ОБЕСПЕЧЕНИЕ ПРЯМОЙ И ОБРАТНОЙ СОВМЕСТИМОСТИ СЕРВИСОВ И ДАННЫХ В АРХИТЕКТУРЕ ПРИЛОЖЕНИЙ - СТРАТЕГИИ И ЛУЧШИЕ ПРАКТИКИ

Шевцов А.С.

Шевцов Алексей Сергеевич - ведущий инженер компании,
Material Bank,
Boca Raton, Florida, USA

Аннотация: *архитектура микросервисов становится все более популярной благодаря множеству преимуществ, которые она предлагает: модульность, масштабируемость, изоляция отказов и возможность независимого развертывания сервисов. Однако этот стиль архитектуры не лишен своих сложностей. Одной из наиболее важных задач является поддержка обратной и прямой совместимости при развитии сервисов.*

В экосистеме микросервисов одни сервисы часто должны взаимодействовать с разными версиями других сервисов. По мере роста и развития вашего приложения, поддержание совместимости между разными версиями может стать сложной задачей. В этой статье мы рассмотрим стратегии и лучшие практики обеспечения обратной и прямой совместимости, что позволит обеспечить бесперебойное взаимодействие и гладкие развертывания без простоев.

Ключевые слова: *архитектура, высоконагруженные информационные системы, шаблоны проектирования информационных систем, обратная совместимость, прямая совместимость.*

ENSURING FORWARD AND BACKWARD COMPATIBILITY OF SERVICES AND DATA IN APPLICATION ARCHITECTURE - STRATEGIES AND BEST PRACTICES

Shevtsov A.S.

Shevtsov Alexey Sergeevich - leading engineer of the company,
MATERIAL BANK,
BOCA RATON, FLORIDA, USA

Abstract: *microservices architecture is becoming increasingly popular due to the many advantages it offers: modularity, scalability, fault isolation and the ability to independently deploy services. However, this style of architecture is not without its difficulties. One of the most important tasks is to support backward and forward compatibility in the development of services.*

In an ecosystem of microservices, some services often have to interact with different versions of other services. As your application grows and develops, maintaining compatibility between different versions can become a challenge. In this article, we will look at strategies and best practices for backward and forward compatibility, which will ensure smooth interaction and smooth deployments without downtime.

Keywords: *architecture, highly loaded information systems, information system design patterns, backward compatibility, forward compatibility.*

Разница между прямой и обратной совместимостью

Обратная совместимость гарантирует, что новые версии сервиса смогут взаимодействовать со старыми версиями. Это дизайн, совместимый с предыдущими версиями самого себя. В системе с обратной совместимостью клиенты, созданные для работы со старой версией, все еще будут функционировать корректно при взаимодействии с новой версией.

Прямая совместимость, с другой стороны, подразумевает, что система может обрабатывать входные данные, предназначенные для будущей версии самой себя. Это означает, что старый сервис все еще может эффективно взаимодействовать с новым.

Поддержание этих совместимостей критически важно в среде микросервисов. Разные команды, работающие с разной скоростью, могут независимо развертывать свои сервисы, что приводит к смешению активных версий сервисов в любой данный момент времени.

Примеры известных технологий, в которых были заложены принципы совместимости

Многие широко применяемые технологии разработаны с большим фокусом на поддержание обратной или прямой совместимости. Несколько примеров:

1. **HTML**: HTML - это яркий пример технологии, которая уделяет большое внимание обратной совместимости. Новые версии HTML, такие как HTML5, ввели новые теги и функции, которые не нарушают работу старых браузеров. Браузеры, которые не понимают определенный HTML-тег, просто игнорируют его и переходят к следующему тегу, что гарантирует отображение веб-страницы.

2. **SQL**: SQL, язык для взаимодействия с базами данных, остается в значительной степени обратно совместимым во всех его версиях. Это означает, что вы можете запустить SQL-запрос, написанный 30 лет назад, и он все еще будет работать на современных SQL-базах данных. Однако каждая система управления базами данных (СУБД) может иметь свой специфический диалект SQL с дополнительными функциями.

