Правила проектирования API

(API Guidelines)

Версия 3.0g

от 02.11.2021г.

СОДЕРЖАНИЕ

1. Термины и определения

Термин	Сокращение	Определение
HyperText **T**ransfer **P**rotocol	**HTTP**	Протокол передачи гипертекста — протокол прикладного уровня передачи данных
Uniform **R**esource **I**dentifier	**URI**	Унифицированный (единообразный) идентификатор ресурса — последовательность символов, идентифицирующая абстрактный или физический ресурс.
Uniform **R**esource **L**ocator	**URL**	Унифицированный указатель ресурса — система унифицированных адресов электронных ресурсов, или единообразный определитель местонахождения ресурса
Uniform **R**esource **N**ame	**URN**	Единообразное название (имя) ресурса — это постоянная последовательность символов, идентифицирующая абстрактный или физический ресурс.
Representational **S**tate **T**ransfer	**REST**	«передача репрезентативного состояния» или «передача "самоописываемого" состояния» — архитектурный стиль взаимодействия компонентов распределённого приложения в сети.
Application **P**rogramming **I**nterface	**API**	Программный интерфейс приложения — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.
Java**S**cript **O**bject **N**otation	**JSON**	текстовый формат обмена данными, основанный на JavaScript (от подмножества языка стандарта ECMA-262 1999 года).
Ресурс		В REST это первичное представление данных.
Plain **O**ld **J**ava **O**bject	**POJO**	Простой Java-объект, используемый как правило для передачи данных и не реализующий логику.
JSON **W**eb **T**oken	**JWT**	Открытый стандарт (RFC 7519) для создания токенов

		доступа, основанный на формате JSON.

2. Введение

Настоящий документ содержит набор правил проектирования API.

Настоящие правила обязательны к применению при создании API для новых микросервисов Компании.

3. Требования к проектированию REST API

3.1. Использование REST API подхода

При разработке API не в полной мере используется REST API подход. Причины отказа от данного подхода:

- Не все операции по смыслу можно уложить в типы запросов GET, PUT, POST, DELETE. Есть действия по смыслу более сложные и разнообразные.
- Не всякое бизнес-понятие или операцию можно уложить в понятие ресурса.
- При вызове методов из сред аналогичных Postman тип запроса (GET, PUT, POST, DELETE) легко перепутать и вызвать не тот метод. При явном указании действия в имени ресурса ошибиться сложнее.
- Не у всех типов запросов есть тело в запросе и ответе.

Все методы должны иметь тип запроса POST. Все параметры должны находится в теле запроса.

Метод GET может быть использован только в тех случаях, когда необходимо передать ссылку клиенту (например, ссылка для скачивания полиса в теле письма или СМС).

Для передачи данных на сервер в теле запроса должен передаваться объект в формате JSON. Ответ сервера должен содержать объект в формате JSON. С точки зрения backend`а это может быть как POJO, так и Map<String, Object>.

3.2. Структура URI

REST API использует унифицированные идентификаторы ресурсов (URI) для адресации ресурсов.

Необходимо создавать URI так, чтобы они передавали модель ресурсов REST API её потенциальным потребителям. Когда ресурсы названы правильно, API интуитивно понятен и прост в использовании. Иначе API может оказаться трудным для понимания и использования.

В данном документе дальше рассматривается только относительные URI [/resource1/resource2], а не абсолютные [http://my-api.example.com/resource1/resource2].

В относительном URI [/path1/ path2/…/pathN/method] выделяются следующие части:

- Промежуточные - путь до метода:
  [[/path1/ path2/…/pathN/]
- Конечный – сам метод :
  [method]

Данные термины «путь» и «метод» используются ниже.

