Department of Informatics
Der Technischen Universität München

Bachelor's Thesis Information Systems

REST-based Data Integration Services for Software Engineering Domain

Fridolin Jakob Koch

DEPARTMENT OF INFORMATICS

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis Information Systems

REST-BASIERENDER DATENINTEGRATIONSDIENST FÜR DIE DOMÄNE DES SOFTWARE ENGINEERING

REST-BASED DATA INTEGRATION SERVICES FOR SOFTWARE ENGINEERING DOMAIN

Erstbetreuer:	Prof. Dr. rer.nat. Florian Matthes
Zweitbetreuer:	M.Sc Klym Shumaiev
Tag der Einreichung:	15.07.2016

Erklärung

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, den 15.Juli 2016

Fridolin Jakob Koch

Abstract

Data integration is a widespread problem describing the process of synchronizing data from different systems with a target system. Within the software engineering domain architectural data is often spread across different systems. Combining this data into a software architecture knowledge management (SAKM) systems to capture the reasoning behind architectural decisions is an important task. Even if the task of data integration is a very common problem, it is still a main reason for the inhibited adoption of SAKM systems within the software engineering domain. In order to simplify the data integration process for both developers and users a prototypical JSON-Schema based data integration tool for the software engineering domain is introduced in this thesis. Central aspect of the work is to provide a lightweight and easy to extend data integration tool using *node.js* as underlying platform. The application consists of two standalone applications, a server which handles the transformation of data and a state of the art web application which provides a user interface for controlling the server. This thesis describes the uses cases and architectural requirements for such systems. A small evaluation was conducted towards the end to verify the usefulness and extensibility of the tools. After positive feedback was collected in aftermath of the evaluation, future work may focus on further extending the prototypical applications with features satisfying demands of the software engineering domain.

Keywords: Data Integration, Extract-Transform-Load (ETL), Software Architecture Knowledge Management (SAKM), Prototype Implementation, JSON-Schema

Contents

List of Abbreviations

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
DSL	Domain-specific language
ETL	Extract Transform Load
GUI	Graphical User Interface
MVC	Model-View-Controller
MVVM	Model-view-viewmodel
RDBMS	Relational database management system
RDF	Resource Description Framework
SAKM	Software architecture knowledge management
SPA	Single-page application
URI	Uniform Resource Identifier

List of Figures

List of Tables

List of Listings

1. Introduction

1.1. Motivation

The steadily increasing complexity of today's enterprise software systems requires a sophisticated methodology for managing the architecture of such systems. In most modern software projects a wide range of tools is used for managing architectural artifacts [1]. These tools could be collaborative issue tracking applications like *Atlassian JIRA* or just spreadsheet application like *Microsoft Excel*. Software design can be described as a *wicked problem*, therefore it is reasonable to capture the process which lead to certain design decisions. Large software projects often involve more than one person occupying the role of the software architect (SA), accordingly it is important that all involved SAs have the same *architectural knowledge* to take decisions [2]. *Software architecture knowledge management* (SAKM) systems aim to provide a solution for this particular problem domain. As in the most environments project data is spread among different tools, the data has to be integrated into the SAKM system, this process is still a challenging task and often inhibts the adoption of SAKM systems[1][3]. Although there is a wide range of commercial and open-source data integration applications, the most of them are either tailored to specific use-cases or very generic and require a lot effort to integrate into existing system environments. The goal of this work is to design and implement a framework that supports software engineers to build data integration services.

1.2. Research questions

To achieve the goal of this thesis the following three research questions where defined:

RQ1: What are the use cases of data integration services?

RQ2: What are the features of the existing data integration service providers?

RQ3: How to design a framework for the data integration services in software engineering domain?

1.3. Thesis Structure

This thesis focuses on the design, implementation and evaluation of the application prototype called *SyncPipes*. The strucute of the thesis aligns with the previously defined research questions. Chapter 2 focuses therefore on answering **RQ1** by deriving use-cases from relevant literature. To answer **RQ2** a short analysis of similar application is conducted in Chapter 3. Rather then analyzing all tools in detail it was the goal to find similarities in differences in the tools to have an overview how the problem of data integration commonly solved. The following Chapters focus on answering **RQ3**. In chapter 4 the architecture of the application is defined based on the use cases elicited in chapter 2. Following the defined architecture, Chapter 5 describes how the application was implemented in detail. The focus there is to describe the process of data transformation. Chapter 6 describes the method and results of the evaluation of the implement prototype. The thesis concludes with a conclusion

2. Use Cases

For the design and implementation of the application it was necessary to derive use cases. These uses cases where derived from common problems in software architecture knowledge management. The architectural data of a project is often spread across different tools or systems, integrating this data into a SAKM system and vice versa, is still a challenging task [1]. Often this circumstance inhibits the adoptions of SAKM systems [3]. Based on the problem two roles where identified using such a data integration system.

The role of the **Developer** (DEV) represents software developers who extend the application with custom services implementing predefined interfaces to support compatibility with other systems. Actors having the role of a **Software Architect** (SA) use the application to define, execute and monitor data integration pipelines. SA's have good knowledge about the source and target system's domain model but do not necessarily have deeper knowledge about the application. Derived from these preliminary definitions, for both the **Developer** and the **Software Architect**, we defined the following use cases.

A DEV who wants to extend the application with a custom service supporting data integration from or into an external system **(UC 1)** needs to be provided with guidance in the form of technical documentation as well as with an application architecture that supports a simple way of creating new services, for example through predefined interfaces. Because not all systems require or support bidirectional data integration, the services are split into two different categories. Either a service extracts data from a system or a service loads data into a system. This categorization reduces the implementation effort for the DEV. Implemented services should be usable as generic as possible, therefor the DEV defines certain configurable parameters like the credentials of a web service or query parameters for an API **(UC 2)**. The system supports configurable services by providing an interface which allows the DEV to define the parameters of a service. Some project data which shall be integrated may not be extractable from a system, this could be files which are only on the local computer of the SA. To be able to extract this kind of data, the relevant data has to be passed to the pipeline at execution time. Therefore the DEV uses an interface provided by the system to declare if an implemented service is either *Active*, if the service extracts the data actively from another system e.g. an API, or *Passive* for services needing additional data at run-time (**UC 3**). The DEV exposes the domain model of the underlying system using a mechanism provided by the application (**UC 4**). To enable this the system must provide or define a standardized meta model to the DEV enabling the exposure of a domain model.

The system exposes a graphical interface (GUI) to the end user for storing and modifying multiple instances of a service's configuration **(UC 5)**. While this interface will mainly be used by the SA, the DEV could also use the graphical interface to test and validate his implemented service **(UC 6)**. The SA uses the application's graphical interface to explore the domain model of the source or target system **(UC 7)**. The SA uses the application's graphical interface for creating mappings between the source and the target system (**UC 8**). The DEV also uses the application to test the functionality of his implemented services (**UC 9**). The Mappings shall be stored independently of the service configurations to maximize its re-usability. To integrate data from a source to a target system the SA uses the application's GUI to create pipelines between the two systems (**UC 10**). A pipeline is composed of one service configuration for each the extractor and the loader service and a mapping previously created. The DEV uses the same functionality to test and verify the correctness of implemented services (**UC 11**). If a pipeline was defined by the SA or the DEV, the graphic interface of the system can be used to invoke the execution of that pipeline. The DEV uses this functionality for testing purposes when implementing services (**UC 12**), while the SA uses the functionality in a production environment where actual project data is integrated in or from a SAKM system (**UC 13**). The SA uses the application's graphical interface to verify the correctness of executed pipelines by reviewing logs generated while a pipeline was executed (**UC 14**). When the data model of the underlying source or target system changes, the SA uses the application to modify existing mapping configurations using the same graphical interface described before (**UC 15**). The SA uses the application's API to trigger the execution of a pipeline using events or a time based scheduling tool (e.g. *crontab*[1]) (**UC 16**). Table 2.1 presents the summarized list of the use cases grouped by the actor's roles.

DEV use cases	
UC 1	A developer implements services using predefined interfaces to extend the functionality of the service e.g. for extracting Data from a specific source like GitHub or Microsoft Excel.
UC 2	The DEV defines configuration parameters which are required to execute the service.
UC 3	The DEV defines if data is fetched actively (e.g. REST-Based) or passively (e.g. Microsoft Excel files) by the service.
UC 4	The DEV exposes the domain model of the source or target system.
UC 6	The DEV uses the application's graphical interface to test the correctness of the configuration of his implement service.
UC 9	The DEV creates a mapping configuration between a source and a target system's domain model using his implement service(s).
UC 11	The DEV creates a pipeline using created services to verify its correctness.
UC 12	The DEV invokes the execution of a pipeline through the application's GUI to test created pipelines and its underlying services.
SA use cases	
UC 5	The SA provides configuration data like a URI, credentials or query parameters which are specific to the selected service.
UC 6	The SA uses the application's graphical interface to explore the data model of a service.
UC 7	The SA uses the application's GUI to explore the domain model of a service's underlying system.
UC 8	The SA creates a mapping configuration between the source and the target systems domain model.
UC 10	The SA creates pipelines by selecting a extractor and loader service configuration and a mapping using the application's GUI.
UC 13	The SA invokes the execution of a pipeline through the application's GUI to integrate project data from a system to another.

[1] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html

UC 14	The SA uses the application's GUI to review the logs of executed pipelines.
UC 15	The SA modifies existing pipeline if there is an change in the data model.
UC 16	The SA uses the API of the application to automate the execution of pipelines.

Table 2.1.: Use cases of the SyncPipes prototype

3. Existing data integration tools

There are several data integration tools on the market, some of them are even available as open-source software. To identify possible alternatives for solving the identified problem and observe how the existing solutions handle identified use cases, a review of available data integration tools is required. The review process consisting of three steps has been conducted and described below.

1. Using search engines like *Google*, *Google Scholar* and *Scopus*[2] multiple searches involving combinations of the keywords *data integration tools*, *ETL tools* and *ETL and "linked data"* were performed. The tools which where found and considered relevant where:

    - Apatar[3]
    - CloverETL[4]
    - IBM InfoSphere DataStage[5]
    - Informatica[6]
    - Pentaho[7]
    - RhinoETL[8]
    - Talend Open Studio for Data Integration[9]
    - UnifiedViews[10]

[2] https://www.scopus.com
[3] http://www.apatar.com/
[4] http://www.cloveretl.com/
[5] http://www-03.ibm.com/software/products/en/ibminfodata
[6] https://www.informatica.com/products/data-integration.html
[7] http://www.pentaho.com/
[8] https://hibernatingrhinos.com/oss/rhino-etl
[9] https://www.talend.com/download/talend-open-studio#t4
[10] http://unifiedviews.eu/

2. Only tools which are at least provide the core functionality as open source components where selected for the review:
    - Apatar
    - CloverETL
    - Pentaho
    - RhinoETL
    - Talend Open Studio for Data Integration
    - UnifiedViews
3. By analyzing the user documentations, manuals and demonstration videos of the selected tools the key technologies and principles were identified and described in the following paragraphs.

Apatar is an open-source data integration application based on *Java*. The software can be used as desktop or server application or it can be embedded into third party software. It supports a wide range of interfaces to mainly relational databases and web services. *Apatar* features a visual designer where data integration pipelines can be defined without the need of programming. However the developers community is no longer active (last commit in 2011[11]).

CloverETL is a *Java*-based partially open-source data integration product. The product is available in three different versions[12]. *CloverETL Designer* is a desktop application based on the *Eclipse Rich Client Platform* (RCP) [13] enabling data integration from a desktop environment. The *CloverETL Designer* allows visual composition of extract-transform-load (ETL) pipelines. The created pipelines can be used with the commercial *CloverETL Server* which adds scheduling and monitoring capabilities. The application also supports the implementation of more complex operation throughout a *Java* based domain-specific language (DSL) called *CTL*.

Pentaho Data Integration[14] consists of a *Java*-based desktop application, enabling the user to visually compose ETL pipelines, and an engine which

[11] https://sourceforge.net/p/apatar/code/1391/log/?path=
[12] http://www.cloveretl.com/products
[13] https://wiki.eclipse.org/Rich_Client_Platform
[14] http://www.pentaho.com/product/data-integration

can either be deployed on the same machine, another machine or on multiple machines as a cluster. The desktop application features a large library of pre-build components which can be used to compose custom pipelines.

Talend Open Studio for Data Integration is also based on *Eclipse RCP* and therefore a desktop application. In contrast to the other tools mentioned before, it only operates as code generator and does not integrate the data itself. A visual designer is used to define data integration pipelines. The tool then generates *Java* scripts based on the defined pipeline. These scripts can then be executed to do the data integration.

These applications, as well as similar, can be summarized as generic ETL tools with a focus on the problem domain of data-warehousing and business intelligence. These tools all have in common that they focus on visually creating data pipelines. Non of them requires the end user to have software engineering skills. Even if all tools are extensible, if not explicit their are at least partially open-source, their main focus is not the application's extensibility but its rich tool set of predefined connectors and data-transformers. The goal of these applications is to cover a very broad domain and achieve a very high interoperability between all kinds of different systems. This makes the tools more complex and requires a higher effort to understand and integrate them into existing infrastructure landscapes.

Another application worth mentioning is **UnifiedViews**, which focuses on ETL processes handling linked-data [4]. The application handles the integration of *Resource Description Framework* (RDF) data and therefore, is very domain specific. The application is implemented in *Java* and is available as open-source software[15]. The application is very similar to other ETL tools and provides a graphical user interface for creating pipelines between a source and a target system.

Another identified tool the *C# .net* ETL framework **RhinoETL**. The framework is very minimalistic and does not provide a graphical user interface. It focuses on the developers implementing custom ETL pipelines rather then on the end user. Even though being minimalistic the framework provides a lightweight DSL and a library of typical data transformation operations like aggregation or partitioning.

[15]https://github.com/UnifiedViews/Core

4. Architecture

Based on the use cases described in Chapter 2, a prototype of the application was created. The following chapter describes which architectural approach was chosen for the server and the client application. On the server side the Hypertext Transfer Protocol (HTTP) is used to provide a RESTful API. The client side application is a single-page web application which connects to the server using the HTTP protocol via *XMLHttpRequest*s from the end users browser.

Both the server and the client follow the SOLID object orient design principle, which was originally described by Robert C. Martin. SOLID is an acronym for the following design principles [5]:

- **Single responsibility principle** "A class should have one, and only one, reason to change."
- **The Open Closed Principle** "You should be able to extend a classes behavior, without modifying it."
- **The Liskov Substitution Principle** "Derived classes must be substitutable for their base classes."
- **The Interface Segregation Principle** "Make fine grained interfaces that are client specific."
- **The Dependency Inversion Principle** "Depend on abstractions, not on concretions."

The technology stack used for the client- and server-side application is often refereed as *MEAN-Stack* which is an acronym for the used technologies [6]:

- **M**ongoDB
- **E**xpress.js
- **A**ngular.js
- **N**ode.js

Using a predefined set of technologies enables a quicker prototypical implementation, since important architectural choices already have been made. Also the used technologies are mature and have proven their capabilities in the industry.

4.1. Server

4.1.1. Overview

Derived from use case **UC 1** the application is build to be extensible by a developer. The application consists of a core which holds the logic for providing a RESTful interface, storing configurations (**UC 5**), mappings (**UC 8**), pipelines (**UC 10**) and transforming data (**UC 13**). The application is extensible with two different types of services:

1. **Extractor-Services** enable the developer to extract data from custom source systems. As stated in use case **UC 3** data can be fetched actively or passively, therefore the developer is required to implement a method indicating the type of the service (either passive or active) to the application. *Passive* Extractor-Services are supplied with the raw data, to process, at run-time.
2. **Loader-Service** enables the developer to load transformed data into a specific target system.

Both service types require the developer to expose a service specific configuration which the application will use to store service specific configuration parameters like security credentials or request parameters (**UC 2**).

The application uses the JSON-Schema[16] meta model to enable the developer to describe the domain model of service (**UC 4**).

Despite most popular web application application follow the Model-View-Controller (MVC) architectural design pattern, the SyncPipes application does not utilize the MVC pattern. The obvious reason for this is the lack of a view layer. The HTTP server uses JSON as data interchange format which is nativity supported by *JavaScript*. For this reason no view layer is necessary. The application however strictly separates the Model and the Controller to adhere to the separation of concerns software design principle.

The application provides a Kernel class which implements two run modes. These run modes act as a producer-consumer construct. The first run mode, called server, exposes an HTTP API to provide an interface for configuring and monitoring the application. The API utilizes the RESTful principle originally described by Roy Thomas Fielding [7] to provide a standardized and well known way of accessing the application's data. Using the REST-Interface the client invokes the execution of pipelines, instead of immediate execution, the pipeline is en-queued into a dedicated message queue. The en-queued messages are distributed to at least one worker. The worker is a dedicated process which uses the application's kernel in its second run-mode. The worker run-mode acts therefore as a consumer for the message queue.

4.1.2. Technology

TypeScript was chosen as programming language for the prototypical implementation of server side application. TypeScript is an open-source programming language developed by *Microsoft*. The language is "[...] a typed superset of JavaScript that compiles to plain JavaScript". [8]. The core of the JavaScript programming language is described in the ECMA Standard 262 [9]. With the 6th edition of the ECMAScript language specification, published by *Ecma International* in June 2016, features like *modules*, *classes* and *arrow functions*

[16] http://json-schema.org/

where introduced. TypeScript also provides these features along with other features:

- Generics [10, Sec. 1.9]
- Interfaces [10, Sec. 7]
- Decorators also known as annotations [10, Sec. 8.6]
- Enumerated types [10, Sec. 9]

The main reason for choosing *TypeScript* over *JavaScript* was to make the extensibility of the application (**UC 2**) as intuitive as possible for the developers. JavaScript does not provide interfaces in the way object oriented programming languages like *Java* or *C#* do, this makes it more difficult for developers to implement new services for the application. Even if TypeScript provides interfaces only at compile-time [10], it makes it easier for developers to find errors based on the output of the Typescript compiler.

```
services/githubIssueExtractor/Configuration.ts(5,14): error
↪  TS2420: Class 'Configuration' incorrectly implements
↪  interface 'IServiceConfiguration'. Property 'getSchema' is
↪  missing in type 'Configuration'.
```

Listing 4.1.: TypeScript compiler output for missing method implementation

JavaScript is mainly used to enhance the user experience on the client side, but it can also be used on the server side of a web application. A commonly used server-side *JavaScript* solution is *node.js*[17], which is based on the *V8*[18] *JavaScript* engine. The minimum *node.js* version which is required for executing the application is *v6.0.0*, because the application is compiled to JavaScript complying with the ECMAScript 2015 language Specification. External dependencies of the application are managed through the *node.js* packages manager *npm*[19].

Since the application was developed using *TypeScript*, it has to be compiled to *JavaScript*. Along with compiling the application, other tasks like copying files,

[17] https://nodejs.org/en/
[18] https://developers.google.com/v8/
[19] https://www.npmjs.com/

deleting old built artifacts or executing unit tests are required to be executed repeatedly. To Achieve this task automatically so called build automation tools are used. A widely used build automation tool for *JavaScript* is *gulp.js*[20] which is build upon *node.js*. Using a build automation tool helps new developers, which are creating services for the application, to quicker use their service because they don't need to understand the whole build process in detail.

The application is build upon the *express.js*[21] web framework, which is widely used within the *node.js* environment. Many other frameworks and applications are build upon *express.js*[22]. The framework provides an simple and lightweight abstraction layer to the HTTP protocol, which allows users to quickly implement HTTP-Servers.

The application needs to be able, to store various configuration values (**UC 10**). Because the application is *JavaScript* and *JSON-Schema* based, it was decided to use *MongoDB* as database system. *MongoDB* is a document-oriented database which uses JSON as data interchange format. This allows seamless integration into *JavaScript* applications.

As stated before the application utilizes the producer-consumer design pattern. The communication between the producer and the consumer is performed using the opensource message broker software *RabbitMQ*[23]. *RabbitMQ* implements the *Advanced Message Queuing Protocol* (AMQP) which is an "[...] an open standard for passing business messages between applications or organizations" [11]. Using *RabbitMQ* allows the application to distribute work across multiple processes or physical systems.

4.2. Client

4.2.1. Overview

As the Software Architect needs to create, maintain, execute and monitor pipelines easily a basic graphical interface was build using modern web develop-

[20] http://gulpjs.com/
[21] http://expressjs.com
[22] http://expressjs.com/en/resources/frameworks.html
[23] https://www.rabbitmq.com/

ment techniques. For the client application a single-page web (*SPA*) application was implemented. In contrast to classic web application were the whole page is reloaded for each action taken by the user, a *SPA* only reloads partial areas of the page using *XMLHttpRequest*s, also known as asynchronous JavaScript and XML (*AJAX*).

