge entities User Address as 'Address.user'
 ]
```

Please refer to deliverable D6.5 for further details about the Polystore Continuous Evolution tool.

How to deploy and use the Polystore Continuous Evolution component

The user can deploy the Polystore Continuous Evolution component with the help of the TyphonDL Creation Wizard. As this component exploits the Polystore monitoring mechanisms developed in Work Package 5, the user must obligatorily check the "*Use Typhon Data Analytics*" option. Once checked, other options are revealed (as shown in Figure 67).

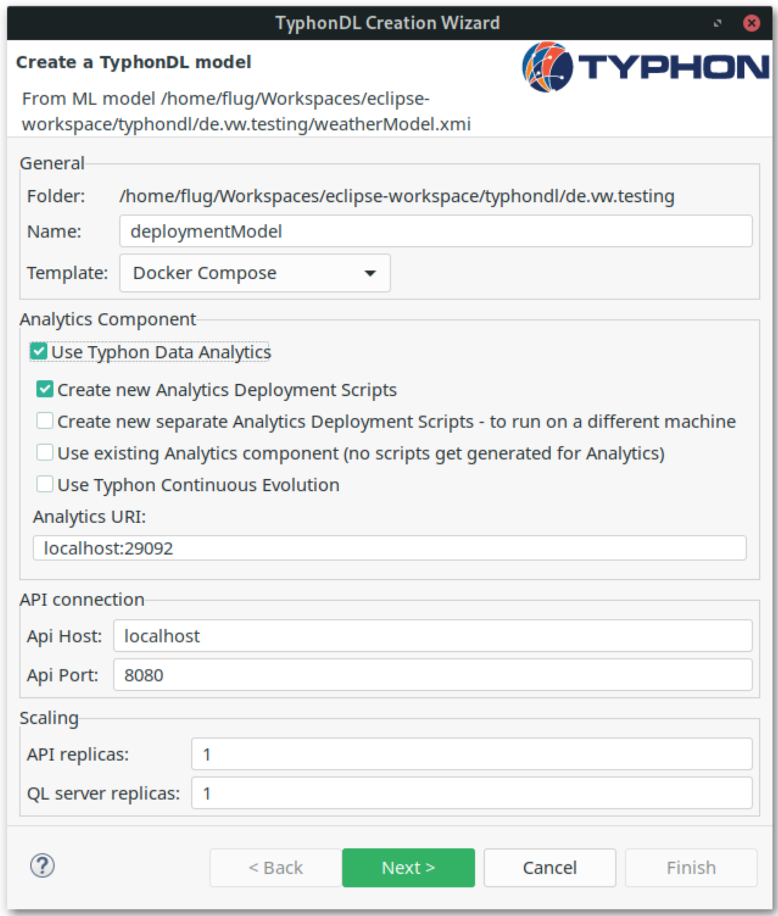


Figure 67: TyphonDL Creation Wizard

To install the Polystore Continuous Evolution component, the user must check the "*Use Typhon Continuous Evolution*" option and finalize the creation.

Once the configuration of the TyphonDL Creation Wizard is completed, the Wizard generates the different Docker deployment scripts.

In the main Docker YML (.yml) file, the user can find the Polystore containers definition. At the end of this file, is located the definition of the **four** containers of the Polystore Continuous Evolution component (see Figure below):

1. **evolution-mongo**: the MongoDB database populated during Step 2, constituting the main input of the web application.
2. **evolution-java**: the Java application which captures, parses the QL queries and which stores the query information in the MongoDB database.

3. **evolution-backend**: the Node.js backend of the web application.
4. **evolution-frontend**: the Angular frontend of the web application; it provides users with visual analytics.

```
evolution-mongo:
  image: mongo:latest
  environment:
    MONGO_INITDB_ROOT_USERNAME: admin
    MONGO_INITDB_ROOT_PASSWORD: admin
    MONGO_INITDB_DATABASE: Analytics
  ports:
    - 27017:27017

evolution-java:
  image: meuriceloup/typhon-evolution-analytics-java:latest
  environment:
    WAKEUP_TIME_MS_FREQUENCY: 10000
    KAFKA_CHANNEL_IP: kafka
    KAFKA_CHANNEL_PORT: 9092
    WEBSERVICE_URL: http://typhon-polystore-service:8080/
    WEBSERVICE_USERNAME: admin
    WEBSERVICE_PASSWORD: admin1@
    ANALYTICS_DB_IP: evolution-mongo
    ANALYTICS_DB_PORT: 27017
    ANALYTICS_DB_USER: admin
    ANALYTICS_DB_PWD: admin
    ANALYTICS_DB_NAME: Analytics
  depends_on:
    - evolution-mongo
    - typhon-polystore-service
    - kafka
evolution-backend:
  image: meuriceloup/typhon-evolution-analytics-backend:latest
  environment:
    BACKEND_DEPLOYMENT_PORT: 3000
    ANALYTICS_DB_URL: mongodb://evolution-mongo:27017/
    ANALYTICS_DB_NAME: Analytics
    ANALYTICS_DB_USER: admin
    ANALYTICS_DB_PWD: admin
    WEBSERVICE_URL: http://typhon-polystore-service:8080/
    WEBSERVICE_USERNAME: admin
    WEBSERVICE_PASSWORD: admin1@
  depends_on:
    - evolution-mongo
  ports:
    - 3000:3000

