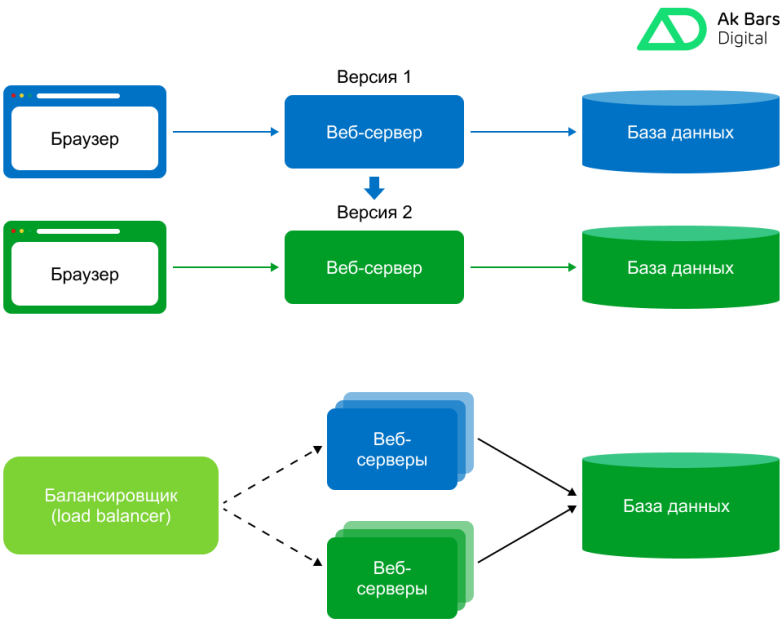
Blue-green deployment, canary release: рецепт приготовления безрисковых релизов

habr.com/ru/companies/abdigital/articles/668478

Банковские сервисы по умолчанию не должны падать и ложиться хоть на секундочку, даже (и в особенности) когда мы обновляемся. Ведь всего лишь какие-то секунды могут привести к потерям с множеством нулей. Чтобы этого не произошло мы используем blue-green deployment (BGD).

Простым языком, blue-green deployment — это способ развертывания, который позволяет обновлять приложения, не отклоняя ни одного запроса, без остановок. Как это сделать, расскажу и покажу на одном большом примере. Статья подойдет DevOps-инженерам и бэкенд-разработчикам, особенно на HighLoad-проектах, а также моим будущим коллегам, как методичка по безрисковым релизам, чтобы прод не падал каждые 2 недели по графику релизов (а такое тоже бывало). В статье будет минимум теории и максимум практики.



Дисклеймер: немного о «методичке»

В нашей компании постоянно растет количество сотрудников. Одна из моих функций, как Software Engineering Manager’a, повышение как роста технической и технологической зрелости ребят, так и качества продуктов, разрабатываемых командами. Поэтому мне постоянно приходится повторять тему про BGD.

Реакции бывают разные. Кто впервые слышит, что можно обновлять версии продуктов не останавливая обслуживание, часто восклицают: «А что, так можно было?!». Те же, кто пытается применять впервые на практике, неожиданно для себя

сталкиваются с проблемами при реализации и просят как можно подробнее освещать буквально каждый шаг в отдельности.

Поэтому я и решил изложить данный материал максимально подробно. Одна из целей статьи — сделать материал доступным для разработчиков любого уровня: от стажера до матерого синьора. Важно, чтобы каждый понимал, как это работает, чтобы осознанно и полноценно участвовать в разработке продуктов.

Большинство материалов про blue-green deployment не освещают тему достаточно полно, поэтому многие разработчики так и не берутся за реализацию. Поэтому считаю важным отметить, что в статье отражена «сквозная история» от API, до уровня базы данных.

Эта статья описывает подход, поэтому подойдет для любого языка программирования или ORM, которые, конечно, могут оказывать определенное влияние на реализацию, но приведенная методика все равно будет работать.

Некоторый код я поместил под спойлеры, чтобы он вас не отвлекал.

Как читать статью:

- Если вы впервые знакомитесь с темой и решили просто присмотреться, «а стоит ли оно того», предлагаю читать только заголовки и смотреть картинки-туториалы, все будет и так понятно.
- Если решите применить BGD на практике — стоит читать весь текст, да еще и раскрывать спойлеры.
- Для тех, кто использует C# и ORM Entity Framework Core, хорошие новости: я написал код примеров так, чтобы вы сразу смогли достичь нужного результата, используя метод copy-paste.

Отдельно хочу поблагодарить пользователя @poxvuibr. При просмотре записи доклада на митапе, в котором я описывал этот подход, он справедливо отметил неточность, которая могла привести к потере данных при определенных обстоятельствах. Эта ошибка стала еще одной причиной написания данной статьи — устранить указанный дефект и существенно дополнить материал.

Я обещал пример — поехали!

Пример приложения

Представим сферического заказчика в вакууме, который дает нам задание срочно сделать приложение по выбору лучшего работника месяца.

> Чтобы не отвлекаться на UI, это будет веб-API приложение.

Исходное состояние: v 1.0

Рассмотрим детально, как устроена программа v 1.0.

Модели.

▶ Модель API v 1.0 и модель EF Core слоя базы данных v 1.0 совпадают

Контекст базы данных.

▶ DbContext Entity Framework Core

Миграция v 1.0.

▼ Команды миграций
Создание миграции в Visual Studio (PowerShell):

```
Add-Migration InitialCreate
```

или в командной строке (.NET Core CLI):

```
dotnet ef migrations add InitialCreate
```

Выполнение миграции в Visual Studio (PowerShell):

```
Update-Database
```

или в командной строке (.NET Core CLI):

```
dotnet ef database update
```

Примечание. Далее я не буду указывать такие команды по созданию и выполнению миграций (см. документацию), только отмечать особенности.

В результате выполнения миграции будет создана база данных и таблица со столбцами.

