УДК 004.42

# ИМИТАЦИОННЫЙ СЕРВЕР REST API НА ОСНОВЕ СПЕЦИФИКАЦИИ ДЛЯ ТЕСТИРОВАНИЯ ВЕБ-ПРИЛОЖЕНИЙ

**Алпатов А.Н.**

*ФГБОУ ВО «МИРЭА – Российский технологический университет», Москва, e-mail: aleksej01-91@mail.ru*

Интерфейсы прикладного программирования (Application Programming Interfaces (API)) находят всё большее применение при создании современных программных систем различного назначения, спроектированных и реализованных на основе многослойных клиент-серверных архитектур. Сложность проектирования и реализации данного типа систем приводит к тому, что на практике часто возникает разрыв в скорости реализации серверной и клиентской частей программного обеспечения, интегрированных с помощью интерфейсов прикладного программирования. То есть возникает такая ситуация, что по мере готовности фронтендовой части приложения серверная часть приложения может быть не полностью реализована и, следовательно, провести полноценное тестирование, например фронтенд-части приложения, невозможно. В данной работе будет рассмотрен вопрос создания имитационных серверов, которые можно применять для тестирования современных веб-приложений, использующих связанность между компонентами системы, посредством интерфейсов прикладного программирования (API). В качестве основного результата работы представлен подход к построению имитационных API для распределённых приложений на основе взаимодействия с реалистичными данными. В работе подробно описан способ построения имитационных API на основе рабочей спецификации, с промежуточным слоем программного обеспечения, для разрешения задачи с возникновением ошибки CORS и возможности переключения между имитационным и рабочим интерфейсом. Определены основные преимущества и недостатки представленного решения и рекомендации по его применению на практике.

**Ключевые слова: интерфейсы прикладного программирования, API, RESTful, тестовые двойники, MOCK, REST, разработка через тестирование, разработка через поведение**

# SIMULATION SERVER REST API BASED ON THE SPECIFICATION FOR TESTING WEB APPLICATIONS

**Alpatov A.N.**

*Federal State Budget Educational Institution of Higher Education «MIREA – Russian Technological University», Moscow, e-mail: aleksej01-91@mail.ru*

Application Programming Interfaces (APIs) are increasingly being used to create modern software systems for various purposes, designed and implemented on the basis of multilayer client-server architectures. The complexity of design and implementation of this type of systems leads to the fact that in practice there is often a gap in the speed of implementation of server and client parts of the software integrated through the application programming interfaces. That is, there is such a situation that as soon as the frontend part of an application is ready, the backend part of the application may not be fully implemented and, therefore, it is impossible to perform full-fledged testing, for example, the frontend part of the application. This paper will consider the creation of simulation servers that can be used to test modern web applications that use connectivity between system components through application programming interfaces (APIs). As the main result of the work the approach to building simulation APIs for distributed applications based on interaction with realistic data is presented. The paper describes in detail how to build simulation APIs based on the working specification. with an intermediate software layer, to solve the problem with the CORS error and the ability to switch between the simulation and working interface. The main advantages and disadvantages of the presented solution and recommendations for its practical application are defined.

**Keywords: application programming interface, API, RESTful, test doubles, MOCK, REST, test-driven development, behavior-driven development**

Использование программных интерфейсов для интеграции разрозненных частей программного обеспечения стало неотъемлемой частью современной веб-разработки. Так, на сайте any-api.com представлена документация «для более 1400 общедоступных API» [1]. Растущая потребность в интеграции программных компонентов привела к широкому применению интерфейсов для решения различных задач ИТ-индустрии. В работах [2; 3] представлен интересный подход к использованию интерфейсов прикладного программирования для обеспечения связанности функциональных блоков в микросервисной архитектуре.

Связывание программных систем посредством использования программных интерфейсов обеспечивает независимость разработки разрозненных компонентов друг от друга, возможность распараллеливания работ при проектировании и реализации частей программного обеспечения и в целом позволяет повысить скорость разработки и ускорить получение результата при итерационных моделях разработки.

