# A web-based code editor using the Monto framework

**Ein web-basierter Code Editor unter Verwendung des Monto Frameworks**
Bachelor-Thesis von Wulf Pfeiffer
Tag der Einreichung: 9. Oktober 2015
1. Gutachten: Dr. rer. nat. Sebastian Erdweg
2. Gutachten: Prof. Dr.-Ing. Mira Mezini

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Software Technology Group

A web-based code editor using the Monto framework
Ein web-basierter Code Editor unter Verwendung des Monto Frameworks

Vorgelegte Bachelor-Thesis von Wulf Pfeiffer

1. Gutachten: Dr. rer. nat. Sebastian Erdweg
2. Gutachten: Prof. Dr.-Ing. Mira Mezini

Tag der Einreichung: 9. Oktober 2015

## Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 9. Oktober 2015

(Wulf Pfeiffer)

## Contents

# List of Figures

## Abstract

Code editors and *Integrated Development Environments* (IDEs) are widely used tools for developers to improve coding productivity. They help visualizing code by highlighting specific keywords, show warnings about erroneous code and increase working speed by providing suggestions or completions, while writing. Often, such programs also have a *Application Programming Interface* (API) to enable users the development of plug-ins for extension. However, those plug-ins are not interchangeable through several different editors or IDEs, leading to several reimplementations for the same purpose. This also results in plug-ins with different quality.

To overcome this problem, a framework, called *Monto*, was created by *Sloane, et al.* [36]. The framework consists of a broker and services for several languages with different tasks. To use the services, an IDE must implement a plug-in that enables it to communicate with the broker and to process products received by the services. This leads to only one implementation for, for example, a new programming language as a new service for *Monto*, instead of an implementation for each IDE. Every IDE that uses the *Monto* framework can then access the service. *Keidel* introduced some standardized products and a broker [22] that keeps state of services to manage dependencies between each other in his Master Thesis [35]. He also implemented several new services for the programming language *Java* and developed a *Monto* plug-in for the IDE called *Eclipse* [23].

In the present thesis, the framework is even further improved and extended by creating a web-based code editor that uses *Monto*, implementing several services for *JavaScript* and designing a new *errors* product type for services. Also the functionality of registering services to the broker, discovering available services and configuring services and the broker through clients are added to the framework.

# 1 Introduction

Developers nowadays often write code using an code editor or an IDE to improve their effectiveness and to reduce syntactical and semantical mistakes.

A code editor typically achieves this by providing instruments like syntax highlighting or code completion for several programming languages. An IDE though, does not only consist of a code editor and several other tools, that support the developer, but also often contains toolchains with compilers and debuggers to run and analyze a project within the IDE. This significantly improves the process of development. As IDEs usually contain a code editor, in the following the term IDE stands for both, a code editor and an IDE.

Most IDEs also provide an API to enable its community to create plug-ins for further improvements of the IDE. A reason to implement a new plug-in would be, for example, a new programming language that should be supported. However, there exist a large amount of different programming languages and not every IDE provides the same support for all of them. Also it is rarely the case that a plug-in, that is written for one IDE, can also be used for another one.

Therefore the developers or people from the community of such programming tools have to reimplement support for each language, which should be usable. This has to be done for every IDE that should support the language. For $m$ programming languages and $n$ IDEs, this leads to $m * n$ implementations and yet each IDE would not have the same quality of support for a programming language.

To address this problem, *Sloane, et al.* [36] came up with an approach to outsource tools into services so that each service can be accessed by every IDE. This leads to $m + n$ implementations for $m$ programming languages and $n$ IDEs. This approach was realized by *Sloane, et al.* in a project called *Monto* [20]. *Monto* is a framework that consists of services, which each provide a tool for a programming language, and a broker that IDEs can communicate with to access the services. *Sloane, et al.* call this a *Disintegrated Development Environment* (DDE).

There are services that only depend on the source code to work with, but also others that additionally require output of other services. The initial *Monto* project focuses on a simple, easy to implement and stateless broker that does not care about such dependencies and in essence just forwards source code and resulting service output to other services and clients. On the other hand, services have to be stateful if they require another service and have to wait to receive all dependencies until they can complete their task. This might end in services that are blocked forever if they do not receive every dependency. Also implementing a new service is a non-trivial task as the developer has to implement state if the service has more dependencies than the source itself.

For those reasons *Keidel* modified the *Monto* framework in his Master Thesis [35] to invert the responsibility of dependencies. In his version of *Monto* the services are stateless and do not take care of their dependencies because the broker is stateful and only sends a complete package of all required dependencies to the service when everything is available.

Right now clients are not aware of which services are available through the broker and therefore don't know which languages and what kind of services they can use. Hence, clients receive output from every available service and have to sort out everything that they do not need. Also some services would be able to provide more specialized output through configuration, but right now there is no way to do so through the client, as no information about available services and their possible configuration options is accessible.

## 1.1 Scope of Work

Within the scope of this bachelor thesis a web-based editor, that uses the *Monto* framework components, has to be implemented by using an existing *JavaScript* code editor framework. Also a new set of services for the programming language *JavaScript* is created for existing service types that were designed and implemented earlier by *Keidel* for *Java*. To extend the repertoire, also a new product type for services that detect errors is designed. Furthermore the issues discussed in the introduction should be addressed by extending the broker and, if required, the services to enable the client to discover and configure services, but also to configure the broker by providing the possibility to select only required services.

## 1.2 Overview of the Thesis

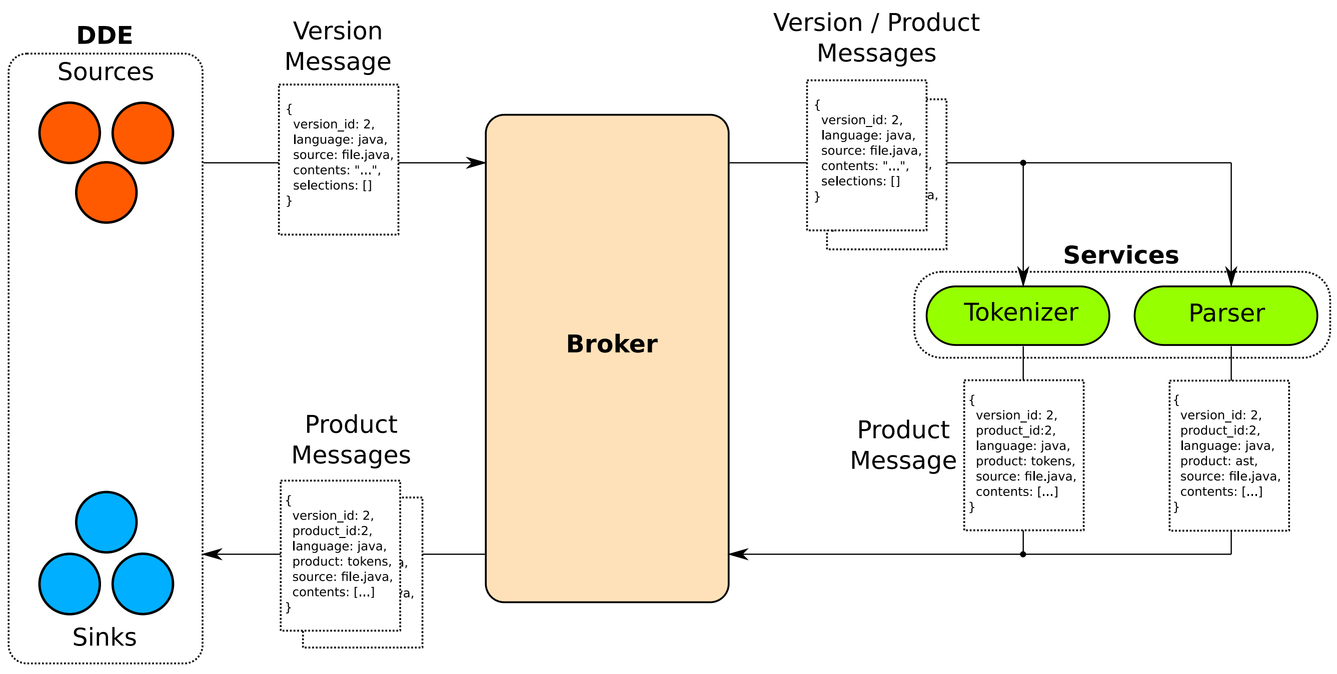
In the next chapter, a more detailed introduction into the *Monto* framework is given to provide a fundamental and solid knowledge of the architecture and implementation. The 3rd chapter describes the process of designing and implementing the web-based editor. In chapter 4 the implementation of *JavaScript* services and a new product type, including a new service, are discussed. Chapter 5 discusses the discoverability of services and their registration to the broker, whereas the 6th chapter contains information about the selection and configuration of services. The evaluation of the previous chapters can be found in chapter 7. Information about related work to the topic of this thesis can be read in chapter 8. In the last chapter a conclusion of the work of this bachelor thesis is drawn and discussed.

# 2 Background: Fundamentals of Monto

This chapter gives a brief summary of the current state of the *Monto* framework modified by *Keidel*. It describes the basic architecture and explains the tasks of its components and how they behave to understand the further content of this bachelor thesis. More details can be found in *Keidels* Master Thesis [35] and information about the original version of *Sloane, et al.* can be found in the paper *Monto: A Disintegrated Development Environment* [36].

## 2.1 Architecture of Monto

*Monto* is a framework that consists of several different parts that together can be used by an IDE that implements a *Monto* plug-in. There are four major parts: one broker, several sources, sinks and services. Sources are client side and send source code in a so called *Version Message* to the broker. The broker then sends the *Version Message* to services that have no other dependencies and gathers the output of those services, called *Product Message*. Then the broker sends *Version* and *Product Messages* to services with these dependencies and repeats this procedure until all services were called. In parallel, the broker also sends each *Product Message* that it receives to the sinks. The sinks are also client side and their task is to process and/or visualize the received products. The architecture can be seen in figure 2.1. A more detailed description to each component can be found in the following chapters.



```
{
 version_id: 2,
 language: java,
 source: file.java,
 contents: "...",
 selections: []
}
```

```
{
 version_id: 2,
 product_id:2,
 language: java,
 product: tokens,
 source: file.java,
 contents: [...]
}
```

```
{
 version_id: 2,
 product_id:2,
 language: java,
 product: ast,
 source: file.java,
 contents: [...]
}
```

**Figure 2.1.:** *Monto* Architecture

The complete communication is done by the middleware framework called *ZeroMQ* (ZMQ) [32]. It provides an easy way for communication, several different message patterns and is available for a vast amount of programming languages. All messages sent are in *JavaScript Object Notation* (JSON) format [18]. JSON is a simple but effective format to send messages between two parties in both, a human and machine readable manner.

