Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

Отчёт

по лабораторной работе № 4
по дисциплине «Программирование на Python»

Тема: «Автодокументация в Python»

Выполнил: Бородин Г.
студент группы ИУ8-13М

Проверил: Зотов М.

г. Москва,2025 г.

1. ЦЕЛЬ РАБОТЫ

Реализовать систему автодокументации для своих модели данных с использованием tox+sphinx+GithubAction+GithubPages

2. ПОСТАНОВКА ЗАДАЧИ

- Описаны ваши классы, методы, атрибуты в одном из популярных форматов(Google, Numpy или Sphinx).
- Добавлен GithubAction, который по пути вашей лабораторной работы, стартует сборку вашей документации.
- Добавлен сопроводительный файл *.rst формата, где вы описываете вашу работу с tox, pytest, pydantic, sphinx
- Добавлена возможность стартовать документацию через tox

3. ХОД РАБОТЫ

ИСПОЛЬЗУЕТСЯ В ДАННОЙ РАБОТЕ: Python; библиотека Pydantic (модели предметной области и аннотации типов); Sphinx для генерации документации; утилиты sphinx-apidoc и расширение autodoc для извлечения docstring из кода; формат reStructuredText (.rst) для текстовых страниц документации; tox для запуска сборки документации в изолированном окружении; GitHub Actions для автоматической сборки при push; GitHub Pages для публикации HTML-документации. Дополнительно: тема sphinx_rtd_theme и расширение sphinx-autodoc-typehints для отображения аннотаций типов.

МОЙ ХОД РАБОТЫ:

ШАГ 1. Подготовка исходного кода к автодокументации.

В исходных моделях данных (UserSpec, ProfileSpec, ItemSpec, ServiceSpec, OrderLineSpec, OrderSpec, OrdersSpec) оформлены docstring в стиле Sphinx: добавлены описания классов, атрибутов, ограничений валидации и примеры использования. Это необходимо, чтобы Sphinx мог автоматически извлекать и формировать документацию из кода.

ШАГ 2. Настройка Sphinx-проекта.

Создан проект документации Sphinx (конфигурация conf.py и стартовые страницы в формате .rst). Включены расширения sphinx.ext.autodoc (автодокументация из кода) и sphinx.ext.napoleon (корректная обработка секций типа “Args/Attributes/Returns”), а также подключена тема оформления. Подготовлен сопроводительный .rst файл, где описан порядок работы с tox/pytest/pydantic/sphinx и структура проекта.

ШАГ 3. Автоматизация сборки через tox.

В tox.ini добавлено окружение для сборки документации (например, docs_build): установка зависимостей Sphinx и запуск команд sphinx-apidoc

(генерация .rst из пакетов проекта) и sphinx-build (сборка HTML). Таким образом, документация собирается одной командой через tox.

ШАГ 4. CI/CD: GitHub Actions и публикация на GitHub Pages. Добавлен workflow GitHub Actions, который при обновлении репозитория запускает сборку документации и публикует получившийся HTML на GitHub Pages. В результате документация обновляется автоматически при изменении кода/описаний.

4. ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы были изучены принципы автодокументации в Python и инструменты Sphinx (autodoc, sphinx-apidoc) для генерации документации по docstring и аннотациям типов. Оформлены docstring для собственных моделей данных на Pydantic, создан Sphinx-проект и сопроводительные .rst страницы. Реализована сборка документации через tox, а также настроен GitHub Actions для автоматической сборки и публикации HTML-документации на GitHub Pages. Полученная документация позволяет просматривать структуру проекта и описание моделей без ручного ведения “бумажного” описания.