4.2.2. Technology

For the client application the *JavaScript* framework *Angular.js*[24] was used. *Angular.js* is based on the *Model-view-viewmodel* (MVVM) architectural design pattern. The MVVM pattern is a variation of Martin Follower's Presentation Model [12] and was first described by John Grossman in 2005 [13]. The idea of the MVVM pattern is the same as of the MVC pattern, to separate the data and logic from the presentation layer. The model layer of the MVVM pattern is like the model layer of the MVC pattern an contains the domain models or an data access layer. In this application the model layer is the REST client which handles accessing the data of the API. The view layer is the same as in the MVC pattern which is the user interface, in our case the view layer is the HTML markup. The view and the model are connected with each other through the ViewModel which exposes properties and methods which can be used by the view. The ViewModel acts as stateful data converter between the View and the Model. *Angular.js* automatically synchronizes the values of the view with the values of the model. A example would be a form field which is bound the the property of a model, if the value of the form field is changes *Angular.js* updates the value of the model and vice versa.

[24] https://angularjs.org/

5. Implementation

The following chapter describes how the application was implemented to fulfill the use cases as well as the architectural constraints described in Chapter 4. The client and the sever applications are strictly separated and do not depend on each others source code. The communication between the two systems is completely done via the HTTP-Protocol utilizing the RESTful API of the server.

5.1. Server

All files related to the server application are stored inside the `server` folder of the project. An overview over all files can be found in Figure 5.1. The `dist` folder contains the compiled application, which is executable by *node.js*. Documentation for developers extending the application is stored inside the `docs` folder, this documentation can also be found in the appendix. The documentation of the API was created using *API Blueprint* which is a *Markdown*-based description language for documenting HTTP APIs[25]. The file `package.json` and the folder `node_modules` are used by *node.js*, the exact functionality is explained in Subsection 5.1.1. The application uses environment variables for the configuration. Besides using the operating system for setting these variables, the `.env` file can be used to set these variables. The `.env` file is processed by a *npm* package called *dotenv*[26] which parses the file an makes the defined variables available to the applications process. The `Dockerfile`, `docker-compose.yml`, `.dockerignore` and `docker.js` are used for making the application compatible with Docker[27], details about the implementation of *Docker* can be found in Subsection 5.1.4. The `gulpfile.js` defines the build process of the application and is elucidated in Subsection 5.1.2. *TypeScript* definition dependencies are

[25] https://apiblueprint.org/
[26] https://github.com/motdotla/dotenv
[27] https://www.docker.com/

stored inside the `typings.json` a complete description of this file and the associated functionality can be found in Subsection 5.1.3.

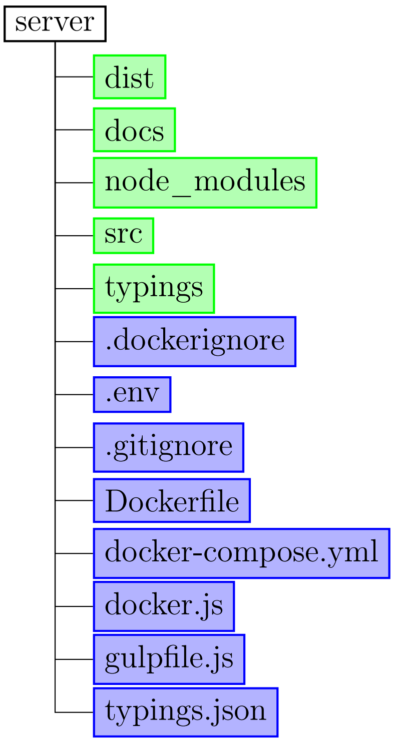


Figure 5.1.: Server directory structure, green boxes are folders and blue boxes are files.

5.1.1. Dependencies

The dependencies of the application are managed using *node.js*'s package manager *npm*, which is shipped with *node.js* by default. Using a package manager has several advantages, it allows to specify the applications dependencies by a specific version. *Npm* provides a special syntax for declaring the required version, that allows for example to specify the minimum required version, if a new version is available *npm* will use this version instead. If a required package has dependencies itself *npm* will resolve this dependencies automatically. All dependencies are stored inside the *package.json* which contains a JSON-Object with several keys describing the package. The dependencies of are package are split into two different keys. The *dependencies* key lists all dependencies required for executing the application, while the *devDependencies* key lists all packages required for building the application. To install the dependencies defined inside the *package.json* the developer has to invoke the *npm install* command inside his terminal.

5.1.2. Gulp

Building the application requires several steps to be executed, to make this process reproducible and easy to use for the developer, *gulp.js* was used. *Gulp.js* is a build automation tool for *node.js* written in *JavaScript*. It uses the *node.js* stream api[28] to transform data. Over 2000 plugins are available for *gulp.js*[29] this allows to quickly implement sophisticated build processes. Plugins are available as *node.js* packages an can be obtained using *npm*. Since these dependencies are only required for building the application, they where defined inside the *devDependencies* section of the applications `package.json` file. The following packages where used to create the build process:

gulp *Gulp.js* it self, provides basic functions for reading, transforming and writing files.

gulp-typescript *Gulp.js* plugin wrapping the *TypeScript* compiler, used for compiling *TypeScript* to *JavaScript*.[30]

del Library for deleting temporary files and old build artifacts generated during the build process.[31]

gulp-live-server Used for managing the applications processes during development.[32]

run-sequence Lightweight utility to run depended gulp task in order.[33]

mocha *JavaScript* test framework with support for *node.js*.[34]

chai Assertion library with support for *node.js* used in conjunction with *chai*[35].

gulp-mocha *Gulp* plugin to run *Mocha* tests with *gulp.js*.

[28]https://nodejs.org/api/stream.html
[29]https://github.com/gulpjs/gulp#what-is-gulp
[30]https://github.com/ivogabe/gulp-typescript
[31]https://github.com/sindresorhus/del
[32]https://github.com/gimm/gulp-live-server
[33]https://github.com/OverZealous/run-sequence
[34]https://mochajs.org/
[35]http://chaijs.com/

Gulp itself consists of two different *npm* packages. The `gulp` package is a library which is included in the `gulpfile.js` and where the build steps are defined. The second package required for using gulp is `gulp-cli` which is a command-line-tool for executing the `gulpfile.js` in the current directory. The applications `gulpfile.js` defines several tasks which can be invoked on the terminal by entering `gulp <task-name>`, where task name is one of the following:

default The **default** task is invoked if *gulp* is executed without an argument. The **default** task is an alias for the **build** task.

build The **build** task builds a executable *node.js* application by running the **clean**, **copy** and **compile** tasks synchronously in that exact order. This is necessary because *gulp* tasks run with maximum concurrency[36]. Running **clean** in parallel with **copy** or **compile** would potential result in a situation where files are deleted immediately after their creation.

clean The **clean** task deletes contents of the `dist` directory where the compiled application is stored. The task is necessary to prevent side effect through orphaned files, for example if the corresponding file has been deleted.

copy The **clean** task copies non *Typescript* files to the `dist` folder. An example would be `.json`-files containing the JSON-Schema of a service.

compile The **compile** task compiles all *TypeScript* files ending with `.ts` to *JavaScript* files and stores them inside the `dist` directory.

watch The **watch** task observes all *TypeScript* files for changes and executes the **build** task if a file was changed.

serve The **serve** task executes the application, for this two processes are started. One processes is executing the `Kernel` in worker mode, the other in server mode. Like the **watch** task the **server** task observes the file-system for changes an compiles the *TypeScript* files. In addition to compiling the source code, the two processes are restarted after successful compilation.

[36] https://github.com/gulpjs/gulp/blob/master/docs/recipes/running-tasks-in-series.md

5.1.3. TypeScript Definitions

Most of the available *node.js* packages are written in *JavaScript*, this leads to the problem that the *TypeScript* compiler does not know the structure of the package. That results in semantic errors when compiling a *TypeScript* file accessing methods, classes or functions of a *JavaScript* file. To solve this problem *TypeScript* provides so called *declaration files*[37], which describe the structure of the library used. Writing and managing these definitions is a time intensive process for developers. A large collection of *TypeScript* definition files is made available through an open source project called *DefinitelyTyped*[38]. While the project has it's own utility for managing *TypeScript* definition file dependencies, the authors of the tool suggest to use another utility[39] called *Typings*[40]. We followed this suggestion for the implementation of our application. *Typings* command-line application can be obtained using *npm* (`npm install -global typings`). All required definition file dependencies are stored inside a file called `typings.json`. The concept is very similar to the `package.json` of *npm*. To install the stored dependencies a developer has to invoke `typings install` on the command line.

5.1.4. Docker

Docker is an open-source project for lightweight visualization and software deployment, it uses operating-system-level virtualization. We decided to use Docker to provide a simple mechanism of running the application without having to install several dependencies like *MongoDB* or *RabbitMQ*. A Docker container is an runtime-instance of an image[41]. Docker images can be derived from other Docker images, that minimizes the effort for building and running custom containers. A public repository with images that can be used and customized is available at https://hub.docker.com/. We used `node:onbuild`[42] as base image for our application. Only minimal tailoring was necessary to create a custom image. The instruction for creating the image are stored inside

[37] https://www.typescriptlang.org/docs/handbook/writing-declaration-files.html
[38] http://definitelytyped.org/
[39] https://github.com/DefinitelyTyped/tsd/issues/269
[40] https://github.com/typings/typings
[41] https://docs.docker.com/engine/reference/glossary/#image
[42] https://hub.docker.com/_/node/

the `Dockerfile` inside the root folder of the application. Running SyncPipes requires two processes to be started, by default only one process runs inside a Docker container. As the current application is prototype we decided to run both processes inside a single container. To achieve this an auxiliary *JavaScript* file called `docker.js` was created. The file serves as entry point for the container and starts two child processes using the *node.js* `child_process` module[43]. Within the script a function `start` is defined which takes a file name as argument. The function spawns a new *node.js* process and redirects the `stdout` and `stderr` to the parent process. If a child process exists, it is automatically restarted after five seconds. The script can be found in Listing 5.1.

```
var child_process = require('child_process');

function start(file) {
    var proc = child_process.spawn('node', [file]);
    proc.stdout.on('data', function (data) {
        console.log(data.toString());
    });

    proc.stderr.on('data', function (data) {
        console.error(data.toString());
    });

    proc.on('exit', function (code) {
        console.log('Child process exited with code ' + code);
        setTimeout(() => {
            start(file);
        }, 5000);
    });
}

start('./dist/server.js');
start('./dist/worker.js');
```

Listing 5.1.: Docker entrypoint node.js script

[43] https://nodejs.org/api/child_process.html

5.1.5. Application Core

All files related to the applications core are stored inside the `src/app` directory. The directory structure is described in Figure 5.2. The directory structure was chooses to reach the architectural goal of SoC.

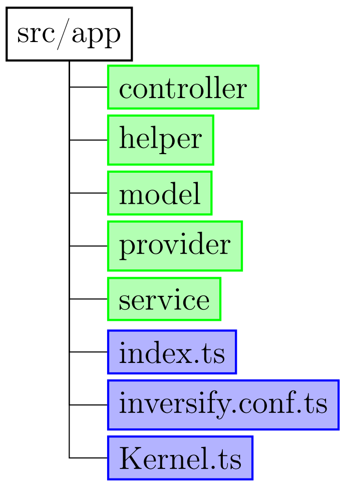


Figure 5.2.: application core directory structure, green boxes are folders and blue boxes are files.

The `Kernel.ts` contains a `Kernel` class which is central entry point of the application. Initialization of the application components is handled inside that class. To follow the inversion of control design principle (IoC) the *npm* package *InversifyJS*[44] was used. The package provides a lightweight dependency injection component for *TypeScript*. We followed the package author's suggestion[45] to configure the IoC container in a separate file `inversify.conf.ts` which handles the configuration and loading of service-providers. Available service-providers are stored inside the `provider` folder. All HTTP controllers required for handling request and responses to or from the REST API are held inside the `controller` folder. The `helper` folder contains several auxiliary classes and functions required for data mapping and transformation. Models of the ODM are stored inside the `model` folder. The `service` folder contains interfaces and classes required for extending the application, these files are re-exported within a single file called `index.ts`, this concepts is taken from *Angular.js 2*[46] framework and is called *barrel*[47].

[44] https://github.com/inversify/InversifyJS
[45] https://github.com/inversify/InversifyJS#step-3-create-and-configure-a-kernel
[46] https://angular.io/
[47] https://angular.io/docs/ts/latest/glossary.html#!#barrel

5.1.6. RESTful API

For implementing the API the *Express* framework was used, which is a popular and widely used web framework. The framework is only a minimal layer on top of the HTTP-Protocol providing features like routing and middleware handling. *Express* describes several ways to structure an application[48], we decided to use the *MVC style controllers* approach where every resource has its own controller. For the routing *TypeScript decorators*[49] were used, *decorators* are syntactic metadata and can be compared to *Java* annotations[50]. We used *decorators* as an idiomatic approach for using *TypeScript* with *Express*. Listing 5.2 illustrates a controller using decorators for the routing.

```
import { Request, Response } from 'express';
// Import decorators
import { AbstractController, RoutePrefix, Route } from "./Controller";

// Define a prefix for all @Route decorators within the class
@RoutePrefix('/mappings')
export class MappingController extends AbstractController {

    // Method is available at /mappings/ via HTTP GET
    @Route('/', 'GET')
    index(request: Request, response: Response) {}

    // Method with dynamic parameter is available at /mapping/:id via
     ↪ HTTP-GET.
    // :id is dynamic and can be any value
    @Route('/:id', 'GET')
    view(request: Request, response: Response) {}
}
```

Listing 5.2.: Express MVC controller using TypeScript decorators

For storing the metadata about the routing, a *npm* package called *reflect-metadata*[51] was used. The package provides a method `defineMetadata` which allows to define arbitrary metadata for a class, method or property. Listing 5.3 shows how the *reflect-metadata* package is used for storing routing metadata.

[48] http://expressjs.com/en/starter/faq.html#how-should-i-structure-my-application
[49] https://www.typescriptlang.org/docs/handbook/decorators.html
[50] http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.7
[51] https://github.com/rbuckton/ReflectDecorators

The `RoutePrefix` function is a decorator for classes which defines a prefix for all methods that are decorated with the `@Route` decorator, which is defined below.

```
export function RoutePrefix(prefix: string) {
    return function (target: Function): void {
        Reflect.defineMetadata('route:prefix', prefix, target);
    };
}

export function Route(route: string, method: string) {
    return (target: Object, propertyKey: string, descriptor:
 ↪  TypedPropertyDescriptor<any>) => {
        Reflect.defineMetadata('route:path', route, target, propertyKey);
        Reflect.defineMetadata('route:method', method, target,
 ↪  propertyKey);
    };
}
```

Listing 5.3.: SyncPipes routing decorators

The defined metadata is later used in the `Kernel`'s `server` method, which initializes the *Express* framework. The `server` method creates a new instance of the *Express* router using `express.Router()`[52]. Afterwards the private method `loadController`, which takes an instance of `AbstractController` as an argument, is called. The method iterates over the methods of the given instance, checks it for routing metadata and adds the route to the *Express* router instance.

Besides initializing the routing the `Kernel`'s `server` method also configures *Express* to use a middleware called *body-parser*[53] which parses the body of HTTP *PUT* or *POST* requests. The *body-parser* middleware offers to parse different formats, since our API only uses *JSON* as data interchange format we just use the *JSON* handler. After the initialization is done `listen` with the configured port is called on the *Express* instance.

[52]http://expressjs.com/en/4x/api.html#express.router
[53]https://github.com/expressjs/body-parser

5.1.7. Persistence Layer

The application is required to persist several entities for this purpose *MongoDB* was used. To access *MongoDB* we used *mongoose*[54] which is a *JavaScript* based Object-Document-Mapper (ODM) for accessing *MongoDB*. The purpose of an ODM is the same as the purpose of an Object-Relational-Mapper (ORM), but for document-orientated databases. Based on the class diagram in Figure 5.3 the structure of the *MongoDB* documents were defined.

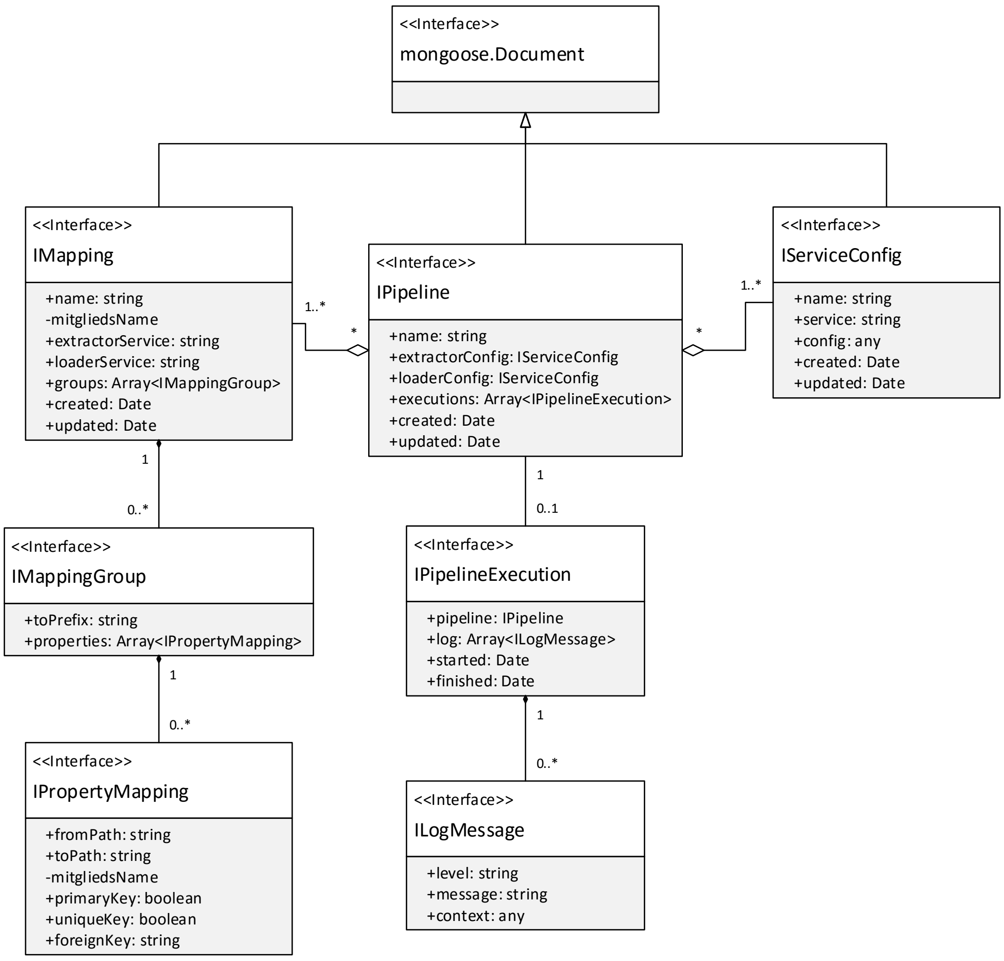


Figure 5.3.: Persistence Layer Class Diagram

To implement a new model the following steps where conducted, first an interface was created based on the UML model from Figure 5.3. The schema of

[54] http://mongoosejs.com/

the *MongoDB* document was defined using `mongoose.Schema`[55]. The last step was to create a `mongoose.Model`[56] using both the defined interface and the defined schema. The example implementation of the `ServiceConfig` model can be found in Listing 5.4.

```
import * as mongoose from 'mongoose';
// Define interface
export interface IServiceConfig extends mongoose.Document {
    name: string,
    service: string,
    config: any,
    created: Date;
    updated: Date;
}
// Define Schema
var ServiceConfigSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    service: {
        type: String,
        required: true
    },
    config: mongoose.Schema.Types.Mixed,
    created: {
        type: Date,
        "default": Date.now
    },
    updated: {
        type: Date,
        "default": Date.now
    }
}).pre('save', function (next) {
    this.updated = new Date();
    next();
});
// Create Model using the schema and the interface
export var ServiceConfig: mongoose.Model<IServiceConfig> =
 ↪  mongoose.model<IServiceConfig>('ServiceConfig', ServiceConfigSchema);
```

Listing 5.4.: Implementation of a model using mongoose

[55]http://mongoosejs.com/docs/api.html#schema_Schema
[56]http://mongoosejs.com/docs/api.html#model_Model

The connection to *MongoDB* is established in the `Kernel`'s `boot` method.

5.1.8. Message Queuing

The communication between the worker process and API process is done using the open source message broker *RabbitMQ* which implements the *AMQP* messaging protocol. To be able to use *AMQP* with *node.js* a *npm* package called *amqplib* was used. Like the database connection, the connection to *RabbitMQ* is established inside the `Kernel`'s `boot` method. The *AMQP* communication is encapsulated inside the `JobScheduler` class which handles both the sending and the receiving of messages through *RabbitMQ*. The `JobScheduler` is a service provider and therefore stored inside `provider` folder.

5.1.9. Third Party Services

One central aspect of the application is the extensibility through so called services, for that we defined several interfaces to which the developers service must adhere. These interfaces are stored inside the `service` folder. A complete overview of the interfaces can be found in Figure 5.4.

