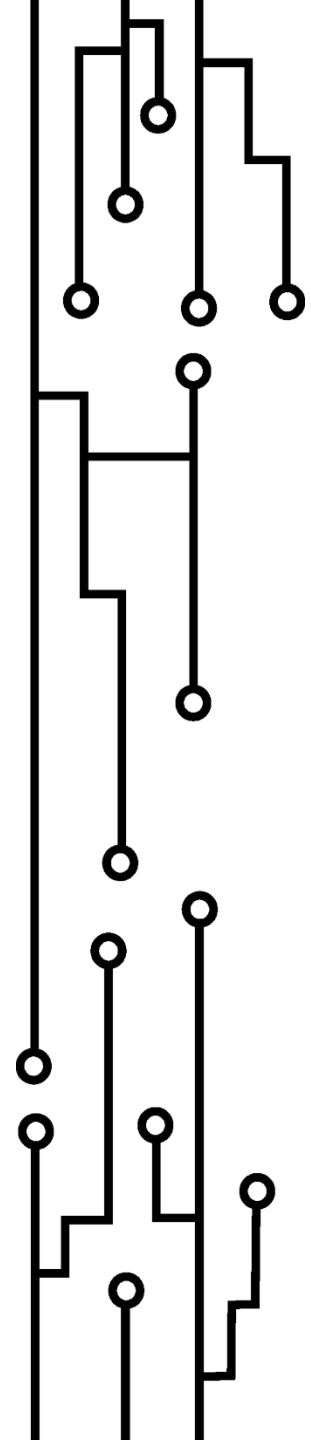


SolarLab>_

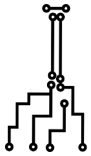
ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА РЕШЕНИЯ: СЛОИ, ПРОЕКТЫ, СВЯЗИ (IoC), .NET CORE

г. Севастополь

2018

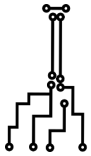
Содержание

- Независимость данных в многослойной архитектуре
- Инверсия управления, IoC-контейнер
- .NET CORE и .NET STANDARD
- Дополнительные материалы: N-уровневая архитектура, выделение слоев



НЕЗАВИСИМОСТЬ ДАННЫХ В МНОГОСЛОЙНОЙ АРХИТЕКТУРЕ

Отделение представления от логики и данных



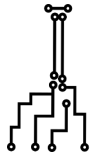
ПОЧЕМУ АРХИТЕКТУРА ТАК ВАЖНА?

Как и любая другая сложная структура, программное обеспечение(ПО) должно строиться на прочном фундаменте. Неправильное определение ключевых сценариев, неправильное проектирование общих вопросов или неспособность выявить долгосрочные последствия основных решений могут поставить под угрозу всё приложение. Современные инструменты и платформы упрощают задачу по созданию приложений, но не устраняют необходимости в тщательном их проектировании на основании конкретных сценариев и требований. Неправильно выработанная архитектура обусловливает нестабильность ПО, невозможность поддерживать существующие или будущие бизнес-требования, сложности при развертывании или управлении в среде производственной эксплуатации.

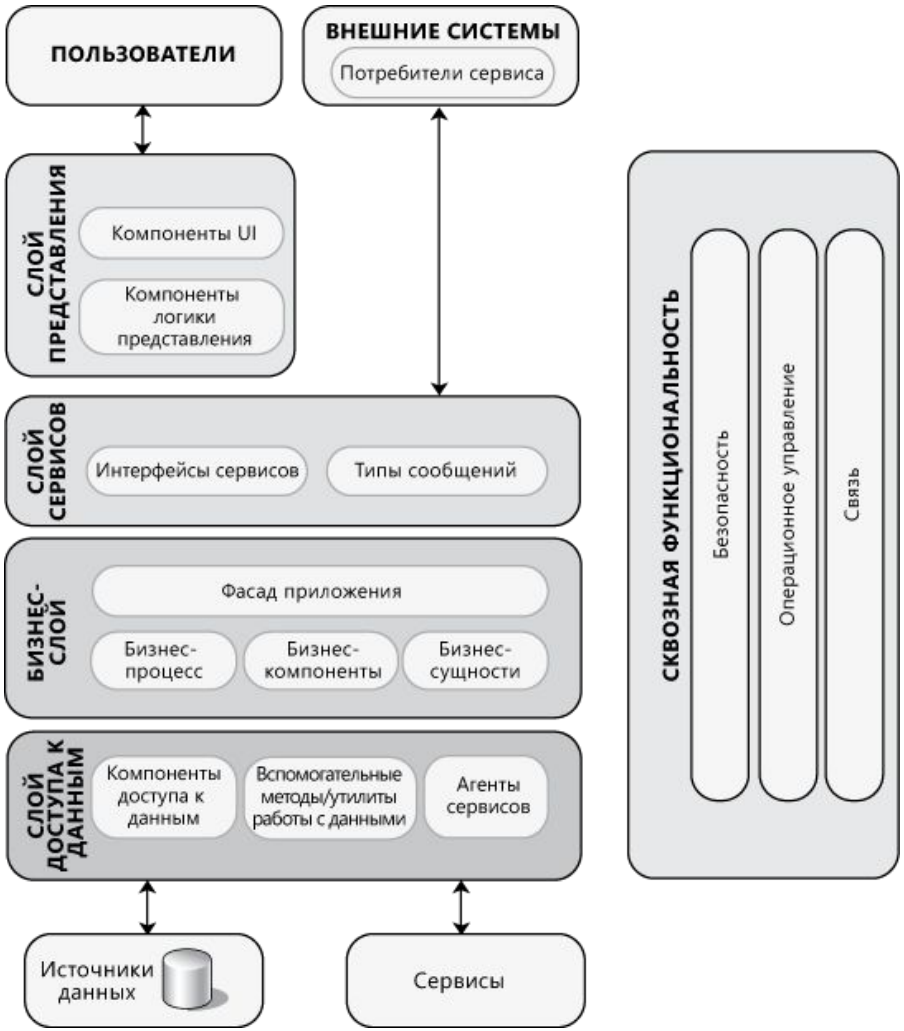
Проектирование систем должно осуществляться с учетом потребностей пользователя, системы (ИТ-инфраструктуры) и бизнес-целей. Для каждой из этих составляющих определяются ключевые сценарии и выделяются важные параметры качества (например, надежность или масштабируемость).

