

??????????

- [IPython](#)
- [Poetry](#)

# IPython

[Интерактивная оболочка](#) для языка программирования [Python](#), которая предоставляет расширенную [интроспекцию](#), дополнительный командный синтаксис, подсветку кода и автоматическое дополнение.

# Poetry

## ??? ????? Poetry?

Poetry - это инструмент для управления проектами на Python, который предоставляет следующие возможности:

- управление зависимостями с воспроизводимыми установками и резолвером конфликтов
- автоматическое управление виртуальными окружениями
- сборка и публикация.

## ????????

Установка Poetry выполняется очень просто как на Unix-системах:

```
curl -sSL https://install.python-poetry.org | python3 -
```

Так и на Windows:

```
(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content | py -
```

Далее, в зависимости от вашей системы, необходимо добавить соответствующую директорию в PATH:

```
# Linux, MacOS, WSL
$HOME/.local/bin

# Windows
%APPDATA%\Python\Scripts
```

Перезагружаем оболочку и проверяем корректность установки:

```
poetry --version

# Вывод должен быть примерно таким
# Poetry (version 1.4.2)
```



```
Compatible Python versions [^3.8]:
```

```
Would you like to define your main dependencies interactively? (yes/no) [yes] no
```

```
Would you like to define your development dependencies interactively? (yes/no) [yes] no
```

```
Generated file
```

```
[tool.poetry]
name = "my-project2"
version = "0.1.0"
description = "My Project, but second"
authors = ["Timur Kasimov"]
license = "MIT"
readme = "README.md"
packages = [{"include = "my_project2"}]

[tool.poetry.dependencies]
python = "^3.8"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

```
Do you confirm generation? (yes/no) [yes] yes
```

?????????? ?????????????? ???????????

Команды **new** и **init** не создают виртуальных окружений. При первом выполнении команд, связанных с установкой зависимостей, Poetry создает виртуальное окружение, выбрав базовый интерпретатор по следующей логике:

1. Poetry проверяет, активировано ли уже какое-то виртуальное окружение. Если да, то оно будет использовано
2. Если никакое виртуальное окружение не активировано, то Poetry попытается использовать Python, который был использован при установке Poetry
3. Если версия Python с предыдущего шага несовместима с версией, указанной в **pyproject.toml**, то Poetry попросит явно активировать нужную версию

Советую сразу указать корректный базовый интерпретатор, выполнив в папке проекта следующую команду:

```
poetry env use python3.8 # Если python3.8 есть в PATH
poetry env use /path/to/python # Можно указать и полный путь
```

Если вы используете `pyenv`, можно использовать экспериментальную фичу Poetry:

```
poetry config virtualenvs.prefer-active-python true
pyenv install 3.9.8
pyenv local 3.9.8
```

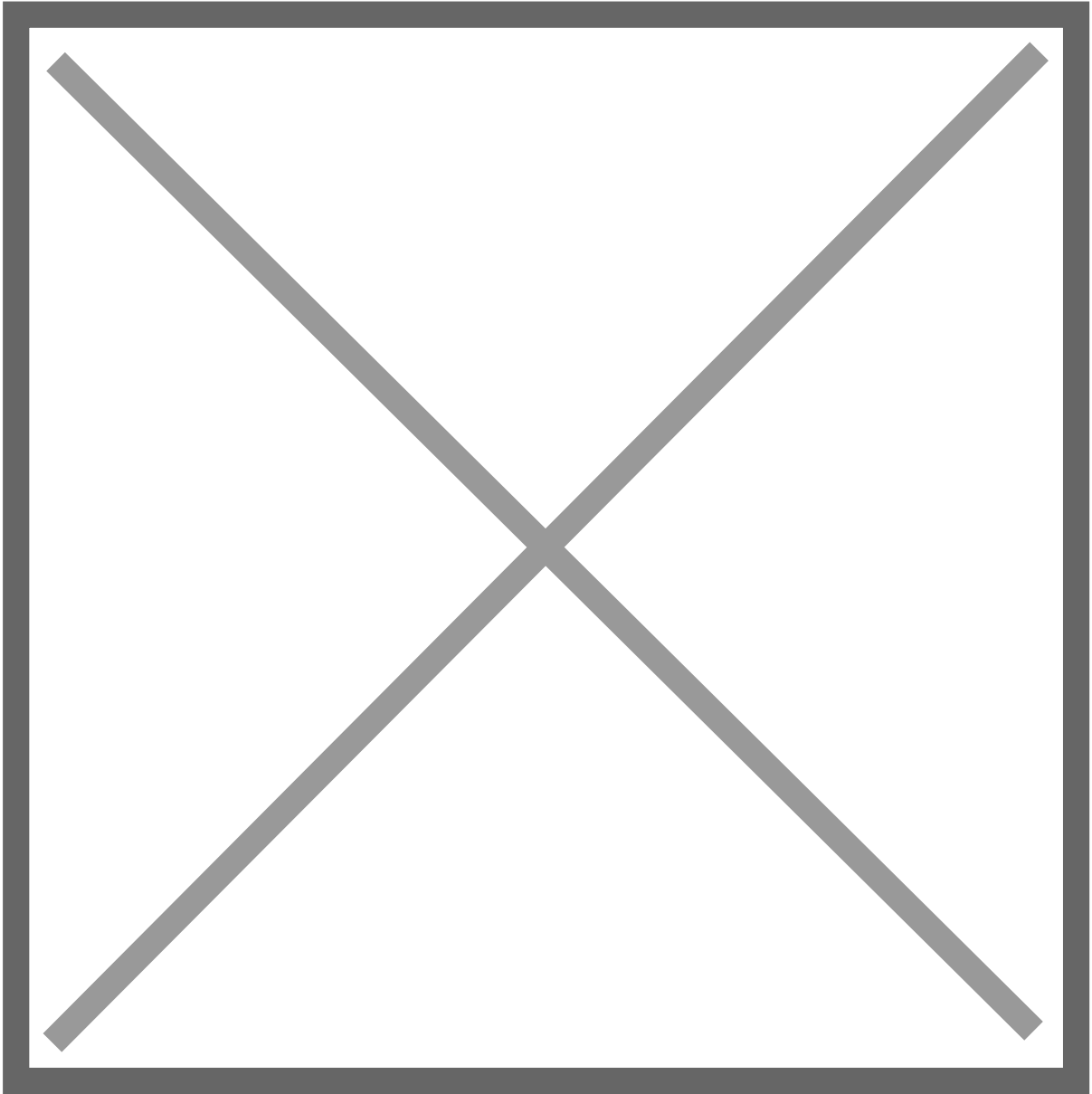
По умолчанию, Poetry создает виртуальные окружения в папке `{cache_dir}/virtualenvs`. Если вы хотите, чтобы виртуальное окружение находилось в папке проекта, можно выполнить следующую команду:

```
poetry config virtualenvs.in-project true
```

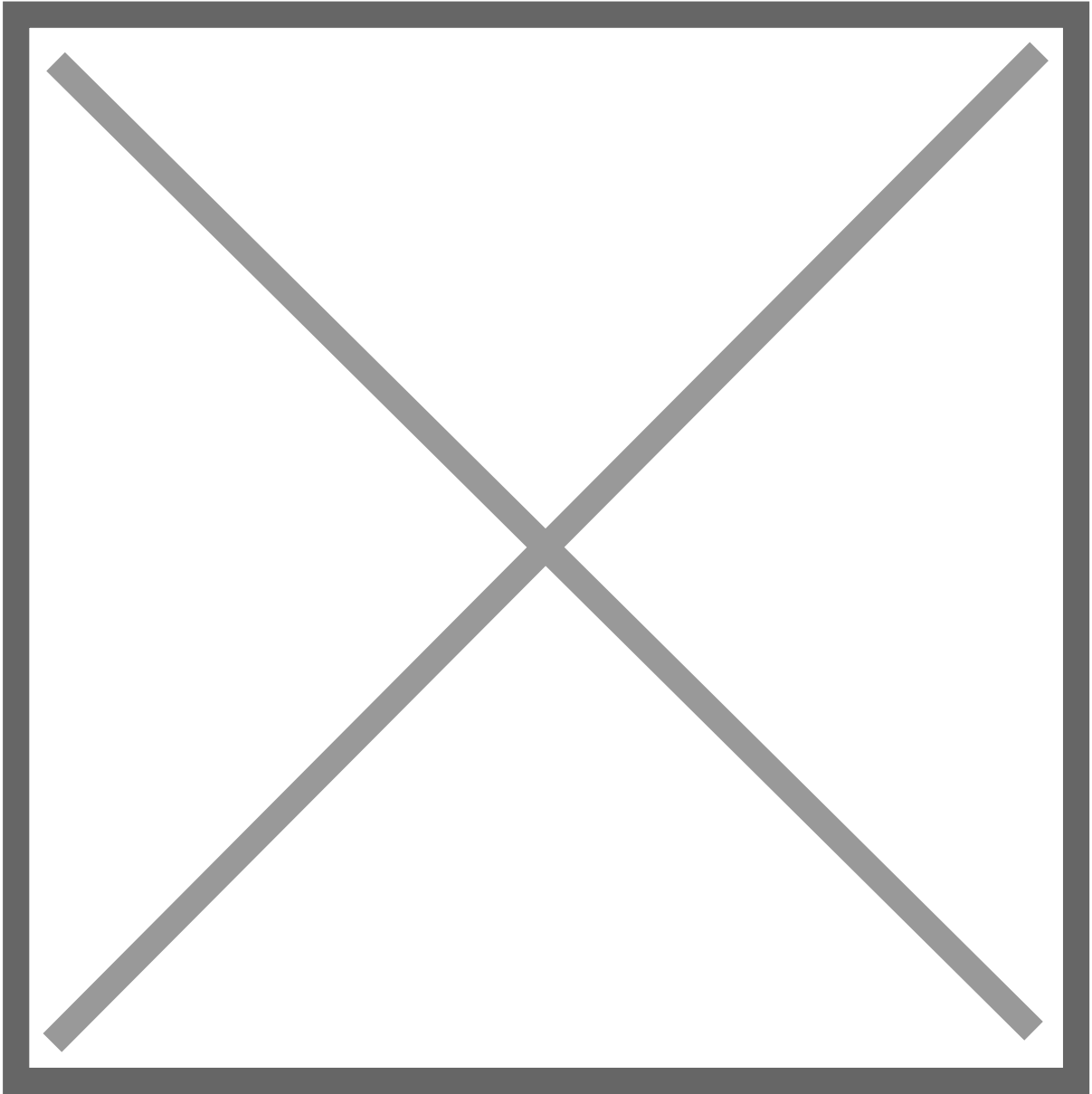
Теперь создаваемое виртуальное окружение будет находиться в папке `.venv` в корне проекта.

## ??? ??????? ??????? ?????? PyCharm

PyCharm поддерживает интеграцию с Poetry. Можно выбрать Poetry как при создании нового проекта:



Так и в существующем проекте. Для этого необходимо в правом нижнем углу нажать "No Interpreter" (если у вас до этого не был настроен интерпретатор в проекте) или на имя интерпретатора, далее "Add New Interpreter" → "Add Local Interpreter", и в открывшемся окне выбрать "Poetry environment":



## pyproject.toml ? poetry.lock

**pyproject.toml** - это файл, который содержит в себе всю информацию о проекте: метаданные (имя, версия и т.п.) и зависимости, а также в нем могут присутствовать настройки других инструментов.

Файл **poetry.lock** же содержит в себе все зависимости проекта с зафиксированными версиями и формируется автоматически (пожалуйста, не редактируйте его вручную).

**[tool.poetry]** содержит в себе метаданные:

- name - имя проекта. Должно быть [валидным по PEP 508](#);
- version - версия проекта. Должна быть [валидной по PEP 440](#);
- description - короткое описание проекта;
- license - лицензия;

- authors - авторы проекта в формате "name <email>". Должен присутствовать как минимум один автор.

Остальные спецификаторы можно найти в документации, они не являются обязательными.

**[tool.poetry.dependencies]** содержит в себе версию Python и основные зависимости проекта (так называемую main-группу).

В [PEP-517](#) был представлен стандартный способ определять альтернативные системы сборки для Python-проектов. Poetry совместим с PEP-517 и использует poetry-core для сборки, что и обозначено в секции **build-system**:

```
[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Многие питоновские инструменты поддерживают конфигурацию через **pyproject.toml**. Например, в моих проектах в нем уютно расположились настройки isort и mypy:

```
[tool.isort]
line_length = 120
...
force_grid_wrap = 2

[tool.mypy]
python_version = 3.9
...
follow_imports = "skip"
```

?????????? ? ??????????? ??????????

