Overview of Architecture

Архитектура ПО

- Архитектура (ПО) заключает в себе ряд важных решений об организации программной системы, среди которых выбор структурных элементов и их интерфейсов, составляющих и объединяющих систему в единое целое; поведение, обеспечиваемое совместной работой этих элементов; организацию этих структурных и поведенческих элементов в более крупные подсистемы….

Филипп Крачтен (Philippe Kruchten), Грейди Буч (Grady Booch), Курт Биттнер (Kurt Bittner) и Рич Рейтман (Rich Reitman)

Архитектура ПО

- Разделение системы на составные части в самом первом приближении;
- принятие решений, которые трудно изменить впоследствии;

Архитектура ПО

- Архитектура программной или вычислительной системы – это структура или структуры системы, включающие программные элементы, видимые извне свойства этих элементов и взаимоотношения между ними. Архитектура касается внешней части интерфейсов; внутренние детали элементов – детали, относящиеся исключительно к внутренней реализации – не являются архитектурными

Архитектура ПО

- Архитектура программной системы – это способ организации или структурирования компонентов системы, для поддержки определенной функциональности, а также способ их взаимодействия между собой.

Что является архитектурой?

- Является ли некоторое решение архитектурным полностью зависит от того, считают ли разработчики его важным или нет.

Архитектура

- Набор **ВАЖНЫХ** решений
- Но что делает решение **важным**?

Важность решений

- Важность решений напрямую зависит от того, на какое количество соседних компонентов системы оно влияет и насколько его тяжело изменить
- Самые важные решения – это решения, которые необратимы

- Одним из главных отличий архитектуры зданий от программной архитектуры является то, что многие решения, принятые при строительстве сложно изменить. Очень сложно (хотя и возможно) взять и изменить фундамент здания.

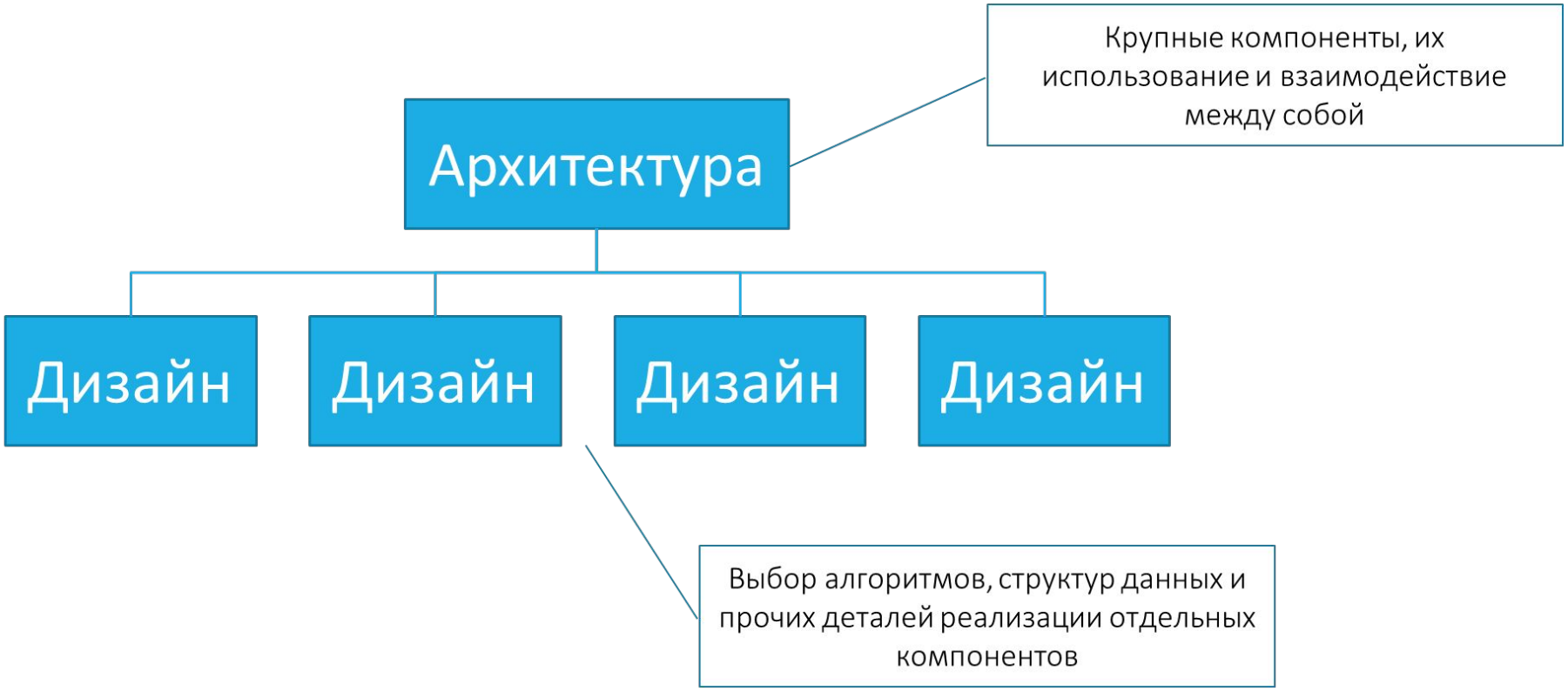
- Но теоретических причин, почему было бы сложно поменять что-либо в программной системе, не существует. Если взять один любой аспект программы, то можно сделать так, чтобы его было легко изменить в будущем. *Проблема в том, что мы не знаем, как сделать так, чтобы легко было изменить любой аспект системы. Когда мы делаем некоторый аспект системы простым в изменении, то вся система при этом становится немного сложнее. Если же мы сделаем любой аспект системы простым в изменении, то это приведет к невероятному усложнению всей системы.* Именно сложность препятствует модификации наших систем. Сложность и дублирование.

- Любое решение должно быть оправдано и продумано его влияние в долгосрочной перспективе
- Преждевременная гибкость, как и преждевременная оптимизация, - зло!
- Добавление гибкости должно происходить в тех местах, где это действительно необходимо, и там, где мы точно знаем, что это окупится

Архитектура ПО

- Архитектура программной системы – это способ организации или структурирования *важных* компонентов системы, для поддержки определенной *важной* функциональности, а также способ их взаимодействия между собой.

Архитектура и дизайн



Архитектура и дизайн

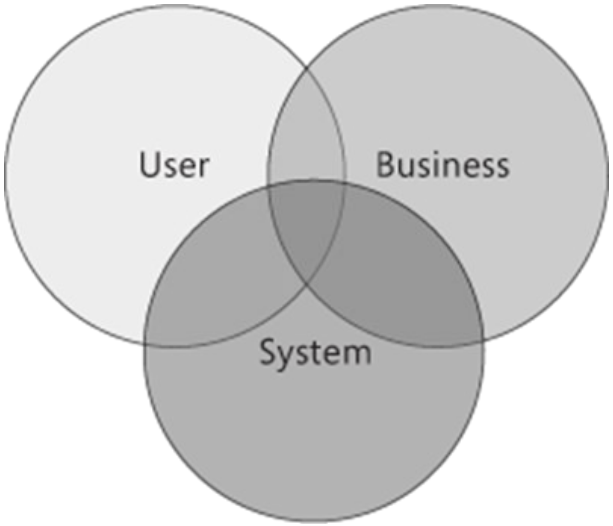
Goals of Architecture

Зачем?

- Только приложение с грамотно проработанной архитектурой может эффективно расти и развиваться.

Принимаемые решения

- Потребности пользователей, ИТ-инфраструктуры и бизнеса



Принимаемые решения

- Для каждой из этих сторон есть свои ключевые сценарии, свои требуемые атрибуты качества (производительность, безопасность, надежность), свои критерии оценки.

- Эти критерии, сценарии и атрибуты, естественно, друг другу противоречат.

Решения зависят

- Как пользователь будет использовать приложение?
- Как приложение будет развертываться и обслуживаться при эксплуатации?
- Какие требования по атрибутам качества, таким как безопасность, производительность, возможность параллельной обработки, интернационализация и конфигурация, выдвигаются к приложению?

Цель проектирования

- Цель архитектуры – выявить требования, оказывающие влияние на структуру приложения.
- Хорошая архитектура снижает бизнес-риски, связанные с созданием технического решения.
- Хорошая структура обладает значительной гибкостью, чтобы справляться с естественным развитием технологий, как в области оборудования и ПО, так и пользовательских сценариев и требований.

Main Points of Architecture

- Дизайн будет эволюционировать со временем
- Невозможно наперед знать все то, что необходимо для проектирования системы
- Новые сведения появляются в ходе разработки и тестирования.
- Создавайте архитектуру, ориентируясь на возможные изменения, чтобы иметь возможность адаптировать их к требованиям, которые в начале процесса проектирования известны не в полном объеме.

- Не пытайтесь создать слишком сложную архитектуру и не делайте предположений, которые не можете проверить.
- Некоторые аспекты дизайна должны быть приведены в порядок на ранних стадиях процесса, потому что их возможная переработка может потребовать существенных затрат.