## 2.2 Monto Sources

Sources are client side interfaces that communicate with the broker. The communication between them and the broker is done via the *publish-subscribe* pattern [34], where the sources are the publishers and the broker is the subscriber. In *publish-subscribe*, publishers can publish messages to topics, which subscribers can subscribe to. A subscriber will only receive messages from the topics it is subscribed to. In the original version of *Monto*, no topic is used for publishing and subscribing. Sources send *Version Messages* to the broker when the source code changes. Often, a source represents a source code file tab in an IDE.

```
{
  "source": "test.js",
  "version_id": 1,
  "language": "javascript",
  "contents": "console.log(\"test\");",
  "selections": []
}
```

**Listing 2.1:** Example of a *Version Message*

The *Version Message* contains several information about the source code that should be processed by the services. To keep track which *Product Messages* belongs to which *Version Message*, a *version id* exists. This id is increased with every *Version Message* that is sent by a source. In the *contents* attribute, the source code itself is transported, whereas in the attribute called *selections* a position within the text is given that for example can be used by a code completion service to determine which text should be completed. The *Version Message* also contains the name of the source code file in the *source* attribute and the used programming language in the *language* attribute. An example *Version Message* can be seen in listing 2.1 with all the discussed fields.

## 2.3 The Monto Broker

The broker is server side and as already mentioned, is a subscriber to *Version Messages* from the sources. It implements a state machine that has the capability of modeling service dependencies. It ensures that each service receives the *Version* and *Product Messages* that belong together and which the service depends on. The state machine is built for the given set of services at startup.

When the broker receives *Version Messages* from the sources it determines through its state machine which services have no dependency and sends the *Version Message* directly to those services through a ZMQ pair connection, which allows communication in both directions. After a short period of time the broker will receive the *Product Messages* from the services and publish these to the sinks. As the broker is a subscriber to soures, it is a publisher to sinks. Then, the broker will determine which services have one or more of those products as dependencies and will send the *Version Message* along with the products to those services. This procedure is repeated until all services, which have dependencies, have received the required messages.

```
{
  "product": "outline",
  "contents": [
    {
      "identifier": {
        "offset": 0,
        "length": 20
      },
      "description": "program"
    },
  ],
  "product_id": 1,
  "language": "javascript",
  "version_id": 1,
  "source": "test.js",
  "dependencies": [
    {
      "product": "ast",
      "product_id": 1,
      "language": "javascript",
      "tag": "product",
      "version_id": 1,
      "source": "nofile"
    }
  ]
}
```

**Listing 2.2:** Example of a *Product Message* from an *outline* service

A *Product Message* contains information about the processed output of a service. The message has several attributes like *product* which determines the product type, the *product id* field and the actual *contents* of the product, which differs from type to type. Also some attributes that contain information form the *Version Message* are included. Such fields are the *language* field, the *version id* and the *source* field. There is also an attribute called *dependencies* in which dependency *Product Messages* are listed, that were used by the services, but excluding the *contents* attribute. Listing 2.2 shows an example *Product Message*.

## 2.4 Services for Monto

Services are server side and communicate with the broker over a ZMQ pair connection. They specify a programming language they can handle, a product type they produce and several dependencies that they require to function properly. A service receives *Version Messages* and, if any dependencies are given, corresponding *Product Messages* from the broker. After receiving those, the service processes them and computes the contents for a *Product Message* that, on completion, is sent back to the broker. An example for a service could be a tokenizer, that produces tokens out of the source code for the *Java* language. Currently, there are some product types defined and also implemented for *Java*. These products are *tokens* which can be used for syntax highlighting, *ast* that models an abstract syntax tree and is commonly used for others services and *outline* as well as *completions* that both are self-explanatory.

## 2.5 Monto Sinks

A sink is, like a source, client side, however they have the opposite task as sources. The broker communicates with the services to get *Product Messages* and those *Product Messages* are published without a topic so that each subscribed sink can receive those *Product Messages*. On reception a sink has to determine if the product should be processed by for example checking the version id, product type or language of the message. If the sink decides to process the product further it will visualize its contents or also perform other tasks on the product. An example could be a token product that is received by a sink. The sink then will colorize the source code according to the contents of the product and therefore provide syntax highlighting. A source and a sink can appear as a pair for a file tab in an IDE, so that the source sends changes of the source code to the broker and the sink then processes the contents of the *Product Messages* to visualize the impact of the changes made to the source code.

# 3 Web-based Code Editor

The web-based code editor was developed in scope of this bachelor thesis. It is presented in this chapter, which contains information about the concept, design, implementation, visualization and also about modifications to the *Monto* framework that had to be made. The code can be found on *GitHub* [29].

## 3.1 Concept and Development

Before implementing software, the requirements and boundaries of the requested product have to be identified. To do so, several meetings with the supervisor of this thesis, *Dr. Erdweg*, had taken place where the author interviewed *Dr. Erdweg*. Within these interviews, some rough requirements and possible boundaries were surveyed. As meetings were regular scheduled on a weekly basis to inspect progress, a prototypical and agile development was suitable and advisable and therefore chosen as the development method. In those meetings, *Dr. Erdweg* was informed about the current state of the project, occuring problems, their possible solutions and was shown the working parts. Also, possible deficits were discussed and new or changed concrete requirements for the next steps were elaborated. Hence, the correct execution of requirements was given and the quality of the overall result of the project was ensured.

## 3.2 Websockets

A web application, also called web app, is a server-client based application, whose client runs in the browser and is built using *Hypertext Markup Language* (HTML), *Cascading Style Sheets* (CSS) and *JavaScript*. The server can be just a simple webserver, but also a more advanced back-end that can perform several, more complex tasks.

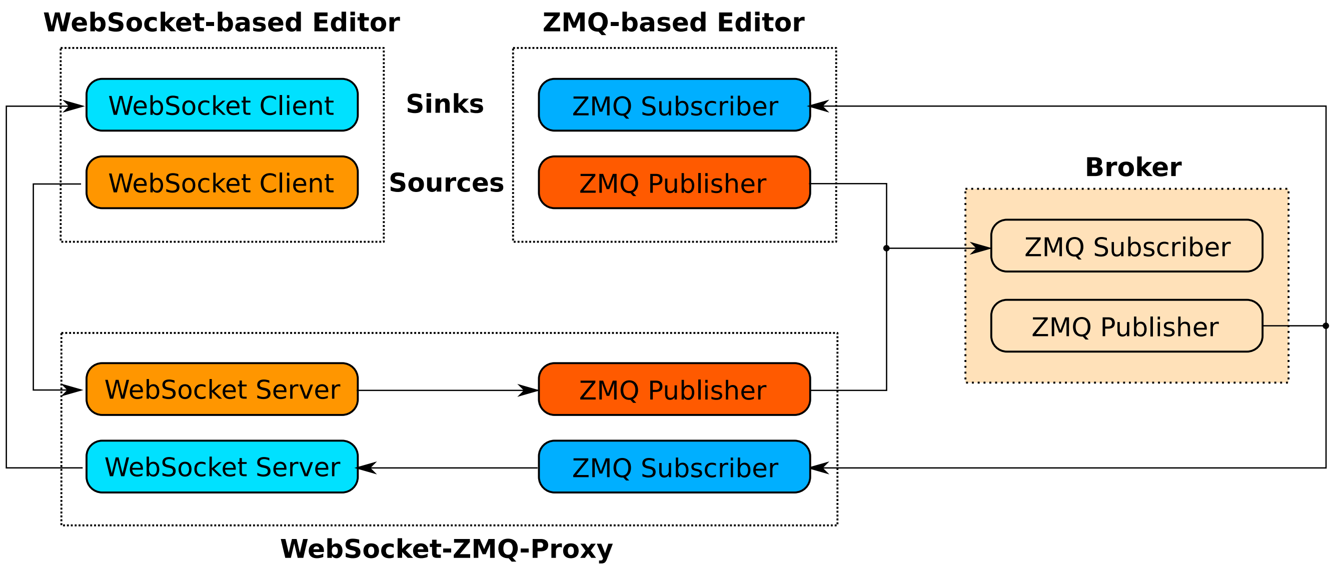


**Figure 3.1.:** WebSocket-ZeroMQ-Proxy

The web-based code editor is also part of a web app and represents the client. The *Monto* framework, in this case, is the back-end. The editors logic is implemented in *JavaScript* which leads to one major problem: The *W3C* has not yet released a final standard for a *TCP* and *UDP* sockets API [15], therefore no browser has yet implemented it. Some browsers like *Google Chrome* or *Mozilla Firefox* have an API for *TCP* sockets, but they are not standardized and hence not suitable for this project as it would require a specific browser or some complex work to provide support for different browsers. *Monto* uses ZMQ and to communicate, a *TCP* or *UDP* connection is required. To overcome this problem, a standardized web technology by *W3C* called *WebSockets* can be used. WebSockets use an underlying *TCP* connection to connect a server and a client and can be used by all current major browsers [31]. ZMQ right now does not support connections over WebSockets but there is a raw specification to make it possible [33]. However, this approach is not suitable as right now it is not part of the ZMQ standard and also only a binding for *C#* is available. As the broker is implemented in *Haskell*, there is no binding available and the WebSocket approach can't be used.

To make WebSockets available to the broker, so that web apps can communicate with it, two possibilities are available. The first option is to implement a WebSocket server into the broker, the second to develop a program that has a WebSocket and a ZMQ part. The second option can be seen in figure 3.1 and is called the *WebSocket-ZeroMQ-Proxy* (WS-ZMQ-Proxy). This program just forwards messages from the WebSocket server to the ZMQ publisher and from the ZMQ subscribes to the WebSocket server. Both approaches were implemented prototypically and tested by the author, which lead to the conclusion that the first approach is more complex and didn't work out well in the implementation due to different behaviour of the WebSocket and ZMQ connections in *Haskell*. Especially reconnection or multiple connections over WebSockets were significantly different from the way ZMQ works. Also the second option abstracts the connection over a different protocol away from the broker, keeping the broker more simple and lightweight and the *Monto* project more modular. In addition, no noticeable performance impact was detected due to the intermediate proxy for WebSocket and ZMQ. Hence, the second approach with the WS-ZMQ-Proxy was used for the addition of WebSockets to the *Monto* project.

## 3.3 Software Design

Implementing a new code editor is a non-trivial task that is not feasible to do in the scope of this bachelor thesis. Hence, an existing *JavaScript* editor framework was used to design and implement a web-based editor for the *Monto* framework. For this purpose two different frameworks were considered: *Ace* [1] and *CodeMirror* [8]. Both frameworks are quite similar in the manner how they work and both are used by some large, well known projects. The decision was made to use *CodeMirror* as the API seemed to be easier and to provide more possibilities and also *CodeMirror* seemed better documented than *Ace*.

To use *CodeMirror* in the browser, one must only create a text area in HTML and give it an ID. With *JavaScript* the ID can be used to call a function that creates a new *CodeMirror* instance in the text area. *CodeMirror* defines so called modes to extend support for different programming languages. Those modes are used to define rules for syntax highlighting for a language and are called every time the content of the editor area is changed.