Применительно к веб-сервисам, API также активно применяются для обеспе-

чения связанности между клиентской стороной (фронтенд) и серверной частью (бэкенд) веб-приложения, что позволяет вести часто независимую разработку серверной и клиентской частей веб-сервиса.

Таким образом, в ходе разработки может выделяться часть команды, отвечающая за серверную часть, и часть команды, отвечающая за фронтенд. Задачей команды, отвечающей за бэкенд, является реализация функционала разрабатываемой системы, при этом команда фронтенд-разработки не привязана к бэкендовой части и может вести разработку независимо от неё. Для неё часто важны лишь данные, предоставляемые посредством интерфейсов прикладного программирования. В реальных проектах получение данных от серверной части и их отображение происходит посредством HTTP-запросов обращения к соответствующим конечным точкам API и получения представления ресурсов в каком-то формате данных.

Однако на практике при использовании такого подхода к организации работ часто может возникать ситуация, связанная с отставанием реализации серверной части веб-приложения от клиентской части. Данное обстоятельство может существенно замедлить весь процесс разработки веб-сервиса, осложнить процесс тестирования реализованных компонентов и демонстрацию практической реализации (англ. Proof of concept) функциональных требований заказчика. Особенно остро проблема отставания серверной части от клиентской части проявляется в случаях использования техники разработки через тестирование (test-driven development) или через поведение (англ. behavior-driven development) для фронтенда, при которой тест, покрывающий вносимые изменения в программную часть фронтенда, не может быть исполнен в связи с отсутствием требуемого функционала, реализуемого серверной частью. На практике это приводит к необходимости дополнительной реализации и использования Mock-объектов, для имитации работы той части серверного приложения, к которой посредством программного интерфейса обращается фронтенд, что приводит к необходимости написания дополнительного кода, предназначенного только для проведения тестирования, что приводит к общему удорожанию проекта и снижению скорости разработки.

Ещё одной причиной широкого использования имитационных объектов API в разработке является необходимость обеспечения независимости тестовых примеров от внешних источников, например от источника данных, при модульном тестировании. Это необходимо для того, чтобы упростить повторное использование разработанных тестовых случаев в других проектах и задачах, а также провести полноценное модульное тестирование функционального блока, без риска перехода в интеграционное тестирование.

Исходя из вышеизложенного, можно сделать вывод, что потребность в имитационных интерфейсах прикладного уровня только возрастает и задача создания имитационных API с возможностью конфигурирования является актуальной. В данной работе будет предложен подход к построению имитационных RESTful API с возможностью конфигурирования под различные задачи на основе спецификации.

Целью данного исследования является разработка новых подходов к построению имитационных интерфейсов прикладного программирования, основанных на повторном использовании спецификации рабочего API и с возможностью конфигурирования режимов работы, для обхода ошибки CORS.

В течение долгого времени основой для построения веб-сервисов был SOAP (Simple Object Access Protocol), являющийся развитием метода вызова удалённых процедур XML-RPC. SOAP – прежде всего специализированный протокол, который использует WSDL в качестве основы для описания веб-сервиса и его взаимодействия посредством передачи сообщений. В настоящее время SOAP разрабатывается и поддерживается World Wide Web (W3C).

Понятие «REST» было предложено в 2000 г. Роем Филдингом [4]. REST не является официальным протоколом, а представляет собой набор рекомендаций по архитектурному проектированию веб-сервисов. В отличие от SOAP, где в качестве основного формата кодирования сообщений используется XML, REST, как архитектурный стиль, не привязан к конкретному формату сообщений (данных). В качестве формата может быть XML, JSON, RSS и т.д. Формат сообщения непосредственно выбирается при проектировании API. Но всё-таки стоит учесть, что на практике большинство реализаций API REST используют формат JSON (нотацию объектов JavaScript), но возможны ситуации, когда совместно с JSON используют XML. Также REST позволяет обеспечить значительно более низкую связанность между компонентами, чем интеграция посредством SOAP.

