Лабораторная работа №1: Проектирование сервиса сокращения URL

Постановка задачи: Вы должны спроектировать сервис сокращения URL, аналогичный TinyURL. Сервис должен принимать длинный URL и возвращать уникальный короткий URL. При переходе по короткому URL пользователь должен быть перенаправлен на исходный длинный URL.

Функциональные требования:

1. Пользователь может ввести длинный URL и получить короткий.
2. При переходе по короткому URL происходит HTTP-редирект на оригинальный URL.
3. Пользователи могут задавать кастомные короткие URL (псевдонимы).
4. Короткие URL должны иметь ограниченный срок действия (например, 1 год).

Нефункциональные требования:

1. **Высокая доступность:** Сервис должен быть доступен 99.9% времени.
2. **Низкая задержка:** Редирект должен происходить как можно быстрее (<100 мс).
3. **Масштабируемость:** Система должна быть рассчитана на 10 миллионов новых URL в месяц. Соотношение чтений (редиректов) к записям (создание URL) — 100:1.

Задания:

- **Оценочные расчеты:** Рассчитайте ожидаемое количество запросов в секунду (QPS) на чтение и запись, а также требуемый объем хранилища на 5 лет.
- **Проектирование API:** Определите эндпоинты REST API для создания и получения URL.
- **Модель данных:** Спроектируйте схему базы данных. Обоснуйте выбор между SQL и NoSQL.
- **Технические решения:**

- Предложите и сравните 2-3 алгоритма для генерации уникальной короткой части URL (например, хэширование + Base62, инкрементальный счетчик + Base62).
- Опишите, как будет реализована поддержка кастомных псевдонимов и как обрабатывать конфликты.
- Предложите стратегию кэширования для ускорения редиректов.

- **UML-диаграмма:** Нарисуйте компонентную диаграмму высокого уровня, показывающую основные сервисы (например, веб-сервер, сервис приложений, базу данных, кэш) и связи между ними.
- **Анализ узких мест:** Определите потенциальные узкие места в вашем дизайне (например, генерация уникальных ключей при высокой нагрузке на запись, нагрузка на базу данных) и предложите способы их устранения.

Решение

1. Произведем оценочные расчеты.

Сначала определим ожидаемое количество запросов в секунду (QPS) на чтение и запись. По условию, нам сказано, что на каждое создание нового url происходит 100 чтений (редиректов). То есть чтений в 100 раз больше, чем записей.

Итак, 10 000 000 новых url в месяц – это 10 000 000 новых записей в месяц.

1 месяц = 2 592 000 секунд.
=> 10 000 000 / 2 592 000 ≈ 4 запроса в секунду на записи
4 * 100 = 400 запроса в секунду на чтение

Теперь, определим требуемый объем хранилища на 5 лет. Если в месяц будет 10 000 000 новых записей, то за 5 лет будет примерно 600 000 000 записей на url. Однако так было бы, если бы записи не удалялись. По

условию, каждый короткий URL имеет срок жизни – 1 год. То есть каждый год записи, которые были созданы ровно год назад будут недействительными и будут удаляться. Поэтому нужно смотреть на пиковую нагрузку за эти 5 лет. Максимальное количество записей в базе будет достигаться через 1 год. С этого момента, каждый месяц будут добавляться 10 000 000 новых записей, а удаляется столько же.

Поэтому за 1 год потребуется 120 000 000 записей. Столько и будет максимум в базе данных за 5 лет.

Нам нужно решить сколько гигабайт нам нужно под такое количество записей.

Рассмотрим, что у нас есть, а также сколько примерно места в памяти нужно для этого:

- URL_ID – короткий URL. Это будет небольшая строка, которую будет формировать генерирующий алгоритм нашей системы. Либо это будет небольшая строка, которую задаст сам пользователь

- Длинный URL — исходный URL, предоставляемый пользователем.
  Средняя длина URL в реальном интернете была оценена эмпирически по выборке реальных веб-адресов из открытого веб-архива Internet Archive (проект Common Crawl) и составляет порядка ≈ 77 символов.
  При этом распределение длины URL имеет длинный «хвост»: URL с параметрами, трекингом и динамической генерацией часто значительно превышают среднее значение и могут достигать 150–200 символов и более.
  Из-за высокой вариативности длины URL и в целях консервативной оценки объёма хранилища, в расчётах используется значение 200 байт на один URL, что покрывает большую часть реальных случаев и обеспечивает запас по ёмкости.

Источники:

- https://archive.org
- https://index.commoncrawl.org

- Дата истечения (TTL) короткого URL. Лучше всего подойдет тип timestamp, который весит обычно 8 байт

- Служебные данные базы данных (индексы, метаданные). Обычно закладывают ×2 от "голого" размера.