ЦЕЛИ АРХИТЕКТУРЫ

- Раскрывать структуру системы, но скрывать детали реализации
- Стремиться реализовывать все варианты использования и сценарии
- По возможности отвечать всем требованиям различных заинтересованных сторон
- Выполнять требования, как по функциональности, так и по качеству
- Другие цели



ТИПОВАЯ АРХИТЕКТУРА ПРИЛОЖЕНИЯ



Основные принципы проектирования архитектуры ПО

1. **Разделение функций.**
2. **S.O.L.I.D.**
3. **Принцип минимального знания**
4. **Не повторяйтесь (Don’t repeat yourself, DRY).**
5. **KISS, YAGNI, GRASP и многие многие другие**

Основные принципы проектирования архитектуры ПО

1. **Разделение функций.**

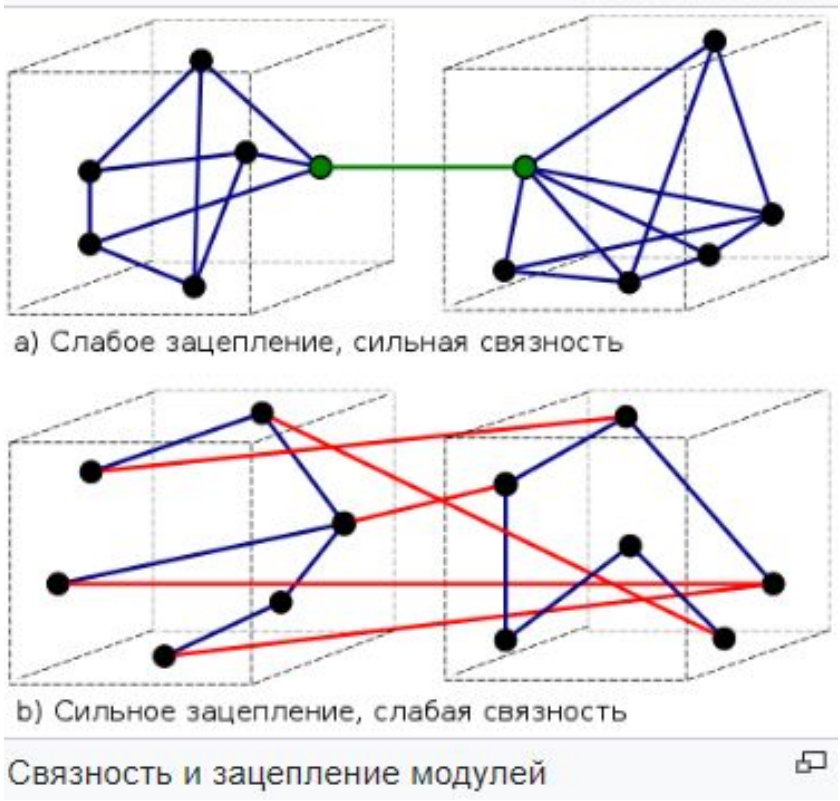
Разделите приложение на отдельные компоненты с, по возможности, минимальным перекрытием функциональности.

Важным фактором является предельное уменьшение количества точек соприкосновения между модулями или логическими структурными единицами, что обеспечит высокую связность (high cohesion) и слабое зацепление (low coupling).

Высокая связанность означает, что обязанности данного элемента тесно связаны и сфокусированы.

Неверное разграничение функциональности может привести к низкой связанности и сложностям взаимодействия, даже несмотря на слабое перекрытие функциональности отдельных компонентов.

Методы уменьшения зацепления



Связность и зацепление модулей

Существуют различные методы уменьшения зацепления (decoupling). Как правило, они описаны в виде шаблонов проектирования. Одними из ключевых методов является «инверсия управления», и, в частности, «внедрение зависимостей».

Снизить зацепление также помогает использование многослойной архитектуры приложений, например Model-View-Controller, Model-View-Presenter, Model-View-ViewModel и т. п.

2 S.O.L.I.D.

- S - Принцип единственной ответственности (The Single Responsibility Principle)

Каждый отдельно взятый компонент или модуль должен отвечать только за одно конкретное свойство/функцию или совокупность связанных функций.

- O - Принцип открытости/закрытости (The Open Closed Principle)

Сущности должны быть открыты для расширения и закрыты для модификации

2 S.O.L.I.D.

- L - Принцип подстановки Барбары Лисков (The Liskov Substitution Principle)

Классика:
«Пусть q(x) является свойством верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T.»

Попытка №1:
Необходимо так строить иерархию наследования объектов, чтобы в любом месте программы на место базового класса можно было подставить потомка и это не нарушило бы корректность программы.

Попытка №2:
Наследующий класс должен дополнять, а не замещать поведение базового класса.

SolarLab>_

2 S.O.L.I.D.

- Нарушение принципа Лисков

```
---- File:./rsl/square.cpp

class Square : Rectangle {
  public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};
void Square::SetWidth(double w){
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}
void Square::SetHeight(double h){
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}

 ---- End Of File:./rsl/square.cpp
```

```
---- File:./rsl/violation.cpp

void LSPV(Rectangle& r){
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}

---- End Of File:./rsl/violation.cpp
```


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

2 S.O.L.I.D.

- I - Принцип разделения интерфейса (The Interface Segregation Principle)

Несколько специфичных интерфейсов лучше, чем один общий интерфейс

- D - Принцип инверсии зависимостей (The Dependency Inversion Principle)

Модули должно зависеть от абстракций, а не реализаций

Основные принципы проектирования архитектуры ПО. Продолжение

3. Принцип минимального знания (также известный как Закон Деметера (Law of Demeter, LoD)). Компоненту или объекту не должны быть известны внутренние детали других компонентов или объектов.

4. Не повторяйтесь (Don’t repeat yourself, DRY). Намерение должно быть обозначено только один раз. В применении к проектированию приложения это означает, что определенная функциональность должна быть реализована только в одном компоненте и не должна дублироваться ни в одном другом компоненте.

Слои приложения

Разделяйте функциональные области. Разделите приложение на отдельные функции с, по возможности, минимальным перекрытием функциональности. Основное преимущество такого подхода – независимая оптимизация функциональных возможностей. Кроме того, сбой одной из функций не приведет к сбою остальных, поскольку они могут выполняться независимо друг от друга. Такой подход также упрощает понимание и проектирование приложения и облегчает управление сложными взаимосвязанными системами.

Явно определяйте связи между слоями. Решение, в котором каждый слой приложения может взаимодействовать или имеет зависимости со всеми остальными слоями, является сложным для понимания и управления. Принимайте явные решения о зависимостях между слоями и о потоках данных между ними.

