TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Andrei Šukurov 154855IAPB

EXTENSIBLE OPEN-SOURCE GALLERY AGGREGATOR

Bachelor's thesis

Supervisor: Gert Kanter
PhD

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Andrei Šukurov 154855IAPB

LAIENDATAV AVATUD LÄHTEKOODIGA GALERIIAGREGAATOR

bakalaureusetöö

Juhendaja: Gert Kanter
PhD

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Andrei Šukurov

23.05.2021

Abstract

The aim of this thesis is to develop an open-source gallery aggregation application for different photo-sharing services provided as plug-ins.

This work provides an overview of the technologies used to implement an extensible application.

As a result, the application is implemented using microservice architecture. Protocol schemas compiler and encapsulating messaging framework were implemented as side projects.

The whole project, including the distribution files, is licensed under a single MIT licence.

This thesis is written in English and is 43 pages long, including 6 chapters, 1 figure and 1 table.

Annotatsioon

Laiendatav avatud lähtekoodiga galeriiagregaator

Antud bakalaureusetöö eesmärgiks oli luua avatud lähtekoodiga platvormidevaheline laiendatav galeriiagregaator.

Terve projekt on litsentseeritud ühe MIT-litsentsi alusel. Ükski levitamisfail ei oma välissõltuvuse. Platvormist sõltumatus saavutatakse JavaScript'i abil. Laiendatavus saavutatakse mikroteenuste arhitektuuri põhimõtete abil.

Galeriiagregaator koosneb kahest eraldi kompileeritud osast - deemonist ja graafilisest rakendusest.

Galeriiagregaatorit saab laiendata kahel viisil - pistikprogrammide või IPC kaudu.

Mikroteenuste arhitektuuri rakendamiseks loodi kaks abivahendit - protokolli skeemide kompilaator ja kapseldav sõnumivahetuse raamistik. Kompilaator genereerib TypeScript'i koodi ja pakub võimalust määratleda oma tüüpe ja kasutada avaldisi. Raamistik kapseldab valideerimis- ja koostamisloogika ja saadab õiged sõnumid valitud suunas.

Lisaks tutvustatakse projekti tulevikuplaane. Antud bakalaureusetöö raames loodud rakendused on osa suuremast süsteemist.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 43 leheküljel, 6 peatükki, 1 joonist, 1 tabelit.

List of abbreviations and terms

AMD64	A 64-bit Processor Architecture
API	Application Programming Interface
ARM	Advanced RISC Machines
AVA	Test Runner for Node.js
c8	Code Coverage Reporting Tool for Node.js
Chromium	Open-Source Web Browser
Codecov	Code Coverage Solution for CI
CSS	Cascading Style Sheets
Deno	Runtime Environment for JavaScript
Electron	Combination of Chromium and Node.js
eslint	Pluggable JavaScript Linter
Fetch API	Interface for Fetching Resources
GTK	Cross-platform Widget Tool
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IPC	Inter-Process Communication
JavaScript	Interpreted Programming Language
jsdom	JavaScript Implementation of Many Web Standards
JSON	JavaScript Object Notation
Node.js	Runtime Environment for JavaScript
npm	Node Package Manager
PGP	Pretty Good Privacy
Pug	Template Engine
TCP	Transmission Control Protocol
Terser	JavaScript Compression Tool
TypeScript	Compiled Programming Language
Uint16Array	Array of 16-bit Unsigned Integers
webpack	Module Bundler for JavaScript

WebSocket	Communication Protocol
x86	A 32-bit Processor Architecture
XML	Extensible Markup Language

Table of Contents

List of Figures

List of Tables

1 Introduction

Today, there are many online photo-sharing services. Each of them comes with its own unique user interface. In addition, multiple unofficial clients exist for many of these services.

This overabundance of software creates some inconveniences for the end user. If the user wants to use the software, it is necessary to learn its interface. The more services the user uses, the more interfaces it is required to learn. The interfaces constantly change, causing the need to relearn the software. If the user wants to use multiple software at the same time, the result will be reduced productivity because of the constant switching. If the software is not a website, the user must install it on the system. If the system is based on packages (almost all Linux distributions) and the user forgets the name of the installed software, this can effectively cause loss of hard disk space due to unused software using space. In addition, every software can have security issues - the more applications user installs, the less secure the system becomes.

To avoid the problems described above, it is better to have one application for different services. Such an application should be extensible with plug-ins because different services take different approaches to retrieve the content.

To facilitate the distribution, the entire project should be released under a single MIT licence. The same licence must be applied to compiled files, which means that project is not allowed to have dependencies.

The application cannot be a website because websites do not have direct access to the file system. Direct access is needed to increase the variety of plug-ins. For example, to create a plug-in that acts as a file manager.

2 Choice of Technology and Requirements

2.1 Licence

To facilitate the distribution, the entire project should be released under a single permissive licence. The problem is to choose a licence that is compatible with widely used open-source licences [1]. This is required to allow this project to be used in other projects. In theory, to give others unlimited rights, the software could be released into the public domain. However, the public domain is ambiguous in some countries and requires additional legal steps [2]. Also, some companies (for example, Google [4]) do not accept patches released under the public domain. Therefore, a licence with an attribution clause must be chosen. To be more compatible, the licence should be chosen from the list of the most popular permissive licences. MIT, Apache, and BSD are the most popular permissive licences [3]. But only the MIT licence can be safely combined with all of them (if a project with the MIT licence tries to use a project with the Apache licence, the resulting licence will be Apache) [1]. Therefore, for the best compatibility, this project must be licensed under the MIT licence.