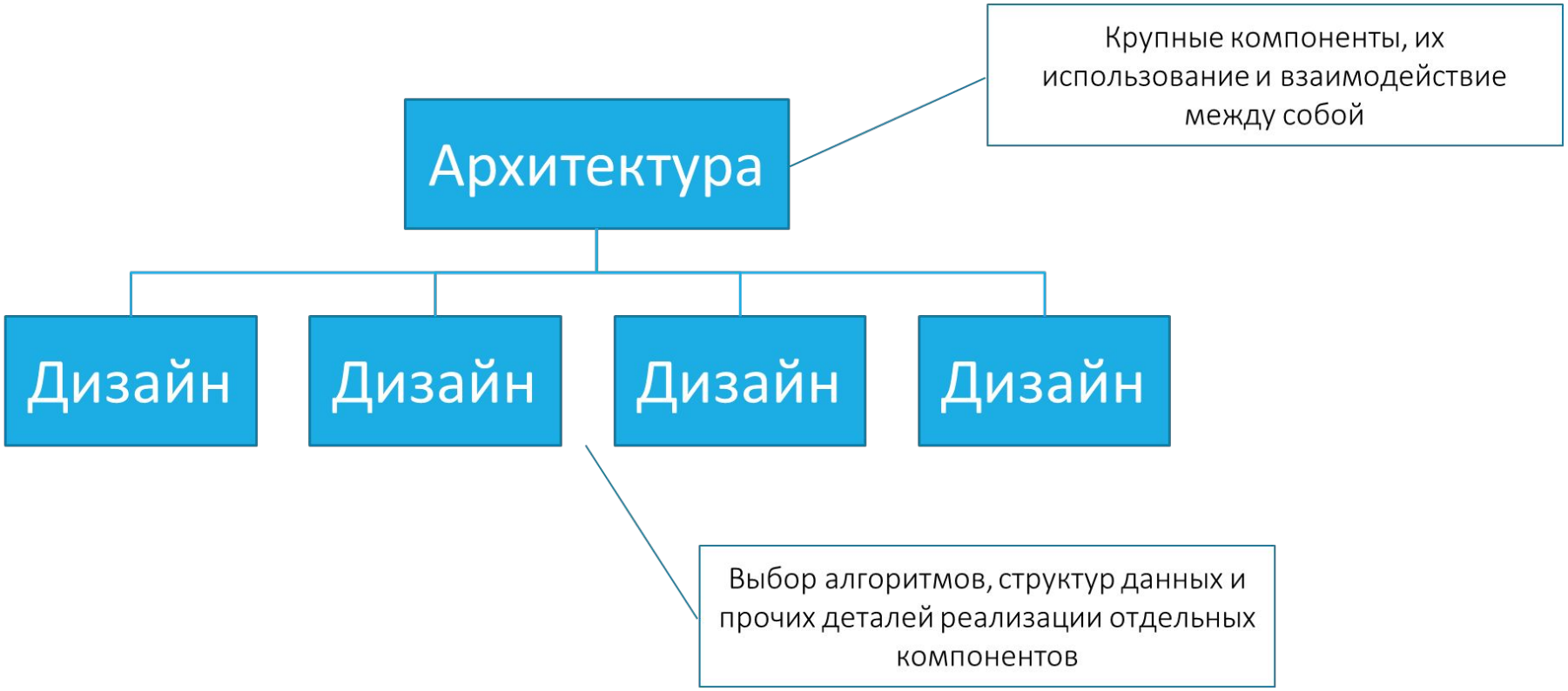
Полезные вопросы

- Какие части архитектуры являются фундаментальными, изменение которых в случае неверной реализации представляет наибольшие риски?
- Какие части архитектуры вероятнее всего подвергнуться изменениям, а также проектирование каких частей можно отложить?
- Основные допущения, и как они будут проверяться?
- Какие условия могут привести к реструктуризации дизайна?

Проверка новой итерации

- Какие допущения были сделаны в этой архитектуре?
- Каким явным или подразумеваемым требованиям отвечает данная архитектура?
- Основные риски при использовании такого архитектурного решения?
- Каковы меры противодействия для снижения основных рисков?
- Является ли данная архитектура улучшением базовой архитектуры или одним из возможных вариантов архитектуры?

Итоги


Крупные компоненты, их использование и взаимодействие между собой
Архитектура
Дизайн
Дизайн
Дизайн
Дизайн
Выбор алгоритмов, структур данных и прочих деталей реализации отдельных компонентов

Итоги

- При проектировании ПО, постоянно приходится идти на компромиссы между противоречивыми требованиями от различных сторон, а так же между простотой и гибкостью

- Важно тщательно оценивать последствия проектных решений, а так же понимать их влияние на различные компоненты системы

Architecture Principles

Принципы проектирования

- **Разделение функций**. Разделите приложение на отдельные компоненты с, по возможности, минимальным перекрытием функциональности. Важным фактором является предельное уменьшение количества точек соприкосновения, что обеспечит высокую связность (*high cohesion*) и слабую связанность (*low coupling*). Неверное разграничение функциональности может привести к сложностям взаимодействия

Принципы проектирования

- Цель проектирования – максимальное упрощение дизайна через его разбиение на функциональные области

Принципы проектирования

- **Принцип единственности ответственности**. Каждый отдельно взятый компонент или модуль должен отвечать только за одно конкретное свойство/функцию или совокупность связанных функций.
- **Information Hiding.** Компоненту или объекту не должны быть известны внутренние детали других компонентов или объектов.

Принципы проектирования

- **Не повторяйтесь (DRY)**. В применении к архитектуре это означает, что функциональность должна быть реализована только в одном компоненте и не должна дублироваться ни в одном другом компоненте.
- **Минимизируйте проектирование наперед. (YAGNI)** Проектируйте только то, что необходимо. *В некоторых случаях, когда стоимость разработки в случае неудачного дизайна очень высоки, может потребоваться полное предварительное проектирование и тестирование*.
  В других – можно избежать масштабного проектирования наперед (big design upfront). Если требования к приложению четко не определены, или существует вероятность изменения дизайна со временем, старайтесь не тратить много сил на проектирование раньше времени.

Принципы проектирования

- **Придерживайтесь единообразия шаблонов проектирования в рамках одного слоя**. По возможности, в рамках одного логического уровня структура компонентов, выполняющих определенную операцию, должна быть единообразной.
  Однако для задач с более широким диапазоном требований может потребоваться применить разные шаблоны, например, для приложения, включающего поддержку бизнес-транзакций и составления отчетов.

Принципы проектирования

- **Применяйте определенный стиль написания кода и соглашение о присваивании имен для разработки**. Поинтересуйтесь, имеет ли организация сформулированный стиль написания кода и соглашения о присваивании имен. Если нет, необходимо придерживаться общепринятых стандартов. В этом случае вы получите единообразную модель, все участники группы смогут без труда работать с кодом, написанным не ими, т.е. код станет более простым и удобным в обслуживании.

Принципы проектирования

- **Учитывайте условия эксплуатации приложения**. Определите необходимые показатели и эксплуатационные данные, чтобы гарантировать эффективное развертывание и работу приложения. Приступайте к проектированию компонентов и подсистем приложений, только имея ясное представление об их индивидуальных эксплуатационных требованиях, что существенно упростит общее развертывание и эксплуатацию.

Architecture Styles

Архитектурные стили

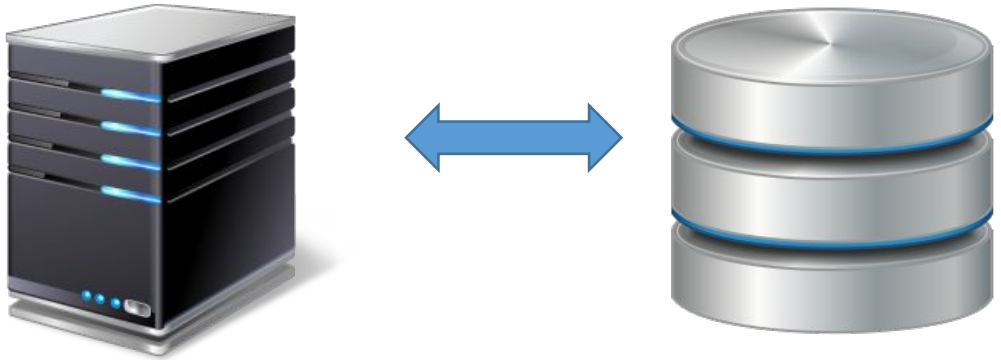
- Архитектурный стиль, иногда называемый архитектурным шаблоном – это набор принципов, высокоуровневая схема, обеспечивающая абстрактную инфраструктуру для семейства систем. Архитектурный стиль улучшает секционирование и способствует повторному использованию дизайна благодаря обеспечению решений часто встречающихся проблем.

