УДК 004.043

К. Абрикосов [1], В. А. Попова [1]

[1] Институт математики и информационных технологий, Иркутский государственный университет, г. Иркутск, Россия

СТАТИЧЕСКАЯ ТИПИЗАЦИЯ В ВЕБ-ПРИЛОЖЕНИЯХ НА NODE.JS: ПРОЦЕСС ВНЕДРЕНИЯ И ВЫЯВЛЕНИЕ ОШИБОК НЕСООТВЕТСТВИЯ ТИПОВ

Аннотация. Создание качественного программного обеспечения, которое обладает высокой степенью отказоустойчивости при различных факторах, является непростой задачей для разработчиков. К одной из проблем, которая существенно влияет на качество программ, относится несоответствие типов конструкций в динамически типизированных языках программирования. В связи с этим создаются и внедряются инструменты, которые позволяют устранять немалое количество ошибок типизации на этапе разработки. Проблема несоответствия типов также актуальна для веб-приложений, которые нашли широкое применение для решения повседневных задач пользователей в различных сферах деятельности. На текущий момент особо популярным языком программирования для создания как серверной, так и клиентской части веб-приложений является JavaScript. Несмотря на удобство использования и высокую скорость разработки благодаря этому языку программирования, присутствующие проблемы динамической типизации, которые не всегда может отследить разработчик, существенно затрудняют использование созданных веб-приложений. Это связано с высокой вероятностью возникновения ошибок во время работы программы, что способно нарушить целостность данных и остановить рабочие процессы. В данной работе описывается процесс внедрения фреймворка Nest.js, поддерживающего статически типизированных язык программирования TypeScript, в серверную часть веб-приложения, которая реализована на платформе Node.js. Перечисляются виды ошибок, которые были найдены в результате внедрения Nest.js.

Ключевые слова: статическая проверка типов, динамическая типизация, Nest.js, ошибки несоответствия типов, качество программного кода, архитектура информационных систем.

C. Abrikosov [1], V. A. Popova [1]

[1] Institute of Mathematics and Information Technology, Irkutsk State University, Irkutsk, Russian Federation

STATIC TYPE-CHECKING IN WEB APPLICATIONS ON NODE.JS: INTEGRATION PROCESS AND IDENTIFICATION OF TYPE MISMATCH ERRORS

Abstract. Quality software development that has a high degree of fault tolerance under various factors is not an easy task for developers. One of the problems that significantly affects the program's quality is the type mismatch of constructs in dynamically typed programming languages. In this regard, there are tools being created and implemented that allow you to eliminate a considerable number of typing errors at the development stage. The problem of type mismatch is also relevant for web applications that are widely used to solve everyday tasks of users in various fields of activity. At the moment, JavaScript is a particularly popular programming language for creating both server-side and client-side web applications. Despite the ease of use and high development speed due to this programming language, the dynamic typing problems that are present, which the developer cannot always track, make it very difficult to use the created web applications. This is due to the high probability of errors occurring during the operation of the program, which can violate the integrity of data and stop workflows. This paper describes the process of implementing the Nest.js framework, which supports the statically typed TypeScript programming language, into the backend of a web application, which is implemented on the Node.js platform. Lists the types of errors that were found as a result of implementing Nest.js.

Keywords: static type checking, dynamic typing, Nest.js, type mismatch errors, programming code quality, information systems architecture.

Введение. Стремительное развитие информационных технологий позволяет с каждым годом всё больше автоматизировать различные производственные процессы. В связи с потребностью внедрения в программы новых функциональных возможностей повышается риск возникновения ошибок, что обусловливается усложнением программного кода, которые со временем становится труднее поддерживать разработчикам. Нередко дефекты не выявляются на этапе разработки, а присутствуют в выпущенных версиях программных продуктов. В та-

ком случае ошибки выявляются уже в процессе использования программы пользователями, что может с большей долей вероятности привести к тому, что рабочие процессы будут приостановлены до момента исправления ошибки, то есть на неопределённый срок, поскольку может потребоваться большое количество модификаций [1].

Поэтому для повышения качества программ [2] следует использовать существующие или создавать новые механизмы, которые позволяют находить ошибки ещё на этапе разработки, а не уже при применении программ пользователями.