It is important not to have dependencies that must be included in the distribution files. Otherwise, such dependencies will affect the final licence. It is also important to check the licences of the build dependencies (and the licences of their dependencies), as they may affect the distribution files. Also, third-party dependencies can become proprietary at any time.

The ease of distribution is not the only reason to have such strict requirements. The lesser third-party dependencies a project has, the more secure it will be. During development, this project had already encountered a security problem caused by reliance on a third-party vendor [5]. But due to strict deployment policies, this backdoor did not affect this project. This incident justifies the chosen intolerance to third-party dependencies.

2.2 Messaging and Plug-ins

The application has two main requirements that affect the entire architecture:

1. The application must be extensible with plug-ins.
2. The application must be transparent to other applications (other applications should be able to track progress and interact).

Plug-ins could be implemented monolithically by defining statically defined methods and making direct calls to them. For example, this is the approach used in webpack [6]. This approach is good in many ways: it is fast, and it allows the use of a single address space, which leads to an efficient exchange of objects. However, this approach creates difficulties in implementing inter-process communication (IPC) because IPC uses an entirely different concept called messaging. Since the application must be able to interact with other applications, the IPC is required. Therefore, it may be better to use the same messaging concepts for plug-ins, as it would be possible to send messages from plug-ins directly to other applications. These concepts should also have a positive impact on testing because plug-ins will be autonomous [7]. To implement such a concept, it is better to divide the application into small parts called controllers. In this case, a plug-in will be a set of controllers. The API will not be restricted since each controller will have the same privileges.

2.3 Messaging Framework

To satisfy the licensing requirements (see Section 2.1), the messaging system has to be created from scratch. The requirements for the framework:

- Complete encapsulation of validation logic of incoming messages.
- Complete encapsulation of creation logic of outgoing messages.
- Complete encapsulation of incoming directions.
- Complete encapsulation of outgoing directions.

- Complete encapsulation of a protocol.
- Must provide events for directions and errors.
- Must catch and redirect errors to a certain event.
- Must provide types for messages.
- Cross-platform (must work in Node.js, Deno and web browser environments).

2.4 Validation of Messages and Protocol Schema Compiler

Since the validation of messages is an independent task, it is better to separate it from the framework (see Section 2.3). There are various tools available for JavaScript to provide JSON validation (see Table 1). But none of them meet the licensing requirements (see Section 2.1). In addition, almost all of them use JSON Schema as the standard, which has disadvantages – it is too verbose and has limitations. JSON Schema allows developers to operate with only known keywords, which is not enough in practice. Because of this, there are extension projects which add additional keywords on top of validation packages (*ajv-keywords* for *ajv*, as an example). As a result, projects are moving away from the original idea of cross-platform schemas. In addition, none of the reviewed tools offer the ability to build an object according to a schema. This functionality is required to omit default values and identify possible mistakes. All these tools can generate only JavaScript code, which can lead to difficulties in linking with TypeScript. Therefore, there is a need to develop a compiler that meets all the following requirements:

- Compilation into TypeScript.
- The compiler must generate appropriate types for objects defined in a schema.
- The compiler must generate validation logic for objects defined in a schema.
- The compiler must generate creation logic for objects defined in a schema.
- The generated code must be compatible with the messaging framework (see Section 2.3).

Table 1. Comparison of the most popular (according to npm weekly downloads) JSON validators available for JavaScript.

Package Name	**Vocabulary**	**Imported at Runtime**	**Affects Licence**	**Encapsulates Creation**
ajv	– JSON Schema draft-06/07/2019-09/2020-12 – JSON Type Definition RFC8927	Yes	Yes	No
djv	JSON Schema draft-04/06	Yes	Yes	No
joi	joi API	Yes	Yes	No
jsonschema	JSON Schema draft-04	Yes	Yes	No
z-schema	JSON-Schema	Yes	Yes	No
@exodus/ schemasafe	JSON Schema draft-04/06/07/2019-09	Yes	Yes	No
is-my-json-valid	JSON Schema draft-04	Yes	Yes	No

2.5 Communication

2.5.1 Application Layer

The communication between controllers (see Section 2.2) must be performed using messages in the form of JSON. The reason is the ease of integration. For example, JSON is used in the communication between the renderer and the main process of Electron [13], in the communication between processes in Sway, in the communication between browser extensions and native applications [14]. Parsing JSON is also faster than, for example, XML [12]. Also, since JSON is a subset of JavaScript [15], it simplifies the serialization.

2.5.2 Transport Layer

To deliver messages in the form of JSON from one process to another, a transport protocol is required. It is important not to lose data, so the TCP protocol should be used. Since the absolute limitation of TCP packet size is 64 kilobytes (16 bits) [16] and a message in the form of JSON does not have size limitations, it is necessary to apply an extra protocol level on top of TCP. It is not possible to use HTTP protocol because it does not allow to send a message from the server to a client without a request. In theory, could be used WebSocket protocol. But, unfortunately, Deno (see Section 2.7) does not provide WebSocket as a part of the runtime API but provides it as a standard library [17], which is against licensing requirements (see Section 2.1). Therefore, it is necessary to develop a simple protocol that will be optimal for sending JSON messages. The protocol used to communicate browser extensions with native applications [14] can be used as a basis. Since it is required to send a message over a network, it is necessary to explicitly define the endianness of a message length. The browser extension protocol uses native byte order, which is not network-friendly since destination and source can use different endianness. This is more theoretical problem because this project targets the x86 and AMD64 architectures, which are little-endian [18], and ARM, which is bi-directional but uses little-endian by default [19]. But still, it is better to resolve the potential conflict. In the Internet protocol suite exist convention to use big-endian order [20]. However, in this project, it is better to use little-endian order. There are two reasons for that. Firstly, since this is the default endianness of the targeting architectures, it should be easier to implement message length parsing in third-party applications. Secondly, it will open an optimization option if there would be a need to implement a browser extension, because it will be possible to send incoming message pieces directly to a browser without having to re-encode them.