3.3. Общие требования к именованию URI

При разработке API и формировании URI следует придерживаться следующих правил:

1. Все наименования элементов образуются только из слов английского языка, а не русских слов с простой заменой кириллицы на латиницу (транслитерацией).
2. Не следует применять редко используемые англоязычные переводы, «неологизмы» и «американизмы». Если где-то в системе Вы уже встречали объект со сходной семантикой (пусть не абсолютно совпадающей), следует воспользоваться его наименованием (за исключением неверного) и не выдумывать нового.
3. Допускается использовать сокращения элементов, когда сокращение является общеупотребимым или привычным.
   Например: “vat” вместо “value-added-tax”,
   ins вместо insurance,
   param вместо parameter.
4. При формировании составных наименований объектов порядок следования элементов от общего к частному (от родительского объекта к атрибутам дочернего).
   Например: contract-number или risk-brief.
5. Запрещается использовать подчеркивание (_).
   Если использовать подчеркивание вместо дефиса в качестве разделителя, то может возникнуть ситуация, когда в зависимости от шрифта приложения, символ подчеркивания (_) может быть частично или полностью скрыт.
   Необходимо использовать вместо подчеркивания дефис (-).
6. При именовании необходимо использовать нотацию kebab-case. Запрещается использовать символы в upper case. Дефисы (-) позволяют улучшить читаемость URI.
   Например: get-by-number.
7. Запрещается использовать завершающую косую черту (/) в URI.
   Как последний символ в пути URI, косая черта (/) не добавляет семантического значения и может вызвать путаницу.
   Например:

```
/contract/insured/list-all      /*правильно*/
/contract/insured/list-all/     /*НЕ правильно*/
```

3.4. Требования к именованию пути в URI

Для облегчения понимания структуры URI следует придерживаться следующих правил:

1. В пути URI необходимо использовать только существительные, обозначающие бизнес-объекты, в единственном числе. Данное требование аналогично требованию к именованию таблиц
   Например:
   /contract/get-by-id
2. Иерархия бизнес объектов (вложенные бизнес-объекты) выстраивается в пути URI через косую черту.
   Например:
   /contract/get
   /contract/insured/list
   /contract/insured/address/list

Примечание: ресурс необходимо организовывать таким образом, чтобы корневой путь можно было указать в его аннотации.

3.5. Требования к именованию методов в URI

Так как допускаются только запросы типа POST, то в наименовании метода должно содержаться действие (глагол).

Название метода формируется по следующему шаблону:

<action>-[<attribute>]-[<parameter>]

где:

- <action> - наименование действия, выполняемого над бизнес-объектом (или набором экземпляров объектов). Состоит из стереотипа действия, указывающего суть операции.
  Обязательный элемент наименования.
- <attribute> - наименование атрибутов бизнес-объекта (или дочерних объектов), собственно над которыми выполняется действие.
  Опциональный элемент наименования.
- <parameter> - наименование атрибутов бизнес-объекта (или дочерних объектов), выступающими ограничениями или условиями выполняемого действия. Отделяется от названия действия и атрибутов суффиксом «by» или «for». Суффикс «for» используется исключительно для указания периода (find-for-

period - отбор за период). Во всех остальных случаях используется суффикс «by».
Опциональный элемент наименования.

Наименование бизнес-объекта НЕ включается в наименование метода, а содержится в последнем элементе пути, предшествующем методу.

Примеры:

```
/contract/get-by-id                /*получить договор по идентификатору*/
/contract/update-payment-flag      /*изменить признак оплаты*/
/party/create                      /*создать лицо*/
/party/update                    /*изменить лицо*/
/contract/list-by-param            /*список договоров по параметрам*/
```

Типовые действия <action>:

Действие	**Значение**
create	создать объект
save/update	изменить объект
delete	удалить объект
list	получить список объектов
get	получить один объект

3.6. Запрос на сервер

3.6.1. Указание типа передаваемого контента

При отправке запроса на сервер клиента всегда должен указывать тип передаваемого контента.

Тип контента передаётся в HTTP заголовке **Content-Type**

Тип контента	**Значение для Content-type**
JSON	application/json

XML	application/xml
Файл	multipart/form-data

В случае, если это текстовый контент (JSON или XML), то в **Content-Type** так же необходимо указать кодировку возвращаемого контента.

```
Content-Type: application/json; charset=utf-8
```

3.6.2. Авторизованный запрос на сервер

Для авторизации используется открытый протокол OAuth 2.0, который позволяет предоставить третьей стороне ограниченный доступ к защищённым ресурсам пользователя без необходимости передавать ей (третьей стороне) логин и пароль.