В своей работе [4] Рой Филдинг предложил архитектурный подход для реализации интероперабельности в веб-приложениях посредством использования универсального идентификатора ресурсов (англ. URI) и протокола передачи данных HTTP. REST

использует понятие ресурса, как некоторого объекта, к которому можно получать доступ в соответствии с требуемым HTTP-методом. То есть взаимодействие с REST API происходит путём обмена представлением выбранного ресурса. Данный подход позволяет реализовать не только полноценный программный интерфейс, но и определить действия, которые можно производить на ресурсе. Обычно REST отображает операции создания, чтения, обновления (редактирования) и удаления данных (CRUD) посредством соответствующих HTTP-запросов, что позволяет использовать простой набор вызовов для API. На рис. 1 показана обобщённая модель REST API.

Как было уже сказано выше, REST является лишь рекомендацией по архитектурному проектированию интегрированных слабосвязанных приложений, соответственно, выделяют некоторые базовые принципы, которым должен соответствовать REST API. В 5-й главе своей диссертации Рой Филдинг выделил 6 основных ограничений, которым должен соответствовать REST [4]:

– разделение функциональности на клиент – сервер;

– отсутствие сохранений состояний на сервере / состояние сеанса полностью зависит от клиента;

– кеширование данных на стороне клиента;

– единообразие интерфейса для обеспечения интероперабельности;

– использование многослойных архитектур;

– расширение функциональности клиента только по требованию.

На практике обеспечить полное соответствие архитектурным требованиям REST, которые предложены Роем Филдингом, бывает затруднительно, и такое решение приводит к созданию REST based services, то есть таких сервисов, которые частично соответствуют вышеуказанным принципам. В случае если решение полностью обеспечивает указанные выше принципы, то такие API принято называть RESTful. Стоит понимать, что со временем разработчики, помимо указанных выше ограничений, стали добавлять и другие.

### *Имитационные API и способы их реализации*

Рассмотрим более подробно задачу создания имитационных интерфейсов прикладного программирования для обеспечения модульного тестирования. Как было уже сказано выше, интерфейсы прикладного программирования играют важную роль для обеспечения интероперабельности в современных распределённых системах. В случае веб-сервисов это приводит к зависимости веб-интерфейса от более сложного объекта – серверной части. Данное обстоятельство приводит к усложнению процессов модульного тестирования. Под модульным тестированием понимается процесс «тестирования каждой атомарной функциональности приложения отдельно, в искусственно созданной среде» [5]. Таким образом, для того чтобы провести полноценное модульное тестирование, например веб-интерфейса, необходимо разрешить все возникшие зависимости от внешних по отношению к тестированному компоненту элементов, что в случае отставания в скорости разработки серверной части от клиентской части сделать бывает затруднительно.

Одним из подходов, позволяющих решить данную задачу, является использование сущностей, которые имитируют поведение реальных объектов. Использование имитационных объектов при тестировании позволяет сосредоточиться именно на тестируемом коде, а не на тестировании корректности связывания с другим модулем или проверке функционирования другого компонента с тестируемым компонентом в условиях их совместного функционирования. То есть такой подход позволяет провести полноценное модульное тестирование без риска незаметного перехода в интеграционное тестирование.