To enable the editor to process *Product Messages*, a new *Monto* mode was implemented. The *Monto* mode is responsible for creating syntax highlighting and outlining. The outlines are represented by a unordered list in HTML that is contained in another panel. *CodeMirror* provides the possibility to implement helpers that perform a specific action when they are called. There are several types that can be used to define when a helper is called. Such a type is called *hint*. This type is used for code completion and a new *hinter* was implemented called *monto-hinter*. When writing source code, one might also write erroneous code that can be detected and visualized by an editor. *CodeMirror* also provides a helper type for this called *lint*. Again, a new *lint* helper called *monto-lint* is created.

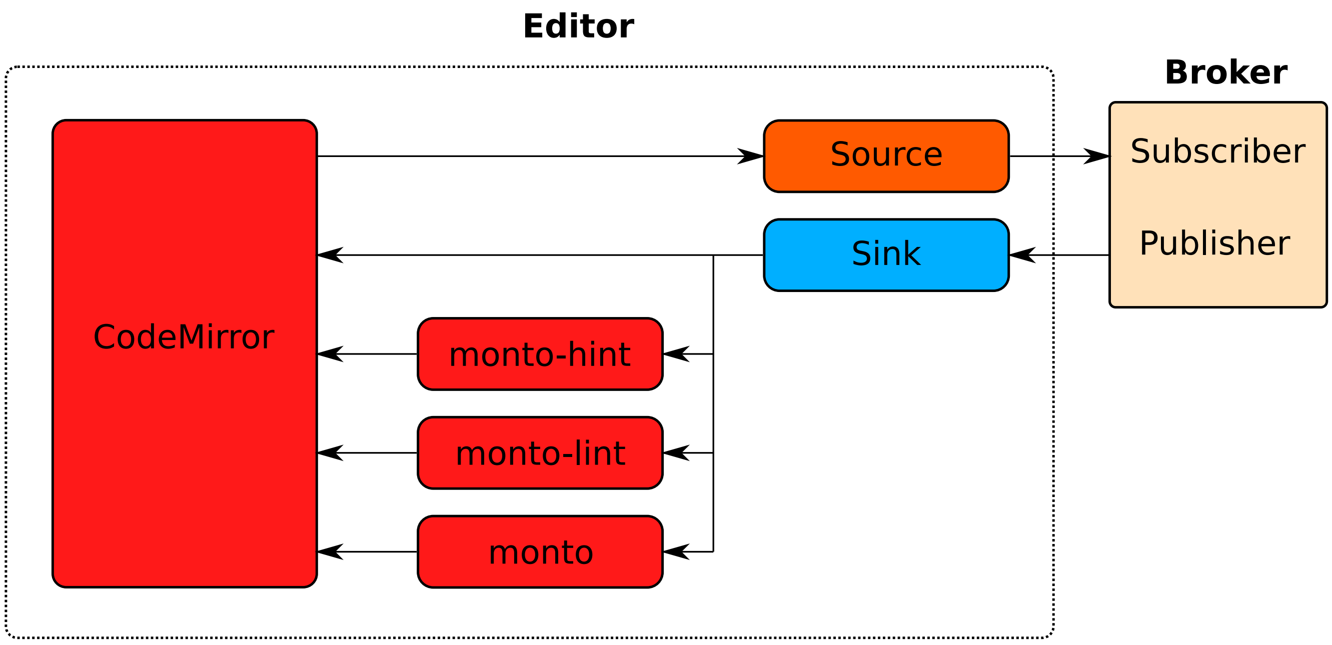


**Figure 3.2.:** Web-based Editor Design

These modes and helpers only provide functionality to process *Product Messages* that are received from the broker. But to receive products from the broker a connection has to be established, and since a web-based editor is build, the connection will be made over WebSockets and the WS-ZMQ-Proxy. Those connections are established by a source and a sink object, one to send *Version* and one to receive *Product Messages*. They are equivalent to the sources and sinks that were defined in chapter 2. When the text in the editor is changed, the source sends a *Version Message* to the broker and the sink then receives the *Product Messages* and invokes the corresponding actions to the product type. The design can be seen in figure 3.2.

*CodeMirror* is not the only framework or library that was used to create the web-based editor. For visual purposes and easier handling of different screen sizes the *Bootstrap* framework [6] was used. Not only it provides a polished look by the usage of CSS, but also some useful functionalities like tabs that are enabled via *JavaScript*. Also *Bootstrap* can be used to write websites that are usable as both, desktop and mobile applications.

*Bootstrap* depends on another *JavaScript* library called *jQuery* [17]. The *jQuery* library provides easier access to *Document Object Model* (DOM) elements in the browser and some other useful functions. It is widely used and even if it would not be a dependency of *Bootstrap* it would have been used in this project.

For saving modified editor contents to the local storage, another small library was used called *FileSaver.js* [11]. It provides an easy way of downloading the editors content. In fact there is no real download as the editor runs local in the browser, but for saving purposes this library calls the download dialog so that you can chose the destination where the new file should be saved.

*Bootstrap* does not provide CSS classes for checkboxes and radio buttons for beautification, hence an add-on called *Awesome Bootstrap Checkbox* [5] was used to have a more consistent look. This add-on however also requires the framework *Font Awesome* [13] that provides icons.

## 3.4 Implementation of the Editor

The source and sink are in fact JSON objects that are created through a function, which is called once the *JavaScript* file is executed. One could say that this is similar to the *singleton* pattern [37] and creates only one instance of this object. The function that creates these objects is a closure function. With the help of closures it is possible to use private variables, that are only visible within the object. The closure returns a JSON object that defines functions and variables in its attributes. The source object provides public functions to change content and send *Version Messages*, whereas the sink objects offers functions to access the *Product Messages* and also to register a new function to the sink, that is called each time it receives a new *Product Message* from the broker. Both objects operate on a WebSocket client connection to the WebSocket proxy.

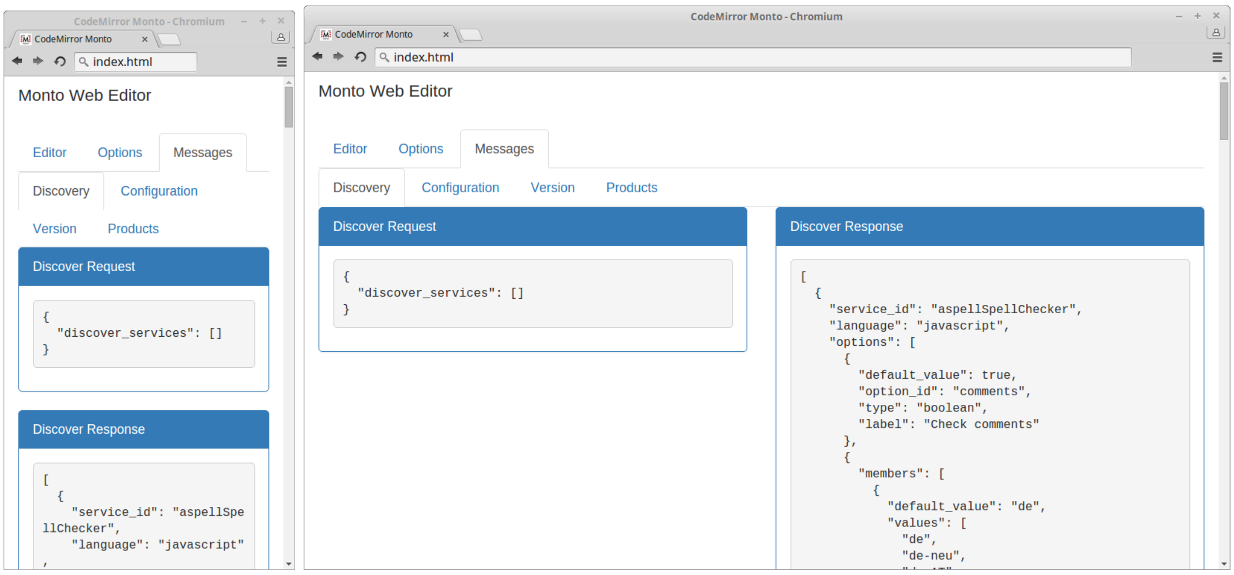
*CodeMirror* modes typically have a function that is called when the editor field changes and processes a line of the source code. This function then has to tokenize this line and highlight each token according to its type. This function is triggered immediately when the editor field changes, but at this point the *Version Message* is just sent by the source, hence the current *Product Messages* are not back at the sink at this time. The *Monto* mode therefore works differently. It registers a function to the sink that reads the new tokens *Product Message* and highlights the given tokens through the usage of the *CodeMirror* API. The function for tokenization just skips the line it is provided and does nothing. The registered function also contains handling for *outline* products and just rebuilds the unordered list and inserts it into the outline panel.

The *monto-hint* and *monto-lint* helpers are, like the *Monto* mode, triggered too early when the recent products are not available to them. *CodeMirror* helpers have a function that is executed when the helper is triggered, this function processes the products that are received. The helpers also register a function to the sink that just retriggers themselves, so they can operate on new products whenever they are available.

Each function that is registered to the sink has to defined, which kind of product type it wants top operate on. If no type is given, all *Product Messages* are available.

## 3.5 Graphical User Interface of the Editor

Since the web-based editor is a web app and is written using HTML, CSS and *JavaScript*, it can be viewed in the browser. The *Graphical User Interface* (GUI) itself is created by a HTML document. As plain HTML is unaesthetic, CSS is used to beautify and polish the graphical interface. As mentioned earlier, *Bootstrap* plays a major role in improving it. *Bootstrap* also adds responsive behaviour to the web app which can be seen in figure 3.3. When the browser size is decreased, the tabs in the tab row are wrapped into new lines and panels that are next to each other get rearranged to fit the window size as good as possible. Although this feature of *Bootstrap* aims to allow an easy creation of both, Desktop and Mobile apps, the web-based editor is not available as an app for Android or any other mobile device. The functionality behind the surface is done via *JavaScript*, which processes input from the user or from the *Monto* broker.



```
{
  "discover_services": []
}
```