2.6 TypeScript

The application must work on multiple operating systems. Therefore, it is better to choose an interpolating language to avoid recompiling for each platform. One of such languages could be JavaScript. It has two main advantages that set it apart from the others: it is used as the default interpolation language in most major web browsers, and it allows programmers to have one code base for the whole solution. Since JavaScript is

an interpolating language, it also allows projects to have the same licence for source and distribution files. It also should suit well for plug-ins implementation (see Section 2.2) [7]. However, JavaScript can create difficulties because it does not provide type checks. Therefore, it is better to choose TypeScript over JavaScript because it provides type checks and all benefits of JavaScript (TypeScript is compiled into JavaScript). The TypeScript compiler is licensed under the Apache 2.0 licence [8]. Unlike, for example, PureScript, which adds its own code to the bundle file, the TypeScript compiler acts as a transpiler and produces code without any additions to a project licence. But, in some cases, TypeScript compiler can generate code that will have an additional licence [9]. An example of such a case would be the use of EcmaScript decorators. In this case, the compiler will add a function from the *tslib* library, which is licensed under the Apache 2.0 licence. Therefore, the compiler should be used with caution to satisfy the licensing requirements (see Section 2.1) and language features that require external functions must be omitted.

2.7 Deno

To execute JavaScript outside of a web browser, a back-end JavaScript runtime environment is required. The most popular JavaScript runtime environment is Node.js. Unfortunately, Node.js does not implement Fetch API which makes it poorly compatible with web browser environments [10]. The compatibility is essential for better portability (to use one code base for front-end and back-end). Therefore, it is better to use an alternative for Node.js called Deno, which is designed with browser compatibility in mind [11].

2.8 Electron

To implement a graphical interface, a widget toolkit is required. Toolkits such as GTK can be effective but must be included in the distribution files, which is against licensing requirements (see Section 2.1). To satisfy the licensing requirements, the toolkit must be overlayed by a platform capable of running JavaScript. An example of such a platform is a web browser. Unfortunately, it is impossible to directly use a web browser, because of the messaging protocol (see Section 2.5.2). Web browsers have a lot of limitations due to security policies. Therefore, it is necessary to have a platform that bypasses these

limitations. One of such platforms is Electron. Electron combines in itself Chromium and Node.js. Unfortunately, there is no such solution that uses Deno instead of Node.js. For Deno exist only a third-party Webview module [21], but it still under development and does not satisfy this project licensing requirements. The standard graphical user interface for Deno is only on the discussion stage in Deno [22]. Therefore, Electron is the only platform that satisfies the requirements. Electron cannot use Deno because Electron embeds Node.js [23]. Therefore, this project should also partially support Node.js.

2.9 Pug

Since the graphical interface does not target the Web, it is better to use templating language over HTML. Templating languages allow developers to split a single representation file into multiple files, while HTML does not have such functionality. This functionality is essential for the efficient implementation of single-page applications because such division should make code more readable. The single-page architecture should reduce the response time of the application [26]. Pug templating engine, compared to EJS, provides a more compact and readable syntax. Pug is minimalistic and should fulfil all needs of this project.

2.10 Node Package Manager and The JavaScript Package Registry

To avoid manual installation of dependencies and to automate the build and versioning processes, an appropriate build system should be chosen. In the case of Deno, the Deno command-line tool could be used [24]. Since this project also targets Node.js (see Section 2.8), it is better to use Node Package Manager. In addition, compared to Deno, the Node Package Manager also provides a more friendly to the TypeScript compiler import system [25]. Node Package Manager offers an ability to publish public packages in their database called registry. The registry should be used to version all parts of the project. This approach should make progress more traceable and allow developers to use different versions of the same package, which should be helpful in the development of the application protocols.

2.11 webpack

To link all parts of the project together, a linking tool is required. Deno does not support Node.js imports, which should be used because of npm, so the linking tool should be able to resolve them. In addition, the linking tool must be able to prepare the code to work in Electron. The linking tool must also be able to link HTML and CSS files. All requirements are met in webpack. webpack consists of plug-ins. It is therefore important to choose plug-ins that do not violate licensing requirements.

2.11.1 Terser Webpack Plugin

One of the key features of webpack is compression. By default, webpack uses the Terser project for compression [33]. Terser is licensed under the 2-clause BSD licence [34]. Terser does not include its source codes or its binaries on compression into compressed files. Therefore, the licence should not affect the resulting files.

2.11.2 HTML Webpack Plugin

For the graphical user interface, it is required to inject JavaScript and CSS files into HTML files. The HTML Webpack Plugin owns all required features. The plug-in is licensed under MIT licence and uses Terser (see Section 2.11.1). The plug-in meets the licensing requirements (see Section 2.1).

2.11.3 HTML Loader

This plug-in is required by HTML Webpack Plugin (see Section 2.11.2). The plug-in is licensed under MIT licence and uses Terser (see Section 2.11.1). The plug-in meets the licensing requirements (see Section 2.1).

2.11.4 Mini CSS Extract Plugin

For the graphical user interface, it is required to concatenate CSS files into one and pass the link to the HTML Webpack Plugin (see Section 2.11.2). The plug-in is licensed under MIT licence and meets the licensing requirements (see Section 2.1).

2.11.5 CSS Loader

This plug-in is required by Mini CSS Extract Plugin (see Section 2.11.4). The plug-in is licensed under MIT licence and meets the licensing requirements (see Section 2.1).