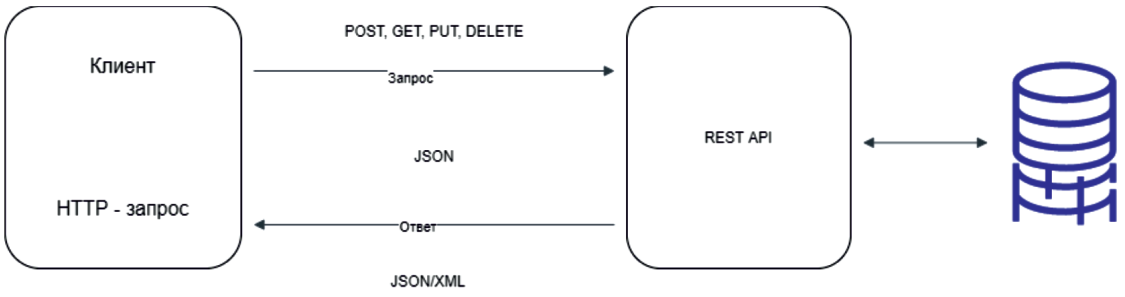


*Рис. 1. Обобщённая модель REST API*

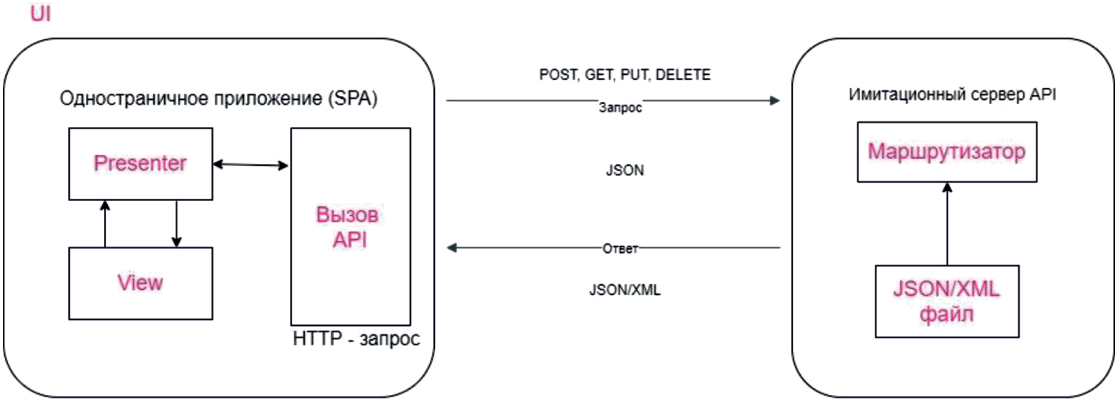


*Рис. 2. Пример архитектуры имитационного сервера API*

В отечественной и зарубежной литературе отсутствует единый подход к определению такого типа объектов. Часто в литературе встречаются такие названия, как dummy, fake, stub, mock, для обозначения такого рода объектов. Джерард Месарош данные объекты обозначает единым термином «тестовый двойник» [6].

Применительно к задаче обеспечения тестирования в распределённых системах, в случае использования интерфейсов прикладного программирования для обеспечения связанности разрозненных компонентов, необходимо создать тестовых двойников реальных API, что позволит отдельно имитировать работу серверной или клиентской частей без их практической реализации. На практике для таких целей распространён подход, показанный в работах [7; 8], основанный на конфигурировании отдельного сервера таким образом, чтобы сервер, при обращении к нему с соответствующим HTTP-запросом, выдавал требуемые данные. Для это требуется определить и дать описание каждой конечной точки, задействованной в проекте, и задать представление ответа в требуемом формате (JSON). В качестве наиболее распространённого стека технологий обычно используется минимальная связка NodeJS и пакета (зависимости) json-server. Для такого решения характерно частое использование сгенерированных случайных данных, а не использование реальных данных. Таким образом, данный подход сводится к фактической замене ответа, создаваемого программным кодом, на ответ из файла.

На рис. 2 показана архитектура такого решения для одностраничного приложения (SPA) на основе архитектурного шаблона MVP (Model-View-Presenter).

