



2 268,94

Рейтинг

Подписаться

RUVDS.com

VDS/VPS-хостинг Скидка 10% по коду **HABR**

Моя лента Все потоки Разработка Администрирование Дизайн Менеджмент Маркетинг Научпоп



ru_vds 30 мая 2017 в 15:33

Node.js и cote: простая и удобная разработка микросервисов

Автор оригинала: Armağan Amcalar

Блог компании RUVDS.com, [Разработка веб-сайтов](#), [JavaScript](#), [Node.JS](#), Микросервисы

Перевод

Многие считают, что микросервисы — это очень сложно. На самом же деле, при правильном подходе, это совсем не так.

Микросервисы сегодня весьма популярны, а настоящие приверженцы этой архитектуры едва ли не кланяются всему, на чём написано «микросервис». Однако, если отбросить фанатизм, подобный подход к разработке ПО — это достойный шаг вперёд,

микросервисы навсегда могут изменить то, как создают серверные части приложений. Вокруг микросервисов много информационного шума, поэтому стоит выделить по-настоящему важные свойства этой архитектуры и поработать над тем, чтобы упростить её внедрение и использование там, где это действительно нужно.



Если вы относитесь к микросервисам с осторожностью, или чувствуете, что в этой теме запутались, знайте, что вы не одиноки. С архитектурной точки зрения микросервисы устроены не так уж и просто. Дело усугубляется сложившейся вокруг них экосистемой. Вместо того, чтобы напрямую решать сложные задачи, сопутствующие работе с микросервисами, те, кто этим

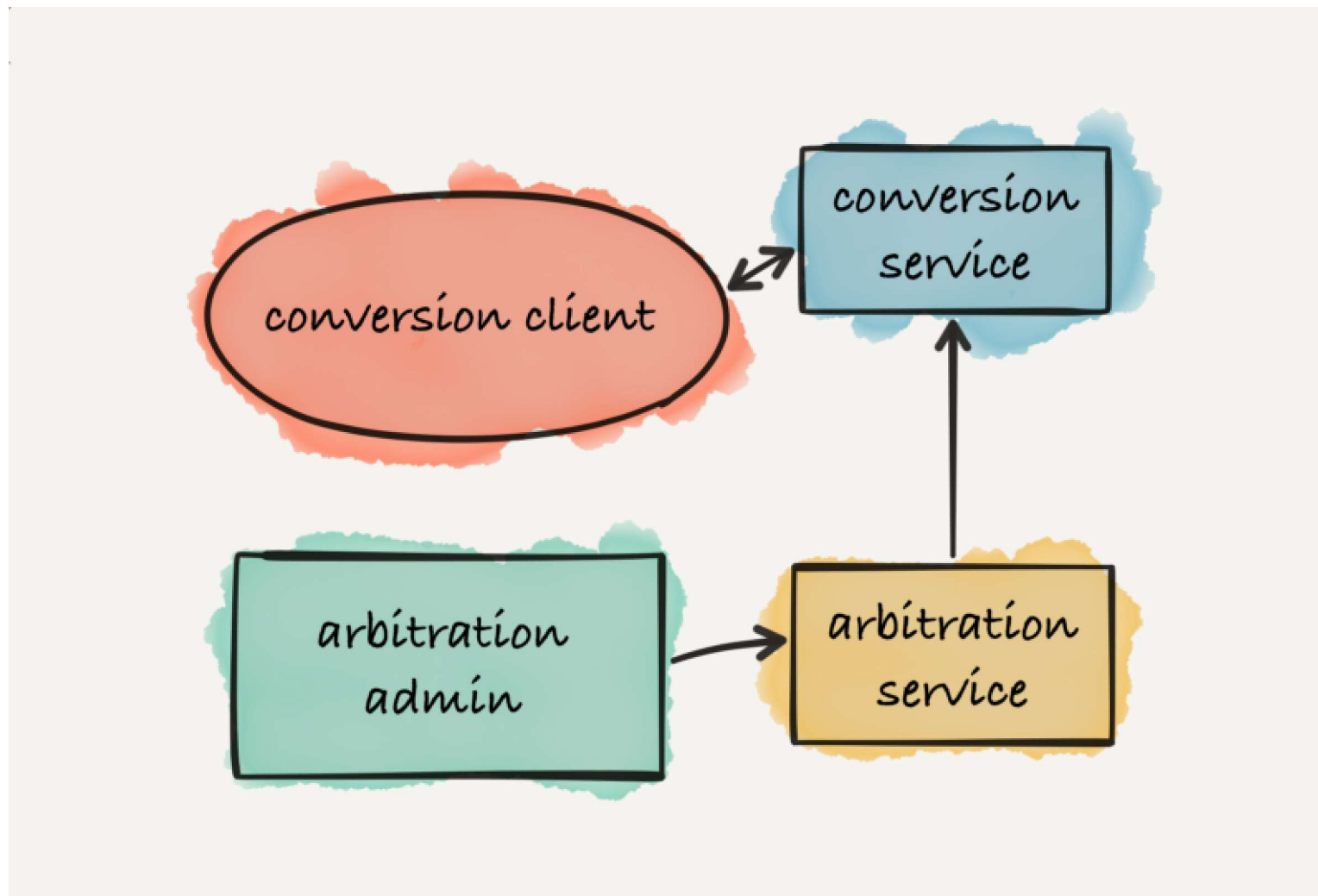
занимаются, пытаются уйти от прямых решений, упрощая одно и усложняя другое. В итоге это приводит к тому, что приходится поддерживать большой набор технологий и иметь дело с ошибками, которые невозможно отлаживать именно из-за сложности всей конструкции. Нелегко даже просто проверить работоспособность приложения. Пришло время эту ситуацию исправить.