2.11.6 Copy Webpack Plugin

This plug-in is required to perform a file copying operation and should be used to form distribution directories. The plug-in is licensed under MIT licence and meets the licensing requirements (see Section 2.1).

2.11.7 Source Map Loader

This plug-in is required to generate the GUI source code map file, which is required to link the source code to the distribution file for unit testing purposes and code coverage calculation. The plug-in is licensed under MIT licence and meets the licensing requirements (see Section 2.1).

2.12 AVA

To cover the project with unit tests, a testing framework is required. In theory, could be used Deno built-in test runner. But at the time of selection, the coverage function was marked as unstable [27]. It was therefore decided to choose a test runner for the Node.js environment. There are various test runners for Node.js. For example, to the most known belong Jasmine, Mocha, Jest, and AVA. Jest is the most popular of them, but AVA has the most minimalistic syntax. All these runners except AVA use chaining for writing assertions, which makes the syntax overwhelmed. The only disadvantage of AVA is no ability to write mock tests. But this project does not require this feature because every code part should be reachable by messaging (see Section 2.2). If a situation arises in which it is necessary to use mocking, a third-party provider of mocking can be used [28]. AVA is released under MIT licence, and since a test runner is not going to be distributed by this project, it meets all the requirements of the project and can be selected for use.

2.13 c8

As test coverage needs to be monitored and AVA (see Section 2.12) does not generate the *lcov* reports that Codecov (see Section 2.17) requires, a tool that will generate them must be used. The well-known test coverage tools for Node.js are nyc and c8. The nyc is the most popular, but c8 is used by the AVA project. So, for better compatibility, c8 should be used.

2.14 jsdom

To test the graphical interface using AVA (see Section 2.12), it is necessary to simulate a web browser environment [29]. Therefore, jsdom was chosen as a recommendation from the AVA project.

2.15 ESLint

Since this project should be extensible by using plug-ins (see Section 2.2), it is necessary to use a JavaScript linter as a build tool to warn developers about unsafe usages of Function to avoid possible threats [30]. Linter is also needed to follow the code style. ESLint is chosen for this project as the most popular linter [31].

2.16 GitHub Actions

To be more transparent, it is better to expose the build status to the public. Therefore, a server that will run the project with every commit is required. There are various solutions available that provide that opportunity (for example, Jenkins, Travis, CircleCI). Since the project is hosted on GitHub, it is better to use GitHub Actions to reduce the number of third parties involved. The use of continuous integration principles should also reflect positively on the development. Therefore, the project should avoid using *package-lock.json* files and mark the versions of development dependencies as the latest in the *package.json* files. This approach may seem like an anti-pattern [32], but it will force the project to adapt the code for the latest changes as soon as possible, so the project will not use deprecated technologies.

2.17 Codecov

To be more transparent, code coverage should be made publicly available. Unfortunately, GitHub (see Section 2.16) does not provide any code coverage solutions. Therefore, a third-party solution should be used. Since this project uses the same technology as the AVA project (see Section 2.17), it is possible to use the same solution to upload the coverage. Therefore, Codecov is chosen. It fulfils all the needs of this project.

3 Implementation

3.1 Messaging Framework

3.1.1 Overview

The framework defines logic for two entities. The first entity is called *Message Center*. The second is called *Controller*. The *Message Center* shall be treated as an entry point to an application. It receives messages from an external direction and asynchronously forwards them to all controllers. In addition, it defines events and provides an ability to listen for a specified event. The *Controller* provides a way to handle a message and asynchronously send back another message.

The framework represents *Message Center* and *Controller* as JavaScript classes. And exposes factory functions that dynamically construct classes according to a protocol. The factory functions will construct and inject all required by a protocol methods and events. These functions also provide strict types, so the protocol is fully encapsulated.

3.1.2 Limitations

It is not possible to wait until a message has reached its destination, so the sending is not reliable. This limitation is added purposely to avoid possible memory leaks. For example, if one direction sends a message to the other direction and the other direction sends a message that triggers the original sender, the memory will grow until the stack overflow.

The dynamically defined processing and sending methods are exposed as class properties, not as class methods. This is due to the fact that TypeScript does not provide a way to apply template literal types on method names.

3.1.3 Events

The *Message Center* uses two different statically defined events for errors. First for protocol errors (*protocol-error*) and second for controller errors (*controller-error*). This approach should help to separate internal bugs (controller errors) from externally caused errors (protocol errors).

The *Message Center* dynamically defines (according to passed protocol) events for directions. Messages sent by a controller will trigger the corresponding *message-to-direction* event, where the *direction* is the actual name of the direction specified in the protocol and converted to lowercase. Messages sent to a controller will trigger the corresponding *message-from-direction* event.

3.1.4 Message Sending

The *Message Center* dynamically defines a sending method per direction in the form of *sendToDirection*. These methods trigger corresponding sending events (*messate-to-direction*). To be able to receive messages sent by a controller, the direction must attach a listener to the appropriate event.

The *Controller* dynamically defines sending methods in the form of *sendToDirectionMessageHeader*. When a controller sends a message, the *Message Center* receives, identifies, completes (applies the creation method on it), and sends the copy (because of the completion) of the message to the corresponding *sendToDirection* method.

If an error occurs during this phase (which is only possible if the creation method within the protocol is broken or types have been mechanically ignored), the *protocol-error* event will be dispatched.

3.1.5 Message Receiving

The *Message Center* dynamically defines a processing method per direction in the form of *processFromDirection*. These methods perform the validation by using validation methods provided by a protocol. In case if a message is not valid, the *protocol-error* event will be dispatched. Otherwise, the corresponding event (*message-from-direction*) will be triggered. And the message will be passed to all controllers by triggering

methods defined in the form of *processFromDirectionMessageHeader*. If a processing method throws an error, the *controller-error* event will be dispatched.