▼ Скрипты TSQL
Примечание: для скриптов миграции указывается не сгенерированный код, а команды TSQL, которые будут фактически выполняться.

```
CREATE TABLE [dbo].[Persons](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] nvarchar NOT NULL,
    [LastName] nvarchar NOT NULL,
 CONSTRAINT [PK_Persons] PRIMARY KEY CLUSTERED
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Вставка данных приводит к такому виду запроса БД:

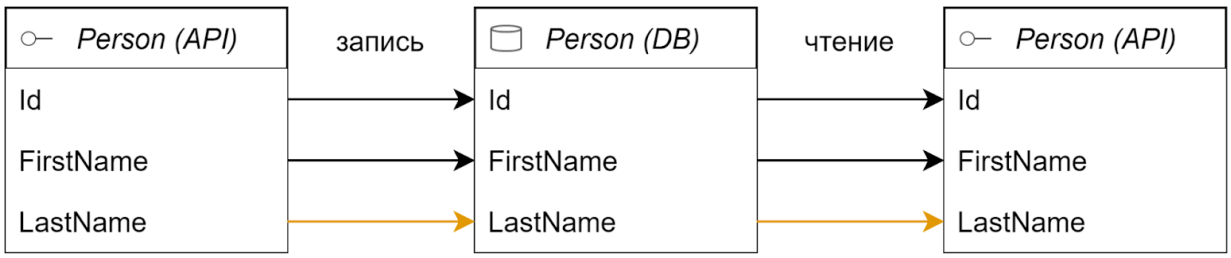
```
INSERT INTO Persons (FirstName, LastName) VALUES (‘Василий’, ’Ленцов')
```

Развертывание v 1.0

Примечание: в схемах развертывания я не буду отображать неизменяющиеся свойства `Id` и `FirstName`, чтобы сосредоточиться на самом важном.

Сопоставления с БД v 1.0

Версия приложения v 1.0, версия API v 1.0:



Сопоставление моделей API и таблицы БД v1.

Примечание: в таких подразделах я буду указывать схемы сопоставления моделей API и таблицы базы данных, вне зависимости от языка программирования, минуя реализацию бизнес-слоя и механизм используемой ORM.

Когда потребовались доработки…

Я мог бы придумать запрос на изменение каких-либо нетривиальных бизнес-требований, например выбрать именно лучшего работника, а не случайного, но в контексте статьи важнее понять сам принцип. Возьмём наглядный пример: потребуем переименование `LastName` в `Surname`.

Классическое решение

Чтобы нагляднее продемонстрировать преимущества предлагаемого метода, сначала рассмотрим, как проходит классический релиз, какие возникают проблемы при его развертывании, а затем — как от них избавиться при помощи blue-green deployment.

В классическом подходе мы переименовываем одно свойство модели на другое.

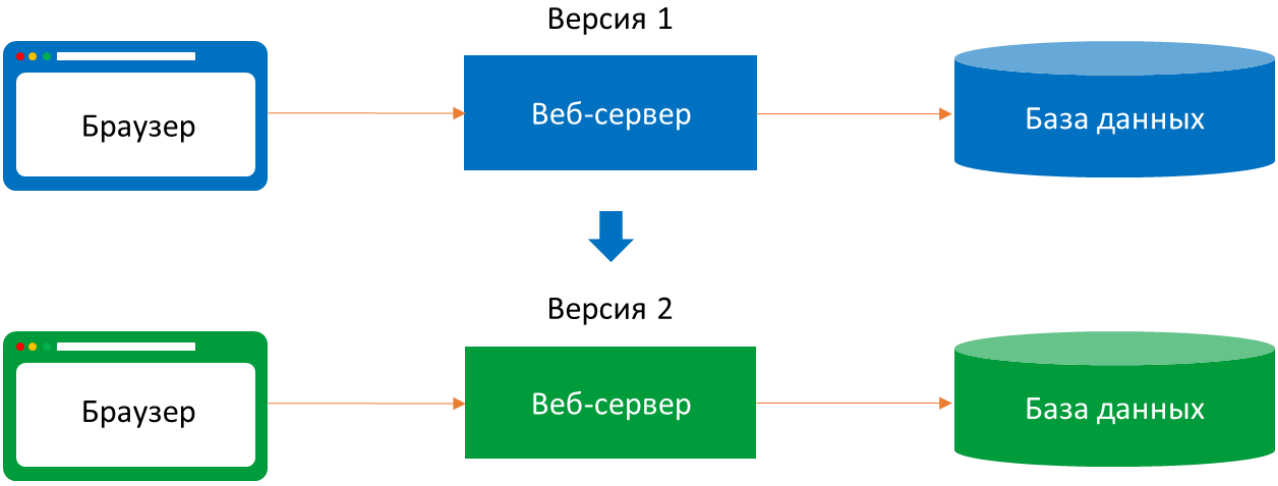
Модель.

▶ Модель API v 2.0 классического решения

Миграция на основе изменений.

▶ Скрипт TSQL

Развертывание. Простое классическое.

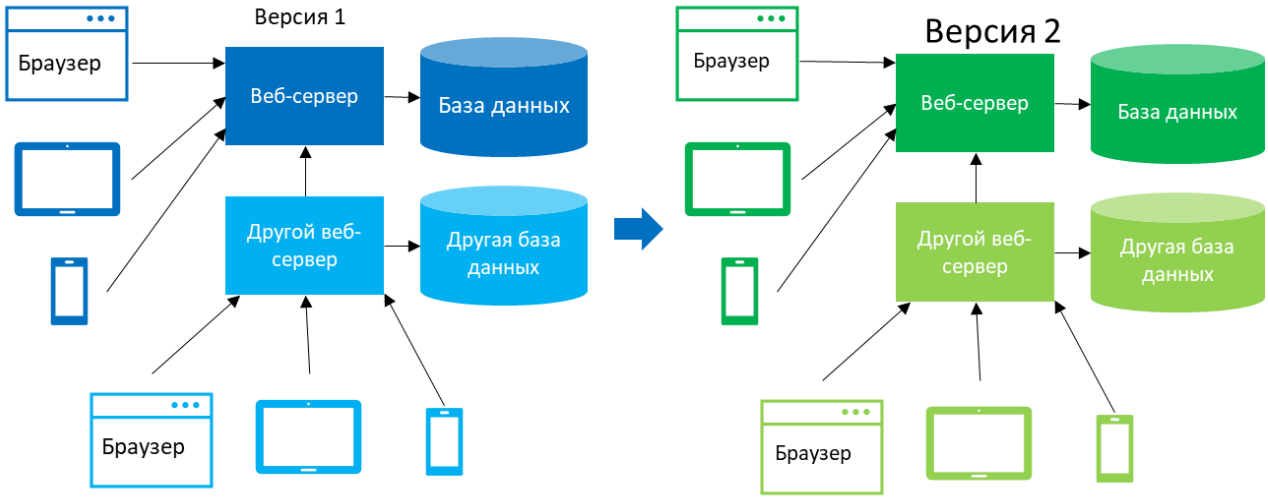


Все просто, все знакомо. Но… «Хьюстон, у нас проблемы»

Шесть проблем классического развертывания

У нас возникает проблема **остановки сервера для обновления**, если сервер простой. А если сложный, включающий в себя микросервисы, то добавляется еще и проблема **обновления версии сервера с зависимостями**.

Когда мы обновляемся с версии 1 на версию 2, все наши зависимости тоже должны обновиться на новую версию, использующую эту вторую версию.



Классическое развертывание с зависимостями.

Но если что-то пойдет не так, то возникает третья проблема — **отказ от новой версии сервера с зависимостями приводит к цепной реакции**. Нам придется откатиться полностью.