Значит одна запись будет весить где-то (14 + 200 + 8) * 2 = 444 байт.

=> 444 * 120 000 000 = 53 280 000 000 байт ≈ 53.28 Гб – потребуется на все записи по URL.

Однако нельзя брать ровно 53.28 Гб. Нужно иметь какой-то capacity:

- Если вдруг данные будут удаляться не ровно через 1 год, а чуть позже, то в базе может временно храниться больше записей. Поэтому желательно добавить 10-20% запаса на такой случай: 53.28 * 1.2 ≈ 64 Гб
- Если данные реплицируются (например, для отказоустойчивости), то объём хранилища умножается на количество реплик. Например, если 3 реплики: 64 ГБ × 3 = 192 Гб
- Возможно, мы захотим хранить какие-нибудь логи, а также статистику. Может чуть позже расшириться функционал и нам потребуется хранить еще данные. Поэтому желательно добавить 10-30% запаса на такой случай: 192 Гб + 20 Гб (30% от 64 с округлением вверх) = 212 Гб

То есть за 5 лет потребуется ≈ 212 Гб.

2. Определим эндпоинты REST API для создания и получения URL:

Путь	Тип запроса	Требуется авторизация	Что делает
/{url_id}	GET	Нет	Находит в базе данных по переданному url_id – исходный длинный URL и выполняет редирект для пользователя
/api/urls/{url_id}	GET	Нет	Находит в базе данных по переданному url_id соответствующую запись о данном URL и возвращает JSON с подробной информацией о нем без редиректа
/api/urls	POST	Нет	Создает для пользователя в базе данных новый id для короткого URL, связывая его с исходным длинным URL

Путь	Параметр пути	Тело запроса	Тип данных	Значение по умолчанию
/{url_id}	url_id	-	String	-
/api/urls/{url_id}	url_id	-	String	-
/api/urls	-	originalURL	String	-
		customID	String?	null

Путь	Тело ответа в случае успешного выполнения запроса	Тип данных
/{url_id}	-	-
/api/urls/{url_id}	originalURL	String
	customID	String
	userID	String
/api/urls	originalURL	String
	customID	String

Путь	Коды ответов	Описание
/{url_id}	301	Успешный редирект. В заголовке Location будет исходный URL
	404	Запрашиваемый ресурс не найден, либо у клиента нет к нему доступа
	500	Внутренняя ошибка сервера
/api/urls/{url_id}	200	Запрос выполнен успешно. Тело ответа содержит JSON с информацией о запрошенном url
	401	Нужно пройти авторизацию перед запросом
	404	Запрашиваемый ресурс не найден, либо у клиента нет к нему доступа
	500	Внутренняя ошибка сервера
/api/urls	201	Исходный URL был успешно сокращен. Тело ответа содержит информацию о выполненной операции
	400	Ошибка валидации в теле запроса. Тело ответа содержит описание ошибки
	401	Нужно пройти авторизацию перед запросом
	500	Внутренняя ошибка сервера

3. Проектирование схемы базы данных

Прежде чем переходить к проектированию схемы базы данных, выберем какой тип базы данных лучше всего подходит под данные требования – SQL, NoSQL, а также какая СУБД лучше всего подходит под заданные требования.

Обратимся к CAP-теореме. Согласно ей, в распределенной системе невозможно одновременно обеспечить три свойства: консистентность

(Consistency), доступность (Availability) и устойчивость к разделению (Partition Tolerance). Это означает, что при проектировании распределенной базы данных надо делать выбор в пользу двух из этих свойств и жертвовать третьим:

- Консистентность: означает, что все узлы (серверы) в распределённой системе видят одни и те же данные в один и тот же момент времени. То есть, если ты записал данные в систему, то при чтении из любого узла ты получишь самую последнюю версию этих данных.

  Пример: есть два банкомата (узла) одного банка. Я снял 100 евро с одного банкомата. Консистентность гарантирует, что если я сразу же проверю баланс на втором банкомате, то увижу уже обновлённую сумму — без 100 евро.

  Что будет, если нет консистентности: я снял деньги с одного банкомата, а второй банкомат ещё не узнал об этом и показывает старый баланс. Это может привести к ошибкам — например, я смогу снять больше денег, чем у меня есть на счёте.

- Доступность: означает, что система всегда готова ответить на запрос — даже если часть узлов вышла из строя. То есть, если я обращаюсь к системе, я всегда получу ответ (пусть и не всегда самый актуальный).

  Пример: если один из банкоматов сломался, то второй должен продолжать работать и выдавать мне информацию о балансе или позволять снимать деньги.