Здесь мы пошагово разберём разработку веб-приложения, основанного на микросервисах. Этот пример вполне может быть вашим первым подобным проектом, а если вы разбираетесь в разработке для Node.js, на то, чтобы всё у вас заработало, уйдут считанные минуты.

Прежде чем мы приступим к делу, мне хотелось бы уточнить, что я являюсь автором библиотеки для Node.js [cote](#), которая упрощает и ускоряет разработку приложений, основанных на микросервисах.

Пример реализации приложения, основанного на микросервисах

Существует множество способов демонстрации реализации архитектуры микросервисов, и, на самом деле, основательно проработанный [пример](#) — это всё, что нужно опытному разработчику. Однако, этот материал рассчитан на программистов с разным уровнем подготовки, в том числе и на новичков, поэтому для демонстрации был выбран простой, но полноценный пример: приложение для конвертации валют. Мы создадим систему из четырёх сервисов, совместная работа которых позволит организовать конверсию в условиях колебания курсов валют.



Обзор сервисов. Каждый из них может независимо масштабироваться. Стрелки указывают направления потоков данных.

Мы создадим систему, состоящую из следующих частей.

1. Клиент конвертера валют (conversion client) — он может запрашивать пересчёт некоего количества денег в одной валюте, в другую валюту.

2. Сервис конвертера валют (conversion service) — ему известны курсы валют, он может отвечать на запросы клиента. Кроме того, этот микросервис может принимать обновления, касающиеся курсов пересчёта валют.
3. Сервис арбитража (arbitration service) — он поддерживает в актуальном состоянии сведения о курсах валют, и, при их изменении, публикует обновления.
4. Средство администрирования арбитражного сервиса (arbitration admin) — это сервис, который вы можете реализовать самостоятельно, в качестве домашнего задания. Он нужен для управления сервисом арбитража, позволяя сообщать ему о необходимости обновить какой-либо курс пересчёта валют.

Прежде чем мы приступим к делу, поговорим немного об истории микросервисов.

Субъективный взгляд на историю микросервисов

Давайте взглянем в прошлое. Десять лет назад, когда микросервисы ещё и не появились, в почёте была сервис-ориентированная архитектура (SOA, service-oriented architecture). Существовали сотни решений для построения сервис-ориентированных приложений, а вокруг всего этого был построен серьёзный консалтинговый бизнес.

Продвинемся лет на семь ближе к нашему времени, здесь мы уже встретимся с микросервисами. Хотя на фундаментальном уровне они очень сильно отличаются от SOA, те, кто раньше строил сервис-ориентированные приложения, переключились на новую технологию. Всё бы хорошо, но за плечами у этих разработчиков был багаж из прошлого, в котором много лишнего.

В одночасье существующие подходы к разработке были перенесены в новую среду. Хотя основные предпосылки, лежащие в основе микросервисов, отличались от SOA, эти первые решения, основанные на старых технологиях, преподносились общественности как «единственно верные». В результате, кроме прочего, у нас были AMQP, HTTP, или даже, боже упаси, SOAP. Это были nginx, zookeeper, etcd, consul и несколько других решений, которым пытались придать вид инструментов, жизненно необходимых для создания микросервисов.

Проблема со всем этим заключалась в том, что данные технологии изначально создавались для решения совсем других

проблем. Их, так сказать, «перепаяли» под микросервисы. Однако, после того, как разработчики использовали некоторые из этих инструментов для построения собственных сервисов, они, в лучшем случае, оставляли ощущение временных подручных средств. Это напоминало забивание гвоздей плоскогубцами. Когда больше ничего под рукой нет — гвоздь забить можно, но не лучше ли поискать нормальный молоток?

Всегда было очевидно, что барьер входа в сферу микросервисов слишком высок, хотя переход на них и сулил экономические выгоды. Однако, за окном 2017-й год, все мы заслуживаем лучшего, чем старые технологии, с помощью которых решают новые задачи. Пришло время заявить, что для того, чтобы изучать, создавать и использовать микросервисы, для того, чтобы добиться их масштабирования при решении реальных проблем, нужен только Node.js.

Теперь поговорим об архитектуре современных микросервисов.

Пять требований к микросервисам

Вот требования, которым должны соответствовать современные системы, основанные на микросервисах.

1. **Автоматическое конфигурирование системы.** Любая система, основанная на микросервисах, скорее всего, будет включать в себя сотни сервисов. Ручная настройка IP-адресов, портов и возможностей API — задача практически невыполнимая.
2. **Высокий уровень избыточности системы.** В вышеописанной системе вполне возможны отказы микросервисов, поэтому очень важно наличие механизма аварийного переключения на резервные мощности, создание которых не должно доставлять особых хлопот и издержек.
3. **Отказоустойчивость системы.** Система должна выдерживать и достойно обрабатывать нештатные события, такие, как сбой в сети, ошибки при обработке сообщений, тайм-ауты, и так далее. Если даже некие сервисы перестали функционировать, все остальные, не связанные с ними, должны работать.