К существенным ошибкам, которые значительно влияют на работоспособность программ, относятся ошибки несоответствия типов.

Проблемы, связанные с типами, могут возникать в любом типизированном языке программирования, но процесс выявления ошибок отличается в языках статической и динамической типизациями. Так, в языках программирования со статической типизацией, например, Java и C++, ошибки несоответствия типов выявляются на этапе компиляции [3, с. 7]. Благодаря этому ошибки исправляются в процессе разработки. В свою очередь, для языков программирования с динамической типизацией, к которым относятся, к примеру, JavaScript, Python и 1C, ошибки могут быть выявлены только в процессе работы программы.

Для того чтобы отслеживать ошибки динамической типизации, следует внедрять механизмы, поддерживающие статический анализ программного кода, при котором ошибки несоответствия типов будут выявляться до момента запуска программы [4, 5].

В настоящий момент особый интерес с точки зрения контроля несоответствия типов представляет язык программирования JavaScript, который активно применяется для создания веб-приложений различной степени сложности. Чтобы находить ошибки в программном коде на JavaScript, существуют такие инструменты, как JsLint [6] и Closure Compiler [7]. Для платформы Node.js [8], которая позволяет разрабатывать серверные части веб-приложений, находят применение JsHint [9] и Nest.js [10].

JSHint является инструментом статического анализа программного кода, который показывает предупреждения в тех местах, где присутствует неявное преобразование, неверное обращение к свойству, а также другие ошибки синтаксиса.

Nest.js является полноценным фреймворком, реализованным на языке программирования TypeScript. Помимо нахождения ошибок несоответствия типов, Nest.js обеспечивает приложение удобной и хорошо поддерживающейся архитектурой. В связи с преимуществами фреймворка его уместно внедрить в уже существующее приложение, чтобы избежать в дальнейшем ошибок типизации и при этом поддержать высокое качество программного кода благодаря имеющимся архитектурным решениями в Nest.js [11].

Необходимость внедрения Nest.js в проект «Расписание ИГУ». В настоящий момент актуальной является разработка проекта «Расписание ИГУ» для представления расписания в электронном формате [12] для Иркутского государственного университета (ИГУ). Серверная часть проекта создаётся при помощи фреймворка Node.js. Поскольку система предназначается для большого количества пользователей, а также предполагает сложную обработку данных, было решено внедрить в проект фреймворк Nest.js для дополнительных проверок параметров API, выявления ошибок несоответствия типов, а также устранения архитектурных недочётов, которые были выявлены в процессе разработки проекта.

Существенными недостатками архитектуры проекта «Расписание ИГУ» являлась высокая степень вложенности программного кода, а также отсутствие проверок типов, что могло бы привести к неоднозначному поведению в процессе эксплуатации системы. Важно отметить, что ошибки несоответствия типов можно с высокой долей вероятности избежать только в случае, если для всех фрагментов программного кода написаны тестовые наборы данных, которые запускаются после каждой итерации разработки. Но поскольку Nest.js решает не только проблему динамической типизации, но и поддерживает практику соблюдения принципов качественного построения архитектуры, внедрение такого механизма стало лучшим решением, которое существенным образом повлияло на удобство поддержки и развития проекта.

Ключевые особенности Nest.js. В информационной системе «Расписание ИГУ» имелись следующие недостатки:

– ошибки несоответствия типов: опечатки, ошибки приведения типов;

– архитектура информационной системы на динамически типизированном языке программирования, препятствующая развитию проекта.

Поскольку было решено устранить ошибки несоответствия типов, было необходимо выбрать релевантные поставленной задаче инструменты, главным из которых стал фреймворк Nest.js. Вместе с решением основной проблемы несоответствия типов, данный инструмент предлагает ряд архитектурных решений, позволяющих проектировать более гибкие информационные системы.

Nest.js поддерживает язык программирования TypeScript, вместе с тем, реализует модульную архитектуру, а благодаря мощному слою абстракции, позволяет внедрять уже готовые решения в виде отдельных модулей.