Что будет, если нет доступности: если один узел упал, система может перестать отвечать на запросы вообще, пока кто-то не починит узел.

- Устойчивость к разделению: означает, что система продолжает работать, даже если связь между узлами нарушена (например, из-за сбоя сети). То есть, узлы могут временно "не видеть" друг друга, но каждый продолжает функционировать независимо.

  Пример: если связь между двумя банкоматами пропала (например, из-за обрыва кабеля), то каждый банкомат должен продолжать работать самостоятельно, а не "зависать" в ожидании восстановления связи.

  Что будет, если нет устойчивости к разделению: при обрыве связи система может полностью перестать работать, пока связь не восстановится.

По заданным требованиям, упор делается в сторону доступности, а также устойчивости к разделению. То есть консистентность не так важна здесь:

- Доступность: Сервис должен быть доступен 99.9% времени и обеспечивать низкую задержку (<100 мс). То есть если пользователь переходит по короткому URL, он должен всегда получить ответ (редирект), даже если часть системы временно недоступна.

- Устойчивость к разделению: У нас система должна быть рассчитана на 10 миллионов новых URL в месяц и соотношение чтений к записям 100:1. Это очень много. То есть мы рассчитываем, что у нас будут пользователи из разных городов/стран.Чтобы обеспечить

низкую задержку (<100 мс) для пользователей по всему миру, сервис должен быть распределённым (например, с серверами в разных регионах). Разделы сети неизбежны при такой архитектуре (например, проблемы с связью между дата-центрами, или перегрузка сети, или проблемы у интернет-провайдера и т.д.). Если система не устойчива к разделам, то при обрыве связи между регионами часть пользователей потеряет доступ к сервису — это нарушит требование высокой доступности (99.9%).

Распределенная система – это набор независимых компьютеров (узлов), которые работают вместе как единое целое, чтобы выполнить общую задачу. Они связаны через сеть (например, интернет) и обмениваются данными, но не имеют общей памяти или часов. То есть несколько машин (серверов) работают вместе, как одна большая система. Они координируют свои действия через сеть (например, обмениваются сообщениями). Если одна машина ломается, система продолжает работать

- Консистентность: Некритично, если разные узлы системы временно имеют немного разные данные. Например, если пользователь создал короткий URL, а другой узел ещё не узнал об этом — это не катастрофа. Главное, что через несколько секунд/минут данные синхронизируются, а также, что сервис всегда доступен и перенаправляет пользователей, пусть даже с небольшой задержкой в синхронизации.

SQL-базы данных, такие как MySQL и PostgreSQL, традиционно концентрируются на консистентности и устойчивости к разделению. Они обеспечивают строгую целостность информации, следуют принципам ACID.

В условиях высоких нагрузок и сетевых сбоев такие системы могут жертвовать доступностью ради поддержания консистентности.

А комбинацию доступности и устойчивости к разделению лучше всего поддерживают NoSQL-базы.

Также для нас в приоритете именно горизонтальная масштабируемость, а не вертикальная, так как:

- Вертикальная масштабируемость имеет предел: даже самый мощный сервер не справится с ростом нагрузки, если пользователей станет в 10 раз больше, а горизонтальная масштабируемость позволяет добавлять новые серверы по мере роста нагрузки, например: условно можно добавить ещё N серверов для обработки редиректов, если нагрузка вырастет до M запросов в секунду.

- У нас на каждую запись (создание короткого URL) приходится 100 чтений (редиректов). То есть нагрузка на чтение в 100 раз выше, чем на запись. Сюда больше подходит горизонтальная масштабируемость, так как чтения (редиректы) можно распределить между многими серверами (например, через балансировщик нагрузки). Записи (создание URL) тоже можно распределить. Вертикальная масштабируемость не поможет: один сервер не сможет обработать 100% нагрузки на чтение, если она вырастет в десятки раз.

- Система должна быть доступна 99.9% времени (то есть не более 8.76 часов простоя в год). Для этого больше подходит горизонтальная масштабируемость, так как горизонтальная масштабируемость позволяет дублировать серверы в разных зонах доступности (например, в AWS или Google Cloud). Если один сервер или дата-центр выходит из строя, другие серверы продолжают работать, обеспечивая доступность. Вертикальная масштабируемость не даёт

отказоустойчивости: если один мощный сервер упадёт, вся система станет недоступной.

- Редирект должен происходить менее чем за 100 мс. Для этого больше подходит горизонтальная масштабируемость, так как чтобы уменьшить задержку, серверы должны быть распределены географически (например, в Европе, США, Азии). Пользователь из Германии будет перенаправлен на ближайший сервер (например, во Франкфурте), а не на единственный мощный сервер в США. Вертикальная масштабируемость не решает проблему географической задержки: один сервер не может быть "близко" ко всем пользователям одновременно.