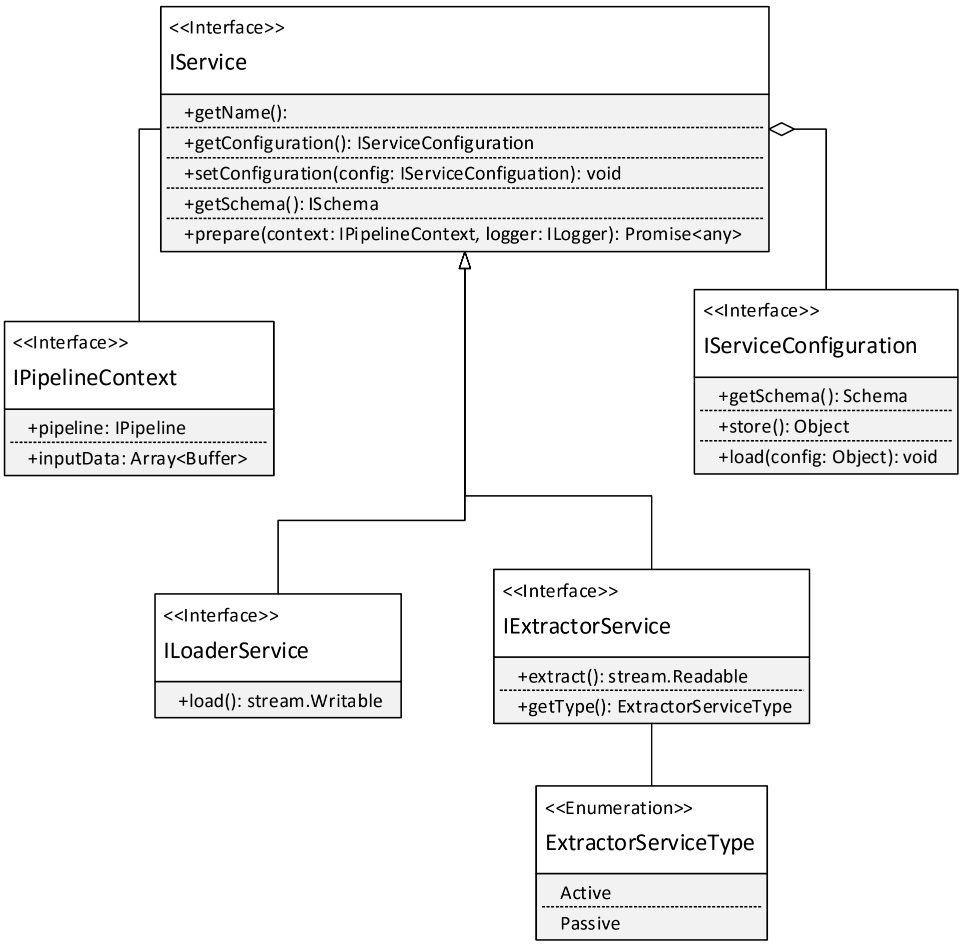


Figure 5.4.: Service extension layer

If a service requires a custom configuration, this could be credentials for accessing a database or web service, the service must create a custom configuration class implementing the `IServiceConfiguration`. The service configuration should describe the structure of the configuration data using JSON-Schema therefore it must implement the `getSchema()` method which has to return an instance of the `ISchema` interface. The `ISchema` interface is the applications internal representation of a JSON-Schema instance. An implementation of the `ISchema` interface is provided by the application and can be used by the developer. To persist instances of the configuration, the service must implement the `store()` method, which returns a plain JSON object of the configuration values. The object must validate against the configuration's JSON-Schema otherwise the application will throw an error. The `load()` method is used by the application to load persisted configuration data from the database into the service's configuration.

As mentioned in Chapter 4 the system is extensible with two different types of services. Both extractor and loader service have the same base interface `IService`. The `IService` interface defines five methods which must be implemented by the service. The `getName()` method must return the name of the service as a string, the service name is used by the system and is exposed through the REST API. The `getConfiguration()` method should return an empty instance of the service's configuration or null, if the service has no specific configuration. The `setConfiguration()` method is called by the application to set a specific configuration used for the extraction or loading process. A service must describe the data it provides or expects using JSON-Schema, for this purpose the `getSchema()` method has to be implement to return an instance of the `ISchema` interface. The `prepare()` method is called by the SyncPipes application right before the extraction or loading process starts. An instance of `IPipelineContext` is passed along with an instance of the `ILogger` interface to the `prepare()` method. The `IPipelineContext` has two properties, the first property `pipeline` is an instance of `IPipeline` which is the pipeline that will be executed. The second property `inputData` is an array of `Buffers`[57] which is only available for passive extractor services. The second argument passed to `prepare()` is an instance of `ILogger` which enables the services to write log messages, which are associated with the current pipeline execution by the application. Inside the body of the `prepare()` method the developer should perform tasks which are required before the actual processing can start. A example would be connection to a database system or setting up an HTTP client with credentials. For indication if the initialization of the service was successful or failed the method has to return a `Promise`[58] object.

[57] https://nodejs.org/api/buffer.html
[58] https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise

A service which extracts data from a source system has to implement the `IExtractorService` interface which extend the `IService` interface with two additional methods. The `getType()` method must return a value of the `ExtractorServiceType` enumeration which is either `Active` or `Passive`. If the type is `Passive`, the application expects to be provided with additional data through the REST API. The data is made available to the service through the `IPipelineContext.inputData` property. Secondly a `IExtractorService` has to implement the `extract()` method which must return a `stream.Readable`[59]. The extractor service then uses this stream to push extracted data to the application.

Loading data into another system is done with loader services. To implement a loader the service the developer has to implement the `ILoaderService` interface which also extends the `IService` interface. The interface requires the implementation of the `load()` method which must return an instance of `stream.Writable`[60]. Through that stream the services is provided with data, which shall be loaded into the target system, by the application.

All third party services are stored inside the `src/services` folder, an example of the file structure of a service is provided in Figure 5.5.

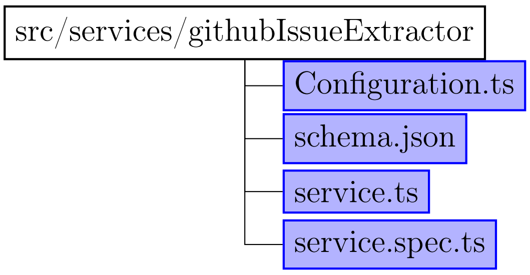


Figure 5.5.: File structure of a third party extraction service, where the `Configuration.ts` contains an implementation of the `IServiceConfiguration` interface. The `schema.json` contains a JSON-Schema describing the data which the service extracts. `service.ts` contains the actual implementation of the `IExtractorService`. The `service.spec.ts` file contains unit test, this file is optional.

A third party service has to be loaded by the application, for this purpose the `Kernel` class provides a `loadService` method which takes an instance of

[59]https://nodejs.org/api/stream.html#stream_class_stream_readable
[60]https://nodejs.org/api/stream.html#stream_class_stream_writable

the service, which should be loaded, as an argument. The service loading is conducted inside the `bootstrap.ts` file.

There are four example services provided with the source code of the application. The `GitHubIssueExtractorService` uses the *GitHub* REST API[61] to extract repositories, issues and comments from a *GitHub* organization, which can be defined in the configuration. The `PureIssueLoaderService` is the corresponding loader service to the `GitHubIssueExtractorService` service. It uses the data from *GitHub* to store it in a *MySQL* database. The extensions also creates foreign key relations. While the `GitHubIssueExtractorService` is a active extractor service, because it fetches the data on its own, the `RequirementsExcelExtractorService` is a passive extractor service. The service extracts requirements and test-cases from a *Microsoft Excel* file. For that the service needs to be provided with at least one *XLSX* file by the application, therefore the service is passive. The last example service is the `RequirementsExcelExtractorService` which loads the extracted requirements and test cases into a *MySQL* database.

[61] https://developer.github.com/v3/

5.1.10. Pipeline Execution

Transforming data using a extractor and loader services requires executing a defined pipeline, for that all modules of the application have to work together, Figure 5.6 provides an overview how the modules interact.

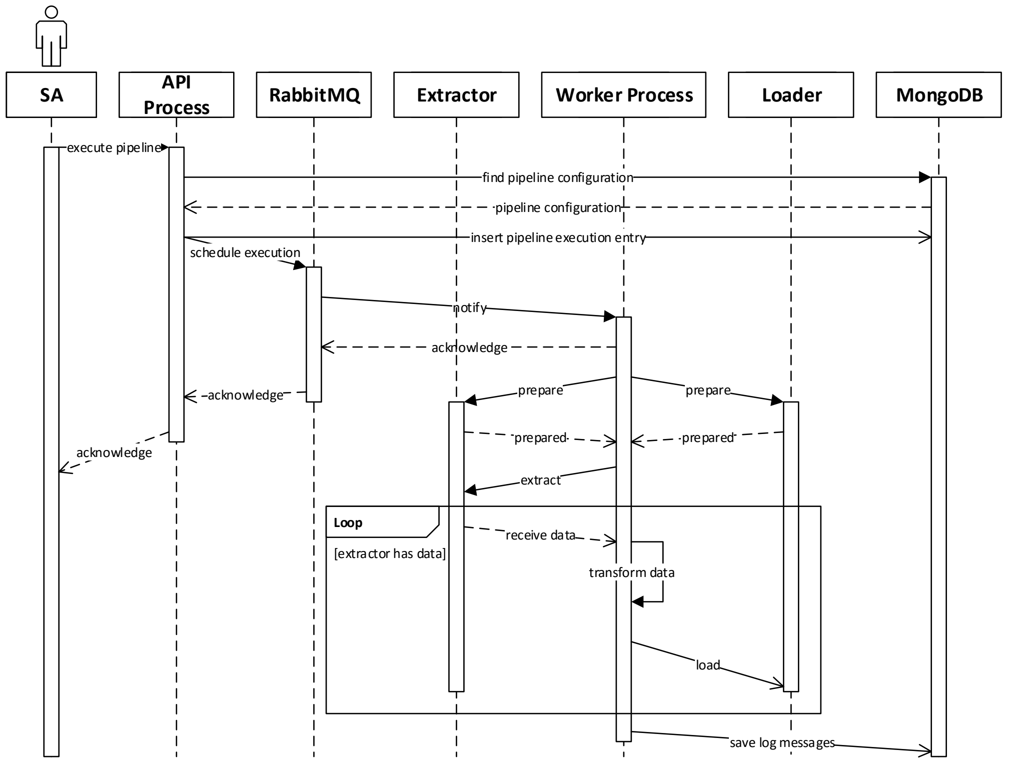


Figure 5.6.: UML sequence diagram describing the processing process

The execution of a pipeline is started via the REST API. For that a special resource exists at `/api/v1/pipelines/:id/actions/execute`, where `:id` is the id of the pipeline which should be started. The url has to be invokes using a HTTP POST request. If the extractor service which is associated with a pipeline that is passive, the endpoint expects a `mulitpart/form-data`[62] request providing the data which should be processed. For parsing `mulitpart/form-data` requests the application uses a *Express* middleware called *Multer*[63]. After verifying that the requested pipeline exists, the associated services are loaded and if the request is valid, the application creates a new `IPipelineExecution`

[62] https://www.ietf.org/rfc/rfc2388.txt
[63] https://github.com/expressjs/multer

object and persists it in the database. Afterwards a new `PipelineContext` is created and populated with data. The `PipelineContext` is then serialized to JSON and sent to the message broker. Finally a HTTP response containing a JSON representation of the `IPipelineExecution` is sent to the client. The worker process is connected to *RabbitMQ* listening for messages, if a message is received it deserializes the payload back into a `PipelineContext`. Inside the worker process the services associated with the pipeline are loaded from the `ServiceManager`. Afterwards `prepare()` is called on both the extractor and the loader service, the worker waits until both `prepare()` methods are finished. Using the *node.js* stream api the worker reads the data from the extractor service and pushes it to the mapper, which is an instance of `stream.Transform`[64]. Using a mapping stored in the database, the mapper transforms the data provided by the extractor service and pushes it to the loaded service. The functionality of the mapper is described detailed in Subsection 5.1.11.

5.1.11. Data transformation

The following subsection describes the process of data transformation through an exemplary use case described below. We constructed a use case with data that could have been extracted from an issue tracker application. The data is deeply nested and should be transformed into a flatter JSON structure, the transformation process should also work backwards. An excerpt of the nested data, which was used as input data, can be found in Listing 5.5. The expected output data, which is flat structured, is described in Listing 5.6.

[64]https://nodejs.org/api/stream.html#stream_class_stream_transform

```
{
    "projects": [
        {
            "id": 1,
            "name": "Project A",
            "issues": [
                {
                    "id": 1,
                    "title": "Mapper is not working",
                    "body": "The mapper is not working",
                    "author": {
                        "id": 1,
                        "username": "test.user1",
                        "gender": "male"
                    },
                    "comments": [
                        {
                            "id": 1,
                            "body": "Its working for me",
                            "author": {
                                "id": 2,
                                "username": "test.user2",
                                "gender": "male"
                            }
                        },
                        // more comments...
                    ]
                },
                // more issues...
            ]
        },
        // more projects...
    ]
};
```

Listing 5.5.: Example data which is deeply nested, authors can occur multiple times.

```
{
    "projects": [
        {
            "id": 1,
            "name": "Project A"
        },
        // All remaining projects
    ],
    "issues": [
        {
            "id": 1,
            "title": "Mapper is not working",
            "body": "The mapper is not working",
            "project_id": 1,
            "author_id": 1
        },
        // All remaining issues
    ],
    "comments": [
        {
            "id": 1,
            "body": "Its working for me",
            "issue_id": 1,
            "author_id": 2
        },
        // All remaining comments
    ],
    "users": [
        {
            "id": 1,
            "username": "test.user1",
            "gender": "male"
        },
        // All remaining users
    ]
}
```

Listing 5.6.: Example of a flat dataset, the association between the entities is preserved by using foreign keys like `project_id`.

Three fundamental elements are required to conduct the data transformation process:

1. Both services must describe the data they either provide or expect using JSON-Schema. These schemata are passed to the mapper as `ISchema` objects.
2. A mapping, which describes how to transform the data between the source and the target system.
3. Data which will be transformed, the data must be validatable against the JSON-Schema of the extractor service.

The mapping is composed using the three different classes `IMapping`, `IMappingGroup` and `IPropertyMapping`. The classes are structured in the following way:

IMapping Meta data container for the mapping.

- **name** The name of the mapping used for identifying withing the user interface.
- **extractorService** The name of the extractor service.
- **loaderService** The name of the loader service.
- **created** Date when the mapping was created.
- **updated** Date when the mapping was last modified.
- **groups** An array of `IMappingGroups`.
  - **toPrefix** The Prefix path inside the target object.
  - **properties** An array of `IPropertyMappings`.
    - **fromPath** The path from where the value is extracted.
    - **toPath** The path were the value should be inserted.
    - **primaryKey** If this flag is set to `true` the data transformer will merge the target object, if mapped multiple times, to an object.
    - **uniqueKey** If the target object is an array this flag will, if set to `true`, configure the data transformer to only add the object once.
    - **foreignKey** Placement path of the current mapping group.

The mapping is defined using the `IMapping` which encapsulates the `IMappingGroup` which then again encapsulates the `IPropertyMapping`. A core concept of the mapper is a special file-path like construct to access properties of a JSON object. For example using the path `projects/issues/author/name` on the object from Listing 5.5 would refer to a list with the names of all issue authors. These paths are used for defining the mapping.

The **IMapping** is a top level object which holds meta-data about the mapping. The only relevant part for the actual data transformation process is the `groups` property, which contains an array of `IMappingGroup` instances.

A **IMappingGroup** encapsulates `IPropertyMapping` objects which are mapped to the same prefix path. The prefix path is a path to a JSON object or array inside the target object. The purpose of the `IMappingGroup` is to group the property mappings by their common target. Following the example of the nested to flat mapping, the properties are grouped by the four top level arrays `projects`, `issues`, `comments` and `users`. The reverse mapping from flat to nested would then define groups like `projects`, `projects/issues`, `projects/issues/author`, `projects/issues/comments`, etc. The groups are necessary for the data transformer to correctly re-associate properties in the target object, e.g. if a user *John Doe* has the email address *john.doe@acme.com* in the source object, the user should have the same email address in the target system as well.

The **IPropertyMapping** class has two properties **fromPath** and **toPath** which use the same format as the `IMappingGroup`'s `toPrefix` property. The `fromPath` property defines which values should be extracted from the source object. The extracted values are then inserted into the target object using the `toPath` property.

The **uniqueKey** is a boolean flag, if set to true, it indicates that a mapped property is a unique key. The transformer takes this information and won't insert duplicate objects into the target. The `uniqueKey` is only used if the `toPrefix` of the outer `IMappingGroup` points to an array. Particular this is useful for denormalized data structures, an example would be the `author` entity in Listing 5.5, which exists, multiple times, to prevent duplicate entries in the target object, the `uniqueKey` flag can be used. An example mapping for transforming the `author` from the nested to the flat dataset, using the `uniqueKey` property, can be found in Listing 5.7.

```json
{
  "toPrefix": "users",
  "properties": [
    {
        "fromPath": "projects/issues/author/id",
        "toPath": "users/id",
        "uniqueKey": true
    },
    {
        "fromPath": "projects/issues/author/username",
        "toPath": "users/username",
        "uniqueKey": false
    },
    {
        "fromPath": "projects/issues/author/gender",
        "toPath": "users/gender",
        "uniqueKey": false
    },
    {
        "fromPath": "projects/issues/comments/author/id",
        "toPath": "users/id",
        "uniqueKey": true
    },
    {
        "fromPath": "projects/issues/comments/author/username",
        "toPath": "users/username",
        "uniqueKey": false
    },
    {
        "fromPath": "projects/issues/comments/author/gender",
        "toPath": "users/gender",
        "uniqueKey": false
    }
  ]
}
```

Listing 5.7.: Example mapping of the author from nested to flat, in favor of readability only properties important for this example are used. The mapping configures the mapper to extract all authors from the source dataset using the `id` as an unique identifier.

The **foreignKey** property, is used as placement information to preserve relations between the entities. The `foreignKey` property controls where the parent `IMappingGroup` has to be placed. This property is especially important if a flat data structure is mapped to a nested data structure. Without the `foreignKey` property, the mapping of the `issues` from the flat to the nested data-set would look like in Listing 5.8.

```
[
  {
    "toPrefix": "projects/issues",
    "properties": [
      {
          "fromPath": "issues/id",
          "toPath": "projects/issues/id",
          "uniqueKey": true
      },
      {
          "fromPath": "issues/title",
          "toPath": "projects/issues/title",
          "uniqueKey": false
      },
      {
          "fromPath": "issues/body",
          "toPath": "projects/issues/body",
          "uniqueKey": false
      },
      {
          "fromPath": "issues/project_id",
          "toPath": "projects/id",
          "uniqueKey": false
      }
    ]
  }
]
```

Listing 5.8.: Mapping without using the `foreignKey` property, in favor of readability only properties important for this example are used.

The problem in Listing 5.8 is that the mapper cannot automatically detect which `issue` has to be associated with which `project`. For this purpose the `foreignKey` property was introduced, it gives the mapper the information required for preserving the associations. The enhanced mapping with the use of the `foreignKey` property can be found in Listing 5.9.

```
[
  {
    "toPrefix": "projects/issues",
    "properties": [
      {
          "fromPath": "issues/id",
          "toPath": "projects/issues/id",
          "uniqueKey": true
      },
      {
          "fromPath": "issues/title",
          "toPath": "projects/issues/title",
          "uniqueKey": false
      },
      {
          "fromPath": "issues/body",
          "toPath": "projects/issues/body",
          "uniqueKey": false
      },
      {
          "fromPath": "issues/project_id",
          "toPath": "projects/issues/$foreignKey"
          "uniqueKey": false,
          "foreignKey": "projects/id"
      }
    ]
  }
]
```

Listing 5.9.: Mapping using the `foreignKey` property, this is the refactored solution for Listing 5.8.

The **primaryKey** boolean flag is only relevant if the `toPrefix` of the parent `IMappingGroup` points to an object. If the flag is set to true the mapper merges the properties of the extracted values with the properties in the target object. An annotated example describing how the **primaryKey** property is interpreted by the data transformer can be found in Listing 5.10.

```
// Source object
{
  "users": [
    {"id":1, "name": "John Doe"},
    {"id":1, "location": "Munich"},
  ]
}
// Mapping
[{"fromPath": "users/id", "toPath": "user/id", "primaryKey": true},
{"fromPath": "users/name", "toPath": "user/name"},
{"fromPath": "users/location", "toPath": "user/location"}]
// Object after 1. iteration
{
  "id": 1,
  "name": "John Doe"
}
// Object after 2. iteration with primaryKey = true
{
  "id": 1,
  "name": "John Doe"
  "location": "Munich"
}
// Object after 2. iteration with primaryKey = false
{
  "id": 1,
  "location": "Munich"
}
```

Listing 5.10.: Mapping using the `primaryKey` property. The blocks in the Listing are separated with comments. The first block describes the object from the source system with just a single key `users`. In the second block a mapping is defined, in favor of readability only reduced `IPropertyMapping` objects are present. The `toPrefix` of the mapping group would be `user`. The third block shows how the `user` property of the target object would look like after transforming the first entry of the `users` array. The fourth and fifth block shows each how the object would look like after the transforming the second entry from `users` array. The fourth block with `primaryKey` set to true, the fifth set to false.