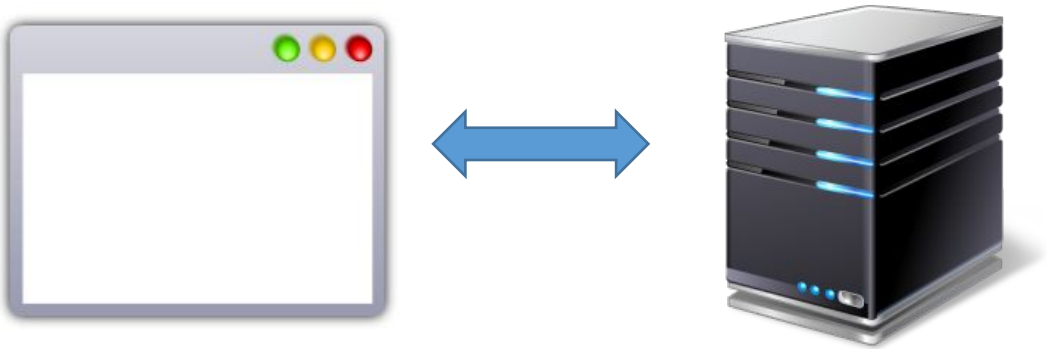
Архитектурные стили

Category	Styles
Deployment	Monolithic, Client\Server, 3-Tier, N-Tier
Structure	Object-oriented, Layered Architecture, Domain Driven Design
Distribution	Service-Oriented Architecture (SOA), Micro-services, Pipeline Architecture

Monolithic style

Client-Server / 2-Tier





Client-Server / 2-Tier

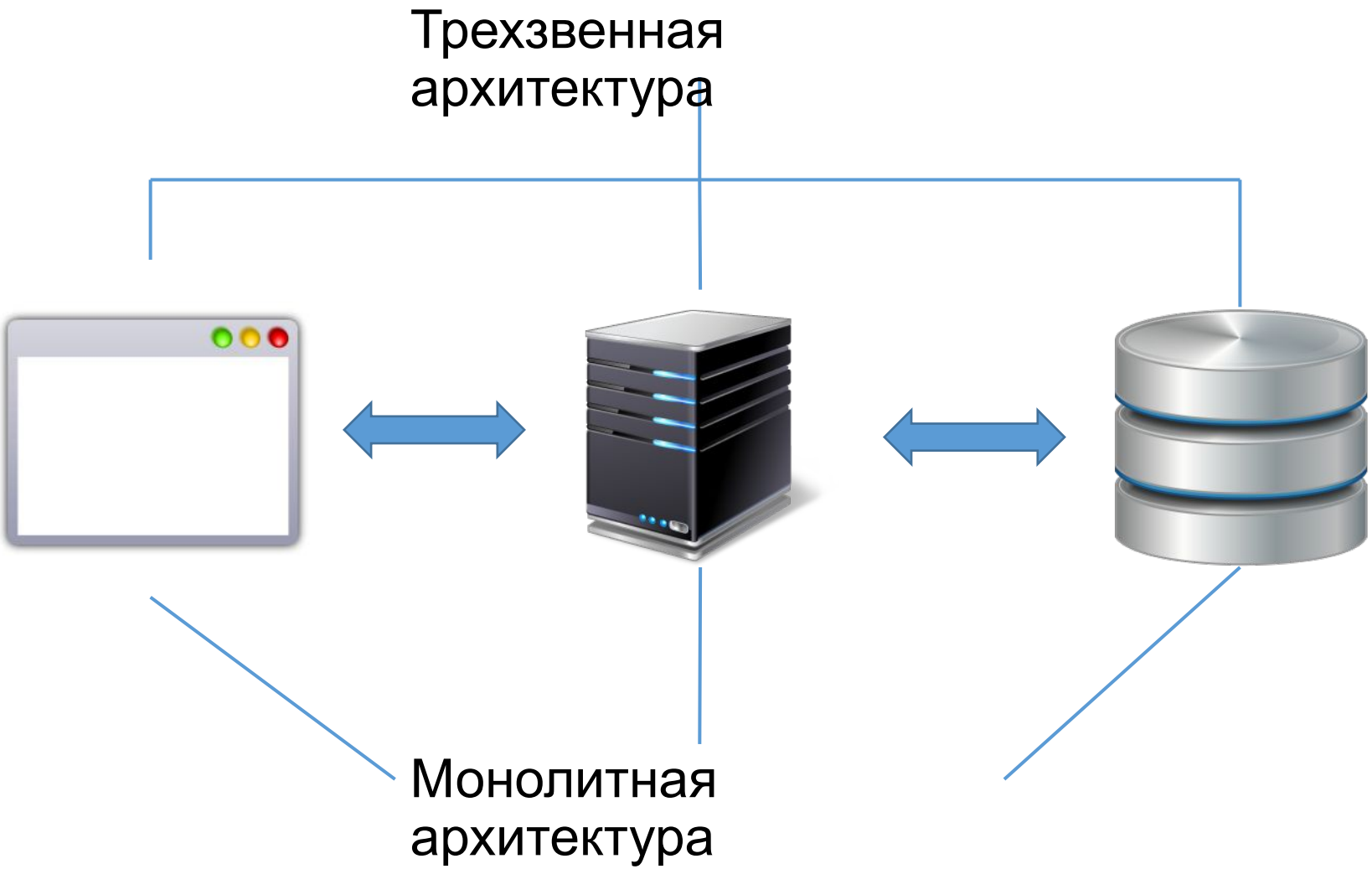
- Серверное приложение, к которому напрямую обращаются множество клиентов
- Исторически – сервер БД, с логикой в хранимках + GUI клиент к ней.
- Веб-приложения;
- Настольные приложения Windows, выполняющие доступ к сетевым сервисам данных;
- Приложения, выполняющие доступ к удаленным хранилищам данных (такие как программы чтения электронной почты, FTP-клиенты и средства доступа к базам данных);

3 – Tier/ N – Tier

3 – Tier/ N – Tier

- **Удобство поддержки**. Уровни не зависят друг от друга, что позволяет выполнять обновления или изменения, не оказывая влияния на приложение в целом.
- **Масштабируемость**. Уровни организовываются на основании развертывания слоев, поэтому масштабировать приложение довольно просто.
- **Гибкость**. Управление и масштабирование каждого уровня может выполняться независимо, что обеспечивает повышение гибкости.
- **Доступность**. Приложения могут использовать модульную архитектуру, которая позволяет использовать в системе легко масштабируемые компоненты, что повышает доступность.

Point of view



Object-oriented style

- Разделение ответственностей приложения или системы на самостоятельные пригодные для повторного использования объекты, каждый из которых содержит данные и поведение, относящиеся к этому объекту.
- Система моделируется как набор взаимодействующих объектов, а не как набор подпрограмм.

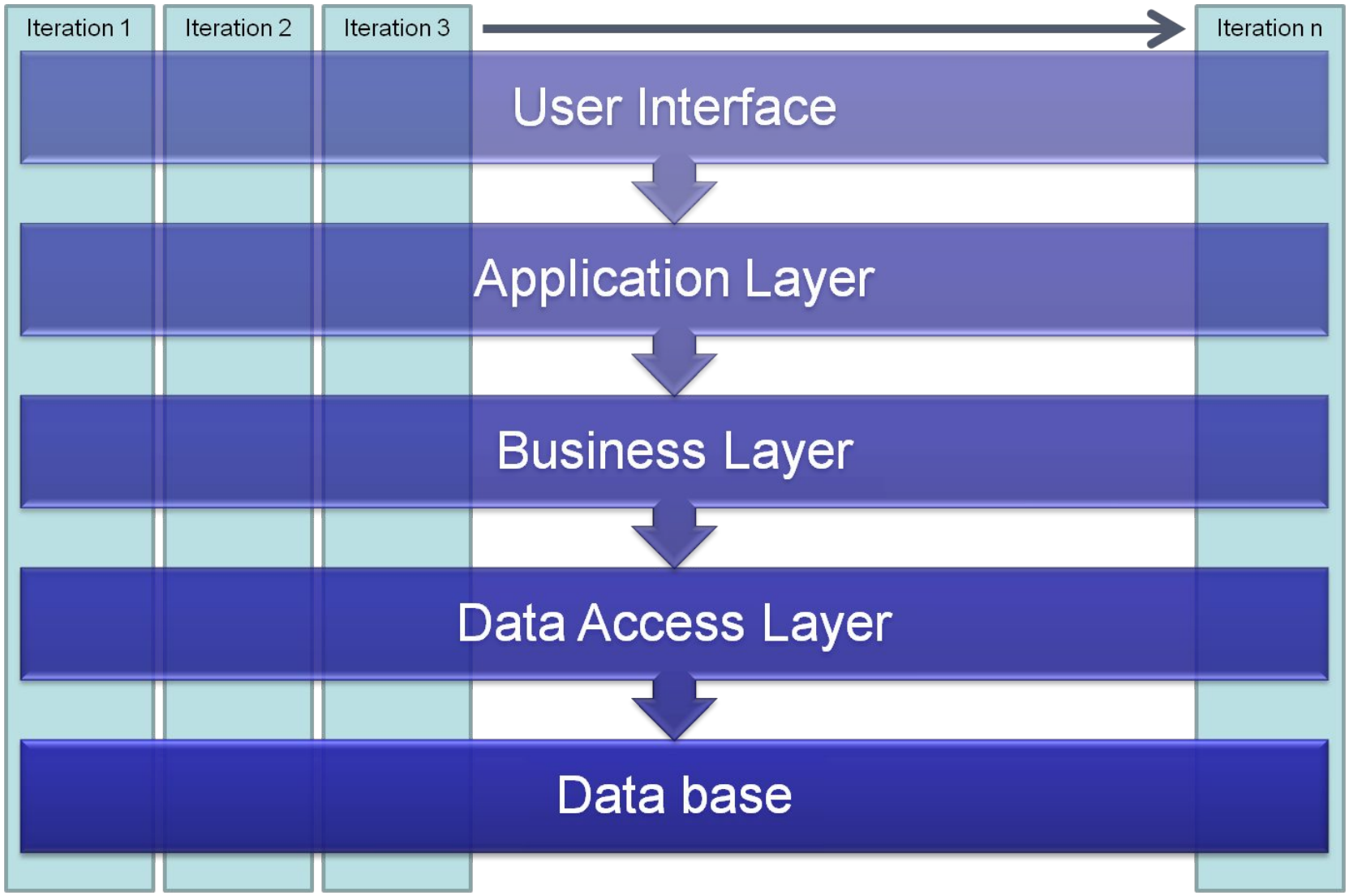
Layered architecture

- Многоуровневая архитектура обеспечивает группировку связанной функциональности приложения в разных слоях, выстраиваемых вертикально, поверх друг друга. Функциональность каждого слоя объединена общей ролью или ответственностью. Слои слабо связаны, и между ними осуществляется явный обмен данными.