В современных версиях языков программирования JavaScript и TypeScript существует возможность использования декораторов — функций, позволяющих добавить некоторое поведение другим объектам (функции и методы классов тоже являются объектами). Этот инструмент активно используется в фреймворке Nest.js и интегрированных модулях и реализует декларативный стиль программирования. Список основных библиотек, использованных в проекте:

– Swagger для документирования API [13],
– Sequelize [14],
– Passport [15].

Перевод моделей. В начале выполнения перевода информационной системы на новый инструмент было необходимо перевести сущности проекта в модели контекста фреймворка Nest.js. Каждая модель описывается в программном коде в виде модуля, который может состоять из следующего набора файлов:

– описание модуля,
– описание модели,
– файл-сервис,
– файл-контроллер,
– объекты транспортировки данных,
– файлы, необходимые для тестирования программного кода.

Для наглядности, пример структуры модуля, описывающего сущность записи расписания, представлен на рисунке 1.

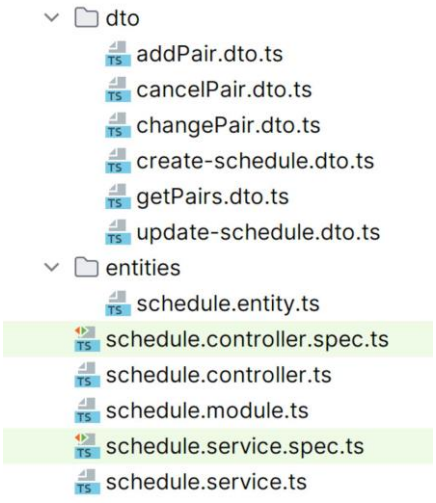

Рис. 1. Модуль сущности *schedule*

Основные функции-декораторы, использованные при переводе моделей:

1. *Table* — декоратор для обозначения таблицы сущности в рамках системы объектно-реляционного преобразования.

2. *ApiProperty* — декоратор для описания атрибута, свойства произвольного объекта, по которому будет составляться документация.

3. *Column* — декоратор для обозначения столбца таблицы сущности.

4. *IsString, IsPositive* — декораторы-валидаторы.

5. *HasOne, HasMany, BelongsTo, BelongsToMany* — декораторы для указания отношений между сущностями.

6. *ForeignKey* — декоратор, обозначающий столбец, значением которого является внешний ключ.

Настройка отношений между моделями. Отношения между моделями определяются с помощью декораторов *BelongsTo, BelongsToMany, HasOne, HasMany*. Сложности возникают из-за нестандартных обозначений (в Java Spring, например, используются более классические обозначения *OneToMany, ManyToOne*, и т. д.). А декоратор *BelongsTo* используется для обозначения как отношений типа один-ко-многим, так и один-к-одному. Вместе с тем, необходимо указывать внешние ключи, либо в дополнительных параметрах декораторов для обозначения отношений, либо отдельными свойствами, декорированными функциями *ForeignKey*.

Перевод логики серверной части в сервисы фреймворка Nest.js. Сервисы — классы, декорированные функцией *Injectable*. Они реализуют логику каждого модуля, могут быть внедрены или внедрять другие модули. Основная задача данного этапа — корректный перенос ранее написанных методов. Параллельно производилась типизация и документирование перенесенных методов. Сервисы содержат набор методов, необходимых для работы информационной системы, содержание, аргументы, возвращаемые значения которых могут быть произвольными в зависимости от задачи, которую они решают.

Контроллеры — классы, отмеченные декоратором *Controller*. Задача контроллера — декларировать маршруты и возвращать нужные ответы. Обычно, контроллеры внедряют сервисы и в декларации необходимых маршрутов возвращают результат вызова соответствующего метода сервиса. Под декларацией следует понимать полное определение маршрута: собственно, маршрут и метод запроса. Определяется маршрут с помощью встроенных функций-декораторов, название которых соответствует методу запроса (*Get, Post, Put, Patch, Delete*), а параметром является, собственно, маршрут.

Определение и документирование объектов транспортировки данных. Nest.js интегрировал инструмент для документирования методов — Swagger, в рамках фреймворка он доступен как модуль библиотеки и набор сопутствующих методов. С его помощью можно документировать конечные точки приложения, структуру ответа сервера и объектов полезной нагрузки запросов.

