

## ! Security

- Scalability
  - Scaling
  - Load balancing
- AutoScaling
- Scaling database
  - Database Replication
- Caching

# Вступление

До сих пор мы обсуждали, как создавать простые веб-страницы с помощью HTML и CSS и как использовать Git и GitHub для отслеживания изменений в нашем коде и совместной работы с другими. Мы также познакомились с языком программирования Python, начали использовать Django для создания веб-приложений и научились использовать модели Django для хранения информации на наших сайтах. Затем мы познакомились с JavaScript и узнали, как с его помощью сделать веб-страницы более интерактивными, а также поговорили об использовании анимации и React для дальнейшего улучшения пользовательских интерфейсов. Затем мы поговорили о некоторых лучших практиках разработки программного обеспечения и о некоторых технологиях, обычно используемых для достижения этих лучших практик.

Сегодня, в нашей заключительной лекции, мы обсудим вопросы масштабирования и обеспечения безопасности наших веб-приложений.

## Scalability


# Масштабируемость

Раньше я запускал сервера на локальном компьютере, но так-же можно запускать веб приложения, доступные для всех пользователей в сети интернет. Для этого приложения запускают на **серверах**, они бывают 2-х типов: физические и облачные, которые можно арендовать например у амазон и гугл. У обоих вариантов есть свои плюсы и минусы:

- **Кастомизация** - Физические сервера обладают большей кастомизацией чем облачные.

- **Экспертиза** - Проще разместить сайт на готовом облаке, чем монтировать собственный сервер.
- **Цена** - Облачные сервера в долгосрочном периоде дороже, но физические на начальном этапе более затратны.
- **Масштабируемость** - Масштабировать облачные сервера намного легче.

Одна из основных проблем в том, что любой сервер (неважно какой) имеет ограниченное количество ответов в секунду, т.е. если наш сайт имеет больше клиентов (запросов в минуту) и одного сервера не хватает на обработку, то придется всегда покупать новые сервера.

 А вообще, я думаю, идет тренд к облачным серверам, а не физическим.

Хотя, лично я хотел бы себе физический сервер для своих нужд.

Scaling

## Масштабирование

Т.к. наш сервер имеет ограниченное количество ответов в секунду, то наше приложение можно улучшать (масштабировать). Существует 2 способа:

- **Вертикальное** масштабирование - увеличение работоспособности одного сервера (добавление большего кол-во процессоров, памяти и т.д.)
- **Горизонтальное** масштабирование - покупка большего количества серверов.

Load balancing

## Балансировка нагрузки

Когда мы используем горизонтальное масштабирование, то мы должны распределять нагрузку между серверами. Существует несколько методов сделать распределение нагрузки и вот некоторые из них:

- **Рандом** - отправлять запросы на случайный сервер.
- **Round-Robin** - последовательно отправляет запросы на все сервера. С начала в А, потом в В и так по кругу.
- **Fewest Connection** - отправляет запрос на сервер, который на данный момент меньше всего нагружен.

Не существует лучшего метода балансировки, и на практике используются сразу несколько. Но так-же у балансировки нагрузки есть свои проблемы, например сессии. У нас могут быть сессии, которые хранятся на одном сервере, но не на другом. Вот некоторые решения для таких проблем.

- **Sticky sessions** - Когда пользователь посещает сайт, балансировщик нагрузки запоминает на каком сервере его сессия и отправляет его запросы на данный сервер. Пользователь буквально *прилипает* к серверу. Одна из проблем в том, что есть риск перегрузки сервера из-за количества пользователей на нем.
- **Database sessions** - База данных хранится на отдельном сервере и все сервера имеют доступ к ней. Проблема в том, что это занимает больше времени и нуждается в большей мощности.
- **Client-Side sessions** - Вместо хранения данных на сервере, мы храним их в куках браузера. Проблема заключается в безопасности (подделка куки) и в точности передачи информации.

Так-же как и балансировка нагрузки, данные методы на практике комбинируются.