evolution-frontend:
  image: meuriceloup/typhon-evolution-analytics-client:latest
  environment:
    BACKEND_ENDPOINT: http://evolution-backend:3000
  depends_on:
    - evolution-backend
  ports:
    - 5000:5000
```

Figure 68: Containers of the Polystore Continuous Evolution component

Note that each container has its own configurable parameters:

evolution-mongo:

The user can specify the credentials to connect the MongoDB database;

- MONGO_INITDB_ROOT_USERNAME: it specifies the user login to create.
- MONGO_INITDB_ROOT_PASSWORD:it specifies the user password to create.
- MONGO_INITDB_DATABASE: it specifies the database name to create.

evolution-java:

- WAKEUP_TIME_MS_FREQUENCY: the continuous evolution tool also extracts - at regular time intervals - information about the Typhon Polystore; this variable specifies, in milliseconds, this wakeup interval.
- KAFKA_CHANNEL_IP: it specifies the kafka container ip. This variable is required so that the java application can consume the PostEvent generated by the analytics queue of the WP5 monitoring infrastructure.
- KAFKA_CHANNEL_PORT: it specifies the kafka container port.
- WEBSERVICE_URL: it specifies the Polystore service url.
- WEBSERVICE_USERNAME: it specifies the user login necessary to connect the Polystore service.
- WEBSERVICE_PASSWORD: it specifies the user password necessary to connect the Polystore service.
- ANALYTICS_DB_IP: it specifies the evolution-mongo database ip.
- ANALYTICS_DB_PORT: it specifies the evolution-mongo database port.
- ANALYTICS_DB_USER: it specifies the user login necessary to connect the evolution-mongo database.
- ANALYTICS_DB_PWD: it specifies the user password necessary to connect the evolution-mongo database.
- ANALYTICS_DB_NAME: it specifies the evolution-mongo database name to connect.

evolution-backend:

- BACKEND_DEPLOYMENT_PORT: it specifies the port on which will be deployed the Node.js backend.
- ANALYTICS_DB_URL: it specifies the evolution-mongo database ip.

- ANALYTICS_DB_NAME: it specifies the evolution-mongo database name to connect.
- ANALYTICS_DB_USER: it specifies the user login necessary to connect the evolution-mongo database.
- ANALYTICS_DB_PWD: it specifies the user password necessary to connect the evolution-mongo database.
- WEBSERVICE_URL: it specifies the Polystore service url.
- WEBSERVICE_USERNAME: it specifies the user login necessary to connect the Polystore service.
- WEBSERVICE_PASSWORD: it specifies the user password necessary to connect the Polystore service.

evolution-frontend:

- BACKEND_ENDPOINT: it specifies the Node.js backend url.

Once the Polystore (and the four Polystore Continuous Evolution containers) have been deployed, the user can access the web application at http://localhost:5000/.

6.3.6.4 Data Ingestion (D6.5)

The Data Ingestion tool aims to ease the adoption of the Typhon innovative technologies. It allows one to ingest data from (a set of) pre-existing relational database(s) into a Typhon Polystore.

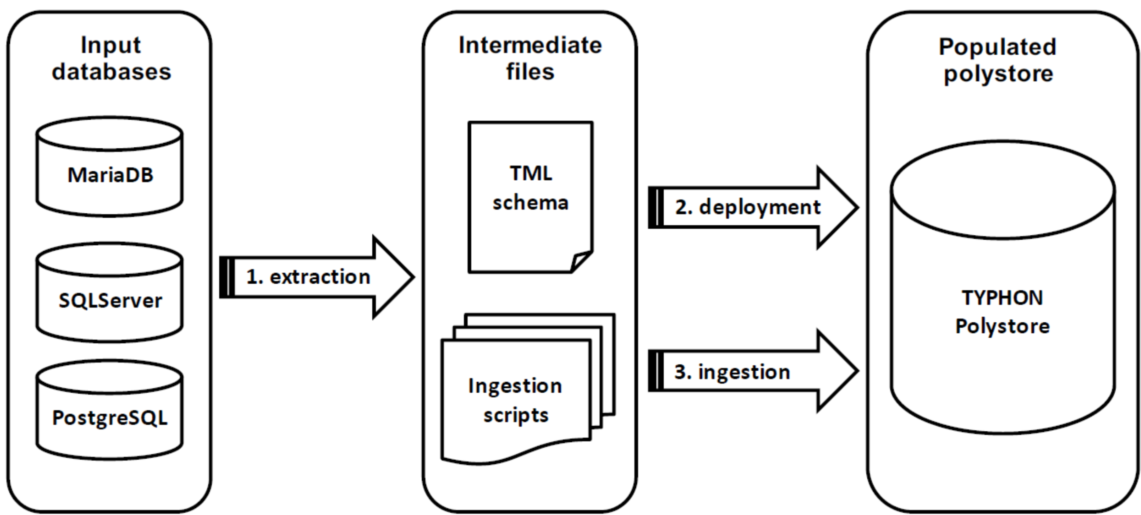


Figure 69: Data ingestion process

Overview

The data ingestion process relies on three steps (as shown in Figure 69):

- **Step 1 - extraction**: The tool first reverse-engineers the relational database schema of each input database, in order to produce a TyphonML schema. It also generated a set of data ingestion scripts (containing prepared QL queries) allowing to transfer the data from the input relational database(s) towards the Polystore, as soon as the latter will be deployed.
- **Step 2 - deployment**: The user takes the automatically extracted TML schema, and uses as starting point to manually deploy a new (empty) Typhon Polystore. This deployment step can be done by means of the tools provided by Work Package 3.
- **Step 3 - ingestion**: The user can then execute the generated data ingestion scripts in order to populate the freshly created Polystore with the data extracted from the input relational databases.

Installation (building with maven)

cd data_ingestion

mvn clean install

Step 1. Extraction

The extraction phase mainly consists in extracting the data structures (schemas) of the relational databases given as input, and to abstract those data structures into a TyphonML schema. This schema abstraction process is achieved according to the following abstraction rules.

- each table including at least one non-foreign key column becomes a conceptual entity;
- each non-foreign key column (except auto-increment identifier) of a table becomes an attribute of the corresponding entity;
- each foreign-key becomes a one-to-many relationship;
- each table that only consists in two foreign keys referencing respectively table t1 and table t2, becomes a many-to-many relationship between the corresponding entities;
- all relational schema elements including identifiers(except auto-increment identifier) and indexes are also translated into corresponding TyphonML schema constructs.

As an example, let us consider the input relational schema of Figure below. This schema includes 4 tables: *Customer, Orders, Product and Details*.

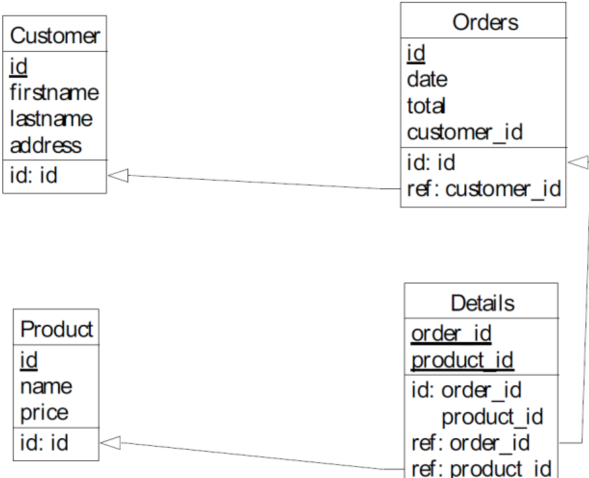


Figure 70: Example input relational schema with four tables

The input schema would be abstracted as three entities. Tables *Customer*, *Orders* and *Product* have been translated into corresponding entities.

```
entity Orders{
    "id": string[32]
    "date" : date
    "total" : float
    "Customer" -> Customer[1]
    "Product" -> Product[0..*]
}

entity Customer{
    "id" : string[32]
    "firstname" : string[32]
    "lastname" : string[32]
    "address" : string[32]
}

entity Product{
        "id": string[32]
        "name": string[32]
        "price": float
        "Orders" -> Orders[0..*]
}

relationaldb RelationalDatabase{
    tables{

        table {
            "Orders" : "Orders"
            idSpec ("Orders.id")
        }

        table {
            "Customer" : "Customer"
            idSpec ("Customer.id")
        }

