Требования к реализации REST-сервиса

Черновик

Крайне приветствуется обратная связь, комментарии и дополнение документа силами читателей.

Версия 1.0

История изменений

Дата	Версия	Описание	Автор
	1.0		Karinski, Andrey

Оглавление

Внимание! К инструкции следует добавить ссылку на пример реализации перечисленных ниже паттернов реализации REST.

1. Введение

Современные информационные системы, как правило, создаются на основе микросервисной архитектуры, базовый элемент которой, сервис, может быть выполнен согласно архитектурному стилю REST (сокращение от Representational State Transfer, паттерн для распределенных систем в сети, был впервые представлен Roy Fielding в 2000 году в своей диссертации «Architectural Styles and the Design of Network-based Software Architectures»). На сегодняшний день REST практически вытеснил все остальные подходы, в том числе дизайн основанный на *SOAP* и *WSDL*. Следует учесть современные альтернативы в виде *GraphQL* и *gRPC*, но об этом будет написано в других документах.

Данная инструкция, несмотря на категоричное название, содержит *рекомендации*, а не строгие *требования,* так как в любом конкретном случае последнее слово за разработчиком, ответственным за реализацию сервиса, так как ему могут быть очевидны особые условия, требующие отклониться от данной инструкции.

Так как содержимое документа может вызывать разногласия либо у кого-то из читателей имеются важные соображения и дополнения, то приветствуется коллективное владение данным документам.

1.1. Цель

Сервисы общаются между собой через собственные API, описанные согласно общим согласованным стандартам и принципам проектирования. Такой подход облегчает работу программистов и позволяет быстрее реализовать сложные бизнес-процессы, используя сервисы в качестве строительных блоков.

Создаваемые API сервисов должны максимально соответствовать архитектурному стилю REST, особенно это важно для сервисов, доступных для внешних систем и стороннего использования. Для поддержки этих целей разработан набор правил, стандартов и соглашений, которые применяются к разработке API-интерфейсов REST.

Основная аудитория для этого документа - разработчики, но он может быть полезен как другим членам команды, так и представителям заказчика.

Примечание	Пример
Машиночитаемый текст (URL, HTTP-методы, JSON и т.д.) представлен шрифтом .	`https://todolist.ru`
URI, содержащие блоки переменных, указываются в соответствии с шаблоном URI RFC 6570.	URL-адрес, содержащий переменную user_id, будет отображаться как `https://todolist.ru/users/{user_id}`
Заголовки HTTP записываются в синтаксисе camelCase + дефисы.	`User-Request-Id`

1.2. Область действия

Документ предназначен прежде всего для backend-разработчиков, но будет полезен и для frontend-разработчиков, тестировщиков и аналитиков. Стоит ориентироваться на содержимое документа прежде всего при создании публичных REST API.

1.3. Определения, сокращения и аббревиатуры

Все специальные термины и сокращения читателям документа знакомы.

1.4. Ссылки

При создании документа использовались следующие стандарты и рекомендации:

- Representational State Transfer (REST)
- Hypertext Transfer Protocol (HTTP/1.1) - RFC 7231
- Additional HTTP Status Codes - RFC 6585
- PATCH Method for HTTP - RFC 5789
- Prefer Header For HTTP - RFC 7240
- Uniform Resource Identifier (URI): Generic Syntax - RFC 3986
- URI Template - RFC 6570
- Web Linking - RFC 5988
- IANA Link Relations
- JSON Data Interchange Format as described in - RFC 7159
- JSON Schema: core definitions and terminology Draft 4
- JSON Schema: interactive and non interactive validation Draft 4
- JSON Hyper-Schema: Hypertext definitions for JSON Schema Draft 4
- JSON Patch
- JavaScript Object Notation (JSON) Patch - RFC 6902
- JavaScript Object Notation (JSON) Pointer - RFC 6901
- OpenAPI Specification v3.0.2
- Country Codes - ISO 3166

- Currency Codes - ISO 4217
- Tags For Identifying Languages BCP 47
- Date and Time on the Internet: Timestamps - RFC 3339
- Date and Time Formats - ISO 8601
- Semantic Versioning 2.0.0

1.5. Обзор

Документ состоит из описания основополагающих принципов построения REST-API, большинство разделов содержат примеры. В конце документа детально описаны шаблоны реализации REST-сервисов.

2. Общие принципы

2.1. Используемые термины

2.1.1. Ресурс (Resource)

Ключевая абстракция в REST. Согласно принципу REST, любое понятие, которое может быть названо, может быть ресурсом: сущность, документ, служба, набор других ресурсов и т.д. Фактически, ресурс - это концептуальное отображение набора сущностей, а не одной сущности. Значения в наборе могут быть представлениями ресурса и/или идентификаторами ресурса. Ресурс также может отображаться на пустой набор, что позволяет делать ссылки на концепцию до того, как будет реализована какая-либо реализация этой концепции.

2.1.2. Идентификатор ресурса (Resource Identifier)

REST использует идентификатор ресурса для выбора конкретного экземпляра ресурса, участвующего во взаимодействии между компонентами. Владелец ресурса (например, организация, предоставляющая API), который назначил идентификатор, позволяющий ссылаться на ресурс, отвечает за поддержание семантической достоверности отображения с течением времени.

2.1.3. Представление (Representation)

Компоненты REST выполняют действия с ресурсом, используя представление для получения текущего или предполагаемого состояния этого ресурса и передавая это представление между компонентами. Представление - это последовательность байтов плюс метаданные представления для описания этих байтов. Под представлением можно понимать JSON, XML, plain text в определенном формате, что позволяет понимать состояние ресурса и модифицировать его. Представление, которое модифицирует состояние ресурса и представление, отображающее ресурс, не обязательно друг другу соответствуют. Например, представление для отображения `user` может содержать ID (так как он уже существует), а представление для создания нового `user` ID не содержит, так как должно быть сгенерировано сервером.

2.1.4. Доменная модель (Domain)

Модель предметной области (доменная модель) - система абстракций, которая описывает выбранные аспекты области знаний или деятельности. Абстракции включают данные, используемые в бизнесе, и правила, которые бизнес использует в отношении этих данных.

Примеры доменов: Платежи, Клиенты, Закупки.

2.1.5. Возможность (Capability)

Возможность представляет собой бизнес-логику организации, ориентированную на клиента. Возможности могут быть использованы для организации в виде стабильного, ориентированного на бизнес представления о своей системе набора API. Фактически, возможности позволяют более тонко организовать API внутри доменов. Фактически, бизнес-возможности - это ресурсы и методы для управления ими, ориентированные на решение определенных и строго очерченных бизнес-задач.

Примеры возможностей для домена Клиенты: Регистрация, Сопровождение, Поддержка.

2.1.6. Пространство имен (Namespace)

Возможности управляют моделированием сервисов и пространств имен. Пространства имен являются частью модели бизнес-возможностей. Пространства имен должны отражать домен, который логически группирует набор бизнес-возможностей.

2.1.7. Сервисы

Сервисы предоставляют общий API для доступа к ресурсам и манипулирования ими, независимо от программного обеспечения, которое обрабатывает запрос. Сервисы - это общие части системы, которые могут выполнять любое количество функций.

2.1.8. Клиенты

Потребители API и ресурсов, внутренние и внешние.

3. Принципы дизайна сервиса

В этом разделе собраны принципы, которыми руководствуются при разработке сервисов, которые предоставляют API внутренним и внешним разработчикам и потребителям.

Ниже приведены основные принципы проектирования сервиса.

3.1. Низкая связанность

Услуги и потребители должны быть слабо связаны друг с другом.

Мера связи сопоставима с уровнем зависимости. Этот принцип рекомендует разрабатывать API с акцентом на уменьшение (ослабление) зависимости между контрактом, его реализацией и потребителями услуг. Принцип ослабления связей способствует независимому проектированию и развитию логики и реализации сервиса, при этом подчеркивая базовую совместимость с потребителями.

Принцип подразумевает следующее:

- Контракт сервиса не должен раскрывать детали реализации.
- Контракт сервиса может развиваться, не влияя на существующих потребителей.
- Сервис определенного домена может развиваться независимо от других доменов.

3.2. Инкапсуляция

Доменная служба может получать доступ к данным и функциональным возможностям, которыми она не владеет, только через другие сервисные контракты.

Принцип подразумевает следующее:

- Сервис имеет четкую границу изоляции - четкую сферу ответственности с точки зрения функциональности и данных.
- Служба не может предоставить данные, которые ей непосредственно не принадлежат.

3.3. Стабильность

Сервисные контракты должны быть стабильными.

Сервисы должны быть разработаны таким образом, чтобы их контракт для существующих клиентов никогда не менялся. Если сервисный контракт должен развиваться несовместимым образом для потребителя, об этом следует четко сообщить.

Этот принцип подразумевает следующее:

- Существующие клиенты сервиса должны поддерживаться в течение заданного периода времени.
- Дополнительные функциональные возможности должны быть введены таким образом, чтобы не влиять на существующих потребителей.
- Должна быть четко прописана политика миграции.

3.4. Переиспользование

Сервисы должны разрабатываться так, чтобы их можно было многократно использовать в разных контекстах и для разных потребителей.

Основная цель API - обеспечить быструю и эффективную разработку приложений за счет использования и объединения сервисов. Это возможно только в том случае, если сервисные контракты были разработаны с учетом нескольких вариантов использования и нескольких потребителей. Этот принцип требует, чтобы сервисы развивались таким образом, чтобы они могли использоваться несколькими потребителями и в разных контекстах, некоторые из которых могут развиваться со временем.

Принцип подразумевает следующее:

- Контракт сервиса должен быть разработан не только для непосредственного контекста, но и для поддержки и/или расширения использования несколькими потребителями в разных бизнес-контекстах.
- Контракт сервиса может нуждаться в постепенном развитии для поддержки множества бизнес-контекстов и потребителей с течением времени.

3.5. Контрактно-центрический дизайн

Функциональность и данные должны предоставляться только через стандартные сервисные контракты.

Этот принцип предполагает, что все функции и данные должны быть доступны только через стандартные сервисные контракты, поэтому потребители услуг могут понимать и получать доступ к функциональности и данным только через сервисные контракты.

Принцип подразумевает следующее:

- Функциональность и данные не могут быть поняты или доступны вне контрактов.
- Каждый фрагмент данных (например, полученный из хранилища) принадлежит только одному сервису.

3.6. Целостность

Сервисы должны следовать общему набору правил, стилей взаимодействия, справочников и общих типов.

Этот принцип повышает удобство использования API за счет сокращения кривой обучения потребителей новых услуг.

Принцип подразумевает следующее:

- Имеется набор стандартов для сервисов.
- Сервис использует общие и согласованные справочники и типы данных.
- Используются совместимые стили взаимодействия сервисов и уровни детализация контрактов.

3.7. Простота использования

Сервисы должны быть простыми в использовании для потребителей и внешних систем, и способными к композиции.

Сервис, которым неудобно пользоваться, уменьшает преимущества микросервисной архитектуры, побуждая потребителей находить альтернативные механизмы для доступа к той же функциональности. Способность к композиции означает, что сервисы могут быть легко объединены, потому что их контракты и протоколы согласованы.