- Пользователи могут задавать свои короткие URL. Для проверки уникальности псевдонимов нужна распределённая блокировка или консенсус между узлами. Горизонтальная масштабируемость позволяет реализовать это через распределённые алгоритмы (например, с использованием Redis). Вертикальная масштабируемость не решает проблему согласованности между узлами.

SQL-базы фокусируются на технике вертикальной масштабируемости. NoSQL-базы данных были разработаны с ориентацией в первую очередь на горизонтальное масштабирование. Это означает, что вместо наращивания ресурсов основного сервера в кластер просто добавляют дополнительные узлы. Также NoSQL позволяет размещать данные ближе к пользователям (например, через географически распределённые кластеры). SQL-базы с шардингом могут добавлять задержки из-за сложных запросов, а у нас есть требование к низкой задержке (< 100 мс). В SQL-базах горизонтальная масштабируемость поддерживается, но ее в разы сложнее реализовать и

поддерживать, чем в NoSQL (в NoSQL горизонтальная масштабируемость встроена "из коробки"), а также используя SQL для горизонтальной масштабируемости – задержки чаще всего более длительные, чем при использовании NoSQL.

Таким образом, лучше использовать NoSQL-базу и мы будем использовать именно ее.

Теперь разберемся какой СУБД лучше подходит под заданные требования. Каждая запись будет иметь следующую структуру (схема базы данных):

```
{
   “url_id”: String, // Короткий URL
   “original_url”: String, // Исходный длинный URL
   “ttl”: timestamp // Время жизни короткого URL
},
```

Причем url_id здесь – это первичный ключ. То есть для заданных требований будет достаточно только одной таблицы.

Для заданных требований больше всего подходит именно DynamoDB, поскольку:

- DynamoDB гарантирует 99.999% доступности (SLA) благодаря мультирегиональной репликации и автоматическому распределению данных

- Это AP-система (по CAP-теореме), что идеально подходит для сервиса сокращения URL, где доступность и устойчивость к разделам важнее строгой консистентности

- DynamoDB автоматически масштабируется под нагрузку без ручного вмешательства

- Поддерживает миллионы запросов в секунду

- Типичная задержка < 10 мс для операций чтения/записи, что значительно быстрее требуемых 100 мс. Это обеспечивает мгновенный редирект при переходе по короткому URL

- DynamoDB оптимизирован для быстрого чтения по первичному ключу (url_id). Задержка <10 мс делает редирект практически мгновенным

- DynamoDB поддерживает условные запросы (Conditional Put), которые позволяют проверять уникальность кастомных псевдонимов за один запрос (у нас есть требование под реализацию кастомных URL). Это быстрее и проще, чем реализация такой логики в других СУБД, особенно в случае, если придется такую логику реализовывать самому

- Не нужно настраивать кластер, следить за репликацией или шардингом. AWS берёт на себя обновления, бэкапы и мониторинг, что экономит время и ресурсы.

В качестве альтернативы можно рассмотреть Cassandra, ScyllaDB, MongoDB. Однако данные СУБД хуже подходят под заданные требования. Например, MongoDB сложнее масштабировать горизонтально (требует ручного шардинга и настройки реплик), а также там выше задержки при большой нагрузке на чтение, что может повлиять на доступность в случае увеличения нагрузки.

ScyllaDB/Cassandra требуют ручной настройки кластера и сложнее в поддержке (Здесь больше контроля. Я сам настраиваю количество узлов, репликацию, шардинг, мониторинг, бэкапы, обновления), а DynamoDB более “автоматический”. По требованиям нет необходимости делать выбор в пользу

ручного контроля. DynamoDB уже хорошо оптимизирован для успешного выполнения заданных требований. Также DynamoDB будет хорошо справляться в случае увеличения нагрузки с большим запасом, а также расширения функционала (авторизация, добавление своего домена и т.д.). Из минусов, вендор-лок (привязка к AWS) и стоимость, что не влияет на заданные требования. А также плюсы DynamoDB значительно перекрывают ее минусы. Если выбирать, то между ScyllaDB и DynamoDB.

ScyllaDB позволяет добиться лучшей производительности. Также здесь придется платить меньше: за железо (или виртуалки), не за операции. Но здесь потребуется сложная ручная настройка и сложнее поддержка. DynamoDB дороже: придется платить за каждую операцию + за хранение. DynamoDB успешно справится с заданными требованиями по производительности (даже с большим запасом). Совсем не факт, что нагрузка в будущем возрастет до производительности, которую может предоставить ScyllaDB. А DynamoDB более простая поддержка и настройка за счет того, что в ней все автоматизировано.

