Computer tools in education, 2019
№ 1: 43–53
http://ipo.spb.ru/journal
doi:10.32603/2071-2340-2019-1-43-53

software engineering

STRONG TYPING FOR EVENT-DRIVEN MICROSERVICE ARCHITECTURE

N. S. Gerasimov[1], postgraduate student, n.gerasimov@2015.spbu.ru

[1]St. Petersburg State University, 28, University pr., Petergof, 198504, Saint Petersburg, Russia

Abstract

Microservice architecture is a popular approach for building various systems. It consists of various dedicated small-sized components from a solid application. Components work together and provide the required business-logic, but each one can be implemented with the most suitable technique. Moreover, overloaded parts can be scaled easily. However, a data transfer layer does not guarantee compatibility of services during their development and maintenance. After changing input or output data type of any microservice of the system, there is a limited set of options to keep up the compatibility. The problem is especially urgent for asynchronous communication because existing systems and standards work mostly with synchronous RPC or REST interoperation or cover high-level description of business-logic. One way to guarantee compatibility is a preliminary check of data types before the services start to serve requests. The current paper describes the development and testing of the method applied to the event-driven system of microservices. The developed approach extends message broker RabbitMQ to store queue types. Also, the approach uses the library to check the microservices ability to use selected queues. Both the library and the extension solves the problem of type inconsistency in a microservice event-driven architecture.

Keywords: *microservice architecture, static typing, event-driven model, SOA, services composition.*

Citation: N. S. Gerasimov, “Strong Typing for Event-Driven Microservice Architecture,” *Computer tools in education*, no. 1, pp. 43–53, 2019 (in Russian); doi:10.32603/2071-2340-2019-1-43-53

1. INTRODUCTION

One of the modern popular approaches to design complex systems is the microservice paradigm [1–3], characterized by splitting the logic of a monolithic application into many separate software components, independent in their work from each other. The benefits of this approach are evident in the development of multi-functional high-loaded web services, the functionality, and architecture of which will no longer undergo fundamental changes. Horizontal scaling of individual components is greatly simplified as microservices usually do not require additional dependencies and have a minimum number of settings. Individual services with a limited range of tasks are easier to maintain and easier to organize quick updates delivery to production space. Besides, the microservice approach motivates developers to work out the logic separation in more detail.

However, there are several disadvantages, along with these advantages. Splitting logic into separate services requires the data transferred between individual components. It often occurs over the network using a message broker or HTTP Protocol, which causes additional time delays. Another weakness is data synchronization, which is likely to be duplicated in different services, or data in some services should be consistent with data in others.

Another typical problem for microservice architectures is the management of service communications and dependencies, as well as synchronization of data exchange formats. When developing a monolithic application, the problem of controlling components, their interaction and types of transferred data is solved using programming language and compiler (or interpreter), based on which this monolithic application is developed. In a microservice architecture, each component becomes a separate application, possibly written in a different programming language, so there is a lack of standard tools. Due to it, problems can occur with the support of the system components composition. Firstly, it is necessary to record microservices interaction. Secondly, it needs compliance of data types expected in the microservice server with ones actually sent by a microservice client. Divergence of types can lead to a situation when the error is detected only at the stage of testing or even operation of the system. Type checking should be made before the service starts.

Various systems have been created to solve the problems of types composition and compliance. However, the systems have several disadvantages, e.g., orientation on the high-level business process (a collection of related tasks to obtain a specific service or product) description. Moreover, existing options mainly consider service-oriented architectures (SOA) and synchronous interacting, whereas no solutions have been found for asynchronous architectures. Lack of solutions is an urgent problem, as asynchronous systems using a message broker as a data channel are popular.

Due to these difficulties, there is a need to develop a tool to describe the interfaces of microservices within the asynchronous model of interaction. Such a tool should automatically determine the compatibility of their interfaces before running using a description in common for all services format of the types of data being transferred. This tool should ensure that service implementation matches the description of the types of transferred data, where the service provides the description. The description form of the interface and types of data transferred should be as simple as possible and compatible with common methods of data description or validation. The use of such a tool should improve the stability of a microservice system in the process of its development and support.

Following the above, the purpose of the work is to develop a tool for verifying the compatibility of microservice interfaces.