Layered architecture

- При строгом разделении на слои компоненты одного слоя могут взаимодействовать только с компонентами того же слоя или компонентами слоя, расположенного прямо под данным слоем

Layered architecture



Layered architecture

- **Абстракция**. Многослойная архитектура представляет систему как единое целое, обеспечивая при этом достаточно деталей для понимания ролей и ответственностей отдельных слоев и отношений между ними.
- **Инкапсуляция**. Во время проектирования нет необходимости делать какие-либо предположения о типах данных, методах и свойствах или реализации, поскольку все эти детали скрыты в рамках слоя.
- **Четко определенные функциональные слои**. Разделение функциональности между слоями очень четкая. Верхние слои, такие как слой представления, посылают команды нижним слоям, таким как бизнес-слой и слой данных, и могут реагировать на события, возникающие в этих слоях, обеспечивая возможность передачи данных между слоями вверх и вниз.
- **Возможность повторного использования**. Отсутствие зависимостей между нижними и верхними слоями обеспечивает потенциальную возможность их повторного использования в других сценариях.

Domain Driven Design

- В качестве ядра ПО выступает модель предметной области, которая является прямой проекцией некого общего языка; с ее помощью путем анализа языка группа разработки быстро находит пробелы в ПО.

Domain Driven Design

«Обратные» связи

- Например, предположим, что из *Сценария* нужно обратиться к *Представлению*. Однако, этот вызов обязан быть не прямым, чтобы не нарушать *Правило Зависимостей* — внутренний круг не знает ничего о внешнем. Таким образом *Сценарий* вызывает интерфейс (на схеме показан как Выходной Порт) во внутреннем круге, а *Представление* из внешнего круга реализует его.
- Обычно мы решаем это кажущееся противоречие с помощью *Принципа Инверсии Зависимостей*.

Итоги

- **Независимость от фреймворка**. Архитектура не зависит от существования какой-либо библиотеки. Это позволяет использовать фреймворк в качестве инструмента, вместо того, чтобы втискивать свою систему в рамки его ограничений.
- **Тестируемость**. Бизнес-правила могут быть протестированы без пользовательского интерфейса, базы данных, веб-сервера или любого другого внешнего компонента.
- **Независимость от UI**. Пользовательский интерфейс можно легко изменить, не изменяя остальную систему. Например, веб-интерфейс может быть заменен на консольный, без изменения бизнес-правил.
- **Независимость от базы данных**. Вы можете поменять Oracle или SQL Server на MongoDB, BigTable, CouchDB или что-то еще. Ваши бизнес-правила не связаны с базой данных.
- **Независимость от какого-либо внешнего сервиса**. По факту ваши бизнес правила просто ничего не знают о внешнем мире.

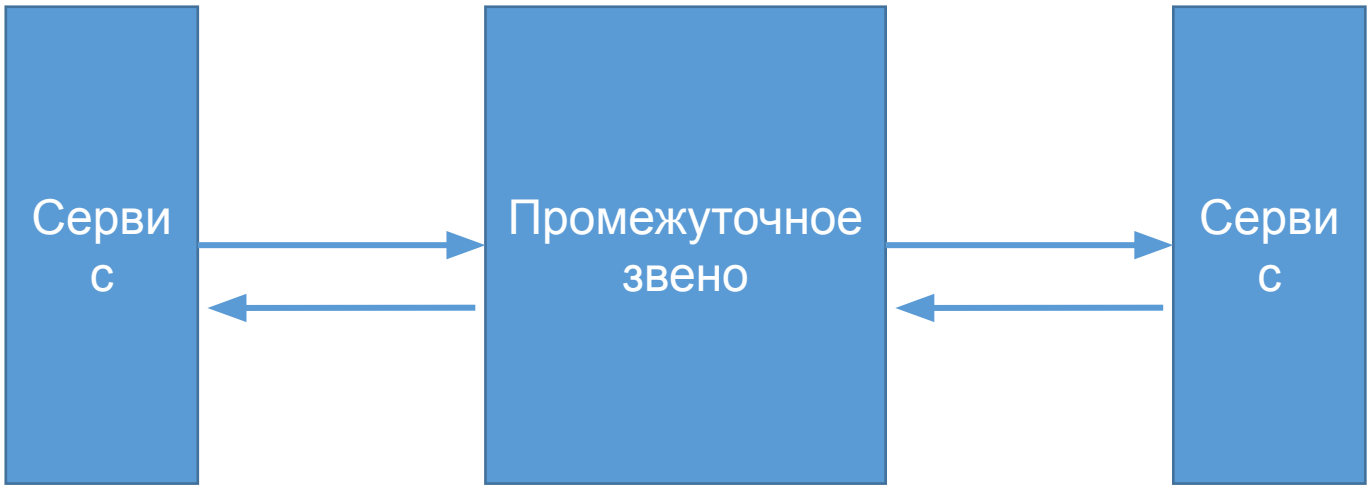
Distributed Styles

Способы взаимодействия приложений

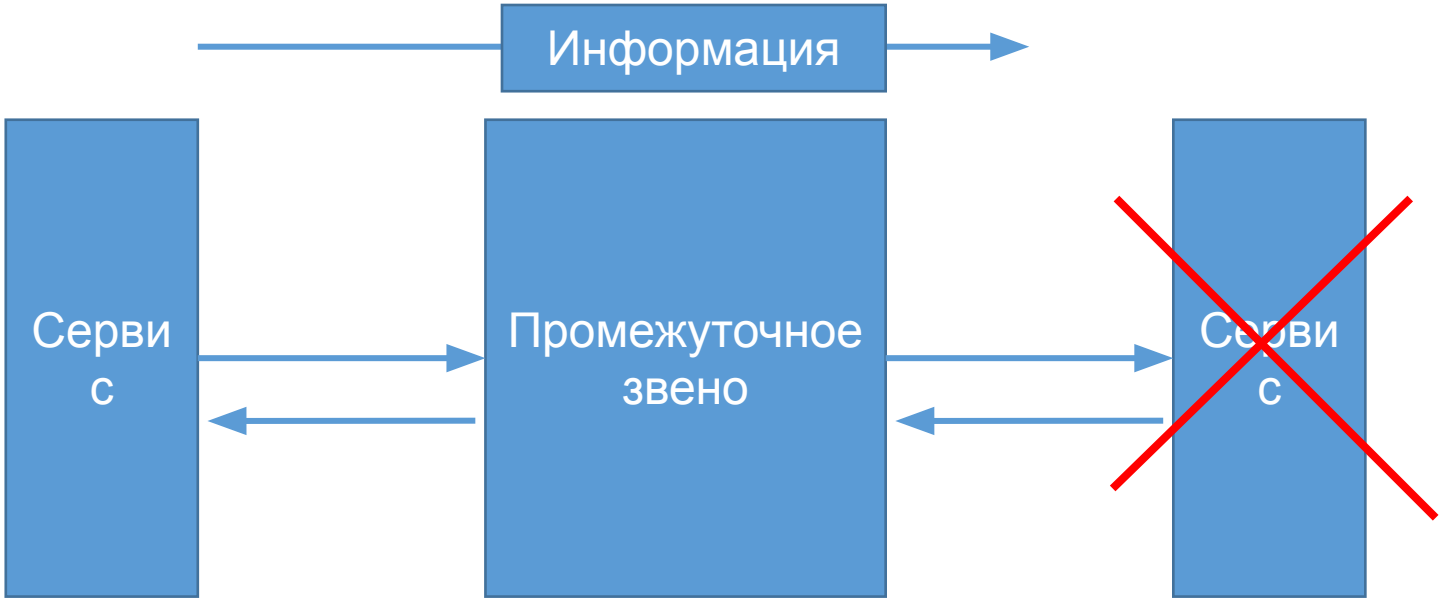
- Файлы
- СУБД
- Сообщения
- Удаленный вызов

Способы взаимодействия приложений

Способы взаимодействия приложений



Способы взаимодействия приложений



Способы взаимодействия приложений

- Удаленный вызов
- REST, SOAP, CORBA
- Синхронный – ожидает ответа
- Идет напрямую к адресату, без посредников

- Используется там, где ответ важен здесь и сейчас

Способы взаимодействия приложений

- Остальные
- Асинхронный – отправили запрос и забыли
- Через промежуточное звено, информация не пропадает, если адресат недоступен

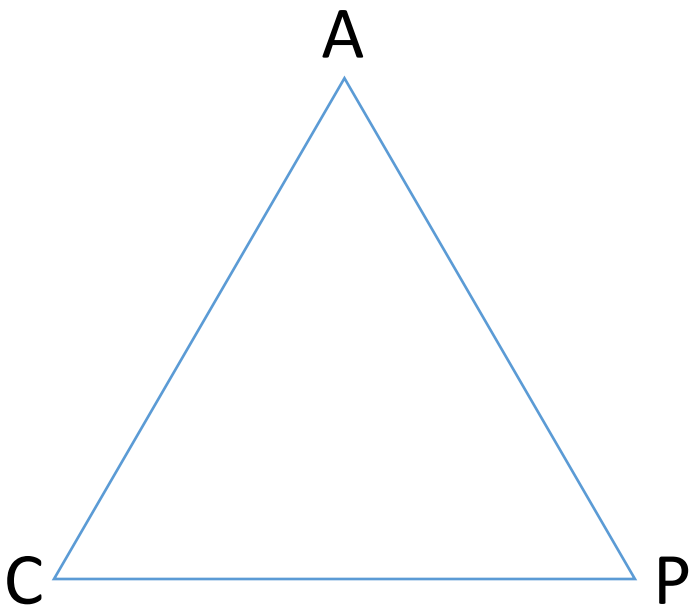
- Используется там, где нужна гарантия доставки*, где ответ сразу не требуется

CAP теорема

- **C**onsistency (Согласованность).
- **A**valability (Доступность).
- **P**artition Tolerance (Устойчивость к разделению системы).