Для формирования авторизованного запроса в HTTP заголовке Authorization необходимо передать Bearer Token

```
Authorization: Bearer <token>
```

3.6.3. Использование query-параметров

В запросах к серверу запрещается использование query-параметров, кроме тех случаев, когда используется GET запрос и иного способа передачи данных на сервер нет.

3.7. Ответ сервера

В стандарте код ответа (состояния) HTTP показывает, был ли успешно выполнен определённый HTTP запрос. В общем случае коды сгруппированы в 5 классов:

1.1.1. Информационные 100 - 199
2.1.1. Успешные 200 - 299
3.1.1. Перенаправления 300 - 399
4.1.1. Клиентские ошибки 400 - 499
5.1.1. Серверные ошибки 500 - 599

Для разрабатываемых сервисов должны быть использованы следующие коды:

Код ответа	Описание
200 OK	Запрос выполнен успешно. Никаких ошибок в ходе выполнения не возникло. Возвращен результат.
400 Bad Request	Указывает, что произошла ошибка и результат операции неуспешный. В логике приложения выброшена исключительная ситуация. Это может быть ошибка бизнес-логики, парсинга данных или их валидации.
401 Unauthorized	Указывает, что запрос не был применён, поскольку ему не хватает действительных учётных данных для целевого ресурса. В этом случае возможна аутентификация на сервере.
403 Forbidden	Указывает, что сервер понял запрос, но отказывается его авторизовать. Доступ запрещён и привязан к логике приложения (например, у пользователя не хватает прав доступа к запрашиваемому ресурсу).
404 Not Found	Указывает, что сервер не может найти запрошенный ресурс.
500 Internal Server Error	Что-то пошло не так и сервер упал из-за непредвиденной ошибки. Такой тип ошибки допускать нельзя.

3.7.1. Успешный ответа сервера

Если запрос клиента был успешно обработан, то сервис должен вернуть код 200.

Не требуется возвращать дополнительные статусы или коды успеха операции. Код 200 говорит о том, что всё выполнилось правильно. Если в ходе возникла ошибка, то необходимо возвращать код 400.

При возврате коллекции сервер должен возвращать объект, содержащий массив запрошенных данных и общее кол-во записей, которое может вернуть сервер с учётом установленных параметров фильтрации (без учёта параметров постраничного вывода).

Пример ответа сервера:

```
{
  "total": 9999,  // Общее кол-во записей
  "records": [    // Массив записей для запрошенной страницы
    {"id": 1, "title": "title 1"},
    {"id": 2, "title": "title 2"},
    {"id": 3, "title": "title 3"},
    {"id": 4, "title": "title 4"},
    {"id": 5, "title": "title 5"},
    {"id": 6, "title": "title 6"}
  ]
```

```
}
```

Если при запросе коллекции не были указаны параметры, ограничивающие выборку, то сервер должен ограничить выборку, в соответствии с параметром, установленным по умолчанию на проекте (данный параметр может отличаться на каждом проекте).

3.7.2. Ответа сервера с ошибкой

Если в результате запроса в сервисе возникла ошибка в бизнес-логике и была сгенерирована исключительная ситуация, то сервер всегда должен возвращать код 400 с описанием ошибки в виде объекта в следующем JSON-формате:

```
{
    "code": 10, // Системный код ошибки
    "text": "Ошибка валидации данных", // Человекочитаемое описание ошибки
    "extendedStatus": { // Детализация ошибки
        "fullText": "Не указано имя\nНе указан пол",
        "violationItems": [
            {
                "property": "name",
                "text": "Не указано имя"
            },
            {
                "property": "gender",
                "text": "Не указан пол"
            }
        ]
    }
}
```

За формирование ответа с кодом 400 отвечает библиотека rest-common.

За формирование ответа с кодом 401, 403 отвечает расширение Keycloak.

3.7.2.1. Внутренние коды ошибок

Для более гибкого управления ошибками используется внутренние коды, который передаются в теле ответа с ошибкой.