In Chapter “State of the art”, we considered existing approaches used in SOA since microservice architecture is an ideological continuation of service-oriented architectures. The found variants of the standards for the description of orchestration and composition do not suit for several reasons, including due to the orientation to the client-server communication model, unnecessary complexity and focus on high-level business process description.

In Chapter “Principle of suitable interfaces discovery”, we considered our version of the service composition system proposed in the previous work [14]. Further development of our system was suggested on its basis in Chapter “Message broker typing support”. In Chapter “Message broker typing support” we focused on supporting only event-oriented architecture and abandoned the central connecting element. This has simplified the process of creating new services and supporting existing ones. In contrast to the existing alternatives, the proposed solution focuses on asynchronous services operation.

The novelty of the approach consists in using type system and in using contracts that are generating from the source code.

2. STATE OF THE ART

A modern web-application built in microservice paradigm consists of multiple independent containerized services as shown at [1]. Each service is developed independently, uses suitable programming language, suitable database, and realizes a limited set of tasks.

Image 1 demonstrates the example of journal-system realizing microservice architecture [1]. Service “front” renders the view of a journal; “entry store” one store records with journal posts; “post” creates new posts in store and in “index” that implements search index. The scheme demonstrates that the collection of the services implements the journal, but each one separated from others is useless. The services are often communicating via lightweight REST protocol or event dispatcher like AMQP server. For example, “post” service might update the search index sending the message with the content of the new post entry via AMQP while “front” and service interacts with other ones via synchronous REST calls.

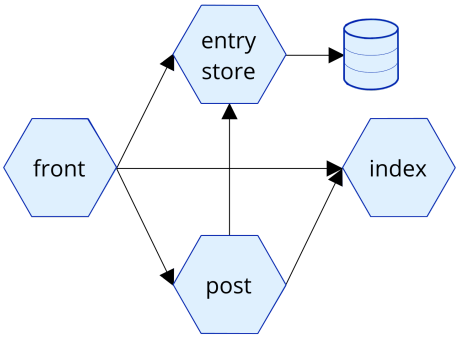

Figure 1. System example

Implementation of a specific business process uses various infrastructure modules like utilities and standards to describe module interfaces, service-discovery, and composition systems.

2.1. Composition

Since microservice architecture is a further development of service-oriented architecture (SOA), which appeared relatively long ago, services composition is well developed. There are approved standards for describing calling and composition sequence in SOA: WS-BPEL, WS-CDL, BPML, ebXML, OWL-S, WSMF, etc. However, they are designed mainly for business processes description. For example, WS-BPEL enables to describe differentiation between access permissions when giving a detailed account of business processes [4]. Microservices do not fully implement any particular business process [2, 3], and this possibility is excessive. Therefore, most of SOA approaches are ill-suited for describing microservice composition. Besides, the use of XML as the basis for the description makes the scheme too overfilled compared to, for example, JSON, YAML.

In addition to approved and known standards, there are prototypes such as eFlow [5], WISE [6], SOA4All [7] and others [4, 8]. Most, however, have the same disadvantages. Some approaches (METEOR-S [9], BCDF [10], SCENE [11]) imply the existence of some runtime or system which cause service. In our understanding, such environment is excessive.

The most suitable, in our opinion, was the work of SWORD [12]. The project provides the possibility to give a detailed account of a service interaction interface using a specification that is simpler and more expressive than XML-based. The written specification is automatically checked for compliance with existing specifications of running services. The software developed as a part of the research provides the composition of the service with others.

Described specifications and systems for checking interface compliance do not fit the conditions specified in the introduction, because, firstly, they focus on synchronous interaction, mainly on SOAP or REST protocols. Secondly, most of them are designed to describe high-level business processes, which does not enable to describe in detail the composition of microservices.

2.2. Discovering services by name

Service-discovery is a way of automatic detection of required services by their name. However, the method is worthy of consideration in the current context, since it eliminates the need to manually determine the dependencies of services and therefore simplifies their deployment and maintenance [13]. Its essence is to allocate central registry with information about existing microservices, their location, and status. A client service requests an external service address at the registry in case of necessity to contact it.