5. ПРИЛОЖЕНИЕ

Реализованный код располагается по следующей ссылке: https://github.com/zezOtik/bmstu--iu8--python/tree/feature/borodin_2/documentation

Листинг 1 – Исходный код моделей данных интернет-магазина на Pydantic с docstring, подготовленный для генерации автодокументации Sphinx (autodoc).

```
import logging
import yaml
from pydantic import BaseModel, Field, ConfigDict, model_validator
from pydantic import EmailStr, HttpUrl
from typing import Literal, Union, List, Optional


# Настройка логирования
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')


# Регулярное выражение для валидации кириллических строк (разрешены пробелы и дефисы)
RUS_RE = r'^[а-яА-ЯёЁ\s\-]+$'



class UserSpec(BaseModel):
    """Базовая модель пользователя маркетплейса.

    Атрибуты:
        user_id (int): Уникальный идентификатор пользователя. Должен быть > 0.
```

```
    username (str): Имя пользователя. Только кириллица, пробелы и
дефисы.
    surname (str): Фамилия пользователя. Только кириллица, пробелы и
дефисы.
    second_name (Optional[str]): Отчество пользователя. Может
отсутствовать.
                                 Если задано — только кириллица, пробелы и дефисы.
    email (EmailStr): Корректный email-адрес.
    status (Literal['active', 'non-active']): Статус пользователя.

  Конфигурация:
    extra="forbid": Запрещает передачу дополнительных полей.

  Пример:
    >>> user = UserSpec(
    ...     user_id=101,
    ...     username="Иван",
    ...     surname="Иванов",
    ...     second_name="Иванович",
    ...     email="ivanov@example.com",
    ...     status="active"
    ... )
  """
  user_id: int = Field(..., gt=0)
  username: str = Field(..., pattern=RUS_RE)
  surname: str = Field(..., pattern=RUS_RE)
  second_name: Optional[str] = Field(None, pattern=RUS_RE)
  email: EmailStr
  status: Literal['active', 'non-active']
```

```
    model_config = ConfigDict(extra="forbid")


class ProfileSpec(UserSpec):
    """Расширенная модель профиля пользователя.

    Наследует все поля от `UserSpec` и добавляет:

    Атрибуты:
        bio (str): Биография пользователя. Только кириллица, пробелы, дефисы и базовая пунктуация (. , ! ?).
        url (HttpUrl): Ссылка на профиль. Только схемы `http` и `https`.

    Пример:
        >>> profile = ProfileSpec(
        ...     user_id=101,
        ...     username="Иван",
        ...     surname="Иванов",
        ...     second_name="Иванович",
        ...     email="ivanov@example.com",
        ...     status="active",
        ...     bio="Люблю покупать товары",
        ...     url="https://example.com/ivanov"
        ... )
    """
    bio: str = Field(..., pattern=RUS_RE)
    url: HttpUrl
    model_config = ConfigDict(extra="forbid")
```

```
class ItemSpec(BaseModel):
    """Спецификация товара (физического объекта) на маркетплейсе.

    Attributes:
        item_id (int): Уникальный идентификатор товара. Должен быть > 0.
        name (str): Название товара. Только кириллица, пробелы и дефисы.
        decs (str): Описание товара. Только кириллица, пробелы и дефисы.
        price (float): Цена товара в рублях. Должна быть > 0.
    """
    item_id: int = Field(..., gt=0)
    name: str = Field(..., pattern=RUS_RE)
    decs: str = Field(..., pattern=RUS_RE)
    price: float = Field(..., gt=0)
    model_config = ConfigDict(extra="forbid")


class ServiceSpec(BaseModel):
    """Модель услуги на маркетплейсе.

    Атрибуты:
        service_id (int): Уникальный идентификатор услуги (> 0).
        name (str): Название услуги. Только кириллица, пробелы и дефисы.
        desc (str): Описание услуги. Только кириллица, пробелы и дефисы.
        price (float): Стоимость услуги (> 0).

    Пример:
        >>> service = ServiceSpec(
        ...     service_id=2001,
```

```
        ...     name="Доставка",
        ...     desc="Быстрая доставка курьером",
        ...     price=500.0
        ... )
    """
    service_id: int = Field(..., gt=0)
    name: str = Field(..., pattern=RUS_RE)