Реализуйте слабое связывание слоев с помощью абстракции.

Это можно реализовать, определяя интерфейсные компоненты с хорошо известными входными и выходными характеристиками, такие как фасад, которые преобразуют запросы в формат, понятный компонентам слоя. Кроме того, также можно определять общий интерфейс или совместно используемую абстракцию (противоположность зависимости), которые должны быть реализованы компонентами интерфейса, используя интерфейсы или абстрактные базовые классы.

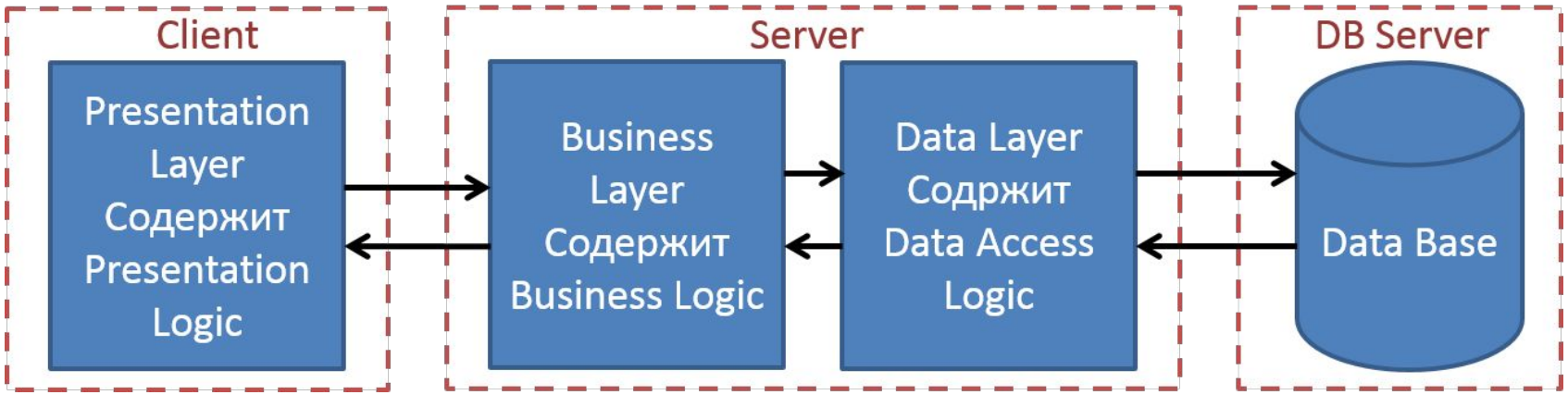
Не смешивайте разные типы компонентов на одном логическом уровне.

Начинайте с идентификации функциональных областей и затем группируйте компоненты, ассоциированные с каждой из этих областей в логические уровни. Например, слой UI не должен включать компоненты выполнения бизнес-процессов, в него должны входить только компоненты, используемые для обработки пользовательского ввода и запросов.

Придерживайтесь единого формата данных в рамках слоя или компонента.

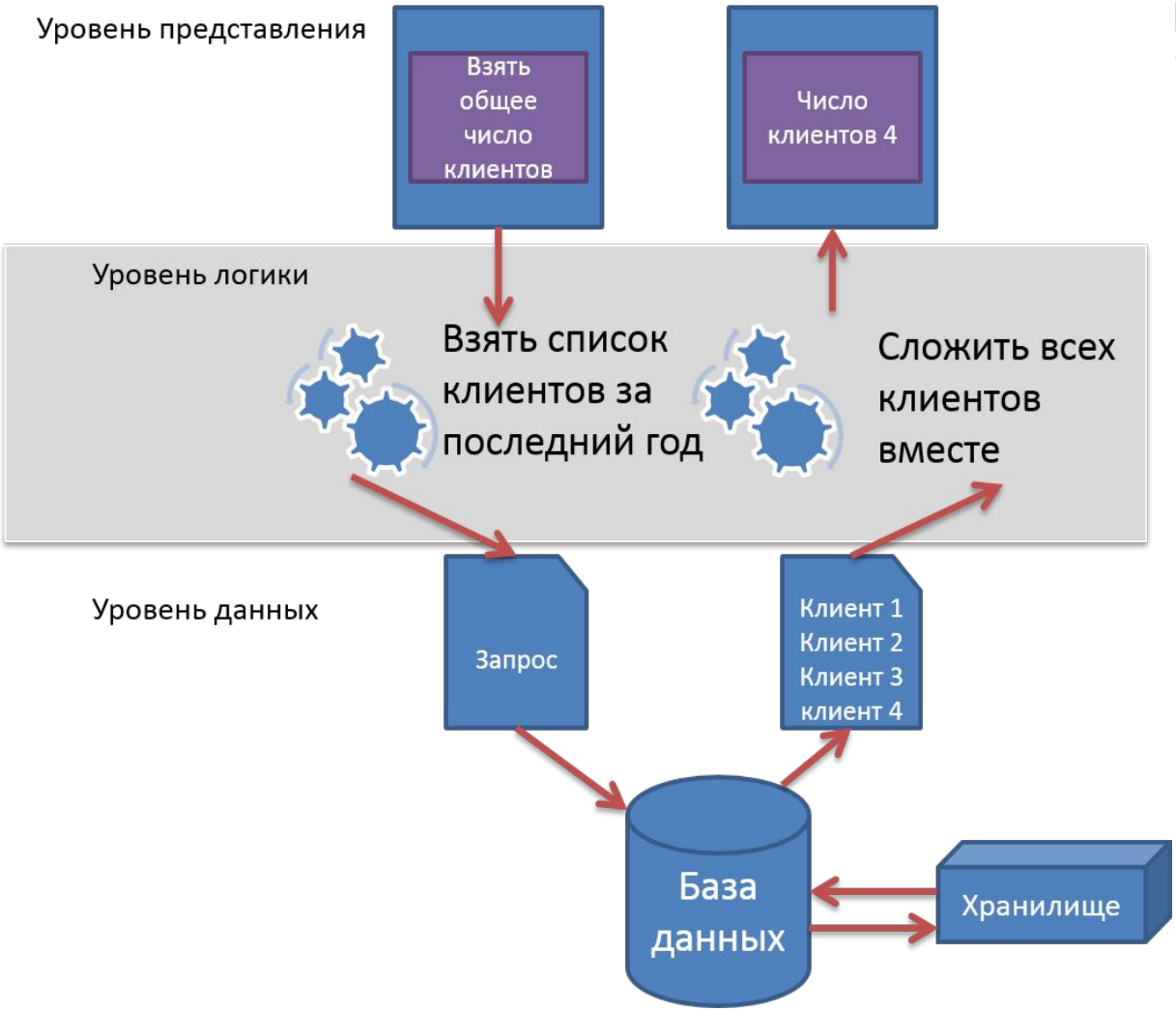
Смешение форматов данных усложнит реализацию, расширение и обслуживание приложения. Любое преобразование одного формата данных в другой требует реализации кода преобразования и влечет за собой издержки на обработку.