        table {
                "Product" : "Product"
                idSpec ("Product.id")
        }
    }
}
```

Figure 71: Input schema extracted as three entities

Table *Details* has been abstracted as a many-to-many relationship. The foreign keys in table *Orders* referencing table *Customer* has been abstracted as a one-to-many relationship between the corresponding entities. This conceptual abstraction will lead to the production of the TyphonML schema given below. This schema can be used as starting point of the deployment step (Step 2).

In order to connect to the input databases, the data ingestion tool requires the user to specify the required URL and credentials. This information must be contained in a configuration file ("*extract.properties*").

In this file, one can specify the connection information of one or several relational databases. The following extraction parameters can be specified, for each input relational database:

- URL : the JDBC URL necessary to connect to the database.
- DRIVER : the JDBC driver necessary to connect to the database.
- USER : a user login with reading permissions.
- PASSWORD : the user password.
- SCHEMA : the name of the input database schema name to connect.

In the case of several input relational databases, the user can use a suffix for each of the above parameters.

Following the pattern PARAMETER#DB, i.e URL2 will be the URL of the second database, URL3 will be the URL of the third database, etc.

In addition, the user can specify two other configuration parameters concerning the data ingestion scripts to generate:

- MAX_QL_QUERIES_PER_FILE: a set of prepared QL queries will be generated at the end of the Extraction Step. Executing these QL queries (see Step 3. Ingestion) will allow to transfer the data from the input relational database(s) towards the Polystore, as soon as the latter will be deployed. Parameter MAX_QL_QUERIES_PER_FILE allows the user to specify the maximal number of QL queries per ingestion file.
- PREPARED_STATEMENTS_BOUND_ROWS: the user can specify the maximal number of rows each prepared query to insert.

The Figure below provides an example of configuration file specifying the credentials of two input relational databases to connect. A configuration file example is copied in the target directory generated during the install phase, and can be edited by the user.

```
1 URL=jdbc:mysql://localhost:3306/sample
2 DRIVER=org.mariadb.jdbc.Driver
3 USER=root
4 PASSWORD=admin
5 SCHEMA=sample
6
7
8 URL2=jdbc:mysql://localhost:3306/sample2
9 DRIVER2=org.mariadb.jdbc.Driver
10 USER2=root
11 PASSWORD2=admin
12 SCHEMA2=sample2
13
14
15 MAX_QL_QUERIES_PER_FILE=1
16 PREPARED_STATEMENTS_BOUND_ROWS=2
```

To execute the Extraction Step, the user must use the files generated during the install phase (in the target directory):

WINDOWS: data_ingestion.bat -extract extract.properties output

LINUX: bash data_ingestion.sh -extract extract.properties output

where extract.properties is the configuration file containing the input relational databases credentials and output the directory in which the TML schema and the data ingestion scripts will be generated at the end of the extraction.

Step 2. Deployment

As result of the Extraction step, a TyphonML schema (*output/schema.tml*) and a set of data ingestion scripts (*output/data/*.tql*) are generated in the output directory:

The Polystore deployment step simply consists in deploying a new Polystore by using the TyphonDL tools (WP3). This process takes as input the TyphonML schema automatically extracted at Step 1 (*schema.tml*). We refer to the WP3 deliverables for more details about the deployment process and supporting tools.

Step 3. Ingestion

Once the new target Polystore has been created and deployed, the last step consists in executing the data ingestion scripts (.tql) generated at Step 1.

The execution of those scripts also requires to specify the credentials necessary to connect the Polystore service allowing to execute the prepared QL queries contained within the data ingestion scripts. This information must be defined in the *inject.properties* file. Figure below gives an example of structure for this configuration file.