Принцип подразумевает следующее:

- Контракт легко обнаруживается, он понятен.
- Контракты и протоколы согласованы во всех аспектах (например, аутентификация, обработка ошибок, разбиение на страницы и т.д.)
- Сервис в явном виде имеет владельца, чтобы потребители могли обращаться к нему напрямую для решения своих проблем.
- Потребитель может легко использовать и тестировать сервис.

4. HTTP-методы, заголовки и коды возврата

4.1. Ресурсы и HTTP-методы

4.1.1. Бизнес-возможности и моделирование ресурсов

Различные бизнес-возможности (то есть данные и методы для управления ими) представлены через API в виде набора ресурсов. Следует максимально избегать дублирования функциональности в API.
Здесь нужно придумать хорошие примеры моделирования ресурсов.

4.1.2. HTTP-методы

Большинство ресурсов в сервисах легко описываются стандартной моделью CRUD (Create, Read, Update и Delete), тем более что такая модель отлично проецируется на стандартные HTTP-методы.

HTTP-метод	Описание

`GET`	Возвращает ресурс
`POST`	Создает ресурс или выполняет над ресурсом сложную операцию
`PUT`	Полностью обновляет существующий ресурс
`PATCH`	Частично обновляет существующий ресурс
`DELETE`	Удаляет ресурс

- Метод `GET` не должен иметь побочных эффектов и не должен менять состояние ресурса.
- Метод `POST` должен использоваться для создания нового ресурса в коллекции.
  Например: `POST` `https://todolist.ru/lists` создаст новый список задач.
- Метод `POST` должен использоваться для создания нового подресурса и установления его взаимосвязи с базовым ресурсом.
  Например: `POST` `https://todolist.ru/lists/12345/tasks` создаст задачу в списке 12345.
- Метод `POST` может использоваться в сложных операциях вместе с именем операции. Такой подход читается исключением из модели REST. Он более применим в тех случаях, когда ресурсы представляют бизнес-процесс, а операции - это шаги или действия, которые должны быть выполнены как его часть.
  Например: `POST` `https://todolist.ru/lists/12345/print` отправит на печать список задач 12345.
- Метод `PUT` следует использовать для обновления атрибутов ресурса или для установления связи между ресурсом и существующим подресурсом.
  Например: `PUT` `https://todolist.ru/lists/12345/tasks/10` обновит задачу 10 из списка задач 12345.

4.1.3. Обработка запросов

Предполагается, что тела запросов и тела ответов должны отправляться с использованием нотации JSON (облегченного представления данных, состоящего из неупорядоченных пар ключ-значение). JSON может представлять четыре примитивных типа (строки, числа, логические значения и null) и два структурированных типа (объекты и массивы).

При обработке вызова метода API должны соблюдаться следующие рекомендации:

- **Модель данных**
  Модель данных для представления должна соответствовать формату обмена данными JSON согласно стандарту RFC
- **Сериализация**
  Методы должны поддерживать в качестве типа контента `application/json`.
- **Строгость входных и выходных параметров**
  Методы должны строго следить за входными и выходными параметрами, их соответствию спецификации и документации, разрешенным интервалам и типам.

4.2. HTTP-заголовки

Цель HTTP-заголовков состоит в том, чтобы предоставить метаданные о теле сообщения или его отправителе единообразным и стандартизированным способом.

Имена заголовков не чувствительны к регистру.

Все используемые сервисом заголовки должны быть описаны в его контракте. Заголовки не должны содержать данные из предметной области.

Возможно, что какой-либо промежуточный сетевой компонент в цепочке вызовов (маршрутизатор, прокси) может отбросить HTTP-заголовок. По этой причине бизнес-логика основываться на заголовках не должна, не следует предполагать, что значение заголовка не было изменено при передаче сообщения.

Также следует ожидать, что инфраструктурные компоненты будут добавлять и изменять некоторые HTTP-заголовки, особенно касающиеся механизмов сетевой безопасности (`Authorization, Proxy-Authenticate`), кэширования (`Age, Cache-Control, ETag, Expires`) и маршрутизации (`Alternate, Content-Disposition`).

4.2.1. Стандартные HTTP-заголовки

Это заголовки, определенные в спецификации HTTP/1.1 (RFC 7231). Их назначение, синтаксис, значения и семантика определены и поняты компонентами инфраструктуры.

HTTP-заголовок	Описание

<table>
<tr><td>Accept</td><td>Список MIME-типов, которые ожидает клиент. Предполагается, что API поддерживают application/json.

Пример: Accept: application/json</td></tr>
<tr><td>Accept-Charset</td><td>Список кодировок, которые ожидает клиент. Предполагается, что поддерживается кодировка UTF-8.

Пример: Accept-Charset: utf-8</td></tr>
<tr><td>Content-Language</td><td>Определить политику локализации.

Пример: Content-Language: ru-RU</td></tr>
<tr><td>Content-Type</td><td>Позволяет клиенту определить MIME-тип тела запроса и ответа.
Единственный поддерживаемый тип на данный момент - application/json (application/problem+json)</td></tr>
<tr><td>Link</td><td>Согласно стандарту RFC 5988, ссылка описывает связь между двумя ресурсами, которые идентифицируются интернационализированными идентификаторами ресурсов (IRI).
Допустимо использование Link для организации постраничного вывода, хотя более корректным способом будет использование ссылок HATEOAS.

Указать ссылку на раздел, посвященный постраничному выводу.</td></tr>
<tr><td>Location</td><td>Используется для перенаправления получателя для завершения запроса (см. реализацию POST) или идентификации нового ресурса. Аналогично, заголовок можно использовать, но нет гарантий, что инфраструктурные компоненты на него не повлияют, поэтому более корректным способом будет использование ссылок HATEOAS.</td></tr>
<tr><td>Prefer</td><td>Используется для указания того, что клиентом предпочитается определенное поведение сервера, но при этом его не требуется для успешного завершения запроса.

Можно использовать для API следующие значения заголовка, если в документации API явно указана поддержка Prefer:

<table>
<tr><th>Значение Prefer</th><th>Описание</th><th>Комментарии</th></tr>
<tr><td>respond-async</td><td>Клиент предпочитает, чтобы сервер обрабатывал запрос асинхронно

Пример: Prefer: respond-async</td><td>Сервер возвращает код 202 (Accepted) и обрабатывает запрос асинхронно.
Клиент может вызвать метод GET, чтобы получить ответ позднее.
Указать ссылку на раздел, посвященный асинхронным операциям.</td></tr>
<tr><td>read-consistent</td><td>Клиент предпочитает, чтобы сервер возвращал ответ полностью согласованные данные из транзакционного хранилища.

Пример: Prefer: read-consistent</td><td>По умолчанию ожидается именно такое поведение.</td></tr>
<tr><td>read-eventual-consistent</td><td>Клиент предпочитает, чтобы сервер возвращал ответ либо из кэша, либо, предположительно, из хранилища данных с поддержкой "согласованности в конечном итоге".

Пример: Prefer: read-eventual-consistent</td><td>Например, из высокопроизводительной нереляционной БД.</td></tr>
<tr><td>read-cache</td><td>Клиент предпочитает, чтобы сервер возвращал ответ из кэша, если он доступен.

Пример: Prefer: read-cache</td><td>Если в кэше запрошенных данных нет, то сервер вернет данные из хранилища.</td></tr>
<tr><td>return=representation</td><td>Клиент предпочитает, чтобы сервер включал в ответ на успешный запрос текущее состояние ресурса.

Пример: Prefer: return=representation</td><td>Это может понадобиться для оптимизации трафика, например, в случае, когда после изменения ресурса методами POST или PUT следует метод GET для получения текущего состояния.</td></tr>
<tr><td>return=minimal</td><td>Клиент предпочитает, чтобы сервер возвращал в ответ на успешный запрос минимальный стандартный ответ.</td><td>Поведение по умолчанию.</td></tr>
</table>
</td></tr>
</table>

4.2.2. Распространение заголовков

Когда сервис получает заголовки запроса, он должен передавать соответствующие заголовки дальше без изменения.

4.3. Коды состояния HTTP

Сервисы REST используют коды состояния для указания результатов выполнения HTTP-методов. Протокол HTTP предписывает, что результатом выполнения запроса будет целое число (код состояния) и сообщение (название состояния).

Протокол HTTP классифицирует коды состояния в определенных диапазонах.

4.3.1. Диапазоны кодов состояния

При ответе на запросы методы API должны использовать следующие диапазоны кодов состояния.

Диапазон	Семантика	Комментарии
2xx	Успех.	Успешное выполнение метода может быть различным, поэтому код указывает, каким именно образом это получилось.
4xx	Ошибка клиента.	Обычно это проблемы с запросом, данными в запросе, неверной аутентификацией или авторизацией и т.д. В большинстве случаев клиент может изменить свой запрос и повторно отправить его.
5xx	Ошибка сервера.	Серверу не удалось выполнить метод из-за сбоя или дефекта программного обеспечения. Коды состояния диапазона 5xx не должны использоваться для логической обработки ошибок.

4.3.2. Состояния

Успех и неудача относятся ко всей операции, а не только к части инфраструктуры или к части бизнес-логики кода.

Ниже приведены рекомендации по использованию кодов состояний:

- Об успехе должно быть сообщено кодом состояния в диапазоне 2xx.
- Коды состояния в диапазоне 2xx должны возвращаться, только если весь путь выполнения кода успешен (что относится и к бизнес-логике, и к инфраструктуре).
- Необходимо сообщать об ошибках в диапазоне 4xx или 5xx, и в случае системных ошибок, и при ошибках прикладной логики.
- В теле ответа должен передаваться согласованный специальный объект с информацией об ошибке в формате JSON (Указать ссылку на спецификацию объекта error).
- Сервер, возвращающий код состояния в диапазоне 4xx или 5xx, должен вернуть тело ответа error.json
- В случае ошибок клиента в диапазоне кодов 4xx описание причины должна предоставлять достаточно информации, чтобы клиент мог определить причину ошибки и ее исправить.
- В случае ошибок сервера в диапазоне кодов 5xx описание причины и ответ об ошибке, следующий за error.json, должны ограничивать объем информации, чтобы избежать предоставления клиентам подробностей внутренней реализации сервиса. Разработчики сервисов должны получать детальную информацию об ошибках из логов.

4.3.3. Перечень разрешенных кодов состояния

Все API должны использовать только перечисленные ниже коды состояния. Коды состояния, выделенные желтым цветом, предназначены для разработчиков инфраструктуры веб-сервисов, которые сообщают об ошибках уровня инфраструктуры, связанных с безопасностью, маршрутизации и т.п.

- API не должны возвращать код состояния, который не определен в этой таблице.
- API могут возвращать только некоторые коды состояния, определенные в этой таблице.

