**The way app is written** - The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It’s a downwards spiral. Чем больше становится приложение, тем дольше оно собирается и стартует, а так же становится сложнее его разрабатывать, так как нужно держать в голове все зависимости со всеми модулями, тогда как в SOA можно думать только о зависимостях в конкретном микросервисе (принцип SOLID)

**Continuous deployment is difficult** - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven’t been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.

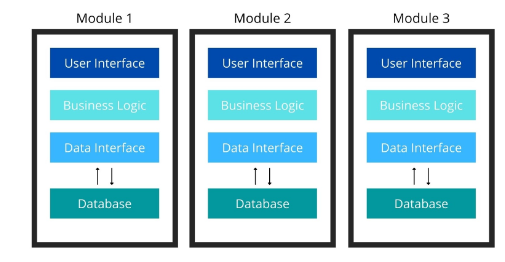
**Scaling the application can be difficult** - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the other hand, this architecture can’t scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently. В случае с SOA, мы можем масштабировать только конкретный микросервис, который больше всего нагружается. В случае же с монолитом, мы должны замаштабировать все сервисы сразу

**Obstacle to scaling development** - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.

**Does it have a state?** Как вправлять stateful состоянием

Если у нас условно в монолите есть 100 функциональностей, а клиенту нужно всего 5, то можно использовать **Modular Monolith** архитектуру. Где приложение будет поделено на модули и клиентам можно выдать только определенный модуль (задеплоить определенный модуль)

The modular monolithic architecture consists of dividing our logic first into modules, and each module will be independent and isolated. Then, each module should have its own business logic — and, if necessary, its database or schema. In that way you can build and modify the layers of each module without affecting the others:



Of course, following this pattern, there should be communication between the modules to work correctly, but this communication must be done through public APIs, allowing access to the logic of each module from the other using public methods, and not using the internal functions and logic of each one.

**Modular Monolithic:**

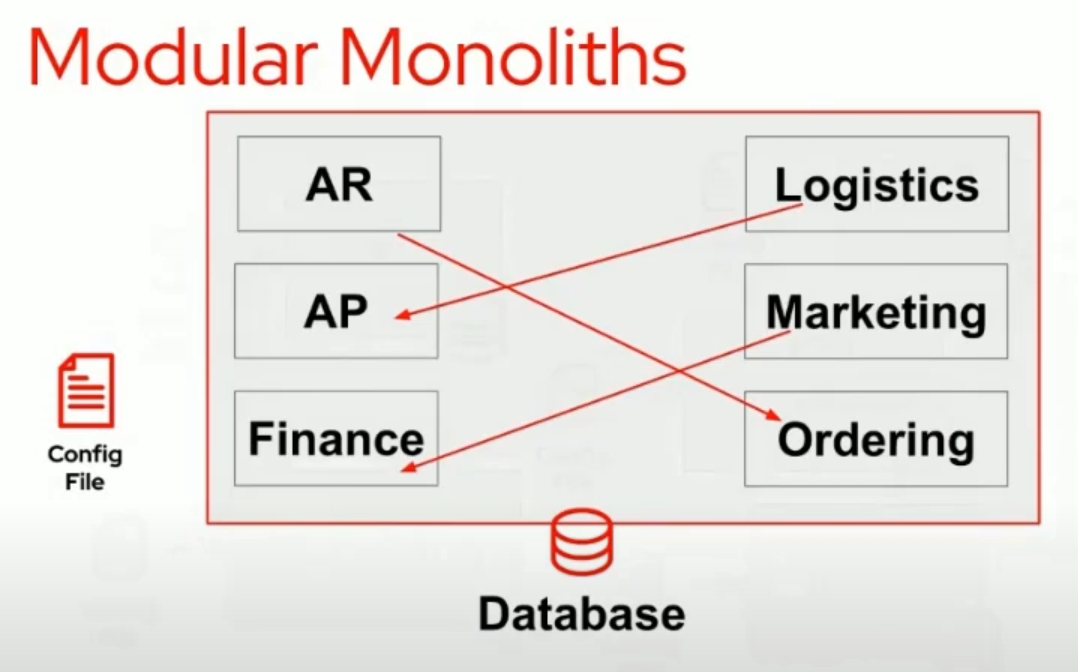
* A module is never completely independent. It has dependencies with other modules, but these should be minimal.
* Modules are interchangeable.
* Code is reusable.
* Better organization of the dependencies compared to traditional monolithic apps.
* Easier to maintain and develop new versions than traditional monolithic apps.
* You can keep the whole project as a single unit, without needing different servers for deployment.
* More scalable than traditional monolithic apps.
* Less complex than microservices architecture.