```
1 POLYSTORE_SERVICE_URL=http://localhost:8080
2 POLYSTORE_SERVICE_USERNAME=admin
3 POLYSTORE_SERVICE_PASSWORD=admin1@
```

The following parameters are required:

- POLYSTORE_SERVICE_URL: the url necessary to connect the Polystore service.
- POLYSTORE_SERVICE_USERNAME: the user login necessary to connect the Polystore service.
- POLYSTORE_SERVICE_PASSWORD: the user password necessary to connect the Polystore service.

To execute the Ingestion Step, the user must use the files generated during the installation phase (in the target directory):

WINDOWS: data_ingestion.bat -inject inject.properties output/data

LINUX: bash data_ingestion.sh -inject inject.properties output/data

where inject.properties is the configuration file containing the Polystore service credentials and output/data the directory in which the data ingestion scripts have been generated at the end of the Extraction Step. Once the data ingestion is completed, the Polystore is populated and ready to use.

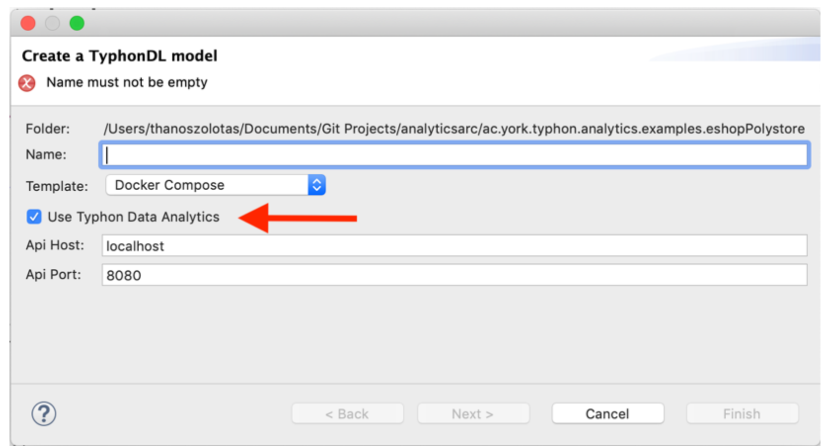
6.3.7 Analytics

6.3.7.1 Prerequisites

1) This guide assumes that you have already installed all the necessary tools to create and run a Polystore (e.g., TyphonML, TyphonDL, etc.). That is, you have done the steps described in Section 6.1.

2) You need to make sure that you have those updated (from their respective Eclipse update sites and by doing a docker-compose pull) to their latest version. As a rule of thumb if you haven't updated after 10th of May 2020, you need to update to get the latest working version.

3) Start by creating the Polystore as described in the previous sections. Make sure that in the appropriate step of the DL wizard you have checked the "Use Typhon Data Analytics" option, as shown in the image below.



4) Run the Polystore.

Warning: The guide assumes that the Polystore is deployed using Docker Compose. The Polystore and the analytics component have not yet been fully tested using a Kubernetes deployment.

6.3.7.2 Work with the Analytics Component

1) Download (if you don't have already done) Eclipse with Epsilon support from here: https://www.eclipse.org/epsilon/download/

2) Download the analytics zip file from here: https://drive.google.com/file/d/12cZY1Gxt-Qv6wCwu6Oa4jmd-IRAPH43R/view?usp=sharing

3) Unzip and import the two projects (named ac.york.typhon.analytics.examples.howto and ac.york.typhon.analytics) into Eclipse by going to File → Import → Existing Projects into Workspace

4) The ac.york.typhon.analytics.examples.howto project is an example project that has a simple analytics scenario (TestAnalyticScenario class) and a runner (AnalyticsRunner class) in it.

   a. If you navigate to the pom.xml file you will see that it has a dependency to the ac.york.typhon.analytics project.

   b. You can either use this project to test analytic scenarios or you can create another Maven project that has the same dependency (to the ac.york.typhon.analytics)

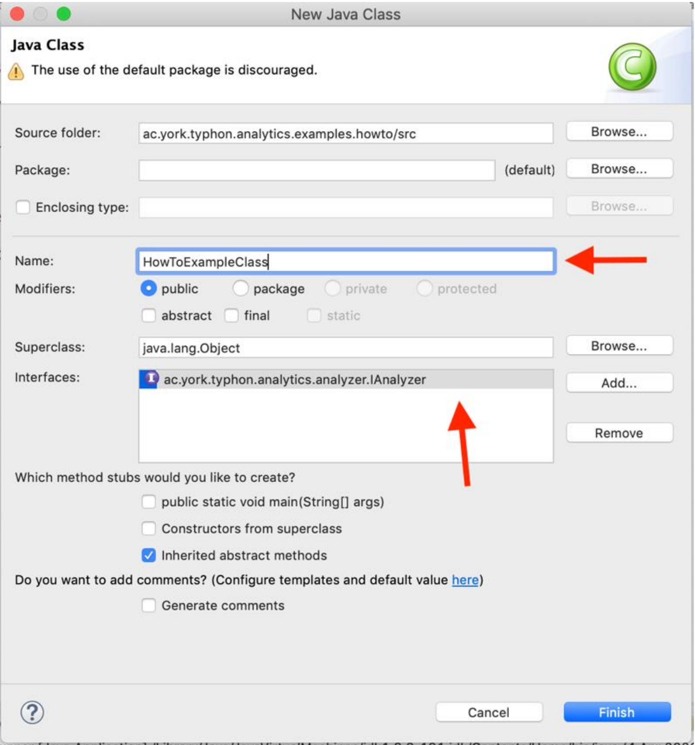
5) The ac.york.typhon.analytics project includes the analytics infrastructure. You don't need to do anything with it. It should just be included as a dependency when you create a new Analytics project, as described above.

Please note: The generated docker compose defines port 29092 for external (outside Docker) access to the Kafka queue and port 9092 for internal (inside Docker) access. As this guide describes how to write analytics in your local IDE, the configuration is set to access port 29092. If you want to export the jar and run it inside Docker, then you need to open the "resources/typhonAnalyticsConfig.properties" file and set the port in line 12 to 9092.

6.3.7.3 Write Analytics

1) Create a new maven project that has a dependency on the ac.york.typhon.analytics project. Of course, you can instead use the example project (ac.york.typhon.analytics.examples.howto) you have imported.

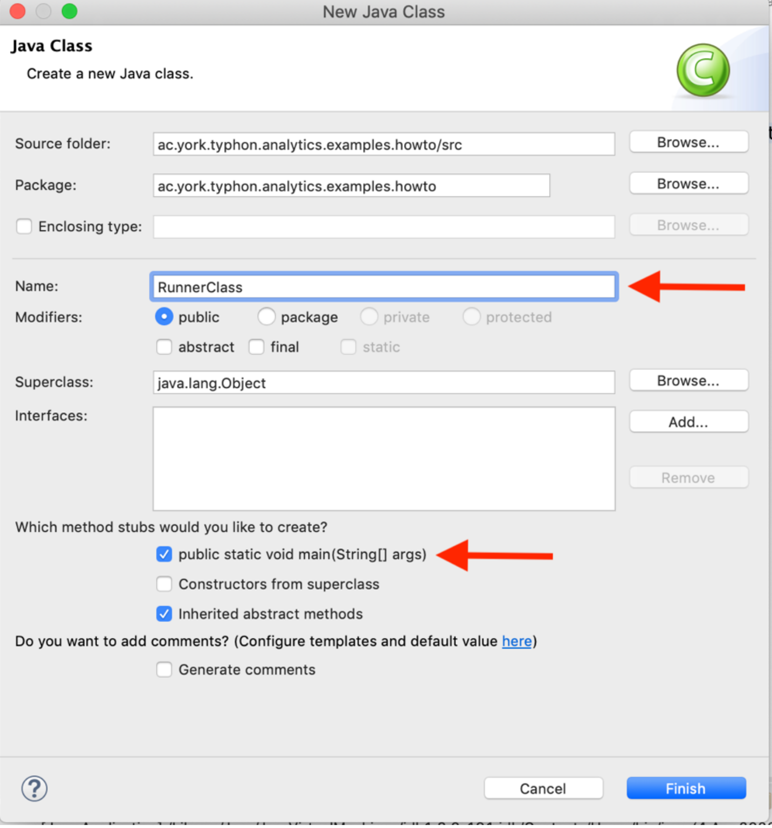
2) Create a new class (right click on the src folder → New → Class) that implements the "IAnalyzer" interface. If you use the example project you will see that such a class already exists (named "TestAnalyticScenario").



3) The new class will include the method you need to implement called "analyze (DataStream<Event> eventsStream). The eventsStream parameter are the PostEvent objects arriving to the POST queue of the analytics architecture. You need to write flink code to consume them and produce analytics of interest (more on this later).

```
 1⊕import org.apache.flink.api.common.functions.FilterFunction;
 9
10 public class HowToExampleClassWithSomeLogic implements IAnalyzer {
11
12⊖    @Override
13     public void analyze(DataStream<Event> eventsStream) throws Exception {
14
15
16     }
17
18 }
19
```

4) In order to run the analytics code you need to create a main class which calls the classes that include analytics. Create a new class (right click on the src folder → New → Class), give it a name (e.g., RunnerClass) and include a main method in it (If you use the example project this is the AnalyticsRunner class.). You need to create such a class only once for each analytic scenario.



a. You need to make a call to the class that includes an analytics scenario using the ChannelBuilder.build(…) method as shown in the code below. This method takes as a first parameter the name of the class that includes analytics code (e.g., HowToExampleClass) and as a second parameter the name of the Kafka topic from which events should consumed. This should be AnalyticTopicType.POST always when writing analytics for Typhon Post Events.

b. Remember to declare that your main class throws an exception (or surround the ChannelBuilder methods with try…catch statements)