The data transformation is centrally handled by the `GraphTransformer`, it is instantiated with the schemata of the extractor and loader service as well as with an instance of `IMapping`. While instantiating, the class converts each

`IMappingGroup` of the passed `IMapping` into a tree. This is done inside the classes `groupMapping()` method, which returns an array of tree `MappingTree` instances. This conversion is necessary to extract the data in the correct order and maintain association while extracting. The `groupMapping()` differentiates between two cases, if all `fromPath` attributes inside a `IMappingGroup` have the same prefix, the first element of the path is taken as root node of the tree. Applied to Listing 5.8 the root node of the tree would be `projects`. If a mapping group contains `fromPath` attributes with have a different prefix, an empty string is taken as root node. The empty string internally refers to the top level element of the input data. In Listing 5.5 the top level element is the object with the `projects` key.

After the `GraphTransformer` has been initialized it can be used to transform instances of the source schema using the provided mapping, for that it provides the `transform()` method. The method can be separated into five different phases which are described in Figure 5.7.

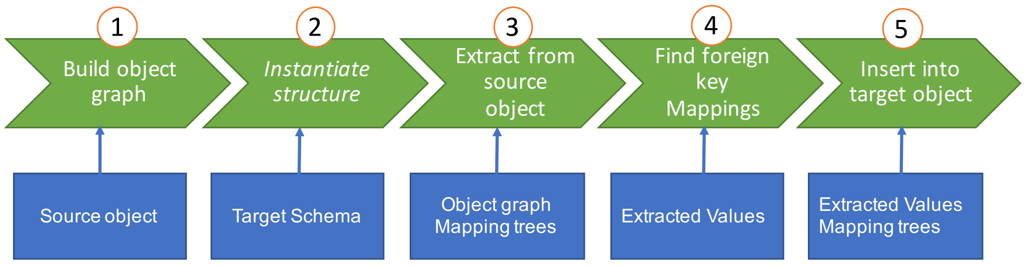

1
Build object graph
2
Instantiate structure
3
Extract from source object
4
Find foreign key Mappings
5
Insert into target object
Source object
Target Schema
Object graph Mapping trees
Extracted Values
Extracted Values Mapping trees

Figure 5.7.: Data transformation workflow, the green arrows are the different steps of the transformation process. The blue boxes describe the required input data for each step.

The method expects a JSON object as parameter. In phase one the passed object will be transformed into a tree structure using the `ObjectGraphNode` and `ObjectGraphLeaf` classes. To implement a tree structure the composite structural design pattern was used [14, 163 et seqq.]. A visualization of an object graph generated based on the data from Listing 5.6 can be found in Figure 5.8.

In the second step of the transformation process an instance of the target schema is created, for that the static method `instantiateStructure()` is called, it takes a JSON-Schema as parameter. The passed schema object

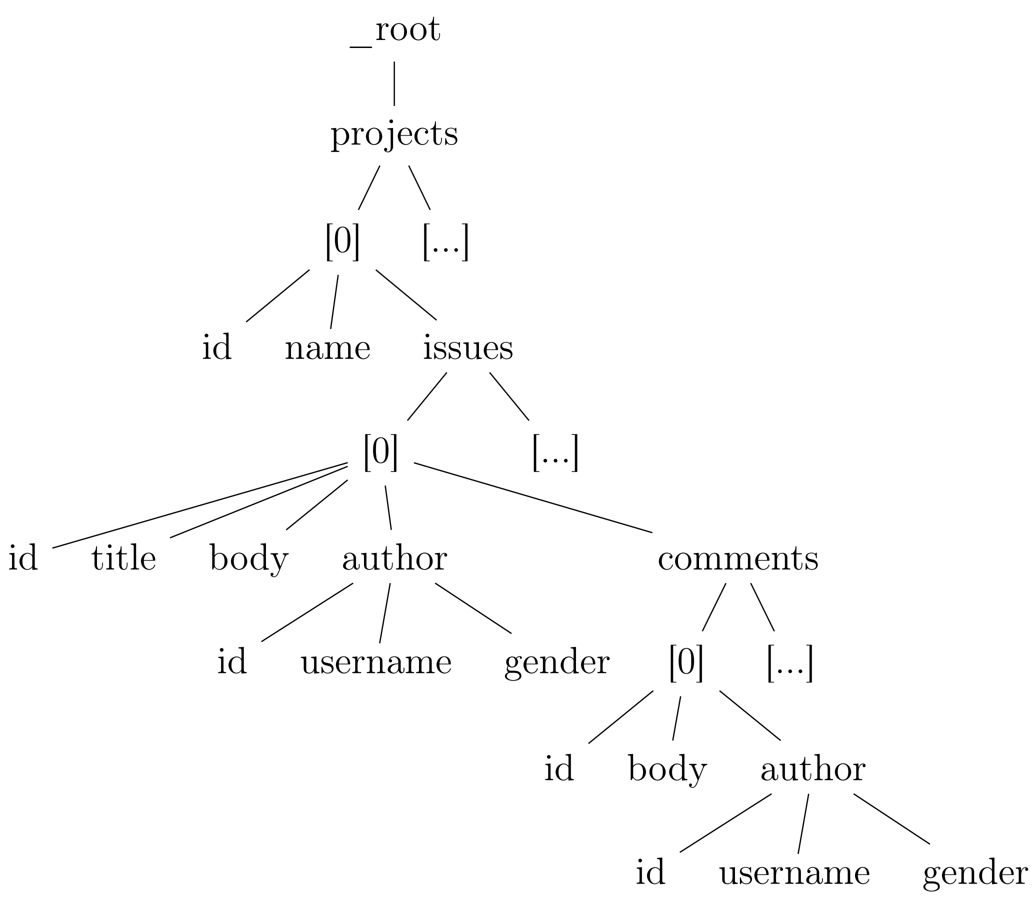


Figure 5.8.: Object graph tree structure from Listing 5.6. Nodes enclosed by square brackets are arrays in the original object.

must be dereferenced, in the context of a JSON schema that means that all references[65] have to be replaced with the actual value. The `Schema` class supports this functionality by having a boolean parameter for the `toObject()` method. Currently the dereferencing implementation only supports inline references. The `instantiateStructure()` recursively traverses the schema and creates the corresponding object. It stops when a node of the array type is found, this is necessary since the length of each array is unknown at this point of the transformation process.

The third step is to iterate over the previously created mapping groups. The first step of each iteration is to extract all values, as defined by the current mapping group, from the object graph. Extracting the values is done inside the `extract()` method, the method expects two arguments. The first argument is an array of partial object graphs. The partial object graphs are extracted

[65] http://json-schema.org/latest/json-schema-core.html#anchor25

from the object graph, generated in step one, by passing the name of the `MappingTree`'s root node to object graph's `getNodeByPrefix()` method, which then returns all partial graphs having a root node with the given name. The second argument of the `extract()` method is the `MappingTree` object of the current mapping group. The `extract()` method itself iterates over the passed array of object graphs and then extracts the values using the passed `MappingTree`. Each extracted value is combined with the mapping information from the `IPropertyMapping` responsible for the extraction of the value. The data structure of the combined object is defined by the `IDestinationValue` interface. Finally the extracted values are grouped inside an array based on the structure of the originally passed array of object graphs, we call these groups *destination value groups*. The method then returns an array of destination value groups, an annotated example of the output can be found in Listing 5.11.

```
groupMapping(
   "projects", // toPrefix
   // fromPath, toPath, isPrimaryKey, isUnique
   propertyMapping("projects/id", "projects/id", true, true),
   propertyMapping("projects/name", "projects/name")
);
[
  // Destination value group of the first project
  [
    {"to": "id", "value": 1, "unique": true, "primary": true, "foreignKey":
↪  null},
    {"to": "name", "value": "Project A", "unique": false, "primary": false,
↪  "foreignKey": null}
  ],
  // Destination value group of the second project
  [
    {"to": "id", "value": 2, "unique": true, "primary": true, "foreignKey":
↪  null},
    {"to": "name", "value": "Project B", "unique": false, "primary": false,
↪  "foreignKey": null}
  ]
]
```

Listing 5.11.: Destination value groups example, which is the output of the `GraphTransformer.extract()` method. The source data structure is described in 5.5. The mapping which was passed the the `extract()` method for generating this output is described in the beginning of the Listing.

In the forth step of the transformation process each *destination value group* is searched for an `IDestinationValue` with non null `foreignKey` property. If a non null property was found it is stored in a temporary variable. This step is an important precondition for the last step of the process.

In the fifth and last step of the transformation process the extracted values are inserted into the target object. Depending on the result from stop four either `insertForeignKey()` or `insert()` is called. `insertForeignKey()` tries to extract all sub objects from the current target object matching the foreign key condition. If no sub-object where found the corresponding sub-object is created within the target object. For that the method traverses the target object using the foreign key path, if a path element is not yet created, the method creates the element using the provided schema to determine its correct type. After the element was created, the method calls itself to actually insert the *destination value group.* For inserting the destination object the foreign key is used to extract all matching objects. For each matched object the method tries to insert the passed *destination value group.* To do this, it is first checked if the path in `IDestinationValue.to` attribute exists in the current foreign key object, if not the path is created using the JSON schema to determine the correct type. The next actual value is inserted into the object. If the `IDestinationValue.to` attributes refers to an object, the properties of the object are merged with the values of the destination value group, if the element where the data should be inserted is an array, the method check if the destination value group has an `IDestinationValue` with to `unique` attribute set to `true`. If so the target array is search for an existing entry matching the value of the `IDestinationValue` with the unique flag, if not the destination value group is just appended to the array. The insertion for destination value groups without a foreign key is similar but simpler. The `insert()` method uses the provided destination prefix to traverse down the path to were the destination value group's entries should be placed. If a path element does not exist, it is created using the JSON schema of the target object. If the traversal has reached the last element of the path the destination is inserted into the element, for that the same procedure `insertForeignKey()` uses is used.

The current implementation of the data transformer has some limitations which should be part of future implementation work. It is not possible to split the value of a property using common string operations like `split` or `substr` and map the result to multiple properties in the target object. Also aggregating multiple properties of the source object to one property of the target object is currently not supported. The current implementation requires the implemented loader services to handle updating or overwriting of existing data, this could be moved the application itself, enabling the developer to do less implementation work.

5.2. Client

As mentioned in Chapter 4 the client application was created using the *Angular.js* framework. The boilerplate for the client application was created using the scaffolding tool *Yeoman*[66]. This step was conducted to improve the velocity of the development process. *Yeoman* is a command-line tool that is extendable with *generators* which can create boilerplate code for all kinds of web related applications. More than 3900 *generators* are available[67] and can be obtained using *npm*. For the client application *generator-gulp-angular*[68] was chosen. The *generator* allows to pick between several different technologies by interactively asking. The application is written using *JavaScript* following the *ECMAScript 6* specification. Since not all browsers fully support the *ECMAScript 6* specification *Babel*[69] is used to compile the code to *ECMAScript 5*.

5.2.1. Dependencies

The generated application uses *Bower*[70] to manage dependencies, *Bower* is a package manager very similar to *npm* but with focus on fronted related packages. All dependencies are stores inside the `bower.json` file.

For communicating with the REST API of the server application the *Restangular*[71] library was selected. The library is a generic HTTP client for interacting with RESTful APIs, it has several advantages over the `$resource` service[72] of *Angular.js*, like returning promises or providing an object oriented interface for building urls.

The application uses the *Material Design*[73] visual language for the user interfaces. *Material Design* was developed by *Google* and describes a way of

[66]http://yeoman.io/
[67]http://yeoman.io/generators/
[68]https://github.com/swiip/generator-gulp-angular
[69]https://babeljs.io/
[70]https://bower.io/
[71]https://github.com/mgonto/restangular
[72]https://docs.angularjs.org/api/ngResource/service/\protect\T1\textdollarresource
[73]https://material.google.com

designing and arranging the components of an user interface. *Angular Material*[74] is a framework based on the *Material Design* specification and was used to create the user interface of this application.

5.2.2. Index Page and Layout

For the user it is important to quickly understand the application, therefore the layout of the interface is kept simple with a focus on the essential. Each page consists of two parts, the navigation bar on the top of the application and the content container below. The navigation bar has five main items and one button for the configuration of the client application it self. The main items are ordered by the logical process of creating a new pipeline. A picture of the navigation bar can be found in Figure 5.9.

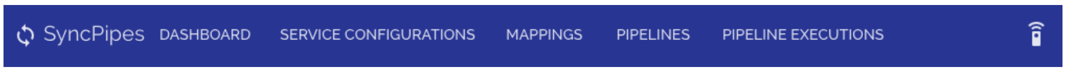


Figure 5.9.: Client application navigation bar with the main items on the left and the endpoint configuration on the right.

The first item links to dashboard which is also the main entry point or the home page. The second and the third items are the service configurations page and the mappings page. Both have no dependency to each other meaning that the user can start by creating a service configuration or a mapping. The forth navigation item is the pipelines page which has a dependency to both the mappings and the service configuration. After a pipeline has been created using a mapping and a service configuration it can be executed, therefore the last main navigation item is the pipeline execution page where the output of each pipeline execution can be viewed. On the right end of the navigation bar a button with a remote control icon is placed. Clicking on the button opens a model dialog allowing the user to configure the REST API endpoint of the SyncPipes server application. A picture of the dialog can be found in Figure 5.10.

[74] https://material.angularjs.org/latest/

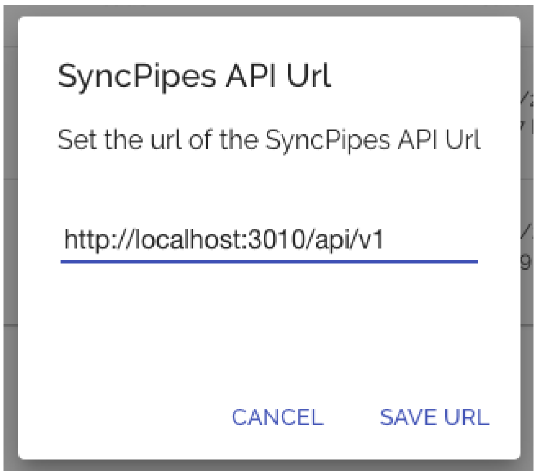
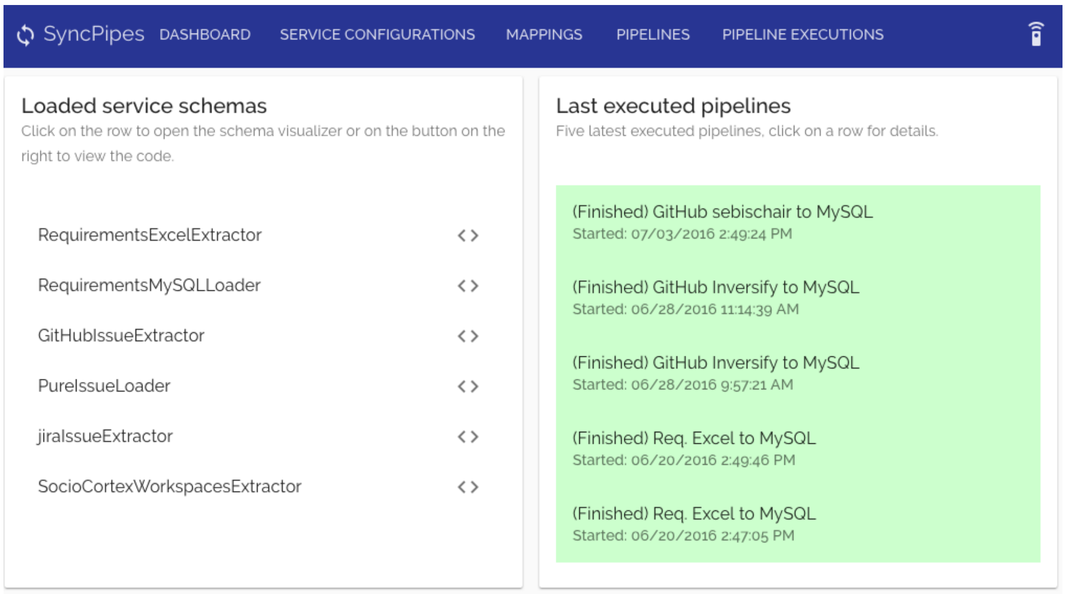

SyncPipes API Url
Set the url of the SyncPipes API Url
http://localhost:3010/api/v1
CANCEL
SAVE URL

Figure 5.10.: Modal dialog for changing the API endpoint

5.2.3. Dashboard

The start page of the application is the dashboard, it enables the user to quickly see all loaded services and see the five last pipeline executions. A picture of the dashboard can be found in Figure 5.11.



Loaded service schemas
Click on the row to open the schema visualizer or on the button on the right to view the code.

RequirementsExcelExtractor
RequirementsMySQLLoader
GitHubIssueExtractor
PureIssueLoader
jiraIssueExtractor
SocioCortexWorkspacesExtractor

Last executed pipelines
Five latest executed pipelines, click on a row for details.

(Finished) GitHub sebischair to MySQL
Started: 07/03/2016 2:49:24 PM

(Finished) GitHub Inversify to MySQL
Started: 06/28/2016 11:14:39 AM

(Finished) GitHub Inversify to MySQL
Started: 06/28/2016 9:57:21 AM

(Finished) Req. Excel to MySQL
Started: 06/20/2016 2:49:46 PM

(Finished) Req. Excel to MySQL
Started: 06/20/2016 2:47:05 PM

Figure 5.11.: Dashboard of the client application, showing all loaded services on the left and the five latest pipeline executions on the right.

By clicking on row in the dashboard's services overview the user can visualize the schema of the corresponding service. The schema is presented in a dialog over the application's content, a image can be found in Figure 5.12. For visualizing the JSON-Schema an external library called *json-schema-viewer*[75] was used. Minor customization were necessary to make the library compatible to *Angular.js*.

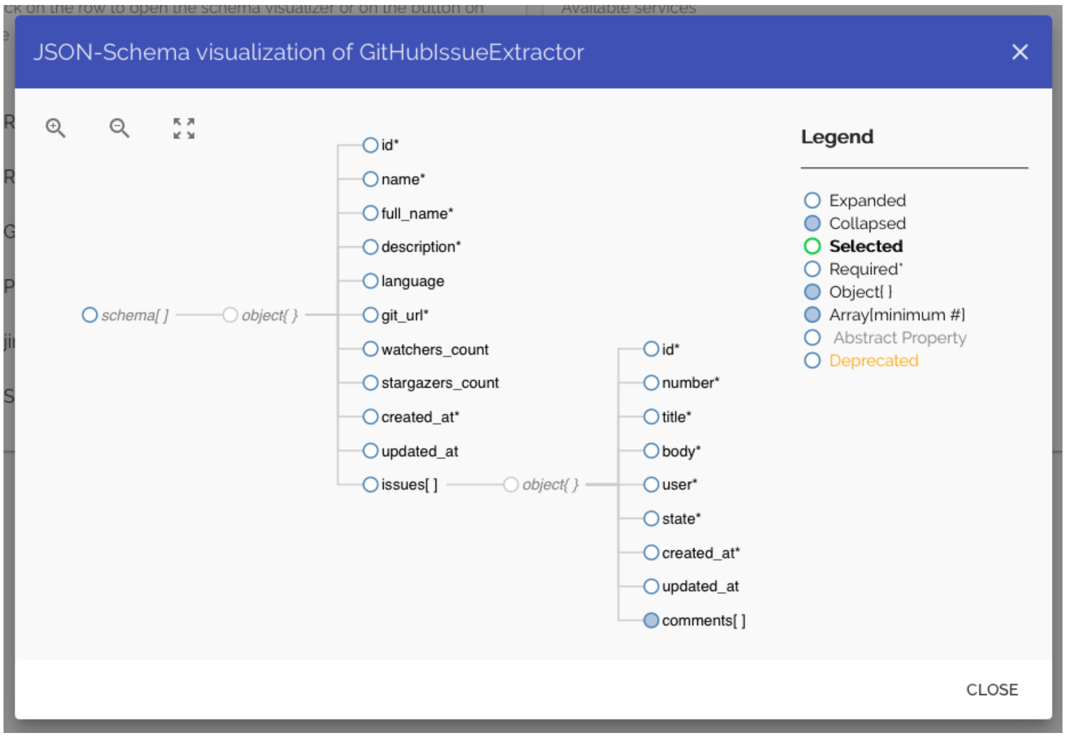
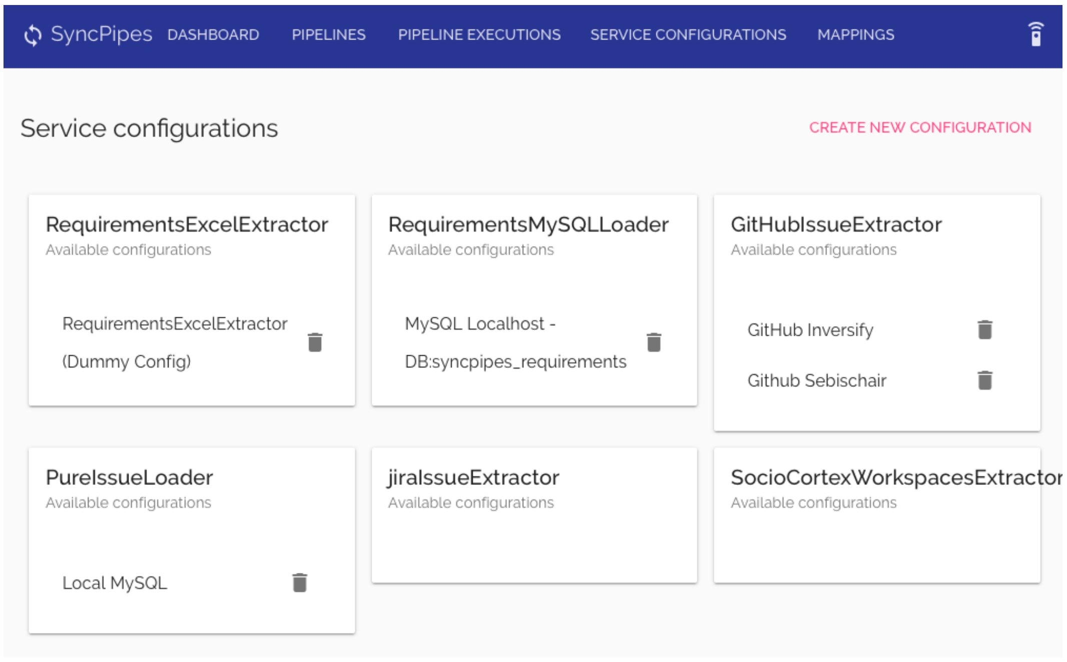


Figure 5.12.: Dialog visualizing the JSON-Schema of a service

5.2.4. Service Configuration

Each service may expose a specific configuration which is editable through the client application. For the service configuration an overview page was created, where all services are listed as boxes. Inside each box all service specific configurations are listed. These configurations can be modified or deleted. For modifying a configuration the user has to click on an entry which opens a dialog. Configurations can be deleted by clicking on the button with the delete icon on the right end of each row. A picture of the page can be found in Figure 5.13.