В рамках документирования объектов транспортировки данных можно описать структуру объектов, участвующих в запросах к серверу и ответах. При описании полей объекта можно указать тип, описание поля, примерное значение, а также необходимость присутствия в объекте.

Определение правил проверки передаваемых данных. Часто возникают задачи, которые достаточно сложно решить с помощью инструментов, имеющихся в распоряжении разработчиков. Поэтому, может возникнуть потребность в расширении возможностей этих инструментов.

Рассмотрим отдельную задачу, решение которой потребовалось в данном проекте. Было необходимо проверить принадлежность строки к определенному формату. Таким образом, при реализации приложения, возникла необходимость создания собственного декоратора-валидатора. Nest.js имеет зависимость от библиотеки class-validator [16], которая предоставляет возможности для проверки передаваемых пользователем данных. TypeScript, в свою очередь, предоставляет возможности для решения поставленной задачи универсальным спо-

собом, во избежание дублирования кода. В общем виде декораторы-валидаторы должны вернуть в методе *validate* булево значение, обозначающее, прошел ли переданный аргумент проверку.

Для создания собственных декораторов-валидаторов был написан специальный метод *createValidationDecorator*, использующий функцию *registerDecorator* из библиотеки class-validator. В нем используется шаблонный параметр <*T*>, позволяющий обобщить решение для возможности применения описанного метода в более конкретных задачах. В обобщенной реализации все, что делает созданный метод – декларирует тип и аргументы необходимые для проверки, и вызывает функцию *registerDecorator*. Аргументами метода *createValidationDecorator* стал следующий набор данных: *name* – название декоратора, *validationFunction* – функция проверки данных, которая зависит от заданного шаблонного параметра <*T*> и *validationOptions* – объект более тонкой настройки поведения проверки данных.

При реализации проекта возникла необходимость проверки соответствия строки определенному формату даты, поэтому, был создан собственный декоратор-валидатор *IsDateString*. Для этого используется библиотека moment. Используя ранее написанный метод createValidationDecorator, было достаточно просто реализовать необходимое поведение. Достаточно было определить функцию для валидации данных, которая выглядит следующим образом: *(arg: string) => moment(arg, 'YYYY-MM-DD').isValid()*, и передать ее в параметры метода.

Документирование методов прикладного интерфейса. Для документирования методов прикладного интерфейса использовался упомянутый ранее инструмент Swagger. За исключением разовой настройки в главном исполняемом сценарии, в основном использовались методы-декораторы, импортируемые из модуля библиотеки, а в результате документирования, становится доступно веб-приложение с описанием всех маршрутов и объектов транспортировки данных.

Абстрактно результат документирования методов прикладного интерфейса можно описать следующим образом:

1. Статус-код ответа сервера на запрос.
2. Структура ответа сервера на запрос.
3. Структура тела запроса для метода.
4. Назначение метода.

Написание инструментов для осуществления аутентификации пользователей. Придерживаясь модульной архитектуры и декларативного стиля написания кода, средства защиты, аутентификации в фреймворке Nest.js реализуются в виде классов-защитников (от англ. Guards), определение которым дается в следующем разделе. Nest.js интегрировал популярную библиотеку Passport в виде модуля и наборов классов и функций. Следуя документации, сначала был создан модуль *auth*, структура файлов которого представлена на рисунке 2.

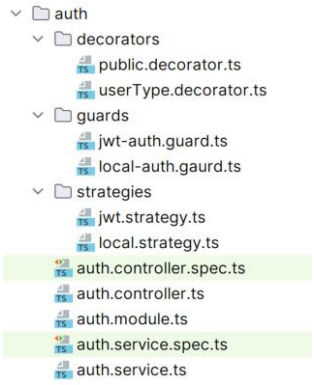

Рис. 2. Модуль auth

Класс локальной стратегии расширяет класс стратегии из модуля Passport, основной метод, отвечающий за авторизацию — *validate*.

Класс следующей стратегии отвечает уже за аутентификацию. Метод *validate* данного класса занимается расшифровкой JWT-токена авторизованного пользователя, и добавляет новое поле user с расшифрованными данными объекту request из библиотеки expressJS.