```
1  import ac.york.typhon.analytics.channel.ChannelBuilder;
2  import ac.york.typhon.analytics.commons.enums.AnalyticTopicType;
3
4  public class RunnerClass {
5
6      public static void main(String[] args) throws Exception {
7          // Caller for the TestAnalyticScenario
8          ChannelBuilder.build(new HowToExampleClass(), AnalyticTopicType.POST);
9
10         // If you need to run other analytics scenarios then copy-paste the above line and replace the
11         // "TestAnalyticScenario()" with the name of the class the contains the new analytic scenario
12     }
13 }
14
```

5) Run this main method as a Java Application. Your analytics code inside the HowToExampleClass will start consuming PostEvents as these arrive in the Polystore. As the analyze method’s body is empty, this will do nothing. More on how to write analytics code is described in the next section.

IMPORTANT!!! Post events in Typhon are created every time a TyphonQL query is executed. Thus, your code will produce results, if and only if you start using the Polystore and execute some TyphonQL queries.

6.3.7.4 Writing Analytics Code with FLink

Flink is a distributed execution framework. By using its **operators** Flink can easily distribute you code without requiring user's input/configuration. The goal of this guide is not to train people on writing Flink code. There are plenty of resources on this online. The basic idea is that Flink works with streams (in the context of the analytics component). Steams as the name suggests, provide continuous real-time input to your programs. In the analytics component, the stream of events is the "eventsStream" parameter is the analyse method. This is configured to automatically consume all the events coming to the POST queue.

To consume streams using Flink, one should use Flink Operators. A comprehensive list of all the available Flink operators is available here: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/

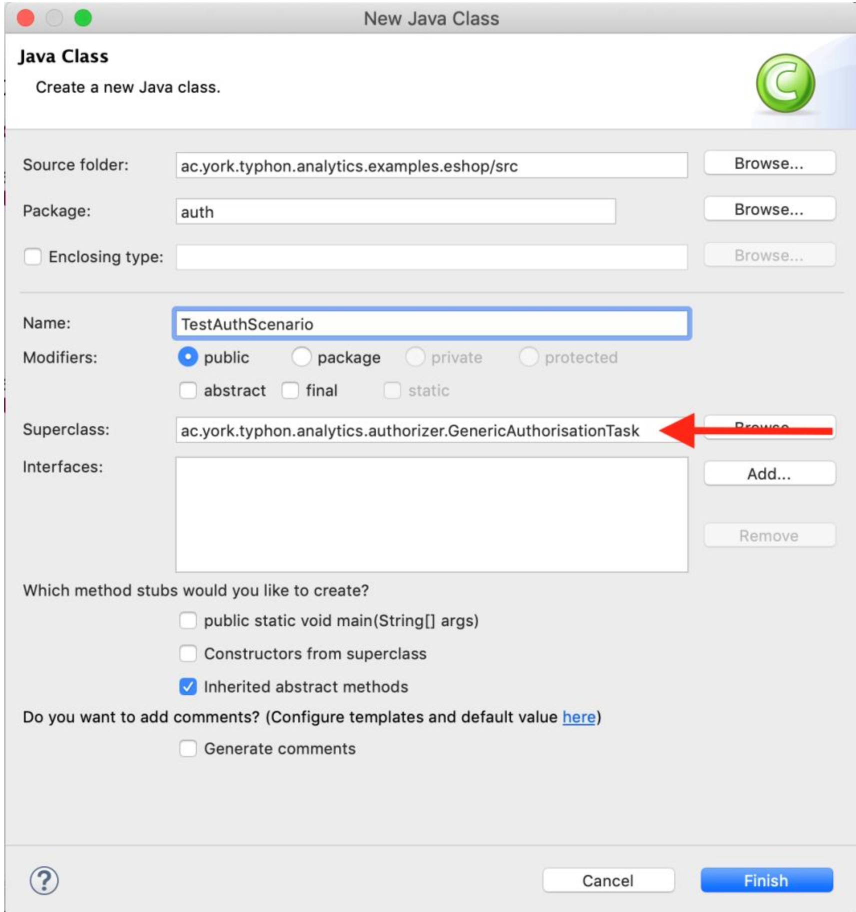
This should be the starting point of anyone trying to use Flink as they contain a brief description and an example of how to make them work. You will find yourself mostly having to use the filter and map function (the first filters events based on a condition, the second is used to transform objects to other forms). Experiment with these 2 first and then you can proceed to more complicated operators. Below is a simple example that consumes Typhon PostEvents and produces at the end a list of the credit card numbers that have expired. You can find the code into the "HowToExampleClassWithSomeLogic" class of the "ac.york.typhon.analytics.examples.howto" project. The example is based on an ECommerce Polystore example.

```
1⊕import org.apache.flink.api.common.functions.FilterFunction;
9
10 public class HowToExampleClassWithSomeLogic implements IAnalyzer {
11
12⊖    @Override
13    public void analyze(DataStream<Event> eventsStream) throws Exception {
14
15⊖        eventsStream.filter(new FilterFunction<Event>() {
16
17⊖            @Override
18            public boolean filter(Event arg0) throws Exception {
19                // Cast Event to PostEvent.
20                PostEvent postEvent = (PostEvent) arg0;
21                // Filter events and get only those that are insert statements of CreditCard entities in the eshop example.
22                if (postEvent.getQuery().contains("insert CreditCard")) {
23                    return true;
24                }
25                return false;
26            }
27        })
28        // Create a Tuple credit card number and credit card expiry year
29⊖        .map(new MapFunction<Event, Tuple2<String,String>>() {
30
31⊖            @Override
32            public Tuple2<String,String> map(Event arg0) throws Exception {
33                PostEvent postEvent = (PostEvent) arg0;
34                // Get the number of the card from the query
35                String number = postEvent.getQuery().split("number: \"")[1].split("\", expiryDate:")[0];
36                // Get the expiry date and then the year from the query
37                String expiryDate = postEvent.getQuery().split("expiryDate: \"")[1].split("\"}")[0];
38                String expiryYear = expiryDate.substring(expiryDate.length()-4, expiryDate.length());
39                // Create the tuple and pass it on
40                return new Tuple2(number, expiryYear);
41            }
42        })
43        // Filter those that has expired since the previous year
44⊖        .filter(new FilterFunction<Tuple2<String, String>>() {
45
46⊖            @Override
47            public boolean filter(Tuple2<String, String> arg0) throws Exception {
48                // Get the year from the tuple (f0 is the first entry, f1 is the second entry in a tuple, etc.
49                int expiryYear = Integer.parseInt(arg0.f1);
50                if (expiryYear <= 2019) {
51                    return true;
52                }
53                return false;
54            }
55
56        })
57        // Print the tuples number and expiry year. That's why we carried the number as well in the tuple.
58        .print();
59    }
60
61 }
62
```

6.3.7.5 Write Authorisation Tasks

1) Create a new maven project that has a dependency on the ac.york.typhon.analytics project in the same way created in step 3.1.

2) Create a new class (right click on the src folder → New → Class) that extends the “GenericAuthorisationTask” abstract class.



3) The new class will include two methods you need to implement called "checkCondition (Event event)" and "shouldIReject(Event event)." In the first you need to provide the logic that decides if this authorisation task is responsible for checking the event passed as parameter. For example, if this task is responsible for checking the validity of the credit card used to place an order then the logic included here should filter and accept only such "insert order" events. The second, should include the rejection logic. For example, include the logic that rejects order events in which credit cards used have expired.