**Microservices:**

* Services are completely independent; each one has its own dependencies and logic.
* Each service could have a different database (this is optional because they can share the same database, but it's preferred).
* Each service can use a different programming language and technology.
* When there is a change in the code, you can deploy only one service without interrupting or affecting the others.
* CI/CD Cloud-ready.
* More scalable than modular monolithic apps.

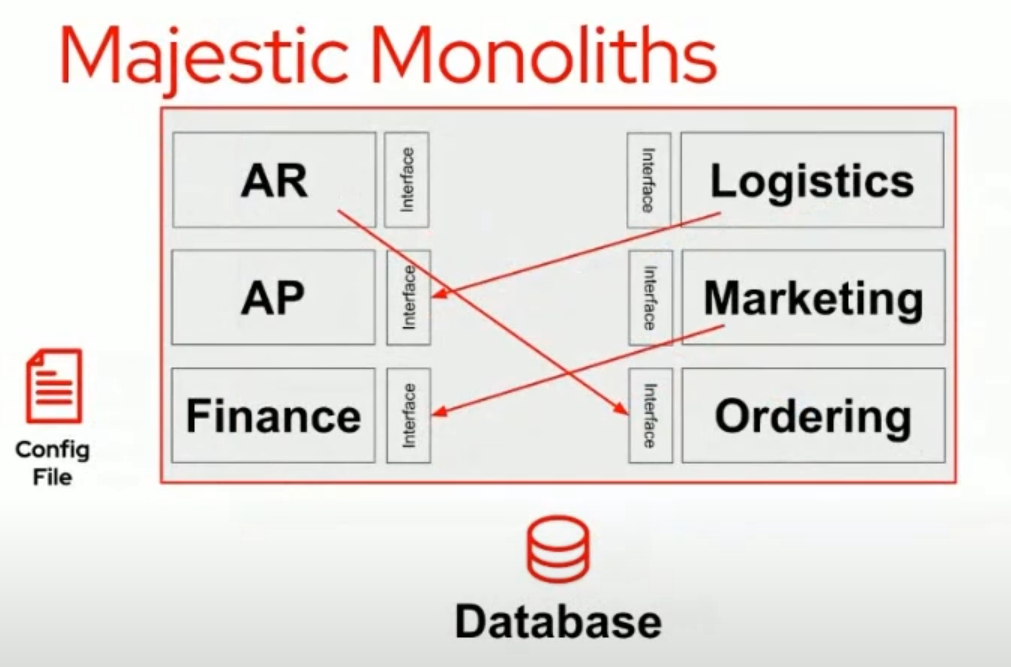
**Модулярный монолит**

1. Каждый модуль имеет свою кодовую базу
2. Каждый модуль может быть развернут как отдельный сервис в TeamCity/Jenkings/… (каждый модуль можно развернуть в Docker – создать Docker image)
3. Каждый модуль может релизиться сам по себе (только если не было существенных изменений в API – например если в Финансы добавили новую опцию, то и добавили новые API запросы, или поменяли тело запроса. Тогда модуль Маркетинга должен быть так же обновлен и развернут), тем самым разработчикам легче писать чистый код, а так же проще деплоить