Но для отката нам нужна **резервная копия БД**, причем когда мы будем откатываться, **сервер опять будет простаивать**. Кроме того, полученные во время второй версии данные будут **безвозвратно утеряны**.

Все это делается для того, чтобы сократить время простоя и минимизировать связанные с этим потери. Но иронично, что серверы продолжают простаивать, убытки нарастают, а разработчики выясняют проблемы. Они придумывают костыли, пытаются чинить, чтоб хотя бы просто работало… И ладно бы все починилось с первого раза, но, увы, так не работает, и убытки продолжают нарастать.

Blue-green deployment и canary release

А теперь переходим к стратегии управления изменениями для выпуска ПО, у которой этих проблем нет. Но прежде, чем двинемся дальше, давайте четче определимся с используемой терминологией:

Сине-зеленое развертывание (blue-green deployment) — это метод внесения изменений в веб-сервер, приложение или сервер базы данных, путем замены чередующихся промышленных и промежуточных серверов.

Канареечный релиз (canary release) — это метод снижения риска внедрения новой версии в промышленную эксплуатацию путем предоставления изменения небольшому подмножеству пользователей с нарастанием до состояния, когда это изменение становится доступным для всех.

Несмотря на то, что сине-зеленое развертывание и канареечный релиз — это разные методы, я покажу, как оба варианта можно приготовить одним способом.

Как это работает

У нас здесь есть два условных веб-сервера — «синий» и «зеленый». Их может быть по одному, но чаще в финтехе они представлены группами. Изначально зелёные — тестовые, синие — рабочие. При этом желательно, чтобы их архитектура, конфигурация и окружение были максимально идентичными, чтобы обойтись без каких-либо побочных эффектов.

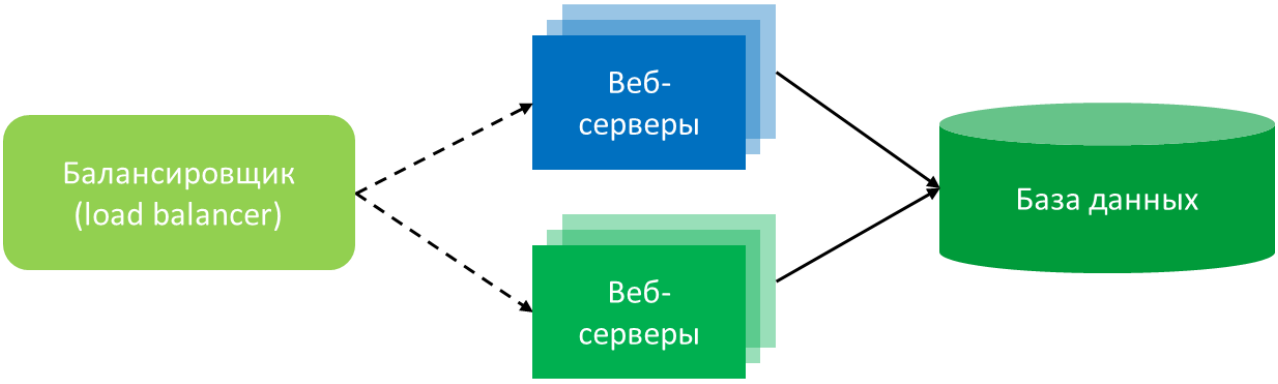
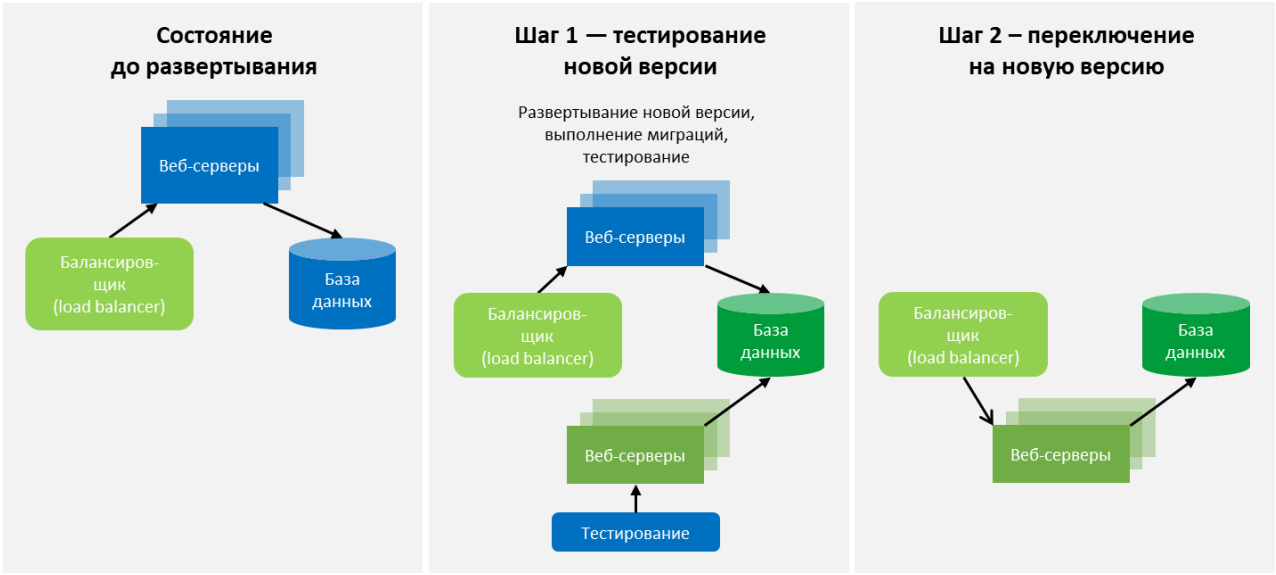


Схема blue-green deployment.

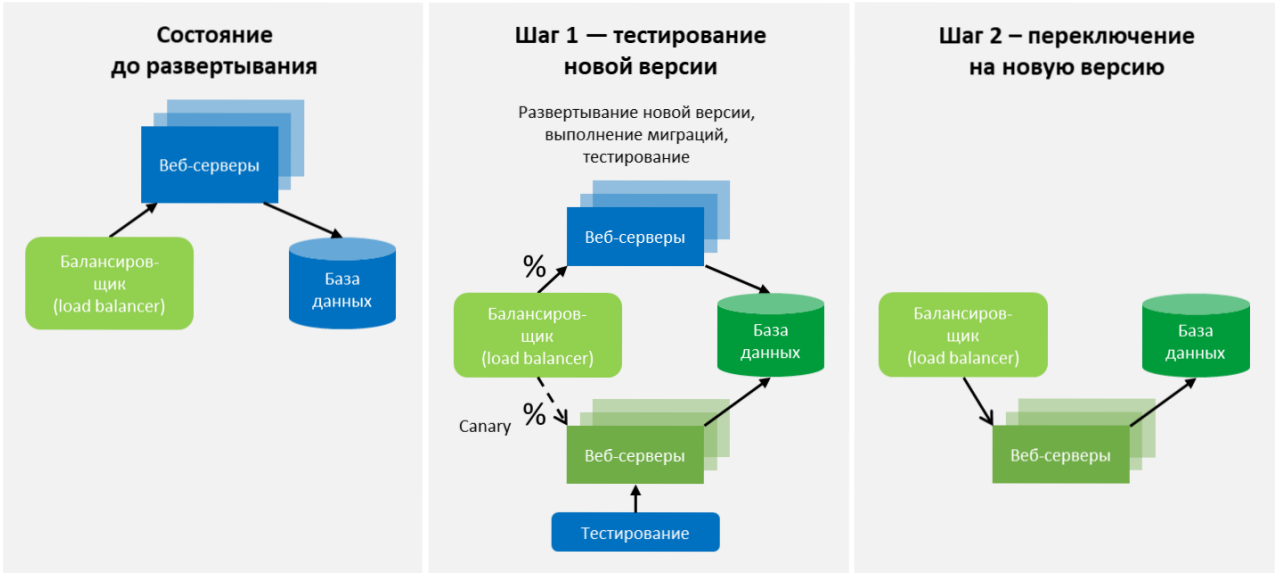
В исходном состоянии у нас в доступе синие веб-серверы и база данных.



Blue-green deployment по шагам:

- **Шаг 1**. Мы раскатываем зеленые серверы: размещаем на них новую версию.
- Накатываем миграции новой версии на базу данных, но по-прежнему у нас работают синие серверы на предыдущей версии. Проводим тестирование зеленых в штатном режиме.
- **Шаг 2.** Переключаем балансировщик на зеленые серверы. Если все хорошо — отключаем синие серверы.

Теперь взглянем на канареечный релиз.



Canary release по шагам.

Ничего сложного: все то же самое, но только после внутреннего тестирования или вместо него, мы балансировщиком постепенно распределяем нагрузку между синими серверами со старой версией и зелеными с новой, вплоть до 100%.