3. **JavaScript/ECMAScript**: Спецификация ECMAScript, которую реализует JavaScript, видела множество обновлений на протяжении многих лет, добавляя функции, такие как стрелочные функции, промисы, async/await и другие. Эти изменения разработаны так, чтобы быть обратно совместимыми и не нарушать работу веб-сайтов. Старый код JavaScript может работать в современных браузерах, а новые функции могут игнорироваться или рассматриваться как обычные объекты в старых браузерах, которые их не поддерживают.

4. **Docker**: Docker, популярная платформа контейнеризации, поддерживает обратную совместимость в своем инструменте Docker Compose. Новые версии Docker Compose совместимы с предыдущими версиями формата файла Docker Compose, что позволяет старым файлам работать без изменений.

5. **RESTful API**: RESTful API часто акцентируют внимание на обратной совместимости. Изменения и улучшения API производятся таким образом, что старые клиенты все еще могут функционировать корректно.

Прямую совместимость сложнее достичь из-за неизвестности о будущих версиях системы. Однако форматы, такие как **XML** и **JSON**, в определенной степени обладают прямой совместимостью. Можно добавить новые поля в сообщение XML или JSON, и если приложения, читающие сообщение, еще не поддерживают их - они могут просто игнорировать нераспознанные поля.

В мире аппаратного обеспечения - **USB** - это еще один пример. Например, порты USB 3.0 были разработаны для подключения устройств USB 2.0, а порты USB 2.0 могут подключаться к устройствам USB 3.0, хотя они будут работать на скорости 2.0.

Микросервисы и совместимость

Микросервисы взаимодействуют друг с другом через четко определенные API, хранят данные в различных базах данных и могут взаимодействовать асинхронно через события. Поддержание совместимости между этими интерфейсами и форматами может быть сложной задачей, особенно учитывая темп, с которым эти сервисы могут развиваться.

Совместимость нужно учитывать для каждого аспекта сервиса: его публичного API, схемы базы данных и формата событий, которые он производит или потребляет. Изменения в любом из этих элементов могут потенциально нарушить совместимость, воздействуя на другие сервисы и общее здоровье приложения.

Стратегии достижения и поддержания совместимости

Версионирование API

Версионирование API - это стратегия, используемая для поддержания совместимости при внесении изменений в API сервиса. Изменения в API могут быть различными, включая добавление новых методов, модификацию существующих или удаление устаревших. Не каждое изменение обратно совместимо, поэтому поддержка различных версий API помогает управлять этими изменениями без нарушения взаимодействия сервисов.

Существуют различные способы версионирования API, такие как использование адреса (пути или параметров в нем), заголовков или самих данных. Независимо от выбранного вами метода, цель состоит в том, чтобы одновременно могло сосуществовать несколько версий ваших API. Старые сервисы могут продолжать использовать старые версии API, а новые сервисы - обновленные версии.

```
app.get('/v1/users', (req: Request, res: Response) => {
    res.json({
        message: "Welcome to version 1 of the users API!",
        users: [{ id: 1, name: 'John Doe' }]
    });
});

app.get('/v2/users', (req: Request, res: Response) => {
    res.json({
        message: "Welcome to version 2 of the users API!",
        users: [{ id: 1, name: 'John Doe', email: 'john.doe@example.com' }]
    });
});
```

Рис. 1. Версионирование метода API.

В этом примере у нас есть две версии метода "users". Первая версия (***/v1/users***) возвращает список пользователей с их id и именем. Позже мы решили включить почтовые адреса пользователей в ответ, поэтому мы создали новую версию (***/v2/users***).

Клиенты, которые были созданы для работы с версией 1 API, могут продолжать взаимодействовать с ***/v1/users***, в то время как новые клиенты могут использовать дополнительные данные, предоставляемые версией 2, взаимодействуя с ***/v2/users***.

Таким образом, мы можем развивать наш API, не нарушая работу существующих клиентов. В этом суть поддержания обратной совместимости и цель версионирования API.

Эволюция базы данных

Так же как ваши API эволюционируют, так будут эволюционировать и схемы вашей базы данных. Однако прямые изменения схем могут нарушить работу ваших сервисов. Один из подходов к развитию схемы вашей базы данных с сохранением совместимости - это паттерн "расширение/сокращение" (expand/contract pattern).