?????????????? ?????????????? + ?????????????????????

Команда add добавляет зависимость в **pyproject.toml**, выполняет разрешение зависимостей и устанавливает зависимость:

```
poetry add fastapi
```

В секции **[tool.poetry.dependencies]** появилась наша зависимость:

```
[tool.poetry.dependencies]
python = "^3.8"
```

```
fastapi = "^0.95.2"
```

Рассмотрим, как указывать версии при установке. Первый вариант - это Caret Requirements:

Требование	Допустимые версии
<code>^1.2.3</code>	<code>&gt;=1.2.3, &lt;2.0.0</code>
<code>^1.2</code>	<code>&gt;=1.2.0, &lt;2.0.0</code>
<code>^1</code>	<code>&gt;=1.0.0, &lt;2.0.0</code>
<code>^0.2.3</code>	<code>&gt;=0.2.3, &lt;0.3.0</code>
<code>^0.0.3</code>	<code>&gt;=0.0.3, &lt;0.0.4</code>
<code>^0.0</code>	<code>&gt;=0.0.0, &lt;0.1.0</code>
<code>^0</code>	<code>&gt;=0.0.0, &lt;1.0.0</code>

Второй вариант - это Tilde requirements, что позволяет указывать минимальную допустимую версию с некоторой возможностью обновления:

Требование	Допустимые версии
<code>~1.2.3</code>	<code>&gt;=1.2.3, &lt;1.3.0</code>
<code>~1.2</code>	<code>&gt;=1.2.0, &lt;1.3.0</code>
<code>~1</code>	<code>&gt;=1.0.0, &lt;2.0.0</code>

Wildcard requirements позволяют обновление до последней версии в позиции, где расположен символ "\*":

Требование	Допустимые версии
<code>*</code>	<code>&gt;=0.0.0</code>
<code>1.*</code>	<code>&gt;=1.0.0, &lt;2.0.0</code>
<code>1.2.*</code>	<code>&gt;=1.2.0, &lt;1.3.0</code>

Inequality requirements позволяют вручную указать диапазон допустимых версий, или же точную версию:

```
>=1.2.0  
>1  
<2  
!=1.2.3  
==1.5.2
```

```
# Можно комбинировать  
>=1.2,<1.5
```

Стоит обратить внимание на следующий момент: если вы указываете точную версию, и другим зависимостям требуется другая версия, то резолвер не сможет разрешить все конфликты и процедура установки (или обновления) не будет выполнена.

## Extras ? groups

Прежде чем идти дальше, хочется ввести 2 понятия - extras и groups. Они довольно схожи, но на деле служат для разных целей.

Dependency groups (далее просто группы) содержат в себе опциональные зависимости, используемые **только при разработке**. Установить зависимости из групп можно только через Poetry. Каждый проект содержит в себе одну неявную обязательную группу - main, которая находится в секции **[tool.poetry.dependencies]**.

Установка в группу выполняется следующим образом:

```
poetry add --group test pytest
```

Группа вместе с зависимостью появилась в pyproject.toml:

```
[tool.poetry.group.test.dependencies]  
pytest = "^7.3.1"
```

Extras же предназначены для введения дополнительных зависимостей, которые включают какую-либо функциональность в вашем проекте. Установить зависимость как extra можно вот так:

```
poetry add --extras postgres psycopg2-binary  
poetry add --extras mysql --extras database mysql-connector-python # Можно перечислять  
несколько extras
```

Зависимости появились в **[tool.poetry.dependencies]** с пометкой extras:

```
[tool.poetry.dependencies]  
python = "^3.8"  
psycopg2-binary = {version = "^2.9.6", extras = ["postgres"]}  
mysql-connector-python = {version = "^8.0.33", extras = ["mysql", "database"]}
```

После сборки и публикации можно будет устанавливать ваш пакет как обычно:

```
poetry add "my-project[postgres,database]"
poetry add "my-project[mysql]"
```

## ????????? ? Git ? ?????????? package registries

Установка из публичных репозиториях выполняется следующим образом:

```
poetry add "git+https://github.com/psf/requests" # будет использован последний коммит с
основной ветки
poetry add "git+https://github.com/psf/requests#update-3.0" # будет использован последний
коммит с ветки update 3.0

# Все то же самое, но через SSH
poetry add "git+ssh://git@github.com:requests/requests.git"
poetry add "git+ssh://git@github.com:requests/requests.git#update-3.0"
```

Для установки из частных git-репозиториях можно воспользоваться SSH, но в таком случае будет выполняться сборка.

