



# ASP.NET Core MVC

(Валидация данных,  
Инверсия контроля и Внедрение зависимостей)

Андрей Голяков

# Валидация данных в MVC

---

Задачи, которые надо решить при валидации данных:

1. Определить правила валидации
2. Проверить данные
3. Информировать пользователя о проблемах



# Столица Таиланда - Бангкок?..

---

กรุงเทพมหานคร อมรรัตนโกสินทร์ มหินทรายุธยา มหาดิลกภพ นพ  
รัตนราชธานีบูรีรมย์ อุดมราชนิเวศน์มหาสถาน อมรพิมานอวตาร  
สถิต สักกะทัตติยวิษณุกรรมประสิทธิ์

Крун Тхеп Маханакхон Амон Раттанакосин Махинтараюттхая Махадилок Пхоп Ноппарат  
Ратчатхани Буриром Удомратчанивет Махасатан Амон Пиман Аватан Сатит Саккатхаттия  
Витсанукам Прасит.

*\* Перевод: город ангелов, великий город, город — вечное сокровище, неприступный город бога Индры, величественная столица мира, одаренная девятью драгоценными камнями, счастливый город, полный изобилия грандиозный королевский дворец, напоминающий божественную обитель, где царствует перевоплощенный бог, город, подаренный Индрой и построенный Вишнукарном.*



# Что именно требуется проверять?

---

Как правило, проверки требуют запросы, которые что-то привносят в серверные данные. Часто проверки удостоверяются запросы следующих типов:

- POST
- PUT
- PATCH

**Проверяем только входные данные!** (выходные данные не проверяются)



# Data Annotations (установка правил валидации)

---

Для валидации можно использовать как сторонние, так и встроенные средства установки правил и проверки данных пользователя.

В ASP.NET Core MVC встроенным средством проверки является Data Annotations.

Атрибуты аннотации данных (**data annotation attributes**) — это специальные атрибуты, которыми можно разметить модель, чтобы обозначить правила её валидации.

Такие атрибуты включают в себя возможность задавать часто используемые правила как “обязательное поле” или “максимальная длина строки”. Также можно определять и более сложные правила.



# ModelState (проверка правил валидации)

---

Для пункта 2 из списка решаемых задач — непосредственно проверки — используется концепция модели состояния.

Это сложный объект, в котором хранится как словарь состояния модели в привязке к конкретным проверкам, так и коллекция ошибок для каждого свойства объекта модели.

Для быстрого анализа можно воспользоваться свойством `IsValid`:  
Если в модели есть проблемы, `ModelState.IsValid` вернёт `false`.



# StatusCode + Response body (информирование)

Чтобы информировать пользователя о проблемах с данными, используется комбинация соответствующего кода статуса (из набора 4xx) и тело ответа для детального описания, какие проблемы обнаружены и с какими данными.

Например:

```
{  
  ...  
  "Title": ["This is a required field."],  
  "Description": ["The maximum length is 500 characters."]  
  ...  
}
```



# Совместная работа

---

Добавление проверки при создании нового города.

Правила модели CityCreateModel:

- Поле **Name**
  - Обязательное поле
  - Максимальная длина: 100 символов
- Поле **Description**:
  - Максимальная длина: 255 символов
- Поле **NumberOfPointsOfInterest**
  - Число в диапазоне от 0 до 100





# Самостоятельная работа

Доделать оставшиеся проверки, необходимые при создании нового города.  
Правила модели CityCreateModel:

- Поле Name
  - Обязательное поле
  - Максимальная длина: 100 символов
- Поле Description:
  - Максимальная длина: 255 символов
    - \* задать собственное сообщение об ошибке *"Description should be not longer than 255 characters"*
- Поле NumberOfPointsOfInterest
  - Число в диапазоне от 0 до 100
    - \* атрибут Range (разбираемся сами через справку по F1)



# Совместная работа

---

Добавление кастомной проверки, отсутствующей в наборе готовых атрибутов.

Правила модели CityCreateModel:

- Поле Name
  - Обязательное поле
  - Максимальная длина: 100 символов
- Поле Description:
  - Максимальная длина: 255 символов
  - Поле не должно иметь такое же значение, как в поле Name
- Поле NumberOfPointsOfInterest
  - Число в диапазоне от 0 до 100



# Inversion of Control и Dependency Injection

Инверсия управления (**Inversion of Control, IoC**) это определенный набор рекомендаций, позволяющих проектировать и реализовывать приложения используя слабое связывание отдельных компонентов.

Для того чтобы следовать принципам инверсии управления нам необходимо:

- Реализовывать компоненты, отвечающие за одну конкретную задачу
- Компоненты должны быть максимально независимыми друг от друга
- Компоненты не должны зависеть от конкретной реализации друг друга

Одним из видов конкретной реализации данных рекомендаций является механизм внедрения зависимостей (**Dependency Injection, DI**). Он определяет две основные рекомендации:

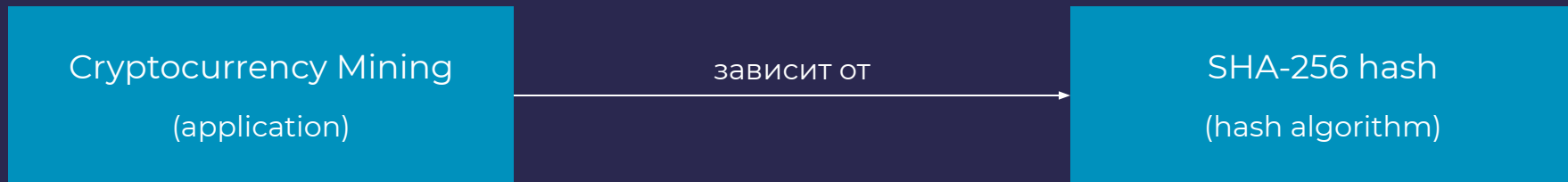
- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

То есть, если у нас будут существовать два связанных класса, то нам необходимо реализовывать связь между ними не напрямую, а через интерфейс. Это позволит нам при необходимости динамически менять реализацию зависимых классов.