Стоит отметить, что данный подход обладает существенными минусами. В частности, сложности в его использовании могут возникать при тестировании веб-приложений, разработанных для разных типов ролей пользователей системы, например для администратора системы и абонента. Данное обстоятельство выражается в необходимости конфигурации под каждую конкретную роль, реализации сценариев аутентификации, часто с заранее прописанными пользователями системы и их ролями, а также в создании минимального так называемого UserService, цель использования которого заключается в управлении пользователями. При этом многие групповые политики в области информационной безопасности организации могут запрещать совместное использование ресурсов между различными источниками данных. В частности, браузеры, соблюдая принцип одного источника (Same Origin Policy) [9], будут предотвращать перекрёстные запросы, что приводит к невозможности использования такого решения без дополнительного использования Cross-Origin Resource Sharing (CORS).

### *Реализация имитационных интерфейсов прикладного программирования на основе спецификации*

Предлагаемый метод построения имитационных интерфейсов прикладного программирования заключается в использовании подхода, основанного на автоматизированном конфигурировании имитационного сервера под конкретную задачу тестирования. Для этого предлагается использовать подход, базирующийся на концепции разработки API через спецификацию. Разработка через спецификацию является на сегодняшний момент набирающим популярность способом проектирования и реализации интерфейсов прикладного программирования. В настоящее время разработано большое количе-

ство спецификаций API. Среди них можно выделить OpenAPI (Swagger) [10; 11], API Blueprint [12], Stoplight [13].

Такой подход, в отличие от подхода, ориентированного на реализацию, позволяет дать описание функционирования и поведения проектируемого интерфейса прикладного программирования, обеспечить будущих пользователей API подробной документацией по использованию каждого конкретного метода. Многие разработанные спецификации содержат дополнительные инструменты, позволяющие автоматизированно контролировать вносимые изменения в спецификацию API, одновременно внося изменения в документации к программному интерфейсу.

Разработка через спецификацию хорошо интегрируется с техникой разработки через поведение (Behavior Driven Development). Разработка через поведение является дальнейшим развитием техник разработки через тестирование (Test-Driven Development) и представляет собой технику реализации программного кода, при которой первоначально пишется тест, а только потом тот программный код, который позволит полностью выполнить данный тест, а сам тест представляет собой спецификацию, описывающую требуемое поведение тестируемой сущности. В случае выполнения проектирования и разработки интерфейса прикладного программирования через спецификацию первоначально даётся обобщённое описания API, его поведения и взаимодействия с другими сущностями, а только потом даётся его реализация на основе описанной спецификации.

Данное свойство техники разработки через спецификацию имеет важное значение для построения имитационных API. Как было показано выше, в случае реализации имитационного интерфейса прикладного программирования часто используется подход, представляющий собой фактическую замену сгенерированных реализуемым функционалом передаваемых данных на данные, выдаваемые при обращении к API из структуры данных, хранящейся в файле JSON. Следовательно, разработчику необходимо повторно провести полноценную реализацию API, то есть определить логику, дать описание каждой конкретной точке и т.д. Предлагаемый подход позволяет этого избежать за счёт повторного использования спецификации API, что позволит упростить и ускорить процесс реализации имитационного интерфейса прикладного программирования.

Архитектура программной системы, реализующей данный подход, представлена на рис. 3.

В предлагаемом решении можно выделить основные блоки:

– *Система непрерывной интеграции* является внешней по отношению к разрабатываемому решению системой, предназначение которой заключается в обеспечении непрерывной интеграции и непрерывной доставки.

– *Менеджер тестов* является подсистемой контроля и управления тестированием.

– *Имитационный сервер* предназначен для имитации ответов при обращении веб-приложения или тестового фреймворка.

– *Тестовый фреймворк* предназначен для генерации тестового сценария в рамках техники разработки через поведение (BDD), а также для контроля за вносимыми изменениями в спецификацию интерфейса прикладного программирования веб-приложения. В случае изменения в спецификацию API веб-приложения тестовый фреймворк должен автоматически внести изменения в спецификацию на стороне имитационного API.