[75] http://jlblcc.github.io/json-schema-viewer/



Service configurations

RequirementsExcelExtractor
Available configurations
RequirementsExcelExtractor (Dummy Config)

RequirementsMySQLLoader
Available configurations
MySQL Localhost - DB:syncpipes_requirements

GitHubIssueExtractor
Available configurations
GitHub Inversify
Github Sebischair

PureIssueLoader
Available configurations
Local MySQL

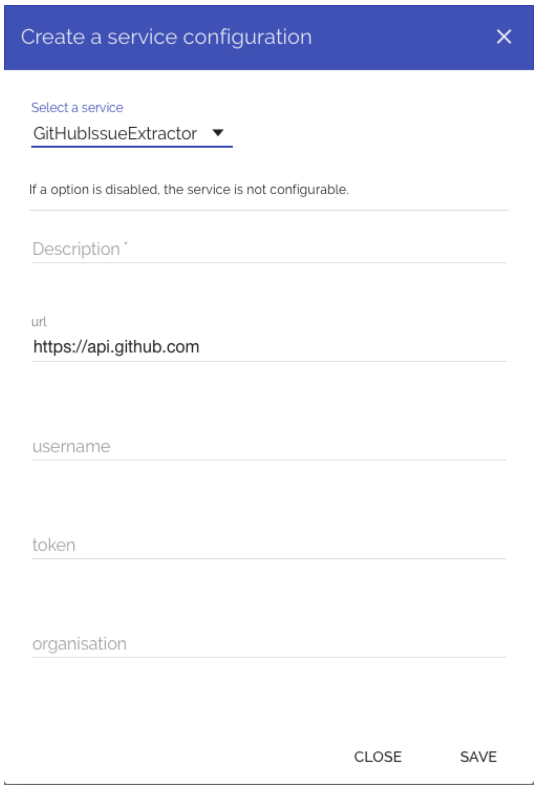
jiraIssueExtractor
Available configurations

SocioCortexWorkspacesExtractor
Available configurations

Figure 5.13.: Service configuration page

New configuration can be created by clicking on the *Create new configuration* button on the top of the page. Clicking on the button opens a modal dialog containing a form. First the user has to select the service for which a new configuration should be created. After selecting a service the application dynamically creates input fields according to the configuration's JSON-Schema of the service. For the dynamic part of the form the *Angular.js* module *Angular Schema From*[76] is used. The module uses a JSON-Schema to generate a form and validate the input. To make the module compatible to the *Angular Material* framework, an additional module called *Angular Material Decorator*[77] was necessary to include. The form dialog is also used when the user clicks on a row on the overview page to edit an existing service configuration. An example of the dialog, displaying the form of the *GitHubIssueExtractor* can be found in Figure 5.14.

[76] https://github.com/json-schema-form/angular-schema-form
[77] https://github.com/json-schema-form/angular-schema-form-material


Create a service configuration

Select a service
GitHubIssueExtractor

If a option is disabled, the service is not configurable.

Description *

url
https://api.github.com

username

token

organisation

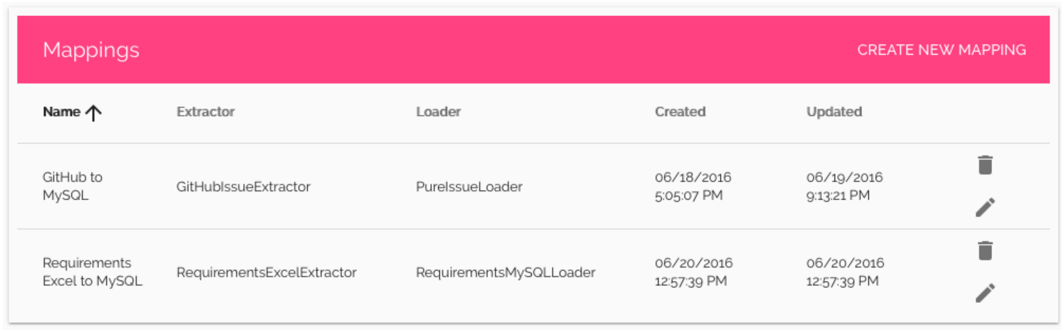
Figure 5.14.: Service configuration form of the *GitHubIssueExtractor*

5.2.5. Mappings

Besides configurations the user needs to define the mapping between the source and the target system, for that another page was created. Existing mappings are listed within a table on the mappings page, where the user can get a quick overview over existing mappings. The table has six columns containing the name, the extractor and loader service, the time when the mapping was created and the time when the mapping was last modified. The last column contains buttons for editing or deleting a mapping. A button for creating new mapping is placed above the table. A screenshot of the table can be found in Figure 5.15. Even if the *Material Design* specification describes a data table component[78] it is not yet implement in the *Angular Material* framework[79]. To overcome this caveat a third party *Angular.js* module called *Material Design Data Table*[80] was used. It provides features like pagination and sorting therefore it was the

[78] https://material.google.com/components/data-tables.html
[79] https://github.com/angular/material/issues/796
[80] https://github.com/daniel-nagy/md-data-table

ideal solution for displaying tabular data while adhering to the *Material Design* specification.

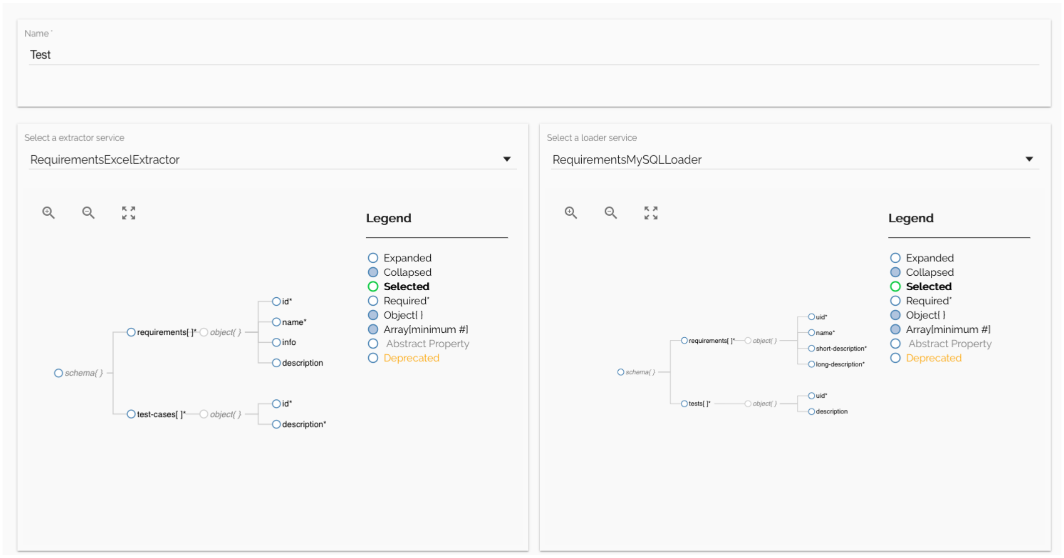


Mappings — CREATE NEW MAPPING

Name ↑	Extractor	Loader	Created	Updated
GitHub to MySQL	GitHubIssueExtractor	PureIssueLoader	06/18/2016 5:05:07 PM	06/19/2016 9:13:21 PM
Requirements Excel to MySQL	RequirementsExcelExtractor	RequirementsMySQLLoader	06/20/2016 12:57:39 PM	06/20/2016 12:57:39 PM

Figure 5.15.: Table of all mappings stored in the system

Due to the complexity of creating or editing mappings the form was implemented as an extra page. In the beginning of the page the user can enter the name of the mapping, which is just a helper for the user and not relevant for the actual process of data transformation. After the input for the name the page is divided into two columns, on the right column the user selects the extractor service, on the left side the loader service. After selecting a loader or extractor service the corresponding JSON-Schema is visualized using the same technique mentioned in Subsection 5.2.3. Visualizing the schemata of the services helps the user to create the mapping since it is easier to understand the data model of the source (extractor) and the target (loader) system. The upper part of the mapping form can be found in Figure 5.16.



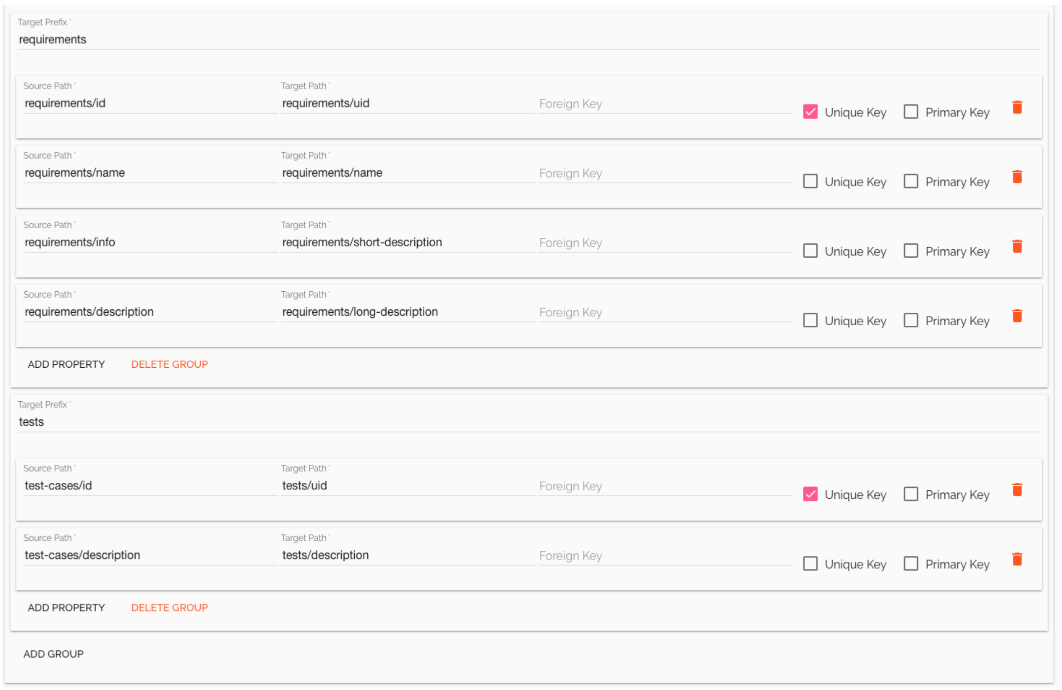
Name *
Test

Select a extractor service
RequirementsExcelExtractor

Select a loader service
RequirementsMySQLLoader

Figure 5.16.: Service selection and schema visualization of the mapping form

The second part of the mapping form is the actual input for the mapping information. The user can create arbitrary mapping groups containing arbitrary property mappings. Following the data model of the mapping, each mapping group has an input field for the `toPrefix`. If the user clicks on a node inside the schema visualizer of the loader service, the input field is automatically filled with the path to the selected node. Inside a mapping group the user can add property mappings by clicking the *Add Property* button inside each group. For each property mapping a new row is added inside the corresponding group containing text input fields for the `fromPath`, `toPath` and `foreignKey` property of the `IPropertyMapping`. The `uniqueKey` and the `primaryKey` properties are mapped to two check-boxes indicating whenever the flag is set or not. The last element of each row is a button with a delete icon enabling the user to delete a property mapping row. An example of a form containing two mapping groups can be found in Figure 5.17.


Target Prefix *
requirements

Source Path *	Target Path *	Foreign Key	Unique Key	Primary Key
requirements/id	requirements/uid		✓	☐
requirements/name	requirements/name		☐	☐
requirements/info	requirements/short-description		☐	☐
requirements/description	requirements/long-description		☐	☐

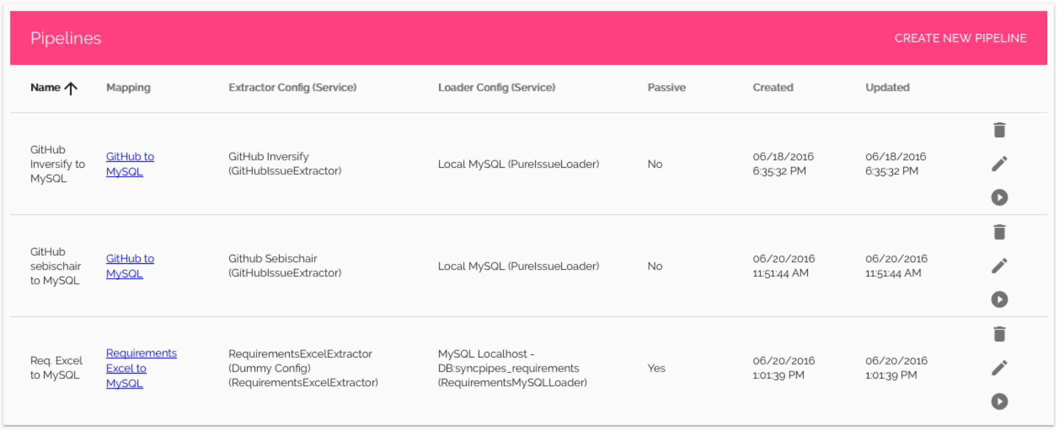
Target Prefix *
tests

Source Path *	Target Path *	Foreign Key	Unique Key	Primary Key
test-cases/id	tests/uid		✓	☐
test-cases/description	tests/description		☐	☐

Figure 5.17.: Mapping form with two mapping groups containing several property mappings

5.2.6. Pipelines

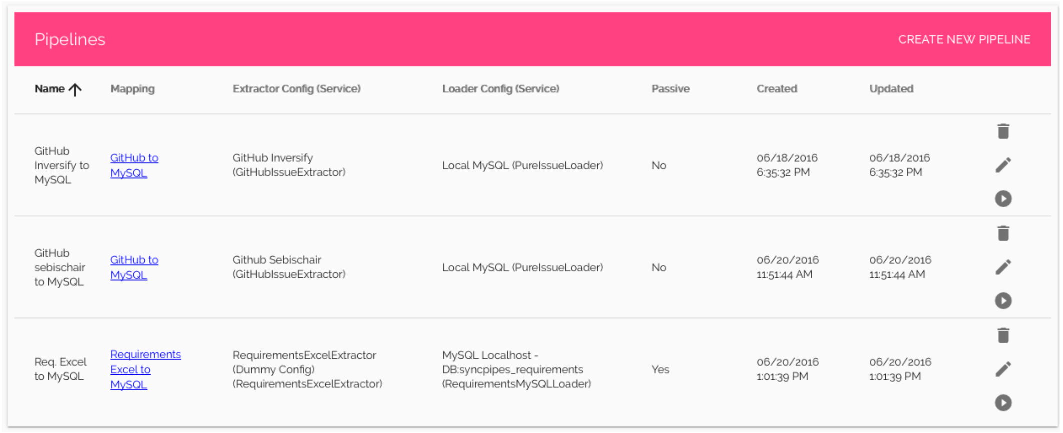
A pipeline is the composition of a loader and extractor configuration and a mapping. All existing mappings are listed within a table on the pipelines page of the application. Like the mappings table, the table uses the *Material Design Data Table* module. The pipeline table contains six columns with the name, the associated mapping, the extractor service configuration, the loader service configuration, a flag indicating whenever the pipeline is passive, the date when the pipeline was created and the date when the pipeline was last modified. The last column contains three different buttons with pipeline specific actions. The pencil icon opens a form in a modal dialog when clicked, enabling the user to modify existing pipelines. Deleting existing pipelines can be done by clicking on the button with the delete icon. Executing a pipeline is done by clicking on the button with the play icon. A screenshot with the table can be found in Figure 5.18.



Name ↑	Mapping	Extractor Config (Service)	Loader Config (Service)	Passive	Created	Updated
GitHub Inversify to MySQL	GitHub to MySQL	GitHub Inversify (GitHubIssueExtractor)	Local MySQL (PureIssueLoader)	No	06/18/2016 6:35:32 PM	06/18/2016 6:35:32 PM
GitHub sebischair to MySQL	GitHub to MySQL	Github Sebischair (GitHubIssueExtractor)	Local MySQL (PureIssueLoader)	No	06/20/2016 11:51:44 AM	06/20/2016 11:51:44 AM
Req. Excel to MySQL	Requirements Excel to MySQL	RequirementsExcelExtractor (Dummy Config) (RequirementsExcelExtractor)	MySQL Localhost - DB:syncpipes_requirements (RequirementsMySQLLoader)	Yes	06/20/2016 1:01:39 PM	06/20/2016 1:01:39 PM

Figure 5.18.: Table with Pipelines

To create a new pipeline the user has to click on the create new pipeline button on the right side above the table, this opens a form dialog. The dialog contains a text input for the name of the pipeline, like the name of the service configuration or the mapping, the name has no relevance to the data transformation process and is just there for the user to ease identifying the correct pipeline. Besides the text input the form dialog contains three select inputs where the user chooses the extractor service configuration, the loader service configuration and the mapping of which the pipeline will be composed. The same dialog is also open when the user clicks the edit button inside the tables actions column. The dialog can be found in Figure 5.19.



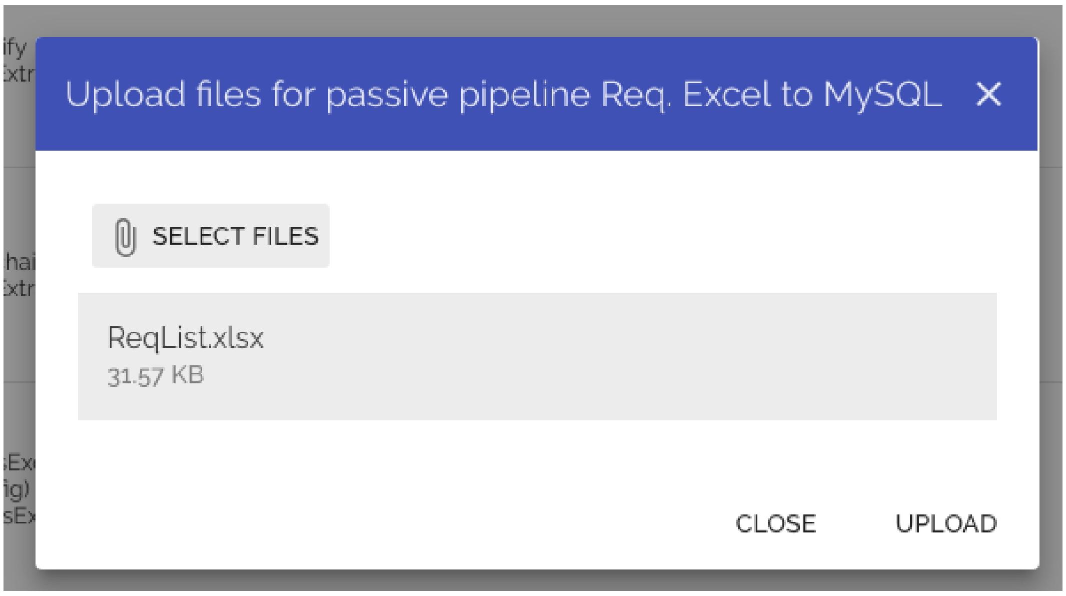
Pipelines

CREATE NEW PIPELINE

Name ↑	Mapping	Extractor Config (Service)	Loader Config (Service)	Passive	Created	Updated
GitHub Inversify to MySQL	GitHub to MySQL	GitHub Inversify (GitHubIssueExtractor)	Local MySQL (PureIssueLoader)	No	06/18/2016 6:35:32 PM	06/18/2016 6:35:32 PM
GitHub sebischair to MySQL	GitHub to MySQL	Github Sebischair (GitHubIssueExtractor)	Local MySQL (PureIssueLoader)	No	06/20/2016 11:51:44 AM	06/20/2016 11:51:44 AM
Req. Excel to MySQL	Requirements Excel to MySQL	RequirementsExcelExtractor (Dummy Config) (RequirementsExcelExtractor)	MySQL Localhost - DB:syncpipes_requirements (RequirementsMySQLLoader)	Yes	06/20/2016 1:01:39 PM	06/20/2016 1:01:39 PM

Figure 5.19.: Modular dialog form for creating or editing pipelines

When the user invokes the execution of a pipeline using the button in the table, the application checks if the pipeline uses a passive extractor service. For passive extractor services the application opens a modal dialog upload form, where the user can select one or more files which should be sent to the pipeline. A picture of the dialog can be found in Figure 5.20. For uploading the file a *Angular.js* module called *ng-file-upload*[81] was used.