Код состояния	Описание	Комментарии
`200 OK`	Успех	
`201 Created`	Используется в качестве ответа на выполнение метода `POST` для указания успешного создания ресурса. Если ресурс уже был создан (например, предыдущим выполнением того же метода), то сервер должен вернуть код состояния 200 OK.	Указать ссылку на раздел с описанием паттернов POST-методов

`202 Accepted`	Используется при выполнении асинхронного метода, чтобы указать, что сервер принял запрос и выполнит его позднее. Для более подробной информации можно перейти к разделу Асинхронные операции.	
`204 No Content`	Сервер успешно выполнил метод, но нет тела объекта для возврата.	Указать ссылку на раздел с описанием паттернов POST-методов
`400 Bad Request`	Запрос не может быть понят сервером. Используется, чтобы указать: • Данные не в ожидаемом формате. • Нарушены валидационные правила.	
`401 Unauthorized`	Запрос требует аутентификации.	
`403 Forbidden`	Клиент не авторизован для доступа к ресурсу, хотя его учетные учетные данные действительны. Сервис может использовать этот код в случае сложных правил авторизации, основанных на данных из предметной области.	
`404 Not Found`	Сервер не нашел ничего, что соответствует URI запроса. Это либо означает, что URI неверен, либо ресурс недоступен.	Например, в базе данных нет записи по этому ключу
`405 Method Not Allowed`	Не поддерживается запрошенный метод.	
`406 Not Acceptable`	Не поддерживается запрошенный клиентом формат контента.	Например, если клиент запросит `application/xml`
`415 Unsupported Media Type`	Не уверен, что нужно	
`422 Unprocessable Entity`	Запрошенное действие не может быть выполнено, Проблема в логике, а не в формате. Не уверен, что нужно. Возможно, стоит ограничиться 400.	
`429 Too Many Requests`	Клиент попытался отправить слишком много запросов за короткое время.	
`500 Internal Server Error`	Внутренняя ошибка сервера. Означает, что, клиент предоставил правильный запрос, но на сервере что-то пошло не так.	
`503 Service Unavailable`	Сервер в режиме обслуживания и временно не может обработать запрос.	

4.3.4. Сопоставление HTTP-методов и кодов состояния

Для каждого метода API следует использовать только коды состояния, помеченные значком «**x**» в таблице ниже.

Код состояния	`200 Success`	`201 Created`	`202 Accepted`	`204 No Content`	`400 Bad Request`	`404 Not Found`	`500 Internal Server Error`

GET	x				x	x	x
POST	x	x	x		x	x	x
PUT	x		x	x	x	x	x
PATCH	x		Нужно подумать	x	x	x	x
DELETE	x		Нужно подумать	x	x	x	x

- GET:
  Цель метода - получить ресурс. В случае успеха ожидается код 200, а в ответе - представление ресурса. В тех случаях, когда коллекции ресурсов пусты, все равно возвращается 200, а в ответе - пустой массив. Если же к ресурсу происходит обращение по идентификатору (например, GET `https://todolist.ru/lists/12345`), а элемента с таким идентификатором не существует, метод GET должен вернуть код 404.
- POST:
  Основная цель - создать ресурс.
  - Если ресурс ранее не существовал и был создан как во время выполнения метода, то следует вернуть код 201.
  - Ожидается, что при успешном выполнении в теле ответа будет возвращена ссылка на созданный ресурс.
  - Если используется подресурс, и идентификатора первичного ресурса не существует, то следует вернуть код 404. Например, POST `https://todolist.ru/lists/12345/tasks`, когда ресурса 12345 не существует.
- PUT:
  Цель метода - полное изменение существующего ресурс. Метод должен возвращать код 204, так как в большинстве случаев нет необходимости возвращать какой-либо контент. Информация из запроса возвращаться не должна.
  - В редких случаях, сгенерированные сервером при обновлении ресурса значения могут потребоваться для оптимизации трафика между клиентом и сервером (например, если клиент обязательно выполнит GET после PUT). В этих случаях допустимо вернуть код 200 и данные в теле ответа.
- PATCH:
  Цель метода - частичное изменение ресурса. Семантика та же, что и у метода PUT.
- DELETE:
  Цель метода - удаление ресурса. Метод должен возвращать код 204, аналогично PUT. Поскольку метод DELETE должен быть также идемпотентным, он возвращает 204, даже если ресурс уже был удален. Обычно потребителю API не важно, когда именно ресурс удален: сейчас или ранее. По этой же причине не имеет смысла использование кода 404.

5. Гипермедиа

5.1. HATEOAS

> Использование HATEOAS при реализации сервисов рекомендуется (в особенности для публичных API), но поддержка реализации может быть относительно дорогой и следует всегда оценивать целесообразность поддержки HATEOAS для каждого сервиса.

HATEOAS (*Hypermedia as the Engine of Application State*) — архитектурный паттерн RETS-сервисов, определяющий, как клиент взаимодействует с сетевым приложением, сервер которого обеспечивает динамический доступ через гипермедиа. REST-клиенту не требуется заранее знать, как взаимодействовать с сервером, так как сервер предоставляет метаданные доступных операций вместе с запрошенным ресурсом. Фактически, сервис, поддерживающий гипермедиа, предоставляет информацию для динамической навигации по REST-интерфейсам, включая гипермедиа-ссылки с ответами.

Гипермедиа, расширение термина гипертекст, является нелинейным носителем информации, который включает в себя графику, аудио, видео, простой текст и гиперссылки. Строгого стандарта для гипермедиа по-прежнему не существует, но имеет смысл ориентироваться на черновик спецификации HAL (*Hypertext Application Language, Язык Гипертекстовых Приложений*) - стандарт для определения гипермедиа в форматах JSON или XML. Кроме того, следует обратить внимание на JSON-схему hypershema.

5.2. Hypermedia-совместимый API

Реализация поддерживающего гипермедиа API предполагает, что сервис будет управляться методами, ссылки на которые он сам и будет предоставлять клиенту.
Ниже рассмотрен пример реализации такого сервиса.

Клиент начинает взаимодействие со службой через фиксированный URI `/users`, поддерживающий операции `GET` и `POST`.

Клиент решает выполнить операцию `POST`, чтобы создать пользователя в системе.

Request:

```
POST https://human-resources.com/users

{
    "first_name": "",
    "last_name" : "",
    ...
}
```

Response:

API создает нового пользователя и возвращает клиенту следующие ссылки:

- Ссылка для получения полного представления пользователя (self link) - GET.
- Ссылка для обновления пользователя - PUT.
- Ссылка для частичного обновления пользователя - PATCH.
- Ссылка для удаления пользователя - DELETE.

```
{
    HTTP/1.1 201 CREATED
    Content-Type: application/json
    ...

    "links": [
    {
        "href": "https://human-resources.com/users/123",
        "rel": "self",
    },
    {
        "href": "https://human-resources.com/users/123",
        "rel": "delete",
        "method": "DELETE"
    },
    {
        "href": "https://human-resources.com/users/123",
        "rel": "replace",
        "method": "PUT"
    },
    {
        "href": "https://human-resources.com/users/123",
        "rel": "edit",
        "method": "PATCH"
    }
    ]

}
```

... , ?

6. Соглашения об именовании

В этом разделе перечисляются соглашения об именах для URI, параметров запроса, ресурсов, полей. Важно придерживаться данного соглашения для всех создаваемых сервисах, особенно при использовании вместе с `HATEOAS` (так как предполагается автоматическое построение запросов).

6.1. Именование компонентов URI

Структура URI должна соответствовать спецификации RFC 3986, рекомендации в этом разделе определяют структуру и семантику URI в соответствии с данным стандартом.

6.1.1. Компоненты URI

Согласно RFC 3986, общий синтаксис URI состоит из иерархической последовательности компонентов, как показано в примере ниже.

```
https://example.com:8080/over/there?name=ferret#nose
\___/   \______________/\_________/\_________/\__/
  |            |            |        |        |
scheme      authority       path      query   fragment
```

6.1.2. Соглашения об именах URI

URI для REST-сервиса должен соответствовать следующему шаблону:

```
scheme://host:port/v{major-version}/namespace/resource?query
```

- URI должны начинаться с буквы и содержать только строчные буквы.
- Литералы URI должны быть разделены дефисом (-).
- Литералы в строках запроса должны быть разделены подчеркиванием (_).
- Коллекции ресурсов должны именоваться существительными во множественном числе (`/users, /accounts, /documents`).
- Отдельный ресурс в коллекции указывается непосредственно после коллекции (`/users/{user_id}`).
- Коллекции подресурсов указываются непосредственно под отдельным ресурсом (`/users/{user_id}/skills`).
- Индивидуальные ресурсы подресурсов могут существовать, но их следует избегать в пользу ресурсов верхнего уровня.

Примеры:

```
https://human-resources.com/v1/hr/candidats

https://human-resources.com/v1/users/123

https://human-resources.com/v1/users/123/skills
```

6.1.3. Именование ресурсов

При моделировании сервиса как набора ресурсов следует руководствоваться следующими правилами:

- Должны использоваться существительные, а не глаголы.
- Для коллекций существительные во множественном числе.
- Имена ресурсов должны быть в нижнем регистре и содержать только буквы, цифры и дефисы.
- Символ дефиса (-) должен использоваться в качестве разделителя слов в литералах URI.
  Следует обратить внимание, что это единственное место, где дефис используется как разделитель, в других ситуация для этого используется символ подчеркивания (_).

6.1.4. Именование параметров запросов

- Литералы/выражения в строках запроса должны быть разделены подчеркиванием (_).
- Параметры запроса должны быть percent-encoded.
- Параметры запроса должны начинаться с буквы и быть в нижнем регистре.
- Должны использоваться только буквы, цифры и символ подчеркивания (_).
- Параметры запроса должен быть необязательными.

6.1.5. Именование полей

- Модель представления данных должна соответствовать формату JSON. Значениями могут быть сами объекты, строки, числа, булевы значения и массивы.
- Имена полей должны быть словами в нижнем регистре, разделенными символом подчеркивания (_).
- Префиксы (такие, как is или has) не должны использоваться для полей типа boolean.
- Поля, содержащие коллекции, должны именоваться во множественном числе.

6.1.6. Именование перечислений

Значения перечислений должны состоять только из прописных букв, цифр и символа подчеркивания (`NOT_EQUAL, IN_PROGRESS`).

6.1.7. Именование файлов

JSON-схемы, использованные в API, должны храниться в отдельных файлах, имена которых состоят из слов, цифр и символов подчеркивания (`user_skill.json`).

7. URI

7.1. Путь к ресурсу

Как было указано ранее, путь к ресурсу содержит версию, пространство имен, имя ресурса и необязательные подресурсы.

Например:

`https://human-resources.com/v1/hr/candidats/123`

В таблице перечислены различные части пути ресурса указанного выше URI.

Часть пути	Определение	Комментарии
`v1`	Основная версия API	Основная версия API используется для различения двух несовместимых версий одного и того же API. Основная версия API - это целочисленное значение, которое должно быть частью URI.
`hr`	Пространство имен	Идентификаторы пространства имен используются для предоставления контекста и области действия для ресурсов. Они определяются логическими границами в модели бизнес-возможностей.
`candidats`	Имя ресурса	Если имя ресурса представляет собой коллекцию ресурсов, то метод `GET` для ресурса должен получить список ресурсов. Для указания критериев поиска могут использоваться параметры запроса.
123	Идентификатор ресурса	Чтобы извлечь конкретный ресурс из коллекции, идентификатор ресурса должен быть указан как часть URI.

7.1.1. Подресурсы

Подресурсы представляют собой отношение одного ресурса к другому. Если соотношение элементов 1:1 (один-к-одному), то дополнительная информация не требуется. В противном случае, подресурсу следует предоставлять уникальный идентификатор подресурса. Если количество элементов 1:* (один-ко-многим), то будут возвращены все подресурсы. Должно поддерживаться не более двух уровней подресурсов.