– *Разрабатываемое веб-приложение* содержит в себе фронтендовую и бэкендовую части, интегрируемые между собой посредством интерфейса прикладного программирования.

Представленное решение позволяет запускать веб-приложение, как в рабочем режиме, при котором взаимодействие компонентов веб-приложения будет происходить с реальным API, реализованным на основе спецификации OpenAPI 3.0, так и в тестовом режиме, при котором реальный API будет заменяться на имитационный интерфейс прикладного программирования. При этом, в силу того что тестовый фреймворк контролирует все вносимые изменения в спецификацию разрабатываемого API, это позволит упростить процесс внесения в функционал имитационного сервера. В качестве программного обеспечения для создания имитационного сервера предлагается использовать связку Node.js [14], Prism [15] и Redoc [16]. Prism представляет собой HTTP-сервер, реализованный по принципу «вначале проектирование, потом всё остальное» (Design-First) [17]. Использование Prism в данном проекте обусловлено для обеспечения построения полноценного имитационного интерфейса в соответствии со спецификацией, взятой из основной ветки проекта, с возможностью генерации как статического, так и динамического ответов. Для этого необходимо вызвать Prism из тестового фреймворка, с передачей параметров местоположения спецификации интерфейса прикладного программирования в вызов. Формат данного вызова может быть следующим: *prism mock -p 4010 l местоположение_yaml*.