AutoScaling

## Автомасштабирование

Есть еще одна проблема. В определенное время количество запросов на сервер может быть больше обычного или меньше обычного. В таком случае нам может либо не хватать серверов, либо они будут просто тратить ресурсы без применения. Для такой проблемы используют автомасштабирование, т.е. **увеличение или уменьшение кол-ва серверов, в зависимости от кол-ва запросов**. Но все же, этот метод не идеален.

## Падение сервера

Так-же, горизонтальное масштабирование, или же некоторое количество серверов, помогают избежать ошибки падения сервера, при выходе из строя 1 из серверов, т.к. балансировщик нагрузки увидит, что один из серверов не рабочий, он перенаправит запрос на другой сервер.

Scaling database

## Масштабирование базы данных

В дополнение к масштабированию сервисов, мы должны масштабировать базы данных. Ранее, мы могли хранить данные в 1 sql файле на сервере, но чем больше данных, тем больше место надо и тем сложнее становятся запросы. Вот некоторые способы масштабирования базы данных:

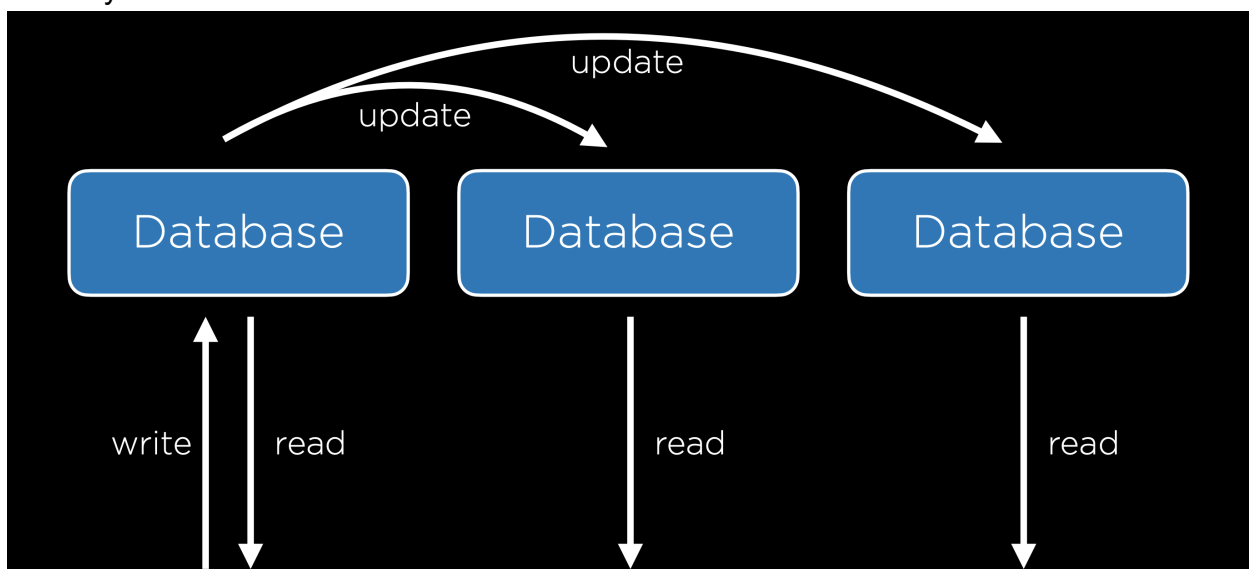
- **Вертикальное разделение** - метод при котором мы разбиваем одну большую таблицу с данными на множество маленьких (подробнее в [SQL](#)).
- **Горизонтальное разделение** - метод при котором мы разделяем информацию с одинаковыми форматами, которые мы будем использовать в разных сервисах. Например, разделить полеты - на внутрирейсовые и интернациональные.