Если же у вас есть частный package registry, последовательность действия следующая (рассматривать будем на примере GitLab). Добавляем источник:

```
poetry source add my-repo "https://my.gitlab.com/projects/1/packages/pypi/simple"
```

Добавленный источник появился в файле **pyproject.toml**:

```
[[tool.poetry.source]]
name = "my-repo"
url = "https://my.gitlab.com/projects/1/packages/pypi/simple"
default = false
secondary = false
```

Настраиваем аутентификацию (для GitLab Package Registry рекомендую использовать [Personal Access Token](#)):

```
poetry config http-basic.my-repo <token-name> <secret-token>
```

Выполняем установку, указав источник:

```
poetry add --source my-repo my-package
```

Не забудьте изменить [my.gitlab.com](https://my.gitlab.com) на адрес вашего GitLab, а также указать корректный ID проекта. Имя источника в примере используется my-pyro, но можно выбрать любое другое на ваше усмотрение.

?????????? ?? ???????

Poetry позволяет устанавливать зависимости как из локальных файлов и папок:

```
poetry add package-1.0.0.tar.gz
poetry add package-1.0.0.whl
poetry add ~/my/local/package
```

Так и с удаленных серверов:

```
poetry add https://download.pytorch.org/whl/cpu/torch-2.0.0%2Bcpu-cp39-cp39-linux_x86_64.whl
```

????? ??? ???????????

В этом разделе кратко пробежимся по основным опциям, которые можно использовать при установке зависимостей:

Опция	Пояснение
--group (-G)	Группа, в которую необходимо добавить зависимость. Если такой группы не существует, она будет создана
--editable	Установить зависимость в editable режиме
--extras	Extras для активации зависимости
--optional	Добавить зависимость как опциональную
--python	Версия Python, для которой зависимость должна быть установлена
--platform	Платформа, для которой зависимость должна быть установлена (linux, darwin или win32)
--source	Имя источника, из которого будет установлена зависимость. По умолчанию используется PyPI, о настройке собственных источников ниже
--allow-prereleases	Разрешить установку пререлизов
--dry-run	Вывести последовательность действий, но не выполнять никаких операций
--lock	Не выполнять установку, только обновить lock-файл

poetry install

Команда `install` при запуске выполняет следующую последовательность действий:

- читает файл **pyproject.toml**
- если существует файл **poetry.lock**, то версии зависимостей берутся из него. Если его не существует, Poetry выполнит разрешение зависимостей и создаст его
- устанавливает зависимости

Рассмотрим основные опции:

Опция	Пояснение
<code>--without</code>	Группы, которые будут проигнорированы при установке
<code>--with</code>	Группы, которые будут установлены
<code>--only</code>	Установить только определенные группы (в этом случае <code>--without</code> и <code>--with</code> будут проигнорированы)
<code>--only-root</code>	Установить только проект, проигнорировав все зависимости
<code>--sync</code>	Синхронизировать виртуальное окружение с lock-файлом
<code>--no-root</code>	Не устанавливать сам проект
<code>--dry-run</code>	Вывести последовательность действий, но не выполнять никаких операций
<code>--extras (-E)</code>	Extras, которые необходимо установить
<code>--all-extras</code>	Включить все extras в установку
<code>--compile</code>	Транслировать исходники в байт-код

????????? ????????

Для удаления какой-либо зависимости можно воспользоваться командой `remove`:

```
poetry remove requests
```

Если зависимость находится в какой-то группе, используйте опцию `--group`:

```
poetry remove --group my-group requests
```

????????? ????????????????