3-уровневая архитектура



- Каждый слой может быть потенциально запущен на отдельной машине
- Представление логика и данные разделены

Многослойное приложение



Принципы архитектуры:

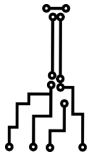
- Клиент-серверная архитектура
- Каждый слой (данные, представление и логика) не зависит от остальных и не зависит от реализации
- Несоединённые слои вообще никогда не взаимодействуют
- Изменение платформы влияет только на тот уровень который на ней находится

3-уровневая архитектура

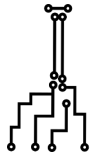
- Уровень представления
  - Статический или динамически сгенерированный контент отображаемый через браузер (front-end)
- Уровень логики
  - Уровень подготовки данных для динамически генерируемого контента, уровень сервера приложений (application server). Middleware платформы: ASP.NET, JavaEE, PHP, ColdFusion
- Уровень данных
  - База данных включающая в себя данные и систему управления над ними или же готовая RDBMS система, предоставляющая доступ к данным и методы управления (back-end)

Преимущества

- Независимость уровней
  - Лёгкость в поддержке
  - Отдельные компоненты можно использовать в других задачах
  - Задача разработки хорошо делится и поэтому может быть быстрее решена (уровни можно разрабатывать параллельно)
    - Web дизайнер делает уровень представления
    - Инженер (Software Engineer) делает логику
    - Администратор БД делает модель данных



Инверсия зависимостей, Внедрение зависимостей (DI), IoC(DI)-контейнер



Что такое инверсия зависимостей?

- **Принцип инверсии зависимостей** (англ. *dependency inversion principle*, *DIP*) — важный принцип объкетно-ориентированного программирования, используемый для уменьшения зацепления в компьютерных программах. Как мы помним, входит в пятёрку принципов SOLID.

Формулировка:

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Что такое внедрение зависимостей?

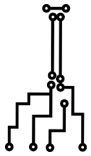
Внедрение зависимостей (Dependency Injection, DI) — методика для создания слабосвязанных приложений. Она предоставляет возможности для упрощения кода, извлечения и обработки зависимостей между объектами и автоматического создания экземпляров зависимого объекта. Внедрение зависимостей описывает процесс разработки приложений — вместо указания конкретных зависимостей в приложении во время разработки и создания необходимых объектов в коде во время выполнения приложение решает, какие объекты ему требуются, а потом создает и внедряет их в приложение.

Преимущества IoC и DI

- **Ослабление соединения между классами**. Зависимости четко определены в каждом классе. Сведения о сопоставления между интерфейсами и реализующими их классами хранятся в IoC-контейнере, который и используется при внедрении зависимостей. При необходимости зависимости можно обновить лишь переконфигурировав IoC-контейнер.

- **Создание кода, который лучше поддается проверке**. Можно легко узнать из типов конструкторов, свойств или методов пользовательских классов, какие объекты они используют и какие у них существуют зависимости.

- **Упрощение тестирования**. Если воспользоваться преимуществом внедрения зависимостей, можно легко подставить moq или stub, той или иной зависимости, что делает компоненты легче поддающимися тестированию, а ПО надёжнее.



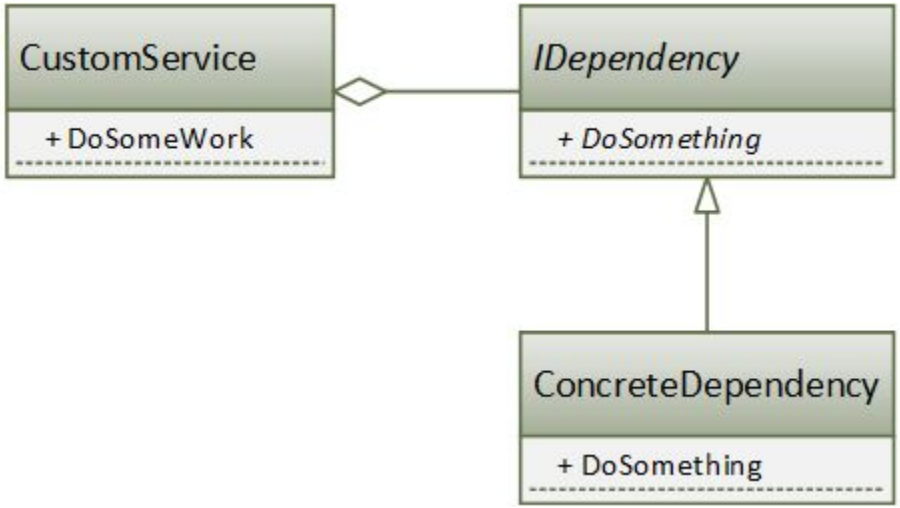
Преимущества

Главным преимуществом внедрения зависимостей через конструктор является четкое разделение ответственностей и явность (explicitness) интерфейса. Основная проблема использования глобальных объектов и синглтонов заключается в том, что читая «заголовок класса» (его публичный интерфейс) невозможно определить сложность его поведения, и для этого нужно проанализировать *реализацию* этого класса со всеми его методами. В случае, когда все внешние зависимости передаются через конструктор, сложность класса становится более очевидной.

Потенциальной проблемой использования конструктора для передачи зависимостей может быть чрезмерное увеличение параметров конструктора. Но, на самом деле, это не является недостатком этого паттерна, а скорее является его преимуществом.