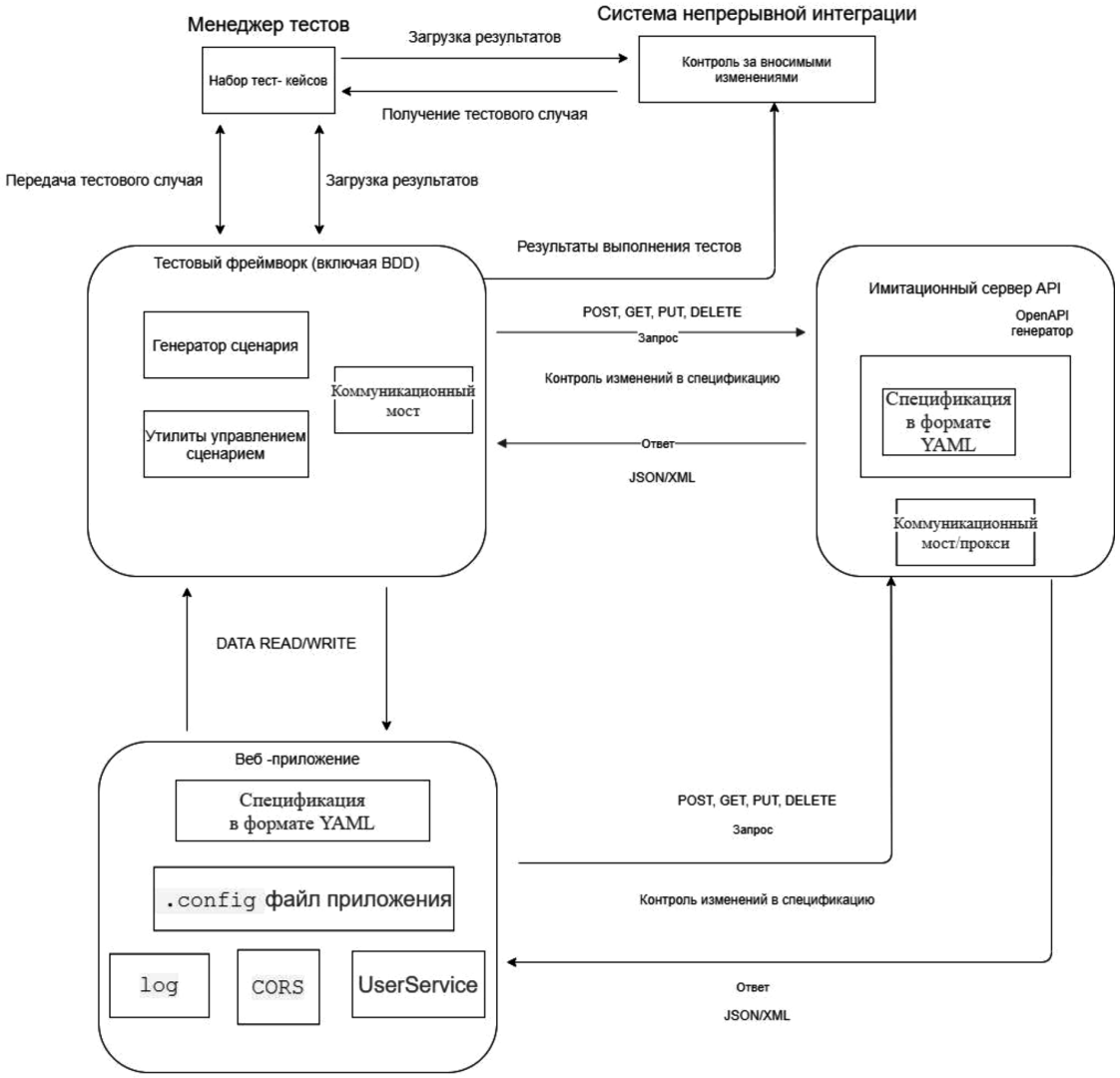


*Рис. 3. Архитектура системы для ведения разработки через поведение с использованием имитационного API*

Для того чтобы создать полноценную документацию по спецификации OpenAPI, можно воспользоваться Redoc. Использование в данной системе программного обеспечения Redoc позволит создать полноценную документацию по API, с контролем вносимых изменений в спецификацию API в формате yaml. Для этого достаточно передать в Redoc информацию о местоположении файла спецификации в соответствии с командой *redoc-cli bundle -o местоположение_html местоположение_yaml* и, в дальнейшем, запустить Redoc сервер командой *redoc-cli serve.*

Для того чтобы иметь возможность переключаться между имитационной и рабочей версиями API, в системе предусматривается наличие конфигурационного файла, представляющего собой JavaScript-код, который предназначен для того, чтобы обеспечить технологию единого входа (SSO), то есть необходимо реализовать механизм авторизации. Это, в свою очередь, позволит относительно легко переключаться между рабочим и имитационным интерфейсом путём изменения параметра URL-адреса API.

Как было отмечено выше, в случае если веб-сервис явно не настроен на обработку запросов к стороннему серверу в рамках совместного использования ресурсов между источниками, то при попытке обращения к ресурсу за пределами источника доступ к ресурсу будет заблокирован. Для решения данной задачи в рамках разработанной системы предлагается использовать реали-

зацию собственного прокси. Назначение прокси в данном случае заключается в разрешении запросов из любых источников путём использования *Access-Control-Allow-Origin: **. Однако данное решение может показывать невысокую производительность. Для улучшения производительности предложенного решения в разработанную систему вносится слой промежуточного ПО, который реализован с использованием веб-фреймворка Express. Задача промежуточного слоя заключается в установке флага *changeOrigin* в значение *«true»*.

## Заключение

В данной работе предложен способ построения имитационных интерфейсов прикладного программирования для использования при разработке веб-приложений, реализуемых в рамках методологии разработки через поведение. Данный способ позволяет упростить процесс разработки и тестирования программного обеспечения за счёт повторного использования спецификации рабочего API при построении имитационного API. Предложенная связка промежуточного программного обеспечения позволяет контролировать все вносимые изменения в рабочую спецификацию интерфейса прикладного программирования для автоматического исправления в спецификации имитационного API. Такой подход позволяет упростить процесс создания имитационных интерфейсов, обеспечив полное соответствие рабочим спецификациям в части функционала API. В предложенной системе реализована система конфигурирования для возможности переключения между рабочей и тестовой конфигурацией приложения, что позволяет легко переключиться между имитационным и рабочим API. Дополнительным преимуществом является также автоматизация процесса документирования API за счёт использования спецификации API [18] и соответствующего модуля системы ReDoc. Дополнительно предложен подход к ошибке CORS, который основан на использовании слоя промежуточного программного обеспечения.