4. **Самовосстановление системы.** К сбоям и разного рода ошибкам надо относиться как ко вполне ожидаемым явлениям. В реализацию системы должно быть заложено автоматическое восстановление любого утраченного вследствие сбоя функционала или сервиса.

5. **Автоматическое обнаружение сервисов.** Существующие сервисы должны автоматически идентифицировать новые сервисы, введённые в систему, начинать взаимодействие с ними без ручного вмешательства или простоев.

Если ваша архитектура соответствует этим требованиям и если вы разбили выполнение большинства ваших API-запросов на несколько независимых сервисов, в таком случае то, что вы сделали — это и есть микросервисы.

Это — не микросервисы

Мне хотелось бы, ещё раз подчеркнуть одну важную вещь: архитектура микросервисов не привязана к какой-то определённой технологии. Скажем, старые добрые рабочие очереди и потоки-потребители — это не микросервисы. Почтовые демоны, механизмы уведомлений, любые вспомогательные службы, которые лишь потребляют события и ничего не делают для обработки запросов пользователей — это не микросервисы.

Некое административное приложение и клиентская программа — это разные вещи? Такое разделение тоже не означает применение микросервисной архитектуры. Серверные демоны, которые могут получать и обрабатывать HTTP-запросы... И это не микросервисы. А что, если у нас есть кластер серверов, компьютеры, входящие в который, названы в честь лун Юпитера? Нужно ли администратору вмешиваться в их работу? Если да — то и тут нет никаких микросервисов.

Понятно, что на волне популярности новой технологии заманчиво называть «микросервисными архитектурами» всё подряд. Однако, это лишь создаёт информационный шум и не даёт другим людям понять, что такое микросервисы, а значит и использовать их в своих разработках.

Микросервисы всегда стремились к минимализму. «Тяжёлые» технологии, о которых говорят, что только они делают возможным создание микросервисов — это прямая им противоположность.

Библиотека cote и микросервисы

Микросервисы, построенные на базе библиотеки `cote`, конфигурируются автоматически. Библиотека использует широковещательные или многоадресные IP-рассылки, в результате демоны в одной сети находят друг друга и автоматически обмениваются необходимой для настройки соединения информацией. В этом отношении `cote` удовлетворяет требованиям №1 и №5: «Автоматическое конфигурирование системы» и «Автоматическое обнаружение сервисов».

С помощью `cote` можно весьма эффективно, с малыми затратами ресурсов, создавать множество копий сервисов, при этом обработка запросов масштабируется автоматически. Это значит, то библиотека соответствует и требованию №2: «Высокий уровень избыточности системы».

Если в системе нет сервисов для удовлетворения некоего запроса, `cote` кэширует его и ждёт до тех пор, пока нужный сервис не окажется доступен. Так как сервисы обычно независимы, подобная организация системы означает её отказоустойчивость, а это соответствует требованию №3: «Отказоустойчивость системы».

Оставшееся требование №4: «Самовосстановление системы», выполняется благодаря использованию Docker, давая возможность перезапускать сервисы при сбоях. Так как в системе работает автоматическое обнаружение сервисов, даже если Docker решит развернуть упавший сервис на новой машине, все оставшиеся сервисы найдут его и будут с ним взаимодействовать.

Таким образом `cote` даёт разработчику полную свободу в плане инфраструктуры, беря на себя заботу о выполнении основных требований к системам, построенным на базе микросервисов.

С помощью `cote` вы можете сосредоточиться на самых важных аспектах разработки приложений, оставив библиотеке выполнение вспомогательных задач.

Полагаю, я доступно и понятно рассказал о том, что такое — настоящие микросервисы. Теперь давайте поговорим о том, как всё это реализовать.

■ Установка `cote`

Мы создадим систему, основанную на микросервисах, в среде Node.js, с помощью библиотеки `cote`, основные возможности которой мы рассмотрели выше. Она доступна в виде `npm`-пакета для Node.js.

Установим cote:

```
npm install cote
```

■ Первое знакомство с cote

Хотите ли вы интегрировать cote с существующим веб-приложением, например, основанным на express.js, как показано [здесь](#), собираетесь ли переписать некоторую часть монолитного приложения, или решили перевести несколько существующих микросервисов на cote, работа будет заключаться в том, чтобы создать несколько экземпляров компонентов cote и использовать их сообразно вашим нуждам. Среди этих компонентов, например — [Responder](#), [Requester](#), [Publisher](#), [Subscriber](#), подробности о них вы узнаете ниже. Эти компоненты рассчитаны на взаимодействие друг с другом, позволяя реализовать наиболее распространённые сценарии разработки приложений.

Простое приложение, или очень маленький микросервис, вполне могут быть построены так, что на один Node.js-процесс будет приходиться реализация одного компонента cote. Однако, более сложные проекты требуют более тесной связи и совместной работы множества компонентов и микросервисов. Библиотека cote поддерживает и такие сценарии.

