Урок 8. Work List Planner 2.0.0: развитие копилки, API-тесты и выпуск новой версии

В предыдущих уроках мы создали и развернули Work List Planner на VPS, настроили FastAPI, базу данных SQLite, авторизацию, CI/CD и резервное копирование.

В этом уроке продолжим развивать приложение уже как полноценный программный продукт. Мы расширим функциональность копилки, добавим проверки API, подготовим документацию к релизу и выпустим версию **2.0.0**.

Главная цель занятия — не просто написать новый код, а научиться безопасно изменять работающий проект, проверять результат автоматическими тестами и выпускать новую версию без ручного копирования файлов на сервер.

Что было до начала работы

К началу занятия приложение уже работало на сервере:

- проект располагался в `/var/www/planner-app`;
- backend был написан на Python с использованием FastAPI;
- SQLite использовалась для хранения пользователей, задач и данных приложения;
- приложение запускалось через systemd-службу `planner.service`;
- исходный код хранился в GitHub-репозитории `im449/work-list-planner`;
- GitHub Actions выполнял автоматические проверки и деплой.

Основной задачей было развитие раздела накоплений и подготовка следующего стабильного выпуска.

Шаг 1. Развиваем раздел накоплений

Ранее накопления представляли собой календарь, в котором пользователь мог планировать откладываемые суммы. В процессе работы стало понятно, что одного календаря недостаточно: пользователь может копить деньги на несколько разных целей и должен видеть состояние каждой из них отдельно.

В результате доработки реализовали следующие возможности:

- создание нескольких целей накоплений;
- указание параметров и целевых сумм;
- редактирование целей;
- изменение сумм и отметок по отдельным дням календаря;
- получение сводки по накоплениям;
- динамический пересчёт показателей и рекомендаций;
- архивирование завершённых или временно неактуальных целей;
- восстановление целей из архива.

Например, пользователь может одновременно копить на резервный фонд, отпуск и крупную покупку. Для каждой цели ведутся собственные записи, а данные разных пользователей не должны пересекаться.

Отдельное внимание уделили архивированию. Архивирование не должно уничтожать историю накоплений: при восстановлении цели сохранённые записи должны оставаться доступными.

Это важный принцип разработки: изменение статуса объекта и удаление его данных — разные операции.

Шаг 2. Проверяем API накоплений

После добавления функциональности необходимо проверить не только нормальный сценарий, но и ситуации, в которых пользователь вводит неправильные данные или пытается обратиться к чужой информации.

Для этого расширили набор автоматических API-тестов.

Тесты проверяют:

- регистрацию и авторизацию;
- создание, получение, изменение и удаление задач;
- создание и редактирование целей накоплений;
- чтение и сохранение календаря;
- расчёт сводных показателей;
- архивирование и восстановление целей;
- запрет отрицательных сумм;
- обработку некорректных параметров и диапазонов дат;
- запрет записи в дни за пределами календаря;
- обработку бесконечных и некорректных числовых значений;
- запрет уменьшения календаря ниже уже сохранённых дней;
- изоляцию данных разных пользователей;
- невозможность удалить последнюю активную цель;
- сохранение записей при архивировании цели.

Особенно важны проверки изоляции данных. Если один пользователь знает идентификатор чужой цели, он не должен получить возможность читать, изменять, архивировать или удалять её.

Запускаем тесты

На сервере переходим в каталог проекта:

```
cd /var/www/planner-app
```

Сначала проверяем синтаксис Python:

```
.venv/bin/python -m py_compile main.py
```

Затем запускаем автоматические тесты:

```
.venv/bin/python -m pytest tests/ -v
```

Для проверки качества изменений в Git используем:

```
git diff --check
```

Эта команда помогает обнаружить лишние пробелы в конце строк и другие проблемы оформления diff.

Результат

В итоговом прогоне успешно прошли все **21 API-тест**:

```
====================== 21 passed, 41 warnings in 17.97s ======================
```

Код завершения тестового процесса — `0`, то есть проверка завершилась успешно.

При этом pytest сообщил о предупреждениях, связанных с устаревающими возможностями библиотек и использованием `datetime.utcnow()`. Они не помешали прохождению тестов, но их следует учитывать при дальнейшей технической очистке проекта.

Шаг 3. Проверяем PDF-отчёт

Отдельно проверили формирование отчёта по задачам в формате PDF.

В ходе предыдущей отладки обнаружилась проблема с отображением кириллицы. Для её решения была настроена работа с шрифтом DejaVu Sans и его вариантами.

Также проверили соответствие frontend и backend: интерфейс должен обращаться к правильному маршруту API для формирования отчёта.

В итоге пользовательская проверка в браузере подтвердила, что:

- PDF-файл скачивается;
- отчёт содержит задачи;
- русский текст отображается корректно.

Этот этап показывает, почему недостаточно проверить только HTTP-статус. Даже если сервер возвращает `200 OK`, документ всё равно может оказаться пустым или неправильно отображать текст.

Поэтому после исправления проверяем и технический ответ сервера, и фактический результат в браузере.

Шаг 4. Проверяем автоматизацию GitHub Actions

В проекте уже настроен workflow GitHub Actions. Он запускает тестирование, а при успешной проверке изменений в основной ветке выполняет деплой.