Upload files for passive pipeline Req. Excel to MySQL

SELECT FILES

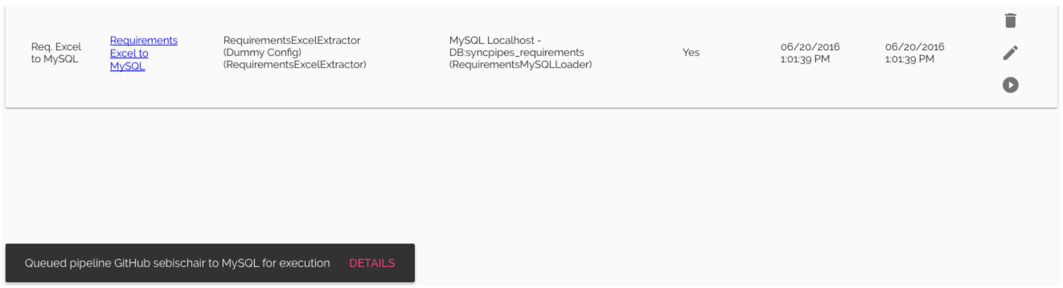
ReqList.xlsx
31.57 KB

CLOSE UPLOAD

Figure 5.20.: Modular dialog for uploading data to execute passive extractor services

[81] https://github.com/danialfarid/ng-file-upload

Whether the extractor service is active or passive, if the pipeline was successfully queued for execution the server return the created pipeline execution object. A toast message[82] with a button linking to created pipeline execution is shown to the user after the client application received a successful response from the server. This allows the user to directly jump to the pipeline execution detail view which is described in Subsection 5.2.7. A example toast message can be found in Figure 5.21.



Req. Excel to MySQL	Requirements Excel to MySQL	RequirementsExcelExtractor (Dummy Config) (RequirementsExcelExtractor)	MySQL Localhost - DB:syncpipes_requirements (RequirementsMySQLLoader)	Yes	06/20/2016 1:01:39 PM	06/20/2016 1:01:39 PM	

Queued pipeline GitHub sebischair to MySQL for execution DETAILS

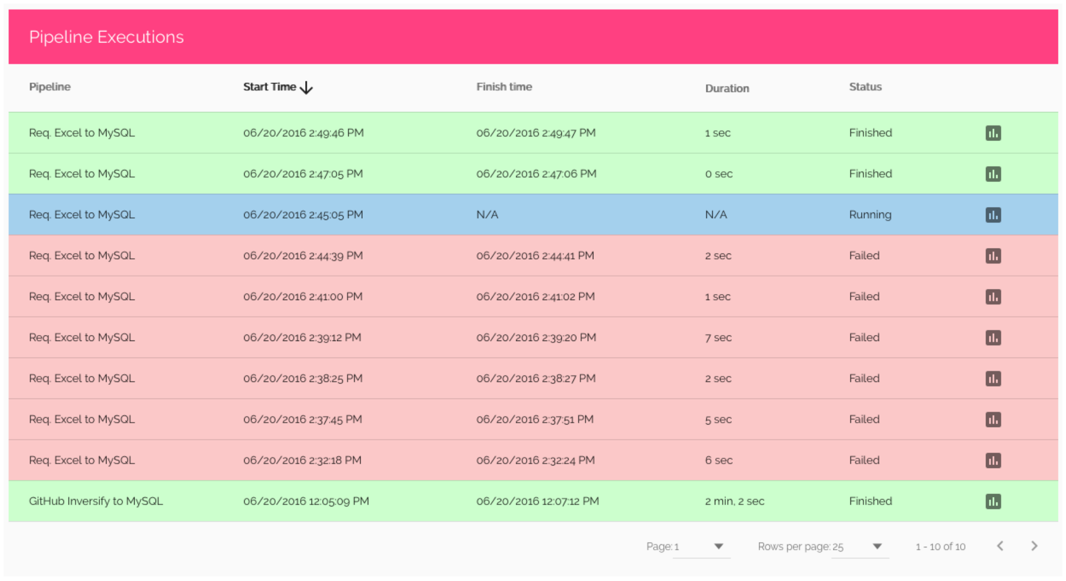
Figure 5.21.: Toast message after successfully queuing a pipeline for execution

5.2.7. Pipeline Executions

For the user it is important to see the status of each pipeline execution, for that two extra pages were created. The first page lists all pipeline executions inside a table. Like all other tables in the client application the table is created using the *Material Design Data Table* module. In contrast to the other tables, the pipeline executions table uses server side sorting and pagination. This was done because a larger amount of entries can be expected for this particular table. The first column of the table contains the name of the executed pipeline. The second column contains the start time of the execution. The finish time of the execution is displayed in the third column, if the execution is still in process *N/A* is displayed instead. The fourth column contains the calculated duration the pipeline has taken for executing. This is information is only available if the pipeline has finished or failed. The duration is calculated using the *JavaScript* library *moment.js*[83]. To format the output of the calculation a *moment.js* plugin called *Moment Duration Format*[84] was used. The fifth column contains the status of the pipeline execution. The status can have three different values,

[82] https://material.google.com/components/snackbars-toasts.html#snackbars-toasts-usage
[83] http://momentjs.com/
[84] https://github.com/jsmreese/moment-duration-format

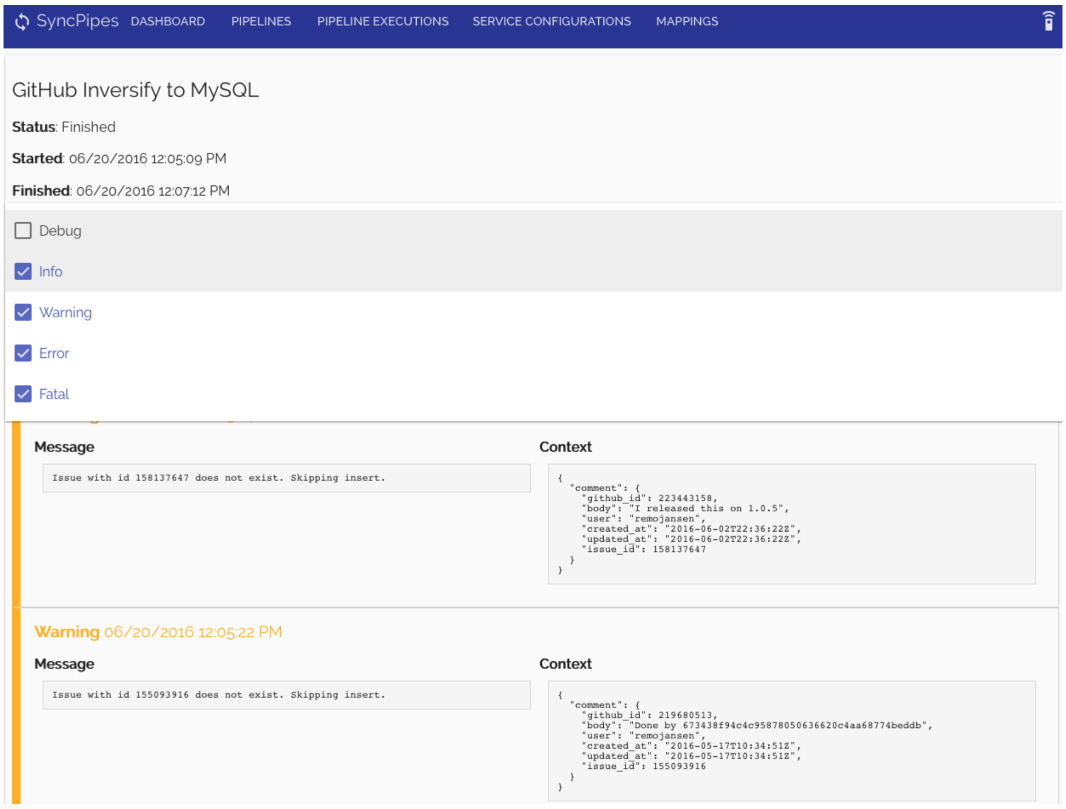
Running indicates that the pipeline execution is currently in progress, *Finished* indicated a successful pipeline execution and *Failed* indicated that an error has occurred while executing the pipeline. To improve the usability of the table each row has a different color based on the execution status. Rows with the status *Running* have a blue background, *Finished* rows have a green background and *Failed* rows have a red background. The last column contains a button linking to the pipeline execution details page. A screenshot of the pipeline executions table can be found in Figure 5.22.


Pipeline Executions

Pipeline	Start Time ↓	Finish time	Duration	Status	
Req. Excel to MySQL	06/20/2016 2:49:46 PM	06/20/2016 2:49:47 PM	1 sec	Finished	📊
Req. Excel to MySQL	06/20/2016 2:47:05 PM	06/20/2016 2:47:06 PM	0 sec	Finished	📊
Req. Excel to MySQL	06/20/2016 2:45:05 PM	N/A	N/A	Running	📊
Req. Excel to MySQL	06/20/2016 2:44:39 PM	06/20/2016 2:44:41 PM	2 sec	Failed	📊
Req. Excel to MySQL	06/20/2016 2:41:00 PM	06/20/2016 2:41:02 PM	1 sec	Failed	📊
Req. Excel to MySQL	06/20/2016 2:39:12 PM	06/20/2016 2:39:20 PM	7 sec	Failed	📊
Req. Excel to MySQL	06/20/2016 2:38:25 PM	06/20/2016 2:38:27 PM	2 sec	Failed	📊
Req. Excel to MySQL	06/20/2016 2:37:45 PM	06/20/2016 2:37:51 PM	5 sec	Failed	📊
Req. Excel to MySQL	06/20/2016 2:32:18 PM	06/20/2016 2:32:24 PM	6 sec	Failed	📊
GitHub Inversify to MySQL	06/20/2016 12:05:09 PM	06/20/2016 12:07:12 PM	2 min, 2 sec	Finished	📊

Figure 5.22.: Pipeline executions table

On the detail page of a pipeline execution the user can see the status, the start time and the finish time. In addition to that all log messages generated during the transformations process are visible. Each log entry is displayed inside a box containing the log level, the time when the log entry was created, the log message and optionally a JSON representation of the context object. Depending on the level of the log message a different color is assigned to left border of each log message entry box. The following colors are assigned: *Debug* (grey), *Info* (blue), *Warning* (yellow), *Error* (red) or *Fatal* (dark red). Above the log messages a select menu is displayed allowing the user to filter for log messages of a specific level. A picture of the page can be found in Figure 5.23.



GitHub Inversify to MySQL

Status: Finished

Started: 06/20/2016 12:05:09 PM

Finished: 06/20/2016 12:07:12 PM

- [ ] Debug
- [x] Info
- [x] Warning
- [x] Error
- [x] Fatal

Message

```
Issue with id 158137647 does not exist. Skipping insert.
```

Context

```
{
  "comment": {
    "github_id": 223443158,
    "body": "I released this on 1.0.5",
    "user": "remojansen",
    "created_at": "2016-06-02T22:36:22Z",
    "updated_at": "2016-06-02T22:36:22Z",
    "issue_id": 158137647
  }
}
```

Warning 06/20/2016 12:05:22 PM

Message

```
Issue with id 155093916 does not exist. Skipping insert.
```

Context

```
{
  "comment": {
    "github_id": 219680513,
    "body": "Done by 673438f94c4c95878050636620c4aa68774beddb",
    "user": "remojansen",
    "created_at": "2016-05-17T10:34:51Z",
    "updated_at": "2016-05-17T10:34:51Z",
    "issue_id": 155093916
  }
}
```

Figure 5.23.: Pipeline execution detail page, with the filter set to displaying all messages except debug messages.

6. Evaluation

To improve the application, find flaws and validate the architecture we conducted an evaluation involving two research associate (RA) at the chair 19 of the computer science faculty. Both RAs have at least four years software engineering experience. The evaluation included both the client and the server application. While the focus for the server application was on the architecture and the implementation process, the focus for the client was on usability. The two RAs were granted access to the source code of the application on the 1st of May 2016. The two RAs where presented with the problem of creating a new pipeline by implementing custom services. Each pipeline requires an extractor and loader service, therefore the RAs had to implement both. The expected result were two piplines or four new services. The RAs where provided with multiple documents containing technical documentation. For the server application three documents where provided. The first document `README.md` describes how to configure the SE's environment to build and execute the application. The second document `HOW_TO_CREATE_A_SERVICE.md` is a step-by-step guideline describing how to implement a new service. The third document `api.html` is a reference documentation for using the SyncPipes REST API. For the client application they where also provided with a `README.md` describing how to setup their environment to execute the client application. All these documents can also be found in the appendix. Besides technical documentation the project contained the source code of the four services described in Chapter 5. Besides for the evaluation irrelevant questions regarding the development environment, RA1 had issues understanding the format of the mapping information format, as a result the documentation regarding the mapping format is improved in the final version. Further the RA wanted to create automated tests using *Mocha*, this was not designated and therefore not documented. We discussed the matter and provided guidance for implementing basic tests. As a result the documentation will be supplemented with details about writing tests. Besides documentation the next iteration of the prototype should contain mock objects enabling to assert that the implemented service works as expected within the application [15, 18, pp.]. Further it would be useful for the DEV to have a test factory enabling testing without having to test with actual data. This can be further enhanced by integration it into the *gulp* process providing a continues testing approach.

While implementing the extractor service RA2 wanted to provide dynamic configuration values which depend on each other. An example would be a relational database system where the SA first selects a schema from a list and then a list of tables of schema is presented to the SA. After some discussion we agreed that this would be possible in general, but that is out of the scope of this work. The use case described by the RA also requires the application to be able to handle services with dynamic schemata. After discussing the issue with both RAs the proposed re-factored solution is to extend the `IService`'s `getSchema` to return a `Promise`. This enables the DEV to dynamically load a schema based on configuration parameters or other input parameters like the mapping. RA2 did a first implementation providing the described functionality. When implementing the services RA2 found minor bugs withing the data transformation module. After discussing the discovered issues we were able to fix the bugs enabling the RA to further implement the service. Besides issues related to the application both RAs had minor problems related to the asynchronous programming concept of *node.js* in conjunction with the use of recursion and loops. As a result future version should contain a document with pattern or idioms about working with mixture of asynchronous and synchronous *JavaScript* code.

The evaluation of the client application was conducted in the form of an open interview with the two RAs mentioned above. Some of the proposed changes resulted in findings from the server application evaluation. The first proposal by RA2 was that it is necessary to extend the mapping form with a second select box below the select box where the services are selected. The select box contains all available configurations for the above select service. This is necessary to support services with dynamic schemata depending on configuration parameters. Along with the proposal the RA implemented the functionality into the prototype. Resulting from proposal of more dynamic configurations, also the client application has to be changed. The component rendering the form for a service configuration based on its JSON-Schema has to improved to support custom JSON-Schema types and asynchronous loading of values present through a select box. The select box should also support to load these values based on another service configuration parameter select prior. As this feature is rather complex it was decided to postpone the implementation to future prototype. Another finding was the order of the items in the applications, both RAs where confused since the order of the items did not reflect the actual process of creating a new pipeline. Therefore the ordering of the navigation items has been changed to align with the logical order of the pipeline creation process. RA1 proposed to enhance all data-tables with filters for some columns. For example to be able to filter the pipeline executions table by a specific service, pipeline or status. The main issues noticed by both RAs was the usability of the mapping form. It was noticed that it should be possible to connect the properties of the source and the target system graphically e.g. thought drag and drop. Also it was mentioned that it would be easier for the end user to have the mapping form prefilled with all properties declared as required in the JSON-Schema. In addition to that it was mentioned there should be a possibility to validate the mapping before saving it. Syntactical validation of the mapping is already performed, but this is not sufficient for the end user to verify if the mapping works as expected. A possible solution to this would be a *dry run* mode in the form of a modal dialog where the data transformation is only simulated. The remarks about the mapping form need further discussion and are out of the scope of this work but should be implemented in some form in the next development iteration of the client application.

7. Conclusion

By analyzing literature and existing data integration tools it was shown that there is clearing a shortfall in lightweight extensible data integration tools for the SAKM domain. Implementing a first prototype seemed to be the obvious choice. Using JSON-Schema as foundation for the presentation of the domain models as well as foundation for the data integration algorithm is a new approach for the problem domain. This approach is certainly useful when integration data from or into services having a RESTful JSON API since most API are described using the JSON-Schema meta-model. The implemented application is designed as extensible framework allowing developers to quickly build new services which allow connecting the application to other system. By providing an REST API the application is integrable in almost every environment independently of the technologies around it. The use of *TypeScript* is a good balance between providing a wide range of typical object orient language constructs and while still having a lightweight and less error prone development platform (*node.js*). The evaluation results revealed the aspects, that should be addressed by the future work. In the current state of the application functionalities like updating data, creating associations (e.g. Foreign Key inserts) or removing orphaned data have to be implemented in each service by the developer. These functionalites should be implemented in the application's core to further reduce the developers effort. The current implementation of the data transformation algorithm supports only the most basic operations, this could be improved with features like aggregation or partitioning. A potential solution would be the introduction of a custom DSL or function which are applicable to each property mapping. Also the client application should be improved in the means of usability. A potential way to do this would be to conduct expert interviews.

Bibliography

[1] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, and M. Babar, "10 years of software architecture knowledge management: Practice and future", *Journal of systems and software*, vol. 116, pp. 191–205, 2016, cited By 0. DOI: 10.1016/j.jss.2015.08.054 (pages 1, 3).

[2] H. v. Vliet, "Software architecture knowledge management", in *19th australian conference on software engineering (aswec 2008)*, Mar. 2008, pp. 24–31. DOI: 10.1109/ASWEC.2008.4483186 (page 1).

[3] R. Weinreich and I. Groher, "A fresh look at codification approaches for sakm: A systematic literature review", *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)*, vol. 8627 LNCS, pp. 1–16, 2014, cited By 2. DOI: 10.1007/978-3-319-09970-5_1 (pages 1, 3).

[4] T. Knap, M. Kukhar, B. Macháč, P. Škoda, J. Tomeš, and J. Vojt, "Unifiedviews: An etl framework for sustainable rdf data processing", *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)*, vol. 8798, pp. 379–383, 2014, cited By 0. DOI: 10.1007/978-3-319-11955-7_52 (page 9).

[5] R. C. Martin. (2003). The Principles of OOD, [Online]. Available: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (visited on 05/25/2016) (page 10).

[6] V. Karpov. (2013). The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js, [Online]. Available: http://blog.mongodb.org/post/49262866911/the-mean-stack-mongodb-expressjs-angularjs-and (visited on 07/02/2016) (page 11).

[7] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”, Dissertation, University of California, Irvine, 2000 (page 12).

[8] Microsoft. (2016). TypeScript Website, [Online]. Available: https://www.typescriptlang.org/ (visited on 04/11/2016) (page 12).

[9] Ecma International, *ECMAScript 2015 Language Specification*, 6th. Geneva, 2015 (page 12).

[10] Microsoft. (2016). TypeScript Language Specification, Version 1.8, [Online]. Available: https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md#2-basic-concepts (visited on 06/06/2016) (page 13).

[11] OASIS. (2016). Amqp is the internet protocol for business messaging, [Online]. Available: http://www.amqp.org/about/what (visited on 06/06/2016) (page 14).