На этапе расширения вы вносите изменения, обратно совместимые. Например, вы можете добавить новый столбец в таблицу, но ваш существующий код сервиса еще не использует этот столбец. После тестирования и убеждения в том, что все работает как ожидалось, вы затем обновляете свои сервисы, чтобы начать использование нового столбца, что отмечает конец этапа расширения.

На этапе сокращения вы удаляете устаревшие элементы схемы базы данных, которые ваши сервисы больше не используют. И снова, это делается обратно совместимым образом, обеспечивая непрерывность работы сервисов.

Для примера, рассмотрим приложение, использующее таблицу `users` со следующими данными:

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);
```

Рис. 2. Создание таблицы users.

Предположим, что вашему приложению нужно начать отслеживать дату создания каждого аккаунта. Вы можете добавить поле created_at в таблицу пользователей с помощью оператора ALTER TABLE:

```
ALTER TABLE users
ADD COLUMN created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
```

Рис. 3. Добавление колонки про время создания в таблицу users.

Используя значение по умолчанию `CURRENT_TIMESTAMP`, существующие строки в таблице будут иметь поле `created_at`, установленное в момент выполнения оператора ALTER TABLE. Это позволяет вам поддерживать обратную совместимость с существующими данными, в то время как схема вашей базы данных развивается для поддержки новых функций.

Однако прямое ручное применение изменений к вашей рабочей базе данных рискованно. Более безопасный и грамотный подход - использовать инструмент миграции баз данных для управления изменениями вашей схемы базы данных. Это позволяет вам версионировать схему вашей базы данных, откатывать изменения, если что-то пошло не так, и применять изменения контролируемым образом.

Ниже приведен пример того, как вы могли бы написать эту миграцию с помощью инструмента наподобие Knex.js, который является конструктором SQL-запросов, поддерживающим миграции схем:

```
exports.up = function(knex: Knex): Promise<void> {
    return knex.schema.table('users', table => {
        table.timestamp('created_at').defaultTo(knex.fn.now());
    });
};

exports.down = function(knex: Knex): Promise<void> {
    return knex.schema.table('users', table => {
        table.dropColumn('created_at');
    });
};
```

Рис. 4. Описание миграции схемы данных “вверх” и “вниз”.

Функция up добавляет поле created_at в таблицу users, а функция down удаляет его. Это позволяет вам, при необходимости, откатить миграцию.

Форматы сообщений в Event-Driven системах

В Event-Driven архитектурах сервисы взаимодействуют асинхронно через события. Так же, как и с API и схемами баз данных, формат этих событий должен развиваться, сохраняя при этом совместимость.

Один из подходов - версионирование событий и включение номера версии в метаданные события. Когда сервис получает событие, он проверяет номер версии и соответствующим образом обрабатывает его. Этот метод требует, чтобы ваши сервисы были способны обрабатывать несколько версий событий, подобно тому, как они работают с несколькими версиями API.

Рассмотрим следующий пример с гипотетической библиотекой обмена сообщениями. Мы покажем базовую структуру полезной нагрузки события и то, как она может эволюционировать со временем без нарушения работы существующих клиентов.

Начнем с простого события о создании юзера UserCreated:

```
interface UserCreatedV1 {
    type: 'UserCreatedV1';
    payload: {
        id: string;
        name: string;
        email: string;
    };
}
```

Рис. 5. Описание события о создании юзера.

Событие UserCreatedV1 содержит базовую информацию о пользователе. Допустим, теперь мы хотим включить поле createdAt в полезную нагрузку события. Вот как мы могли бы сделать это обратно совместимым образом:

```
interface UserCreatedV2 {
    type: 'UserCreatedV2';
    payload: {
        id: string;
        name: string;
        email: string;
        createdAt: string;  // New field
    };
}
```

Рис. 6. Новая версия события о создании юзера.

Создав новую версию события, UserCreatedV2, мы можем добавить новые данные, не влияя на клиентов, которые ожидают только события UserCreatedV1.

Теперь давайте рассмотрим, как мы могли бы обрабатывать эти события в наших клиентах:

```
function handleEvent(event: UserCreatedV1 | UserCreatedV2) {
    switch (event.type) {
        case 'UserCreatedV1':
            handleUserCreatedV1(event.payload);
            break;
        case 'UserCreatedV2':
            handleUserCreatedV2(event.payload);
            break;
    }
}

function handleUserCreatedV1(payload: UserCreatedV1['payload']) {
    // Handle version 1 of the event
}

function handleUserCreatedV2(payload: UserCreatedV2['payload']) {
    // Handle version 2 of the event
}
```

Рис. 7. Обработка событий разных версий на клиенте.

Клиенты, которые знают, как обрабатывать события UserCreatedV2, могут использовать дополнительное поле createdAt, в то время как потребители, которые знают только, как обрабатывать события UserCreatedV1, могут продолжать функционировать, как и раньше.

Лучшие практики и техники

Тестирование контрактов

По мере роста и развития вашей системы, отслеживание взаимодействий между всеми вашими службами может стать довольно сложным. Тестирование контрактов - метод, который может помочь справиться с этой сложностью. При тестировании контрактов вы определяете контракт, который указывает, как должен вести себя API вашего сервиса, а затем проверяете, соответствует ли ваш сервис этому контракту.

Контракт может определять такие вещи, как название методов, которые предоставляет ваш сервис, принимаемые ими входные данные и данные ответов. Регулярно выполняя эти тесты вы можете убеждаться, что сервис продолжает вести себя так, как ожидается, даже после внесения изменений.

Пример теста контракта с фреймворком Pact:

```
describe('User API', () => {
  beforeAll(() => provider.setup());
  afterAll(() => provider.finalize());

  describe('get /users/:id', () => {
    beforeAll(() => {
      return provider.addInteraction({
        state: 'a user exists',
        uponReceiving: 'a request for a user',
        withRequest: {
          method: 'GET',
          path: '/users/1',
        },
        willRespondWith: {
          status: 200,
          body: {
            id: 1,
            name: 'John Doe',
            email: 'john.doe@example.com',
          },
        },
      });
    });

    it('sends a request according to the contract', async () => {
      const res = await axios.get('http://localhost:1234/users/1');
      expect(res.status).toBe(200);
      expect(res.data).toEqual({
        id: 1,
        name: 'John Doe',
        email: 'john.doe@example.com',
      });
    });
  });

  // Verify that the provider has met all the contract requirements
  afterAll(() => provider.verify());
});
```

Рис. 8. Тестирование контракта при помощи Pact.

Если сервер изменится и не будет соответствовать требованиям контракта, этот тест не пройдет, предупредив нас об несовместимости до того, как она вызовет проблемы в рабочем окружении.

Клиентские контракты

Клиентские контракты (Consumer-Driven Contracts) - это паттерн, при котором контракт составляется с точки зрения службы-потребителя. Он определяет, как служба-потребитель использует API службы-поставщика. Эти контракты передаются поставщику, который может использовать их для того, чтобы не вносить изменения, которые могли бы нарушить работу их потребителей.

В CDC каждый потребитель указывает, чего ожидает от поставщика, что позволяет нескольким потребителям определить свои контракты с одним и тем же поставщиком. По мере того как служба-поставщик развивается, она проверяет свои изменения на соответствие этим контрактам, чтобы гарантировать, что она не нарушит никаких существующих потребителей.

Фича-флаги

Функциональные переключатели, или фича-флаги, предоставляют способ изменить поведение системы без изменения ее кодовой базы. Они разделяют развертывание кода и выпуск функций, позволяя разработчикам развертывать код в продакшне, сохраняя при этом новые функции скрытыми до момента их готовности к выпуску.

Это может быть особенно полезно в среде микросервисов, где разные службы могут находиться на разных стадиях готовности функций. Функциональные переключатели позволяют вам постепенно внедрять функции, тестировать их в производстве с небольшим набором пользователей и быстро откатывать их, если найдены проблемы.

Постепенная деградация (Graceful Degradation)

Проектирование сервисов с прицелом на возможность постепенной деградации может улучшить устойчивость вашей системы в случае недоступности определенных функций или зависимостей. Этого можно достичь путем внедрения стратегий отката в ваши службы.

Например, если служба зависит от другой службы, которая в настоящее время не работает, вместо того чтобы полностью выйти из строя, она может переключиться на кэшированную версию данных, или вернуть значение по умолчанию, или сообщение об ошибке, которое не нарушает пользовательский опыт.

Пример - допустим, у вас есть микросервис "OrderService", который извлекает данные пользователя из "UserService" для обработки заказа. Если "UserService" недоступен, "OrderService" мог бы контролируемо деградировать, обрабатывая заказ без деталей пользователя.

Вот пример реализации этой логики на TypeScript с использованием Axios и паттерна "circuit breaker" с библиотекой opossum:

```
import axios from 'axios';
import CircuitBreaker from 'opossum';

// Фунцкия запроса информации о пользователе
const fetchUserDetails = async (userId: string) => {
  const response = await axios.get(`http://user-service/users/${userId}`);
  return response.data;
};

// Circuit breaker для функции выше
const options = {
  timeout: 3000, // If our function takes longer than 3 seconds, trigger a failu
re
  errorThresholdPercentage: 50, // When 50% of requests fail, trip the circuit
  resetTimeout: 10000, // After 10 seconds, try again
};
const breaker = new CircuitBreaker(fetchUserDetails, options);

// Функция-обработчик, если Circuit breaker сработал
breaker.fallback(() => ({
  message: 'User service is currently unavailable. Processing order without user
details.',
}));

const userId = '123';
breaker.fire(userId)
  .then(userDetails => console.log(userDetails))
  .catch(error => console.error(error));
```

Рис. 9. Пример использования circuit breaker в коде приложения.

В этом примере, если более 50% запросов на получение информации о пользователе проваливаются, circuit breaker (автоматический выключатель) срабатывает и начинает возвращать ответ от запасного сценария. Это позволяет "OrderService" продолжать обрабатывать заказы, даже когда "UserService" недоступен, эффективно справляясь с возникающими ошибками.

Также стоит отметить, что circuit breaker автоматически сбрасывается после определенного периода времени (в данном случае 10 секунд), после чего он снова попытается вызвать функцию `fetchUserDetails`. Если функция выполняется успешно, circuit breaker включается и снова начинает вызывать нужную функцию.

Сеть Сервисов (Service Mesh)

В микросервисной архитектуре сервисная сеть может значительно помочь в обеспечении совместимости и плавных релизов. Сервисная сеть - это специализированный слой инфраструктуры, который контролирует взаимодействие сервисов.

Она может предоставлять “из коробки” такие возможности, как балансировку трафика (для развертывания по методу "blue-green" или канареечных релизов), искусственное внедрение сбоев (для хаос-инжиниринга и тестирования) и автоматический выключатель (чтобы предотвратить каскадное распространение сбоев в вашей системе).

Заключение

Обеспечение обратной и прямой совместимости в среде микросервисов требует тщательного планирования, но преимущества надежной, устойчивой системы значительно превосходят затраты. Рассмотренные нами стратегии и методы, от версионирования API и эволюции базы данных до тестирования контрактов и постепенной деградации, могут помочь управлять сложностью и поддерживать бесшовное взаимодействие ваших сервисов.

Следуя этим стратегиям и передовым методам, вы замедлите рост сложности разработки нового функционала по мере роста вашей системы и поможете вашим командам создавать более надежное программное обеспечение, быстрее.

Важно помнить, что конкретные практики будут варьироваться в зависимости от уникальных требований и ограничений вашего приложения. Не существует универсального решения, и лучший подход - это тот, который лучше всего подходит вашей команде и вашему приложению.

Список литературы / References

1. *Martin Kleppmann* Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems // O'Reilly Media, 2017.
2. *Alex Xu* System Design Interview – An insider's guide // Independently published, 2020.
3. *Joe Reis* Fundamentals of Data Engineering: Plan and Build Robust Data Systems // O'Reilly Media, 2022.