Начнём с клиента, файл которого назовём `conversion-client.js`. Его роль — запрашивать операцию конверсии валют и, при получении ответа, производить какие-то действия.

Реализация механизма запросов и ответов

Наиболее распространённый сценарий взаимодействия компонентов, реализуемый во множестве приложений — это цикл запросов-ответов. Обычно некий микросервис запрашивает выполнение задачи у другого микросервиса, или выполняет к нему запрос и получает от него ответ. Реализуем подобную схему взаимодействия с помощью cote.

Для начала, в файле `conversion-client.js`, подключим cote.

```
const cote = require('cote');
```

Тут ничего особенного не происходит — обычный вызов библиотеки.

■ Создание компонента Requester

Начнём с компонента Requester, который будет запрашивать выполнение операции конверсии валют. И Requester, и другие компоненты, это функции-конструкторы в модуле cote, поэтому создают их с помощью ключевого слова new.

```
const requester = new cote.Requester({ name: 'currency conversion requester' });
```

В качестве первого аргумента конструкторы компонентов cote принимают объект, в котором, как минимум, должно содержаться свойство name, представляющее собой имя компонента, служащее для его идентификации. Имя используется, в основном, как идентификатор для целей мониторинга компонентов, оно оказывается очень кстати при чтении логов, так как, по умолчанию, каждый компонент логирует имена других обнаруженных им компонентов.

Компоненты, создаваемые конструктором Requester, предназначены для отправки запросов, предполагается, что они будут использоваться вместе с объектами, которые создаёт конструктор Responder, отвечающими на запросы. Если подобных компонентов не обнаружено, компонент Requester будет накапливать запросы в очереди до тех пор, пока не появится Responder, к которому можно будет отправить эти запросы. Если компонентов Responder будет несколько, Requester будет обращаться к ним, используя алгоритм кругового опроса, балансируя нагрузку на них.

Создадим запрос типа convert, который предназначен для преобразования некоей суммы, выраженной в долларах США, в евро.

```
const request = { type: 'convert', from: 'usd', to: 'eur', amount: 100 };  
  
requester.send(request, (res) => {
```

```
    console.log(res);  
  });
```

Напомним, что до сих пор мы работали в файле `conversion-client.js`. Вот, на всякий случай, его полный код.

```
const cote = require('cote');  
  
const requester = new cote.Requester({ name: 'currency conversion requester' });  
  
const request = { type: 'convert', from: 'usd', to: 'eur', amount: 100 };  
  
requester.send(request, (res) => {  
  console.log(res);  
});
```

Для того, чтобы выполнить этот файл, воспользуйтесь такой командой:

```
node conversion-client.js
```

Сейчас выполнение запроса ни к чему полезному не приводит, нет даже логов в консоли, так как в нашей системе пока нет компонента, способного отреагировать на запрос и вернуть что-то в ответ.

Пусть этот процесс node.js выполняется, а мы пока займёмся компонентом `Responder`, который сможет отвечать на запросы.

■ Создание компонента `Responder`

Сначала, как и в прошлый раз, подключим `cote` и создадим экземпляр объекта `Responder`, воспользовавшись ключевым словом `new`.

```
const cote = require('cote');

const responder = new cote.Responder({ name: 'currency conversion responder' });
```

Эта часть нашей системы будет представлена файлом `conversion-service.js`. Каждый `Responder`, кроме прочего, является экземпляром объекта `EventEmitter2`. Ответ на некий запрос, скажем, типа `convert` — это то же самое, что прослушивание события `convert` и обработка его с помощью функции, которая принимает два параметра — запрос и функцию обратного вызова. Параметр запроса содержит в себе сведения об одном запросе, в целом, это тот же объект `request`, который был отправлен компонентом `Requester`, рассмотренным выше. Второй параметр — это функция обратного вызова, которая вызывается с передачей ей того, что должно быть отправлено в ответ.

Вот как выглядит простая реализация вышеописанного механизма.

```
const rates = { usd_eur: 0.91, eur_usd: 1.10 };

responder.on('convert', (req, cb) => { // в идеале, тут хорошо бы привести в порядок входные данные
  cb(req.amount * rates[`${req.from}_${req.to}`]);
});
```

Вот полный код файла `conversion-service.js`.

```
const cote = require('cote');

const responder = new cote.Responder({ name: 'currency conversion responder' });

const rates = { usd_eur: 0.91, eur_usd: 1.10 };

responder.on('convert', (req, cb) => {
  cb(req.amount * rates[`${req.from}_${req.to}`]);
});
```

Сохраним файл, и, в новом терминале, выполним его с помощью node:

```
node conversion-service.js
```

Сразу же после запуска сервиса вы увидите, как будет выполнен первый запрос, поступивший от `conversion-client.js`. Информация об этом попадёт в лог. Собственно говоря, зная всё, о чём мы только что говорили, вы уже можете приступать к созданию собственных микросервисов.

Обратите внимание на то, что мы не настраивали IP-адреса, порты, имена хостов или что угодно другое. А ещё — примите поздравления! Только что вы создали свой первый набор микросервисов.