Рецепт приготовления

Пример с переименованием выбран не случайно — это достаточно сложный сценарий, влекущий за собой множество различных действий. Если приведенный далее материал не покажется чрезмерным для применения, то с реализацией остальных сценариев, требующих меньших усилий, проблем быть не должно.

Примечание для разработчиков на C#, использующих ORM Entity Framework Core: На первый взгляд может показаться, что реализация проста, но на основе массы проведенных экспериментов, могу уверенно сказать, что это не так. В EF Core есть ряд соглашений и различных настроек сопоставлений. Например, если объявить поле не так, как надо, или неверно прописать PropertyAccessMode при конфигурировании ModelBuilder, то результат может неприятно удивить. Из всех возможных вариантов я подобрал самый простой с минимумом кода и рекомендую использовать проверенный мной подход или же тщательно проверять результаты, если выберете иной способ реализации.

Примечание: все протестировано на .Net 6.0 и актуальной сейчас версии EF Core 6.0.5.

Этап 1: релиз приложения версии 2.0

На этом этапе добавляется API v 2.0, в котором модель соответствует целевой — включающей свойство `Surname` и без устаревшего свойства `LastName`. В базу данных добавляется поле `Surname`, необязательное для заполнения. В коде приложения используется только `LastName`, но не `Surname`.

Реализация модели слоя данных...

▶ ..этапа 1

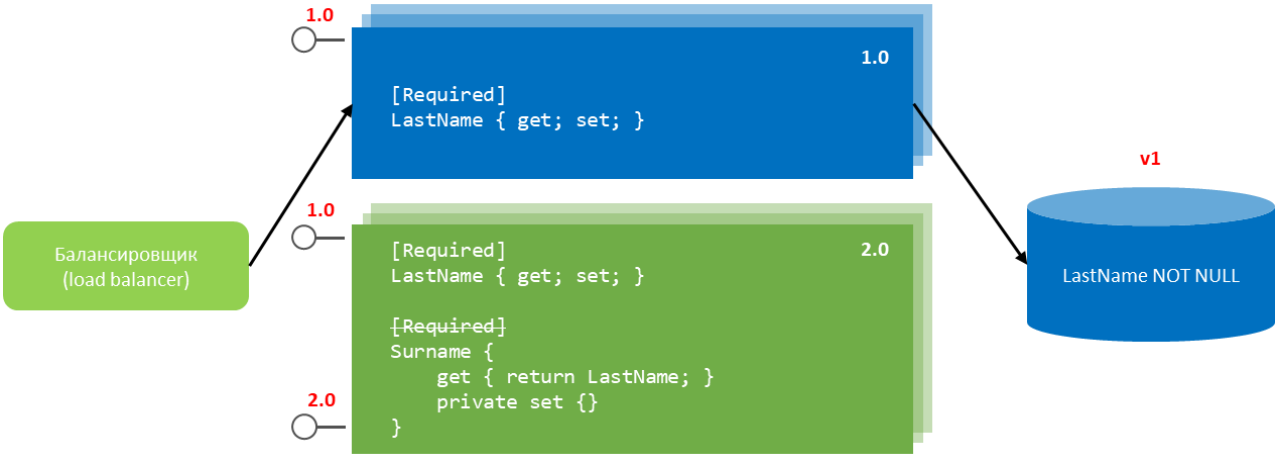
Миграция на основе изменений.

▶ Скрипт TSQL

Этап 1: развертывание v 2.0

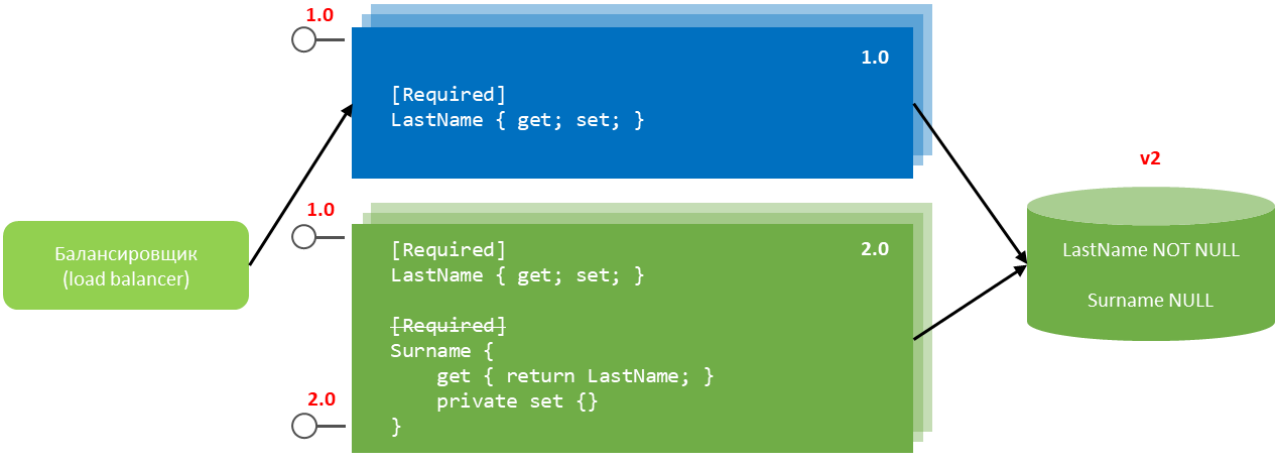
Рассмотрим подробно процесс развертывания новой версии.

Шаг 1. На прошлом этапе приложение версии v 1.0 было развернуто на группе синих серверов, поэтому новую версию v 2.0 разворачиваем на группу зеленых серверов. База данных находится в версии v 1.0.



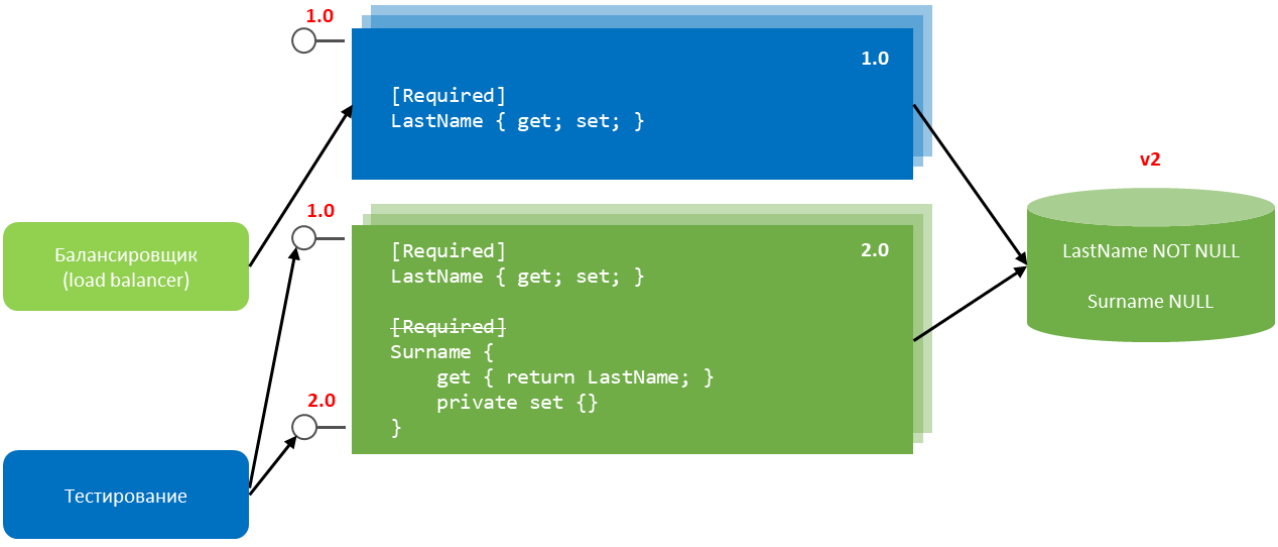
Развертывание приложения v 2.0

Шаг 2. Выполнение миграции, которое добавляет поле базы данных `Surname`, допускающее значения `null`.



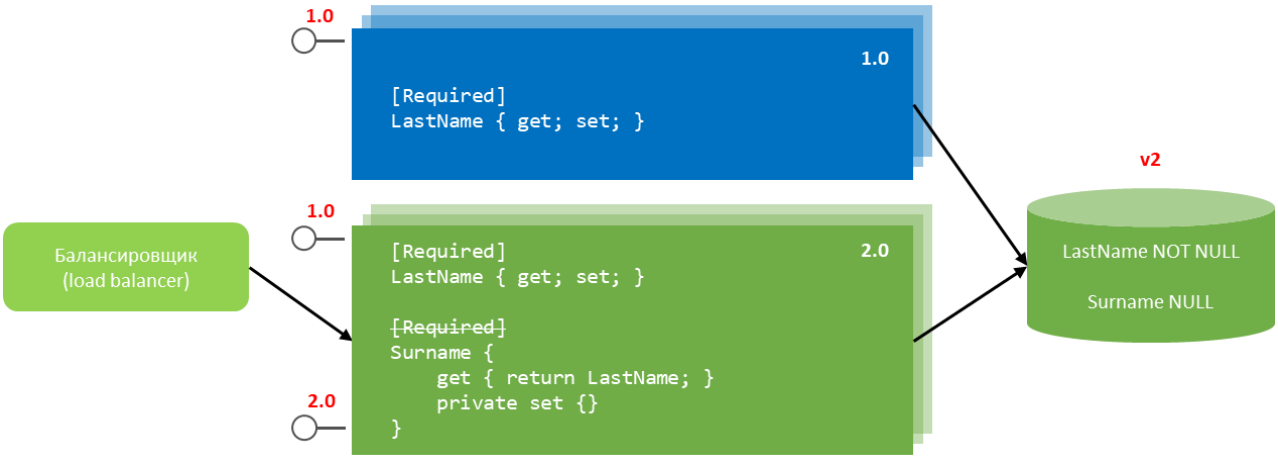
Развертывание приложения v 2.0. Миграция

Шаг 3. Теперь можно приступать к финальному тестированию новой версии v 2.0, которая поддерживает API v 1.0 и v 2.0 одновременно.



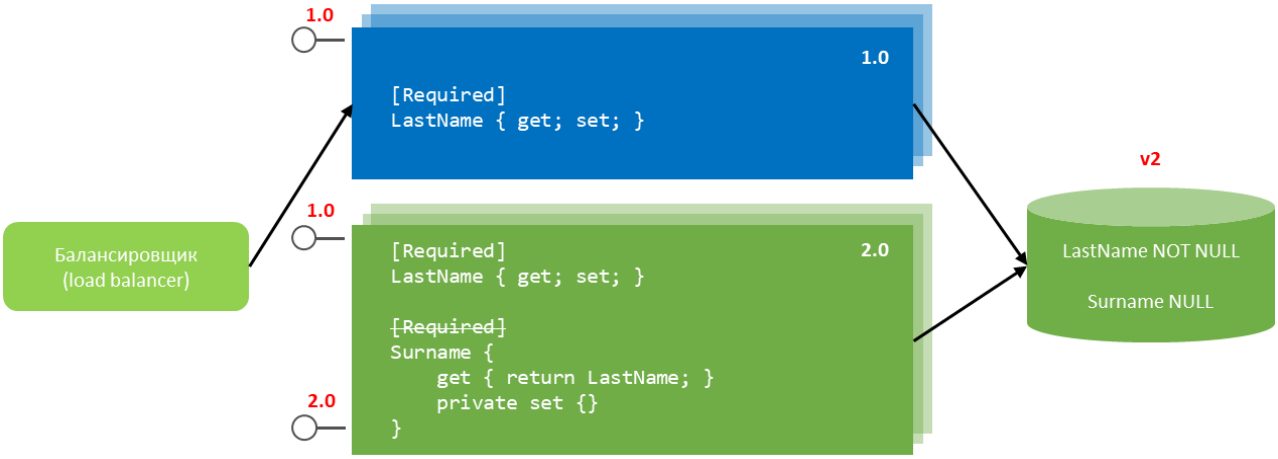
Развертывание приложения v 2.0. Тестирование

Шаг 4. Затем переключаем балансировщик. Это можно делать разом или долями — канареечный релиз в действии.



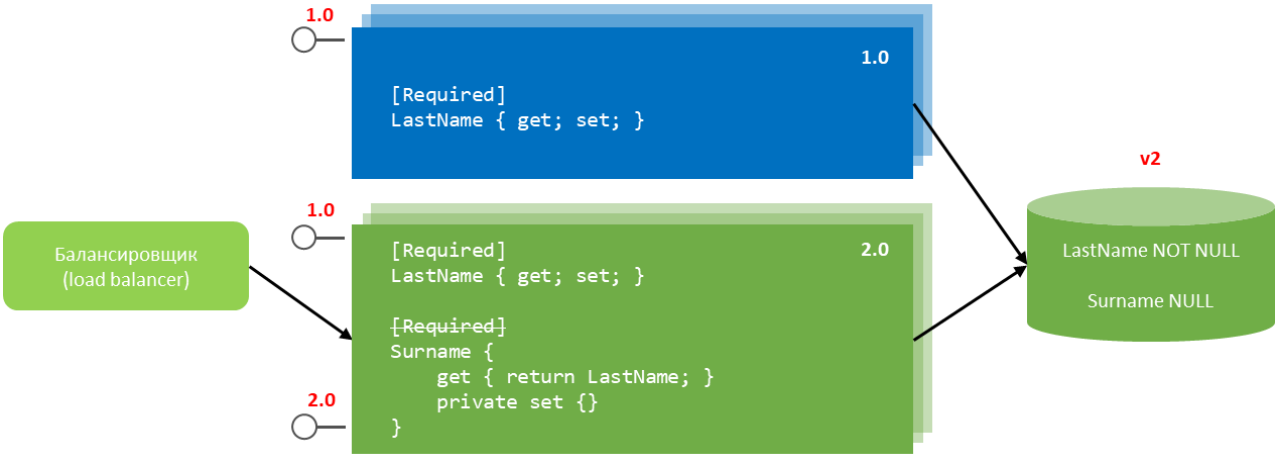
Развертывание приложения v 2.0. Переключение балансировщика