Настройка ограничения доступа к маршрутам. Не все маршруты должны быть общедоступными, в основном, только маршруты для получения данных должны быть открытыми. Решить поставленную задачу можно с помощью инструментов доступных в фреймворке.

Nest.js предоставляет инструмент для контроля доступа к маршрутам в виде классовых компонентов — классов-защитников, отмеченных декоратором *Injectable* и реализующих интерфейс *CanActivate* и/или наследующих встроенный класс *AuthGuard(type)*. *AuthGuard* — встроенная функция, принимающая строку с типом стратегии, результатом которой является тип *Type<IAuthGuard>*, именно поэтому ее можно вызвать после ключевого слова *extends*.

Для решения поставленной задачи было необходимо создать инструмент для указания открытости некоторого маршрута. В фреймворке есть функция *registerDecorator*, над которой можно построить собственные функции-декораторы, и которая позволяет назначить произвольные метаданные некоторому объекту. Стоит отметить, что функции и методы классов в языках программирования JavaScript и TypeScript также являются объектами.

Метаданные хранятся в формате ключ-значение, в качестве ключа удобнее использовать константную строку, а значение может быть произвольного типа. Определим константу *IS_PUBLIC_KEY*, обозначающую ключ, и установим значение метаданных по этому ключу равным *true*. Далее, в методе класса-защитника, мы сможем получить значение произвольных метаданных по ключу, если такие были определены. Стоит отметить удобство имеющегося инструмента: можно назначить метаданные целому классу-контроллеру и отдельным методам. Таким образом, вызвав встроенный в фреймворк метод для получения метаданных по ключу *IS_PUBLIC_KEY*, мы сможем узнать, является ли маршрут общедоступным.

В рамках поставленной задачи нас интересует метод *canActivate*, который возвращает булево значение, и дающий понять, разрешен или запрещен доступ к маршруту. Если этот метод класса-защитника вернул значение *false*, то выбросится значение *ForbiddenException*, в конечном итоге, с использованием внутренних средств перехвата ошибок, от сервера вернется 403 статус-код с сообщением *Forbidden*. Данное поведение может быть изменено, например, в классах-защитниках, реализующих стратегию защиты по JWT-токену в случае возврата значения *false* методом *canActivate*, будет возвращен статус-код 401 с сообщением *Unauthorized*.

Результаты внедрения инструментов. В процессе разработки информационной системы и перевода на инструмент с элементами статической типизации были найдены ошибки в следующем количестве:

– опечатки – 1;

– ошибки отсутствия свойств в объектах – 2.

Стоит отметить, что даже одна ошибка является существенной проблемой для корректного и стабильного функционирования информационной системы, а малое количество найденных ошибок, связанных с проблемой типизации, в исходном проекте не может указывать на степень полезности или бесполезности инструментов статической типизации. В дополнение стоит отметить, что предыдущая версия приложения была выполнена с высокими показателями ответственности и внимательности. Но при этом немаловажно, что без Nest.js поддержка приложения была сложной и требовала больше затрат на разработку ввиду необходимости контроля корректности типов.

Благодаря параллельному документированию методов прикладного интерфейса и объектов транспортировки данных дальнейшая разработка как клиентской части, так и серверной становится более удобной и менее трудоемкой. Пример задокументированного метода прикладного интерфейса представлен на рисунке 3.