Команда `lock` позволяет зафиксировать зависимости, обновив файл **poetry.lock**:

```
poetry lock
```

Будьте внимательны! По умолчанию, **poetry lock** попытается выполнить обновление всех зависимостей до последних допустимых версий. Чтобы этого избежать, используйте опцию `--no-update`.

## ?????? ?????? ????? Poetry

### poetry shell

С помощью **poetry shell** можно запустить оболочку с активированным виртуальным окружением. Если его не существует, то оно будет создано.

Т.к. **poetry shell** не просто активирует виртуальное окружение, а именно создает оболочку, то стоит использовать для выхода **exit**, а не **deactivate**.

### ?????? ? pyproject.toml

В файл **pyproject.toml** можно включить секцию **[tool.poetry.scripts]**, которая содержит в себе описание скриптов, которые будут доступны к использованию при установке проекта:

```
[tool.poetry.scripts]
my-script = "my_package.console:run"
```

Здесь мы описываем скрипт `my-script`, при запуске которого выполнится функция `run` из модуля `console` из пакета `my_package`. При обновлении или добавлении скриптов не забывайте выполнять команду **poetry install**, чтобы сделать их доступными в виртуальном окружении проекта.

### poetry run

Команда **run** позволяет запускать команды в виртуальном окружении проекта. Например:

```
poetry run python --version
```

Что более интересно, с помощью **run** можно запускать скрипты, определенные в **pyproject.toml**:

```
poetry run my-script
```

## ?????? ? ?????????????? ??????????

Представим, что вы разрабатываете какую-то библиотеку, и настал торжественный момент сборки и публикации. Используя Poetry, сделать это можно вот так.

Собираем наш пакет:

```
poetry build # собираем как sdist, так и wheel
poetry build --format sdist # собираем только sdist
poetry build --format wheel # собираем только wheel
```

Для публикации на PyPI предварительно получаем [API token](#) и устанавливаем его:

```
poetry config pypi-token.pypi <my-token>
```

И, наконец, публикуем:

```
poetry publish
```

Если же необходимо опубликовать пакет в приватный registry (например, в GitLab), то настройка репозитория немного усложняется. Предварительно не забываем сгенерировать [Personal Access Token](#):

```
poetry config repositories.gitlab "my.gitlab.com/projects/1/packages/pypi"
poetry config http-basic.gitlab <token-name> <secret-token> # здесь token-name и secret-token
- ваш Personal Access Token
```

Не забудьте изменить [my.gitlab.com](#) на адрес вашего GitLab, а также указать корректный ID проекта. Здесь имя репозитория в Poetry выбрано gitlab, можно использовать другое на ваше усмотрение. Пора публиковать:

```
poetry publish --repository gitlab
```

В качестве бонуса - пример публикации при использовании GitLab CI:

```
build_and_publish:
  stage: build_and_publish
  script:
    - poetry install --without dev
    - poetry build
    - poetry config repositories.gitlab
      "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/pypi"
    - poetry config http-basic.gitlab gitlab-ci-token "${CI_JOB_TOKEN}"
    - poetry publish --repository gitlab
```

# ????????? poetry

В этом разделе кратко пробежимся по основным командам для настройки и управления непосредственно Poetry.

## ?????????? Poetry

Обновиться можно как на более новую, так и на более старую версию. **Будьте аккуратны** - если обновиться на версию ниже 1.2, то обновление на более высокую версию будет невозможно и придется переустанавливать Poetry:

```
poetry self update 1.3.2
```

## ?????????? ???????????

Для установки, удаления и обновления плагинов можно пользоваться командами **self add**, **self remove** и **self update** соответственно. Механизм работы этих команд аналогичен механизму работы команд **add**, **remove**, **update** за исключением того, что команды в пространстве имен **self** выполняется в виртуальном окружении самого Poetry.

При управлении плагинами нам также доступны команды **self lock** и **self install**. Работают они аналогично вышеупомянутым командам.