CAP теорема

- Любые два из этих трех



CAP теорема

- Любая отправка данных во вне – **это разделение**
- Чтение из БД и отображение пользователю – потенциальное нарушение согласованности, потому что данные в БД уже могли поменяться

CAP теорема

- Использование посредника для передачи информации – потеря согласованности данных

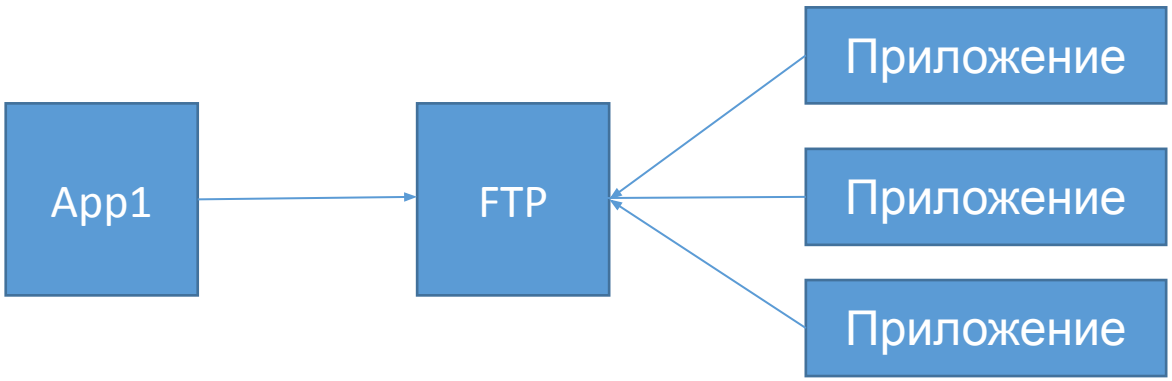
Кроме:
БД

Файлы

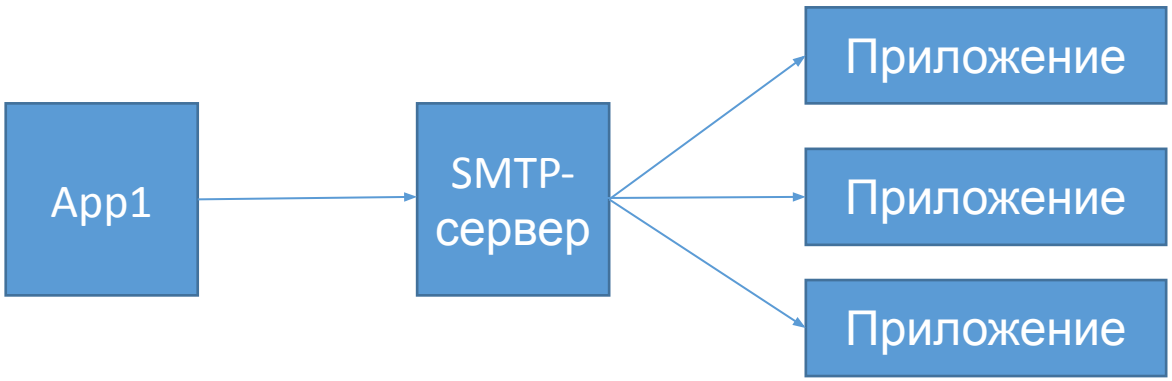
+ Поддержка везде
+ Простота передачи

- Актуальность
- Согласованность данных и форматов
- Нет доступа к общим функциям

Producer-Consumer на файлах



Pub-Sub на файлах

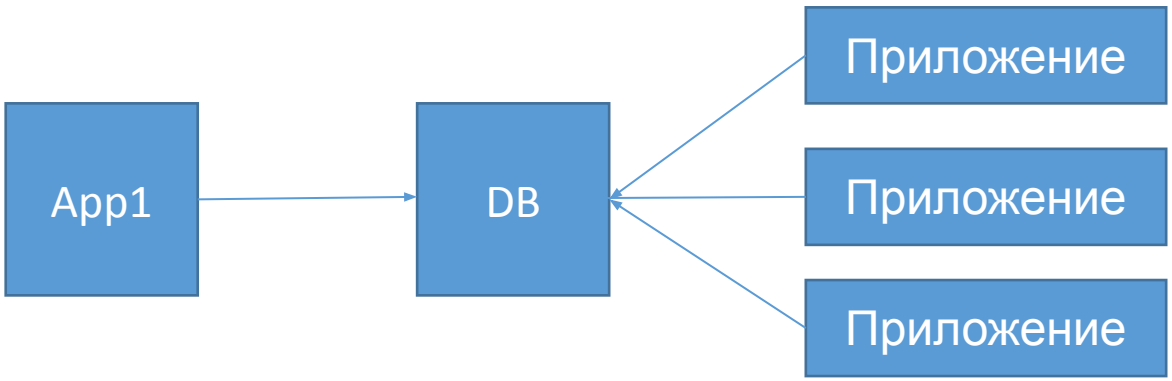


СУБД

+ Поддержка почти везде
+ Единый язык запросов
+ Инструментарий СУБД
+ Согласованность данных

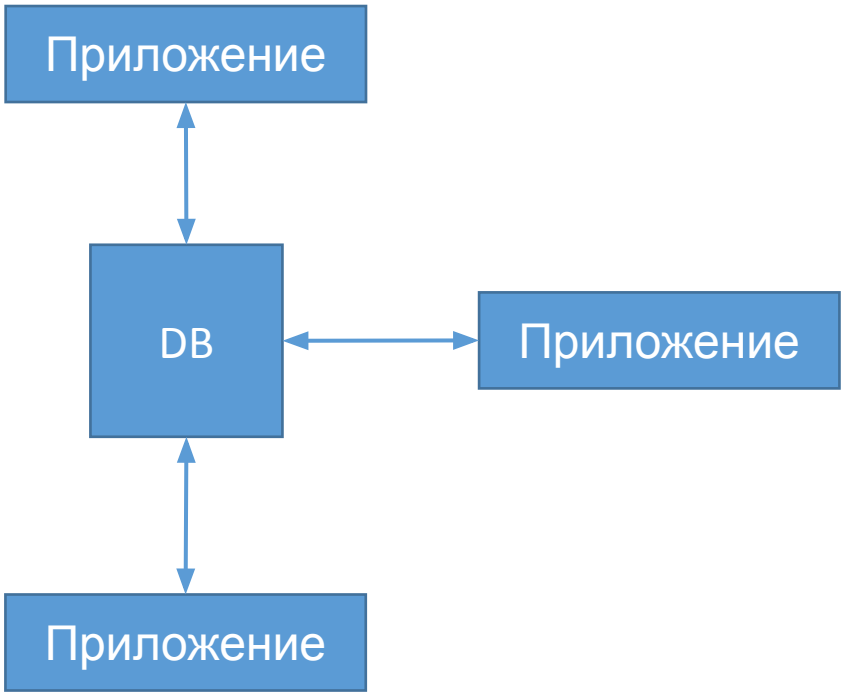
- Зависимость от схемы данных
- Сложность масштабирования
- Узкое место в производительности
- Нет доступа к общим функциям

Producer-Consumer на БД



Id	Field	Field	IsProcessed

Машина состояний в БД



Id	Field	Field	StateId

Вызов процедур

+ Привычный способ работы с вызовом процедур
+ Стандартизация

- Снижает скорость обработки

Очереди сообщений

+ Асинхронный обмен данными
+ Регулирование нагрузки
+ Гибкость настройки и богатый функционал маршрутизации

- Сложная модель программирования
- Узкое место в производительности
- Несогласованность данных

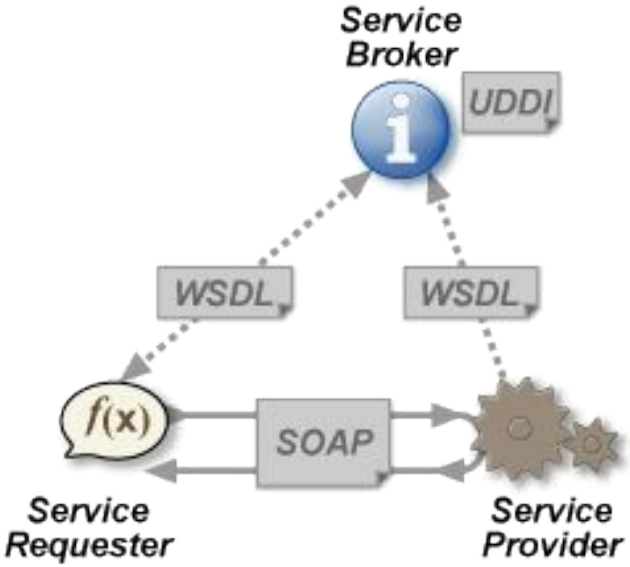
References

- Грегор Хоп, Бобби Вульф. Шаблоны интеграции корпоративных приложений

Services

Классические Service (SOA)