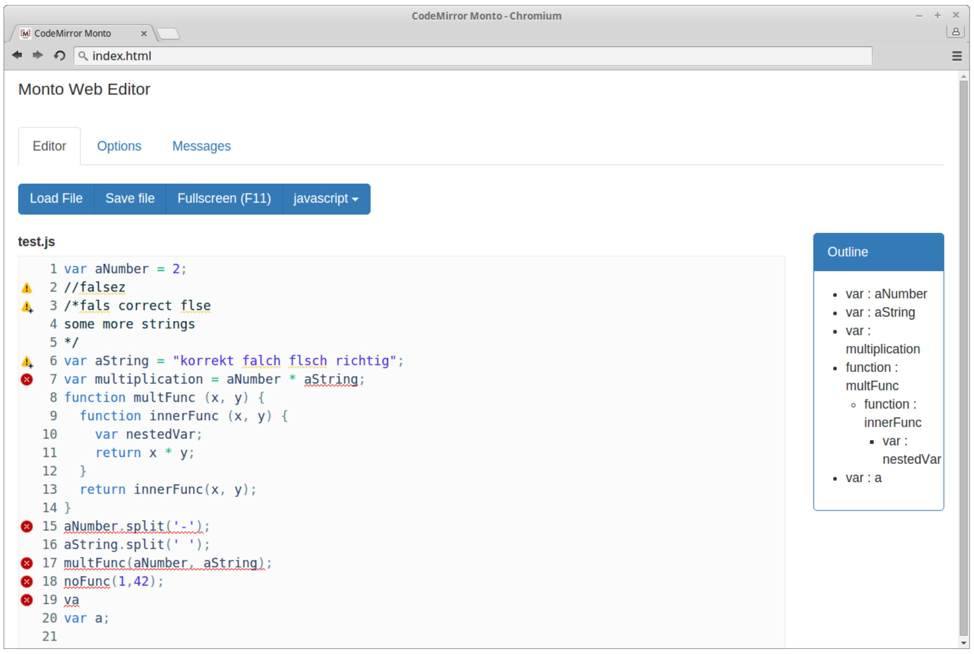
```
[
  {
    "service_id": "aspellSpe
llChecker",
    "language": "javascript"
```

```
[
  {
    "service_id": "aspellSpellChecker",
    "language": "javascript",
    "options": [
      {
        "default_value": true,
        "option_id": "comments",
        "type": "boolean",
        "label": "Check comments"
      },
      {
        "members": [
          {
            "default_value": "de",
            "values": [
              "de",
              "de-neu",
```

**(a)** Small GUI **(b)** Large GUI

**Figure 3.3.:** Responsive Behaviour of the GUI with *Bootstrap*

The GUI consists of a tab bar with three elements. The first tab is the *Editor* tab, which can be seen in figure 3.4. As the name says, in this tab the text field can be found and some buttons for functionality, like loading or saving files, enlarging the text field to screen size and selecting the currently used programming language. The available languages are determined by discovery, which will be discussed in chapter 5. The editor tab also features an outline panel in which *outline* products are visualized. Through the responsive design, the outline tab can be at the right side of the text field for bigger devices or beneath it for smaller devices. Whenever the content of the text field changes, a *Version Message* is sent to the *Monto* broker.

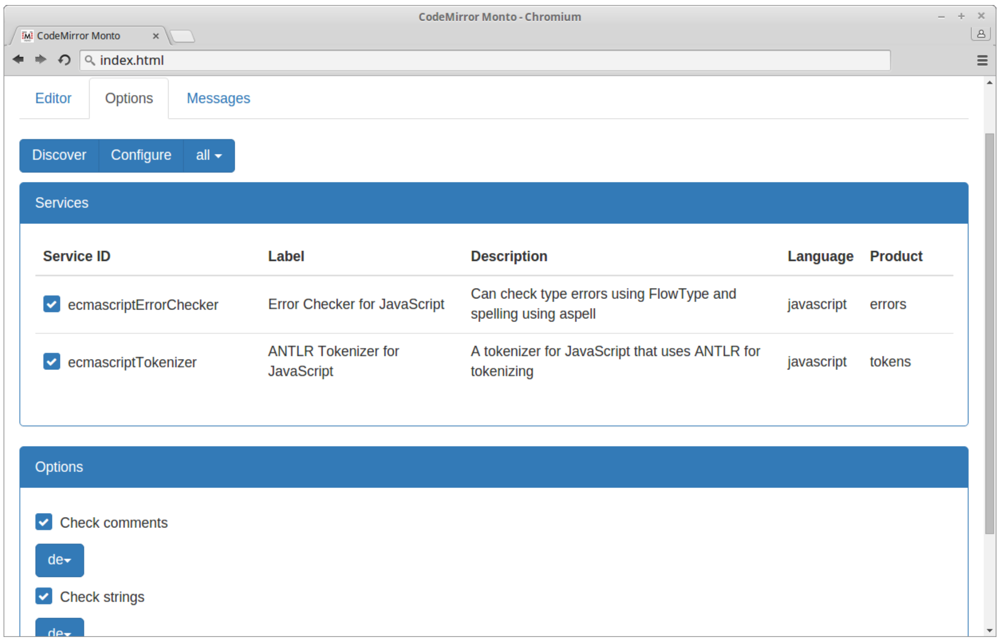


```
var aNumber = 2;
//falsez
/*fals correct flse
some more strings
*/
var aString = "korrekt falch flsch richtig";
var multiplication = aNumber * aString;
function multFunc (x, y) {
  function innerFunc (x, y) {
    var nestedVar;
    return x * y;
  }
  return innerFunc(x, y);
}
aNumber.split('-');
aString.split(' ');
multFunc(aNumber, aString);
noFunc(1,42);
va
var a;

```

**Figure 3.4.:** Editor Tab

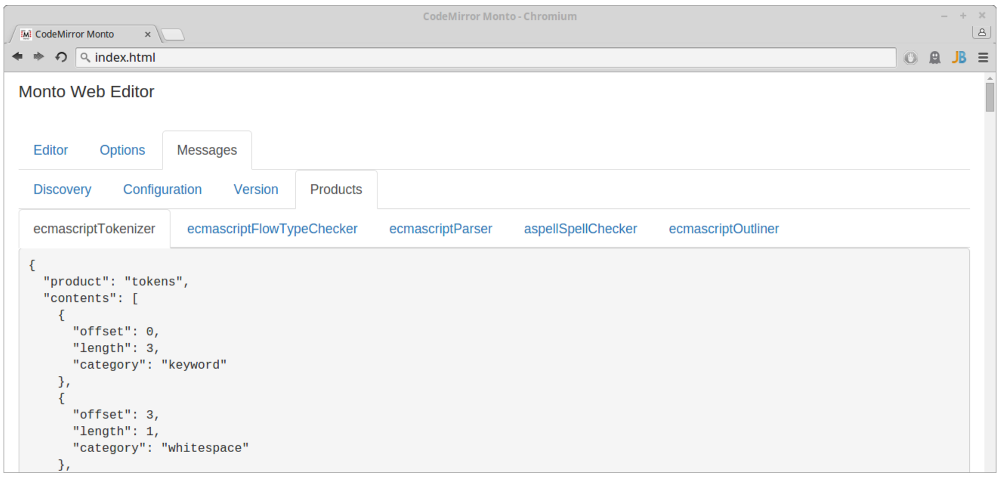
The *Options* tab contains the possibility to configure the editor. Figure 3.5 shows it for several *JavaScript* services. In this tab, the discovery of available services can be executed by pressing the corresponding button. The editor will send a *Discover Request* to the broker, which will be further explained in chapter 5. After a *Discover Response* is received by the editor, in the *Services* panel, all available services are listed with their id, label, description, language and product type. Configuration options for each service are listed below in the *Options* panel. The user can select and deselect services that should be used and configure them as wanted in the Options panel. After the configuration is finished, the *Configure* button can be pressed to inform the services about the new configuration. More about the topic of configuration can be found in chapter 6.



| Service ID | Label | Description | Language | Product |
|---|---|---|---|---|
| ✓ ecmascriptErrorChecker | Error Checker for JavaScript | Can check type errors using FlowType and spelling using aspell | javascript | errors |
| ✓ ecmascriptTokenizer | ANTLR Tokenizer for JavaScript | A tokenizer for JavaScript that uses ANTLR for tokenizing | javascript | tokens |

**Figure 3.5.:** Options Tab

The last tab, called *Messages*, contains visualization for all outgoing and incoming messages of the editor. The last message for each message type and service is displayed and can be navigated in another tab bar. An image of the *Messages* tab is located in figure 3.6.



```
{
  "product": "tokens",
  "contents": [
    {
      "offset": 0,
      "length": 3,
      "category": "keyword"
    },
    {
      "offset": 3,
      "length": 1,
      "category": "whitespace"
    },
```

**Figure 3.6.:** Messages Tab

# 4 Implementation of JavaScript Services

*Keidel* designed several types of product conventions for the *Monto* framework [35]. These product conventions are *tokens*, *ast*, *outline* and *completions*. For each of them, the implementation of services for *JavaScript* is discussed in this chapter and also a new *errors* product and a service for it is introduced. The code for the *JavaScript* service can be found on *GitHub* [28].

## 4.1 Existing Product Types

*Keidel* implemented services for *Java* with the above listed product types, in the scope of his master thesis [35]. Those services were implemented using the programming language *Java* and were integrated into a *Eclipse* IDE plug-in for *Monto* [23]. To make the services accessible without the *Eclipse* plug- in, they were outsourced and modified to run on their own [27]. Also, to simplify the process of implementing new services using *Java*, a base *Java* library was created, which provides an abstract class for *Monto* services, called *MontoService* and also several classes for JSON messages, including de- and encoding. The library can be found on *GitHub* [26].

The *MontoService* class implements the *Runnable* interface and can be used to create a new thread. For communication between the broker and the service itself, a library called *jeromq* [16] is used. This library is a ZMQ binding and enables the use of ZMQ in *Java*. To model JSON messages in *Java*, the library *json-simple* [19] is used. It provides an easy way to encode and decode *Java* objects and JSON objects. The *MontoService* class implements a method called *run* in which at first the service registers itself to the broker with information given on construction and if the registration was successful, it connects on a new port to start the main loop in which *Version Messages* and product dependencies are received from the broker, processed and a resulting *Product Message* is sent back. In this loop, an abstract method called *onVersionMessage* is invoked to process the request from the broker. Another abstract method called *onConfigurationMessage* is executed for processing Configuration Messages from the broker and to configure the service properly. Both methods have to be overridden by a class that extends the *MontoService* class. Information about the registration and configuration processes can be found in chapter 5 and 6. The modified *Java* services use the base library.

To extend *Monto* further, *JavaScript* services for the existing product types were created, using *Java* and the base library. The services were implemented similar to the ones from *Keidel* and therefore also use the same technologies.

The *tokens* service uses *Another Tool for Language Recognition* (ANTLR) [2] for tokenization. ANTLR is a tool that can generate a parser and a lexer by using an ANTLR grammar for a programming language. *JavaScript* is an implementation of the *ECMAScript* language standard [14] which is developed at *ECMAScript International* [10]. For ANTLR 4, a *ECMAScript* grammar is available at *GitHub* [9]. With this grammar, a parser and a lexer for *ECMAScript* were generated by the ANTLR tool and is now used by the *JavaScript* services to process received source code. The ANTLR lexer provides a list of tokens, whose elements are converted to the appropriate *Monto* tokens, since *Monto* defines its own standardized token types. This list then is wrapped in a *tokens Product Message*. The *ast* service uses the ANTLR parser to build an *Abstract Syntax Tree* (AST) and converts it into the correct JSON format. Both, the *outline* and the *completions* services, do not only require the source code but also the AST. They receive both from the broker and analyze the AST to find required nodes. If a node is found which has a type of interest, the correct text is taken form the source code, like for example identifiers of class names for outlining or identifiers of variables that can be used for code completions.