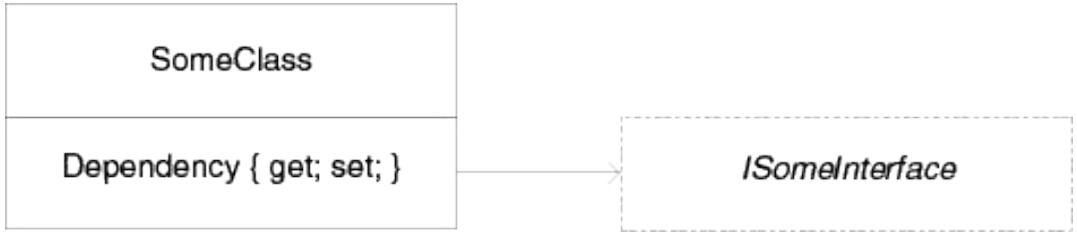
Внедрение конструктора (Constructor Injection)

Суть паттерна сводится к тому, что все зависимости, требуемые некоторому классу передаются ему в качестве параметров конструктора, представленных в виде интерфейсов или абстрактных классов.



Внедрение свойства (Property Injection)

Еще одним достаточно популярным паттерном внедрения зависимостей является Property Injection, который заключается в передаче нужных зависимостей через “setter” свойства. Все современные DI-контейнеры в той или иной мере поддерживают этот паттерн, что делает его использование достаточно простым.

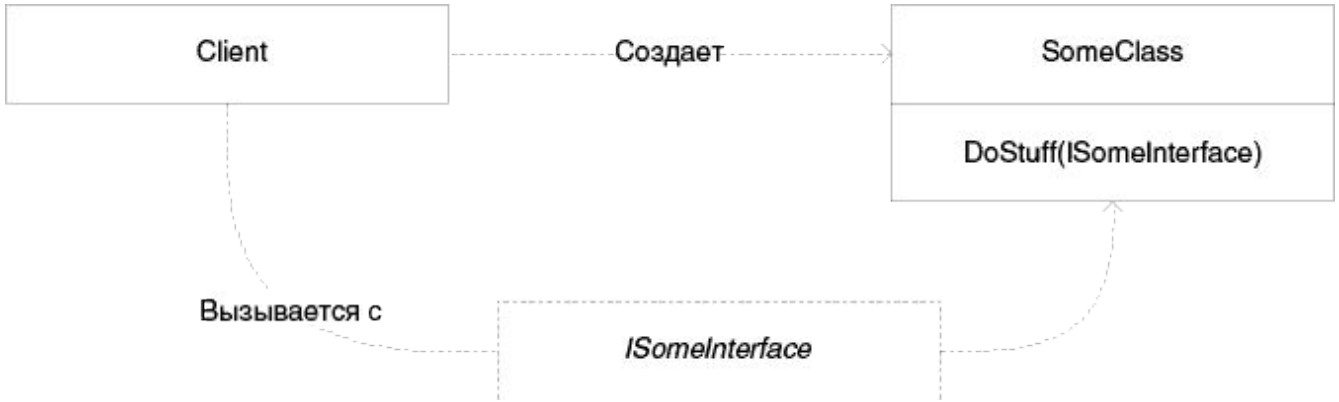


SomeClass имеет опциональную зависимость для ISomeInterface; вместо того, чтобы требовать от вызывающих элементов предоставить экземпляр, он дает вызывающим элементам возможность определить его через свойство.

Внедрение вызова метода (Method Injection)

По сути, этот паттерн аналогичен рассмотренному ранее паттерну Property Injection с закрытом геттером, со всеми преимуществами и недостатками. Он применяется в языках без встроенной поддержки свойств, а также может успешно применяться в языке C#, если вам это больше нравится.

Вторым типом паттерна Method Injection является передача зависимости в метод, который будет использовать ее для решения текущей задачи, а не сохраняться во внутреннем поле для последующего использования.



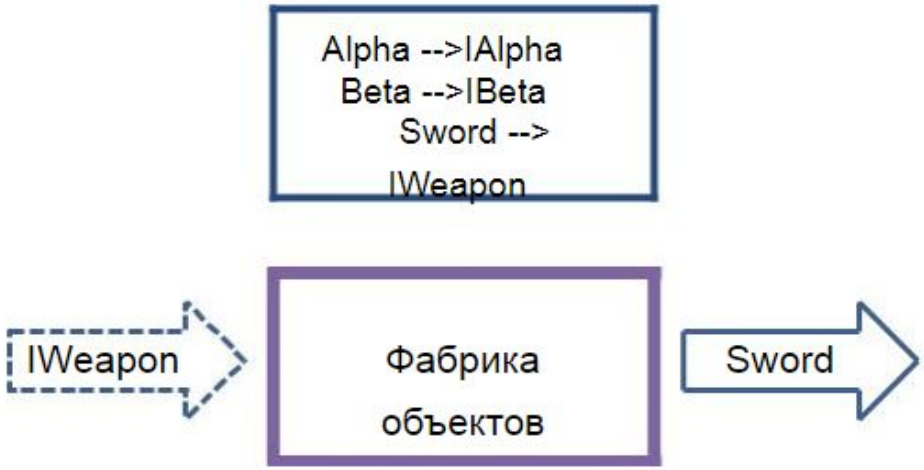
Клиент создает экземпляр SomeClass, но сначала внедряет экземпляр зависимости ISomeInterface с каждым вызовом метода.

IoC-контейнер

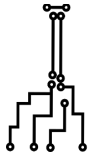
IoC контейнер - это служба для управления созданием объектов.

Составные части контейнера:

1. Регистратор реализаций
2. Фабрика объектов



.NET CORE и .NET STANDARD



.NET Core

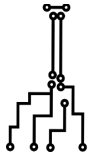
- это самая новая реализация .NET. Это проект Open Source с версиями для нескольких ОС.

.NET Core позволяет создавать кроссплатформенные приложения, сервисы и облачные службы ASP.NET Core.

.NET Standard

Это набор базовых API (другое их название – BCL, библиотека базовых классов), которые должны поддерживаться во всех реализациях .NET.
.NET Standard позволяет создавать библиотеки, подходящие для любых приложений .NET, вне зависимости от реализации .NET или операционной системы, в которой они выполняются.
.NET Standard представляет собой спецификацию API, которые должны содержаться во всех реализациях .NET. Он делает семейство технологий .NET более организованным и позволяет разработчикам создавать библиотеки, которые можно использовать в любой реализации .NET.