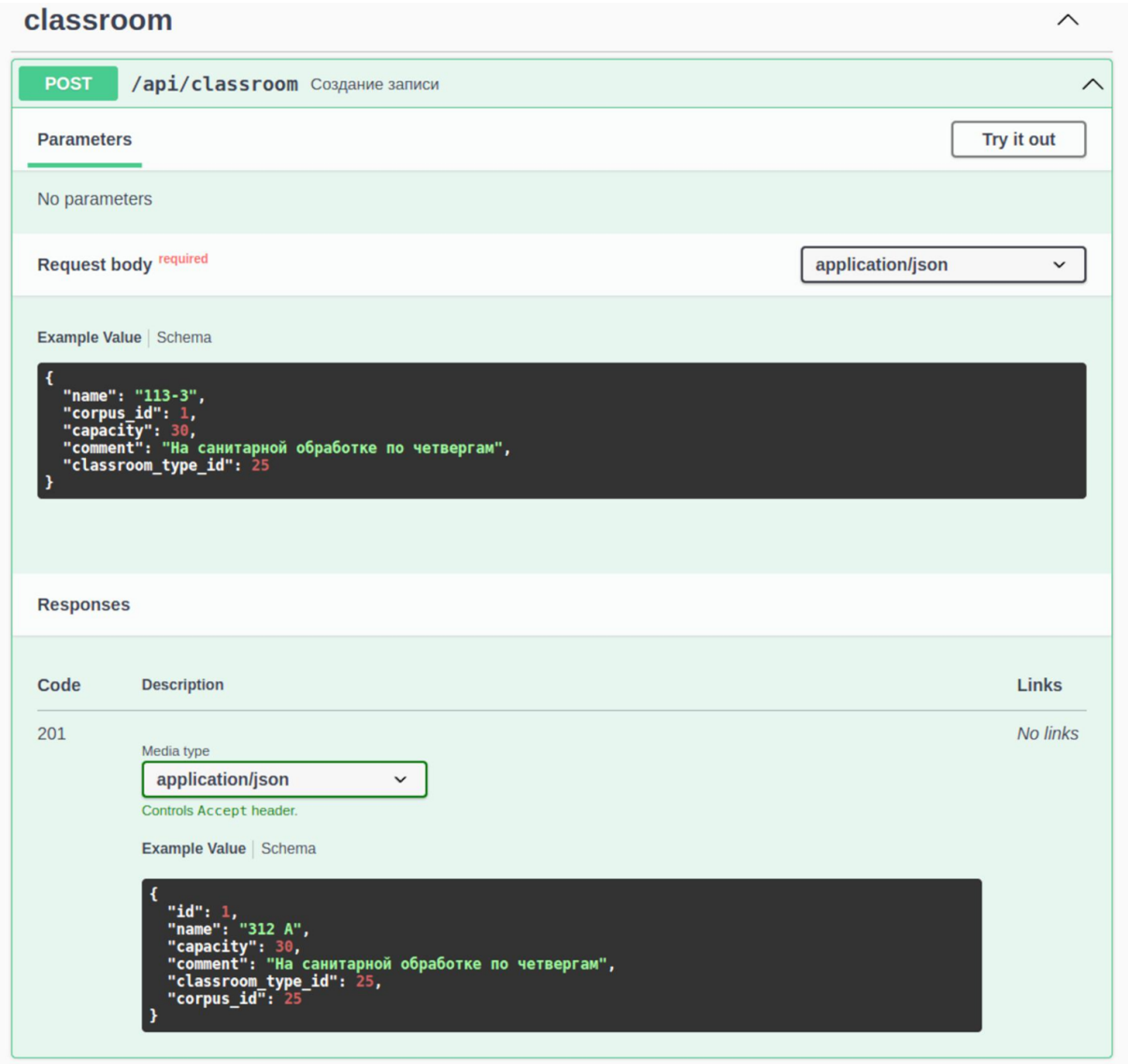

Рис. 3. Веб-документация прикладного интерфейса приложения

Всего было создано 32 модуля для всех сущностей, а суммарно доступен 201 маршрут. Благодаря внедрению Nest.js и сопутствующему улучшению архитектуры приложения, проектирование, поддержка и рефакторинг большого количества модулей и маршрутов не вызывают серьезных трудностей при дальнейшем расширении функциональности системы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Balyuk A. S., Popova V. A. Static type-checking for programs developed on the platform 1C:Enterprise // CEUR Workshop Proceedings : 4, Irkutsk, 14 сентября 2021 года. Irkutsk, 2021. P. 101–111.

2. Карпунин А. А., Ганев Ю. М., Чернов М. М. Методы обеспечения качества при проектировании сложных программных систем // Надежность и качество сложных систем. 2015. № 2 (10). С. 78–84.

3. Макконнелл С. Совершенный код. М. : Издательство «Русская редакция», 2010. 896 с.