```java
package auth;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.Date;

import ac.york.typhon.analytics.authorizer.GenericAuthorisationTask;
import ac.york.typhon.analytics.commons.datatypes.events.Event;

public class CCDateAuthTask extends GenericAuthorisationTask {

    @Override
    public boolean checkCondition(Event event) {
        if (event.getQuery().toLowerCase().contains("order ")) {
            return true;
        } else {
            return false;
        }
    }

    @Override
    public boolean shouldIReject(Event event) {
        String query = event.getQuery();
        // this is specific to orders that only have new creditcards inserted
        String creditcard = query;
        creditcard = creditcard.substring(creditcard.indexOf("CreditCard") + 12);
        creditcard = creditcard.substring(0, creditcard.indexOf("}"));
        String date = creditcard;
        date = date.substring(date.indexOf("expiryDate:") + 13);
        date = date.substring(0, date.indexOf("\""));
        SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
        // Parsing the given String to Date object
        Date d = null;
        try {
            d = formatter.parse(date);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return d.getTime() <= LocalDateTime.now().toEpochSecond(ZoneOffset.ofHours(0)) * 1000;
    }
}
```

4) Following steps 2 and 3 you can define more authorisation tasks that should be included in the authorisation chain.

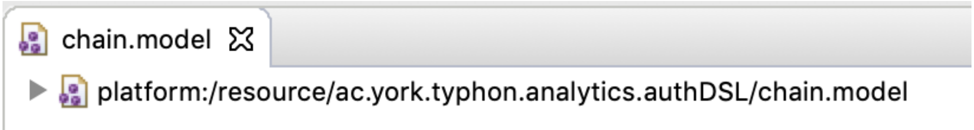
5) In order to run the authorisation chain that includes all the tasks, you need to create a main runner class. We provide a tool that generates this runner class automatically.

    a. Import the 3rd Eclipse project (ac.york.typhon.analytics.authDSL) into your Eclipse IDE.

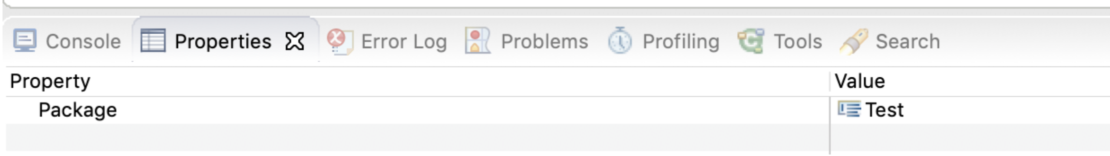
    b. Make sure “authDSL.ecore” has been registered by right-clicking on it and selecting “Registering EPackage”.

    c. Open the “chain.model” file. (you can preferably right-click on it, select “Open With…” and select “Exeed Editor”. If “Exeed Editor” is not available select “Other…” and search for “Exeed Editor” in the list.

    d. Open the top element by clicking the grey arrow on the left



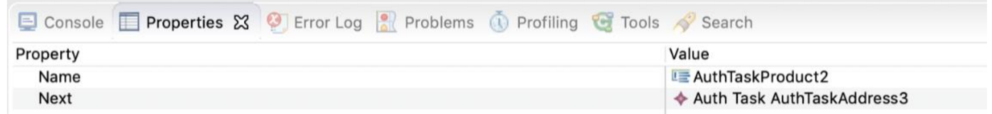
e. Click on the model element of type “Auth Chain” and navigate to the “proeprties” eclipse view at the bottom. If the view is not available you need to click on Window->Show View->Properties. Provide the name of the package in the Maven project that hosts the authorisation tasks you created before



f. For each authorisation task you created you need to repeat the following steps:

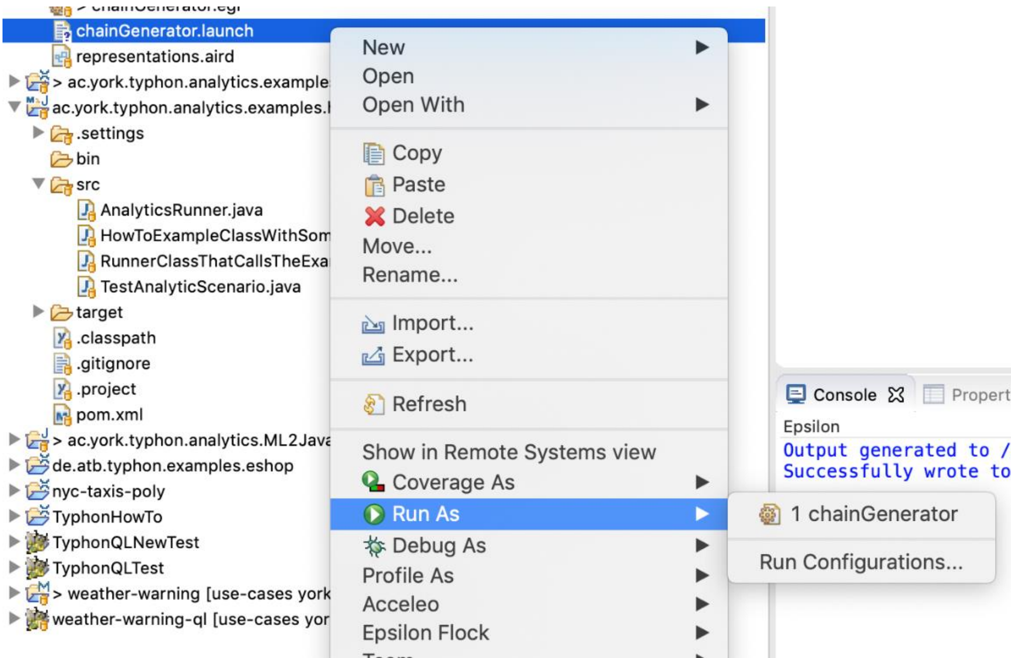
   i. Right-click on the Auth Chain model element and select “New Child”-> Tasks Auth Task. This will create a new Auth Task model element

   ii. Click on the new element created and navigate to the properties view. Provide the name of the auth task. This must be the name of the class that implements the logic for this task (i.e., the name you used in step 4.2)



   iii. Repeat the same process for all the auth tasks you have in the authorisation chain. When finished, re-visit each authorisation task and select its following task in the chain. This is done by selecting the name of the next task in the list of the “Next” property. For example, if your chain has 3 tasks named Task A, Task B and Task C then highlight Task A in your model and select from the list of the “Next” property, the model element named “Task B”. For Task B, you need to set the “Next” property to “Task C”, etc. For the last task in the chain (i.e., Task C in this example”) you leave the field empty.