Подробнее можно почитать на хабре:
https://habrahabr.ru/company/microsoft/blog/340128/

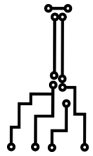


ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ

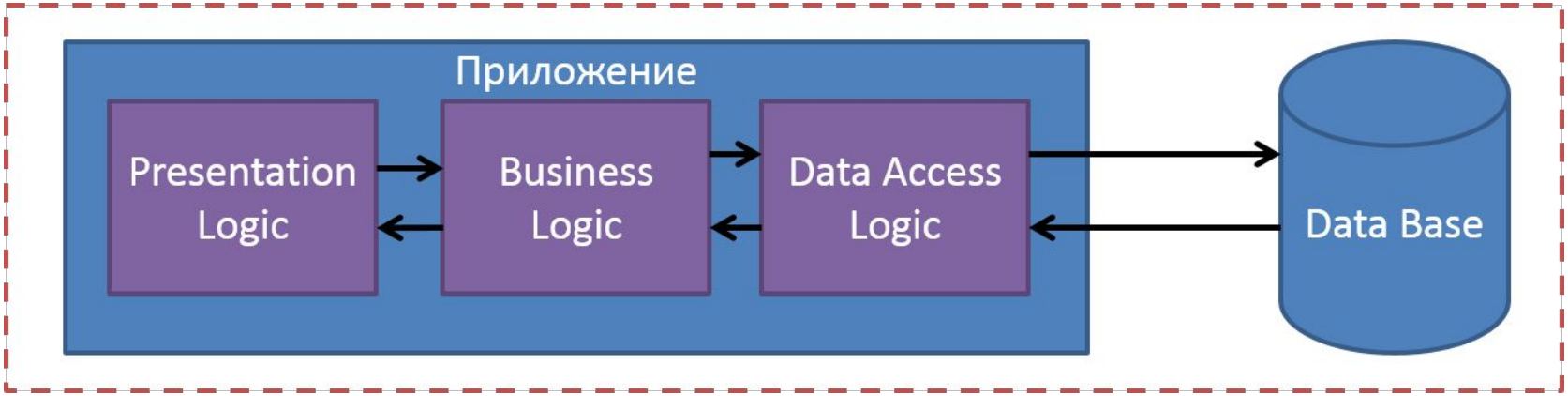
N-УРОВНЕВАЯ АРХИТЕКТУРА

НЕОБХОДИМОСТЬ СЛОЁВ

- N-уровневая архитектура имеет следующие компоненты:
  - Уровень представления
  - Уровень бизнес логики
  - Данные
- N-уровневая архитектура пытается разделить компоненты на разные уровни (слои):
  - Физическое разделение
  - Логическое разделение