Если мы большая компания, для которой траты на DynamoDB – не принесут финансовых проблем, а также есть бизнес-план, по которому это приложение сокращения URL будет приносить прибыль, в разы превышающую траты на DynamoDB, то лучше выбрать DynamoDB.

Если мы маленькая компания, или у нас нет идеального бизнес-плана, где прибыль будет в разы превышать затраты, то лучше выбрать ScyllaDB.

Мы выберем именно DynamoDB (все-таки это учебная лабораторная. Представим, что мы большой игрок и идеальный бизнес-план у нас есть. Или представим, что DynamoDB стал бесплатным).

4. Технические решения:

   <u>Алгоритмы генерации уникальной короткой части URL</u>

   - Хэширование + Base62:
     - Описание алгоритма:

    - Берётся оригинальный URL + current_timestamp
    - Применяется криптографическая хэш-функция, например, SHA-256. Получается длинная строка фиксированной длины. Дальше берем первые 8-12 символов хэша (в таком случае вероятность коллизий становится очень низкой), чтобы короткий URL был действительно коротким

      Полученное значение кодируется в Base62 (цифры, маленькие и большие буквы, без специальных символов), что и будет результатом

  - Плюсы:
    - Вероятность коллизий крайне мала, если использовать достаточно длинный хэш, то есть обеспечивается уникальность
    - Трудно подобрать созданный короткий URL какому-нибудь злоумышленнику
    - Не нужно хранить счётчик или генератор
    - Можно генерировать на разных серверах без синхронизации

  - Минусы:
    - Чтобы гарантировать уникальность, приходится использовать более длинные идентификаторы
    - При высокой нагрузке возможны коллизии, особенно если брать короткий хэш

- Инкрементальный счетчик + Base62:
  - Описание алгоритма:
    - В базе данных хранится глобальный счётчик
    - При добавлении нового URL счётчик увеличивается на 1
    - Новое значение счётчика кодируется в Base62, что и является результатом

- Плюсы:
  - Легко реализовать и гарантировать уникальность
  - Можно использовать короткие идентификаторы
  - Можно контролировать длину короткого URL, увеличивая разрядность счётчика
- Минусы:
  - Нужно синхронизировать счётчик между узлами
  - При высокой нагрузке на запись может возникнуть конкуренция за счётчик
  - Последовательные id легко перебрать злоумышленнику

- UUID + Base62:
  - Описание алгоритма:
    - Генерируется UUID
    - Значение кодируется в Base62
    - Проверяется уникальность в базе данных
  - Плюсы:
    - Вероятность коллизий крайне мала (практически нулевая), что обеспечивает уникальность
    - Не нужно хранить счётчик
    - Можно генерировать на любом сервере без синхронизации
    - Трудно подобрать созданный короткий URL какому-нибудь злоумышленнику
  - Минусы:
    - UUID длинный (36 символов), даже если взять только часть, короткий идентификатор будет длиннее, чем у счётчика или хэша

- Если мы будет использовать не все сгенерированное значение, а только ее часть, то нужно правильно выбирать эту часть UUID, чтобы не потерять уникальность

Поддержка кастомных псевдонимов и обработка конфликтов

Каждая запись в базе данных будет иметь следующую структуру:

```
{
  “url_id”: String, // Короткий URL
  “original_url”: String, // Исходный длинный URL
  “ttl”: timestamp // Время жизни короткого URL
},
```

Где url_id – это короткий URL, сгенерированный нашим алгоритмом, либо URL, который придумал сам пользователь.

Наш backend-сервис сможет понять – хочет пользователь создать именно свой кастомный псевдоним, или сгенерировать с помощью нашего алгоритма с помощью DTO-класса, который будет передаваться от клиентской части к нам на backend-сервис. Если в этом классе поле url_id будет равно null, то это будет означать, что короткий URL должен быть сгенерирован с помощью нашего алгоритма. Если это поле не будет равно null, то это означает, что пользователь хочет задать именно этот кастомный псевдоним. Соответственно, на клиентской части пользователь сможет выбрать – генерацию короткого URL с помощью алгоритма или задание собственного псевдонима.

DynamoDB позволяет не проверять самостоятельно наличия в базе данных указанного ключа. То есть когда пользователь будет делать POST-запрос для создания своего короткого URL, достаточно просто сделать запрос для создания новой записи, а DynamoDB сам все проверит. Если запись с таким коротким URL уже существует, то вернется ошибка. В таком случае, мы сообщим пользователю о том, что такой короткий URL уже занят. Если

записи с таким коротким URL нет, то ошибка не вернется. Тогда мы вернем пользователю ответ о том, что запись была успешно создана.