Схема выглядит следующим образом:

1. Разработчик фиксирует изменения в Git.
2. Изменения отправляются в GitHub.
3. GitHub Actions запускает автоматические тесты.
4. При успешном завершении тестов выполняется задание деплоя.
5. На сервере запускается `deploy.sh`, который получает обновлённый код, проверяет синтаксис Python и перезапускает приложение.
6. Скрипт проверяет доступность приложения и основных маршрутов.

В проекте используется скрипт:

```
/var/www/planner-app/deploy.sh
```

В нём настроены строгая обработка ошибок, получение изменений из `origin/main`, компиляционная проверка `main.py`, перезапуск `planner.service` и проверки доступности приложения.

Это позволяет обнаруживать проблемы раньше и уменьшает количество ручных операций при выпуске обновлений.

Во время подготовки версии 2.0.0 мы убедились, что задания `test` и `deploy` в GitHub Actions завершились успешно.

Шаг 5. Обновляем номер версии

Когда функциональность реализована, тесты пройдены и основные сценарии проверены, необходимо зафиксировать версию приложения.

В `main.py` обновили ответ маршрута `/version`:

```
return {
    "version": "2.0.0",
    "build": "2026-10-13"
}
```

Таким образом, приложение сообщает собственную версию и дату сборки.

Это удобно для диагностики: при обращении к серверу можно быстро проверить, какая версия действительно работает, а не ориентироваться только на номер коммита в GitHub.

После изменения проверили синтаксис:

```
.venv/bin/python -m py_compile main.py
```

Затем убедились, что `git diff --check` не обнаруживает проблем.

Шаг 6. Добавляем историю изменений

Для релиза создали файл `CHANGELOG.md`.

Его задача — зафиксировать основные изменения версии в понятном для разработчиков и пользователей виде.

В историю изменений версии 2.0.0 вошли:

- поддержка нескольких целей накоплений;
- редактирование сумм и отметок по отдельным дням;
- сводка накоплений и динамические рекомендации;
- архивирование и восстановление целей;
- расширенные API-тесты;
- улучшение календаря и расчёта показателей;
- проверки изоляции пользовательских данных;
- автоматическое тестирование и деплой через GitHub Actions.

История изменений помогает понимать, что вошло в конкретный выпуск, не изучая все коммиты по отдельности.

Шаг 7. Фиксируем релиз в Git

Перед коммитом проверили состояние рабочего дерева:

```
git status --short
git diff --check
```

В коммит включили только подготовленные файлы:

```
git add main.py CHANGELOG.md
```

Проверили состав изменений:

```
git diff --cached --stat
git diff --cached --check
```

Результат показал два изменённых файла: `main.py` и `CHANGELOG.md`.

После этого создали коммит:

```
git commit -m "release: prepare Work List Planner 2.0.0"
```

Git создал коммит:

```
3783f87 release: prepare Work List Planner 2.0.0
```

Теперь отправляем изменения в GitHub:

```
git push origin main
```

В результате ветка `main` на сервере и в GitHub была синхронизирована с коммитом релиза.

Важно: резервные копии и временные файлы в этот коммит не добавляли. В репозитории должны находиться исходный код, тесты и необходимые документы, а не случайные копии рабочих файлов.

Шаг 8. Проверяем приложение после деплоя

После отправки изменений проверили версию непосредственно на сервере.

Проверка версии

```
curl -sS http://127.0.0.1:8000/version
```

Ответ:

```
{
  "version": "2.0.0",
```

```
  "build": "2026-10-13"
}
```

Проверка health endpoint

```
curl -sS http://127.0.0.1:8000/health
```

Ответ:

```
{
  "status": "ok",
  "service": "planner"
}
```

Проверка systemd-службы

```
systemctl is-active planner
```

Результат:

```
active
```

Проверка Git

```
git status --short
git log -1 --oneline
```

Рабочее дерево оказалось чистым, а последний коммит совпал с отправленным релизным коммитом.

Это подтвердило, что изменения зафиксированы и сервер сообщает версию 2.0.0.

Шаг 9. Фиксируем версию тегом v2.0.0

Обычный Git-коммит фиксирует конкретное состояние кода. Для обозначения выпуска используется тег.

В проекте проверили тег:

```
git describe --tags --exact-match HEAD
```

Результат:

```
v2.0.0
```

Затем проверили его содержимое:

```
git show --no-patch --format=fuller v2.0.0
```

Тег указывает на коммит:

```
3783f87f948099c137fe47ed215c4f332c750898
```

То есть на коммит `release: prepare Work List Planner 2.0.0`.

При повторной попытке создать тег Git сообщил:

```
fatal: tag 'v2.0.0' already exists
```

Это не означает, что выпуск сломан: тег уже существовал. Проверка `git show` подтвердила, что он указывает на нужный коммит. Поэтому повторно создавать тег или принудительно перемещать его не потребовалось.

При работе с релизами важно различать три сущности:

- **Коммит** — конкретное состояние исходного кода.
- **Git-тег** — именованная отметка конкретного коммита, например `v2.0.0`.
- **GitHub Release** — опубликованная на GitHub страница выпуска с описанием и ссылкой на тег.

Наличие т

Revision #1
Created 2026-10-13 19:48:36 UTC by im
Updated 2026-10-13 19:49:05 UTC by im