# Inversion of Control и Dependency Injection

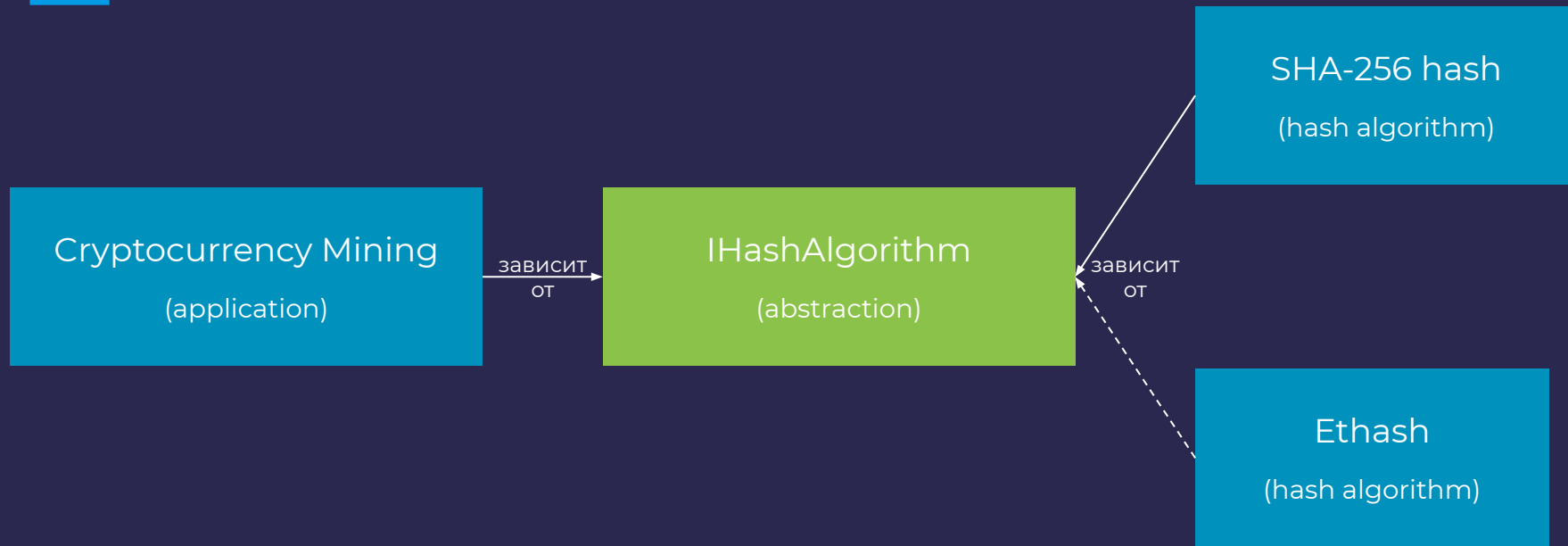
---



нет внедрения зависимостей  
принципы инверсии контроля не соблюдаются



# Inversion of Control и Dependency Injection



внедрение зависимостей присутствует !  
принципы инверсии контроля соблюдаются :)



# Совместная работа

---

1. Использование встроенного сервиса логирования
2. Использование NLog, логирование в файл
  - Делает по рекомендациям авторов NLog:  
<https://github.com/NLog/NLog.Web/wiki/Getting-started-with-ASP.NET-Core-2>



# Регистрация собственных сервисов

---

Собственные сервисы регистрируются в методе `ConfigureServices`.

Можно воспользоваться одним из трёх методов в зависимости от желаемого жизненного цикла сервиса:

- **AddTransient** — объект пересоздаётся при каждом обращении к сервису создается новый объект сервиса. В течение одного запроса может быть несколько обращений к сервису, соответственно при каждом обращении будет создаваться новый объект. Подобная модель жизненного цикла наиболее подходит для легковесных сервисов, которые не хранят данных о состоянии.
- **AddScoped** — объект пересоздаётся для каждого запроса создается свой объект сервиса. То есть если в течение одного запроса есть несколько обращений к одному сервису, то при всех этих обращениях будет использоваться один и тот же объект сервиса.
- **AddSingleton** — объект сервиса создается при первом обращении к нему, все последующие запросы используют один и тот же ранее созданный объект сервиса.



# Совместная работа

---

Оформляем CitiesDataStore как сервис.





# Домашняя работа

---

Добавить валидацию на все оставшиеся методы.

Вынести декларацию и имплементацию Data Access Layer (DAL) за пределы приложения.



# Спасибо за внимание.