    desc: str = Field(..., pattern=RUS_RE)
    price: float = Field(..., gt=0)
    model_config = ConfigDict(extra="forbid")


class OrderLineSpec(BaseModel):
    """Позиция в заказе: товар или услуга.

    Атрибуты:
        order_id (int): Идентификатор заказа (> 0).
        order_line_id (int): Идентификатор строки заказа (> 0).
        item_line (Union[ItemSpec, ServiceSpec]): Объект товара или услуги.
        quantity (float): Количество (> 0).
        line_price (float): Итоговая цена позиции (= quantity * item_line.price).

    Валидация:
        Автоматически проверяет, что `line_price == quantity * item_line.price`.

    Пример:
        >>> item = ItemSpec(item_id=1001, name="Ноутбук",
desc="Геймерский", price=59999.99)
        >>> line = OrderLineSpec(
```

```
        ...     order_id=1,
        ...     order_line_id=10,
        ...     item_line=item,
        ...     quantity=1.0,
        ...     line_price=59999.99
        ... )
    """
    order_id: int = Field(..., gt=0)
    order_line_id: int = Field(..., gt=0)
    item_line: Union[ItemSpec, ServiceSpec]
    quantity: float = Field(..., gt=0)
    line_price: float = Field(..., gt=0)

    @model_validator(mode='after')
    def check_line_prices(self):
        """Проверяет корректность итоговой цены позиции."""
        calculated = self.quantity * self.item_line.price
        if self.line_price != calculated:
            raise ValueError(
                f'line_price {self.line_price} != '
                f'quantity * item_line.price {calculated}'
            )
        return self

    model_config = ConfigDict(extra="forbid")



class OrderSpec(BaseModel):
    """Полный заказ пользователя.
```

```
    Атрибуты:
        order_id (int): Уникальный идентификатор заказа (> 0).
        user_info (ProfileSpec): Профиль пользователя.
        items_line (List[OrderLineSpec]): Список позиций в заказе (может быть
пустым).

    Пример:
        >>> profile = ProfileSpec(...)
        >>> line = OrderLineSpec(...)
        >>> order = OrderSpec(order_id=1, user_info=profile, items_line=[line])
    """
    order_id: int = Field(..., gt=0)
    user_info: ProfileSpec
    items_line: List[OrderLineSpec]
    model_config = ConfigDict(extra="forbid")


class OrdersSpec(BaseModel):
    """Контейнер для списка заказов.

    Атрибуты:
        market_place_orders (List[OrderSpec]): Список всех заказов.

    Пример:
        >>> order = OrderSpec(...)
        >>> orders = OrdersSpec(market_place_orders=[order])
    """
    market_place_orders: List[OrderSpec]
```

```
    model_config = ConfigDict(extra="forbid")



def get_data_from_yaml(yaml_path: str) -> OrdersSpec:
    """Загружает и валидирует данные из YAML-файла с использованием Pydantic.

    Эта функция:
    - читает YAML-файл по указанному пути,
    - парсит его в Python-объект,
    - выполняет валидацию через корневую модель `OrdersSpec`,
    - логирует результат при успехе,
    - выбрасывает исключение при ошибках.

    Args:
        yaml_path (str): Путь к YAML-файлу с данными о заказах.

    Returns:
        OrdersSpec: Валидированная структура данных.

    Raises:
        yaml.YAMLError: При ошибках разбора YAML.
        pydantic.ValidationError: При невалидных данных.
        Exception: При других ошибках валидации.
    """
    try:
        with open(yaml_path, 'r', encoding='utf-8') as f:
            data = yaml.safe_load(f)
        orders = OrdersSpec.model_validate(data)
```

```
        logging.info("Данные успешно загружены и валидированы.")
        logging.info(orders.model_dump_json(indent=2, ensure_ascii=False))
        return orders
    except yaml.YAMLError as e:
        logging.error(f"Ошибка при чтении YAML: {e}")
        raise
    except Exception as e:
        logging.error(f"Ошибка валидации данных: {e}")
        raise



# Загрузка данных при импорте модуля (можно закомментировать в продакшене)
orders = get_data_from_yaml("students_folder/Borodin/LabaMore1/data.yaml")
```