Код	Текст ошибки	Описание
1	Другая ошибка	внутренняя ошибка сервиса

2		обрабатываемая ошибка сервиса (ошибка бизнес-логики и т.д.)
3		обрабатываемая бизнес-ошибка сервиса (все случаи, когда нужно логирование в WARN)
4	Ошибка парсинга входящего JSON	ошибка разбора входящего JSON
10	Ошибка валидации данных	ошибка валидации данных

3.7.3. Ответ сервера с передачей файла

Если сервер возвращает файл для скачивания, то сервер должен указывать в заголовке ответа тип файла, его имя и размер.

```
Content-Type: application/ms-excel
Content-Disposition: attachment; filename="MyFileName.xlsx"
Content-Length: 23999
```

Если возвращаемый файл необходимо открыть средствами просмотра браузера, то в **Content-Disposition** вместо **attachment**, необходимо указать **inline**

```
Content-Type: application/pdf
Content-Disposition: inline; filename="MyFileName.pdf"
Content-Length: 13418
```

3.7.4. Правила использования HTTP-заголовков

TODO: Описать разрешённые заголовки

TODO: Описать правила использования кастомных заголовков

4. Взаимодействие frontend и backend

4.1. Микрофронтед и микросервис

Микрофронтед является прямым продолжением идеи микросервисной архитектуры для фронтенд-разработки. Таким образом, по аналогии с микросервисом, микрофронтенд – это небольшая, логически отделенная часть приложения, а точнее, в контексте разработки frontend, часть пользовательского интерфейса вашего приложения. Часть, которая в первую очередь не зависит от остальных частей системы с точки зрения ее развертывания. Кроме того, Микрофронтед должен инкапсулировать некоторую часть пользовательского интерфейса бизнес-логики приложения и должен принадлежать исключительно команде, которая отвечает за эту часть приложения от начала до конца.

4.2. Правила взаимодействия микрофронтенда и микросервиса

При проектировании и реализации функционала необходимо придерживаться следующих правил:

1. Микрофронтенд может только с тем микросервисом, которому он принадлежит (это «свой» микросервис).
   Запрещается вызывать напрямую из микрофронтенда «чужой» микросервис. Если есть необходимость в вызове функционала «чужого» микросервиса, то необходимо «проксировать» вызов «чужого» микросервиса через «свой» микросервис. Т.е. в «своём» микросервисе сделать endpoint, через который будет осуществляться вызов другого микросервиса.
2. Часть пользовательского интерфейса микрофронтенда, который может использоваться в других микрофронтендах, необходимо выносить в отдельные компоненты.
   Данные компоненты должны также вызывать только «свой» микросервис.
3. Общие компоненты должны иметь чёткое версионирование.
4. На текущий момент без явной необходимости не применяет шаблон BFF.
   Под явной необходимостью понимается, например, разные типы клиентских приложений (десктоп-веб, мобильный клиент).

Пример взаимодействия изображён на схеме ниже.