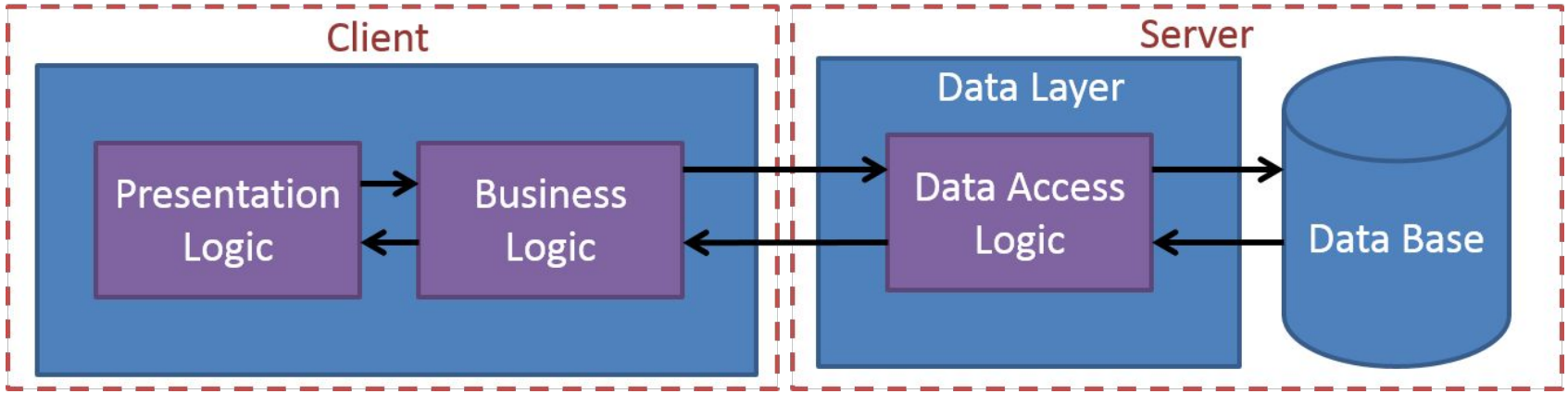


1-уровневая архитектура



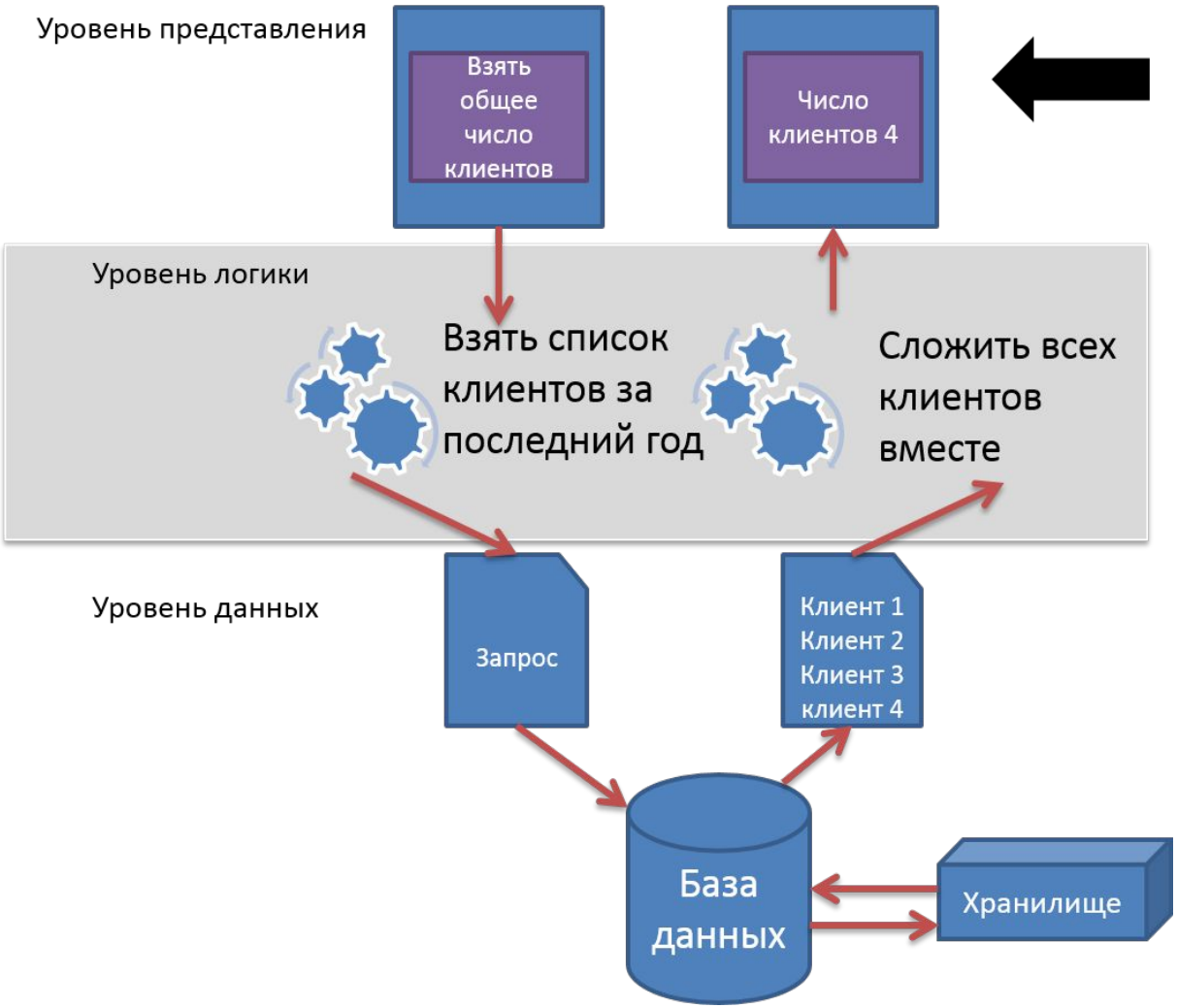
- Все 3 слоя на одной машине
  - Весь код и все процессы происходят на одной машине
- Представление, логика, уровень данных сильно связаны
  - **Расширяемость (Scalability):** на одном процессоре можно запустить только ограниченное число процессов
  - **Переносимость (Portability):** для перемещения на новую машину придётся переписать весь код
  - **Поддержка (Maintenance):** при изменении одного слоя придётся изменять остальные

2-уровневая архитектура



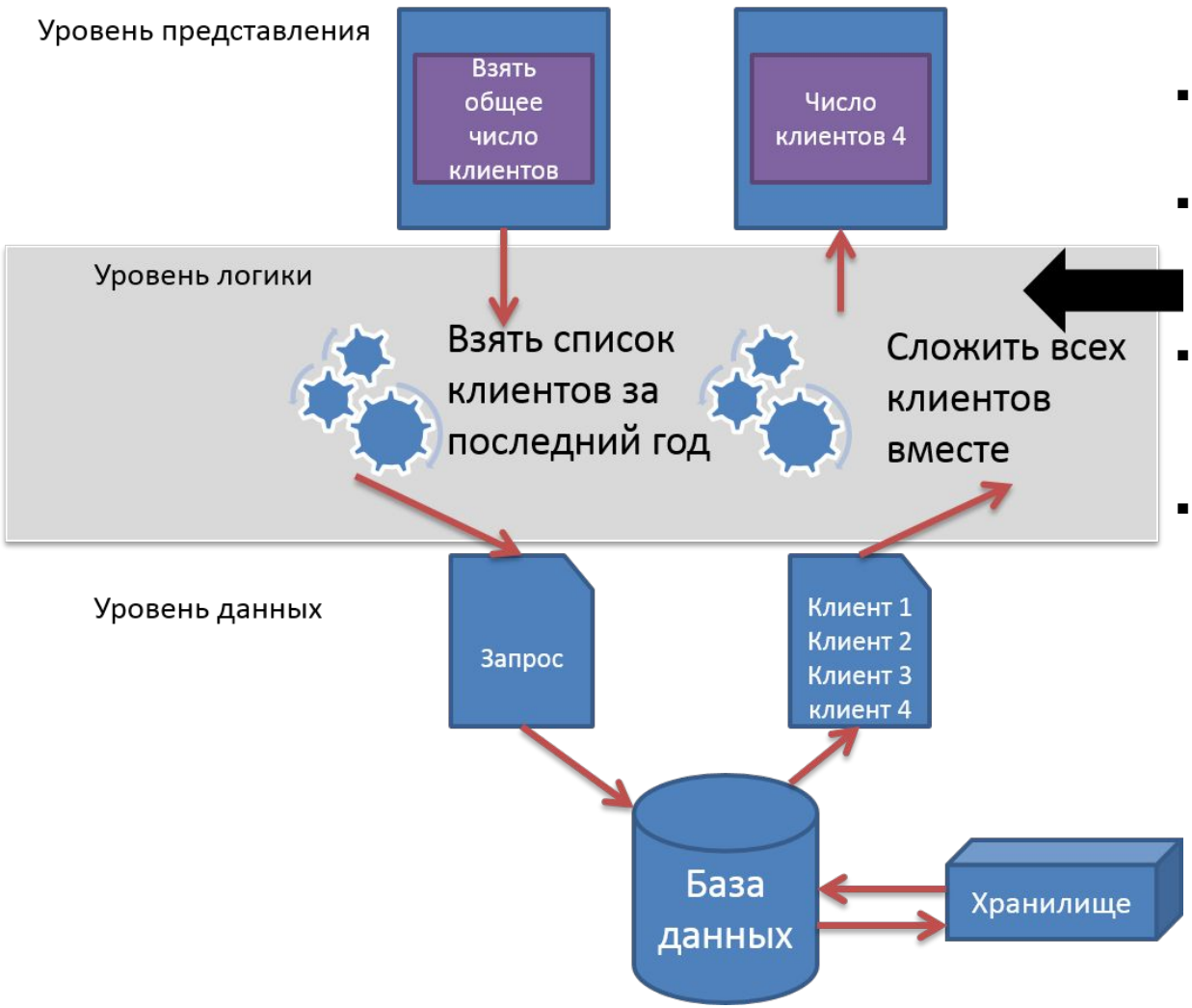
- База данных работает на сервере
  - Отделение данных от клиента
  - Клиент может переключить базу данных
- Отделение представления от данных
  - Высокая нагрузка на сервер
  - Потенциальные проблемы с задержкой в сети
  - Уровень представления и логики остаются сильно связанны