This approach simplifies the containerization of services and their scaling but does not solve the problem of compatibility control since the registry contains only information about the location and status of services by their name.

2.3. API service description

As for holding parameters types and types of return values of the microservice interface, the de-facto standards in case of synchronous interaction are OpenAPI for REST[1], WSDL for SOAP[2], Protobuf for GRPC[3]. Validation standards JSON-Schema and XSD for JSON and XML, respectively, represent types for asynchronous interaction. Apache Avro[4] suits well for validation and description of JSON structures due to support of subtypes.

None of the described standards or systems of composition and interfaces description does not fully meet the requirements specified in the introduction: support for asynchronous communication, interface compatibility verification, and guarantee that the implementation matches the interface.

3. PRINCIPLE OF SUITABLE INTERFACES DISCOVERY

The previous work [14] described the principle of finding suitable services, called “contract discovery”. The idea of “service discovery” formed the basis, although descriptions of the interaction interface and types of data exchanged are used instead of the dependency name.

3.1. Type system

To describe the types of data transferred in the “contract discovery” it is used a modified validation standard of JSON-structures “JSON-Schema” [15]. However, JSON-Schema is a set of data

[1] http://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md
[2] https://www.w3.org/TR/wsdl
[3] https://developers.google.com/protocol-buffers/docs/overview
[4] http://avro.apache.org/docs/current/

Listing 1 Type A

```
{
"type": "integer"
}
```

Listing 2 Type B

```
{
"type": "integer",
"minimum": 18
}
```

validation rules, and to associate an interface provider with its consumer, one should compare structures of the transferred data to the possibility of those structures extending by a provider.

Type system supports polymorphism through inheritance. Let us assume there is a function a waiting for input data of type A, but it is transmitted data B. If type A and provided type B are actually different, and the function will handle B correctly, it can be concluded that B can be a subtype or type equivalent to type A. That is, B is a subtype of A if every term B can be safely used in a context where A is expected (3.1) [16].

$$\frac{\Gamma \vdash x : A \quad A <: B}{\Gamma \vdash x : B} \tag{3.1}$$

Inheritance is reflective (3.2), and transitive (3.3).

$$A <: A \tag{3.2}$$

$$\frac{A <: B \quad B <: C}{A <: C} \tag{3.3}$$

Moreover, inheritance is possible into depth, i.e. the types of each corresponding field of the user-defined type may differ but must be in relation to inheritance (3.4), where l_i is the field of the user-defined type.

$$\frac{\forall i, A_i <: B_i}{\{l_i : A_i^{i \in 1..n}\} <: \{l_i : B_i^{i \in 1..n}\}} \tag{3.4}$$

At the same time fields rearrangement does not signify (3.5).

$$\frac{\{k_j : A_j^{j \in 1..n}\} \text{rearrangement} \{l_i : B_i^{i \in 1..n}\}}{\{k_j : A_j^{j \in 1..n}\} <: \{l_i : B_i^{i \in 1..n}\}} \tag{3.5}$$

We will adduce examples of type definition: integer type A is presented at listing 1.

Integer type with the constraint on minimal value is presented at listing 2 and is being a subtype of A: $A <: B$.

Listing 3 Type checking

```
bool isSubtype(a, b) {
subtype = true
if (a.type == scalar) {
subtype = (a.type != b.type) ||
(a.preds != b.preds)
} else if (a.type == object) {
for (f = range a.fields) {
subtype = subtype &&
isSubtype(f, b.fields[f])
}
subtype = subtype &&
(b.required in a.required)
}
return subtype
}
```

3.2. Type checking

The type validation algorithm shown in the listing 3 determines that any field of A type is present and has the same characteristic in the type B. Entry $a.preds$ in the given pseudocode returns a list of all predicates of type a. Pseudocode $b.fields[f]$ returns f field of b type. Pseudocode $a.fields$ returns all fields of type a. $b.required$ returns a list of required object fields. The algorithm does not try to compare predicates of fields, but only checks if they are equivalent. Objects are checked recursively. The list of required fields of B type should include a list of A type, if $A <: B$.

3.3. Contract

The basic concept in the "contract discovery" approach is a contract that fixes the data exchange protocol as a chain of types of transferred values between a server and a client within a single data transfer session. Each element of the chain is marked with a flag — “input” or “output” so that the parameters of the service call could be separated from the result of its work.