## 4.2 The 'Errors' Product

As most IDEs inform the user about erroneous code or give warnings and hints about code that could be improved or is problematic, this feature also should be included in the *Monto* project. To enable such services, a new product type is introduced, called *errors*. This products' contents attribute contains a list of objects that contain information about errors and warnings, including a *description*, a *category*, an error *level* and the position in form of an *offset* and the *length*. The *level* describes the severity of the object. Possible values are *warning* or *error*. The *description* should contain some short information about the reason why this particular position is detected as an error. With the *category* field, the kind of found error should be described further. For example, for a type error the value is *type*. In listing 4.1 an example of an *errors* product *contents* can be seen. Listing B.1 in the appendix shows the JSON schema.

```
[
  {
    "offset": 69,
    "length": 7,
    "level": "error",
    "description": "string is incompatible with number",
    "category": "type"
  },
  {
    "offset": 209,
    "length": 8,
    "level": "error",
    "description":  "identifier multFunc Could not resolve name",
    "category": "type"
  }
  ...
]
```

**Listing 4.1:** *Errors* Product Example

*JavaScript* is an untyped programming language and therefore highly susceptible for writing erroneous code. However, there are tools like *FlowType* [12] that can check *JavaScript* code for type errors and provide detailed information about them to the user. As an example and proof-of-concept an *errors* service for *JavaScript* was implemented. This service is capable of analyzing *JavaScript* code for type errors, using *FlowType* but it can also check the spelling of comments and strings by using the *tokens* product and a program called *aspell* [3]. *Aspell* can check texts in different languages by the use of dictionaries and if it cannot find a word it can suggest similar words from the dictionary. To provide a service that also makes use of configurations, which will be introduced in chapter 6, the *errors* service can be configured to check strings and/or comments, which languages should be used for them, if suggestions should be given and if so, how many should be provided.

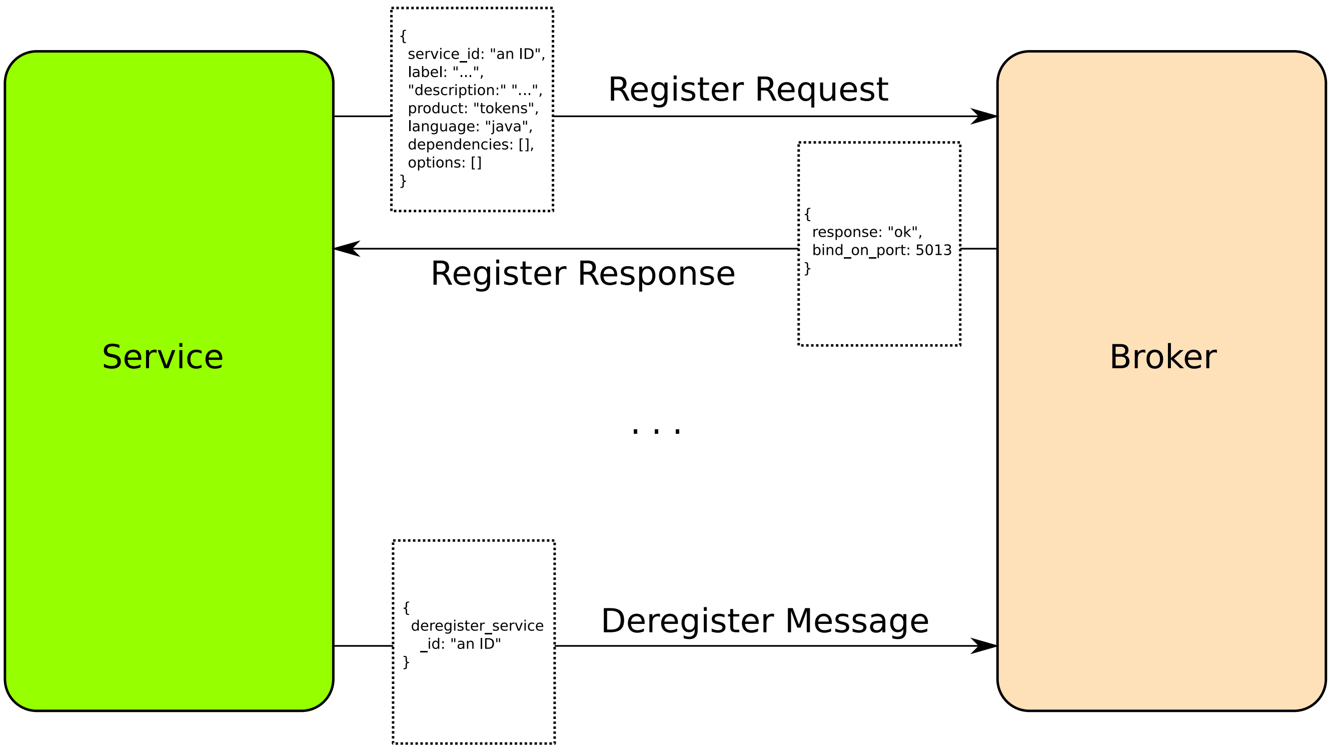
# 5 Discoverability in the Monto Framework

To enable dynamic addition and removal of services to or from the broker, a possibility of notifying the broker about service changes is required. Also sources and sinks need to know, which services are available to the broker and therefore require the possibility to retrieve such information. This chapter discusses these two problems and points out a solution. All new message specifications can be found on *GitHub* [24] as well as modifications to the broker [21].

## 5.1 Registering Services

The broker in its current state, requires defining services through a command line parameter. This is a static procedure and requires the restart of the broker every time a new service should be added or an existing removed. To make handling of services more dynamic some form of registration is required.

Registration of services brings several problems with it that have to be solved to provide fully working and consistent addition and removal of services. The services and the broker have to be able to communicate with each other. Therefore a ZMQ pair connection could be a good solution, but actually there are some problems.



```
{
  service_id: "an ID",
  label: "...",
  "description:" "...",
  product: "tokens",
  language: "java",
  dependencies: [],
  options: []
}
```

```
{
  response: "ok",
  bind_on_port: 5013
}
```

```
{
  deregister_service
   _id: "an ID"
}
```

**Figure 5.1.:** Registration Process

The first problem is that the service does not know on which port it can reach the broker, because each service requires its own port, so it is not sufficient to open some ports on the broker. To overcome this problem a port for registration only is opened by the broker. This port is bound by a ZMQ response connection, which is part of the *request-response* pattern. Multiple services can simultaneously connect on this port with a ZMQ request connection and can register and ask the broker on which port they can continue their communication. The broker has to ensure that it does not assign multiple services to the same port, so that each port is only used by one service at the same time.

Although the lack of communication channels is solved, the question, what to send over those channels, is still open. When a server registers with the broker over the registration connection, the broker requires some specific information about the service to handle the request. To enable communication, a format has to be defined. Since the *Monto* framework uses JSON objects for messaging, the registration will also use them. There are three messages that are required:

1. *Register Request*
2. *Register Response*
3. *Deregister*

When a new service wants to register with the broker, at first it sends a *Register Request*. It contains several information about the service that the broker requires to handle it properly. The information are a unique id for the service, a label, a description, the specification which language and which product the service can handle, possible options which will be discussed in chapter 6 and the dependencies that the service has. Listing 5.1 gives an example message and listing B.2 provides the full JSON schema.

```json
{
  "service_id": "ecmascriptTokenizer",
  "label": "ANTLR Tokenizer for JavaScript",
  "description": "A tokenizer for JavaScript that uses ANTLR for tokenizing",
  "language": "javascript",
  "product": "tokens",
  "options": [],
  "dependencies": []
}
```

**Listing 5.1:** *Register Request* Example

The second message is sent after receiving a registration request. This message has the purpose to tell the service whether the registration is successful or not and if it was successful, on which port the service can bind, so that the broker can start sending *Version Messages* and receive *Product Messages* from the service. A registration request could look like the example in listing 5.2. For the JSON schema see listing B.3.

```json
{
  "resposne": "ok",
  "bind_on_port": 5013
}
```

**Listing 5.2:** *Register Response* Example

The last message is sent by a service when it is about to terminate and aims to deregister itself from the broker, so that the broker can then reassign the port to other, new services. It only contains the service id of the deregistering service. Listing 5.3 provides an example *Deregister* message, whereas the JSON schema can be found in listing B.4. The complete process of registration can be seen in figure 5.1.

```
{
  "deregister_service_id": "ecmascriptTokenizer"
}
```

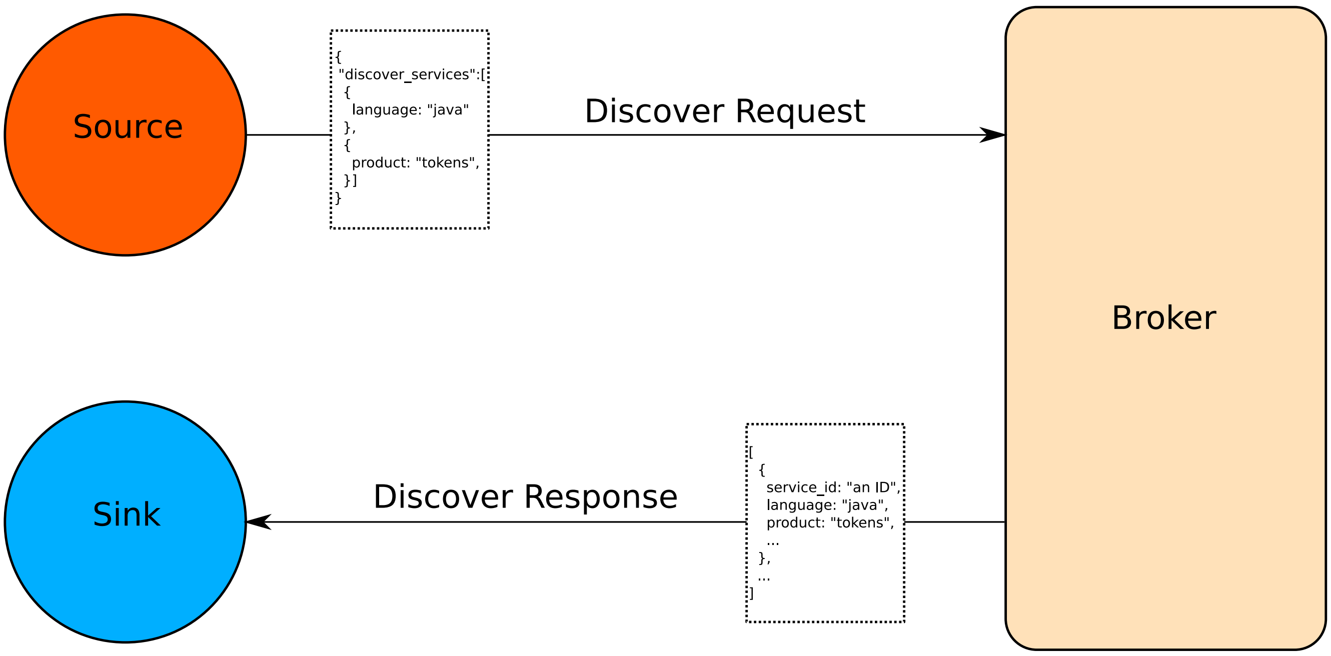
**Listing 5.3:** *Deregister* Example

## 5.2 Discovering Services

Sources and sinks, or more general clients, do not know what services are available to the broker. There is no intended way to retrieve information about active services, so sinks cannot rely on a specific product to receive.