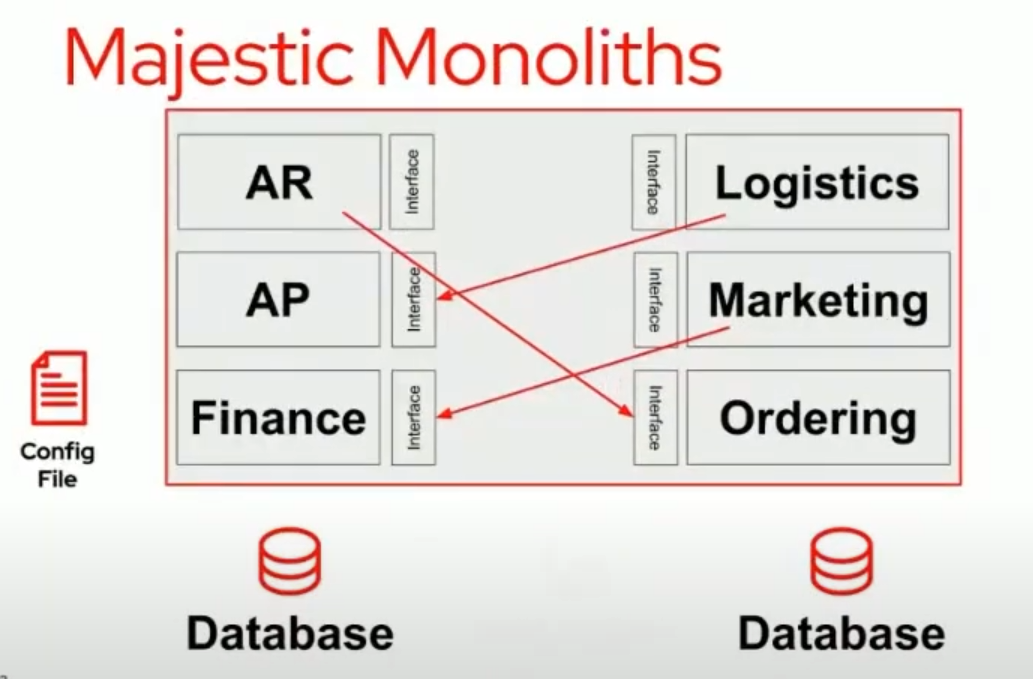


**Магический монолит**

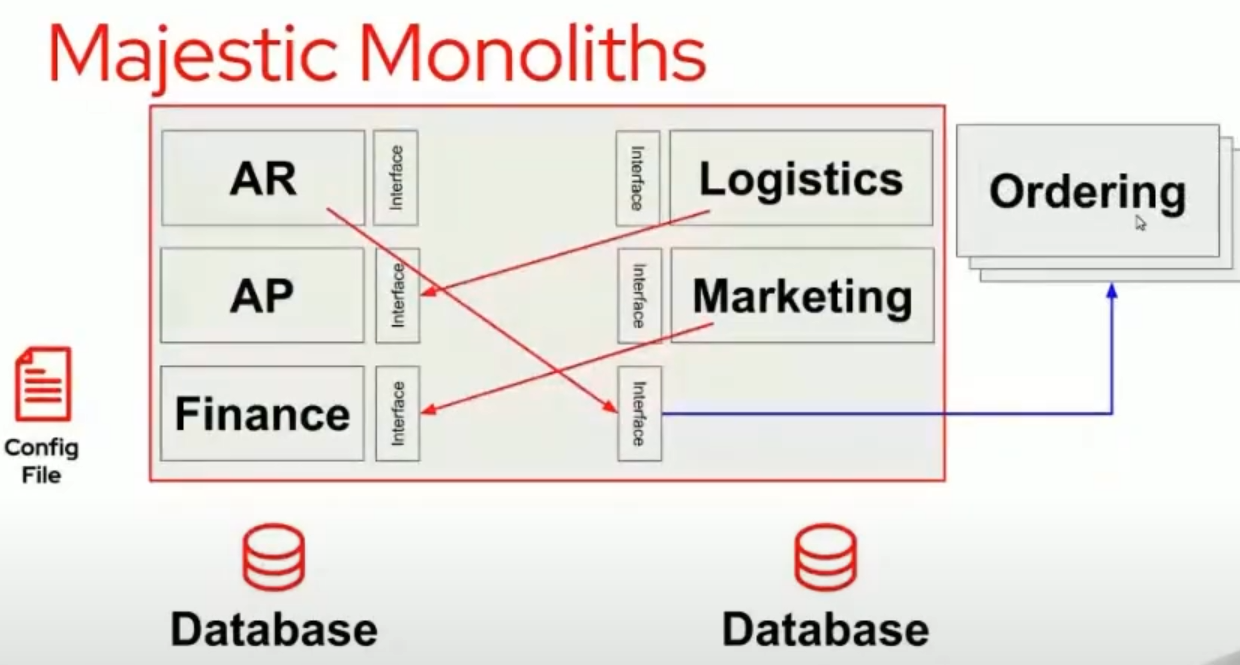
1. Такой тип монолита делает границы модулей (функциональностей) строгими. Например модуль логистики не может напрямую обратиться к модулю финансов, все общение должно пройти через интерфейс. Например в Java 9 можно использовать Java Modules, что бы добиться такого