A typical description of REST or RPC interface in this paradigm consists of 2 chain elements: input parameters description and response description. At the same time, a list of contracts describing the entry points of REST or RPC of one service will serve as analogues to OpenAPI or WSDL format, respectively. A typical description of one event in the system is one contract with one link. For a service waiting for an event, this link will be marked with the "input" flag. For a service emitting an event, this link will be marked with the “output” flag.

In addition to the chain of transferred data types, the contract contains information about location of a service providing the contract, and information about how to check the relevance of the contract.

3.4. Principle of Contract-discovery and implementation

To implement type checking, the contract discovery service was developed. Service tasks:

1. Register a new supplier (service) of contract.
2. Register the contract usage by the new service.
3. Find the suitable providers for the services using the contract.
4. Track contracts usage.

The final version of the service is implemented in Golang language; it uses Redis as a database and provides the HTTP interface. The source code of the service prototype in PHP is available in the Github repository[5].

The described approach was used in the implementation of the data conversion system built by asynchronous-communicating microservices. The experiment has shown that using contract discovery enables to make sure that the service expects the correct message fields at the time the service starts, and not during operation.

However, the contract discovery service is a central link containing complex logic, which reduces the overall reliability of the system. Creating and updating contracts for protocols based on RPC requires writing a lot of duplicate code, which is also a disadvantage. Finally, there is no guarantee that the service provides exactly the interface that the former declares in the contract.

4. MESSAGE BROKER TYPING SUPPORT

Due to the shortcomings of the contract discovery service described in the previous chapter, we have developed a new approach devoid of some shortcomings from the previous one.

The updated approach is based on the same type system and subtype checking algorithm as before. However, now the task of type checking falls either on the message broker or on each microservice separately. The broker is responsible for storing the types.

To realize such types storage, and in order to simplify interface description, we refused to support synchronous interaction in favour of a message queue. The functionality that the message broker gets responsible for, is partially moved from the list of the functionality of the contract-discovery service:

1. Type checking;
2. Selection of a suitable queue according to the data type.

4.1. Type checking

When a new service connects, a message broker should perform a number of checks based on the information transferred to it:

1. The service connects to the specifically named message channel for reading or writing:
   (a) If the queue with the same name already exists and the type of data to be transferred is specified, the message broker should check that the type declared by the service when connecting is a subtype or identical to the one specified for the queue.
   (b) If the queue with this name does not exist, the message broker should create it with the type passed by the service.
2. If the service does not specify to which queue, it is connecting for reading or writing, and at the same time, it determines the type of data transferred – the broker should choose the appropriate queue based on information about existing ones.
3. In all other cases, a connection error should occur.

[5] http://github.com/tariel-x/cc

If the connection is successful, further checks are not performed, the service can transfer data. Checks are performed both to try to read from the queue and to write to it.

The given example of the algorithm made it possible to protect the system from typing errors; it is not overcomplicated by excessive versatility; the system does not add unnecessary binding elements.

4.2. Service update

The microservice interface update in such a work scheme causes difficulties. If the new service version has an interface incompatible with the previous version, they cannot be started at the same time to work with the same queue. If the previous microservice version has been disabled at first, there are still other microservices working with the outdated data type. In this case, they should be turned off.

In the situation, when no more microservices are working with the particular queue, the broker should delete that queue.

4.3. Implementation and testing

Among all considered popular message brokers (RabbitMQ, Apache Kafka, NATS) none of them enables to implement any type checking entirely by one's means. However, RabbitMQ makes it possible to save additional information for queues. There was an implementation developed to be integrated into each microservice in order to check algorithm efficiency. The library performs all checks instead of the broker but uses RabbitMQ as a centralized type store.

The library implementing the algorithm is developed in [6], uses code generation and reflection to ensure that declared input and output types of the microservice correspond to reality. That is possible thanks to the construction of JSON-Schema for data types descriptions based on the analysis of the microservice source code.

The approach is applied to the ETL system. The first microservice from the chain provides the REST interface, validates input data, and passes it to the next one. The second microservice loads external resources like images or video to internal storage and passes the data with local links further. The third one saves validated and updated data to the database and into the search index.