3.1.6 Protocol Structure

The framework accepts an object as a protocol (see Appendix 2). The object can have keys in the form of *ALLOWED_MESSAGES_TO_DIRECTION*. Such keys define the messages that can be sent in the specified direction. To describe messages that can be sent from a direction, the keys must be in the form of *ALLOWED_MESSAGES_FROM_DIRECTION*. These properties contain an array of message groups. A group is a JavaScript class that defines messages as static properties. A message is a JavaScript object with two methods *validate* and *create*. The *validate* method accepts any type of message and returns an array of objects describing the errors that occurred. The *create* method accepts a strongly typed JavaScript object with all mandatory (or optional) fields. This method must return a copy of the original object with all missing default values included.

The protocol should not be described manually because of potential mistakes in the *create* and the *validate* methods. Therefore, this framework should be used only in pair with the protocol schema compiler.

3.2 Protocol Schema Compiler

The compiler is designed to help represent protocols for the messaging framework (see Section 3.1.6). For an example of a protocol, see Appendix 3.

3.2.1 Architecture

The compiler is designed with portability in mind. Currently, the compiler uses Deno (see Section 2.7) as the runtime environment. The way to add Node.js or the browser environment is simple, as file system operations are encapsulated by adapters. To simplify the use of compiler with *package.json*, the build tool, which requires Deno, is called from a Node.js script. This allows developers to use the alias (*fsc*) in *package.json*.

The compiler consists of three main groups:

- Compilation registry
- Registration controllers of models
- Output code registration controllers

The core part of the compiler is the compilation registry. The registry defines registration events and is responsible for informing controllers about an event. Whenever a structure enters the registry, the registry informs about that the corresponding controllers. The controllers then decide whether the structure is suitable for them, and if so, they can create a new structure and register it in the registry. For example, if an interface model schema enters the registry, the interface model registration controller takes the schema and creates an interface model, then registers that model in the registry. The interface code creation controller then takes the model from the registry, generates the appropriate lines of code, and places them in the registry. After that, the file writing controller takes the lines from the registry and writes them into the output file. This architecture is supposed to be reactive, so structures are not allowed to refer to not yet registered structures.

The compiler presents all protocol structures as models. If a model can directly depend on other models, these models will be inserted into the Binary Expression Tree of this model. After that, the compiler will not be required to directly work with expressions, or the tree, since the compiler is able to minimize the Binary Expression Tree into a disjunctive array (in all situations). This approach has helped to make the code cleaner.

The use of models makes the compiler more abstract. Therefore, it should be possible to design a custom language in the future, instead of schemas in the JSON format, without redesigning the compilation logic.

The need to migrate to a custom language is obvious because of the limitations of JSON. As an example, JSON does not allow to add comments to a logic. It also does not add syntax highlights to expressions. But designing a language is too complex a task for this thesis, so JSON was chosen.

3.2.2 Primitive Values

The compiler provides a way to use primitive values as a type. The complete list of primitive values:

- Strings (must be wrapped in single quotes)
- Numbers
- *true*
- *false*
- *null*

3.2.3 Custom Types

The main advantage of this compiler is the ability to define custom types. The compiler has no hard-coded primitive types (arrays and tuples are an exception, as they are generic). The compiler provides primitive types defined through the schema. These types are:

- *TString*
- *TNumber*
- *TInteger*
- *TByte*
- *TBoolean*

The compiler provides an easy way to define a custom type. It allows developers to use JavaScript to write the restrictions. But unfortunately, the compiler currently evaluates that code, so the usage of untrusted schemas can lead to arbitrary code execution.

3.2.4 Type Expressions

The compiler implements three basic operations required to form expressions:

- Conjunction

- Disjunction
- Parentheses

Conjunctions can only be used with compatible types, otherwise a compilation error will be thrown. The compiler is intelligent enough to detect incompatible types in complex expression. This is achieved by converting the Binary Expression Tree into a disjunctive array.

The disjunction can be used to specify the default value to a field. In this case, the first self-contained model will be used as the default value. A self-contained model is a model that does not have properties with no default value specified (deeply). Arrays cannot be used as default values, but tuples can.

3.2.5 Arrays

The compiler allows the use of arrays of any depth. It is also allowed to use type expressions inside of an array. The compiler will handle them and generate appropriate creation and validation logic.

3.2.6 Tuples

The compiler allows the use of tuples of any depth and the use of type expressions inside of a tuple.

3.2.7 Interfaces

The compiler provides the ability to define interfaces. The syntax of interfaces is fully compatible with objects and classes. The compiler compiles the interfaces into TypeScript interfaces.

3.2.8 Objects

The compiler provides the ability to define objects. The syntax of objects is fully compatible with interfaces and classes. The objects are compiled into JavaScript objects (dictionaries). To construct an object, the compiler will collect all default values of the structure.

3.2.9 Classes

The compiler provides the ability to define classes. The syntax of classes is fully compatible with interfaces and objects. The classes are compiled into JavaScript classes. The *validate* and the *create* methods will be created for every class property.

3.2.10 Mixed Structures

Mixed structures are an alternative way of representing interfaces, classes, and objects. This way should be used if it is required to generate structures with the same properties.

At the model level, mixed structures do not exist. When the compiler receives this structure, it transpiles it into specified structures. Therefore, it is not possible to refer to a mixed structure.

3.2.11 Inheritance

The properties of Objects, Interfaces, and Classes can be inherited by using the *@ancestors* keyword.