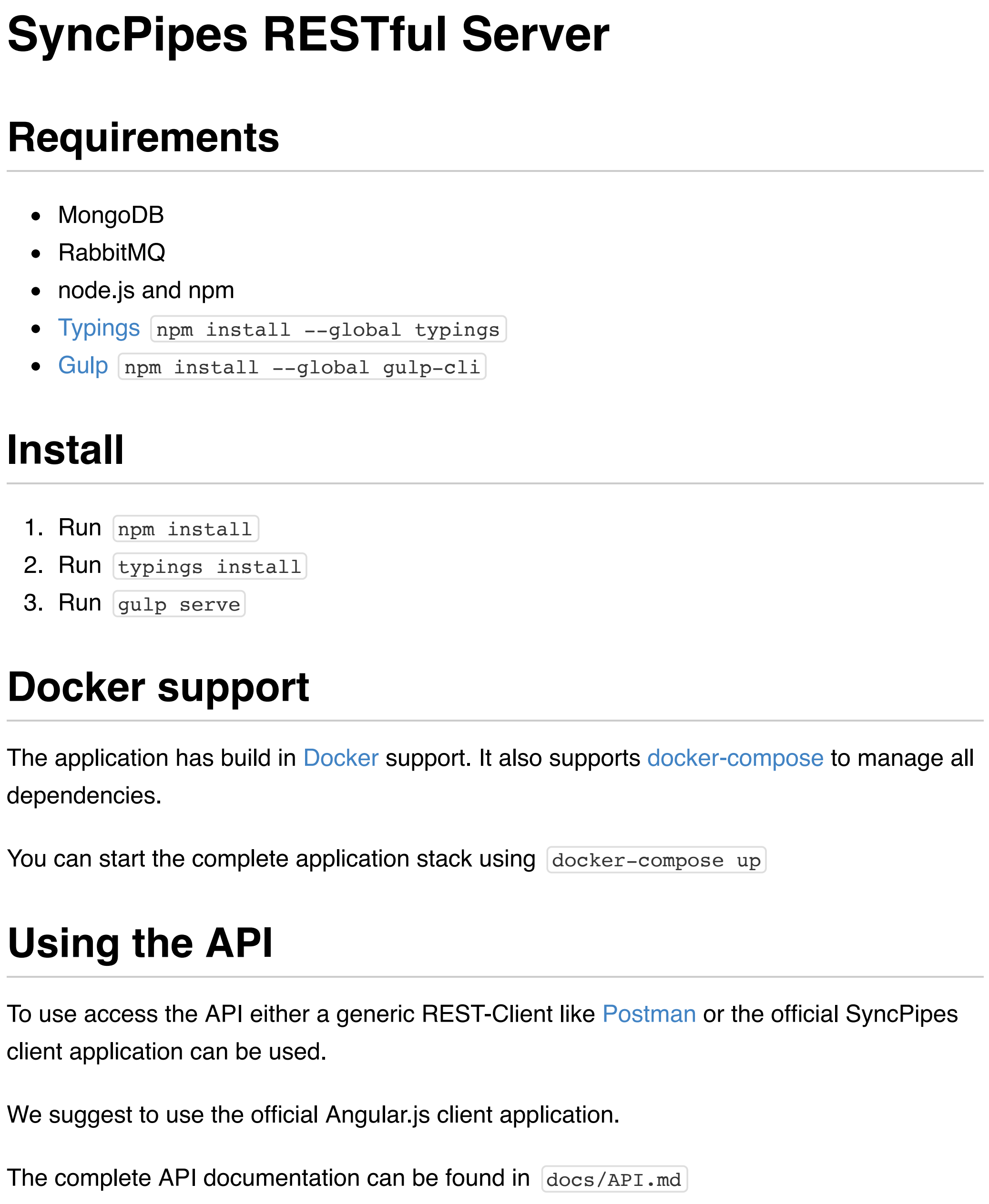
[12] M. Fowler. (2014). Presentation model, [Online]. Available: http://martinfowler.com/eaaDev/PresentationModel.html (visited on 06/26/2016) (page 15).

[13] J. Grossman. (2005). Introduction to Model/View/ViewModel pattern for building WPF apps, [Online]. Available: https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/ (visited on 06/26/2016) (page 15).

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0-201-63361-2 (page 42).

[15] S. Freeman and N. Pryce, *Growing object-oriented software, guided by tests*, 1st. Addison-Wesley Professional, 2009, ISBN: 0321503627, 9780321503626 (page 61).

A. Appendix



SyncPipes RESTful Server

Requirements

- MongoDB
- RabbitMQ
- node.js and npm
- Typings `npm install --global typings`
- Gulp `npm install --global gulp-cli`

Install

1. Run `npm install`
2. Run `typings install`
3. Run `gulp serve`

Docker support

The application has build in Docker support. It also supports docker-compose to manage all dependencies.

You can start the complete application stack using `docker-compose up`

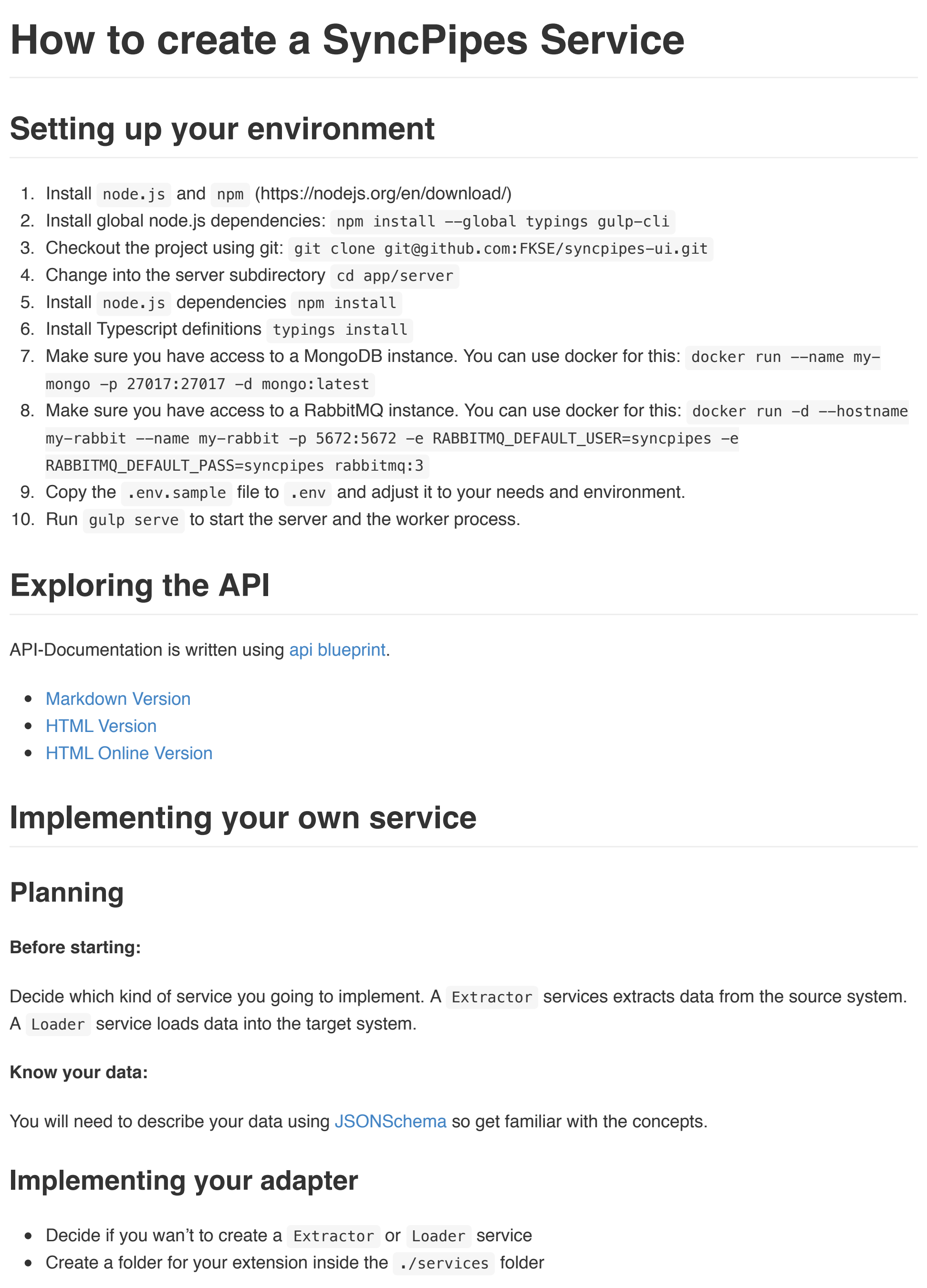
Using the API

To use access the API either a generic REST-Client like Postman or the official SyncPipes client application can be used.

We suggest to use the official Angular.js client application.

The complete API documentation can be found in `docs/API.md`

Figure A.1.: Readme file of the SyncPipes REST Server



How to create a SyncPipes Service

Setting up your environment

1. Install `node.js` and `npm` (https://nodejs.org/en/download/)
2. Install global node.js dependencies: `npm install --global typings gulp-cli`
3. Checkout the project using git: `git clone git@github.com:FKSE/syncpipes-ui.git`
4. Change into the server subdirectory `cd app/server`
5. Install `node.js` dependencies `npm install`
6. Install Typescript definitions `typings install`
7. Make sure you have access to a MongoDB instance. You can use docker for this: `docker run --name my-mongo -p 27017:27017 -d mongo:latest`
8. Make sure you have access to a RabbitMQ instance. You can use docker for this: `docker run -d --hostname my-rabbit --name my-rabbit -p 5672:5672 -e RABBITMQ_DEFAULT_USER=syncpipes -e RABBITMQ_DEFAULT_PASS=syncpipes rabbitmq:3`
9. Copy the `.env.sample` file to `.env` and adjust it to your needs and environment.
10. Run `gulp serve` to start the server and the worker process.

Exploring the API

API-Documentation is written using api blueprint.

- Markdown Version
- HTML Version
- HTML Online Version

Implementing your own service

Planning

Before starting:

Decide which kind of service you going to implement. A `Extractor` services extracts data from the source system. A `Loader` service loads data into the target system.

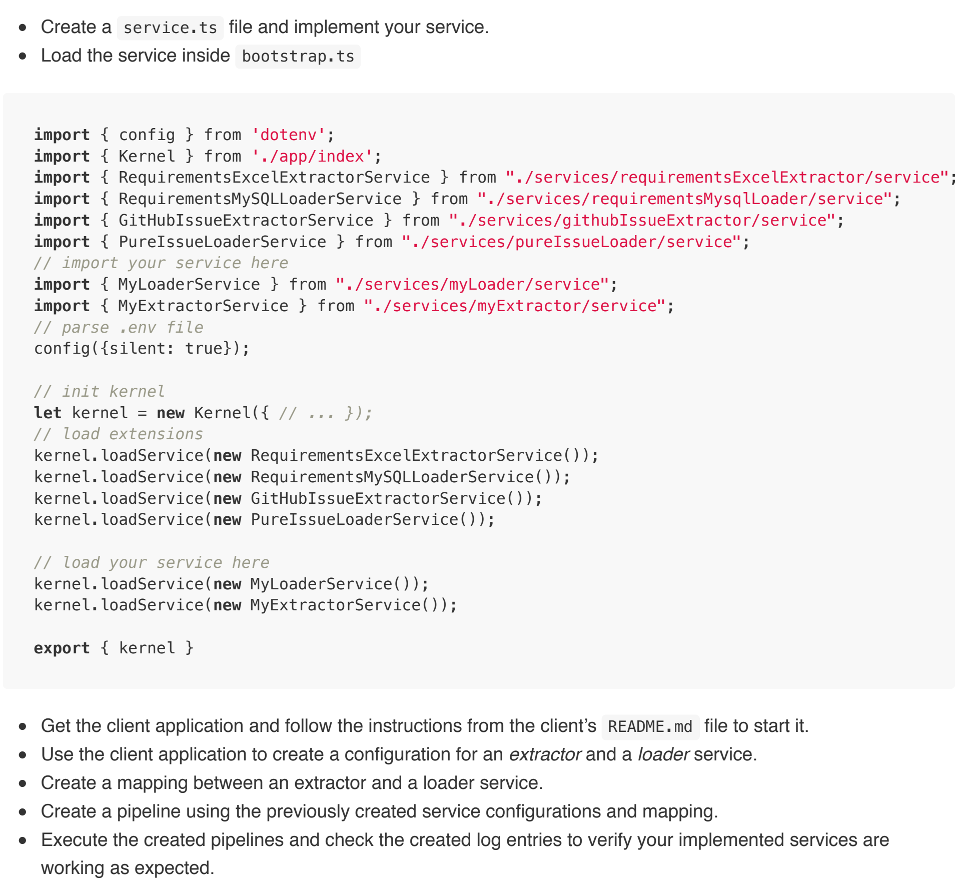
Know your data:

You will need to describe your data using JSONSchema so get familiar with the concepts.

Implementing your adapter

- Decide if you wan't to create a `Extractor` or `Loader` service
- Create a folder for your extension inside the `./services` folder

Figure A.2.: Guide for implementing a new service page 1

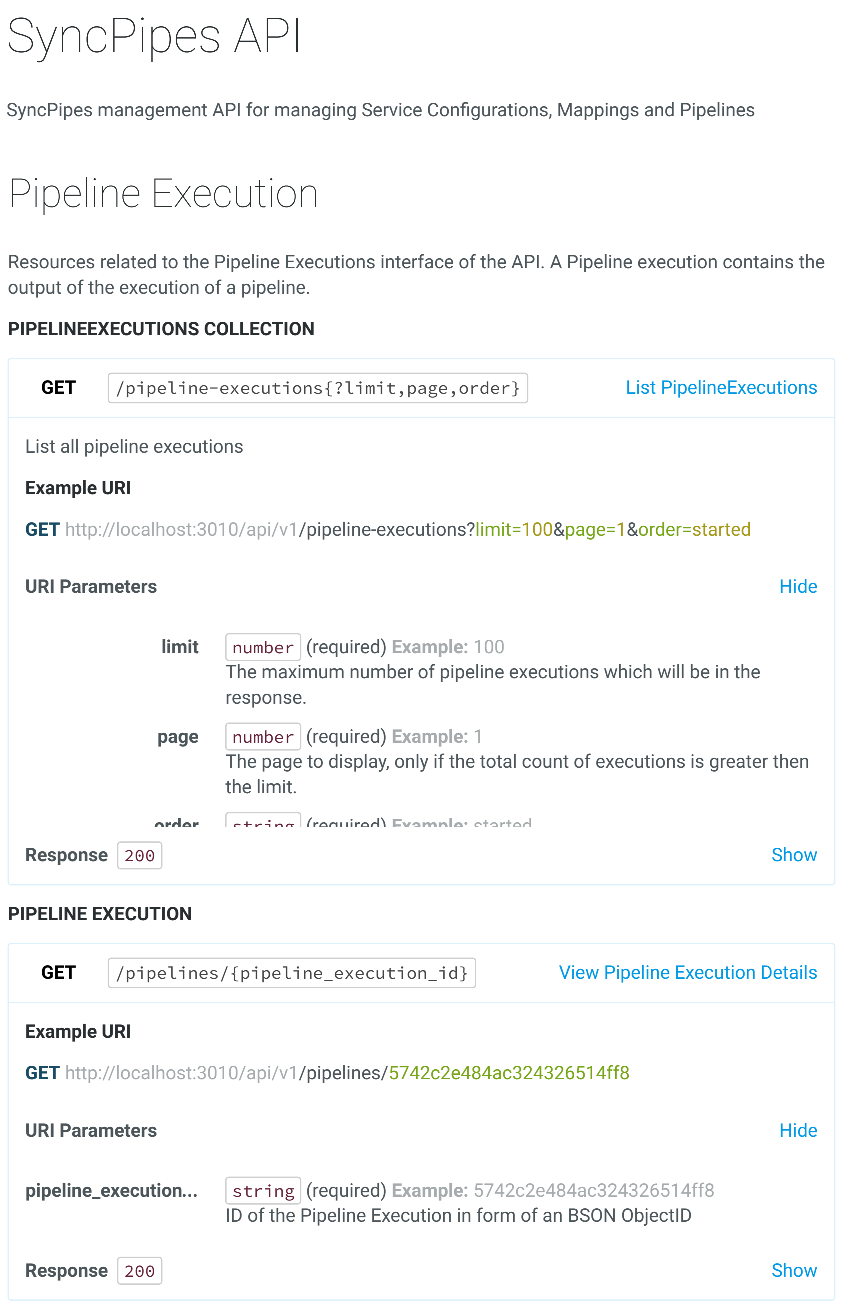


- Create a `service.ts` file and implement your service.
- Load the service inside `bootstrap.ts`

```
import { config } from 'dotenv';
import { Kernel } from './app/index';
import { RequirementsExcelExtractorService } from "./services/requirementsExcelExtractor/service";
import { RequirementsMySQLLoaderService } from "./services/requirementsMysqlLoader/service";
import { GitHubIssueExtractorService } from "./services/githubIssueExtractor/service";
import { PureIssueLoaderService } from "./services/pureIssueLoader/service";
// import your service here
import { MyLoaderService } from "./services/myLoader/service";
import { MyExtractorService } from "./services/myExtractor/service";
// parse .env file
config({silent: true});

// init kernel
let kernel = new Kernel({ // ... });
// load extensions
kernel.loadService(new RequirementsExcelExtractorService());
kernel.loadService(new RequirementsMySQLLoaderService());
kernel.loadService(new GitHubIssueExtractorService());
kernel.loadService(new PureIssueLoaderService());

// load your service here
kernel.loadService(new MyLoaderService());
kernel.loadService(new MyExtractorService());

export { kernel }
```

- Get the client application and follow the instructions from the client's `README.md` file to start it.
- Use the client application to create a configuration for an *extractor* and a *loader* service.
- Create a mapping between an extractor and a loader service.
- Create a pipeline using the previously created service configurations and mapping.
- Execute the created pipelines and check the created log entries to verify your implemented services are working as expected.

Figure A.3.: Guide for implementing a new service page 2



SyncPipes API

SyncPipes management API for managing Service Configurations, Mappings and Pipelines

Pipeline Execution

Resources related to the Pipeline Executions interface of the API. A Pipeline execution contains the output of the execution of a pipeline.

PIPELINEEXECUTIONS COLLECTION

GET `/pipeline-executions{?limit,page,order}` List PipelineExecutions

List all pipeline executions

Example URI

GET http://localhost:3010/api/v1/pipeline-executions?limit=100&page=1&order=started

URI Parameters Hide

limit	`number` (required) **Example:** 100 The maximum number of pipeline executions which will be in the response.
page	`number` (required) **Example:** 1 The page to display, only if the total count of executions is greater then the limit.
order	`string` (required) **Example:** started

Response `200` Show

PIPELINE EXECUTION

GET `/pipelines/{pipeline_execution_id}` View Pipeline Execution Details

Example URI

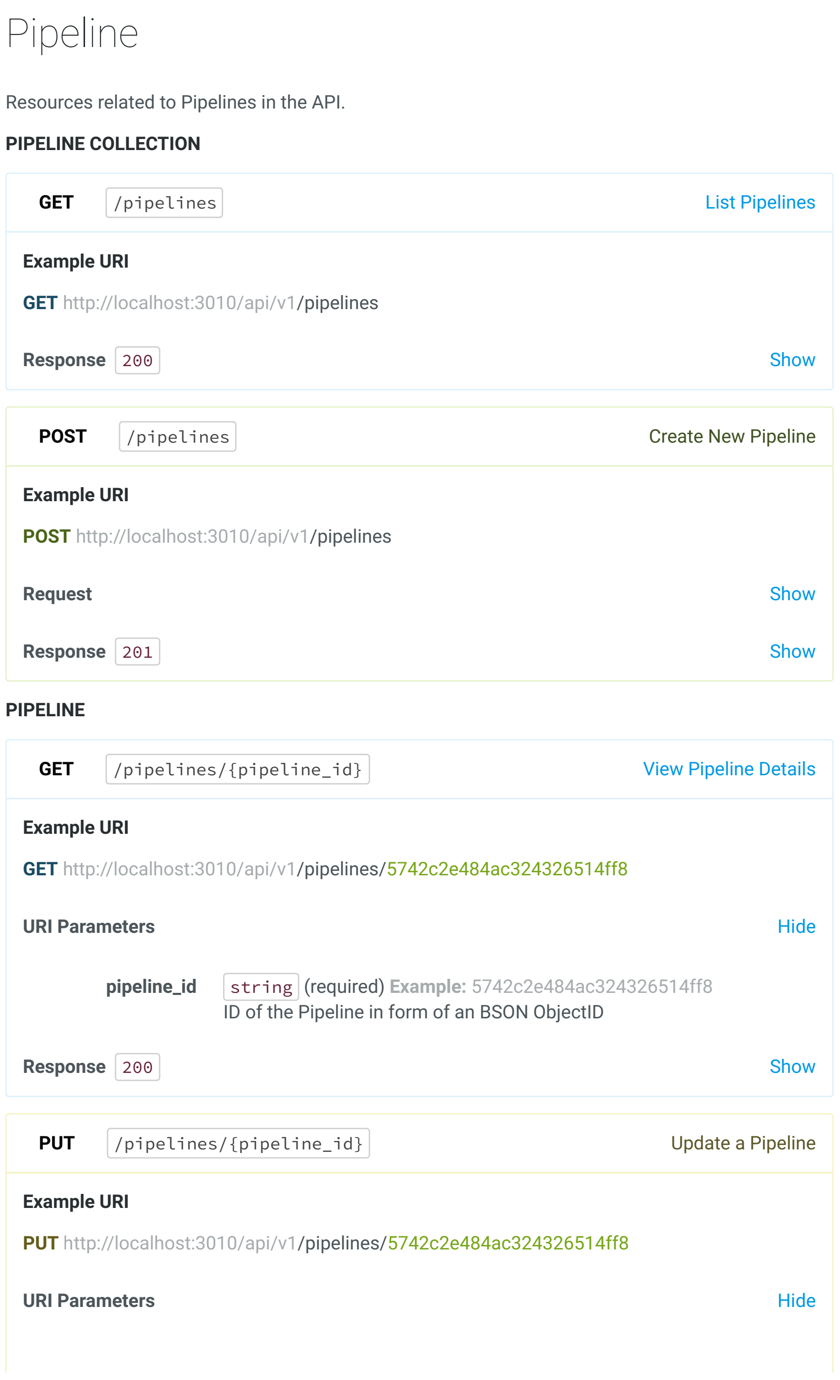
GET http://localhost:3010/api/v1/pipelines/5742c2e484ac324326514ff8

URI Parameters Hide

pipeline_execution...	`string` (required) **Example:** 5742c2e484ac324326514ff8 ID of the Pipeline Execution in form of an BSON ObjectID

Response `200` Show

Figure A.4.: SyncPipes REST API Documentation page 1



Pipeline

Resources related to Pipelines in the API.