The process of development, operation, and support of such a system has shown that the described approach rules out types mismatch for the microservices interfaces. Also, the approach does not have the shortcomings listed in the previous work. The ease of operation and scaling level the limitations of the described approach in comparison with those described in the previous paper.

In the described approach, the data type is extracted from the microservice source code to ensure that the microservice declares exactly the types of data that it expects for input and output. However, this is only true for strongly typed languages (e.g. Golang). The use of languages, in which it is impossible or time-consuming to determine the data type, leads to the loss of the described advantage.

Another drawback of the current implementation is the dependency on the RabbitMQ API. In the implementation, current queues of messages and types attached to these queues are received through a separate API, which usually requires a separate connection.

[6] https://github.com/tariel-x/tsc

Comparing with the approaches mentioned in Chapter 2 is presented in table 1. Even though presented in the table alternatives implement very different tasks, everyone can be used for guaranteeing compatibility of interacting services.

Table 1. Comparison of the proposed solution and alternatives.

Solution / characteristic	WS-BPEL, WS-CDL, BPML, and analogs	SWORD	Protobuf (GRPC)	WSDL, OpenAPI	Suggested option
Check for services compatibility	–	+	–	–	+
Check for interfaces correct implementation	–	–	+	+	+
Support of async communication	–	–	+	–	+
Support of sync communication	+	+	+	+	–

Criteria for the comparison of the approaches base on the requirements listed in the Introduction. The column “WS-BPEL, WS-CDL, BPML, and analogs” indicates both the standards and systems listed in the title and their analogs defined in the same Chapter. The system described in Chapter “Message broker typing support” meets all requirements presented in the Introduction section, although it does not supports synchronous communication model in contrast to the principle of contract-discovery described in the previous work [14].

According to the table, the introduced system is the only one which able to guarantee services compatibility with asynchronous microservice architecture.

5. CONCLUSION

In this paper, we have presented the approach of type-safe microservices composition for event-driven architectures. The approach uses RabbitMQ message broker and custom type system to guarantee that microservices are able to communicate with each other. Usage of code generation and reflection enables to ensure compliance of the interface description with its implementations for programming languages with static type checking ability. According to the comparison with the alternatives, the introduced approach is the only one which able to guarantee services compatibility with event-driven microservice architecture.

Thus we implemented a preliminary check of data types transferred by microservices for compatibility before their launch or update. The check helps to avoid errors related to data types during the new components development and maintenance.

However, the presented realization uses API RabbitMQ’s that is non-standard for microservices due to the requirement of separate authorization for API and message transfer. Therefore, production implementation requires replacing the existing implementation with a full-fledged message broker with typing support.

There is another avenue of future work: carry out the composition of services based on a scheme written with DSL. That will make it possible to check the types of services and their composition before they are deployed.