- SOAP (Simple Object Access Protocol)
- WSDL (Web Services Description Language)



WSDL

Основанный на XML язык описания веб сервисов и доступа к ним

Содержит:

- определение типов данных (types) — определение вида отправляемых и получаемых сервисом XML-сообщений
- элементы данных (message) — сообщения, используемые web-сервисом
- абстрактные операции (portType) — список операций, которые могут быть выполнены с сообщениями
- связывание сервисов (binding) — способ, которым сообщение будет доставлено

WSDL

```
<!-- Abstract type -->
<types>
    <xs:schema>
        <xs:element name="request"> ... </xs:element>
        <xs:element name="response"> ... </xs:element>
    </xs:schema>
</types>

<!-- Abstract interfaces -->
<interface name="Interface1">
    <fault name="Error1" element="tns:response"/>
    <operation name="Get" pattern="http://www.w3.org/ns/wsdl/in-out">
        <input messageLabel="In" element="tns:request"/>
        <output messageLabel="Out" element="tns:response"/>
    </operation>
</interface>
```

WSDL

```
<!-- Concrete Binding Over HTTP -->
<binding name="HttpBinding" interface="tns:Interface1"
type="http://www.w3.org/ns/wsdl/http">
  <operation ref="tns:Get" whttp:method="GET"/>
</binding>

<!-- Concrete Binding with SOAP-->
<binding name="SoapBinding" interface="tns:Interface1">
  <operation ref="tns:Get" />
</binding>

<!-- Web Service offering endpoints for both bindings-->
<service name="Service1" interface="tns:Interface1">
  <endpoint name="HttpEndpoint" binding="tns:HttpBinding"
address="http://www.example.com/rest/"/>
  <endpoint name="SoapEndpoint" binding="tns:SoapBinding"
address="http://www.example.com/soap/"/>
</service>
```

SOAP

Основанный на XML протокол для RPC, расширенный позднее для обмена произвольными сообщениями

Работает поверх SMTP, FTP, **HTTP**, HTTPS и др.