## Database Replication

### Репликация базы данных

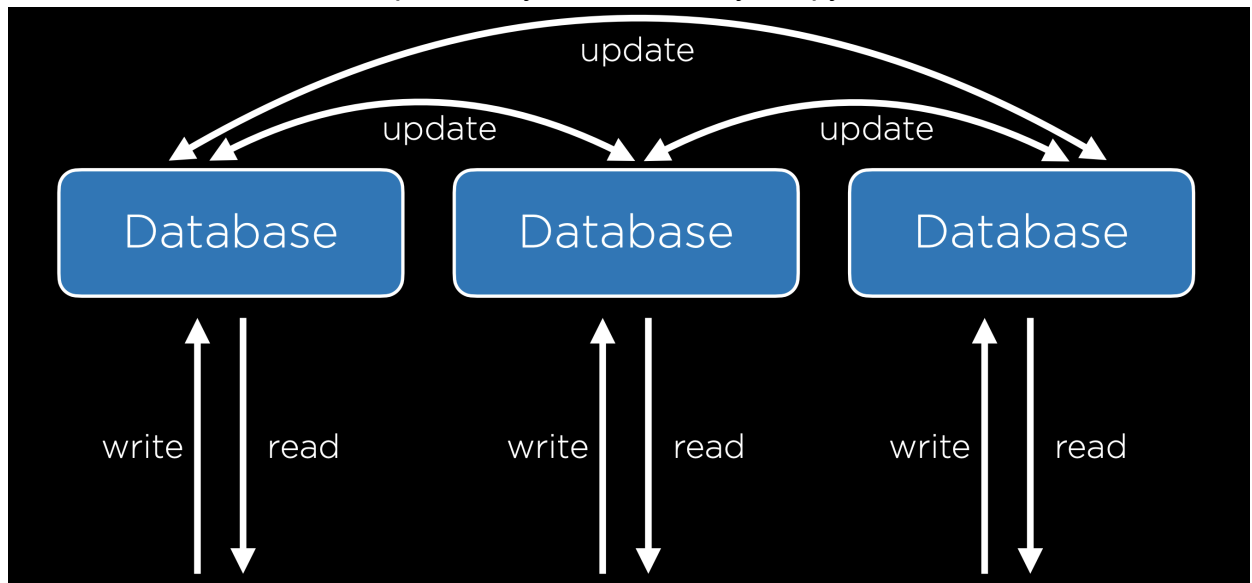
Добавляя дополнительные сервера для сервисов, мы получали гарантию того, что наш вебсайт не перестанет работать даже после отказа одного из серверов. Масштабируя базу данных, логично было бы сделать так-же с ней. Вот несколько наиболее популярных методов репликации базы данных:

- **Single-Primary Replication** - метод при котором мы имеем 1 основную базу данных, которую мы можем изменять и читать, а так-же добавляются доп. базы данные которых можно только читать, а изменяются они вместе с основной базой. Единственная проблема, что этот метод все ещё не гарантирует отказоустойчивость сайта.



- **Multi-Primary Replication** - метод при котором все базы данных можно изменять и читать, а так-же они взаимосвязанно обновляют друг друга. Чем то схоже с гитом. Вот основные проблемы данного метода

- *Update conflict* - когда пользователи меняют одновременно одну ячейку это приводит к проблемем синхронизации.
- *Uniqueness conflict* - каждая ячейка в базе данных должна быть со своим id. Мы можем присвоить один и тот же id двум разным ячейкам разных баз данных.
- *Delete conflict* - один юзер может удалить ячейку, а другой обновить ее.