Пример	Описание
GET `https://human-resources.com/v1/hr/candidats/123/skills`	Возвращает все навыки кандидата 123.
GET `https://human-resources.com/v1/hr/candidats/123/skills/5`	Возвращает детали конкретного навыка для конкретного кандидата. В общем случае следует избегать таких вложенных запросов.
GET `https://human-resources.com/v1/hr/candidats/appointments`	Недопустимо возвращать множество множеств!
GET `https://human-resources.com/v1/hr/candidats/123/9080`	Недопустимо использовать два идентификатора ресурса один за другим.

7.1.2. Идентификаторы ресурса

Идентификаторы ресурса должны соответствовать следующим рекомендациям:

- Идентификатор ресурса должен принадлежать доменной модели ресурса, которая может гарантировать уникальную идентификацию отдельного ресурса.
- Нельзя использовать значения идентификаторов, полученных из базы данных (в особенности первичных ключей).

- Предпочтительный формат идентификатора - UUID или HMAC (например, `26c5eac67a9f059cb4ec3bc5b88b4d99`).
- Из соображений безопасности и целостности данных все идентификаторы подресурсов ДОЛЖНЫ быть ограничены только родительским ресурсом.
  Например, при запросе `https://human-resources.com/v1/hr/candidats/123/skills/5` сущность `skill` 5 не может быть получена, если даже она существует, но не связана с сущностью `candidate` 123.
- Значения перечисления могут использоваться как идентификаторы подресурсов, должно быть использовано строковое представление.
- Идентификаторы ресурса должны использовать либо символы UUID, либо символы ASCII. Идентификатор не должен содержать символы UTF-8.

7.2. Параметры запроса

Параметры запроса - это пары имя/значение, указанные после пути к ресурсу, как предписано в стандарте RFC 3986. Также следует помнить о Соглашении о именовании.

7.2.1. Фильтр для коллекции ресурсов

- Параметры запроса должны использоваться только в целях ограничения выборки ресурсов или в качестве критериев поиска или фильтрации.
- Идентификатор ресурса не должен использоваться для фильтрации.
- Параметры для постраничного вывода должны соответствовать требованиям из раздела (указать раздел!)
- Порядок сортировки по умолчанию должен рассматриваться как неопределенный.
- Если требуется задать явный порядок сортировки, то следует использовать параметр запроса sort со следующим синтаксисом: `{field_name}|{asc|desc},{field_name}|{asc|desc}`

7.2.2. Параметр запроса на единичном ресурсе

В типичных случаях, когда используется один ресурс (например, `/candidats/123`), параметры запроса использовать не следует.

7.2.3. Параметры запроса вместе с POST

Для большинства случаев не рекомендуется использовать с POST параметров запроса, предпочтительно использование тела запроса. В тех случаях, когда POST предоставляет постраничные результаты (как правило, в сложных API-интерфейсах поиска, где GET по какой-либо причине не подходит), параметры запроса могут использоваться для предоставления гипермедиа-ссылок на следующую страницу результатов.

7.2.4. Передача нескольких значений для одного параметра запроса

- При использовании параметров запроса для функции поиска часто необходимо передавать несколько значений. Например, это может быть случай, когда у ресурса может быть много состояний, таких как ОТКРЫТО, ЗАКРЫТО и НЕДОПУСТИМО. Что, если клиент API хочет найти все элементы с состоянием ЗАКРЫТО или НЕДОПУСТИМО?
  Рекомендуется использование повторение параметра запроса.
  Например: `?status=CLOSED&status=INVALID`
- URI имеют практические ограничения по длине (около 2000 символов), что может подтолкнуть разработчика вместо повторения параметра применять перечисление его значений.
  `:?status=CLOSED,INVALID`
  В общем случае, такое решение не рекомендуется.

8. JSON-схема

Для описания моделей запросов и ответов должна применяться спецификация OpenAPI версии 3.0.2.

9. JSON-типы

При разработке REST-сервисов следует по возможности использовать ряд стандартных типов, для которых в дальнейшем будет создан специальный реестр под управлением системы версионного контроля.

9.1. Примитивные типы

Все примитивные типы должны быть совместимы со спецификацией OpenAPI 3.0.2.

- **Строка**
  Как минимум, для строки нужно явно указывать параметры `minLength` и `maxLength`.
  Без максимальной длины невозможно надежно определить столбец базы данных для хранения заданной строки.
  Без максимума и минимума также невозможно предсказать, нарушит ли изменение длины обратную совместимость с существующими клиентами.
  Без минимальной длины клиенты могут отправлять пустую строку, даже если это недопустимо.
  Если формат строки известен (например, номер телефона), то рекомендуется использовать регулярное выражение.
- **Перечисление**
  Перечисление следует использовать только когда варианты строго фиксированы и не изменятся в будущем.
  В противном случае рекомендуется использовать строки.

- **Число**
  Может описывать целое число или число с число с фиксированной точкой.
  Следует быть осторожным с этим типом при десериализации, т.к. неизвестно, что именно в нем хранится: дробное или целое.
  Рекомендуется ограничивать диапазон чисел 32-битным знаковым.
  При использовании как целочисленного обязательно указывать `minimum` и `maximum`.

- **Массив**
  Спецификация определяет массив как неограниченный.
  Рекомендуется ограничивать размер массива (`maxItems`) 16-битным знаковым числом.
  Рекомендуется определить параметр `minItems`, описывающий, допустима ли пустая коллекция.

- **NULL**
  Не допускается использование типа NULL, т.к. это может вызвать сложности десериализации и является признаком плохого дизайна.

9.2. Общие типы

Представления ресурсов в API должны повторно использовать определения общих типов данных, где это возможно. В следующих разделах приведены подробности о некоторых из этих распространенных типов.

9.2.1. UUID

Добавить спецификацию

9.2.2. Имя

Рекомендуется использовать тип name.json для описания персональной информации об имени физического лица.

9.2.3. Телефон

Добавить спецификацию

9.2.4. Электронная почта

Добавить спецификацию

9.2.5. Адрес

Рекомендуется использовать тип address.json для описания адресной информации (страна, населенный пункт, улица, т.п.).

- Обратная совместимость с микроформатами адреса hCard.
- Прямая совместимость со стандартом проверки адресных полей Google i18n-api.
- Упрощает использование служб нормализации адресов (например, DaData.ru).

9.2.6. Деньги

Рекомендуется использовать тип money.json для представления денежных величин.

- Код валюты и значение должны быть указаны.
- Некоторые валюты, такие как JPY, не имеют субвалюты, что означает, что десятичная часть значения должна быть ".0"
- Значение не может быть отрицательным. Отрицательный или положительный - это внутреннее понятие, связанное с конкретным счетом и в отношении типа выполняемой транзакции, поэтому в API не отражается.

9.2.7. Процент

Рекомендуется использовать тип percentage.json для представления процентов.

- Тип гарантирует, что процент будет задан в виде числа с фиксированной точкой.
- Необходимо соблюдать все указанные в типе правила валидации.
- Поле `description` должно информировать, как именно работает представление процента. Например, в зависимости от страны, десятичным разделителем может быть запятая (,) или точка.

9.2.8. Интернационализация

Рекомендуется использовать следующие типы для представления страны, валюты, текущего языка:

- country_code.json
  Все коды стран должны соответствовать стандарту ISO 3166 в представлении alpha-2.

- currency_code.json
  Все коды валют должны соответствовать стандарту ISO 4217 в представлении alpha-3.

- language.json
  Все языки должны соответствовать спецификации bcp47.

- locale.json

- province.json
  Все данные по административной структуре стран должны соответствовать спецификации ISO 3166.

9.2.9. Дата, время и временные зоны

При работе с датой и временем все API должны соответствовать следующим правилам:

- Строка даты и времени должна соответствовать универсальному формату даты и времени, определенному в разделе 5.6 RFC3339.
- Все API должны в ответах указывать только время UTC.
- При обработке запросов API должен принимать поля даты и времени, содержащие смещение от UTC. Например, 2016-09-28T18: 30: 41.000 + 05: 00 будут приняты как эквивалентные 2016-09-28T13: 30: 41.000Z.
- Для передачи часового пояса рекомендуется использовать не смещение UTC, а отдельное поле/параметр в запросе.
- Формат часового пояса должен соответствовать стандарту IANA.
- Для представления плавающих значений времени (например, истечения срока действия договора, даты публикации) не следует задавать связь с часовым поясом.

9.2.9.1. Общие типы даты и времени

Следующие общие типы должны использоваться для выражения различных форматов даты и времени:

- date_time.json
  Должен соответствовать типу `date-time` стандарта RFC3339.

- date_no_time.json
  Должен соответствовать типу `full-date` стандарта RFC3339.
- time_nodate.json
  Должен соответствовать типу `full-time` стандарта RFC3339.

- date_year_month.json
  Следует использовать для выражения плавающей даты, которая содержит только месяц и год. Например, срок действия карты (2016-09).

- time_zone.json
  Следует использовать для выражения часового пояса `date-time` или `full-time`.

10. Обработка ошибок

Согласно спецификациям HTTP, результат выполнения запроса представлен в виде пары из целого числа и сообщения (кода состояния и сообщения). Коды состояния HTTP в диапазоне 4xx указывают на ошибки на стороне клиента (валидации или логики), в то время как коды в диапазоне 5xx указывают на ошибки на стороне сервера (как правило сигнализируют о неисправности). Однако эти коды состояния и понятное человеку сообщение о причине ошибки недостаточны для передачи полной информации об ошибке машиночитаемым способом. Чтобы

устранить последствия такой ошибки, клиенты API (особенно это касается других информационных систем, а не людей) нуждаются в более детальном представлении.

Следовательно, API должен возвращать представление ошибки в формате JSON, соответствующее схеме error.json, определенной в репозитории общих JSON-типов (то есть всегда доступный при проектировании API через спецификацию OpenAPI). Рекомендуется, чтобы с пространством имен, к которому принадлежит API, был связан *Каталог ошибок*.

10.1. Схема

Ответ с описанием ошибки error.json должен соответствовать спецификации RFC7807, описывающей подробные сведения об ошибках в ответе HTTP, чтобы избежать необходимости определять новые форматы ошибок. Согласно стандарту, при возвращении error используется специальный MIME-тип ответа `application/problem+json`.

Тип error должен содержать следующие поля:

Поле	Тип	Описание	Обязательный
`type`	`string`	URL-адрес документа, описывающий условие ошибки (если описания нет, то возвращается «about: blank»).	Да
`title`	`string`	Короткий, понятный человеку заголовок для общего типа ошибки.	Да
`status`	`number`	Код состояния HTTP, он здесь для того, чтобы вся информация находилась в одном месте, а также для исправления возможного влияния прокси-серверов.	Да
`detail`	`string`	Развернутое, понятное человеку описание конкретной проблемы. Описание должно быть ориентировано на помощь клиенту, а не на раскрытие подробностей внутреннего устройства API.	Нет
`instance`	`string`	Идентификатор с уникальным URI конкретной ошибки, может указывать на ошибку в журнале.	Нет
`links`	`array`	Ссылки HATEOAS, относящиеся к сценарию ошибки и используемые для предоставления дополнительной информации об ошибке и способах ее устранения.	Нет

Рекомендуется расширить `error.json` дополнительными полями, которые дадут клиенту больше информации об ошибке.