Таким образом, если пользователь задаст кастомный псевдоним, который уже существует, то DynamoDB не даст создать такую запись. Даже если несколько пользователей одновременно попытаются создать новую запись с одним и тем же кастомным псевдонимом, то все-равно кто-то создаст запись чуть раньше. Тому, для кого запись создастся раньше – вернется ответ о успешном создании записи. А другому пользователю DynamoDB уже не даст создать такую запись и ему вернется сообщение об конфликте имен. Тогда этот пользователь сможет придумать другое имя для своего псевдонима и заново попробовать его создать.

Балансировка нагрузки

Для выполнения заданных нефункциональных требований, не обойтись без балансировщика нагрузки. Если у нас есть несколько нод, балансировка позволяет нам распределить правильно нагрузку на эти несколько серверов и кэширование будет ускорять доступ к данным, снижая нагрузку на базу данных, что очень важно.

Есть два варианта реализации балансировщика нагрузки:

- Балансировщик нагрузки можно реализовать на уровне L4
- Балансировщик нагрузки можно реализовать на уровне L7

Нам больше подойдет именно L7, что мы и реализуем. Дело в том, что при балансировщике нагрузки L4 мы не можем маршрутизироваться по содержимому запроса, а на L7 можем. Сейчас это может быть не очень нужно, но в будущем наверняка появится дополнительный функционал, к примеру возможность авторизации. Тогда это будет прямо необходимо. Также с помощью L7 мы сейчас сможем распределять нагрузку следующим образом: запросы к эндпоинту на редирект будем направлять к одним серверам, а запросы к эндпоинту на создание новой записи – к другим. Также

на L7 мы сможем расшифровывать HTTPS-трафик на себе, снимая это нагрузку с backend-сервера, что также хорошо.

Теперь разберемся какой алгоритм балансировки лучше взять. Разберем самые популярные алгоритмы:

- Round Robin – циклический обход по нашим серверам
- Least Connection – выбираем тот сервер, на котором наименьшее количество активных соединений
- Source IP Hash – алгоритм, использующий hash IP-адреса клиента, чтобы всегда консистентно подключаться к одному и тому же серверу

Для заданных требований лучше всего подойдет Round Robin. То есть Балансировщик нагрузки просто циклически будет итерироваться по всем серверам по очереди: первый запрос на сервер1, второй запрос на сервер2 и т.д.

Плюсы:

- Это абсолютно простой алгоритм в реализации
- При этом он дает равномерную нагрузку, если наши сервера являются одинаковыми

Часто, но не всегда, основная стратегия масштабирования – это репликация, когда нам требуется балансировщик нагрузки. У нас гомогенные сервера, которые являются примерно одинаковыми, то есть наша стратегия масштабирования – это репликация, то мы можем гарантировать с алгоритмом Round Robin равномерное распределение нагрузки по ним.

Минус:

- Он не учитывает текущую загрузку сервера, поэтому если наши сервера не гомогенные, то мы можем перегрузить более медленные сервера, потому что нагрузка будет распределяться без учета того, сколько реального ресурса наши сервера имеют

Однако у нас сервера именно гомогенные, поэтому такой проблемы не будет. Из-за этого, Round Robin – идеальный вариант.

Least Connection излишен здесь, так как он больше для долгоживущих соединений, типа WebSocket, которых у нас нет. К тому же Least Connection дает дополнительную нагрузку, дополнительный overhead на свое собственное содержание. А нам дополнительная нагрузка не нужна. То есть здесь преимущества Least Connection не нужны, так как Round Robin идеально справится. Round Robin дает меньше нагрузки, а также он будет равномерно распределять нагрузку по нашим серверам, так как они гомогенные.

Source IP Hash также нас не подходит, так как нам не требуется обеспечивать Session Affinity.

В качестве такого балансировщика нагрузки L7 мы выберем Cloudflare, поскольку он предоставляет надежную защиту от Ddos-атак, что очень важно для заданных нефункциональных требований.

Стратегия кэширования для ускорения редиректов

Для обеспечения требуемой доступности, а также соблюдения требования по быстрому редиректу, необходимо использовать кэш.

Кэширование – это ускорение ответов системы за счет сохранения часто используемых ответов в некотором быстром сегменте памяти. С кэшем повторные запросы обслуживаются из памяти, что в разы быстрее.

Наша задачи кэшировать данные, которые были запрошены недавно и данные, которые часто запрашиваются вместе. Нужно сделать так, чтобы данные чаще были взяты из кэша, а не из базы данных.

Понятно, что у нас уже есть Client Side Cashing – кэш на уровне браузера. Однако для соблюдения заданных нефункциональных требований этого недостаточно. Это лишь часть от необходимой стратегии кэша.