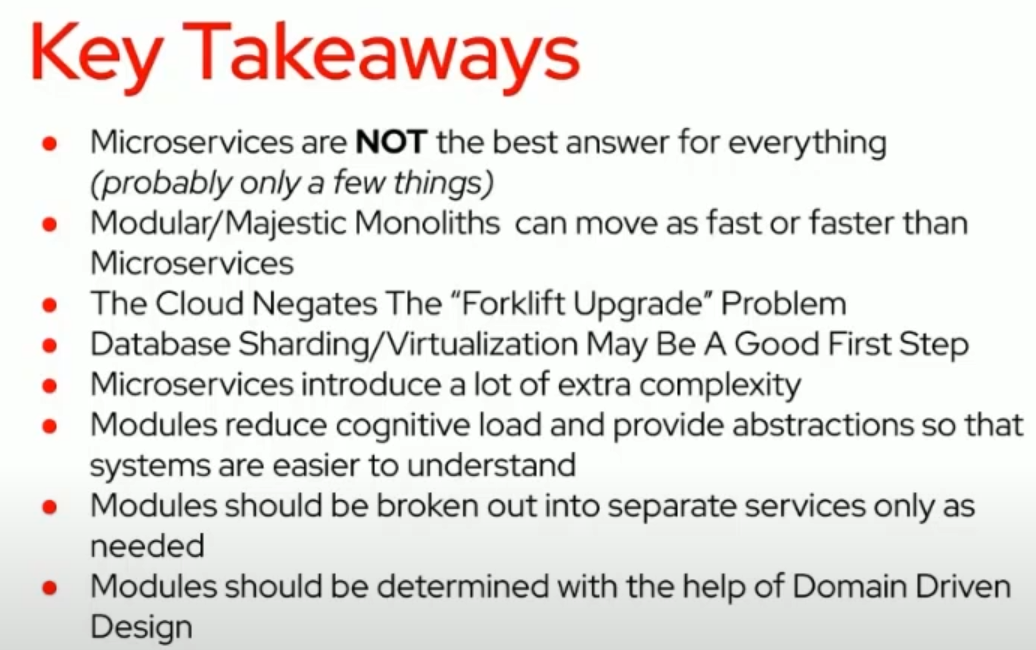


1. Так же можно сделать разделение БД. Например модуль Ordering сильно нагружает Input/Output БД, из за этого можно вынести для этого модуля БД отдельно. Для приложения БД будет выглядеть единой, но Input/Output будет разделен на несколько машин



1. Если перфоманса не хватает, то мы можем начать выделять част ифункциональностей в отдельные микро/макросервисы. До тех пор пока контракты API не изменились, для приложения все будет выглядеть так же как раньше, но теперь мы можем переписать этот модуль на более быстром языке, или отмасштабировать его. Теперь вызовы к интерфейсу являются API вызовами через сеть (REST, MQ, gRPC)





**Вертикальная масштабируемость**

Когда мы выделяем серверу/вирутальной машине, на которой крутится наше приложение больше ресурсов (CPU, RAM)

Есть ряд проблем

1. Невозможно бесконечно наращивать ресурсы
2. Порой увеличение ресурсов не приведет к увеличению производительности (например если есть bottleneck в очереди сообщений и что воркеры не успевают разгребать очередь, то увеличение ресурсов не приведет к увеличению производительности)

**Горизонтальная масштабируемость**

Когда мы добавляем новые сущности текущих сервисов для того что бы больше обрабатывать запросов, а так же быть более отказоустойчивым