Если что-то пошло не так, то балансировщик переключает 100% нагрузку обратно на прежнюю версию приложения v 1.0, а версию v 2.0 дорабатывают, тестируют и снова разворачивают на зеленые серверы.



Развертывание приложения v 2.0. Обратное переключение балансировщика.

Шаги 1, 3 и 4 повторяются столько, сколько требуется, при этом выполнение миграции (шаг 2) повторять не имеет смысла. В итоге достигается стабильно работающая версия приложения v 2.0 и принимает 100% входящих запросов.



Развертывание приложения v 2.0. Стабилизация новой версии

С этого момента синяя группа серверов может использоваться для любых целей разработки или может быть отключена.

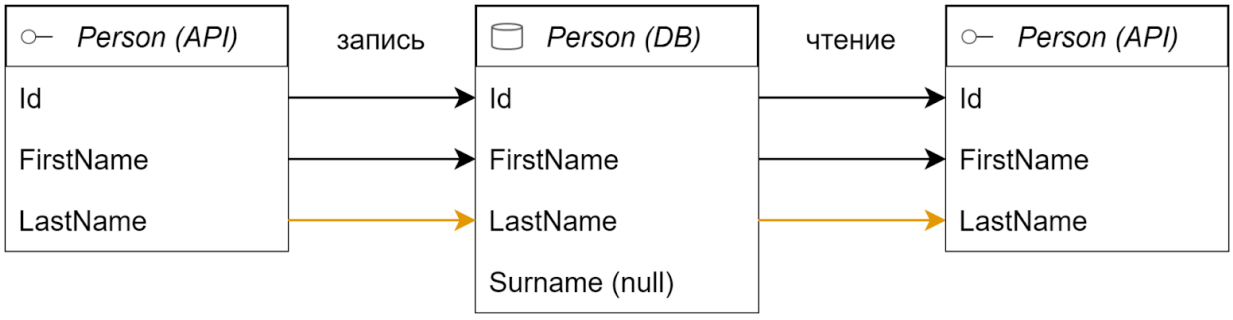
Потребители API v 1.0 уведомляются о необходимости перехода на новую версию API v 2.0. Это будет промежуточный этап, когда часть потребителей будет использовать API v 1.0, а другая часть уже перейдет на использование API v 2.0.

В рамках времени жизни этой версии приложения рекомендуется приравнять значения в колонке `Surname` соответствующими значениями из колонки `LastName`. Это можно делать разными способами, которые я не буду рассматривать в этой статье, а предложу позже лишь резервную версию такой трансформации.

Этап 1: сопоставления с БД v2.0

Посмотрим, как описанный процесс будет исполняться «под капотом» после выполнения миграции БД, которая в результате перейдет в версию v 2.0.

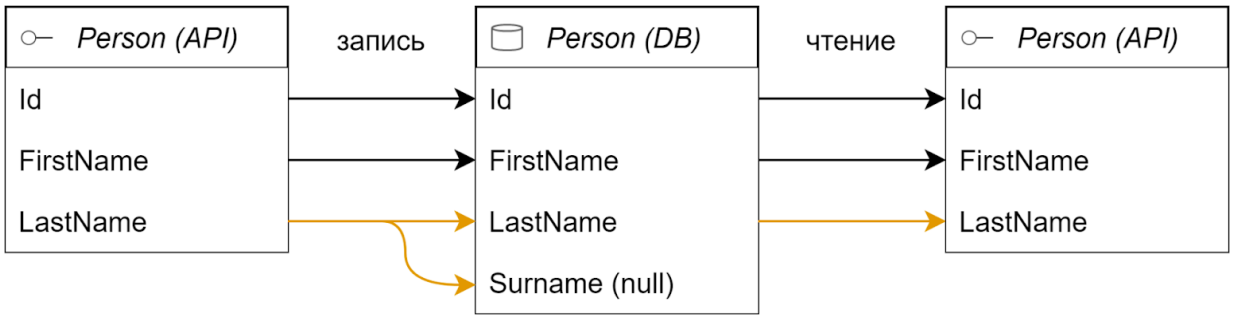
Версия приложения v 1.0, версия API v 1.0:



Сопоставление моделей API v 1.0 приложения v 1.0 и таблицы БД v2

Как видно приложение v 1.0 ничего «не знает» о поле `Surname` и оно приобретает значения `null`.

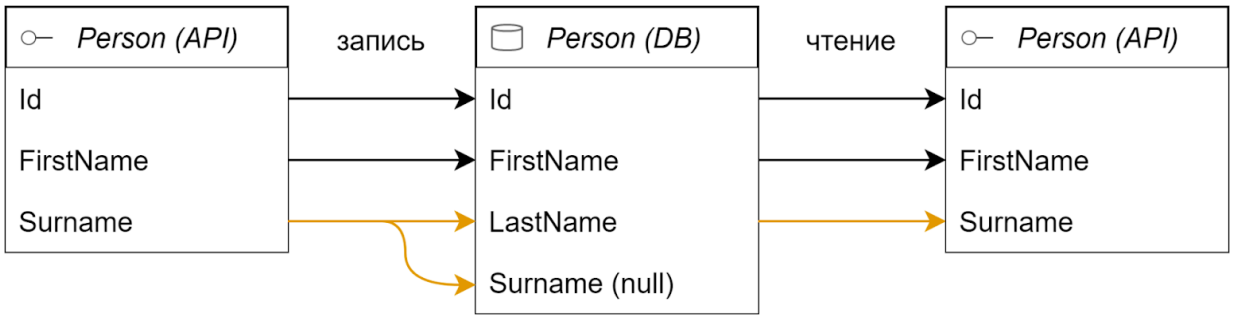
Версия приложения v 2.0, версия API v 1.0:



Сопоставление моделей API v 2.0 приложения v 1.0 и таблицы БД v2

В этом случае основное поле — `LastName`, а поле `Surname` — дублер.

Версия приложения v 2.0, версия API v 2.0:



Сопоставление моделей API v 2.0 приложения v 2.0 и таблицы БД v2

Здесь тоже поле `Surname` — просто дублер `LastName`.

Этап 2: релиз приложения версии 2.1

На этом этапе значения колонки `Surname` назначаются равными соответствующим значениям колонки `LastName`, если это не сделали раньше. Колонка `Surname` становится обязательной для заполнения, а вот `LastName` наоборот — будет необязательной, допускать значения `NULL`.

В результате поля меняются ролями: `Surname` будет основным, а `LastName` — дублером. В коде приложения заменяем использование `LastName` на `Surname`, `LastName` при этом помечаем устаревшим.

Если к этому моменту не все потребители переключились на API v 2.0, то мы можем продолжать поддержку API v 1.0 в этой версии. Рассмотрим именно такой случай, поскольку чаще всего такое переключение происходит не быстро.