Уровень представления (Presentation Layer)



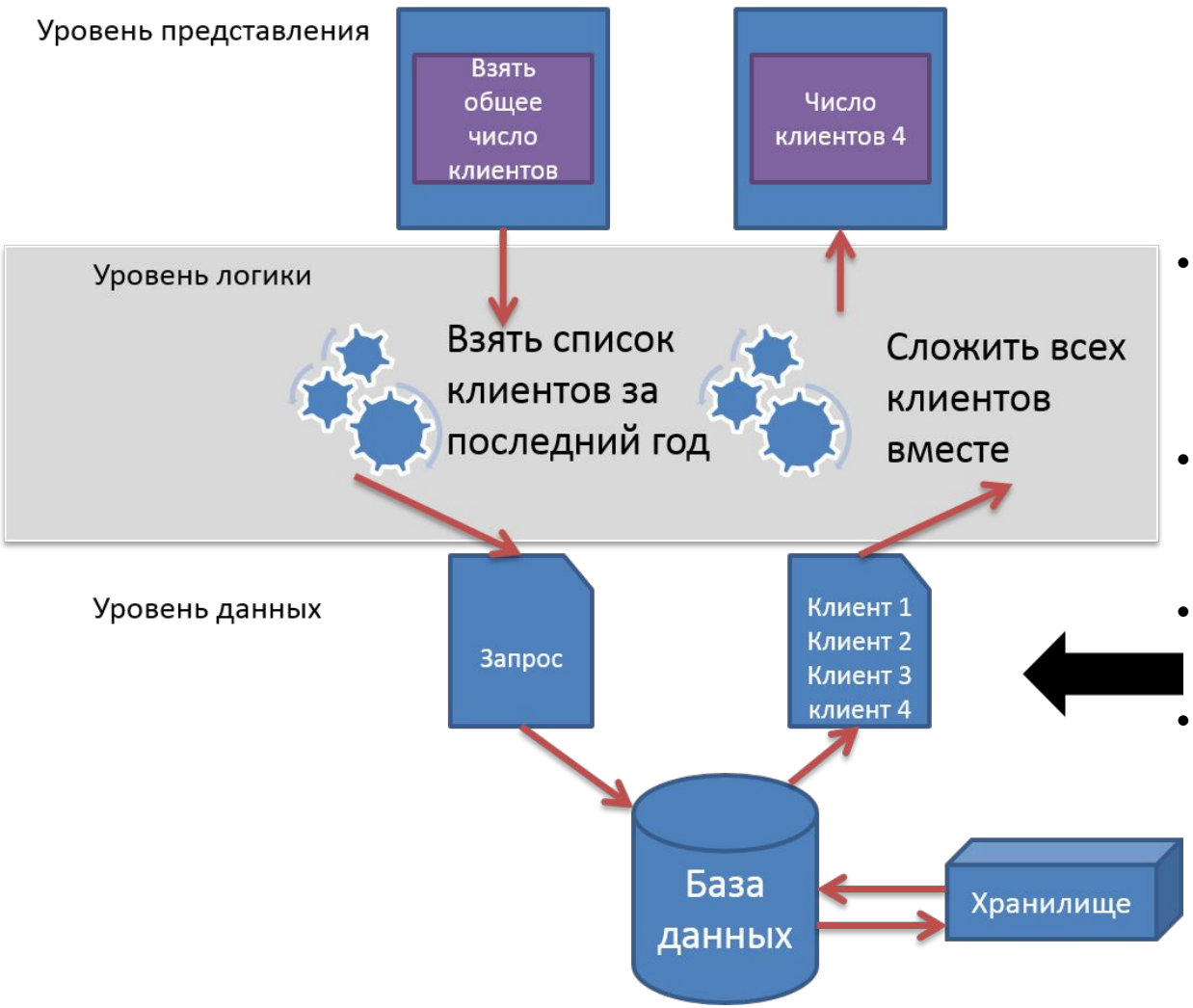
- Предоставляет графический интерфейс
- Обрабатывает пользовательские события
- Иногда называют GUI или client view of front-end

Бизнес-логика (Business Logic Layer)

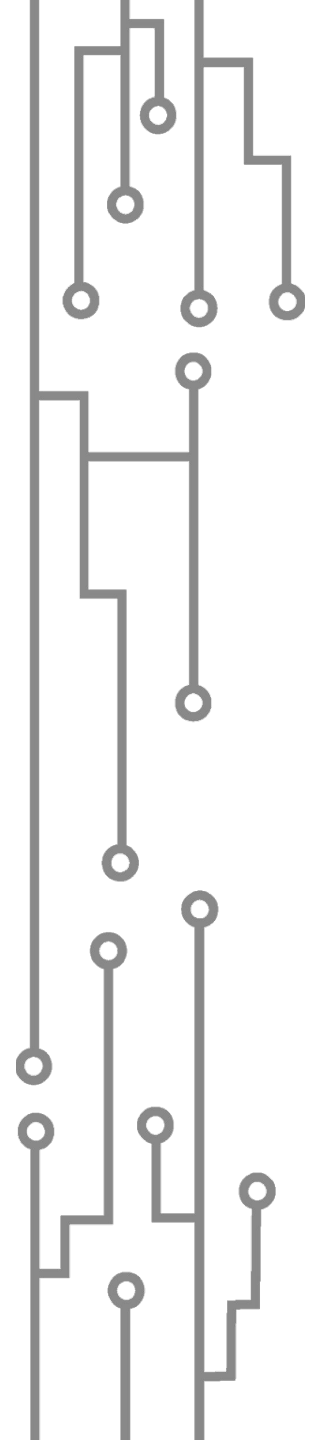


- Набор правил для работы с данными
- Может обрабатывать запросы нескольких пользователей
- Иногда называют middleware или back-end
- Не должен содержать пользовательских форм или непосредственно обращаться к данным

Уровень доступа к данным (Data Access Layer)



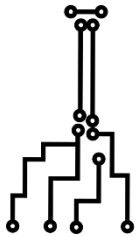
- Физическое хранилище для данных (persistence)
- Управляет доступом к БД или файловой системе
- Иногда называется back-end
- Не должен содержать пользовательских форм или бизнес логики



Info :

1.https://docs.microsoft.com/en-us/aspnet/core/index?view=aspnetcore-2.1

2.https://simpleinjector.readthedocs.io/en/latest/



SolarLab>_

Спасибо за внимание!