3.2.12 Import

It is possible to import a structure (or all structures) from another schema file. The *@import* keyword should be used.

3.3 Transport Protocol

The protocol package contains two classes – decoder and encoder.

The encoder has a method that converts the object to JSON, encodes the string into a buffer, calculates the buffer size, adds four extra bytes to the beginning of the buffer, and puts the calculates size to the added bytes using little-endian byte order.

The decoder provides a JavaScript generator method. It accepts chunks of messages, and if a message is ready, it yields it.

The encoder and decoder also provide an ability to choose the endianness. This feature is added to make the protocol package more universal and add a way to implement the connection with a browser as an extension in the future (see Section 2.5.2).

3.4 Gallery Aggregator Architecture

The gallery application is divided into two main parts – the daemon and the graphical application (see Figure 1). This division is required because the application must be transparent for other applications (see Section 2.2). With such division, it should be simple to connect to the daemon application and listen for messages that it sends to the graphical application. It also allows other applications to communicate with the graphical application because every connected application has the same level of priority and trust. Since the graphical application and the daemon should communicate with each over, they share the same protocol. Therefore, this protocol is separated into another repository. This separation makes easier the versioning of the protocol and avoids duplicates.

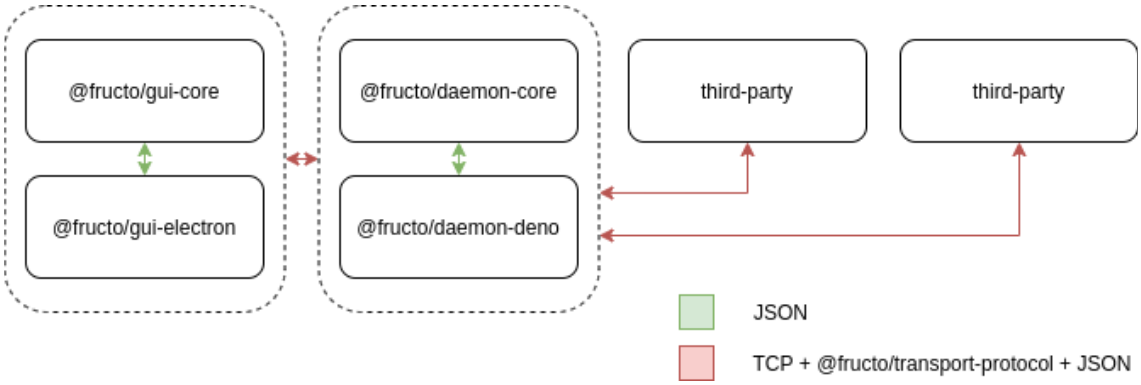

Figure 1: Communication between the parts of the aggregator

3.5 Gallery Aggregator Protocols

The protocols are described and compiled into TypeScript using the schema compiler (see Section 3.2). The TypeScript then is compiled into JavaScript, which is then distributed through the npm registry.

The package contains separately compiled protocols for every part of the project. The *package.json* file defines a custom entry point for every protocol using the *exports* and *typesVersions* fields.

3.6 Gallery Aggregator Daemon Application

The daemon application is divided into two parts - the platform-independent core and the application part, which uses Deno (see Section 2.7). The communication between these two parts is done using JSON (see Section 2.5.1). This division is required to make the code more portable and testable.

The core part is responsible for:

- Transmission of plug-ins.
- Launch of plug-ins.
- Transmission of settings.

The application part is responsible for:

- Connection with clients.

3.7 Gallery Aggregator Graphical Application

The graphical application consists of two parts – the core part, which requires the browser environment, and the application, which requires Electron (see Section 2.8). The communication between these two parts is done using JSON (see Section 2.5.1).

The core part is responsible for:

- Transmission of plug-ins.
- Transmission of settings.
- Graphical interface (input fields, buttons, translations).

The application part is responsible for:

- Connection with the daemon.

4 Validation of Results

The division into sub-projects opened great opportunities for testing. Almost every layer is covered with unit tests.

4.1 Messaging Framework

The line coverage of the messaging framework is 99%.

The unit tests check the following cases (this list shows only essential cases):

- Errors caused by a controller dispatch the *controller-error* event.
- Unknown messages dispatch the *protocol-error* event.
- Messages from a specific direction dispatch the corresponding event.
- Messages to a specific direction dispatch the corresponding event.
- Messages from a specific direction are transferred to controllers.
- Messages from a controller are transferred to the message center.
- The *setUp* method of a controller is called on attachment.
- The *setUp* method of a message center is called on creation.
- A message center attaches controllers from *CONTROLLERS* property if specified.

4.2 Protocol Schema Compiler

Due to unstable public API and constant refactoring, the compiler has only one smoke test, which covers 89% of the lines.

The smoke test covers the following features:

- Type Expressions (Conjunction, Disjunction, Parentheses)
- Default Custom Types (TString, TNumber)
- Primitive Values (Strings, Numbers)
- Inheritance
- Structures (Classes, Interfaces)

4.3 Transport Protocol

The line coverage of the transport protocol is 99%.

The unit tests check the following cases:

- Decoding of a message with the length stored in the big-endian order.
- Decoding of a message with the length stored in the little-endian order.
- Decoding of a message with the length stored in the native byte order (only one check).
- Encoding of a message using the big-endian order.
- Encoding of a message using the little-endian order.

Only one line of the protocol is not covered by the tests - the line detecting the native byte order. It is technically difficult to implement a test as it requires emulation of the big-endian architecture. *Uint16Array* replacement could be used as a workaround. But this approach can hide a potential bug, so it is better to leave the uncovered code clear.