Мы также должны будем использовать:

- CDN (Edge-кэш) – кэш на уровне веб-сервера. Это стратегия кэширования для статического и динамического контента. То есть у нас есть распределенные по миру (по условному миру) сервера, на ноды которых мы можем положить статический контент, который будет отображаться на нашем сайте, чтобы он грузился быстрее и не читался из хранилища. Тут две стратегии: либо создатель веб-приложения сам пушит в CDN контент, который он хочет отображать, либо у нас происходит кэш-мисс и кладем туда необходимые данные

- Также нам потребуется Reverse Proxy – кэш на уровне веб-сервера. Здесь также речь идет про Nginx. Сюда кэшируются часто запрашиваемые страницы. То есть мы выставляем некоторую Policy, которая определяет как мы определяем часто посещаемую страницу и как долго мы храним этот кэш

- Также нам нужен Application cache – кэш для самого приложения – это такая общая зона для всех наших распределенных нод. Некоторый набор, хранящийся в памяти информации, которая может быть быстро запрошена всеми нашими серверами. К этому типу кэша относится Redis. Этот вид кэша ширится внутри кластера

- А также Database cache - встроенные в протокол DBMS буфера и кэширование результата запросов. Это позволяет нам немного сэкономить на дорогих больших запросов в базу данных, если предполагается, что они будут происходить более одного раза. Как правило, мы стараемся избегать злоупотребления кэширования на уровне базы данных, потому что предполагается, что если мы часто запрашиваем одни и те же данные – они будут кэшированы на

уровне приложения. Но тем не менее, в качестве механизма, сама база данных, сама СУБД нам, как правило, дает такую возможность

CDN, Reverse Proxy – будет внедрен на стороне нашего балансировщика нагрузки Nginx. Его настройкой будет заниматься DevOps.

Database cache – DynamoDB будет использовать его автоматически.

Соответственно, разберем именно наш Application cache. В качестве инструмента, лучше всего подойдет Redis, который является самой популярной технологией для кэширования.

Теперь разберемся со стратегией. Есть несколько разных стратегий масштабирования (стратегии кэширования):

- Ленивая загрузка
- Сквозная запись
- Отложенная запись
- Упреждающая обновление

Для заданных требований больше всего подойдет стратегия "Ленивая загрузка". То есть приложение при запросе сначала будет проверять кэш. Если данных нет, то мы получим кэш-мисс и мы запросим эти данные из базы данных, получим их, сохраним в кэш их, а уже после вернем их пользователю. При обновлении данных в базе данных – соответствующий ключ в кэше инвалидизируется.

Плюс:

- Очень просто. Кэшируются только запрашиваемые данные

Минус:

- Первый запрос после промаха (когда нет данных в кэше) – всегда очень медленный

Здесь также подойдет стратегия "Сквозная запись". Однако в таком случае придется после каждого POST-запроса сохранять эти данные в кэш. POST-запросов будет много. Нам придется каждый сохранять в кэш. Здесь минус будет в том, что POST-запросы будут медленнее, а также нам придется хранить данные после каждого POST-запроса в кэше, а это нехорошо в том плане, что если пользователь создаст запись с помощью POST-запроса и в ближайшее время сделать запрос на редирект один раз/не сделает вообще, то мы будет хранить не самые популярные данные в кэше. Конечно, можно настроить хороший тайминг для удаления записей в Redis, но таких данных будет много. Давай лишнюю нагрузку на Redis – не самая лучшая идея с учетом заданных нефункциональных требований.

А с помощью ленивой загрузки мы будем хранить только те данные, к которым действительно есть интерес. Которые запрашиваются. То, что иногда будет происходить кэш-мисс и мы будет получать чуть более долгий запрос – пока не критично. DynamoDB и Redis справятся с заданной нагрузкой в нефункциональных требованиях. Они также будут справлять при более высокой нагрузке. Чтобы возникли проблемы, нагрузка должна возрасти в разы. Тогда уже можно будет подумать о переходе на стратегию "Сквозная запись".

Стратегия "Отложенная запись" нам не подойдет, потому что если кэш выйдет из строя до синхронизации с базой данных, то данные будут потеряны. По заданным требованиям к доступности, мы не можем такое допустить.