Реализация модели слоя данных...

▶ ...этапа 2

Миграция.

▶ Корректировка миграции

▶ Скрипт TSQL

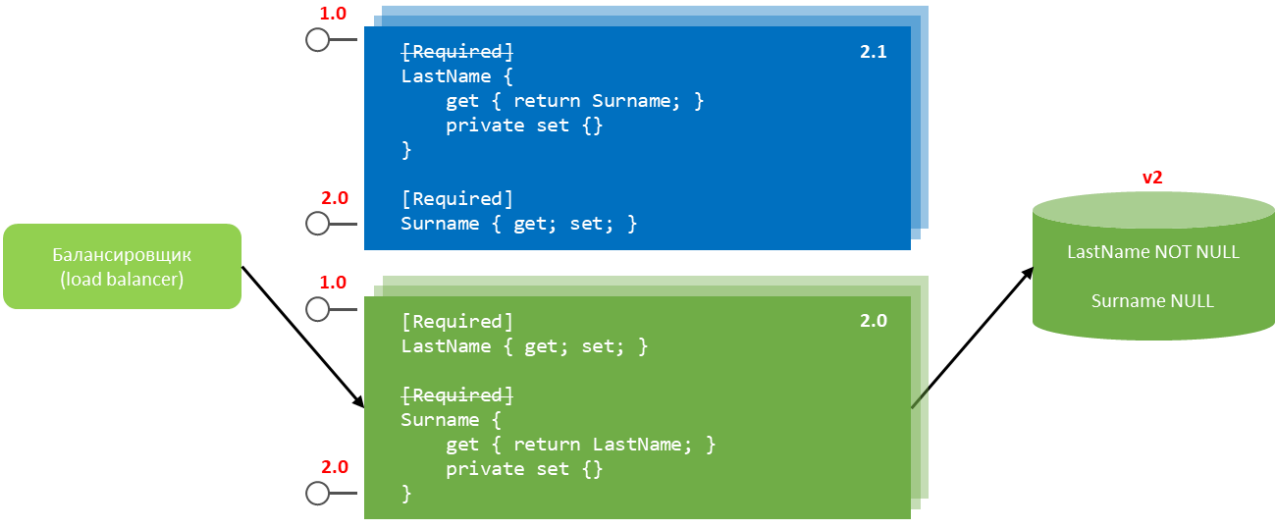
Этап 2: развертывание v 2.1

Поскольку при предыдущем релизе были использованы зеленые серверы, теперь новую версию будем развертывать на освободившиеся ранее синие серверы.

Шаги будут идентичными.

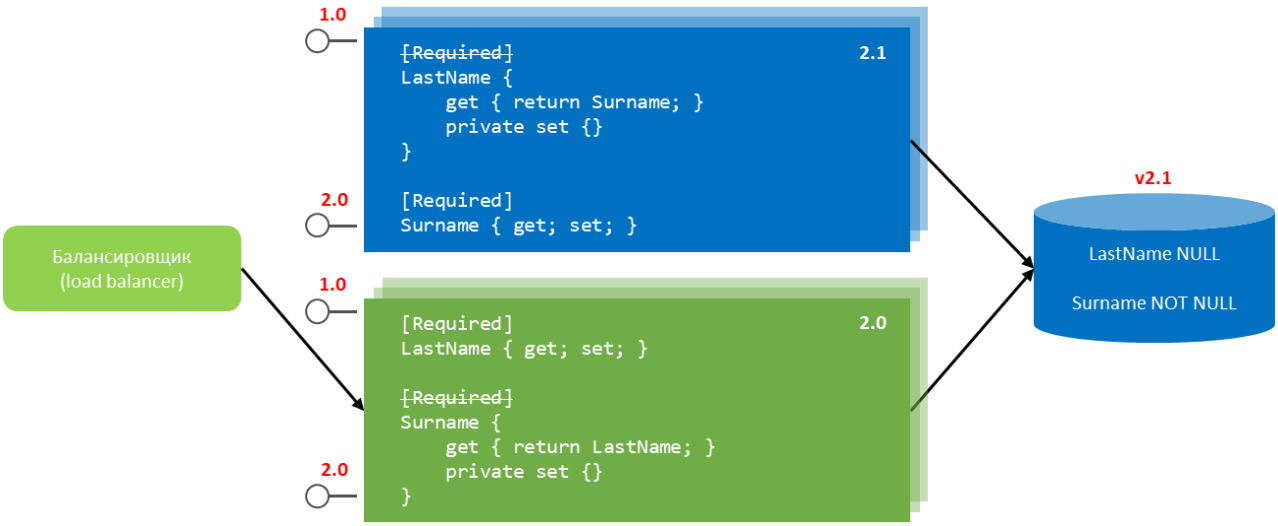
Промежуточные шаги развертывания v 2.1.

Шаг 1. Развертывание новой версии.



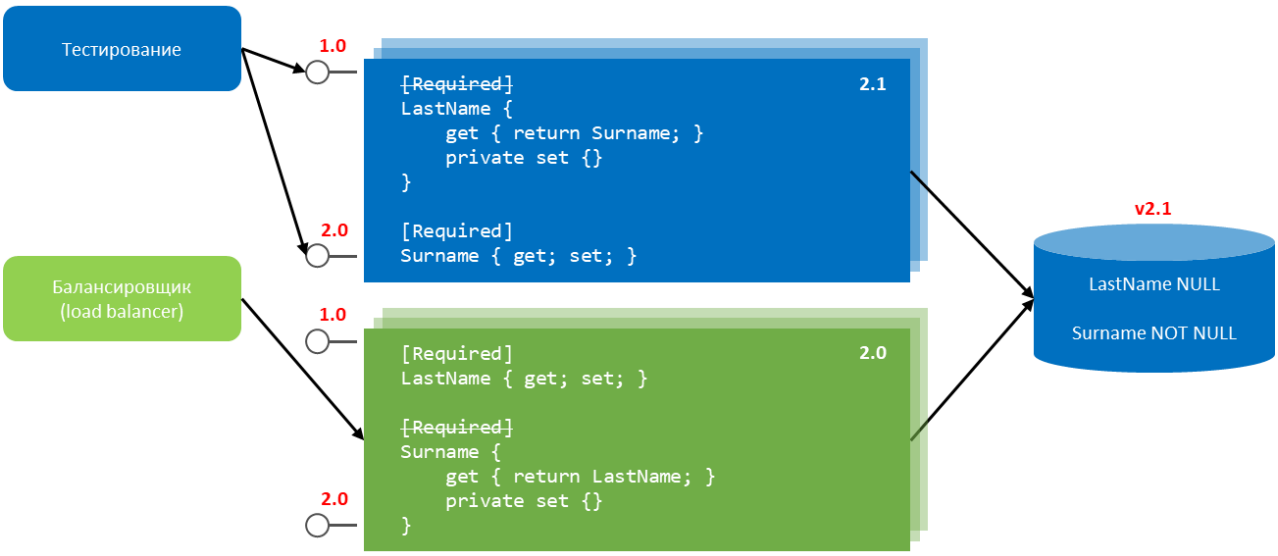
Развертывание приложения v 2.1

Шаг 2. Миграция.



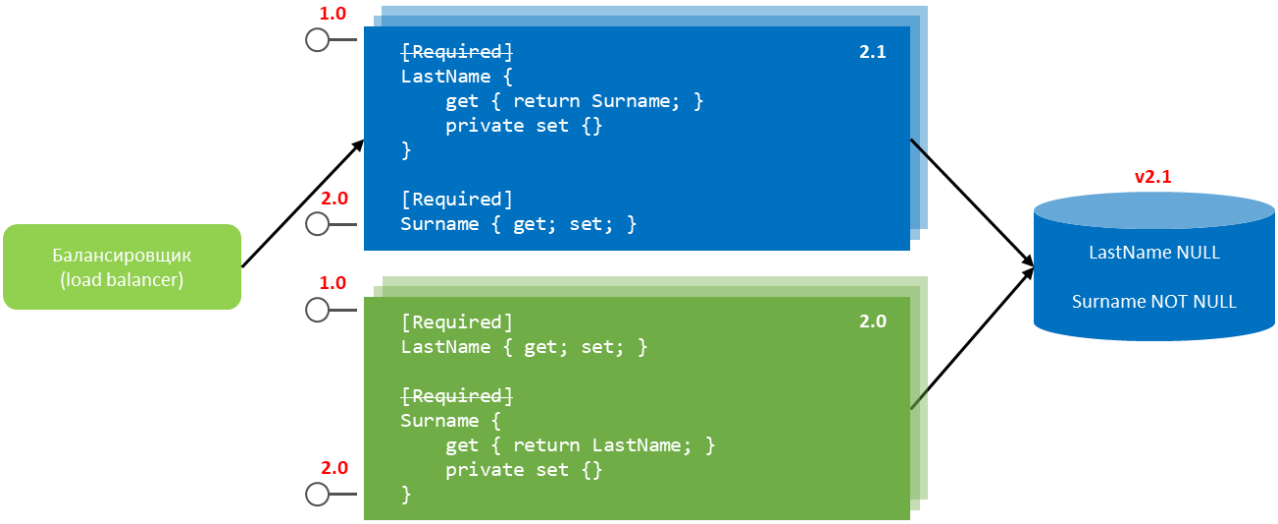
Развертывание приложения v 2.1. Миграция

Шаг 3. Тестирование.



Развертывание приложения v 2.1. Тестирование

Шаг 4. Эксплуатация приложения версии 2.1



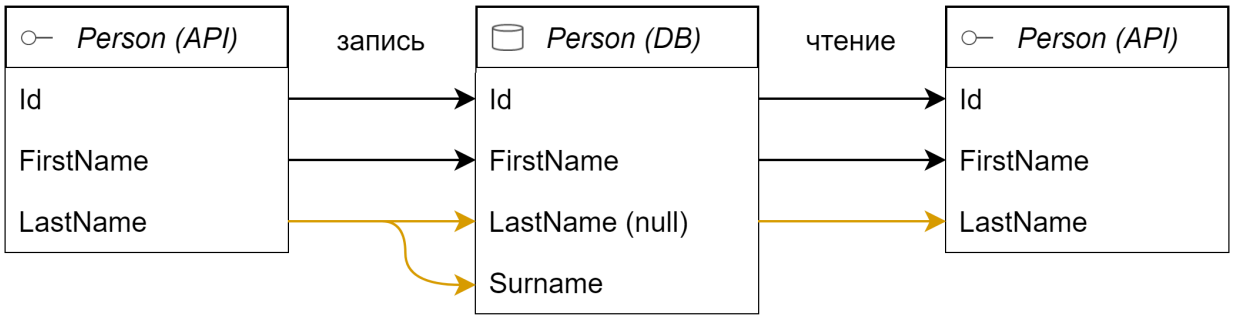
Развертывание приложения v 2.1. Переключение балансировщика

Этап 2: сопоставления с БД v2.1

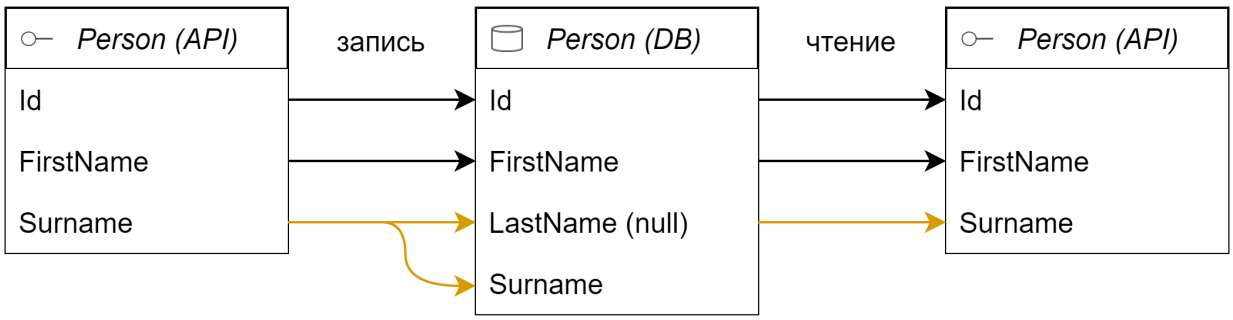
В этом подразделе все тоже предсказуемо. Оставлю видимым функционал, наиболее близкий к целевому.

Сопоставления.

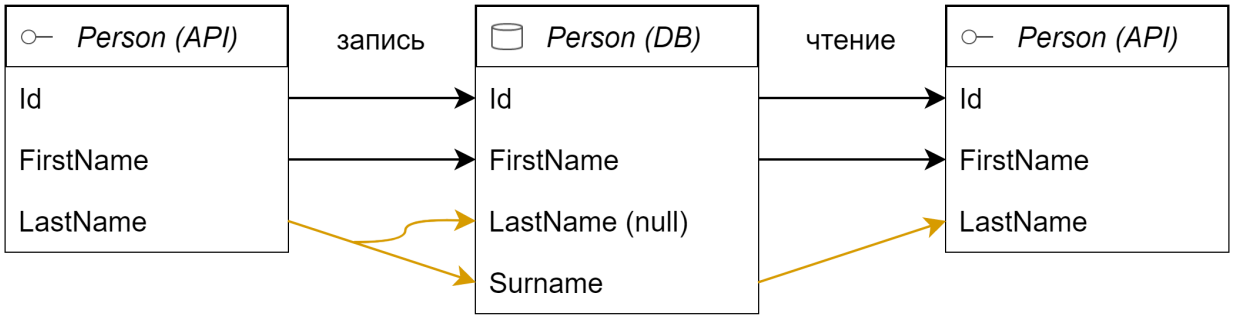
Версия приложения v 2.0, версия API v 1.0:



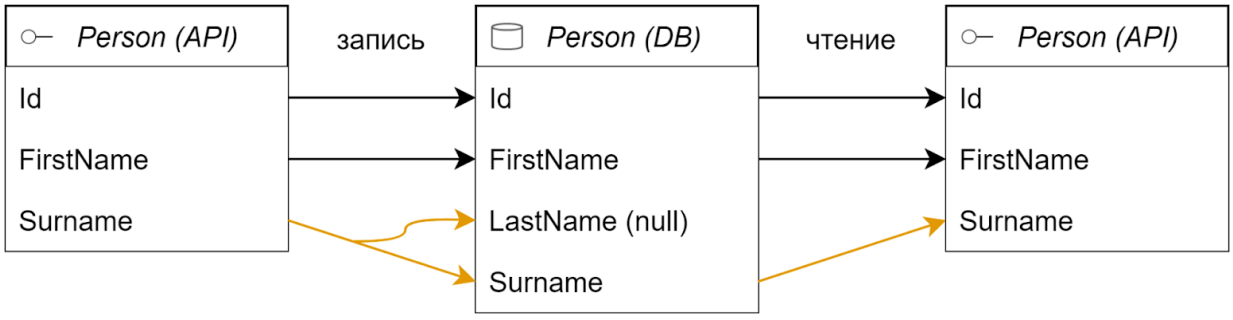
Версия приложения v 2.0, версия API v 2.0:



Версия приложения v 2.1, версия API v 1.0:



Версия приложения v 2.1, версия API v 2.0:



Сопоставление моделей API и таблицы БД v1

Этап 3

На этом этапе мы отказываемся от чтения/записи значений в колонку `LastName`. API v 1.0 все еще можно поддерживать, но в примере будем считать, что все потребители уже переключились на API v 2.0 и поддержкой предыдущей версии можно пренебречь.

Реализация модели слоя данных...

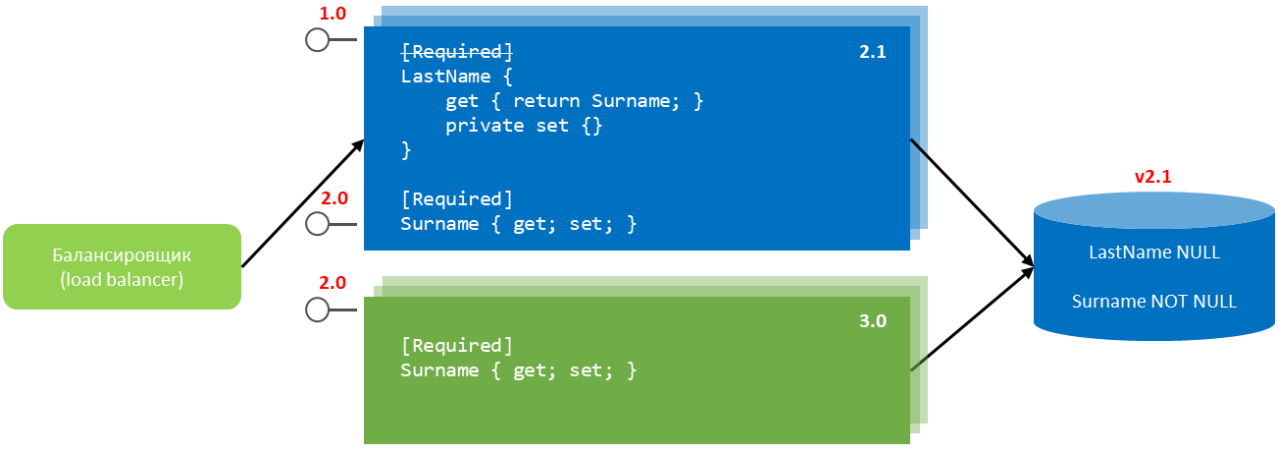
▶ ...этапа 3

Примечание: Я не нашел качественного способа удалить свойство из класса без игнорирования удаления колонки базы данных при генерации миграции. Поэтому могу посоветовать лишь нарушить подход code first и не создавать миграцию в рамках выпуска этой версии.

Этап 3: развертывание v 3.0

Поскольку при предыдущем релизе были использованы синие серверы, теперь новую версию будем развертывать на освободившиеся зеленые серверы.

Шаг 1. Развертывание новой версии.



Развертывание приложения v 3.0

Шаг 2. Ввиду отсутствия миграции шаг 2, его содержащий, отсутствует.

Шаг 3. Тестирование.

Развертывание приложения v 3.0. Тестирование

Примечание: несмотря на отсутствие шага с миграцией я покрасил на схемах базу данных в зеленый цвет, чтобы визуально указать на соответствие версии базы данных актуальной версии приложения.