4.4 Gallery Aggregator Protocols

The unit tests check that the messages are bound to the correct directions and that directions do not export messages unknown to the tests.

Due to the constant changes in the protocols, the code of the protocols is not covered with tests. This problem opens a new issue - to add the ability to describe tests to the schema compiler. But due to lack of time, this is not going to be implemented as part of this thesis.

4.5 Daemon Application

The line coverage of the core part is 98%.

The unit tests check the following activities of the core part:

- Transmission of settings from the daemon application to the daemon core.
- Transmission of settings from a client to the daemon core.
- Transmission of settings from the daemon core to a client.
- Transmission of plug-ins to from the daemon core to a client.
- Transmission of plug-ins from a client to the daemon core.

4.6 Graphical Application

The line coverage of the core part is 92%.

To test the core part of the GUI, it is first compiled into the distribution form. It is then passed to jsdom (see Section 2.14), to simulate running in a web browser. After that, it is possible to simulate clicks on the button from the unit tests. Because of the messaging (see Section 2.2), every button should initiate a message. The unit tests check the correctness of the messages.

The unit tests check the following activities:

- Transmission of settings from the client to the daemon.
- Transmission of settings from the daemon to the client.
- Transmission of plug-ins from the client to the daemon.

5 Future Plans

The project has numerous potential development ideas.

One important missing feature is a security system for plug-ins. The security system could be based on PGP signature verification.

To simplify the installation of plug-ins, a store should be created, from which it will be possible to install the plug-ins.

The uploading and downloading of images should also be implemented. These functions must be pausable, abortable, resumable. There also must be network traffic control mechanics.

A command-line interface is also required. It must be able to monitor the progress of downloading and uploading, initiate upload and download.

The protocol compiler must be able to generate JavaScript in addition to TypeScript. This will make it easier for JavaScript users to use the compiler. It is also required for dynamic use.

In addition, it is necessary to design a cryptography system for sending sensitive information over the network. Currently, the protocol is open and not encrypted.

Also, the application must be able to play video files. And the ability to use a proxy server must be present.

6 Summary

The goal of this thesis was to create an open-source, portable, and extensible image gallery aggregation application.

The project is licensed under a single MIT licence. The distribution files do not have any built-in dependencies. The portability problem was solved using JavaScript. The extensibility problem was solved using the principles of microservice architecture.

The application consists of two independently compiled programs – the daemon and the graphical application.

The application can be extended in two ways – by plug-ins or IPC.

To implement the microservice architecture were designed a protocol compiler and encapsulating messaging framework. The compiler produces TypeScript code and provides a way to define custom types and use type expressions. The framework encapsulates validation and creation logic, explicitly forces to send only valid messages to registered directions.

The results were validated using unit tests.

References

[1] D. A. Wheeler, “The Free-Libre / Open Source Software (FLOSS) License Slide,” Sep. 2007. [Online]. Available: https://dwheeler.com/essays/floss-license-slide.pdf. Accessed on: May 20, 2021.

[2] *Categories of free and nonfree software*, Free Software Foundation, Inc.. [Online] Available: https://www.gnu.org/philosophy/categories.html. Accessed on: May 20, 2021.

[3] B. Balter, “Open source license usage on GitHub.com,” Mar. 9, 2015. [Online]. Available: https://github.blog/2015-03-09-open-source-license-usage-on-github-com/. Accessed on: May 20, 2021.

[4] *Open Source Patching.* Google LLC. [Online]. Available: https://opensource.google/docs/patching/. Accessed on: May 20, 2021.

[5] *Bash Uploader Security Update*, Codecov. [Online]. Available: https://about.codecov.io/security-update/. Accessed on: May 20, 2021.

[6] *Writing a Plugin*. [Online]. Available: https://webpack.js.org/contribute/writing-a-plugin/. Accessed on: May 20, 2021.

[7] J. Huttunen, “Microservice Testing Practices in Public Sector Software Projects,” M.S. thesis, Sch. of Electr. Eng., Aalto Univ., Espoo, Finland, 2017. [Online]. Available: https://aaltodoc.aalto.fi/bitstream/handle/123456789/26673/master_Huttunen_Joel_2017.pdf. Accessed on: May 20, 2021.

[8] *The Licence of TypeScript*, TypeScript Repository, 2014. [Online]. Available: https://github.com/Microsoft/TypeScript/blob/master/LICENSE.txt. Accessed on: May 20, 2021.

[9] *TypeScript Import Helpers*. [Online]. Available: https://www.typescriptlang.org/tsconfig#importHelpers. Accessed on: May 20, 2021.

[10] *Suggestion to Add Fetch Api Into Node.js.* [Online]. Available: https://github.com/nodejs/node/issues/19393. Accessed on: May 20, 2021.

[11] *Deno Goals.* [Online]. Available: https://deno.land/manual#goals. Accessed on: May 20, 2021.

[12] A. Šimec and M. Magličić, “Comparison of JSON and XML Data Formats,” in *Central European Conf. on Information and Intelligent Systems*, 2014, pp. 272-275. [Online]. Available: http://archive.ceciis.foi.hr/app/public/conferences/1/papers2014/696.pdf. Accessed on: May 20, 2021.

[13] *Electron Documentation: ipcMain*. [Online]. Available: https://www.electronjs.org/docs/api/ipc-main. Accessed on: May 20, 2021.

[14] *Browser Extensions: Native Messaging*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messaging. Accessed on: May 20, 2021.

[15] *The JSON Data Interchange Syntax*, ISO/IEC 21778, 2017. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/. Accessed on: May 20, 2021.

[16] *Transmission Control Protocol*, RFC 793, 1981. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc793. Accessed on: May 20, 2021.