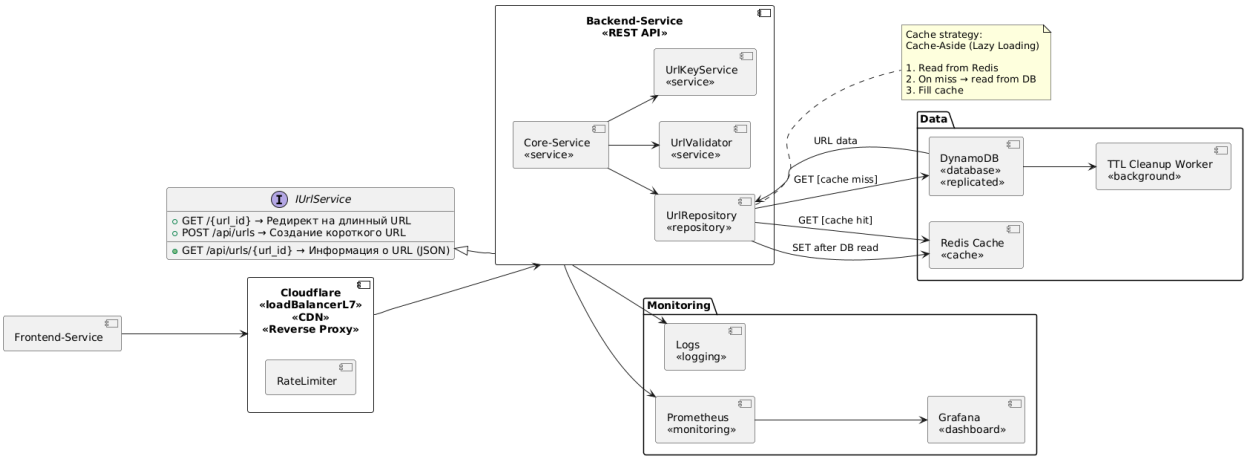
Стратегия "Упреждающая обновление" нам не подходит, так как при такой стратегии нагрузка на базу данных в разы возрастет. А нам нельзя рисковать доступностью (по заданным нефункциональным требованиям). Чтобы это исправить, придется реализовывать сложную балансировку нагрузки на базу данных. То есть алгоритм достаточно сложный для реализации. Пока нет необходимости использовать такой сложный алгоритм. Более простые алгоритмы дадут требуемый результат.

Также ресурс, который мы может хранить в кэше – не бесконечный. Мы должны периодически освобождать его при достижении некоторого лимита нашего capacity. Для нашего Redis лимит capacity – это объем нашей оперативной памяти. Когда кэш заполнен, нужно решить какие данные удалить. Может быть три основных политики:

- Recently used: мы вытесняем самые давно не использованные данные
- Наименее часто используемые данные
- Самые старые данные (first in, first out)

Большинство систем, как Redis по умолчанию используют вариации Recently used. Использовать другую стратегию не стоит. Recently used очень хорошо подходит для заданных нефункциональных требований.

5. Диаграмма компонентов основных сервисов



6. Узкие места в архитектуре
   – Одновременное создание одного и того же кастомного псевдонима в разных регионах может привести к нарушению уникальности

Решение

Для обеспечения строгой глобальной уникальности кастомных псевдонимов используется **single-writer подход**.

Идея решения:

- Для операций создания кастомных псевдонимов выделяется один основной регион - home region
- Все запросы на создание alias, независимо от региона пользователя, маршрутизируются в home region
- В home region используется отдельная таблица Aliases, где alias является **Partition Key**, а создание записи выполняется атомарной операцией с условием attribute_not_exists(alias)

То есть регистрация кастомных псевдонимов выполняется в одном выделенном регионе с атомарной проверкой уникальности на уровне базы данных.

Таким образом:

- Если два пользователя одновременно пытаются создать один и тот же псевдоним, **только первый запрос будет успешно выполнен**
- Второй запрос получит ошибку о том, что псевдоним уже существует
- Уникальность alias гарантируется на уровне базы данных атомарной операцией записи

- Если вдруг Redis, отвечающий за кэш станет не доступным, а также нагрузка в это время в разы возрастет, то DynamoDB может не справиться с ней

Решение

Увеличить пропускную способность DynamoDB. Также можно заняться вертикальным масштабированием. Еще можно добавить больше реплик. Можно использовать Auto Scaling для DynamoDB, чтобы автоматически

увеличивать пропускную способность при росте нагрузки (при резком увеличении нагрузки DynamoDB может начать отклонять запросы из-за превышения лимитов пропускной способности.). Также можно использовать кластер Redis для повышения доступности. Если одна нода упадёт, другие продолжат работу. Можно размещать Redis в нескольких зонах доступности

- Если вдруг Redis, отвечающий за очередь запросов станет не доступным, то запросы, находящиеся в очереди могут потеряться

Решение

Если такое будет происходить, то можно будет перейти на RabbitMQ, где гарантируется доставка в таких случаях, если такая потеря будет критичной, то есть она повлияет на доступность. Либо можно включить персистентность Redis. Лучше переходить на RabbitMQ только если потребуется сложная логика обработки и доставки запросов из очереди. Пока достаточно использования Redis