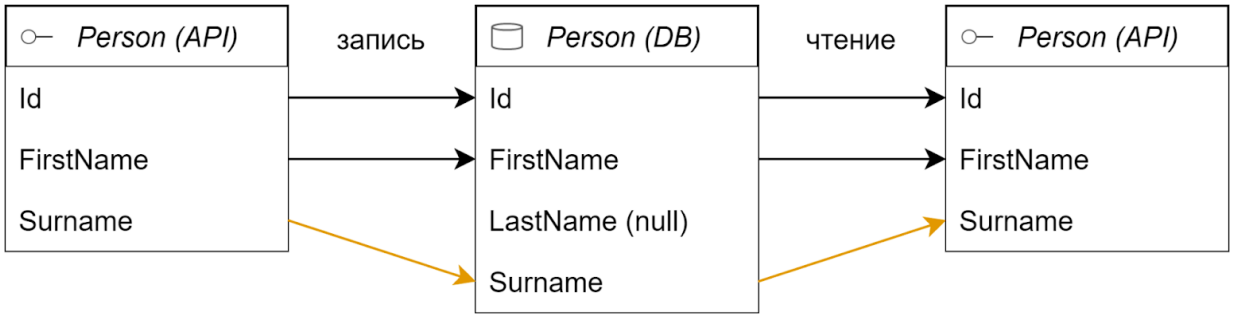
Шаг 4. Эксплуатация приложения версии 3.0.

Развертывание приложения v 3.0. Переключение балансировщика

Этап 3: сопоставления с БД v 2.1

Ввиду того, что версия базы данных не изменилась, сопоставления приложения версии 2.1 смотрите в предыдущем этапе, а здесь укажу лишь сопоставление текущей версии.

Версия приложения v 3.0, версия API v 2.0:



Сопоставление моделей API и таблицы БД v1

Этап 4

Генерация финальной миграции — удаление столбца `LastName` из базы данных.

Реализация v 1.0.

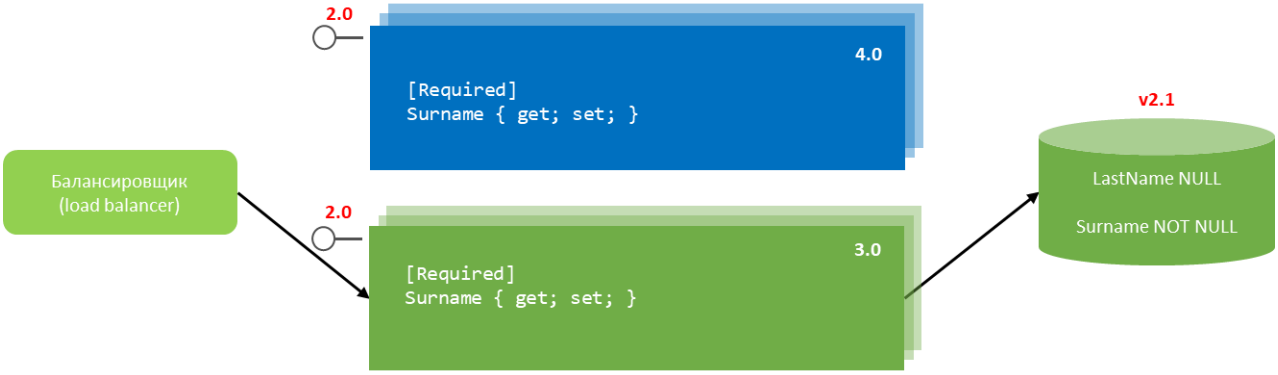
▶ Модели API v 2.0 и базы данных v 3 абсолютно совпадают

Миграция на основе изменений.

▶ Скрипт TSQL

Этап 4: развертывание v 4.0

Шаг 1. Развертывание новой версии.

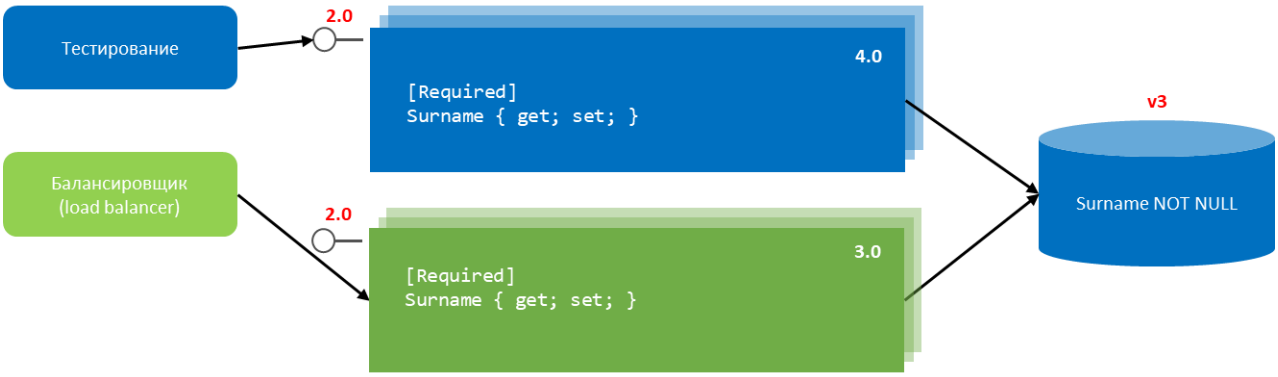


Развертывание приложения v 4.0

Шаг 2. Миграция.

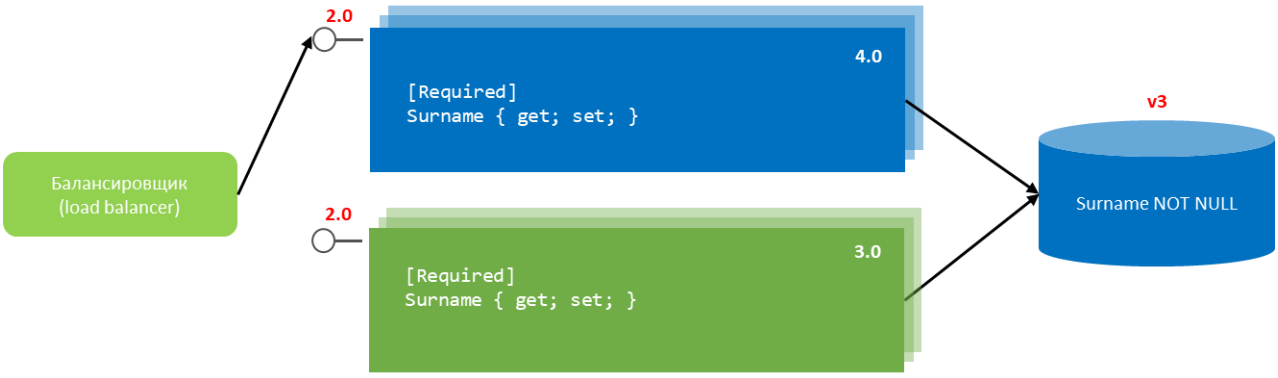
Развертывание приложения v 4.0. Миграция

Шаг 3. Тестирование.



Развертывание приложения v 4.0. Тестирование

Шаг 4. Это финальная версия приложения. Мы достигли поставленной задачи!



Развертывание приложения v 4.0. Переключение балансировщика

Этап 4: сопоставления с БД v 3

Версия приложений v 3.0 и v 4.0, версия API v 2.0 — сопоставления совпадают:

Сопоставление моделей API и таблицы БД v 1

Шпаргалка для бэклога

Здесь сводка того, что мы сделали (а мы много чего сделали), по выполненным шагам.

У нас имелось **приложение версии 1.0 с БД версии v 1.0** (столбец с наименованием `LastName`).

Развернуто приложение версии v 2.0.

- Добавлена поддержка API целевой версии v 2.0.
- Потребителям предлагается начать переход с API версии v 1.0 на версию v 2.0.
- Данные сохраняются в столбцы `LastName` и `Surname`, где `Surname` — это поле-дублер.

- Приложение всегда читает из столбца `LastName`.

Примечание: `Surname` не имеет ограничения `NOT NULL`. БД версии v 2.0.

Развернуто приложение версии 2.1.

- Поддержка API версии 1.0 в этом релизе может быть продлена.
- БД все так же содержит оба столбца `LastName` и `Surname`.
- Столбец `Surname` — копия столбца `LastName`, но они поменялись ролями, теперь `LastName` — поле-дублер.
- Приложение всегда читает из столбца `Surname`.

Примечание: `Surname` имеет ограничение `NOT NULL`, а `LastName` теперь его не имеет. БД версии v 2.1.

Развернуто приложение версии v 3.0, которое сохраняет данные только в столбец `Surname`и читает из `Surname`.

- Столбец `LastName` сохранен только для обратной совместимости и не используется.
- Поддержка API версии v 1.0 еще возможна, но уже прекращена.

Примечание: БД по-прежнему имеет версию v 2.1.