As the services now register with the broker, the broker always knows what services he can offer. Hence, a simple and clean way to get the information for the client would be to ask the broker. In fact, as the sinks are designed to receive messages and sources to send messages, the source should send a request for a list of available services to the broker which then should publish a message with the service list to the sinks.

Again, new messages are required to model the request and response and as in the registration messages we use JSON objects for discovery. In this case, only two messages are required: the *Discover Request* and the *Discover Response*.



```
{
 "discover_services":[
  {
   language: "java"
  },
  {
   product: "tokens",
  }]
}
```

```
[
  {
   service_id: "an ID",
   language: "java",
   product: "tokens",
   ...
  },
  ...
]
```

**Figure 5.2.:** Discovery Process

To enable the client to discover only specific services, for example only *JavaScript* services, the *Discover Response* can contain a list of filter criteria. Each filter criteria is a JSON object that can set three different attributes: a *service id*, a *language* and a *product* type. The *service id* can be used to see if a specific service is still active and available. The other criteria can be used to identify a group of services. All criteria can be mixed together in an object, but also only one criterion can be used. The message has a list of criteria because the client for example could be searching for all *Java* and all *JavaScript* services. An example *Discover Request* message can be seen in listing 5.4. The JSON schema of the *Discover Request* is explained in listing B.5.

```
{
  "discover_services": [
    {
      "service_id": "ecmascriptParser"
    },
    {
      "product": "tokens"
    }
  ]
}
```

**Listing 5.4:** *Discover Request* Example

The *Discover Response* is an array of JSON objects that each contain information for one service. Information includes the *service id*, a *label* and a *description*, the *language*, the *product* type and the *options*, which are explained in chapter 6, of a service. The array only contains services that match one or more criteria from the *Discover Request*. If the *Discover Request* is empty, all services are listed.

In its current state, the broker will also send a complete *Discover Request* without filters whenever a new service registers or a known service deregisters. This enables a more dynamic visualization on the client side, however does not contain any filters. Listing 5.5 shows an example *Discover Response* and the corresponding JSON schema can be found in listing B.6. The complete process of discovering services is visualized in figure 5.2.

```
[
  {
    "service_id": "javaTokenizer",
    "language": "java",
    "options": null,
    "product": "tokens",
    "description": "A tokenizer for Java that uses ANTLR for tokenizing",
    "label": "ANTLR Tokenizer for Java"
  }
  {
    "service_id": "ecmascriptParser",
    "language": "javascript",
    "options": null,
    "product": "ast",
    "description": "A parser that produces an AST for JavaScript using ANTLR",
    "label": "ANTLR JavaScript Parser"
  },
  {
    "service_id": "ecmascriptTokenizer",
    "language": "javascript",
    "options": null,
    "product": "tokens",
    "description": "A tokenizer for JavaScript that uses ANTLR for tokenizing",
    "label": "ANTLR Tokenizer for JavaScript"
  }
]
```

**Listing 5.5:** *Discover Response* Example

# 6 Configurability of the Monto Framework

There are two types of possible configurations within the *Monto* project. The first one is to configure which services are used by a sink and the other one is to configure services themselves, if they provide options that can be configured. In this chapter, both kinds of configurations are discussed in this chapter. Modifications to the broker [21] and new message specifications can be found on *GitHub* [24].

## 6.1 Configuring the Broker

Clients now are able to know which services are available through the discoverability that was discussed in chapter 5, but they might also want to receive only products from specific services, which also helps to reduce the overhead of unused messages. There are at least two ways to achieve this. The first possible solution is to create a unique ZMQ pair connection for each sink. To determine which products should be sent to a sink, it must inform the broker about its preferences. However, this would lead to a much more complex broker because it would need to keep state of sinks and also be able to receive messages from the sinks. The second approach is to make use of the *publish-subscribe* pattern and to publish each product under a topic named after the service id origin of the product. This approach is much easier to implement and makes use of the existing technologies. Hence, the second approach was chosen and implemented. *Product Messages* are now published over service topics. Each service has its own topic and the product of a service is published over its topic. If a sink only wants to receive products from specific services, the sink can just subscribe to their topics and only receive those messages.

However, for the WebSocket Proxy this actually does not work this way because *publish-subscribe* pattern is not available. The ZMQ connections do publish and subscribe with an empty topic, which in fact means subscribing to all topics. Though, as a subscriber always two messages are received. At first it receives the topic and second it receives the message, so by forwarding all messages to the WebSocket connection, the sinks also receive the topic under which a product was published and therefore sinks can choose to only process products from services that they want to.

To improve performance and lower working overhead, a desirable goal would be that the broker only sends *Version Messages* to services that are currently used by one ore more sinks. Yet, to do so, the implementation of the broker would require much more complexity and each sink would require an ID and also a registration to the broker so that it is aware of them and can act if a sink changes its required services. In other words, the broker needs to keep state of the sinks. Each sink then would have to tell the broker which services it requires and the broker then has to see what services are needed, including their dependencies. Though, the broker then would require to rebuild its state machine, so that only required services are included. All this is not part of this bachelor thesis and is not taken into account.

## 6.2 Configuring Services

There are services that have to produce a product that can not be modified, for a example a tokenizer. The *tokens* product should always look the same from all tokenizer services that support the same programming language. Though, there are other services that could produce a product that depends on several options, for example a spell checker for strings that can handle different languages like *German* or *English*.

Services that potentially can have options require a way to express this, but first it is important to define what kind of options are possible. Therefore multiple JSON objects have been defined, where each describe a possible option. All of these objects, except one, have in common that they provide several attributes like an *option id*, a *label*, a *type* and a *default value*. The *option id* is unique over the complete collection of options for a service, the *label* is a short, human readable name, the *type* describes what kind of option it is and the *default value* provides the standard active option if nothing else is set.

The first type of options is the *Number Option*, which can be set to an integer. It has two more attributes to its JSON object, called *from* and *to*. Those two fields describe the range in which the number can be chosen. If one of them is not set, there is no defined limit in the appropriate direction of integers. One must be aware that some programming languages or operating systems can have different integer sizes and therefore the maximum is not uniform. Furthermore, if the value in a configuration is set to an integer out of bounds, the *default value* should be used. An example for a *Number Option* can be found in listing 6.1. The appropriate schema can be seen in listing B.7.

```json
{
  "option_id": "suggestionNumber",
  "default_value": 5,
  "from": 0,
  "label": "Maximum number of suggestions",
  "to": 10,
  "type": "number"
}
```

**Listing 6.1:** *Number Option* Example

The next option type is the boolean type. This type is pretty simple and defines no additional attributes to its JSON object, as it can have only two different values set, true or false. It can be visualized by for example a checkbox or a switch. Listing 6.2 shows an example for a *Boolean Option* and the associated schema is described in listing B.8.

```json
{
  "option_id": "suggestions",
  "default_value": false ,
  "label": "Show suggestions",
  "type": "boolean"
}
```

**Listing 6.2:** *Boolean Option* Example

The *Text Option* type is a type that requires a string as configuration. It defines an attribute called *regular expression* that can be used to define rules for the set string. The *extended POSIX standard for regular expressions* should be used [30]. If this field is empty, no rules are set and all strings can be used for configuration. A *Text Option* example is depicted in listing 6.3, whereas the schema is shown in listing B.9.

```
{
  "option_id": "textOpt",
  "default_value": "some test",
  "label": "A text option",
  "type": "text",
  "regular_expression": ""
}
```

**Listing 6.3:** *Text Option* Example

Another type of options is the *Xor option*. As the name says, it is an exclusive-or option and defines multiple possible string options in the attribute called *options* and only one of those can be selected for configuration. The *Xor Option* can be represented by a dropdown menu or a collection of radio buttons that are connected through an xor link. The corresponding JSON schema is available in listing B.10. An example for the *Xor Option* is given in listing 6.4.

```
{
  "values": [
    "de",
    "en",
    "es",
    "fr"
  ],
  "option_id": "stringLanguage",
  "default_value": "de",
  "label": "Language for strings",
  "type": "xor"
}
```

**Listing 6.4:** *Xor Option* Example

The last option object, is a JSON object that does not share the same fields with other options. Also it does not define something that can be selected for configuration but helps to group options and to en- or disable some when some options have requirements regarding the configuration.

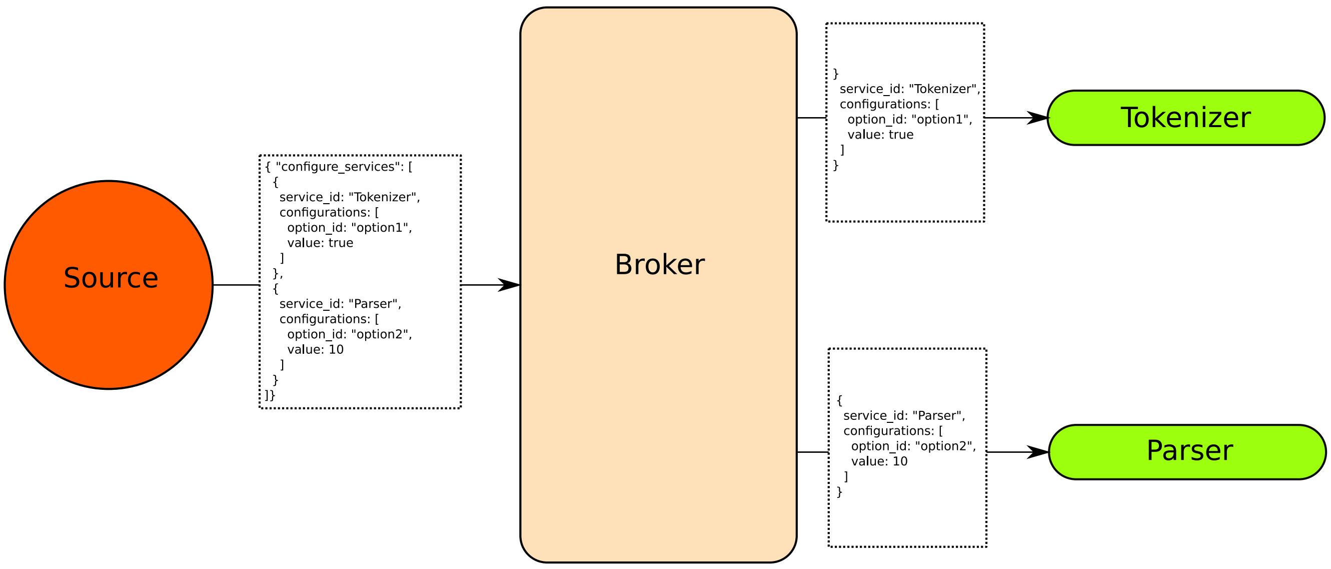
This object is called *Group Option* and contains a string in the *required option* attribute as well as an array of member option objects in the field called *members*. The string contains the *option id* of a *Boolean Option* that must be enabled so that the options in the *members* field can be configured. For the requirements, only a *Boolean Option* can be used, whereas for the *members* all option objects are allowed, including other groups. For multiple *Boolean Options* as a requirement, several *Group Options* can be nested. Listing 6.5 shows a *Group Option*. Listing B.11 holds the JSON schema.

```
{
  "members": [
    {
      "option_id": "suggestionNumber",
      "default_value": 5,
      "from": 0,
      "label": "Maximum number of suggestions",
      "to": 10,
      "type": "number"
```

```
        }
      ],
      "required_option": "suggestions"
    }
```