4. Попова В. А. Применение методики статического анализа для выявления ошибок в программах на динамических языках программирования // Информационные технологии. Проблемы и решения. 2022. № 2 (19). С. 26–31.

5. Попова В. А. Проектирование механизма статического анализа для выявления ошибок несоответствия типов в программах на динамических языках программирования // Динамические системы и компьютерные науки: теория и приложения (DYSC 2022) : Материа-

лы 4-й Международной конференции, Иркутск, 19–22 сентября 2022 года / Иркутск: Иркутский государственный университет, 2022. С. 142–145.

6. JSLint, The JavaScript Code Quality and Coverage Tool [Электронный ресурс]. – URL: https://www.jslint.com/ (дата обращения: 20.03.2023).

7. Closure Compiler. Google Developers [Электронный ресурс]. – URL: https:// developers .google.com/closure/compiler (дата обращения: 20.03.2023).

8. Node.js® is an open-source, cross-platform JavaScript runtime environment. [Электронный ресурс]. – URL: https://nodejs.org (дата обращения: 20.03.2023).

9. JSHint, a JavaScript Code Quality Tool [Электронный ресурс]. – URL: https://jshint.com (дата обращения: 20.03.2023).

10. NestJS – A progressive Node.js framework [Электронный ресурс]. – URL: https://docs.nestjs.com (дата обращения: 21.03.2023).

11. Demashov, D. Node.js Frameworks Performance Comparison / D. Demashov, I. Gosudarev // The Majorov International Conference on Software Engineering and Computer Systems, 12–13 декабря 2019 года. Vol. Выпуск 11. – Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО», 2020. P. 24–28.

12. Попова В. А., Гармаева Д. А., Казимиров А. С. Создание информационной системы для представления расписания занятий высшего учебного учреждения // Молодежный вестник ИрГТУ. 2021. Т. 11, № 1. С. 21–28.

13. Интеграция Swagger в фреймворк Nest.js [Электронный ресурс]. – URL: https://github.com/nestjs/swagger#readme (дата обращения: 22.03.2023).

14. Библиотека sequelize-typescript [Электронный ресурс]. – URL: https://github.com/sequelize/sequelize-typescript#readme (дата обращения: 22.03.2023).

15. Документация Passport.js – URL: https://github.com/typestack/class-validator#readme (дата обращения: 22.03.2023).

16. Библиотека class-validator [Электронный ресурс]. – URL: https://github.com/typestack/class-validator#readme (дата обращения: 22.03.2023).

REFERENCES

1. Balyuk A. S., Popova V. A. *Static type-checking for programs developed on the platform 1C:Enterprise*. CEUR Workshop Proceedings : 4, Irkutsk, 14 September 2021. Irkutsk, 2021. pp. 101–111.

2. Karpunin A. A., Ganev YU. M., CHernov M. M. *Metody obespecheniya kachestva pri proektirovanii slozhnyh programmnyh sistem* [Methods of quality assurance in the design of complex software systems] *Nadezhnost' i kachestvo slozhnyh sistem* [Reliability and quality of complex systems]. 2015, no. 2 (10), pp. 78–84.

3. Makkonnell S. *Sovershennyj kod* [Complete Code]. Moscow, «Russkaya redakciya», 2010, 896 p.

4. Popova V. A. *Primenenie metodiki staticheskogo analiza dlya vyyavleniya oshibok v programmah na dinamicheskih yazykah programmirovaniya* [Application of static analysis techniques to identify errors in programs in dynamic programming languages]. *Informacionnye tekhnologii. Problemy i resheniya* [Information technology problems and solutions] 2022. no. 2 (19). pp. 26–31.

5. Popova V. A. *Proektirovanie mekhanizma staticheskogo analiza dlya vyyavleniya oshibok nesootvetstviya tipov v programmah na dinamicheskih yazykah programmirovaniya* [Designing a static analysis mechanism to detect type mismatch errors in programs in dynamic programming languages]. *Dinamicheskie sistemy i komp'yuternye nauki: teoriya i prilozheniya (DYSC 2022) : Materialy 4-j Mezhdunarodnoj konferencii, Irkutsk, 19–22 sentyabrya 2022 goda* [Dynamic Systems and Computer Science: Theory and Applications (DYSC 2022) : Proceedings of the 4th International Conference, Irkutsk, September 19-22, 2022] Irkutsk: Irkutsk State University, 2022. pp. 142–145.