References

1. R. Rodger, *The tao of microservices*, New York: Manning Publications Company, 2018, pp. 17–19.
2. N. Kratzke, “A Brief History of Cloud Application Architectures,” *Applied Sciences*, vol. 8, no. 8, 2018; doi: 10.3390/app8081368
3. M. Richards, *Microservices vs. service-oriented architecture*, Sebastopol: O’Reilly Media, 2015.
4. Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, “Web services composition: A decade’s overview,” *Information Sciences*, vol. 280, pp. 218–238, 2014; doi: 10.1016/j.ins.2014.04.054
5. F. Casati and M. C. Shan, “Dynamic and adaptive composition of e-services,” *Information Systems*, no. 26, pp. 143–163, 2001; doi: 10.1016/S0306-4379(01)00014-X
6. A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler, “The WISE approach to electronic commerce,” *International Journal of Computer Systems Science and Engineering*, vol. 15, no. 5, pp. 343–355, 2000.
7. F. Lecue, Y. Gorronogoitia, R. Gonzalez, M. Radzimski, and M. Villa, “SOA4All: an innovative integrated approach to services composition,” in *Proc. 8th IEEE Int. Conf. on Web Services* (ICWS’10), Miami, FL, USA, 2010, pp. 58—67; doi: 10.1109/ICWS.2010.68
8. A. L. Lemos, F. Daniel, and B. Benatallah, “Web Service Composition: A Survey of Techniques and Tools,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–41, 2015; doi: 10.1145/2831270
9. K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller, “METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services,” *Information Technology and Management*, vol. 6, no. 1, pp. 17–39, 2005; doi: 10.1007/s10799-004-7773-4
10. B. Orriens and J. Yang, “A Rule Driven Approach for Developing Adaptive Service Oriented Business Collaboration,” *IEEE Int. Conf. on Services Computing* (SCC’06), Chicago, IL, USA, 2006; doi: 10.1109/SCC.2006.14.
11. M. Colombo, E. Di Nitto, and M. Mauri, “SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules,” in *Service-Oriented Computing*, A. Dan and W. Lamersdorf eds., 2006; doi: 10.1007/11948148_16
12. S. R. Ponnekanti and A. Fox, “Sword: A developer toolkit for web service composition,” *Proc. 11th Int. WWW Conf*, Honolulu, HI, USA, 2002.
13. L. Sun, H. Dong, F. Hussain, O. Hussain, E. Chang, “Cloud service selection: State-of-the-art and future research directions,” *Journal of Network and Computer Applications*, vol. 45, pp. 134–150, 2014; doi: 10.1016/j.jnca.2014.07.019
14. N. Gerasimov, “Static typing and dependency management for SOA,” *Position Papers of the 2018 Federated Conference on Computer Science and Information Systems*, vol. 16, pp. 105–107, 2018.
15. A. Wright, H. Andrews and G. Luff, “JSON Schema Validation: A Vocabulary for Structural Validation of JSON,” [Online]. Available: http://jsonschema.org/latest/json-schema-validation.html
16. B. Pierce, *Types and Programming Languages* London: MIT Press, 2002, pp. 251–254.

Received 12.02.2019, the final version — 14.03.2019.

Компьютерные инструменты в образовании, 2019
№ 1: 43–53
УДК: 004.4'6
http://ipo.spb.ru/journal
doi:10.32603/2071-2340-2019-1-43-53

Строгая типизация в событийной микросервисной парадигме

Герасимов Н. С.[1], аспирант, n.gerasimov@2015.spbu.ru

[1]Санкт-Петербургский государственный университет,
Университетский пр., д. 28, Петергоф, 198504, Санкт-Петербург, Россия

Аннотация

Микросервисная парадигма — популярный подход при разработке различных систем, который характеризуется разбиением монолитной системы на большое количество небольших по размеру приложений. Работающие вместе и реализующие необходимую логику, компоненты выполнены с применением наиболее подходящих для них технологий. Подход также позволяет осуществлять простое масштабирование наиболее нагруженных частей, однако, среда передачи данных между компонентами не позволяет в достаточной степени гарантировать их совместимость при разработке и поддержке. Существует очень ограниченный набор средств для поддержания согласованной работы всей системы при изменении входных или выходных параметров какого-либо микросервиса из её состава. Проблема особенно актуальна в случае асинхронного взаимодействия частей системы, так как существующие решения и стандарты покрывают в основном синхронное взаимодействие, основывающееся на протоколах семейства RPC или REST, или используются для высокоуровневого описания автоматизируемых бизнес-процессов. Одним из способов обеспечения совместимости компонентов системы является предварительная проверка типов аргументов перед тем, как сервисы начнут работу. В данной работе представлена разработка и тестирование метода для предварительной проверки типов в событийной модели взаимодействия микросервисов. Разработанный подход использует RabbitMQ в качестве хранилища типов очередей сообщений и специализированную библиотеку для проверки микросервисами возможности использовать выбранную очередь. Таким образом, совокупность брокера и библиотеки позволяют избежать расхождения типов аргументов и результатов микросервисов в событийной микросервисной архитектуре.

Ключевые слова: *микросервисная архитектура, статическая типизация, событийная модель, SOA, композиция сервисов.*

Цитирование: Герасимов Н. С. Строгая типизация в событийной микросервисной парадигме // Компьютерные инструменты в образовании. 2019. № 1. С. 43–53. doi: 10.32603/2071-2340-2019-1-43-53

Поступила в редакцию 12.02.2019, окончательный вариант — 14.03.2019.