Теперь, в разных терминалах, вы можете запустить множество копий каждого сервиса и убедиться в том, что всё это отлично работает. Остановите несколько сервисов конверсии, перезапустите их, и вы увидите, что созданная нами система соответствует требованиям к современным микросервисам. Если вы хотите масштабировать проект, развернуть его на нескольких серверах или в нескольких дата-центрах, вы можете либо использовать собственное решение, либо воспользоваться возможностями Docker, который позволяет решать множество задач по управлению инфраструктурой.

Займёмся дальнейшим развитием нашего конвертера валют.

Отслеживание изменений в системе с использованием механизма издатель-подписчик

Одно из преимуществ систем, построенных на базе микросервисов, заключается в том, что в них очень легко реализовать механизмы, которые раньше требовали серьёзных вложений в инфраструктуру. Среди задач, решаемых с помощью подобных механизмов — управление обновлениями и отслеживание изменений в системе. Ранее это требовало, как минимум, инфраструктуры очереди с разветвлением. Подобные вещи нелегко масштабировать, ими сложно управлять, это было одним из ограничивающих факторов развития подобных систем.

Библиотека `cote` решает такие проблемы совершенно естественным и интуитивно понятным образом.

Предположим, нам нужен арбитражный сервис, который устанавливает курсы валют, и, если произошли изменения, уведомляет все экземпляры сервисов-конвертеров о том, что им надо использовать новые значения курсов.

Самое важное тут — уведомление всех экземпляров сервисов-конвертеров. В высокодоступном приложении, основанном на микросервисах, можно ожидать наличия нескольких одинаковых сервисов, между которыми распределяется нагрузка. Когда происходит обновление курсов валют, все эти сервисы должны быть проинформированы об изменениях. Если бы в приложении был лишь один сервис-конвертер, подобное легко было бы реализовать с помощью механизма запросов-ответов. Но так как мы стремимся к масштабируемой архитектуре, не хотим ограничивать себя в количестве одновременно работающих сервисов, нам нужен механизм, который уведомлял бы каждый из таких сервисов, причём, надо, чтобы все они получали подобные уведомления одновременно. В `cote` это достижимо с помощью механизма издателей и подписчиков.

Конечно, сам по себе сервис арбитража не является независимым, управление им надо реализовать посредством некоего API, так, чтобы он мог принимать сведения о новых курсах посредством специальных запросов. В результате, например, администратор сможет вводить в систему сведения о новых курсах, которые, в итоге, будут доходить до сервисов-конвертеров. Для того, чтобы этого добиться, сервис арбитража должен включать в себя два компонента. Один из них — это компонент `Responder`, ответственный за реализацию API обновления курсов из внешнего источника, а второй — компонент `Publisher`, который публикует обновления, уведомляя сервисы-конвертеры. В дополнение к этому, сами сервисы-конвертеры должны включать в себя компонент `Subscriber`, который позволит им подписываться на обновления курсов. Посмотрим, как всё это сделать.

■ Создание арбитражного сервиса

Разработаем арбитражный сервис. Для начала подключим `cote` и создадим объект `Responder` для API. А теперь — небольшая, но очень важная деталь, касающаяся создания микросервисов с помощью `cote`. Так как объекты `Requester` конфигурируются автоматически, каждый из них подключится к каждому объекту `Responder`, который ему удастся обнаружить, независимо от типов запросов, на которые этот `Responder` может отвечать. Это означает, что каждый `Responder` должен отвечать на в точности один и тот же набор запросов, так как объекты `Requester` будут распределять нагрузку между всеми объектами `Responder`, к которым они подключены, вне зависимости от их возможностей, то есть — не обращая внимания на то, могут ли они обслуживать запросы определённого типа.

В данном случае мы собираемся создать компонент `Responder` для набора запросов, отличающегося от того, который уже могут обрабатывать существующие в системе микросервисы. Это означает, что нам нужно, чтобы новый `Responder` отличался от обычных сервисов, обслуживающих запросы типа `convert`. В `cote` это можно сделать, задав ключ компонента. Ключи — самый простой способ регулирования взаимодействия сервисов. Вот как создать компонент `Responder` для арбитражного сервиса.

```
const cote = require('cote');

const responder = new cote.Responder({ name: 'arbitration API', key: 'arbitration' });
```

Теперь нужен механизм для отслеживания курсов валют в системе. Скажем, они хранятся в локальной переменной в области видимости модуля. Это вполне может быть и база данных, к которой обращается сервис, но, для того, чтобы не усложнять пример, пусть это будет локальная переменная.

```
const rates = {};
```

Теперь компонент `Responder` должен будет отвечать на запросы типа `update rate`, позволяя администратору обновлять курсы валют с помощью служебного приложения. В настоящий момент интеграция нашей системы со вспомогательным приложением, чем-то вроде рабочего кабинета администратора, не важна, однако, вот пример того, как подобное приложение может взаимодействовать с компонентами `Responder`, реализованными с помощью `cote`, работающими на сервере. Арбитражный сервис должен иметь компонент `Responder`, который может принимать сведения о новых курсах валют.

```
responder.on('update rate', (req, cb) => {
  rates[req.currencies] = req.rate; // { currencies: 'usd_eur', rate: 0.91 }
  cb('OK!');
});
```

В качестве упражнения, вы можете создать компонент `Requester` для того, чтобы он, например, периодически делал запросы типа `update rate`, изменяя при этом курсы. Назовите файл с этим механизмом `arbitration-admin.js` и внедрите в него что-то

вроде таймера на базе `setInterval`, выдающего каждый раз разные курсы валют для арбитражного сервиса.