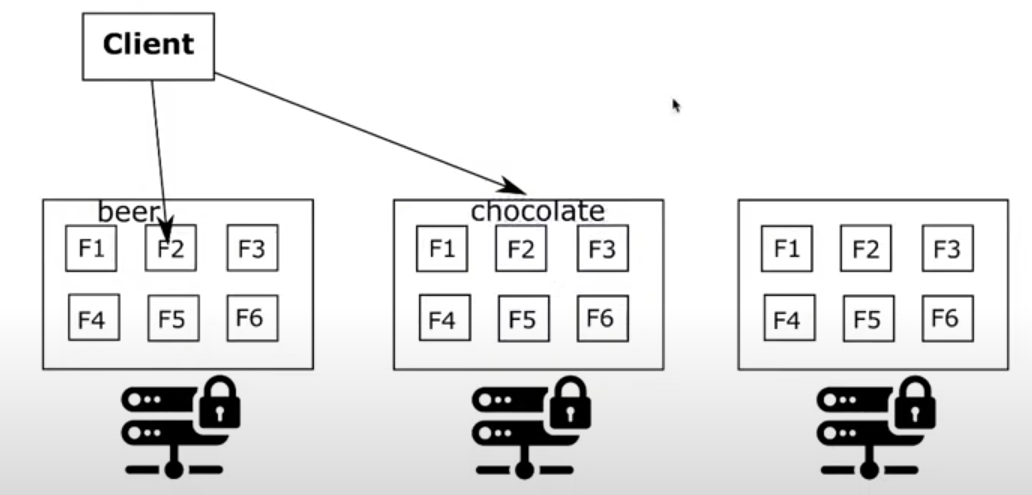
Есть ряд проблем

1. В монолите если мы хотим продублировать какой то сервис (создать новую сущность), то мы должны дублировать весь монолит
2. Если приложение stateful, то оно будет не масштабируемо

**Stateful –** когда приложение хранит данные между различными запросами

Когда клиент отправляет запрос, то один запрос может быть получен одним инстансом приложения, а другой запрос получен другим инстансом приложения

Допустим у нас есть онлайн магазин, а там есть корзина. Каждый запрос от пользователя на добавление товара в корзину, сервер сохраняет выбранный товар в сессии. Допустим клиент добавляет сначало пиво, а потом шоколад в корзину. В результате после перехода в корзину, если запрос придется на первый инстанс монолита, то клиенту отобразиться только пиво, если придется запрос на второй инстанс, то отобразиться только шоколад, а если на третий инстанс, то корзина будет пуста

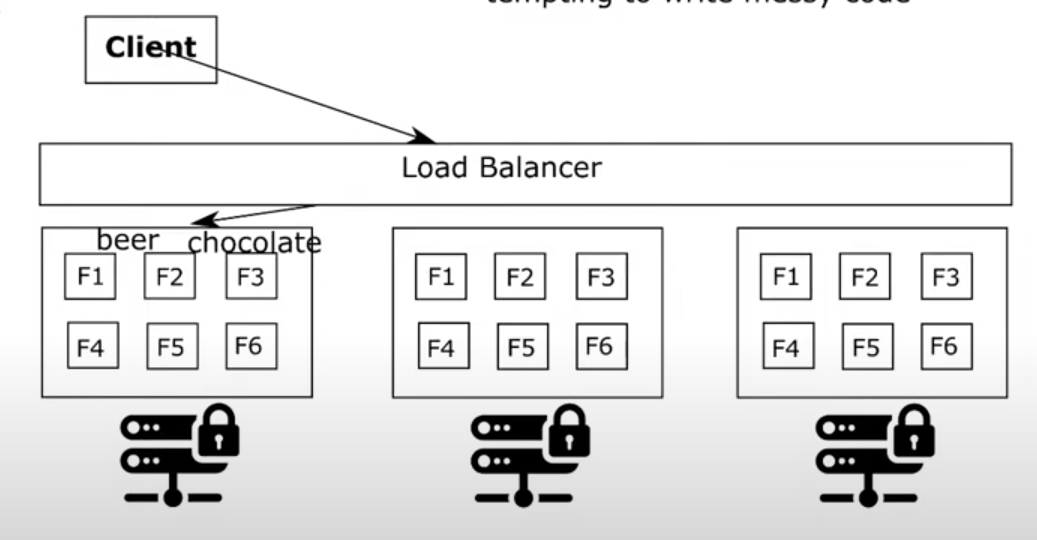


Что бы решить эту проблему, нужно создавать **stateless** приложение – когда процесс никогда не сохраняет данные между запросами

А что делать, если приложение уже **stateful**, как его горизонтально масштабировать?