**Listing 6.5:** *Group Option* Example

Each service can define its options using the objects presented above and list them in the *Register Request* in an array field called *options*. The broker itself does not process these options but stores it along several other information for the service. When a client asks for a list of available services, the broker then sends these options in the *Discover Response* in an array attribute with the name *options*. The sink then has to visualize these options to the user, so that he can set some of the given options. When the user has configured the options to his liking, the services need to be informed. Therefore, another new JSON message is introduced, called *Configuration Message*.



```
{ "configure_services": [
  {
   service_id: "Tokenizer",
   configurations: [
    option_id: "option1",
    value: true
   ]
  },
  {
   service_id: "Parser",
   configurations: [
    option_id: "option2",
    value: 10
   ]
  }
]}
```

```
}
 service_id: "Tokenizer",
 configurations: [
  option_id: "option1",
  value: true
 ]
}
```

```
{
 service_id: "Parser",
 configurations: [
  option_id: "option2",
  value: 10
 ]
}
```

**Figure 6.1.:** Configuration Process

The *Configuration Message* is an array of configuration objects that each contain the *service id* of the service that is configured in the *configurations* field. The *configurations* field is also an array that contains an option id and a value that was chosen by the user. This *Configuration Message* is sent by the source to the broker, which then splits the array and sends each part to the corresponding service with the given *service id*. The service then can process the configuration and set its options accordingly. The next *Version Message* then can be processed with the given configuration. The process of configuration can be seen in figure 6.1. An example *Configuration Message* can be found in listing 6.6 and the corresponding JSON schema in listing B.12.

This approach right now does only work with a single configuration per service. So if multiple clients, or sinks want to have multiple, different configurations this will not work, because the last *Configuration Message* will override the existing one. To overcome this problem, a possible solution would be to create a new instance of a service for each sink so that they can be configured in isolation. This would require each sink to have an ID so that the service instance can be mapped to them. However, this is not in the scope of this bachelor thesis and would, as mentioned earlier, lead to a much more complex broker that needs to keep state of the sinks.

```json
{
  "configure_services": [
    {
      "service_id": "ecmascriptErrorChecker",
      "configurations": [
        {
          "option_id": "comments",
          "value": true
        },
        {
          "option_id": "commentLanguage",
          "value": "de"
        },
        {
          "option_id": "strings",
          "value": true
        },
        {
          "option_id": "stringLanguage",
          "value": "en"
        },
        {
          "option_id": "suggestions",
          "value": true
        },
        {
          "option_id": "suggestionNumber",
          "value": 2
        }
      ]
    }
  ]
}
```

**Listing 6.6:** *Configuration Message* Example

# 7 Evaluation and Validation

Software development is prone to errors and bugs and hence the software quality has to be ensured through several techniques. The more complex a software becomes, the more likely it is to contain errors. As the software developed in scope of this thesis is split into several self-contained parts, each of those programs has a small extend.

In chapter 3.1 it was already mentioned, that the development process was prototypical. A prototype was developed, extended and inspected weekly by the supervisor. Hence, functionality and correct behaviour of the software was ensured. To detect bugs, extensive manual testing of the software was done by the author. Third-party libraries were used where possible, which on the one hand reduced the amount of work to be done and on the other hand also reduced sources of errors.

```javascript
var aNumber = 2;
//falsez
/*fals correct flse
some more strings
*/
var aString = "korrekt falch flsch richtig";
var multiplication = aNumber * aString;
function multFunc (x, y) {
  function innerFunc (x, y) {
    var nestedVar;
    return x * y;
  }
  return innerFunc(x, y);
}
aNumber.split('-');
aString.split(' ');
multFunc(aNumber, aString);
noFunc(1,42);
va
var a;
```

**Listing 7.1:** Test Code Snippet

For manual testing, a *JavaScript* code snippet was written that was used every time to test implemented changes. The snippet within the browser can be seen in figure 3.4 and the snippet itself in listing 7.1. The code snippet contains tokens of different types for syntax highlighting, comments and strings for spell checking, nested functions for outlining, some type errors for error checking and some variables that start with the same sequence of characters which can be used for code completion. For each of the contained elements, several different correct and incorrect values are available in the snippet to achieve a broad coverage. The testing was done via starting the broker, the services and the web-based editor and then inserting the code snippet into the editor field. The editor then, should send a *Version Message* and visualize the received *Product Messages*. If incorrect visualization can be seen, an error must exist in the implementation. With this procedure both can be detected, an error in a service or an error within the web-based editor. This way, bugs can be found but the complete absence of such can not be proven. However to achieve this, a huge amount of work has to be invested, if even possible. As the scope and size of this project is small and manageable, the used method of testing is sufficient.

The configurations were tested by using dummy configuration options in a service. These options could be seen in the editor, set to several configuration and then a result could be logged in the service. This way, the correct handling of options in the web-based editor could be ensured.

Not only the implementation, but also the communication between *Monto* components has to be correct. Communication can be split into four groups. *Registration*, *Deregistration*, *Discovery* and *Configuration*. Each of them consists of at maximum two message types and therefore are almost trivial. To ensure correct implementation and sequence of messages, they were implemented step by step. Sent and received messages were logged to the command line. Behaviour and sequence of messages were checked with the help of a debugger. As no message contains programming language specific information or fields, except the name of the language itself, each of them will work with every programming language.

# 8 Related Work

Web apps have become very popular due to their advantages like availability to every computer with a browser installed or the absence of manual updates as the web app is updated server side. Hence, also web based editors already have been developed.

An example for a web based editor is *Atom*, developed by the *GitHub* team [4]. This editor, though, runs not directly in a browser but acts like a typical desktop application. Applications like this can be built using the build platform *Electron*. Atom is built with HTML, CSS and *JavaScript*. However, features like syntax highlighting are all processed within the application itself, unlike in *Monto*, where such features are outsourced into services, accessible by the broker. This separates it considerably from the web-based editor developed within this thesis. Atom provides a package system to install and remove extensions to the editor, making it possible to extend language support. Those packages are Atom specific and lead to the problem, that *Monto* tries to handle.

A web based IDE would be the *Cloud9* IDE that uses the *Ace* editor as a base [7]. As mentioned earlier in chapter 3, *Ace* is a editor framework based on *JavaScript* and provides similar functionalities as *CodeMirror*. *Cloud9* however, sets its focus on cloud computing. Therefore, *Cloud9* also provides the possibility to write code with other programmers collaboratively in real time and also to save projects into workspaces, stored in the cloud. *Cloud9* consists of a server-side *JavaScript* back-end and the browser front-end. IDE features like syntax highlighting and code completion are done locally in the browser front-end using *JavaScript* and the *Ace* editor, which again significantly differs it from the idea of *Monto* to disintegrate such features from the IDE or front-end itself.

*Van Deursen, et al.* came up with an approach for a browser-based IDE called *Adinda* [38]. *Adinda* consists of a server and several browser clients, that communicate with each other using *Ajax*. In this approach, the goal is to enable a more team based and collaborative developing on a project by collecting information at the server, while developers are working on the source code. This leads to the possibility to see which developers wrote the code and how it changed over time, similar to a *version control system*. But *Adinda* also outsources IDE features like compilation of source code. This project tries to reduce message overhead over *Ajax* by using the DOM to model the AST of the source code and only sending AST fragments that really have changed. The server compiles the new source code on changes and sends the compilation output to the client. *Adinda* follows a similar approach as *Monto* by outsourcing IDE features to a server, however it still is different in the core. *Adinda* does not aim to decouple services and make them exchangeable, like for example *Monto* does with the broker and service architecture.

# 9 Conclusion and Future Work

*Monto* is a rather new project and needs to be extended in both, new features and available programs like editors with *Monto* support, but also with new services for several languages and product types. In this thesis each of those improvements were done. A new editor with *Monto* support was built by using web technologies and existing frameworks. New features like WebSocket support, dynamic service management for the broker, discovering and configuring of services for clients have been introduced and implemented. Each of those simplify the use of *Monto* by decoupling the components in the *Monto* framework. For example when the broker is started now, it is not required to know which services will be available in the future and therefore no configuration at startup has to be provided. The user can start the broker and connect, as well as disconnect all services whenever he wants to. Also the client does not have to know about what services are available by itself, but can discover the services that are available to the broker. A service base library for *Java* has been developed to even further ease implementation of services. The library was used to create new services for the programming language *JavaScript*. To increase the pool of available product types, a new product type called *errors* was introduced. This product type was used to implement a service for *JavaScript* that identifies type errors in *JavaScript* with the usage of *FlowType* and is also capable of producing warnings about spelling for several languages in comments and also strings by using *aspell*. Hence, an extension and improvement of the *Monto* project in the context of this thesis was accomplished. The current state of the *Monto* framework and all modifications, additions and programs developed in scope of this thesis can be found on *GitHub* [25].

For future projects, *Monto* can be extended even further by fixing the existing problems mentioned in chapters 5 and 6. One could solve the problem of service instances for each sink, so that each sink can separately configure a service. This would lead to the possibility to again use multiple sources and sinks in the *Monto* project, without having only one configuration at a time. Also the configuration itself could be extended to allow different and more complex configurations. As the broker right now is not capable of having two different services with the same product/language combination but different dependencies, the underlying dependency graph and the automaton could be extended.

*Monto* is meant to be run locally on one computer. In the future, an evaluation could be made to see if *Monto* is capable of running over a local area network or also over the internet and what could be changed in *Monto* to improve performance. This would lead to the requirement of encrypted communication between the parts and associated source and sink groups to ensure privacy and trust for each participant that wants to use *Monto*. The web-based editor then would be a good foundation to run *Monto* as a web app on the internet.