В качестве недостатка можно отметить, что предложенный подход наиболее применим для интерфейсов прикладного программного обеспечения, реализация которых происходит в рамках разработки через спецификацию, следовательно, использование такого подхода в рамках разработки, ориентированной на реализацию, затруднительно.

Исходя из полученных результатов работы, можно сделать вывод о том, что использование предложенного подхода позволяет создавать полноценные имитационные интерфейсы прикладного программирования.

## Список литературы

1. AnyAPI: Documentation and Test Consoles for Over 1400 Public APIs. [Electronic resource]. URL: https://any-api.com/ (date of access: 18.11.2020).

2. Dragoni N., Giallorenzo S., Lafuente A.L., Mazzara M., Montesi F., Mustafin R., Safina L. Microservices: Yesterday, Today, and Tomorrow. Present and ulterior software engineering. Springer. Cham. 2017. P. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

3. Jaramillo D., Nguyen D.V., Smart R. Leveraging microservices architecture by using Docker technology. Southeast Con. 2016. IEEE. 2016. P. 1–5. DOI: 10.1109/SECON.2016.7506647.

4. Fielding R.T. Architectural styles and the design of network-based software architectures. Ph.D. dissertation, University of California, Irvine, 2000. 162 p.

5. Модульное тестирование. [Электронный ресурс]. URL: https://qalight.com.ua/baza-znaniy/modulnoe-testirovanie (дата обращения: 18.11.2020).

6. Meszaros G. xUnit test patterns: Refactoring test code. Pearson Education, 2007. 301 p.

7. 5 Steps to Build a Node.js Mock Server and API With Random Data. [Electronic resource]. URL: https://medium.com/better-programming/build-a-nodejs -mock-server-api-with-random-data-86303db9156a (date of access: 18.11.2020).

8. How to Mock an API with random data from NodeJS. [Electronic resource]. URL: https://dev.to/mrfrontend/how-to-mock-an-api-with-random-data-from-nodejs-dda (date of access: 18.11.2020).

9. Same Origin Policy – Web Security. [Electronic resource]. URL: https://www.w3.org/Security/wiki/Same_Origin_Policy (date of access: 18.11.2020).

10. OpenAPI Specification. [Electronic resource]. URL: https://swagger.io/specification/ (date of access: 18.11.2020).

11. Karlsson S., Causevic A., Sundmark D. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE. 2020. P. 131–141. DOI: 10.1109/ICST46399.2020.00023.

12. API Blueprint Specification. [Electronic resource]. URL: https://apiblueprint.org/documentation/specification.htm (date of access: 18.11.2020).

13. Design, Document & Build Quality APIs Faster. [Electronic resource]. URL: https://stoplight.io/ (date of access: 18.11.2020).

14. Node.js. [Electronic resource]. URL: https://nodejs.org/en/ (date of access: 18.11.2020).

15. Prism, an Open-Source HTTP Mock & Proxy Server. [Electronic resource]. URL: https://stoplight.io/open-source/prism (date of access: 18.11.2020).

16. Redoc. [Electronic resource]. URL: https://github.com/Redocly/redoc (date of access: 18.10.2020).

17. Patni S. API Design and Modeling. Apress, Berkeley, CA. 2017. P. 11–31. DOI: 10.1007/978-1-4842-2665-0_2.

18. Vasconcelos V.T., Martins F., Lopes A., Burnay N. HeadREST: A Specification Language for RESTful APIs. In Models, Languages, and Tools for Concurrent and Distributed Programming. Springer. 2019. P. 428–434. DOI: 10.1007/978-3-030-21485-2_23.