1. **Sticky Session** in the load balancer (самый популярный)

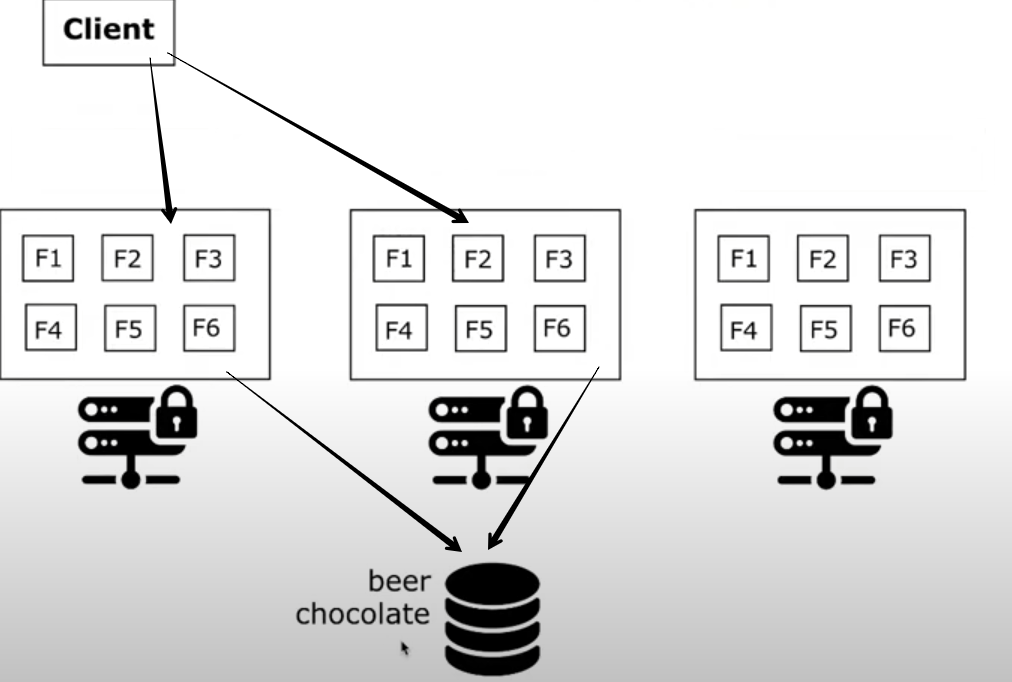
Load Balancer (балансировщик) – доставляет запросы к конкретным сущностям приложения. Sticky Session это способ, при котором балансировщик запоминает клиента, и если от клиента будет идти несколько запросов, то он перенаправляет их к одной сущности приложения. Все запросы все равно идут через 1 балансировщик, но это ничего страшного, так как он по сути только перераспределяет запросы между инстансами приложения, а это не такая трудозатратная задача

****

У этого подхода так же есть недостатки. Например балансировщик может не равномерно распределять пользователей, и на какой то инстанс приложения пойдет запросов больше чем на другой

1. **Blackboarding** (менее попоулярный, так как сложный в настройке и может создать bottleneck в БД)

Когда мы храним данные в одном месте – БД. В данном случае запросы от клиента могут идти на разные инстансы, но сохраняются все изменения не в сессии инстанса, а в БД. Эти данные не хранятся долго, они нужны только для того что бы помнить запросы пользователя. В качестве БД для временного хранения **может использоваться Redis**



1. **Sharing the state through direct communication between the instances** (самый не популярный, так как могуть быть задержки в коммуникации, а так же нужно сохранять информацию запросов на всех инстансах приложения)

Когда все инстансы приложения обмениваются информацией о запросе и сохраняют у себя в сессии