Caching

## Кэширование

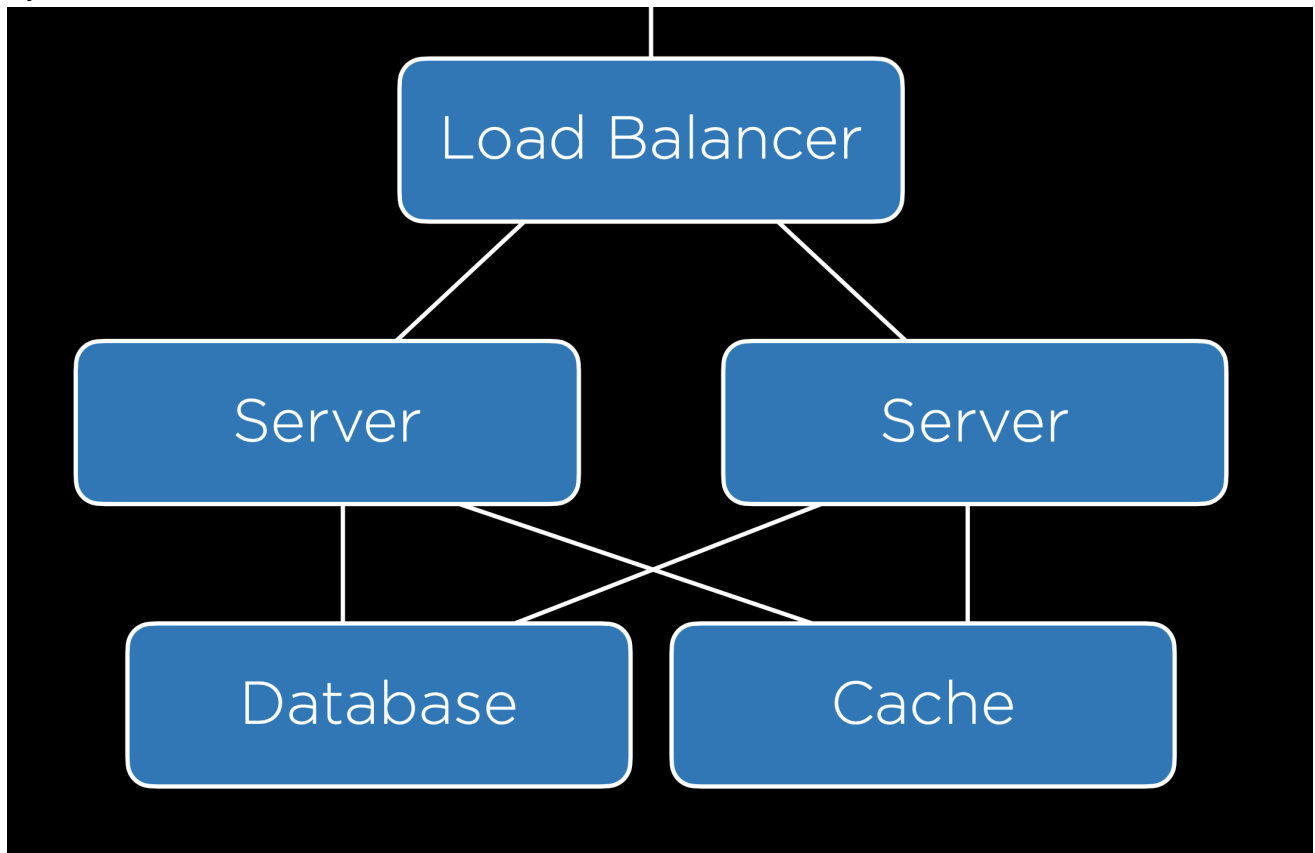
Когда наши базы данных все увеличиваются и увеличиваются, то важно отобрать данные, которые используются чаще всего. Например: наиболее популярные посты, люди и т.д. Подобные данные нет смысла вытаскивать с базы данных напрямую, ведь это - *очень долго*. Чтобы исправить проблему со скоростью, можно использовать кэширование. **Кеш = Скорость**.

Один из способов кэширования, это держать информацию в *кэше браузера*. В таком случае даже не придется отправлять запросы на сервер. Для этого достаточно добавить эту команду в заголовке http запроса:

```
Cache-Control: max-age=86400
```

Эта команда показывает, что при нахождение на сервере 86400 миллисекунд, клиент не должен отправлять запрос на сервер. Это удобно использовать в css файлах, чтобы не отправлять одни и те же стили по 1000 раз. Так-же можно добавить **ETag**, который показывает какие типы файлов не должны отправляться.

Так-же можно использовать кэш на стороне *сервера*. Тогда архитектура приложения будет выглядеть так:



Джанго предлагает их собственный фреймворк, который обладает данными фичами:

- **Pre-View caching** - При загрузке определенной view, повторная загрузка по аналогичной view не будет проходить снова.
- **Template-Fragment caching** - Определенная часть шаблонов не будет повторно рендериться. Например *бар с меню*.
- **Low-Level cache API** - Гибкое кэширование определенных данных.