Развернуто приложение версии v 4.0.

- Финальная миграция приводит БД к версии v 4.0, в которой завершен переход от `LastName` к `Surname` и удален столбец `LastName`.
- Поддержка API версии v 1.0 еще возможна, но уже нецелесообразна.

> Следуя этому подходу всегда доступен откат на предыдущую версию, не нарушая целостности базы данных и совместимости приложений.

Но если все свести к самому минимуму, то мы сделали 4 простых действия:

- добавили поле-дублер и новую версию API v 2.0;
- перенесли данные из исходного в дублер и поменяли их ролями;
- отменили поддержку старого API версии v 1.0;
- удалили старое поле.

С точки зрения организации и контроля последовательности шагов это будет работать так же, как шпаргалка.

Когда вы начинаете делать такие изменения, вкладывайте в бэклог задачи, которые касаются всех шагов.

Например, вы коснулись переименования поля, но у вас еще были задачи по добавлению обязательного поля и по удалению обязательного поля, которые тоже добавляете в бэклог. Декомпозируйте весь процесс детально. Так появится инструкция с рекомендациями, которая поможет выполнить все. Выбираете нужные шаги, добавляете в бэклог, достаете по одной и последовательно выполняете.

Кроме всего прочего, вовсе необязательно растягивать релизы на несколько спринтов. Ничто не мешает делать несколько релизов подряд. Естественным ограничителем будет лишь процесс перехода на новый API версии 2.0 всех потребителей.

Конец. Цели достигнуты!

Подведем итоги:

- **Остановки серверов для обновления нет**. Как в самом начале, так и в случае откатов.
- **Синхронизация обновления версий больше не требуется** — каждый микросервис в цепочке взаимосвязей может иметь свой собственный независимый релизный цикл.
- **Цепной реакции** из-за отказа новой версии с зависимостями **больше нет**.
- Нет нужды использовать **резервную копию БД**.
- **Нет простоя** при откате на предыдущую версию.
- **Нет потерь данных**. Ведь мы не откатывали БД на устаревшую резервную копию.

Если остановка на несколько минут не проблема, например ночью, то blue-green deployment вам, возможно, не нужен. Но в компаниях, работа которых построена на постоянном взаимодействии с клиентами, где остановка хоть на секунду — это большие материальные и репутационные издержки, данный способ программирования очень важен.

Дополнительным бонусом от применяемого подхода становится возможность использования канареечного развертывания. Это само по себе ценно, потому что позволяет контролировать качество исполнения и работоспособность новой версии плавно наращивая нагрузку.

На этом всё. Спасибо за внимание!