# A Abbreviations

| | |
|---|---|
| **ANTLR** | Another Tool for Language Recognition |
| **API** | Application Programming Interface |
| **AST** | Abstract Syntax Tree |
| **CSS** | Cascading Style Sheets |
| **DDE** | Disintegrated Development Environment |
| **DOM** | Document Object Model |
| **GUI** | Graphical User Interface |
| **HTML** | Hypertext Markup Language |
| **IDE** | Integrated Development Environment |
| **JSON** | JavaScript Object Notation |
| **WS-ZMQ-Proxy** | WebSocket-ZeroMQ-Proxy |
| **ZMQ** | ZeroMQ |

## B JSON Schemas

```
{
  "title": "Errors",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "offset": {
        "type": "integer",
        "minimum": 0
      },
      "length": {
        "type": "integer",
        "minimum": 1
      },
      "category": {
        "type": "string",
        "enum": [
          "syntax", "type", "spelling"
        ]
      },
      "level": {
        "type": "string",
        "enum": [
          "warning", "error"
        ]
      },
      "description": {
        "type": "string"
      }
    },
    "required": [
      "offset", "length", "category",
      "level", "description"]
  },
  "minItems": 0,
  "uniqueItems": true
}
```

**Listing B.1:** *Errors* Product Schema

## B.1 Discoverability

```
{
  "title": "Register Request",
  "type": "object",
  "properties": {
    "service_id": {
      "type": "string"
    },
    "label": {
      "type": "string"
    },
    "description": {
      "type": "string"
    },
    "language": {
      "type": "string"
    },
    "product": {
      "type": "string"
    },
    "options" : {
      "type": "array",
      "items": {
        "oneOf": [
          { "$ref": "#/definitions/boolean-option" },
          { "$ref": "#/definitions/number-option" },
          { "$ref": "#/definitions/text-option" },
          { "$ref": "#/definitions/xor-option" },
          { "$ref": "#/definitions/group-option" }
        ]
      }
    },
    "dependencies": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  },
  "required": [
    "service_id", "label", "description",
    "language", "product"]
}
```

**Listing B.2:** *Register Request* Schema

```
{
  "title": "Register Response",
  "type": "object",
  "properties": {
    "response": {
      "type": "string",
      "enum": [
        "ok", "id taken", "no free port"
      ]
    },
    "bind_on_port": {
      "type": "integer",
      "minimum": 1024,
      "maximum": 65535
    }
  },
  "required": [
    "response"
  ]
}
```

**Listing B.3:** *Register Response* Schema

```
{
  "title": "Deregister",
  "type": "object",
  "properties": {
    "deregister_service_id": {
      "type": "string"
    }
  },
  "required": ["deregister_service_id"]
}
```

**Listing B.4:** *Deregister* Schema

```
{
  "title": "Discover Request",
  "type": "object",
  "properties": {
    "discover_services": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "service_id": {
            "type": "string"
          },
          "language": {
            "type": "string"
          },
          "product": {
            "type": "string"
          }
        },
        "required": []
      }
    }
  },
  "required": ["discover_services"]
}
```

**Listing B.5:** *Discover Request* Schema

```
{
  "title": "Discover Response",
  "type": "array",
  "items" : {
    "type": "object",
    "properties": {
      "service_id": {
        "type": "string"
      },
      "label": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "language": {
        "type": "string"
      },
      "product": {
        "type": "string"
      },
      "options" : {
        "type": "array",
        "items": {
          "oneOf": [
            { "$ref": "#/definitions/boolean-option" },
            { "$ref": "#/definitions/number-option" },
            { "$ref": "#/definitions/text-option" },
            { "$ref": "#/definitions/xor-option" },
            { "$ref": "#/definitions/group-option" }
          ]
        }
      }
    },
    "required": ["service_id", "label", "description",
      "language", "product"]
  },
  "required": []
}
```

**Listing B.6:** *Discover Response* Schema

## B.2 Configurability

```
{
  "title": "Number Option",
  "type": "object",
  "properties": {
    "option_id": {
      "type": "string"
    },
    "label": {
      "type": "string"
    },
    "type": {
      "type": "string",
      "enum": [
        "number"
      ]
    },
    "default_value": {
      "type": "integer"
    },
    "from": {
      "type": "integer"
    },
    "to": {
      "type": "integer"
    }
  },
  "required": [
    "option_id", "label", "type",
    "default_value", "from", "to"]
}
```

**Listing B.7:** *Number Option* Schema

```
{
  "title": "Boolean Option",
  "type": "object",
  "properties": {
    "option_id": {
      "type": "string"
    },
    "label": {
      "type": "string"
    },
    "type": {
      "type": "string",
      "enum": [
        "boolean"
      ]
    },
    "default_value": {
      "type": "boolean"
    }
  },
  "required": [
    "option_id", "label", "type",
    "default_value"]
}
```

**Listing B.8:** *Boolean Option* Schema

```
{
  "title": "Text Option",
  "type": "object",
  "properties": {
    "option_id": {
      "type": "string"
    },
    "label": {
      "type": "string"
    },
    "type": {
      "type": "string",
      "enum": [
        "text"
      ]
    },
    "default_value": {
      "type": "string"
    },
    "regular_expression": {
      "type": "string"
    }
  },
  "required": [
    "option_id", "label", "type",
    "default_value", "regular_expression"]
}
```

**Listing B.9:** *Text Option* Schema

```
{
  "title": "Xor Option",
  "type": "object",
  "properties": {
    "option_id": {
      "type": "string"
    },
    "label": {
      "type": "string"
    },
    "type": {
      "type": "string",
      "enum": [
        "xor"
      ]
    },
    "default_value": {
      "type": "string"
    },
    "options" : {
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "uniqueItems": true
    }
  },
  "required": [
    "option_id", "label", "type",
    "default_value", "options"]
}
```

**Listing B.10:** *Xor Option* Schema

```
{
  "title": "Group Option",
  "type": "object",
  "properties": {
    "required_option": {
      "type": "string"
    },
    "members": {
      "type": "array",
      "items": {
        "type": "object",
        "oneOf": [
          { "$ref": "#/definitions/boolean-option" },
          { "$ref": "#/definitions/number-option" },
          { "$ref": "#/definitions/text-option" },
          { "$ref": "#/definitions/xor-option" },
          { "$ref": "#/definitions/group-option" }
        ]
      }
    }
  },
  "required": [
    "requires", "members"]
}
```

**Listing B.11:** *Group Option* Schema

```
{
  "title": "Configurations",
  "type": "object",
  "properties": {
    "configure_services": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "service_id": {
            "type": "string"
          },
          "configurations": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "option_id": {
                  "type": "string"
                },
                "value": {
                  "type": "string"
                }
              },
              "required": [
                "option_id", "value"]
            }
          }
        },
        "required": [
          "service_id", "configurations"]
      }
    }
  },
  "required": ["configure_services"]
}
```

**Listing B.12:** *Configuration Message* Schema

## Bibliography

[1] Ace editor. http://ace.c9.io/. Accessed: 2015-09-07.

[2] Antlr. http://www.antlr.org/. Accessed: 2015-09-02.

[3] aspell. http://aspell.net/. Accessed: 2015-09-24.

[4] Atom editor. https://atom.io/. Accessed: 2015-09-28.

[5] Awesome-bootstrap-checkbox. https://github.com/flatlogic/awesome-bootstrap-checkbox. Accessed: 2015-09-02.

[6] Bootstrap. http://getbootstrap.com/. Accessed: 2015-09-02.

[7] Cloud9 ide. https://c9.io/. Accessed: 2015-09-28.

[8] Codemirror editor. http://codemirror.net/. Accessed: 2015-09-02.

[9] Ecmascript grammar for antlr 4. https://github.com/antlr/grammars-v4/tree/master/ecmascript. Accessed: 2015-09-23.

[10] Ecmascript international. http://www.ecma-international.org/. Accessed: 2015-09-23.

[11] Filesaver.js. https://github.com/eligrey/FileSaver.js. Accessed: 2015-09-02.

[12] Flowtype. http://flowtype.org/. Accessed: 2015-09-24.

[13] Font awesome. https://fortawesome.github.io/Font-Awesome/. Accessed: 2015-09-02.

[14] Javascript as ecmascript implementation. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction#JavaScript_and_the_ECMAScript_Specification. Accessed: 2015-09-23.

[15] Javascript tcp and udp socket api by w3c. http://www.w3.org/TR/2015/NOTE-tcp-udp-sockets-20150723/. Accessed: 2015-09-06.

[16] jeromq, zeromq binding for java. https://github.com/zeromq/jeromq. Accessed: 2015-09-23.

[17] Jquery. https://jquery.com/. Accessed: 2015-09-02.

[18] Json. http://json.org/. Accessed: 2015-09-05.

[19] Json-simple. https://github.com/fangyidong/json-simple. Accessed: 2015-09-23.

[20] Monto. https://bitbucket.org/inkytonik/monto. Accessed: 2015-09-03.

[21] Monto broker. https://github.com/monto-editor/broker. Accessed: 2015-10-05.

[22] Monto broker by sven keidel. `https://github.com/svenkeidel/monto-broker`. Accessed: 2015-10-05.

[23] Monto eclipse plug-in by sven keidel. `https://github.com/svenkeidel/eclipse-monto`. Accessed: 2015-10-05.

[24] Monto message conventions. `https://github.com/monto-editor/message-conventions`. Accessed: 2015-10-05.

[25] Monto organization at github. `https://github.com/monto-editor`. Accessed: 2015-10-05.

[26] Monto service base library for java. `https://github.com/monto-editor/services-base-java`. Accessed: 2015-10-05.

[27] Monto services for java. `https://github.com/monto-editor/java`. Accessed: 2015-10-05.

[28] Monto services for javascript. `https://github.com/monto-editor/services-javascript`. Accessed: 2015-10-05.

[29] Monto web-based dde. `https://github.com/monto-editor/editor-browser`. Accessed: 2015-10-05.

[30] Posix regular expression standards. `http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html#tag_09_03_05`. Accessed: 2015-10-06.

[31] Websockets availability in browsers. `http://caniuse.com/#feat=websockets`. Accessed: 2015-09-06.

[32] Zeromq. `http://zeromq.org/`. Accessed: 2015-09-05.

[33] Zeromq websocket draft. `http://rfc.zeromq.org/spec:39`. Accessed: 2015-09-06.

[34] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, Volume 35(Issue 2):114–131, June 2003.

[35] Sven Keidel. A disintegrated development environment. Master's thesis, Technische Universität Darmstadt, April 2015.

[36] Anthony M. Sloane, Matthew Roberts, Scott Buckley, and Shaun Muscat. Monto: A disintegrated development environment. In *Software Language Engineering*, volume 7, pages 211–220. Springer, November 2014.

[37] Krzysztof Stencel and Patrycja Węgrzynowicz. Implementation variants of the singleton design pattern. In *OTM '08 Proceedings of the OTM Confederated International Workshops*, pages 396 – 406. Springer, November 2008.

[38] Arie van Deursen, Ali Mesbah, Bas Cornelissen, Andy Zaidman, Martin Pinzger, and Anja Guzzi. Adinda: a knowledgeable, browser-based ide. In *ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2, pages 203–206. ACM New York, May 2010.