■ Создание компонента `Publisher`

Теперь у нас есть механизм для обновления курсов валют, но остальной части системы об этом ничего не известно. В частности, речь идёт о сервисах конверсии валют. Для того, чтобы сообщать им об изменениях курсов, надо, в арбитражном сервисе, воспользоваться компонентом `Publisher`.

```
const publisher = new cote.Publisher({ name: 'arbitration publisher' });
```

Теперь, при обновлении курсов, надо воспользоваться возможностями этого компонента. В результате обработчик запроса типа `update rate` надо будет отредактировать так, как показано ниже.

```
responder.on('update rate', (req, cb) => {  
  rates[req.currencies] = req.rate;  
  cb('OK!');  
  
  publisher.publish('update rate', req);  
});
```

Теперь работа над арбитражным сервисом завершена, ниже, как обычно, показан его полный код, который должен находиться в файле `arbitration-service.js`.

```
const cote = require('cote');  
  
const responder = new cote.Responder({ name: 'arbitration API', key: 'arbitration' });  
const publisher = new cote.Publisher({ name: 'arbitration publisher' });  
  
const rates = {};
```

```
responder.on('update rate', (req, cb) => {  
  rates[req.currencies] = req.rate;  
  cb('OK!');  
  
  publisher.publish('update rate', req);  
});
```

Так как пока в системе нет подписчиков на события обновления курсов, сервисы-конвертеры не узнают о том, что курсы валют изменились. Для того, чтобы механизм издатель-подписчик заработал, нужно вернуться к уже известному вам файлу `conversion-service.js` и добавить в него компонент `Subscriber`.

■ Создание компонента `Subscriber`

Порядок работы с компонентами типа `Subscriber` ничем не отличается от других компонентов. Создадим, в файле `conversion-service.js`, новый экземпляр соответствующего объекта.

```
const subscriber = new cote.Subscriber({ name: 'arbitration subscriber' });
```

Объекты типа `Subscriber` расширяют функциональность объектов `EventEmitter2`, и, хотя эти сервисы могут работать на компьютерах, расположенных на разных континентах, любые новые данные, поступающие от издателя, в конечном счёте, будут восприняты подписчиком как событие, которое нужно обработать.

Добавим в `conversion-service.js` следующий код, который позволит прослушивать обновления, которые публикует арбитражный сервис.

```
subscriber.on('update rate', (update) => {  
  rates[update.currencies] = update.rate;  
});
```

Вот и всё. После того, как все наши микросервисы будут запущены, сервисы конверсии будут синхронизироваться с арбитражным сервисом, получая публикуемые им сообщения. Запросы на конверсию, поступающие после обновления курсов, будут выполняться уже с учётом свежих данных.

Только что мы создали три сервиса, совместная работа которых позволила реализовать систему конверсии валют, основанную на микросервисной архитектуре. Вот [репозиторий](#) на GitHub, в котором вы можете найти все три микросервиса, которыми мы тут занимались, и сервис для автоматического обновления курсов валют. Если хотите — клонируйте репозиторий и поэкспериментируйте с его содержимым.

Что дальше?

Если тема разработки микросервисов с использованием cote вам интересна, взгляните на [репозиторий](#) библиотеки на GitHub, [присоединяйтесь](#) к нашему сообществу на Slack. Проект нуждается в активных участниках, поэтому, если вы хотите внести вклад в дело превращения микросервисов в широко распространённый, доступный всем желающим инструмент, дайте нам знать.

Вот ещё один [репозиторий](#), в котором вы можете найти расширенный пример, представляющий собой реализацию простого приложения для электронной коммерции на базе cote. В этом примере, в частности, имеется следующее.

- Административное приложение с функцией обновлений в режиме реального времени для управления каталогом продукции и вывода сведений о продажах с помощью RESTful API (express.js).
- Страничка для пользователей-клиентов, с помощью которой они могут покупать товары. Она тоже обновляется в режиме реального времени, здесь используется технология WebSockets (socket.io).
- Микросервис для обслуживания нужд пользователей, выполняющий CRUD-операции.
- Микросервис для работы с продуктами, опять же, реализующий CRUD-операции.
- Микросервис, позволяющим клиентам совершать покупки.

- Платёжный микросервис, который занимается обработкой финансовых транзакций, возникающих как следствия покупок товаров.
- Конфигурация Docker Compose для локального запуска системы.
- Облачная конфигурация Docker для запуска системы в Docker Cloud.

Если вы хотите поэкспериментировать с cote в Docker или в Docker Cloud, вот [запись вебинара](#). Тут вы найдёте пошаговое руководство, которое, с чистого листа, демонстрирует создание рабочей системы, основанной на микросервисах, поддерживающей масштабирование и непрерывную интеграцию.

Итоги

В этом материале мы, очень кратко, рассмотрели тему разработки приложений на основе микросервисов. Это — введение в архитектуру микросервисов, общий обзор методики разработки. Однако, хотя приложение, которое мы тут создали, состоит всего из нескольких строк кода, тот же подход можно использовать и на гораздо более масштабных проектах. Надо отметить, что и библиотеку cote мы рассмотрели лишь в общих чертах.