[17] *Deno WebSocket*. [Online]. Available: https://deno.land/std@0.95.0/ws. Accessed on: May 20, 2021.

[18] Advanced Micro Devices Inc.. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*. (2020). [Online]. Available: https://www.amd.com/system/files/TechDocs/24592.pdf. Accessed on: May 20, 2021.

[19] ARM Limited. *Cortex™-M3 Technical Reference Manual*. (2006). [Online]. Available: https://developer.arm.com/documentation/ddi0337/e/Programmer-s-Model/Memory-formats. Accessed on: May 20, 2021.

[20] *Assigned Numbers*, RFC 1700, 1994. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1700. Accessed on: May 20, 2021.

[21] *webview_deno*. (0.6.0-pre.0). [Online]. Available: https://deno.land/x/webview. Accessed on: May 20, 2021.

[22] *Deno Standard GUI Discussion*. [Online]. Available: https://github.com/denoland/deno/discussions/3234. Accessed on: May 20, 2021.

[23] *Electron Architectural Notes*. [Online]. Available: https://github.com/electron/electron/issues/23613#issuecomment-629763397. Accessed on: May 20, 2021.

[24] *Deno Script Installer Manual*. [Online]. Available: https://deno.land/manual/tools/script_installer. Accessed on: May 20, 2021.

[25] *Suggestion to Allow TypeScript Extensions in Imports*. [Online]. Available: https://github.com/microsoft/TypeScript/issues/38149. Accessed on: May 20, 2021.

[26] V. Solovei, O. Olshevska, and Y. Bortsova, "The Difference Between Developing Single Page Application and Traditional Web Application Based on Mechatronics Robot Laboratory ONAFT Application," *ATBP*, vol. 10, no. 1, Apr. 2018. [Online]. Available: https://journals.onaft.edu.ua/index.php/atbp/article/view/874/950. Accessed on: May 20, 2021.

[27] *Deno Testing Manual*. [Online]. Available: https://deno.land/manual@v1.7.4/testing. Accessed on: May 20, 2021.

[28] P. Ladaria. *An Example of Mocking with Sinon and Ava*. (2018). [Online]. Available: https://github.com/avajs/ava/issues/1829#issuecomment-414741074. Accessed on: May 20, 2021.

[29] *Setting up AVA for browser testing*. [Online]. Available: https://github.com/avajs/ava/blob/main/docs/recipes/browser-testing.md. Accessed on: May 20, 2021.

[30] K. Rieck, T. Krueger, and A. Dewald, "Efficient Detection and Prevention of Drive-by-Download Attacks," in 2*6th Ann. Computer Security Applications Conf.*, 2010, pp. 31-39. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1920261.1920267. Accessed on: May 20, 2021.

[31] K. F. Tómasdóttir, M. Aniche, and A. V. Deursen, “The Adoption of JavaScript Linters in Practice: A Case Study on ESLint”, *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 863 - 891, Aug. 2020. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8468105. Accessed on: May 20, 2021.

[32] *Configuring npm: package-lock.json*. [Online]. Available: https://docs.npmjs.com/cli/v7/configuring-npm/package-lock-json. Accessed on: May 20, 2021.

[33] *TerserWebpackPlugin*. [Online]. Available: https://webpack.js.org/plugins/terser-webpack-plugin. Accessed on: May 20, 2021.

[34] *The Licence of Terser*, Terser Repository, 2013. [Online]. Available: https://github.com/terser/terser/blob/master/LICENSE. Accessed on: May 20, 2021.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Andrei Šukurov

1 Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Extensible Open-Source Gallery Aggregator", supervised by Gert Kanter

1.1 to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;

1.2 to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.

2 I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.

3 I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

23.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Demo Protocol for @fructo/messaging-framework

```
class MessageToServer {
   static PING = {
      create(basis) {
         // Complex logic, manually created, goes here
         return basis;
      },
      validate(message) {
         // Complex logic, manually created, goes here
         return [];
      }
   }
}

class MessageFromServer {
   static PONG = {
      create(basis) {
         // Complex logic, manually created, goes here
         return basis;
      },
      validate(message) {
         // Complex logic, manually created, goes here
         return [];
      }
   }
}

const PROTOCOL_FOR_SERVER = {
   ALLOWED_MESSAGES_FROM_CLIENT: [MessageToServer],
   ALLOWED_MESSAGES_TO_CLIENT: [MessageFromServer]
};

const PROTOCOL_FOR_CLIENT = {
   ALLOWED_MESSAGES_FROM_SERVER: [MessageFromServer],
   ALLOWED_MESSAGES_TO_SERVER: [MessageToServer]
};
```

Appendix 3 – Demo Protocol for @fructo/schema-compiler

```
{
   "IPingMessage": {
      "@properties": {
        "header": "'ping'"
      }
   },

   "IPongMessage": {
      "@properties": {
         "header": "'pong'"
      }
   },

   "MessageToServer": {
      "@properties": {
         "PING": "IPingMessage"
      }
   },

   "MessageFromServer": {
      "@properties": {
         "PONG": "IPongMessage"
      }
   },

   "PROTOCOL_FOR_SERVER": {
      "@properties": {
         "ALLOWED_MESSAGES_FROM_CLIENT": "[MessageToServer]",
         "ALLOWED_MESSAGES_TO_CLIENT": "[MessageFromServer]"
      }
   },

   "PROTOCOL_FOR_CLIENT": {
      "@properties": {
         "ALLOWED_MESSAGES_FROM_SERVER": "[MessageFromServer]",
         "ALLOWED_MESSAGES_TO_SERVER": "[MessageToServer]"
      }
   }
}
```