Запрос:

```
<soap:Envelope>
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>12345</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

SOAP

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productID>12345</productID>
        <productName>Стакан граненый</productName>
        <description>Стакан граненый. 250 мл.</description>
        <price>9.95</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

Факты

- SOA – это философия дизайна, а не технология
- **Веб**-сервисы не требуются для SOA

Главное

- Выделите функциональность в отдельные независимые и небольшие компоненты (модули, сервисы)
- Определите способы взаимодействия сервисов между собой и сервисов с остальной системой
- Четко определите интерфейс (контракт) каждого сервиса
- Полагайтесь только на интерфейс

Microservices

Microservices architectural style

- SOA – слишком «философское» и размытое понятие под которым скрывается целая группа архитектурных стилей.
- Microservices – ещё один SOA-стиль со своими особенностями.

- Предложен:
  - Fred George
  - James Lewis and Martin Fowler
  - Stefan Tilkov

Microservices

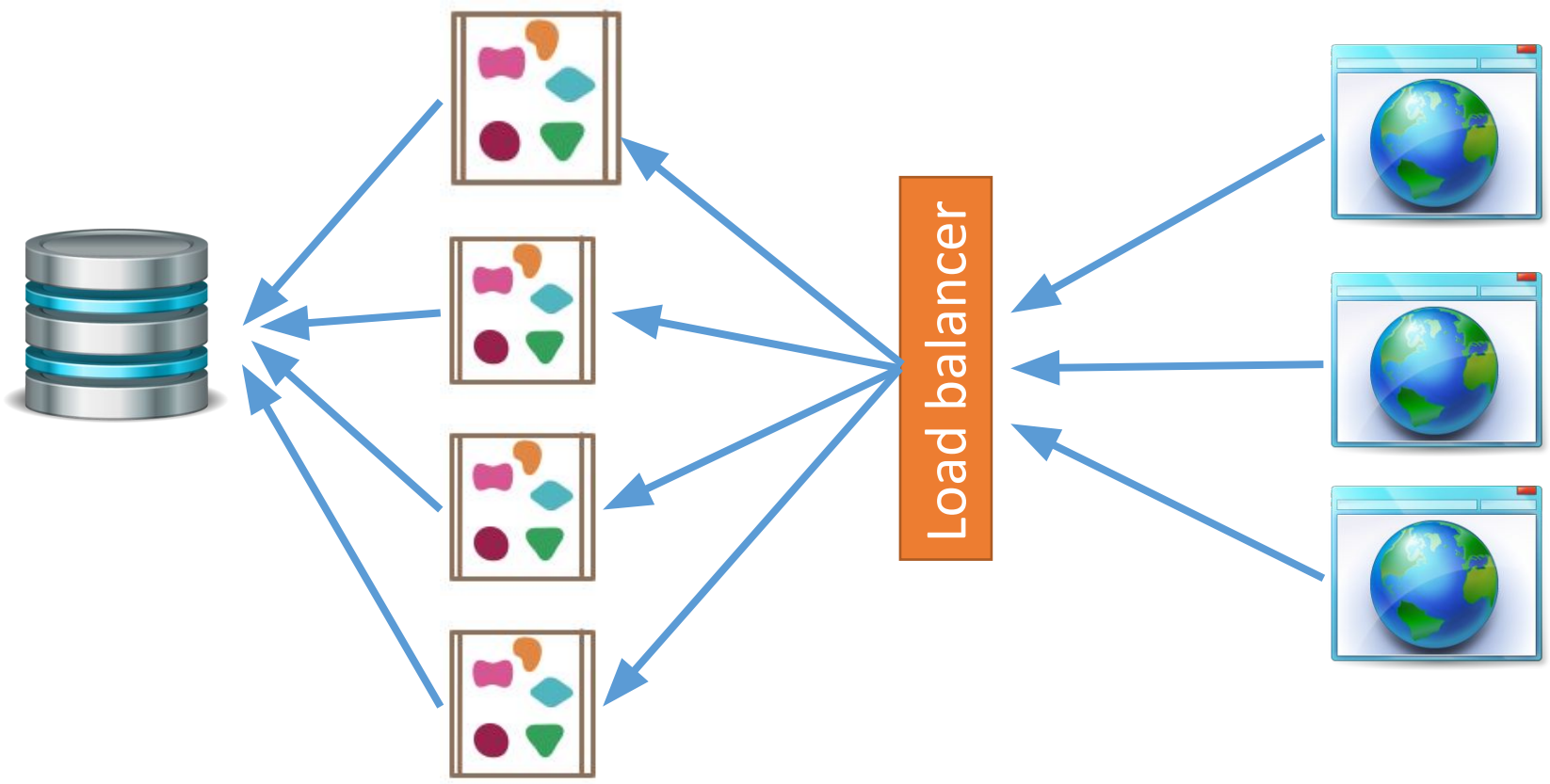
> *…the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API*
>
> - *Martin Fowler &* James Lewis

- Microservices architectural style – подход к разработке одного приложения как набора небольших сервисов, каждый из которых работает в собственном процессе и взаимодействует с другими с помощью простых механизмов, чаще всего HTTP API

http://martinfowler.com/articles/microservices.html

Monolithic style

Monolithic style



Monolithic style

- Естественный путь развития приложения
- Вполне может быть успешным

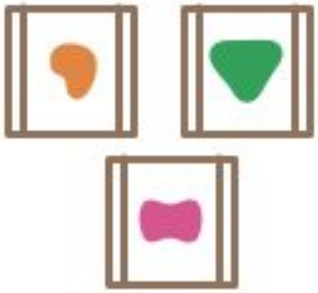
Примеры разочарований:

- Меняем формулу расчета в одном классе, а перезаливать приходится всё приложение
- Тормозит только одна часть, а масштабировать приходится все приложение
- Иногда для какой-то задачи правильнее выбрать другой язык и другую платформу

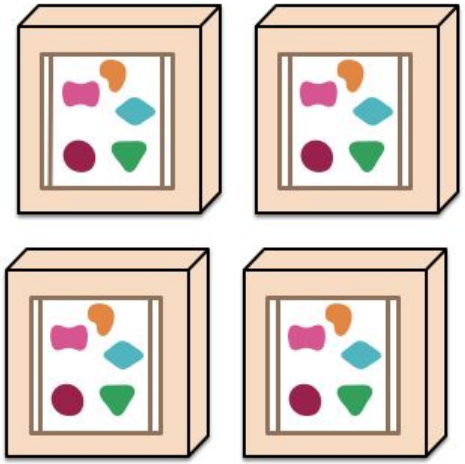
Comparison. Deployment

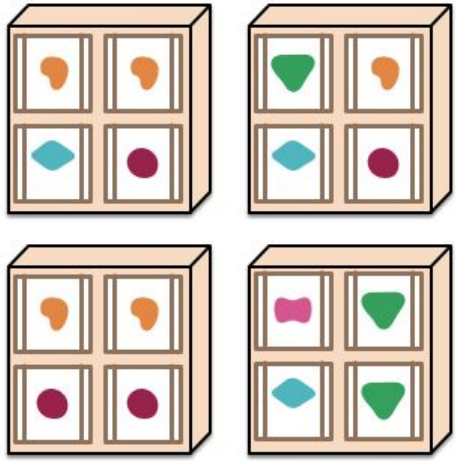
- Все вместе

- Каждый Сервис отдельно

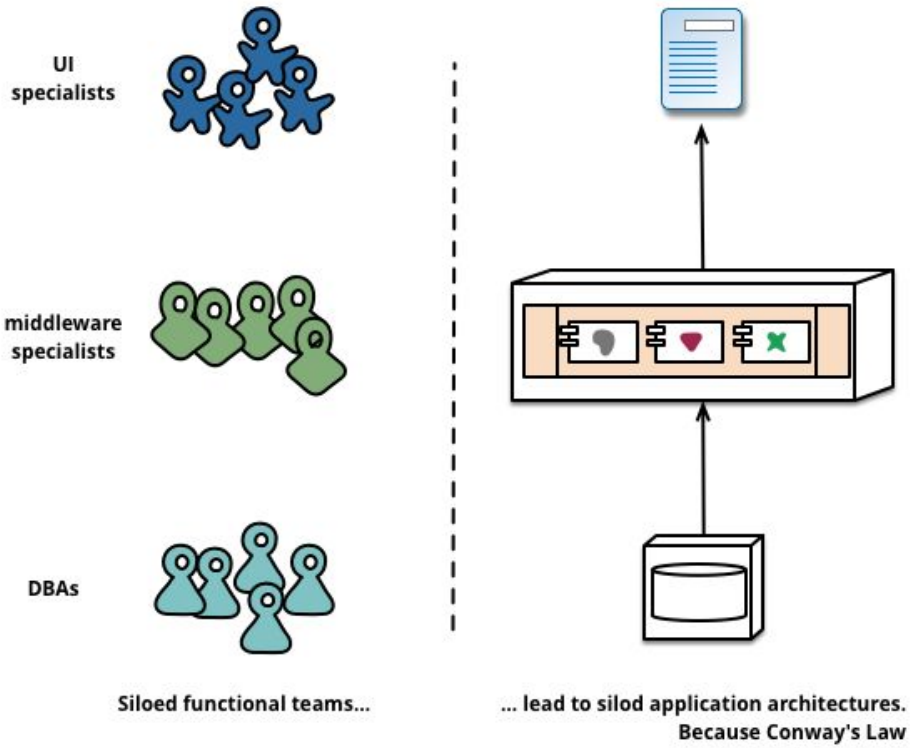


Comparison. Scalability

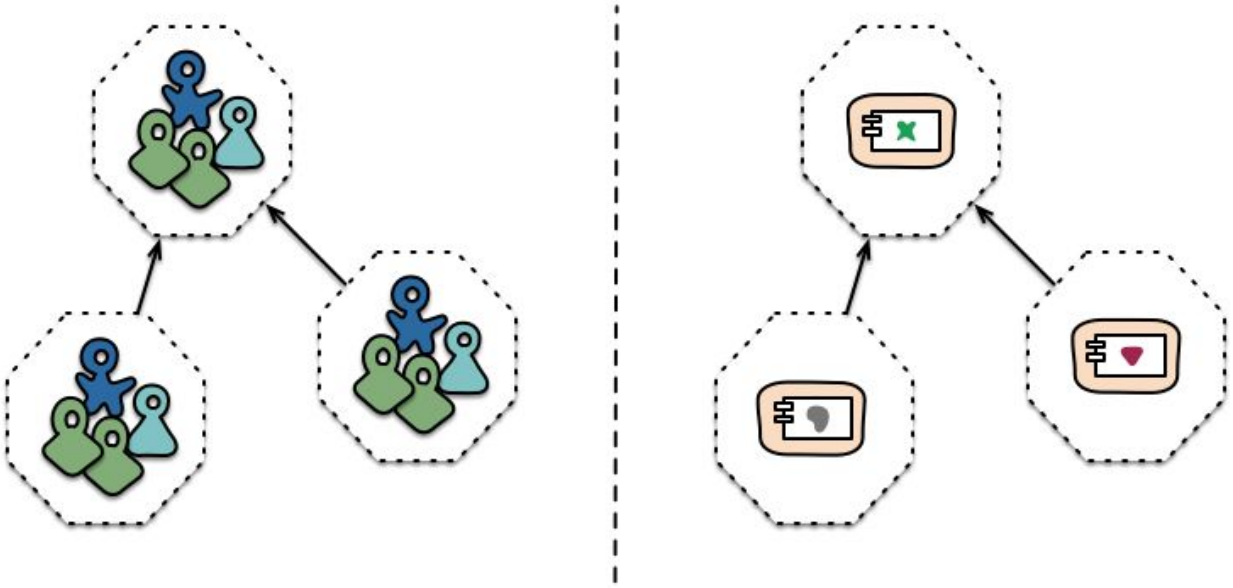