Полагаю, мы находимся на пороге очень интересных времён, когда можно наблюдать изменение подходов к разработке ПО. Я горячо поддерживаю путь упрощения, по которому идёт индустрия веб-разработки. Библиотека cote — один из шагов по этому пути, и, на самом деле, это — только начало. Надеюсь, этот материал помог тем, кто задумывался о микросервисах, но так и не решался внедрить их в свои проекты.

Уважаемые читатели! А вы пользуетесь микросервисами в своих разработках?

Теги: Микросервисы, разработка, Node.js, cote, JavaScript

Хабы: Блог компании RUVDS.com, Разработка веб-сайтов, JavaScript, Node.JS, Микросервисы

↑ +18 ↓ 165 28,7k 15 ➦ Поделиться



RUVDS.com

VDS/VPS-хостинг. Скидка 10% по коду **HABR**

Подписаться



205,5

Карма

436,1

Рейтинг



Подписаться

@ru_vds

Пользователь

Facebook

Twitter

ВКонтакте

ПОХОЖИЕ ПУБЛИКАЦИИ

7 февраля 2020 в 12:30

Node.js, Tor, Puppeteer и Cheerio: анонимный веб-скрапинг

↑ +34

👁 16,7k

🔖 219

💬 16

31 августа 2018 в 11:11

Почему человек из мира Java стал горячим сторонником Node.js и JavaScript?

↑ +20

👁 21k

🔖 89

💬 31

20 декабря 2017 в 15:22

Node.js и JavaScript для серверной разработки

↑ +13


👁 38,2k

🔖 140

💬 49

Комментарии 15


Отслеживать новые в ☐ почте ☐ трекере

 **GDRepo** 30 мая 2017 в 17:05 # 📖

↑ +1 ↓

За «котэ» прям сразу лайк! Если серьезно, в Node.js я лютейший новичок, сейчас пытаюсь изучить его для самостоятельного написания сервисов для своих мобильных приложений, поэтому материал для меня жутко интересный. Думаю, что принцип разработки спокойно перекладывается и на TypeScript/Coffee, тут кому что больше нравится, но вот про котэ-библиотеку надо почитать подробнее, какие именно возможности она дает. Спасибо за статью!

Ответить

 **ElianL**  30 мая 2017 в 20:43 # 📖

↑ +2 ↓

Не силен в микросервисах, да и в бекенде тоже... Но разве микросервисы не подразумевают, что любой микросервис в системе может быть написан на любой технологии и языке? И пока нет реализации cote на других языках и платформах, использовать его для создания большой системы будет проблематично?

А в целом за статью спасибо. Интересно

Ответить

 **YuryZakharov** 31 мая 2017 в 12:00 # 📖 🗨️ ⬆️

↑ 0 ↓

Но разве микросервисы не подразумевают, что любой микросервис в системе может быть написан на любой технологии и языке?

Подразумевают. Но не обязаны.

использовать его для создания большой системы будет проблематично?

Нет, если не ставить себе хипстерских целей.

Ответить

 **dos**  30 мая 2017 в 22:41 # 📖

↑ +1 ↓

Очень похоже на <http://senecajs.org/> — есть информация о том, что лучше/хуже?

Ответить



mytz 31 мая 2017 в 07:09 # 📖

↑ 0 ↓

Спасибо, открыл для себя котЭков :-). Вызвало некоторые ассоциации с вот этим

Решил погонять, хотел ради эксперимента связать SocketCluster (принимает нагрузку от фронтенда + аутентификация + "роутинг" к микросервисам и обратно) + cote (собственно, куча всяких микросервисов, которые по котэвски общаются между собой), но... с горяча что-то не заводится, ск воркеры падают при попытке что-нибудь сделать с котэ (например, запаблишить в котэ при новом подключении к СК). Разбираться где я накосячил с СК лень, котэкодить понравилось. В общем, буду присматриваться.

Сам микросервисы "пользую". Если говорить о технологиях, то у меня джентельменский набор состоит из Docker, NATS / NSQ, Socketcluster, Traefik, Go / Node.

PS: **тру микросервисы на ноде :-)**

Ответить



Movimento5Litri 31 мая 2017 в 11:46 # 📖

↑ 0 ↓

Зачем лепить микросервисы во все поля?
Это же просто очередной buzzword...

Ответить









para3um 31 мая 2017 в 12:36 # 📖 ⌨️ ⬆️

↑ +1 ↓









Это инструмент для конкретных задач (точнее, конкретных требований к решению). Во все поля, конечно, не нужно, но там, где нужно — без них не получится (точнее, по естественным причинам решение выльется в микросервисную архитектуру, даже если архитектор не знал таких слов).