PIPELINE COLLECTION

GET `/pipelines` List Pipelines

Example URI

GET http://localhost:3010/api/v1/pipelines

Response `200` Show

POST `/pipelines` Create New Pipeline

Example URI

POST http://localhost:3010/api/v1/pipelines

Request Show

Response `201` Show

PIPELINE

GET `/pipelines/{pipeline_id}` View Pipeline Details

Example URI

GET http://localhost:3010/api/v1/pipelines/5742c2e484ac324326514ff8

URI Parameters Hide

pipeline_id `string` (required) **Example:** 5742c2e484ac324326514ff8
ID of the Pipeline in form of an BSON ObjectID

Response `200` Show

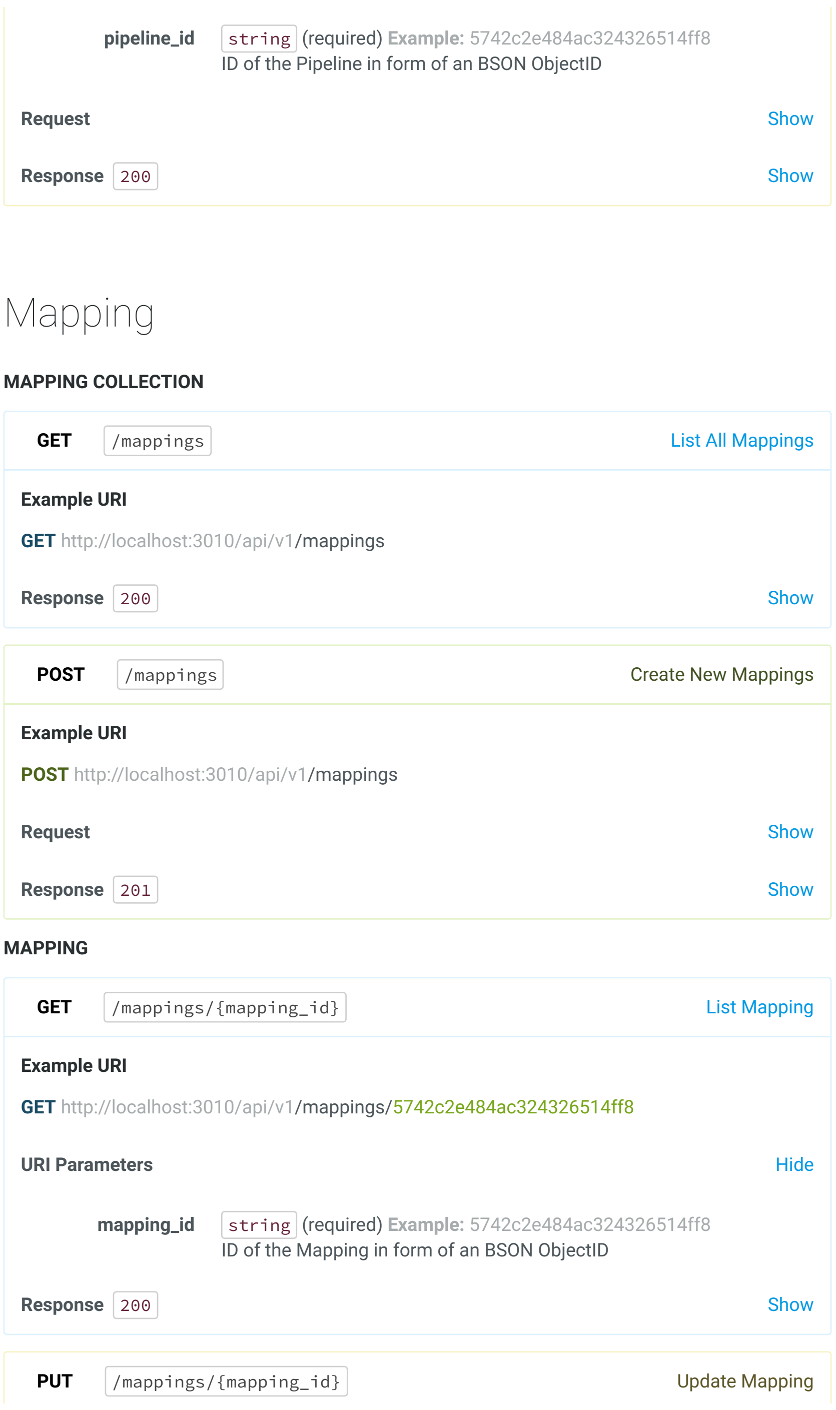
PUT `/pipelines/{pipeline_id}` Update a Pipeline

Example URI

PUT http://localhost:3010/api/v1/pipelines/5742c2e484ac324326514ff8

URI Parameters Hide

Figure A.5.: SyncPipes REST API Documentation page 2



pipeline_id `string` (required) **Example:** 5742c2e484ac324326514ff8
ID of the Pipeline in form of an BSON ObjectID

Request Show

Response `200` Show

Mapping

MAPPING COLLECTION

GET `/mappings` List All Mappings

Example URI

GET http://localhost:3010/api/v1/mappings

Response `200` Show

POST `/mappings` Create New Mappings

Example URI

POST http://localhost:3010/api/v1/mappings

Request Show

Response `201` Show

MAPPING

GET `/mappings/{mapping_id}` List Mapping

Example URI

GET http://localhost:3010/api/v1/mappings/5742c2e484ac324326514ff8

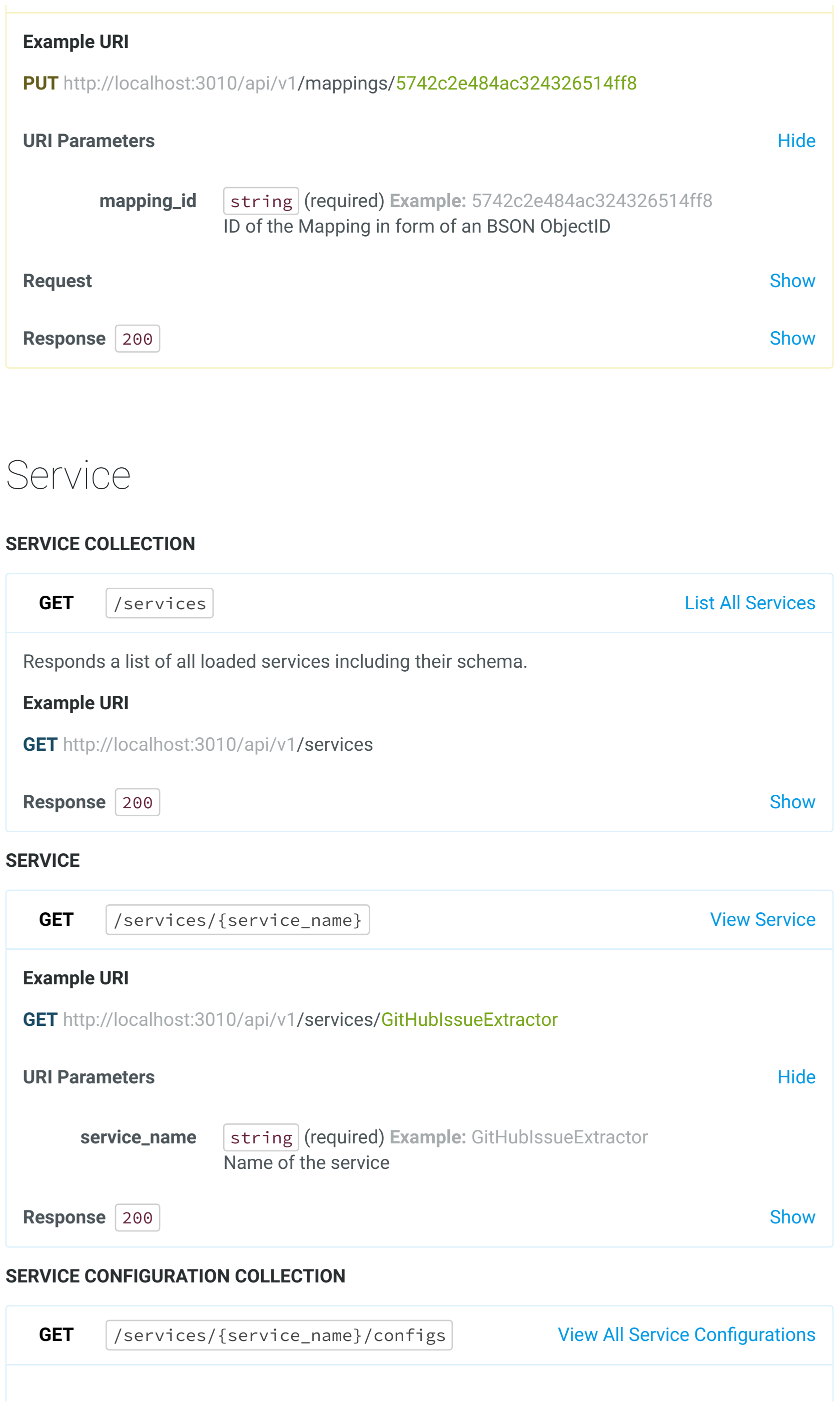
URI Parameters Hide

mapping_id `string` (required) **Example:** 5742c2e484ac324326514ff8
ID of the Mapping in form of an BSON ObjectID

Response `200` Show

PUT `/mappings/{mapping_id}` Update Mapping

Figure A.6.: SyncPipes REST API Documentation page 3



Example URI

PUT http://localhost:3010/api/v1/mappings/5742c2e484ac324326514ff8

URI Parameters Hide

mapping_id `string` (required) **Example:** 5742c2e484ac324326514ff8
ID of the Mapping in form of an BSON ObjectID

Request Show

Response `200` Show

Service

SERVICE COLLECTION

GET `/services` List All Services

Responds a list of all loaded services including their schema.

Example URI

GET http://localhost:3010/api/v1/services

Response `200` Show

SERVICE

GET `/services/{service_name}` View Service

Example URI

GET http://localhost:3010/api/v1/services/GitHubIssueExtractor

URI Parameters Hide

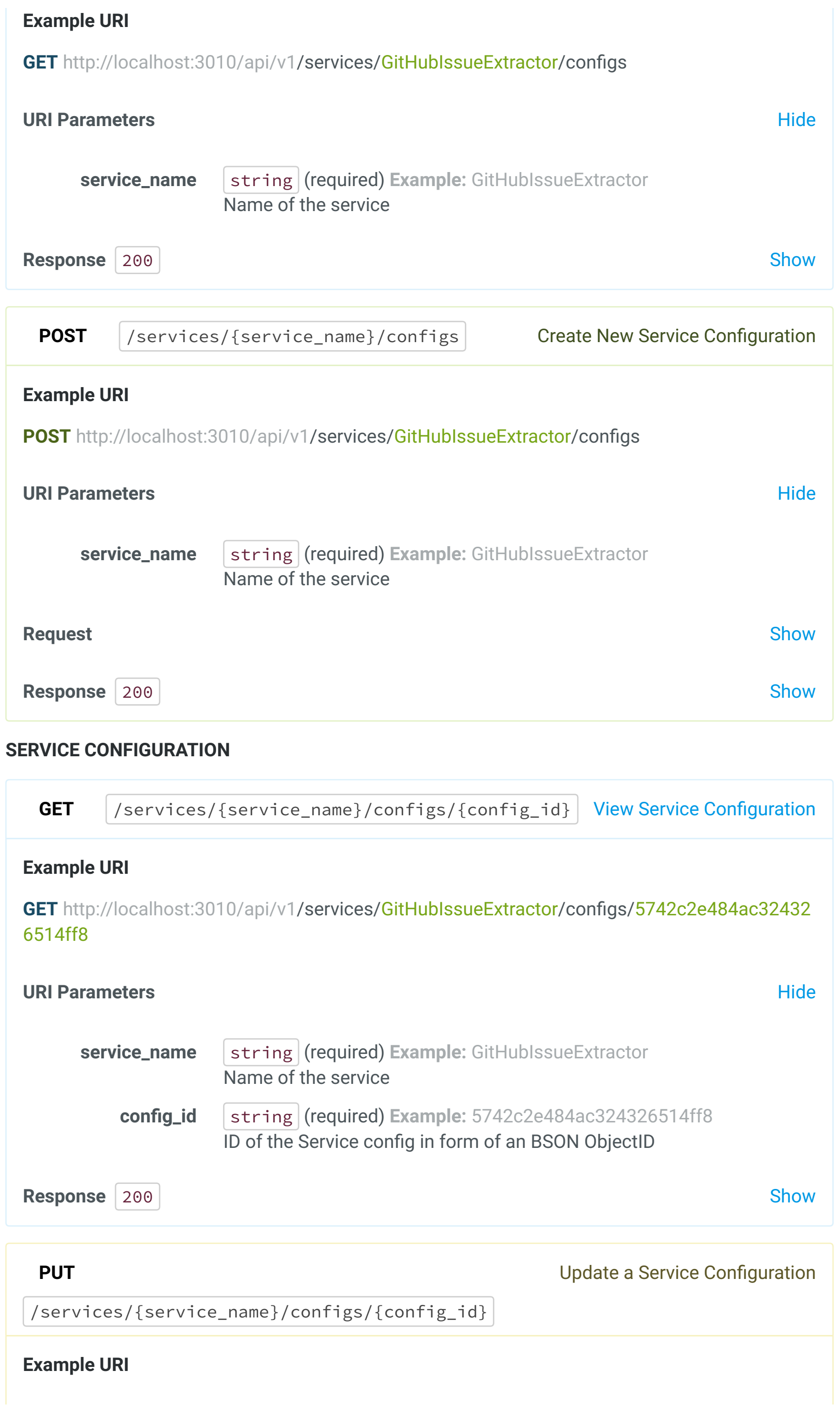
service_name `string` (required) **Example:** GitHubIssueExtractor
Name of the service

Response `200` Show

SERVICE CONFIGURATION COLLECTION

GET `/services/{service_name}/configs` View All Service Configurations

Figure A.7.: SyncPipes REST API Documentation page 4



Example URI

GET http://localhost:3010/api/v1/services/GitHubIssueExtractor/configs

URI Parameters Hide

service_name `string` (required) **Example:** GitHubIssueExtractor
Name of the service

Response `200` Show

POST `/services/{service_name}/configs` Create New Service Configuration

Example URI

POST http://localhost:3010/api/v1/services/GitHubIssueExtractor/configs

URI Parameters Hide

service_name `string` (required) **Example:** GitHubIssueExtractor
Name of the service

Request Show

Response `200` Show

SERVICE CONFIGURATION

GET `/services/{service_name}/configs/{config_id}` View Service Configuration

Example URI

GET http://localhost:3010/api/v1/services/GitHubIssueExtractor/configs/5742c2e484ac32432
6514ff8

URI Parameters Hide

service_name `string` (required) **Example:** GitHubIssueExtractor
Name of the service

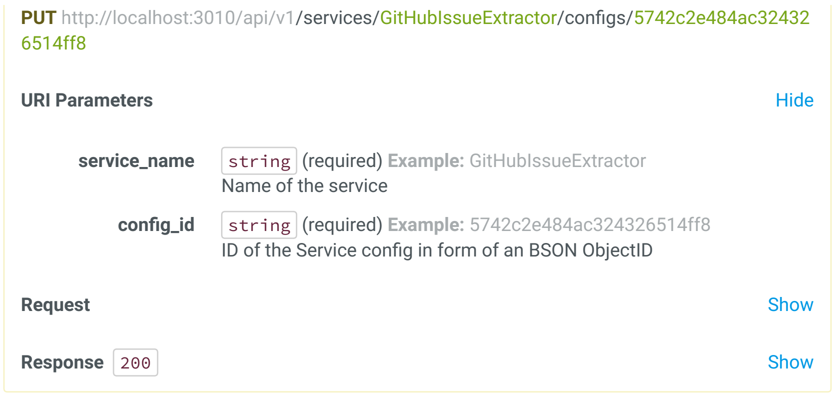
config_id `string` (required) **Example:** 5742c2e484ac324326514ff8
ID of the Service config in form of an BSON ObjectID

Response `200` Show

PUT Update a Service Configuration
`/services/{service_name}/configs/{config_id}`

Example URI

Figure A.8.: SyncPipes REST API Documentation page 5



PUT http://localhost:3010/api/v1/services/GitHubIssueExtractor/configs/5742c2e484ac324326514ff8

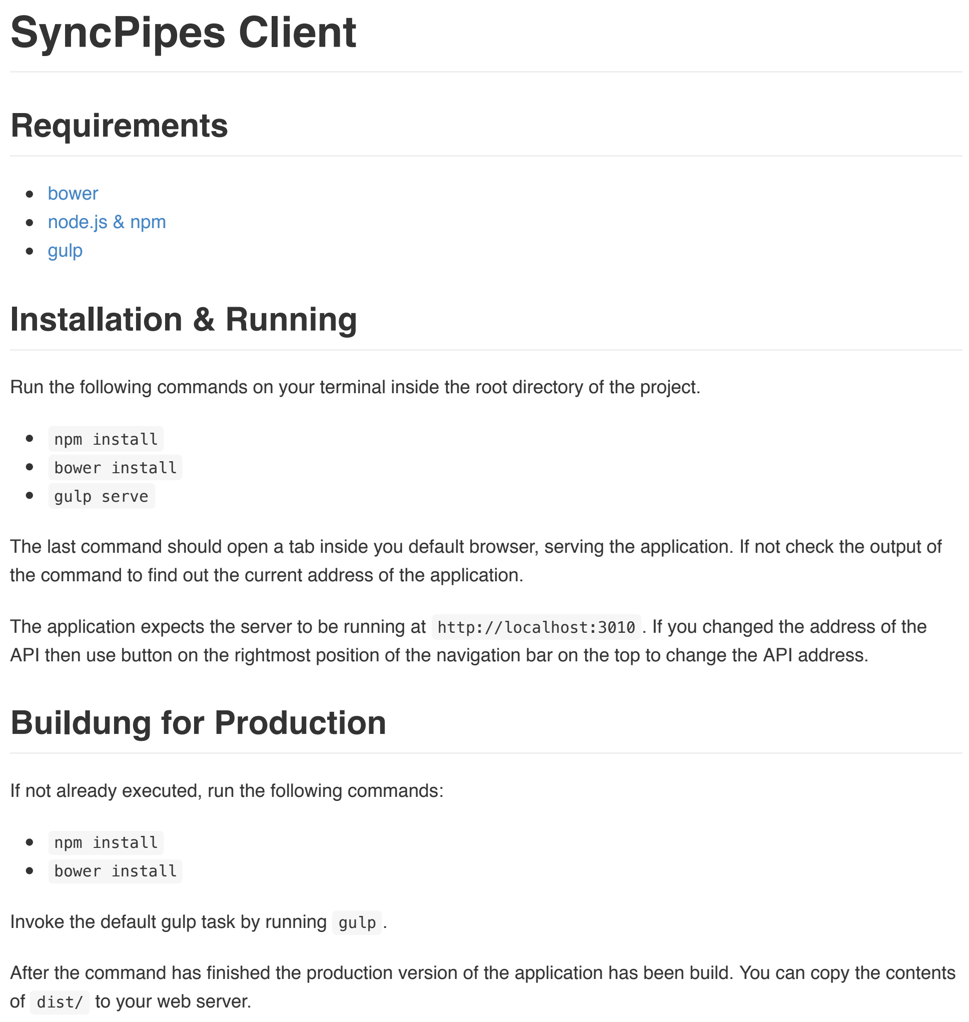
URI Parameters Hide

service_name	`string` (required) **Example:** GitHubIssueExtractor Name of the service
config_id	`string` (required) **Example:** 5742c2e484ac324326514ff8 ID of the Service config in form of an BSON ObjectID

Request Show

Response `200` Show

Figure A.9.: SyncPipes REST API Documentation page 6



SyncPipes Client

Requirements

- bower
- node.js & npm
- gulp

Installation & Running

Run the following commands on your terminal inside the root directory of the project.

- `npm install`
- `bower install`
- `gulp serve`

The last command should open a tab inside you default browser, serving the application. If not check the output of the command to find out the current address of the application.

The application expects the server to be running at `http://localhost:3010`. If you changed the address of the API then use button on the rightmost position of the navigation bar on the top to change the API address.

Buildung for Production

If not already executed, run the following commands:

- `npm install`
- `bower install`

Invoke the default gulp task by running `gulp`.

After the command has finished the production version of the application has been build. You can copy the contents of `dist/` to your web server.

Figure A.10.: README.md of the client application