g. When you are done creating the authorisation chain model right click on the file “chainGenerator.launch” and select “Run As…” -> ”1 chainGenerator”. This will generate the main class (in the “output” folder) that can be used as a runner for run the authorisation chain.



6) Move this class into your analytics/authorisation project in the same package where your authorisation tasks reside. Run this main method as a Java Application.

IMPORTANT!!!

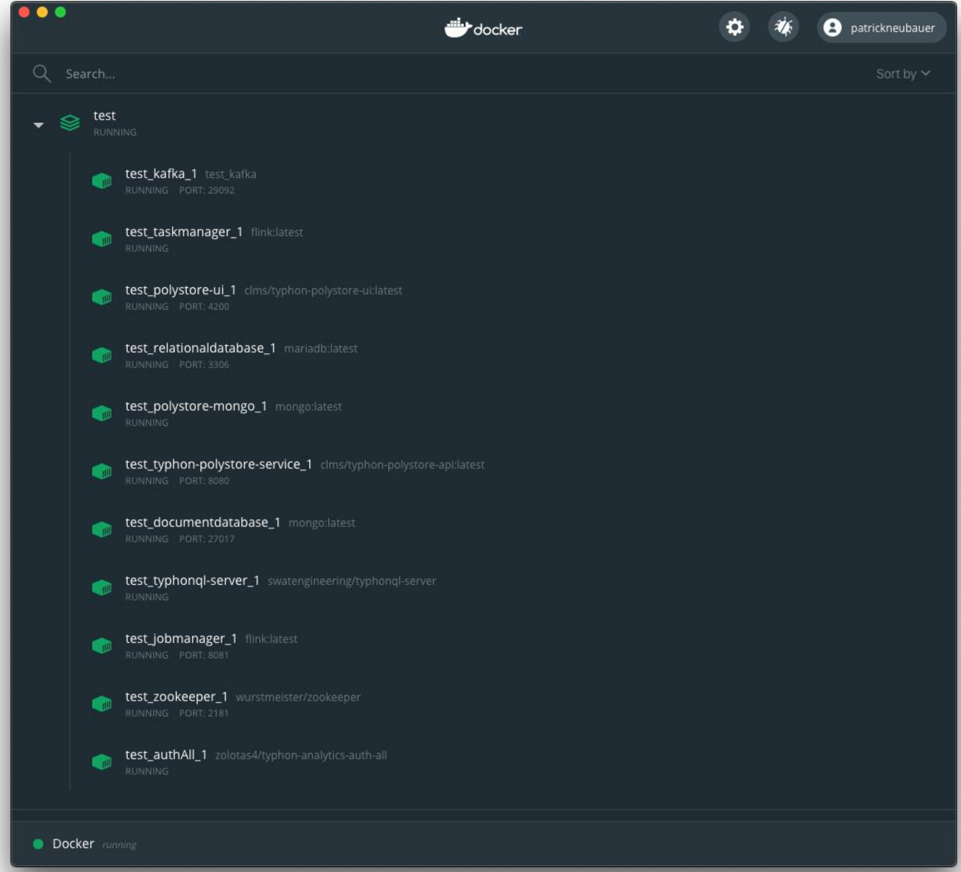
1) Events in Typhon are created every time a TyphonQL query is executed. Thus, your code will produce results, if and only if you start using the Polystore and execute some TyphonQL queries.

2) The current Polystore implementation provides a default authoriser that authorises all the queries for execution. This means that if you need to include authorisation tasks as part of one of your use case scenarios then you should **not** run the container that authorises all the queries. This can be done by removing (or commenting out) the following service from your docker-compose file

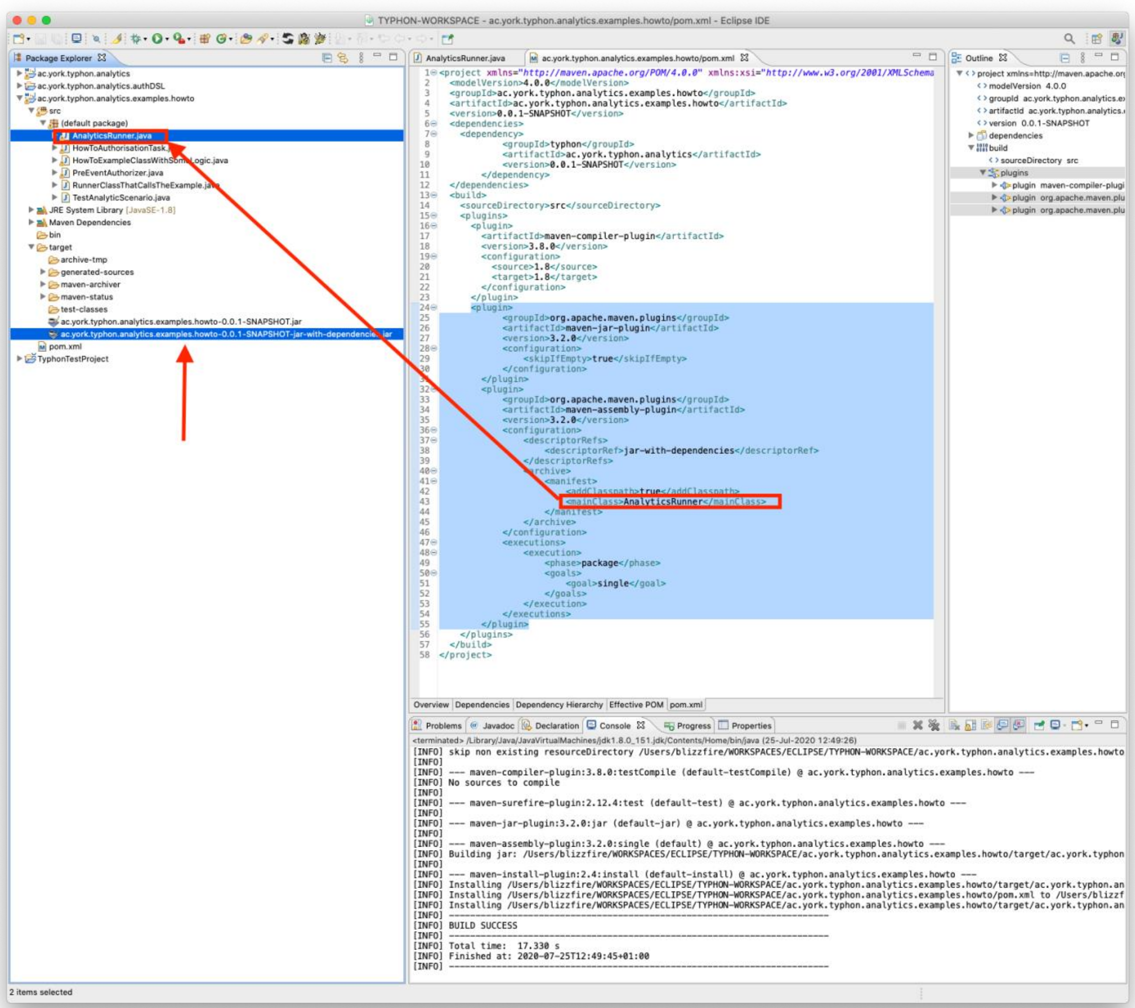
```
47  authAll:
48    image: zolotas4/typhon-analytics-auth-all
49
```

6.3.7.6 Execute Analytics Outside Eclipse

This section describes the execution of Typhon Analytics outside your Eclipse IDE, directly to a Flink cluster deployed as part of the Typhon Polystore. This assumes required components are deployed as described in the Typhon Quick Start Guide which results in a series of Docker containers as depicted in the Docker Dashboard shown below:

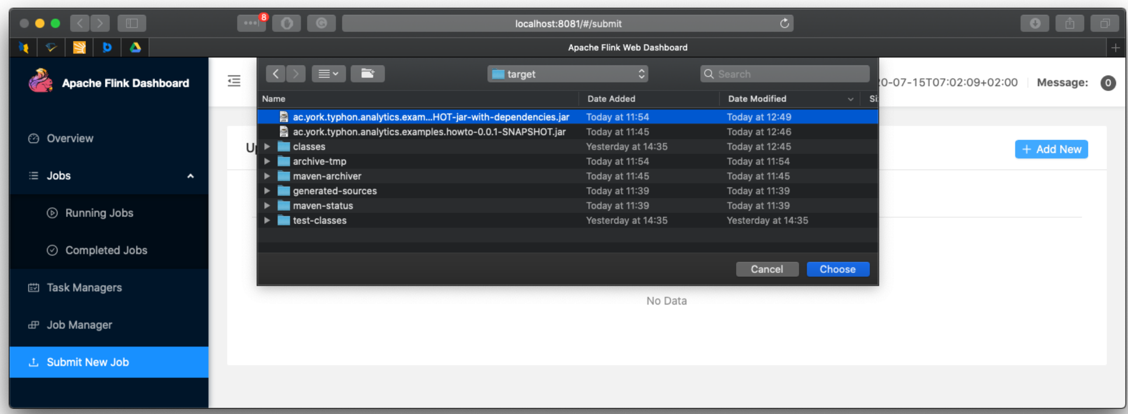


First, to package your analytics code, add lines 24 to 55 in the screenshot below (or as defined in this pastebin) to the pom.xml file of your analytics project (i.e. named ac.york.typhon.analytics.examples.howto in the screenshot below) and adapt the mainClass XML element content in line 43 accordingly. Note that if the AnalyticsRunner class would be located in a package, such as my.package, then the mainClass element content would have to read my.package.AnalyticsRunner.

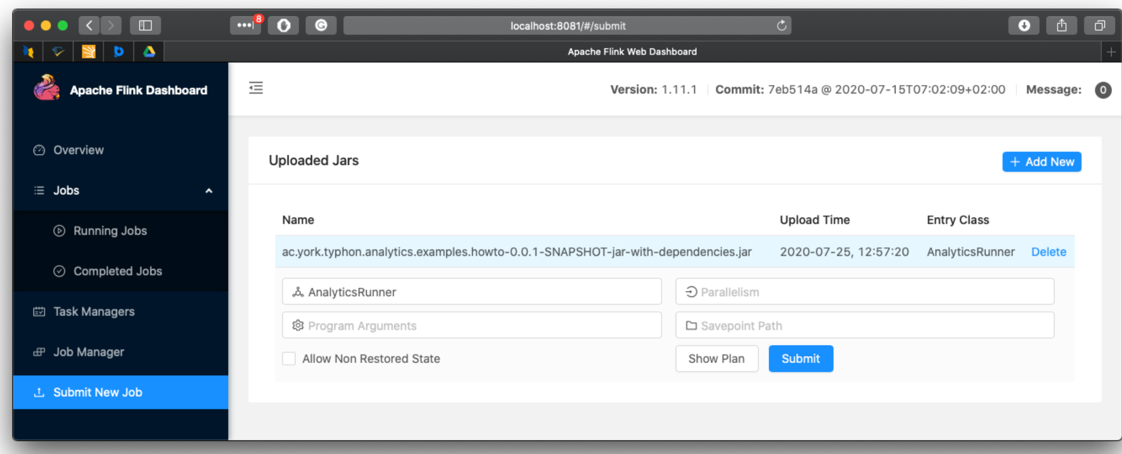


Second, either issue the command-line command mvn clean install from the directory of your analytics project or right-click on the analytics project in Eclipse and select Run As > Maven install. This will produce content in the target directory of your analytics project and in particular a file ending in -jar-with-dependencies.jar (i.e. ac.york.typhon.analytics.examples.howto-0.0.1-SNAPSHOT-jar-with-dependencies.jar in the above screenshot).

Third, use your browser to access the job submission page on the web-based UI of Apache Flink at http://localhost:8081/#/submit. Next, hit the “+ Add New” button and choose the JAR file produced in the previous step:



Forth, as soon as the previous step completes (i.e. uploading the JAR of your analytics project), the Apache Flink web UI will display the uploaded file accordingly. Next, click on the name of the uploaded file to (optionally) setup options and launch the program. The latter is achieved by hitting the "Submit" button in the displayed web UI:



6.4 KNOWN ISSUES

The following are issues that are known (with workarounds where applicable) which the respective component teams are actively working to fix:

- Docker needs more memory than 2GB.
- Polystore UI model upload page sometimes needs refresh after uploading consecutive models
- MariaDB credentials are not always set up correctly. This is a docker volume issue, and to rectify you should remove the MariaDB volume via docker-compose rm –v (**This will cause loss of data)**

7. CONCLUSIONS