Ответить

 **freeart** 1 июня 2017 в 12:26     +1 









Не нашел ни строчки про согласованность данных в этой платформе. Мне показалось что ее нет (смотрю в исходниках и ничего такого) и автор предполагает что все сервисы будут иметь актуальные данные сами собой. Вот как он сделал на распределенной системе, без консенсуса, проведение оплаты <https://github.com/dashersw/cote-workshop/blob/master/services/payment-service.js> Атакуем систему сотней запросов на покупку товара за 5\$. Все микросервисы «payment» распределяют нагрузку и получают запрос на проведение оплаты, каждый запросит мой баланс 100\$ и каждый сделает списание 100\$ — 5\$ перетерев результаты другого. В итоге я имею 100 товаров, купленных за 5\$, вместо 500\$ (утрирую, но примерно так и будет)

Ответить

 **nara3um** 1 июня 2017 в 13:30       -1 








Нет, так не будет, ибо точка истинности в платёжной системе находится не в микросервисах, а в транзакционном персистентном хранилище. Грубо говоря, только один сервис вернёт ОК, а другие вернут сообщение об ошибке. Микросервисность не отменяет программирования непротиворечивой бизнес-логики.

Ответить

 **freeart** 1 июня 2017 в 14:07       +1 

В нашем случае в исходниках видно что хранилище postgresql, т.е. транзакционное персистентное хранилище. И как это хранилище уберезет нас от такой атаки? Никак. Защита от такой атаки либо журнал, либо stateful подход, либо консенсус между всему нодами, так мы получим согласованные данные. А решение этой защиты лежит на вышеупомянутых «зло» продуктах zookeeper, etcd, consul (apache ignite, cassandra и тд) против которых настроен автор. К чему я: автор написал peer2peer месенджер с автопоиском пиров и не более. А статья преподносится будто это панацея какая-то: «отказываемся от всего, используем мое решение».

Ответить

 **nara3um**  1 июня 2017 в 14:12      0 

> И как это хранилище уберезет нас от такой атаки? Никак.

Почему никак? Любая транзакция о покупке товара будет гарантировать, что деньги списаны, а товар начислен (или ошибку в случае невозможности это сделать).

> *Защита от такой атаки либо журнал, либо stateful подход, либо консенсус между всеми нодами, так мы получим согласованные данные.*

В чём заключается «атака»? Если пользователь послал 100 запросов на покупку товаров, то именно 100 раз мы и должны попытаться осуществить интересующую его операцию, разве нет? «Stateful подход» и «консенсус» в данном случае и заключается в том, что мы обращаемся к состоянию персистентного хранилища для выполнения операции, изолируя её SQL-транзакцией (а в более сложных системах могут пригодиться сервисы распределённых транзакций, например).

Ответить



freeart



1 июня 2017 в 14:28



0



Если пользователь послал 100 запросов на покупку товаров, то именно 100 раз мы и должны попытаться осуществить интересующую его операцию, разве нет

Именно что должны, но *cote* этого архитектурно не может предоставить. Эта библиотека сделает именно как я описал.

Почему никак?

Вы точно исходники смотрели? Я говорю о коде который написал автор и который не работает. Там очень небольшой объем кода. Там из базы происходит чтение баланса в память и операция проводится над числом в памяти. Потом базе отдается число на запись.

Ответить



пара3um



1 июня 2017 в 14:31



-1



> *Именно что должны, но cote этого архитектурно не может предоставить. Эта библиотека сделает именно как я описал.*

Эта библиотека не об этом вообще. Транзакционность перпендикулярна микросервисности.

> *Вы точно исходники смотрели?*

Нет, не смотрел. Думаю, это просто концептуальный пример (да, получается, что кривоватый, если чтение, обработка и запись данных в БД выполняются не в рамках одной SQL-транзакции).

Ответить



freeart 1 июня 2017 в 14:34 # 📖 ↻ ⬆

↑ +1 ↓

Существует множество способов демонстрации реализации архитектуры микросервисов, и, на самом деле, основательно проработанный пример — это всё, что нужно опытному разработчик

и ссылка именно на этот код. Для меня «основательно проработанный пример» что-то да значит.

Ответить



para3um 1 июня 2017 в 14:35 # 📖 ↻ ⬆

↑ 0 ↓

Да, с такой претензией спорить сложно :).

Ответить

Вы не можете комментировать эту публикацию

Вы можете комментировать публикации, которые не старше 30 дней, а также те, под которыми уже опубликован хотя бы один ваш комментарий. Вы не можете комментировать публикацию, если другой ваш комментарий к этой публикации еще не прошел проверку.

- Сутки
- Неделя
- Месяц

Как «Ревущий Котёнок» с Reddit заработал 28.500% на акциях GameStop: объясняю простым языком

+178

47,3k

181

251

Рабочее место на 0,5 м2

+46

14,1k

23

61

В чём главные проблемы Intel

+26

21,6k

34

31

Как фотка в портфолио влияет на получение работы и заказов. Обзор исследований

+56

19,6k

77

58

Где хотят жить читатели Хабра. Подводим итоги опроса

Мегапост

Ваш аккаунт	Разделы	Информация	Услуги
Профиль	Публикации	Устройство сайта	Реклама
Трекер	Новости	Для авторов	Тарифы
Диалоги	Хабы	Для компаний	Контент
Настройки	Компании	Документы	Семинары

[ППА](#)[Пользователи](#)[Соглашение](#)[Мегапроекты](#)[Песочница](#)[Конфиденциальность](#)[Мерч](#)

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.

© 2006 – 2021 «**Habr**»

[Настройка языка](#)[О сайте](#)[Служба поддержки](#)[Мобильная версия](#)