Монолит, разработка



Микросервисы, разработка



Cross-functional teams...

... organised around capabilities
Because Conway's Law

Коммуникация

Простые способы, но умные сервисы

- Microservices стиль тяготеет к использованию простых технологий, с вынесением всей логики в сами сервисы.

- REST via HTTP
- RabbitMQ or ZeroMQ

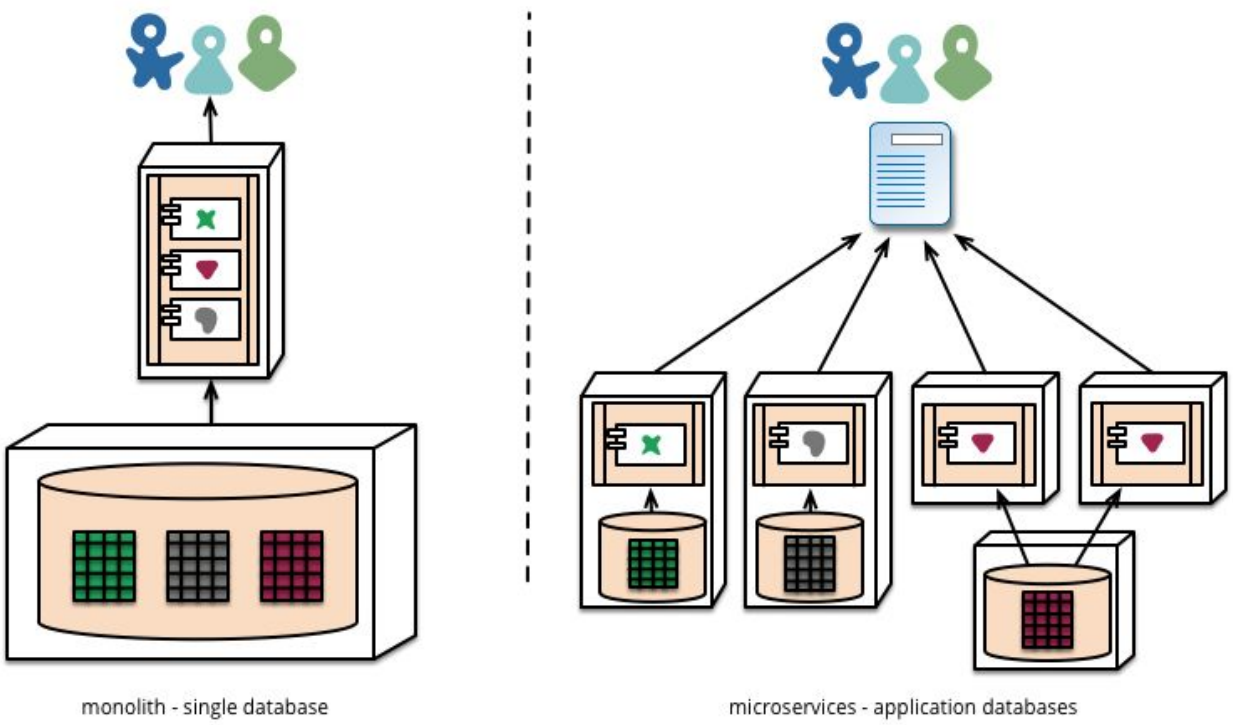
Децентрализация

Каждой задаче свой инструмент

- Microservices подход позволяет создавать приложение используя сервисы разработанные с помощью различных технологий и платформ, в том используя готовые open-source решения

Монолит vs. Микросервисы

Работа с БД



Монолит vs. Микросервисы

Работа с БД

- Распределение данных ведет к невозможности транзакций и потере согласованности.
- Можно рассчитывать только на то, что данные будут согласованы в «конечном итоге» (eventual consistency)
- Применимо только там, где стоимость исправления ошибки меньше, чем работа по поддержанию согласованности.

Разработка с учетом отказа

- Каждый запрос к сервису может завершиться неудачно из-за отказа сервиса
- Отказ одного элемента не должен приводить к остановке всей системы
- Мониторинг становится критически важным!

Примеры

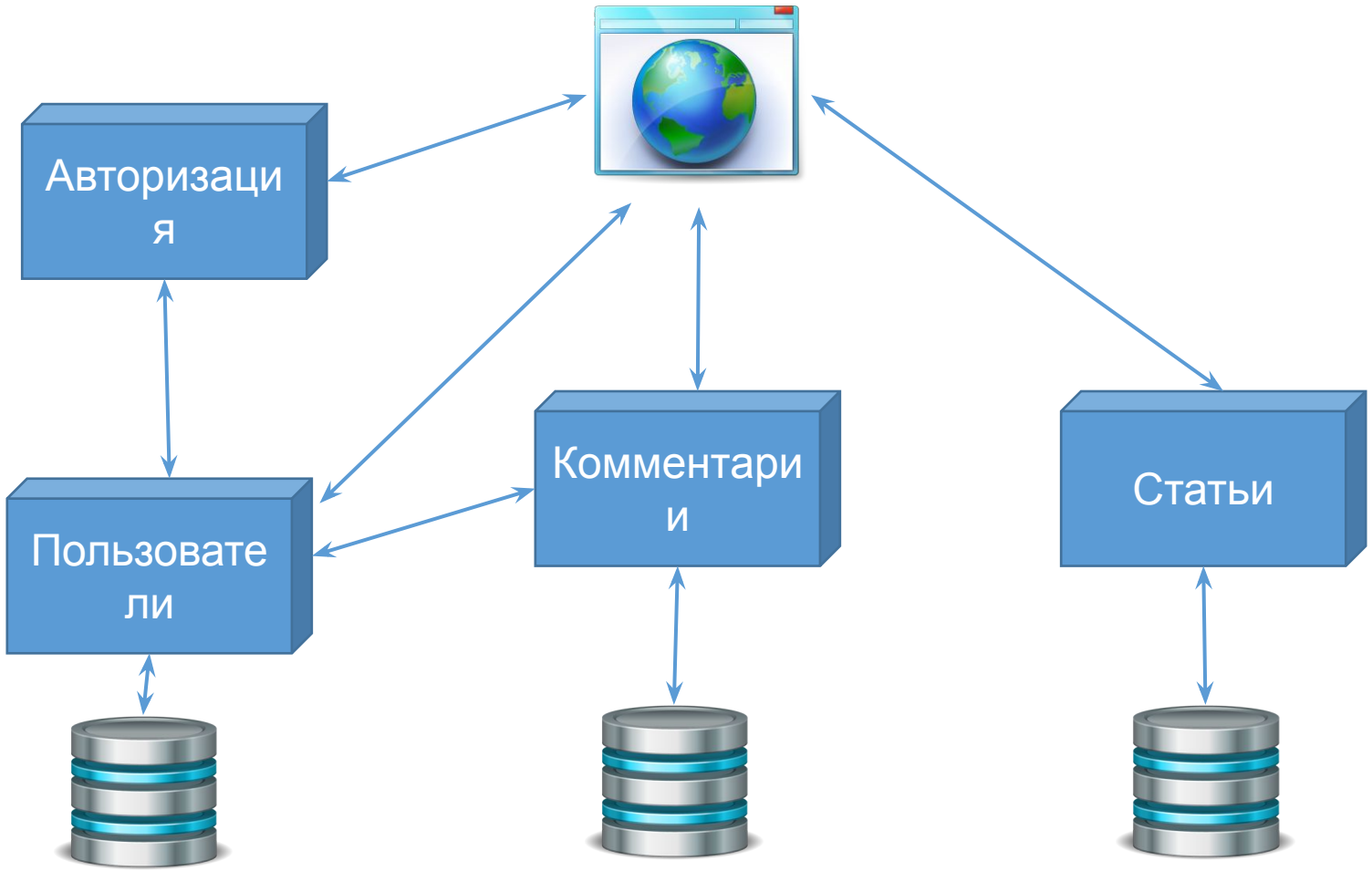
Как построить

Пользователи

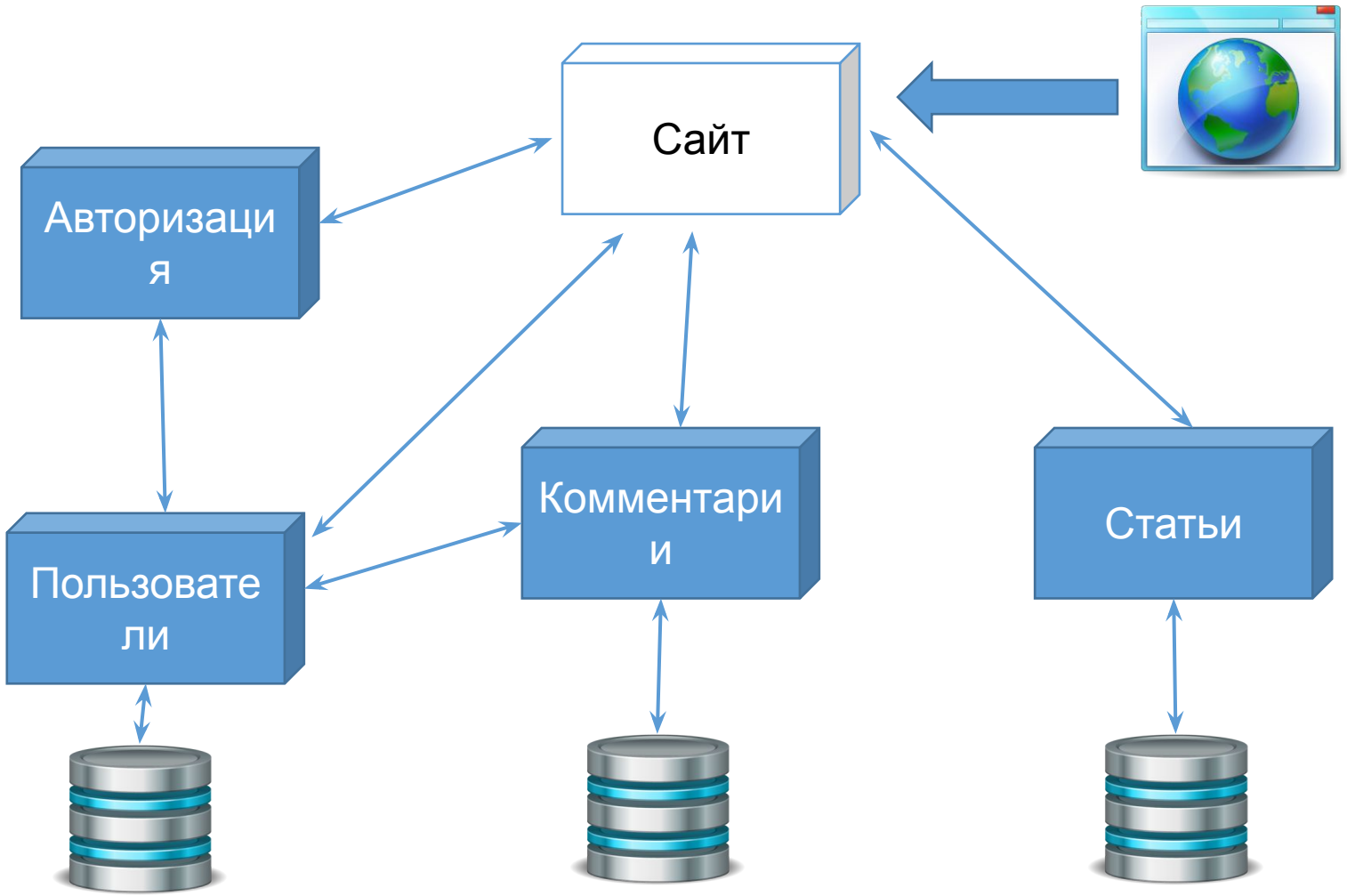
Комментарии

Статья

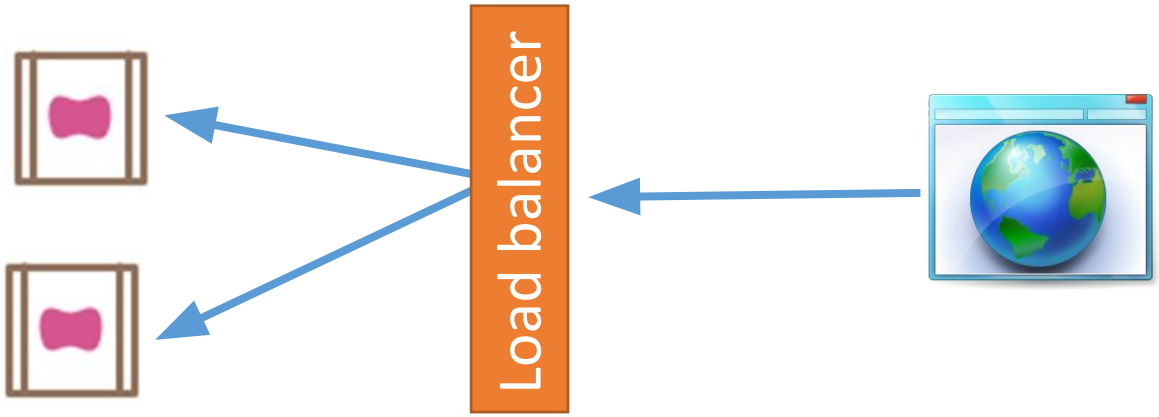
Как построить



Как построить



Масштабируем



Conclusion. Pros

- Low coupling
- Independent deploy and scalability
- Independent development and releases
- Quick development
- Quick onboarding
- Clear contracts between services
- Independent fails

Conclusion. Cons

- Reducing agility
- Increasing of changes complexity
- Difficult debugging and logging, tracing
- A lot of infrastructure
- Change processes in whole company
- Networking
- Distribution

References

- Micro-services архитектуры - избавляемся от монолитного кода
  [http://dotnetconf.ru/materialy/microservicearchitecture]
- Преимущества и недостатки микросервисной архитектуры в HeadHunter [https://www.youtube.com/watch?v=7WT_Rl6m2DU]
- События, шины и интеграция данных в непростом мире микросервисов [https://www.youtube.com/watch?v=l5ug_W9iFUs]

THE END