For the purposes of the Integrated Platform work Package (WP7), a fully working software prototype was developed, tested and finalized. For this complete end-to-end implementation all technical contributions delivered in previous work packages (2-6) were integrated into a single platform. The Polystore Platform enables users to store and retrieve data from Hybrid Polystores that have been entirely designed and deployed using tools developed for the Typhon project.

For the final version of the Integrated Platform, capabilities were to include the ability to continuously evolve multiple segments of the schema and the relevant queries used. Additional external tools were also supplied to compliment data ingestion operations. Furthermore, the Analytics component was fully integrated, allowing users to provide custom built authorization mechanisms and analysis of queries. The Continuous Evolution tool also makes use of this function provide robust performance statistics and recommendations for potential improvements on the schema used.

During the development process, a considerable number of technical challenges, regarding the platform's performance, stability, extensibility, interoperability, and to a degree, scalability, were tackled. Communication between the platform's core components has been tweaked and optimized through a long series of implementation and evaluation cycles. The implementations of the components' public interfaces were finalized, documented and developed.

The Continuous Integration and Distribution architecture was also evolved to accommodate the development of existing and new components and tools. Changes were made across multiple CI/CD component configurations to further optimize and streamline the development and deployment process.

The RESTful API was modified to accommodate all changes in the integrated components, while new services reflecting new features were made available. Additional integration of components not previously completed were finalized in this version.

The web-based user interface that operates on top of the aforementioned RESTful API was modified to be more lightweight, while also adding the new functionalities developed for the API.

The installation of the overall Polystore and its components were simplified. Except for optimizing this process, all steps were documented into an extensive installation and usage guide.

Finally, the table below presents the status of the requirements fulfilled as requested by our Use Case partners.

Table 1 Use Case requirements

Req.-ID	Priority	Description	Status
87	SHALL	The Polystore database shall offer a REST API for executing queries and storing/retrieving data	Implemented
88	SHALL	The Polystore database shall provide a command line tool for configuration and management of the database instance	Implemented
89	SHOULD	A web-based user interface should be provided for updating and accessing data	Implemented

Req.-ID	Priority	Description	Status
90	SHALL	The Polystore interfaces shall be customisable, i.e., the possibility to generate new APIs for added user functions shall be provided	Implemented
91	SHALL	The generated API shall be able to access data stored in a relational database	Implemented
92	SHALL	The generated API shall be able to receive and send data remotely	Implemented
93	SHALL	The generated API shall be able to access data stored in an array database	Implemented
94	SHALL	The generated API shall be able to access data stored in a text store	Implemented
95	SHALL	The generated API shall provide HDFS file system access	Implemented
96	SHALL	The generated API shall provide Hive data access	Implemented
97	MAY	The Polystore database may provide an API to plug processing components that are used to transform data when it's ingested or retrieved	Implemented