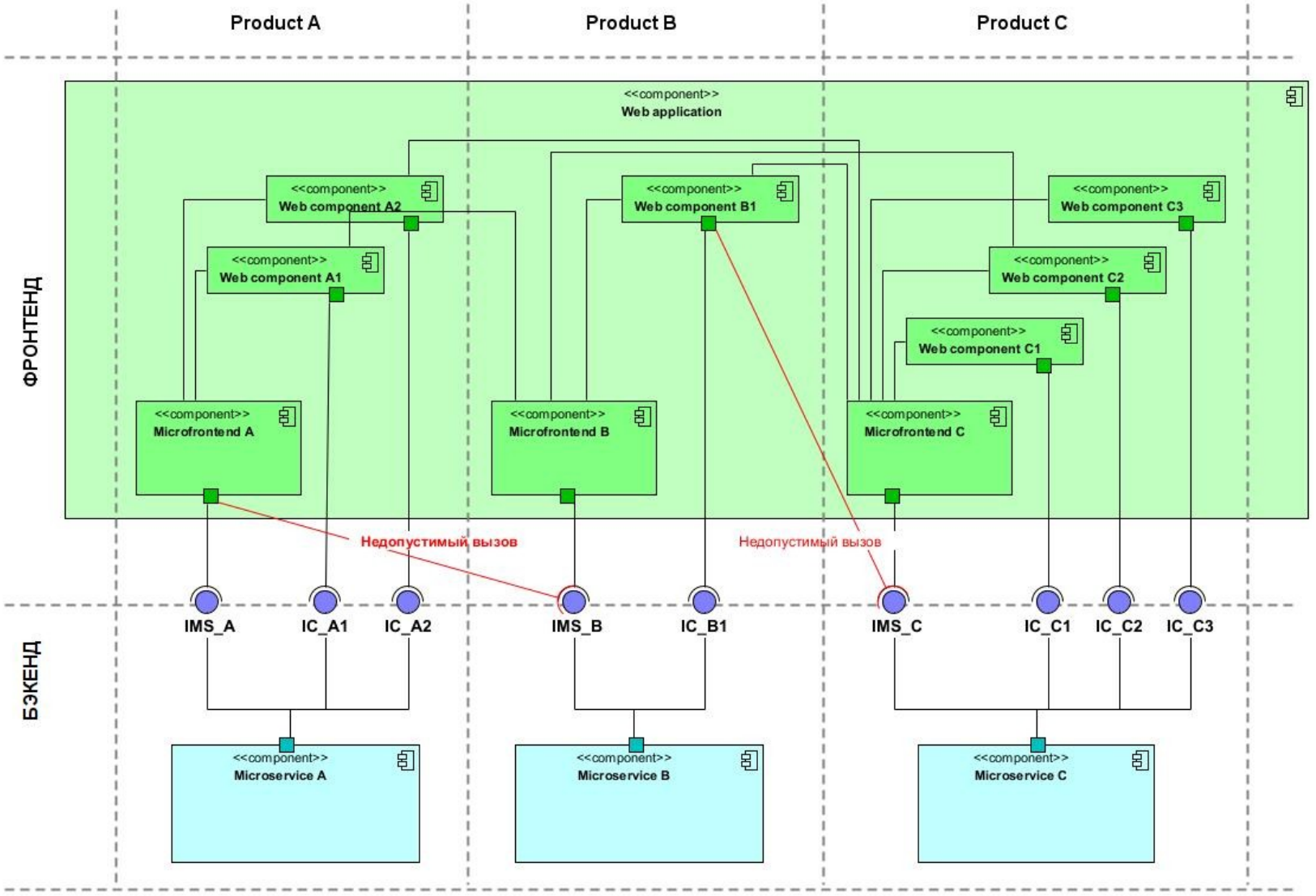

Схема взаимодействия микрофронтенда и микросервиса

Пояснения к схеме:

- Функционал системы состоит из 3-х продуктов (A, B, C).
- В рамках фронтенда есть веб приложение на основе single spa, внутри которого есть 3 микрофронтенда.
- Соответствующие друг другу микрофронтенд и микросервис взаимодействуют через интерфейсIMS_X (Interface Microservice X)
  Микрофронтенд использует, а микросервис реализует интерфейс.
- Функционал пользовательского интерфейса, используемый в «чужих» микрофронтендах, вынесен в компоненты Web component X.
- Веб компоненты взаимодействуют со «своим» микросервисом через интерфейсIC_XY (Interface Component XY).
  Веб компонент использует, а микросервис реализует интерфейс.
- Микрофронтенды могут использовать веб-компоненты «своего» или «чужого» микросервиса.
- Взаимодействия, отмеченные красным цветом запрещены.

4.3. Потенциальные проблемы подхода с выделением компонент

С ростом количества вызов «чужого» функционала растёт потребность выносить всё больше вызываемого интерфейсного функционала в отдельные компоненты.

Это приводит к:

- Необходимости чётко отслеживать версии и совместимость версий при сборке стенда.
  Может получиться так, что для разных микрофронтендов могут потребоваться разные версии одной и той же компоненты.
- Кроме того, поскольку версия компоненты зависит ещё и от микросервиса, то цепочка зависимостей удлиняется. Это своего рода «монолитизация» приложения и повышение зависимости команд друг от друга.

Пока будем считать, что объём одновременно разрабатываемого нами функционала и количество команд позволяют решить эти проблемы аккуратным выстраиванием процесса разработки и сборки.