Листинг 2 – Сопроводительный материал для документации (формат .rst/описание проекта): порядок запуска tox, сборки документации Sphinx, структура каталогов и публикация на GitHub Pages

```
Borodin
Бородин Глеб
В лабораторной работе №3 реализована система валидации данных с помощью Pydantic. Основная цель — создать строгую и надёжную модель данных для маркетплейса, включая: - пользователей (UserSpec, ProfileSpec), - товары и услуги (ItemSpec, ServiceSpec), - заказы (OrderLineSpec, OrderSpec, OrdersSpec).

Все модели используют строгую валидацию: - типы данных, - ограничения (gt=0 для ID и цен), - регулярные выражения для текстовых полей (только кириллица, пробелы и дефисы), - проверка email через EmailStr, - URL-поля через HttpUrl, - запрет дополнительных полей (extra='forbid').
```

Мои модели данных:

```
from pydantic import BaseModel, Field, ConfigDict, EmailStr
from typing import Optional, Literal, List, Union

class UserSpec(BaseModel):
    user_id: int = Field(gt=0)
    username: str = Field(pattern=r'^[a-яA-ЯёЁ\s\-]+$')
    surname: str = Field(pattern=r'^[a-яA-ЯёЁ\s\-]+$')
    second_name: Optional[str] = Field(None, pattern=r'^[a-яA-ЯёЁ\s\-]+$')
    email: EmailStr
    status: Literal['active', 'non-active']
    model_config = ConfigDict(extra='forbid')

class ProfileSpec(UserSpec):
    bio: str = Field(pattern=r'^[a-яA-ЯёЁ\s\-.,!?]+$')
    url: HttpUrl
    model_config = ConfigDict(extra='forbid')

class ItemSpec(BaseModel):
    item_id: int = Field(gt=0)
    name: str = Field(pattern=r'^[a-яA-ЯёЁ\s\-]+$')
    desc: str = Field(pattern=r'^[a-яA-ЯёЁ\s\-]+$')
    price: float = Field(gt=0)
    model_config = ConfigDict(extra='forbid')

class ServiceSpec(BaseModel):
    service_id: int = Field(gt=0)
```

```
    name: str = Field(pattern=r'^[a-яA-ЯёЁ\s\-]+$')
    desc: str = Field(pattern=r'^[a-яA-ЯёЁ\s\-]+$')
    price: float = Field(gt=0)
    model_config = ConfigDict(extra='forbid')

class OrderLineSpec(BaseModel):
    order_id: int
    order_line_id: int
    item_line: Union[ItemSpec, ServiceSpec]
    quantity: float = Field(gt=0)
    line_price: float = Field(gt=0)

    @model_validator(mode='after')
    def validate_line_price(self) -> 'OrderLineSpec':
        calculated = self.quantity * self.item_line.price
        if abs(self.line_price - calculated) > 1e-9:
            raise ValueError(
                f'line_price {self.line_price} != quantity * item_line.price {calculated}'
            )
        return self

    model_config = ConfigDict(extra='forbid')

class OrderSpec(BaseModel):
    order_id: int
    user_info: ProfileSpec
    items_line: List[OrderLineSpec]
    model_config = ConfigDict(extra='forbid')
```

```
class OrdersSpec(BaseModel):
    market_place_orders: List[OrderSpec]
    model_config = ConfigDict(extra='forbid')
```

Тестирование и сборка

Все модели покрыты параметризованными тестами через pytest.

Тесты запускаются с помощью tox в изолированном окружении.

Для запуска тестов лабораторной работы №3 используется команда:

tox -e borodin_lab3

Документация генерируется автоматически с помощью Sphinx и sphinx-apidoc.

Автоматическая сборка документации настроена через GitHub Actions.

Итоговая HTML-документация публикуется на GitHub Pages.

Структура проекта

students_folder/Borodin/LabaMore1/lab_1.py — исходный код моделей

tests/Borodin/ — тесты с маркером @pytest.mark.borodin_lab3

tests/src/Borodin/lab_3/ — YAML-файлы с тестовыми данными

docs_src/ — исходники документации (RST)

docs/_build/html/ — сгенерированная HTML-документация

Эта документация автоматически обновляется при каждом пуше в основную ветку репозитория.