Например, в случае валидационной ошибки, ответ может быть таким:

```
{

    "type": "https://human-resources.com/problems/validation-error",
    "title": "Your request parameters didn't validate.",

    "status": 400,

    "invalid-params": [
      {
        "name": "age",
        "reason": "must be a positive integer"
      },
      {
        "name": "color",
        "reason": "must be 'green', 'red' or 'blue'"
      }
    ]
}
```

Для случаев специфических ошибок, таких как валидация, рекомендуется создавать новые типы, расширяя базовый тип `error` дополнительными полями и помещать эти типы в *Каталог ошибок*. Клиенты, не совместимые с данным типом ошибки, проигнорируют дополнительные поля и обработают ошибку стандартным образом. Перед регистрацией типа его необходимо соответствующим образом зарегистрировать:

1. `type`: URI (обычно, как схему "http" или "https")
2. `title` : краткое описание проблемы
3. `status`: HTTP-код статуса ошибки

Например:

```
{
    "type": "https://human-resources.com/problems/request-parameters-missing",
```

```
    "title": "required parameters are missing",
    "detail": "The parameters: limit, date were not provided",
    "limit_parameter_format": "number",
    "date_parameter_format": "YYYY-mm-ddThh:mm-ss"
}
```

В итоге тип error, найденный по заданному URI, будет идентифицировать конкретную ошибку, и может быть использован повторно в других частях API, а так же служить основой для создания новых типов ошибок.

10.2. Примеры

В следующем примере показана ошибка проверки типа `validation-error` в нескольких полях. Поскольку это ошибка клиента, должен быть возвращен код состояния `HTTP 400 Bad Request`. Дополнительное поле `invalid-params` типа array содержит список полей, не прошедших валидацию.

```
{
        "type": "https://human-resources.com/problems/validation-error",
        "title": "Your request parameters didn't validate.",
        "status": 400,
        "invalid-params": [
          {
            "name": "/user/first_name",
            "reason": "Required field is missing",
            "location": "body"
          },
          {
            "name": "/candidate/salary",
            "value": "GBP",
            "reason": "Currency code is invalid",
            "location": "body"
          }
        ]
}
```

Данный пример показывает, как обрабатываются композитный ошибки. Объект `error` содержит коллекцию `errors`, каждые экземпляр вложенной ошибки представлен как элемент в этой коллекции. Поскольку это ошибки проверки клиента, должен быть возвращен код состояния `HTTP 400 Bad Request`.

```
{
    "type": "https://human-resources.com/problems/validation-error",
    "title": "Your request parameters didn't validate.",
    "status": 400,
    "errors": [
      {
        "type": "https://human-resources.com/problems/validation-error",
        "title": "Your request parameters didn't validate.",
        "invalid-params": [
          {
            "name": "/user/1/first_name",
            "reason": "Required field is missing",
            "location": "body"
          }
        ]
      },
      {
        "type": "https://human-resources.com/problems/validation-error",
        "title": "Your request parameters didn't validate.",
        "invalid-params": [
          {
            "name": "/user/2/age",
            "reason": "Required field is missing",
            "location": "body"
          }
        ]
      }
```

```
    ]
  }
```

В тех случаях, когда входные данные клиента правильно сформированы и действительны, но для выполнения запроса может потребоваться взаимодействие с API или процессами вне этого URI, должен быть возвращен код состояния HTTP `422 Unprocessable Entity`. В данном примере ошибка бизнес-логики возникает в другом сервисе, проверяющем запрошенную кандидатом зарплату.

```
{
    "type": "https://human-resources.com/problems/salary-error",
    "title": "Required salary too big.",
    "status": 422,
    "name": "/candidate/salary"
}
```

10.3. Каталог ошибок

Весьма важно создать централизованный реестр ошибок для различных специализированных сценариев, таких, как валидация, асинхронные операции, параллелизм и прочее. Каждая из таких ошибок (являющаяся расширением базового типа error.json) содержит специфические поля, максимально выражающие потребности того сценария, ради которого и создавались.

Фактически, каталог ошибок - это отдельный JSON-файл, содержащий коллекцию спецификация ошибок, в котором их можно найти по некоторому пространству имен, заданному полем `type`. Каждая спецификация ошибки должна содержать уникальное имя ошибки, описание проблемы и перечень связанных с ней ссылок. Если API поддерживает интернационализацию, то спецификация ошибки должна поддерживать несколько локалей.

Следует привести дополнительное обоснование для ведения каталога ошибок.

- Вывод жестко запрограммированных строк сообщений об ошибках.
  Разработчики не обязательно хорошо пишут сообщения об ошибках. Часто сообщения об ошибках, написанные разработчиком, делают много предположений о контексте и аудитории. Трудно изменить такие сообщения об ошибках, если они встроены в код реализации.

- Локализация сообщений об ошибках.
  Поддержка локализации в коде неудобна и трудозатратна, в отличие от ведения централизованного реестра, которым управляют технические писатели, а не программисты.

- Синхронизация API с документацией.
  Клиенты должны иметь возможность с уверенностью ссылаться на документацию API, посвященную ошибкам времени выполнения, что помогает значительно снизить затраты на поддержку.

Выработать и согласовать формат каталога ошибок.

11. Версионирование

Описание принципов версионирования API и управления его жизненным циклом, в том числе политикой устаревания и вывода из эксплуатации.

11.1. Жизненный цикл API

Пример состояний типичного жизненного цикла:

Состояние	Описание
`PLANNED`	Запланирована разработка API, составлен план выпуска.
`BETA`	API работает и доступен с целью тестирования, проверки и развертывания для новых клиентов.
`LIVE`	API работает и доступен для новых клиентов. Версия API полностью поддерживается.
`DEPRECATED`	API работает и доступен для существующих клиентов. Версия API полностью поддерживается, включая исправления ошибок обратной совместимости. Версия API недоступна для новых клиентов.
`RETIRED`	API больше не доступен для любых клиентов. Все развернутые приложения, находящиеся в этом состоянии, должны быть полностью удалены из рабочей среды.

11.2. Политика версионирования

API всегда обладают определенной версией и должны придерживаться следующих принципов версионирования:

1. Спецификации API должны следовать специальной схеме управления версиями, где (v) представляет версию, основным является порядковый номер, начинающийся с 1 для первого выпуска LIVE, а второстепенным является порядковый номер, начинающийся с 0 для первого вспомогательного выпуска любого основного выпуска.
2. Каждый раз, когда происходит постепенное изменение API, независимо от того, является ли оно обратно совместимым, спецификация API должна быть версионированной. Это позволяет помечать, документировать, проверять, обсуждать, публиковать и распространять изменения.
3. Конечные точки API должны отражать только основную версию.
4. Версии спецификации API отражают изменения интерфейса и могут быть отделены от схем управления версиями реализации сервиса.
5. Дополнительная версия API должна поддерживать обратную совместимость со всеми предыдущими дополнительными версиями в рамках одной основной версии.
6. Основная версия API может поддерживать обратную совместимость с предыдущей основной версией.

Для данного набора функций должна быть только одна версия API в состоянии `LIVE` в любой момент времени для всех основных и вспомогательных версий. Это гарантирует, что подписчики всегда понимают, какой версионированный API они должны использовать. Например, v1.2 `RETIRED`, v1.3 `DEPRECATED` или v2.0 `LIVE`.

11.3. Обратная совместимость

API должны разрабатываться расширяемыми, чтобы поддерживать совместимость и избегать дублирования ресурсов, функциональности и чрезмерного управления версиями.

API должны придерживаться следующих принципов, чтобы считаться обратно совместимыми:

- Все изменения должны быть аддитивными (т.е. только добавляться, но не удаляться или изменяться).
- Все изменения должны быть необязательными.
- Семантика не должна изменяться.
- Параметры запроса и тела запроса должны быть неупорядоченными.
- Дополнительная функциональность для существующего ресурса должна быть реализована:
    - Как необязательное расширение.
    - Как операция над новым дочерним ресурсом.
    - Изменяя тело запроса, при этом поддерживая все предыдущие варианты запроса.

11.3.1. Неполный список правил, обеспечивающих обратную совместимость

- URI ресурса может поддерживать дополнительные параметры запроса, но они не могут быть обязательными.
- Не должно быть никаких изменений в поведении API для URI запроса без вновь добавленных параметров запроса.
- Новый параметр с обязательным ограничением не должен добавляться в запрос.
- Семантика существующего параметра, всего представления или ресурса не должна изменяться.
- Сервис должен распознавать ранее действительное значение параметра и не должен выдавать ошибку при их использовании.
- Не должно быть никаких изменений в кодах состояния HTTP, возвращаемых URI.
- Не должно быть никаких изменений в HTTP-методах, поддерживаемых ранее URI, при этом URI может поддерживать новый HTTP-метод.
- Не должно быть никаких изменений в имени и типе заголовка запроса или ответа URI, дополнительные заголовки могут быть добавлены, если они не являются обязательными.
- Существующее поле в объекте JSON ответа API должно продолжать возвращаться с тем же именем и типом JSON (число, целое число, строка, массив, объект).
- Если значение поля ответа является массивом, то тип содержимого в массиве не должен изменяться.
- Если значение поля ответа является объектом, тогда политика совместимости должна применяться к объекту JSON в целом.
- Новые свойства могут быть добавлены к представлению в любое время, но они не должны изменять значение существующего свойства.
- Добавленные новые свойства не должны быть обязательными.
- Обязательные поля обязательно присутствуют в ответе.
- Для примитивных типов, если нет ограничений, описанных в документации API (например, длина строки, возможные значения для типа ENUM), клиенты не должны предполагать, что значения каким-либо образом ограничены.
- В типах enum не должно быть никаких изменений в уже поддерживаемых перечисляемых значениях.

11.4. Политика вывода из обслуживания

Политика *вывода из обслуживания* регулирует, как версии API переходят из LIVE в состояние RETIRED, и предназначена для обеспечения согласованной и разумной миграции для клиентов API, которым необходимо перейти от старой версии API к новой, и в то же время обеспечить непрерывную работу уже существующих систем.

Дополнить в следующей версии документа.

---

12. Паттерны REST

В этом разделе будут перечислены различные полезные шаблоны проектирования RETS-сервисов, обеспечивающие полную совместимость с данным руководством и другими аналогичными руководствами и спецификациями. Рекомендуется при разработке новых сервисов следовать данным шаблонам.

Нужно в дальнейшем в каждый подраздел поместить ссылку на референсную имплементацию паттерна.

12.1. Создание ресурса

Для создания нового ресурса используется метод `POST`. Тело запроса для `POST` может несколько отличаться от запроса/ответа `GET` или `PUT` (обычно меньше полей, так как некоторые значения генерирует сервер). В большинстве случаев сервис возвращает уникальный идентификатор ресурса. В тех случаях, когда идентификатор предоставляется клиентом API, следует использовать "*Создание ресурса с внешним идентификатором*".

После успешного выполнения метода `POST` будет создан новый ресурс. В том случае, если сервис поддерживает `HATEOAS`, метод `POST` должен вернуть в стандартном ответе ссылку `rel:self`, указывающую на только что созданный ресурс. Рекомендуется также вернуть (в соответствии с принципами `HATEOAS`) ссылки на все другие доступные для ресурса операции. Если же `HATEOAS` не поддерживается, то представление ресурса возвращается без ссылок, а вместо этого указывается заголовок `Location`, содержащий путь к созданному ресурсу.