## ?????????? Poetry

Команда **config**, помимо управления репозиториями, позволяет редактировать настройки Poetry:

```
# [setting-key] - имя настройки
# [setting-value] - значение
poetry config [options] [setting-key] [setting-value1] ... [setting-valueN]
```

Чаще всего используются следующие опции:

- **cache-dir** (строка) - директория для кэша Poetry
- **virtualenvs.create** (true / false) - создавать ли виртуальные окружения для проектов (если не существуют). Будьте внимательны: если вы установите данную настройку в false и Poetry не обнаружит виртуальное окружение в папках {cache-dir}/virtualenvs или {project-dir}/.venv, то установка зависимостей будет выполняться в системный Python.
- **virtualenvs.in-project** (true / false) - создавать виртуальные окружения в корне проекта. По умолчанию, Poetry создает виртуальные окружения в папке {cache-dir}/virtualenvs.

Остальные опции можно найти в [документации](#).

# ????? - Docker-????? ? ?????????? ? ????????????????? Poetry

Давайте рассмотрим, как собрать максимально компактный Docker-образ, если разработку нашего проекта мы вели с помощью Poetry. Имеем следующий **pyproject.toml**:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Your Name <you@example.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = ">=3.8.1,<4.0"
fastapi = "^0.95.2"
sqlalchemy = "^2.0.15"
uvicorn = "^0.22.0"

[tool.poetry.group.test.dependencies]
pytest = "^7.3.1"

[tool.poetry.group.dev.dependencies]
flake8 = "^6.0.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Наше приложение - это вот такое замечательное REST API:

```
from fastapi import FastAPI

app = FastAPI()

@app.get('/hello')
def hello():
    return 'hello, world!'
```

Первый способ, который сразу приходит в голову - установим Poetry в образ, через него поставим зависимости и запустим наш сервис. Dockerfile в таком случае выглядит примерно вот так:

```
FROM python:3.8.16-slim-bullseye

WORKDIR /app

COPY poetry.lock pyproject.toml ./

RUN python -m pip install --no-cache-dir poetry==1.4.2 \
    && poetry config virtualenvs.create false \
    && poetry install --without dev,test --no-interaction --no-ansi \
    && rm -rf $(poetry config cache-dir)/{cache,artifacts}

COPY app.py ./

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Собираем образ:

```
docker image build -t poetry-tutorial-naive:latest -f Dockerfile.naive .
```

Получаем образ размером 225MB. Неплохо, но можно и меньше. Давайте попробуем применить multi-stage сборку. На первом этапе с помощью Poetry подготовим виртуальное окружение, а потом скопируем его в наш итоговый образ. Итак, Dockerfile:

```
FROM python:3.8.16-slim-bullseye AS builder

WORKDIR /app
COPY poetry.lock pyproject.toml ./

RUN python -m pip install --no-cache-dir poetry==1.4.2 \
    && poetry config virtualenvs.in-project true \
    && poetry install --without dev,test --no-interaction --no-ansi

FROM python:3.8.16-slim-bullseye

COPY --from=builder /app /app
COPY app.py ./
```

```
CMD ["/app/.venv/bin/uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Собираем:

```
docker image build -t poetry-tutorial-copy:latest -f Dockerfile.copy .
```

Получаем образ размером 159MB. Уже гораздо лучше!

Может быть, если экспортировать все зависимости в привычный requirements.txt и установить через pip, то получится еще компактнее? Попробуем:

```
FROM python:3.8.16-slim-bullseye AS builder

COPY poetry.lock pyproject.toml ./
RUN python -m pip install --no-cache-dir poetry==1.4.2 \
    && poetry export --without-hashes --without dev,test -f requirements.txt -o
requirements.txt

FROM python:3.8.16-slim-bullseye

WORKDIR /app

COPY --from=builder requirements.txt ./
RUN python -m pip install --no-cache-dir -r requirements.txt

COPY app.py ./

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Собираем образ:

```
docker image build --no-cache -t poetry-tutorial-requirements:latest -f
Dockerfile.requirements
```

Получилось 163MB, что немного больше, чем у предыдущего способа, который и вышел победителем.

Итоговая таблица:

Способ	Размер образа
Нативный	225MB (-0.0%)

Экспорт requirements.txt и установка через pip	163MB (-27.5%)
Копирование venv	159MB (-29.3%)

Обратите внимание на следующие вещи:

- жестко фиксируйте версию Poetry в ваших Dockerfile'ах. Разработчики Poetry очень любят что-то ломать от релиза к релизу, или объявлять ставшие привычными вещи deprecated.
- не тащите лишние зависимости в ваши образы. Используйте флаг --without.