6. JSLint, The JavaScript Code Quality and Coverage Tool. URL: https://www.jslint.com/ (access date: 20.03.2023).

7. Closure Compiler. Google Developers. URL: https:// developers .google.com/closure/compiler (access date: 20.03.2023).

8. Node.js® is an open-source, cross-platform JavaScript runtime environment. URL: https://nodejs.org (access date: 20.03.2023).

9. JSHint, a JavaScript Code Quality Tool. URL: https://jshint.com (access date: 20.03.2023).

10. NestJS – A progressive Node.js framework. URL: https://docs.nestjs.com (access date: 21.03.2023).

11. Demashov D., Gosudarev I. *Node.js Frameworks Performance Comparison*. The Majorov International Conference on Software Engineering and Computer Systems, December 12-13, 2019. Federal State Autonomous Educational Institution of Higher Education "ITMO National Research University", 2020. vol. 11. pp. 24–28.

12. Popova V. A., Garmaeva D. A., Kazimirov A. S. Sozdanie informacionnoj sistemy dlya predstavleniya raspisaniya zanyatij vysshego uchebnogo uchrezhdeniya [Creating an information system for presenting the schedule of classes of a higher educational institution] *Molodezhnyj vestnik IrGTU* [Youth Bulletin of IrSTU] 2021. vol. 11, no. 1. pp. 21–28.

13. Integraciya Swagger v frejmvork Nest.js [Integration of Swagger into the framework Nest.js]. URL: https://github.com/nestjs/swagger#readme (дата обращения: 22.03.2023).

14. Biblioteka sequelize-typescript [Sequelize-typescript library]. URL: https://github.com/sequelize/sequelize-typescript#readme (access date: 22.03.2023).

15. Dokumentaciya Passport.js [Documentation Passport.js]. URL: https://github.com/typestack/class-validator#readme (access date: 22.03.2023).

16. Biblioteka class-validator [Class-validator library]. URL: https://github.com/typestack/class-validator#readme (access date: 22.03.2023).

Информация об авторах

Константин Абрикосов – студент, кафедра алгебраических и информаицонных систем, Институт математики и информационных технологий, Иркутский государственный университет, г. Иркутск, e-mail: kobelogub@gmail.com

Виктория Алексеевна Попова – аспирант, ассистент кафедры алгебраических и информационных систем, Институт математики и информационных технологий, Иркутский государственный университет, г. Иркутск, e-mail: victorypopova1@gmail.com

Authors

Constantine Abrikosov – Student, Algebraic and Information Systems Department, Institute of Mathematics and Information Technology, Irkutsk State University, Irkutsk, e-mail: kobelogub@gmail.com

Victoria Alexeevna Popova – Graduate Student, Assistant of Algebraic and Information Systems Department, Institute of Mathematics and Information Technology, Irkutsk State University, Irkutsk, e-mail: victorypopova1@gmail.com

Для цитирования

Абрикосов К., Попова В.А. Статическая типизация в веб-приложениях на Node.js: процесс внедрения и выявление ошибок несоответствия типов // «Информационные технологии и математическое моделирование в управлении сложными системами»: электрон. науч. журн. – 2023. – №2(18). – С.23-32 – DOI: 10.26731/2658-3704.2023.2(18).23-32 – Режим доступа: http://ismm-irgups.ru/toma/218-2023, свободный. – Загл. с экрана. – Яз. рус., англ. (дата обращения: 17.06.2023)

For citations

Abrikosov C., Popova V.A. Static type-checking in web applications on Node.js: integration process and identification of type mismatch errors // Informacionnye tehnologii i matematicheskoe modelirovanie v upravlenii slozhnymi sistemami: elektronnyj nauchnyj zhurnal [Information technology and mathematical modeling in the management of complex systems: electronic scientific journal], 2023. No. 2(18). P. 23-32. DOI: 10.26731/2658-3704.2023.2(18).23-32 [Accessed 17/06/23]