Request:

```
POST /v1/users

{
    "first_name": "Vassily",
    "last_name": "Pupkin",
    "phone": "322223223322",
    "post": "programmer",
    "department": "GPN"
}
```

Response (с поддержкой `HATEOAS`):

```
HTTP/1.1 201 Created
Content-Type: application/json


{
  "id": "E19612EW4364010KGFNJQI",
  "state": "new",
  "created_on": "2020-01-01T01:11:11Z",
  "first_name": "Vassily",
  "last_name": "Pupkin",
  "phone": "322223223322",
  "post": "programmer",
  "department": "GPN"
  "links": [
    {
      "href": "https://human-resources.com/v1/candidates/E19612EW4364010KGFNJQI",
      "rel": "self",
      "method": "GET"
    },
    {
      "href": "https://human-resources.com/v1/candidates/E19612EW4364010KGFNJQI",
      "rel": "delete",
      "method": "DELETE"
    }
  ]
}
```

:

Response (без поддержки HATEOAS):

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: https://human-resources.com/v1/candidates/E19612EW4364010KGFNJQI

{
  "id": "E19612EW4364010KGFNJQI",
  "state": "new",
  "created_on": "2020-01-01T01:11:11Z",
  "first_name": "Vassily",
  "last_name": "Pupkin",
  "phone": "322223223322",
  "post": "programmer",
  "department": "GPN"
}
```

12.1.1. Создание ресурса с внешним идентификатором

Когда клиент API сам предоставляет идентификатор ресурса, следует использовать метод PUT, поскольку операция является идемпотентной даже во время создания (за исключением служебных полей типа "`created_on`", которые семантически не важны. В таком случае здесь нужно все реализовать точно так же, как и с методом POST: код `201 Created` и в теле ответа возвращается созданный ресурс.

Если же переданный клиентом идентификатор уже существует (а значит, существует и объект с этим идентификатором), то метод GET просто обновляет этот ресурс, кроме того, он должен вернуть код `204 No Content` и без тела ответа.

12.2. Коллекции ресурсов

Коллекция должна возвращать список представлений всех ее ресурсов, включая связанные с ресурсами метаданные.

Метод GET, используемый для чтения коллекций, не должен никак влиять на внутренне состояние сервиса и как либо менять сами коллекции и ресурсы, то есть он должен быть идемпотентным.

Предполагается, что список ресурсов фильтруется на основании некоторых прав и привилегий, доступных клиенту API, а значит, что это клиент должен получить не список всех существующих в домене и коллекции ресурсов, а только лишь те, на которые у клиента имеются разрешения на чтение в текущем контексте. Сокращение объема выборки может также уменьшить нагрузку на сеть, что особенно важно для поддержания способности системы к горизонтальному масштабированию.

В случае, когда сервис предоставляет возможность частичного чтения коллекций, должны поддерживаться следующие паттерны:

12.2.1. Фильтрация по времени

Для выбора набора элементов могут использоваться временной диапазон.

Параметры запроса:

- `start_time` или `{property_name}_after`:
  Отметка времени (`timestamp`) в формате ISO-8601, указывающая начало временного диапазона. Если есть только один параметр, обозначающий время, то он называется `start_time` , в противном случае именование на основе паттерна `{property_name}_after` (например, `created_after, updated_after`).
- `end_time` или `{property_name}_before`:
  Отметка времени (`timestamp`) в формате ISO-8601, указывающая конец временного диапазона, аналогично предыдущему пункту.

12.2.2. Сортировка

Результаты запроса могут возвращаться отсортированными в соответствии с заданными клиентом параметрами: сортировка по значению определенного поля и порядок сортировки.

Параметры запроса:

- `sort_by`:
  Размерность, по которой должны быть упорядочены элементы. Размерность должна быть полем элемента, а значение по умолчанию для него должно быть отражено в документации.

- `sort_order`:
  Значение порядка сортировки (`asc` или `desc`), указывающее на возрастающий или убывающий порядок.

Рекомендуется определять порядок сортировки по умолчанию.

12.2.3. Разбиение на страницы

Любой ресурс, который может возвратить большой, потенциально неограниченный список ресурсов в своем ответе GET, должен реализовать разбиение на страницы в соответствии с описанным ниже паттерном.

Пример URI запроса: `/users?page_size={page_size}&page={page}`

При постраничном выводе рекомендуется применять предварительную сортировку элементов по какому-либо неизменяемому или редко изменяемому полю полю.

Параметры запроса:

- `page_size`:
  Неотрицательное, ненулевое целое число, указывающее максимальное количество результатов, возвращаемых за один раз.
  - Параметр должен быть необязательным.
  - Параметр должен иметь значение по умолчанию, если клиент не задал его самостоятельно.
  - Значение параметра, заданного клиентом, должно быть меньше максимально разрешенного значения для этого параметра, такое максимальное значение должно быть жестко задано как часть реализации сервиса или как параметр конфигурации.
- `page`:
  Ненулевое целое число, представляющее страницу результатов.
  - Параметр должен быть необязательным.
  - Параметр должен иметь значение по умолчанию 1 (соответственно, нумерация страниц с 1, привет программистам).
  - При передаче клиентом неправильного значения параметра (меньше 1) следует отвечать кодом `400 Bad Requst`.
  - При передаче клиентом неправильного значения параметра (больше допустимого значения) следует отвечать кодом `200 OK` и пустым списком результатов.
- `total_required`:
  Логическое значение, указывающее, что сервис должен вернуть в ответе общее количество элементов (`total_items`) и страниц (`total_pages`).
  - Параметр должен быть необязательным.
  - Параметр должен иметь значение по умолчанию `false`.
  - Параметр может использоваться клиентом при первом запросе, чтобы запомнить общее количество элементов и страниц для оптимизации дальнейших запросов.

Ответом на запрос должен быть JSON-объект, содержащий следующие свойства:

Поля ответа:

- `items`:
  Должен быть массивом, содержащим текущую страницу списка результатов.

- `total_items`:
  Следует использовать для указания общего количества элементов в полном списке результатов, а не только на этой странице.
  - Если реализован параметр `total_required`, то значение следует возвращать, только если `total_required` имеет значение `true`.
  - Если `total_required` не был реализован, то значение может быть возвращено в каждом ответе (если это имеет смысл и не сказывается на производительности).
  - Параметр должен быть неотрицательным целым числом.
  - Клиенты не должны считать, что значение `total_items` является постоянным, потому что его значение может изменяться между запросами.
- `total_pages`:
  Следует использовать для указания количества доступных страниц.
  - Если `total_required` был реализован, то значение следует возвращать, только если `total_required` имеет значение `true`.
  - Если `total_required` не был реализован, то значение может быть возвращено в каждом ответе (если это имеет смысл и не сказывается на производительности).
  - Параметр должен быть неотрицательным, ненулевым целым числом.
  - Клиенты должны предполагать, что значение `total_pages` является постоянным, потому что его значение может изменяться между запросами.

Если API поддерживает `HATEOAS`, то дополнительно в ответе должно присутствовать поле `links`, содержащее массив ссылок, организованных для навигации между страницами.

Если поддержки HATEOAS нет, то подобные навигационные ссылки добавляются в заголовок `Link`. Например,

```
Link: <https://human-resources.com/v1/candidates?page=3&per_page=100>; rel="next", <https://human-resources.com/v1/candidates?page=50&per_page=100>; rel="last"
```

В любом случае, навигация между страницами организована следующими ссылками:

Ссылка	Описание
`self`	Ссылка на текущую страницу списка результатов.
`first`	Ссылка на первую страницу списка результатов.
`last`	Ссылка на последнюю страницу списка результатов. Ссылка не является обязательной, ее нужно возвращать только в том случае, если в запросе задан параметр `total_required`.
`next`	Ссылка на следующую страницу списка результатов.
`prev`	Ссылка на предыдущую страницу списка результатов.

Пример ответа с результатами, разбитыми на страницы:

Request:

```
GET /v1/users?page_size=20&page=1&total_required=true
```

Response (с поддержкой HATEOAS):

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "total_items": "166",
    "total_pages": "83",
    "plans": [
      {
        "id": "6EM196669U062173D7QCVDRA",
        "state": "ACTIVE",
        "first_name": "Vassily",
        "create_time": "2019-08-22T04:41:52.836Z",
        "links": [
          {
            "href": "https://human-resources.com/v1/users/6EM196669U062173D7QCVDRA",
            "rel": "self"
          }
        ]
      },
      {
        "id": "83567698LH138572V7QCVZJY",
        "state": "ACTIVE",
        "name": "Petia",
        "create_time": "2019-08-22T04:41:55.623Z",
        "links": [
          {
            "href": "https://human-resources.com/v1/users/83567698LH138572V7QCVZJY",
            "rel": "self"
          }
        ]
      }
    ],
    "links": [
      {
        "href": "https://human-resources.com/v1/users?page_size=2&page=3",
```

```
        "rel": "self"
      },
      {
        "href": "https://human-resources.com/v1/users?page_size=2&page=1",
        "rel": "first"
      },
      {
        "href": "https://human-resources.com/v1/users?page_size=2&page=2",
        "rel": "prev"
      },
      {
        "href": "https://human-resources.com/v1/users?page_size=2&page=4",
        "rel": "next"
      },
      {
        "href": "https://human-resources.com/v1/users?page_size=2&page=82",
        "rel": "last"
      }
    ]
}
```

Response (без поддержки HATEOAS):

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://human-resources.com/v1/users?page_size=2&page=3>; rel="self",
      <https://human-resources.com/v1/users?page_size=2&page=1>; rel="first",
      <https://human-resources.com/v1/users?page_size=2&page=2>; rel="prev",
      <https://human-resources.com/v1/users?page_size=2&page=4>; rel="next",
      <https://human-resources.com/v1/users?page_size=2&page=82>; rel="last"

{
    "total_items": "166",
    "total_pages": "83",
    "plans": [
      {
        "id": "6EM196669U062173D7QCVDRA",
        "state": "ACTIVE",
        "first_name": "Vassily",
        "create_time": "2019-08-22T04:41:52.836Z",
        "links": [
          {
            "href": "https://human-resources.com/v1/users/6EM196669U062173D7QCVDRA",
            "rel": "self"
          }
        ]
      },
      {
        "id": "83567698LH138572V7QCVZJY",
        "state": "ACTIVE",
        "name": "Petia",
        "create_time": "2019-08-22T04:41:55.623Z",
        "links": [
          {
            "href": "https://human-resources.com/v1/users/83567698LH138572V7QCVZJY",
            "rel": "self"
          }
        ]
      }
    ]
}
```

12.3. Чтение единичного ресурса

Один ресурс обычно получается из родительской коллекции ресурсов, часто более детально, чем элемент в представлении целой коллекции. То есть нормально, когда представления одного и того же ресурса различаются для разных запросов.

Очевидно, что выполнение метода GET не должно влиять на состояние системы, то есть запрос должен быть идемпотентным.

Все идентификаторы ресурсов предпочтительно должны быть непоследовательными и не числовыми. В сценариях, где эти данные могут использоваться в качестве подчиненных другим данным, удобнее использовать неизменяемые строковые идентификаторы (т.е. `NAME_OF_VALUE` против `1421321`).

Если заданный клиентом идентификатор ресурса не существует, то должен быть возвращен код `404 Not Found`, в случае успеха - `200 OK`.

Шаблон запроса:

```
GET /{version}/{namespace}/{resource}/{resource-id}
```

Request:

```
GET /v1/staff/employees/F66W27667YB813414MKQ4AKDY
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "first_name": "Vassily",
    "last_name": "Pupkin",
    "email": "v.pupkin@gpn.ru",
    "department": "DIT"
}
```

12.4. Удаление единичного ресурса

Чтобы разрешить повторные попытки (например, при плохой связи), метод `DELETE` рассматривается как идемпотентный, поэтому он всегда должен отвечать кодом `204 No Content`. Код `404 Not Found` использоваться не должен, так как при повторной попытке клиент может ошибочно подумать, что ресурс вообще никогда не существовал. Метод `GET` может быть использован для проверки наличия ресурсов до удаления.

По ряду причин некоторые данные, представленные в качестве ресурса, могут исчезнуть: например, потому что они были удалены, потому что срок их действия истек, из-за применения бизнес-правил и т.п. Сторонние сервисы теоретически могут вернуть код `410 Gone` при сквозном запросе, который внешний ресурс больше не существует. Тем не менее, из создаваемых сервисов в подобных ситуациях рекомендуется возвращать `204 No Content`.

Шаблон запроса:

```
DELETE /{version}/{namespace}/{resource}/{resource-id}
```

Request:

```
DELETE /v1/staff/employees/F66W27667YB813414MKQ4AKDY
```

Response:

```
HTTP/1.1 204 No Content
```

12.5. Обновление единичного ресурса (полное)

Для выполнения обновления всего ресурса должен использоваться метод `PUT`. Представление для обновления ресурса (для метода `PUT`) должно быть идентичным представлению ресурса, возвращаемого методом `GET`. Стоит заметить, что `GET`-представление ресурса может содержать поля "только для чтения" (например, `create_time`), при использовании его с PUT следует игнорировать такие поля на стороне сервиса.

Если обновление выполнено успешно, то возвращается код `204 No Content` (без тела ответа). В некоторых редких случаях (как правило, для оптимизации взаимодействия с клиентом) может использоваться код `200 OK` вместе с телом ответа, однако в целом такого решения стоит избегать. При ошибках валидации возвращается код `400 Bad Request` и стандартный объект `error`. Если ошибка связана со взаимодействием с внешними сервисами, вместо `400 Bad Request` возвращается код `422 Unprocessable Entity`.

Шаблон запроса:

```
PUT /{version}/{namespace}/{resource}/{resource-id}
```

Request:

```
PUT /v1/staff/employees/F66W27667YB813414MKQ4AKDY

{
    "id": "F66W27667YB813414MKQ4AKDY",
    "state": "INACTIVE",
    "first_name": "Fedor",
    "create_time": "2019-08-22T04:41:52.836Z"
}
```

Поля `state` `first_name` были изменены, поля `id` и `create_time` должны быть проигнорированы.

Response:

```
HTTP/1.1 204 No Content
```

12.6. Обновление единичного ресурса (частичное)

В некоторых случаях ресурсы требуют мелкогранулярного обновления, то есть на уровне отдельных полей, а не целиком. В таких случаях API должен предоставлять метод PATCH с реализацией, совместимой со стандартом RFC 6902 JSON Patch.

JSON-patch описывает последовательность операций для применения к целевому JSON-документу. Операции, определенные в спецификации JSON Patch - добавление, удаление, замена, замещение, копирование и проверка. Для поддержки частичного обновления ресурсов API должны поддерживать операции добавления, удаления и замены. Поддержка других операций (замещение, копирование и проверка) оставляется на усмотрение разработчика.

Доработать в следующей версии документа.

12.7. Проекции

На метод `GET` API обычно отвечает полным представлением ресурса. Однако из соображения оптимизации трафика клиент может попросить сервис вернуть *частичное* представление, используя HTTP-заголовок `Prefer`. Определение того, что представляет собой «*минимальный*» ответ, определяется разработчиками сервиса.

Чтобы запросить частичное представление с конкретными полями, клиент может использовать параметр запроса `fields`. Для выбора нескольких полей должен использоваться разделенный запятыми список полей.

Request:

```
GET /v1/staff/employees/F66W27667YB813414MKQ4AKDY?fields=first_name

Prefer: return=minimal
```

Response:

```
HTTP/1.1 200 OK

{
    "id": "6EM196669U062173D7QCVDRA",
    "first_name": "Vassily"
}
```

Ответ содержит только выбранные поля. Следует обратить внимание, что идентификатор ресурса так же возвращается, хотя в списке полей его не было.

12.8. Коллекции подресурсов

Иногда для идентификации ресурса требуется несколько идентификаторов («составные ключи»). В подобных сценариях ресурс реализован как подчиненный другому ресурсу. В целом, следует избегать использования подресурсов, или, во всяком случае, ограничить их вложенность двумя уровнями (то есть `/resource/{resource-id}/sub-resource/{sub-resource-id}`). Сервис должен проверять каждый уровень идентификаторов, чтобы убедиться, что ресурсы правильно связаны друг с другом и к ним есть доступ.

Шаблоны запроса:

```
POST /{version}/{namespace}/{resource}/{resource-id}/{sub-resource}
GET /{version}/{namespace}/{resource}/{resource-id}/{sub-resource}
GET /{version}/{namespace}/{resource}/{resource-id}/{sub-resource}/{sub-resource-id}
PUT /{version}/{namespace}/{resource}/{resource-id}/{sub-resource}/{sub-resource-id}
DELETE /{version}/{namespace}/{resource}/{resource-id}/{sub-resource}/{sub-resource-id}
```

12.9. Единичный подресурс

Когда подресурс имеет отношение один к одному (1:1) с родительским ресурсом, он может быть смоделирован как единичный подресурс (singleton).

Для единичного подресурса имя должно быть существительным в единственном числе. Разумеется, этот ресурс должен всегда существовать (то есть никогда не возвращать код `404`).
Очевидно также, что подресурс должен принадлежать родительскому ресурсу, в противном случае этот подресурс следует повысить до верхнеуровнего ресурса.

Шаблон запроса:

```
GET/PUT /{version}/{namespace}/{resource}/{resource-id}/{sub-resource}
```

Пример:

```
GET /v1/staff/employees/F66W27667YB813414MKQ4AKDY/labor_contract
```

12.10. Идемпотентность

Идемпотентность является важным аспектом построения отказоустойчивого API. Идемпотентный API позволяют клиентам безопасно повторить операцию, не беспокоясь о побочных эффектах, которые может вызвать эта операция. Например, клиент может безопасно повторить идемпотентный запрос в случае сбоя запроса из-за ошибки сетевого подключения.

Согласно спецификации HTTP, метод является идемпотентным, если побочные эффекты более чем одного идентичного запроса совпадают с побочными эффектами для одного запроса. Методы `GET, HEAD, PUT` и `DELETE` (ну и дополнительно, `TRACE` и `OPTIONS`, хотя они в данной инструкции не описаны) являются, очевидно, идемпотентными.

Операции `POST` по определению не являются ни безопасными, ни идемпотентными.

Все реализации сервиса должны гарантировать, что безопасное и идемпотентное поведение методов HTTP реализовано в соответствии со спецификациями HTTP. Сервисы, которые требуют идемпотентности для операций `POST`, должны быть реализованы в соответствии со следующими рекомендациями.

12.10.1. Идемпотентные POST-методы

Операция POST по определению не являются идемпотентной, а это означает, что выполнение POST более одного раза с одним и тем же входными параметрами создает несколько ресурсов. Чтобы избежать создания клонов, API должен реализовывать паттерн, определенный ниже, это гарантирует, что для одного и того же набора входных параметров создается только один ресурс. В общем же случае обеспечение идемпотентности POST должна быть обоснована, так как по умолчанию клиент ожидает от метода неидемпотентного поведения.

Для сценариев, требующих идемпотентности для запросов POST, создание дублированной записи является серьезной проблемой. Например, клоны записей для сценариев, которые создают или выполняют платеж, по определению не допускаются.

Для отслеживания идемпотентного запроса в каждом запросе используется уникальный , определенный в заголовке запроса.

При первом запросе к API клиент создает новый POST-запрос с заголовком `X-Request-Id`, содержащим . Следует отметить, что заголовок может одновременно использоваться и для других целей, например, для трассировки запросов от API Gateway.

Request:

```
POST v1/users
Content-Type: application/json
X-Request-Id: 123e4567-e89b-12d3-a456-426655440000

{
    "state": "ACTIVE",
    "first_name": "Vassily",
    "last_name": "Pupkin"
}
```

При успешном выполнении запроса API вернет код `201 Created`.
В примере использован HATEOAS, без его поддержки ссылка будет указана в заголовке `Location` (см. паттерн POST).

Response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
X-Request-Id: 123e4567-e89b-12d3-a456-426655440000


{
    "id": "CDZEC5MJ8R5HY",
    "links": [{
        "href": "https://human-resources.com/v1/users/CDZEC5MJ8R5HY",
        "rel": "self",
        "method": "GET"
    }]
}
```

Теперь выполняется повторный вызов POST с теми же аргументами.

Request:

```
POST v1/users
Content-Type: application/json
X-Request-Id: 123e4567-e89b-12d3-a456-426655440000

{
    "state": "ACTIVE",
    "first_name": "Vassily",
    "last_name": "Pupkin"
}
```

Сервис, проверив, что вызов идентичен первому, должен вернуть код `200 OK` и ссылку на уже существующий ресурс, демонстрируя, что запрос уже был успешно обработан. Опять же, если не поддерживается HATEOAS, то ссылка на результат будет в заголовке `Location`.

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
X-Request-Id: 123e4567-e89b-12d3-a456-426655440000
{
    "id": "CDZEC5MJ8R5HY",
    "state": "new",
    "created_on": "2020-01-01T01:11:11Z",
    "first_name": "Vassily",
    "last_name": "Pupkin",
    "links": [
      {
        "href": "https://human-resources.com/v1/users/CDZEC5MJ8R5HY",
        "rel": "self",
        "method": "GET"
      },
      {
        "href": "https://human-resources.com/v1/users/CDZEC5MJ8R5HY",
        "rel": "delete",
        "method": "DELETE"
      }
    ]
}
```

12.10.2. Уникальность ключа идемпотентности

Ключ идемпотентности, передаваемый как часть каждого POST-запроса, должен быть, очевидно, уникальным и не может использоваться повторно с другим запросом с другими входными параметрами.

В качестве ключа идемпотентности рекомендуется использовать UUID или аналогичный случайный идентификатор. Также рекомендуется, чтобы сервис обрабатывал ключи идемпотентности с учетом времени и, таким образом, мог очищать или удалять ключ по истечении срока его действия.

12.10.3. Правила обработки сбоев при работе с ключами идемпотентности

- Если для идемпотентного запроса не задан заголовок `X-Request-Id`, сервис должен ответить кодом `400 Bad Request` со ссылкой, указывающей на общедоступную документацию по этому шаблону.
- Если есть попытка повторно использовать с другими входными параметрами запроса, сервис должен ответить кодом `422 Unprocessable Entity` со ссылкой, указывающей на общедоступную документацию об этом шаблоне.
- В случае других ошибок сервис должен отреагировать стандартно.

12.11. Асинхронные операции

Для некоторых типов операций (долго выполняющихся) может потребоваться асинхронная обработка запроса. Чтобы избежать длительных задержек на стороне клиента и предотвратить длительные открытые клиентские соединения, ожидающие выполнения операций, рекомендуется воспользоваться следующим паттерном.

12.11.1. POST-методы

- Метод `POST` возвращает код `202 Accepted` вместо `201 Created`.

Тело ответа должно включать (в случае поддержки `HATEOAS`) несколько ссылок:

- URI ресурса, по которому его можно прочитать, если его идентификатор уже известен. В результате клиент может сделать `GET`-запрос по этому URI для получения полностью готового ресурса. Пока ресурс не готов, такой запрос должен возвращать код `404 Not Found`.
  Шаблон: `{ "rel": "self", "href": "/v1/namespace/resources/{resource_id}", "method": "GET" }`

- URI очереди запросов, где может быть получен статус операции через некоторый временный идентификатор. Клиенты должны сделать `GET`-запрос, чтобы получить статус операции, который может включать такую информацию, как состояние завершения, прогресс выполнения операции и окончательный URI после ее завершения. Реализация должны быть максимально оптимизированной в плане производительности, так как проверка статуса операции клиентом может быть довольно частой.
  Шаблон: `{ "rel": "self", "href": "/v1/queue/requests/{request_id}, "method": "GET" }"`

Если HATEOAS не поддерживается, то параметры передаются в заголовке:

- `Location` - URI ресурса, по которому его можно прочитать.
  Шаблон: `/v1/namespace/resources/{resource_id}`

- Если предполагается, что у создаваемого ресурса сложный жизненный цикл, то имеет смысл вернуть заголовок
  `Queue` - URI очереди запросов, где может быть получен статус операции через некоторый временный идентификатор.
  Шаблон: `/v1/queue/requests/{request_id}`

- В случае простых сценариев достаточно наличия заголовка
  `Retry-After` - оценки того, когда обработка завершится. Этот заголовок предназначен для предотвращения частого опроса сервиса клиентами.

12.11.2. GET/PUT/PATCH/DELETE-методы

Как и `POST`, методы `GET/PUT/PATCH/DELETE` могут быть асинхронным. Поведение их в таком случае должно быть следующим:

- Метод `POST` возвращает код `202 Accepted`.

Тело ответа должно включать (в случае поддержки `HATEOAS`) несколько ссылок:

- URI очереди запросов, где может быть получен статус операции через некоторый временный идентификатор. Клиенты должны сделать `GET`-запрос, чтобы получить статус операции, который может включать такую информацию, как состояние завершения, прогресс выполнения операции и окончательный URI после ее завершения.
  Шаблон: `{ "rel": "self", "href": "/v1/queue/requests/{request_id}, "method": "GET" }"`

12.11.3. Поддержка одновременно синхронных и асинхронных запросов

API, которые поддерживают как синхронные, так и асинхронные операции для конкретного URI и комбинации методов HTTP, должны распознавать заголовок `Prefer` (RFC7240) и иметь следующее поведение:

- Если запрос содержит заголовок `Prefer: response-async`, сервис должен переключить обработку в асинхронный режим.
- Если запрос не содержит заголовок `Prefer: response-async`, сервис должен обрабатывать запрос синхронно.

Желательно, чтобы все асинхронные методы API поддерживали какой-либо механизм Push-уведомления для передачи статуса обработки клиенту, чтобы сократить количество Pull-запросов статуса.

12.12. Ресурсы контроллера

Нередки ситуации, когда невозможно смоделировать сервис, выполняющий бизнес-процесс или часть бизнес-процесса, как чистый сервис REST, то есть в ресурсном стиле.

Например:

- Когда требуется выполнить некоторую функцию на сервере с набором входных параметров (предоставленные клиентом данные или из внешнего хранилища информации).
- Когда требуется объединить одну или несколько операций и выполнить их атомарным способом.
- Если необходимо скрыть многоэтапную операцию от клиента, чтобы избежать ненужного клиент-серверного взаимодействия.

В целом рекомендуется избегать подобного подхода, предпочитая по возможности ресурсно-ориентированный дизайн.

12.12.1. Именование ресурсов контроллера

Поскольку операция контроллера представляет собой действие или функцию на сервере, будет интуитивно понятно выразить назвать ее с помощью глагола, то есть самого действия в качестве имени ресурса (например, `activate, validate, cancel, accept`).

Примеры:

URI	Комментарий
`v1/employees/1/fire`	Запустить процедуру увольнения сотрудника
`v1/employees/search`	Поиск по всем сотрудникам

Нельзя определять подресурс для ресурса контроллера. Также важно организовывать ресурсы контроллера с минимальным уровнем вложенности, чтобы избежать загрязнения API.

12.12.2. HTTP-методы для ресурсов контроллера

В большинстве случаев в качестве метода по умолчанию должен использоваться `POST`. В сценариях, где ответы активно кешируются, следует использовать метод `GET`. Например, запрос `GET /employees/average-salary?post=programmer`, возвращающий среднюю зарплату для программистов, может быть естественным образом закэширован.

12.12.3. HTTP-статусы для ресурсов контроллера

Ниже перечислены основные коды, используемые с ресурсами контроллера:

- `200 OK` - в большинстве случаев, вместе с телом, описывающим результат выполнения операции.
- `201 Created` - если выполнение операции привело к созданию ресурса.
- `204 No Content` - если сервис отказывается возвращать что-либо как результат выполнения операции.
- В случае ошибок могут возвращаться стандартные коды 4XX или 5XX.

Ниже пример операции поиска сотрудников по заданным критериям.

Request:

```
POST v1/employees
Content-Type: application/json
{
    "state": "ACTIVE",
    "created_before": "2019-05-13"
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "items": [
      <<  >>
    ]
    "links": [
      {
        "href": "https://human-resources/v1//employees/search?page=2&page_size=10",
        "rel": "next",
        "method": "POST"
      },
      {
        "href": "https://human-resources/v1//employees/search?page=124&page_size=10",
        "rel": "last",
        "method": "POST"
      },
    ]
}
```

12.13. HATEOAS

Будет дополнен в следующей версии документа.

12.14. Массовые операции

Будет дополнен в следующей версии документа.

---

13. Приложение A. Перечень JSON-типов

13.1. name.json

13.1.1. Описание

13.1.2. Схема

```
{
```

Поправить все схемы во всех типах!

```
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "description": "Name of a party",
    "title": "Name of a party",
    "properties": {
        "prefix": {
            "type": "string",
            "description": "Prefix to the party name. Also known as title.",
            "maxLength": 140
        },
        "given_name": {
            "type": "string",
            "description": "The given name of the party. Also known as first name or name. Required when the party
 name is for a person.",
            "maxLength": 140
        },
        "surname": {
            "type": "string",
            "description": "The surname or family name of the party. Also known as last name. Required when the pa
rty name is for a person. Can also be used to store multiple surnames including matronymic surname, or mother's fa
mily name.",
            "maxLength": 140
        },
        "middle_name": {
            "type": "string",
            "description": "The middle name of the party. Can also be used to store multiple middle names includin
g patronymic name.",
            "maxLength": 140
        },
        "suffix": {
            "type": "string",
            "description": "The name suffix for the party.",
            "maxLength": 140
        },
        "alternate_full_name": {
            "type": "string",
            "description": "The alternate name for the party. Can be business name, nick name, or any other name t
hat cannot be parsed into first, last name. Required when the party name is for a business. Can also be used to re
cord yomikata or phonetic name in Japanese.",
            "maxLength": 300
        }
    }
}
```

13.2. address.json

13.2.1. Описание

13.2.2. Схема

13.3. money.json

13.3.1. Описание

13.3.2. Схема

13.4. percentage.json

13.4.1. Описание

13.4.2. Схема

13.5. country_code.json

13.5.1. Описание

13.5.2. Схема

13.6. currency_code.json

13.6.1. Описание

13.6.2. Схема

13.7. language.json

13.7.1. Описание

13.7.2. Схема

13.8. locale.json

13.8.1. Описание

13.8.2. Схема

13.9. province.json

13.9.1. Описание

13.9.2. Схема

13.10. date_time.json

13.10.1. Описание

13.10.2. Схема

13.11. date_no_time.json

13.11.1. Описание

13.11.2. Схема

13.12. time_nodate.json

13.12.1. Описание

13.12.2. Схема

13.13. date_year_month.json

13.13.1. Описание

13.13.2. Схема

13.14. time_zone.json

13.14.1. Описание

13.14.2. Схема

13.15. error.json

13.15.1. Описание

13.15.2. Схема


```
{
    "id": "https://tools.ietf.org/rfc/rfc7807.txt",
    "$schema": "http://json-schema.org/draft-06/schema#",
    "description": "schema for a rfc7807",
    "definitions": {
      "validation": {
        "type": "object",
        "required": [
          "type"
        ],
        "properties": {
          "type": {
            "description": "A URI reference [RFC3986] that identifies the problem type.",
            "type": "string",
            "format": "uri"
          },
          "title": {
            "description": "A short, human-readable summary of the problem type.",
            "type": "string"
          },
          "status": {
            "description": "The HTTP status code ([RFC7231], Section 6) generated by the origin server for this oc
currence of the problem.",
            "type": "number"
          },
          "instance": {
            "description": "A URI reference that identifies the specific occurrence of the problem.",
            "type": "string"
```

```
        },
        "detail": {
          "description": "A human-readable explanation specific to this occurrence of the problem.",
          "type": "string"
        },
        "debugging": {
          "description": "Debugging information for DEV and QA environments.",
          "type": "string"
        },
        "invalid-params": {
          "description": "An array of validation errors.",
          "type": "array",
          "items": {
            "description": "The validation error descriptor.",
            "type": "object",
            "required": [
              "path",
              "name",
              "reason"
            ],
            "properties": {
              "path": {
                "type": "string"
              },
              "name": {
                "type": "string"
              },
              "reason": {
                "type": "string"
              }
            },
            "additionalProperties": false
          },
          "uniqueItems": true
        }
      },
      "additionalProperties": false
    }
  }
}
```

---

14. Приложение B. Коды ответа HTTP

Примечание: в таблице перечислены только коды, используемые при разработке REST-сервисов. Использовать другие существующие HTTP-коды не рекомендуется.

Код ответа	Название	Описание	Версия протокола	Примечания
200	OK		0.9	
201	Created			
202	Accepted			
204	No Content			
400	Bad Request			
401	Unauthorized			
404	Not Found			
409	Conflict			
500	Internal Server Error			

15. Приложение C. Заголовки HTTP

Примечание: в таблице перечислены только заголовки, используемые при разработке REST-сервисов. Использовать другие существующие HTTP-заголовки не рекомендуется. Заголовки, определенные пользователем, могут не поддерживаться инфраструктурой, отсекаться файрволлами и маршрутизаторами.

Заголовок	Описание	Обязательный	Стандарт	Примечания
`Accept`		Да		
`Content-Type`				
`Link`		Нет		
`Location`		Нет		