ЛЕТНЯЯ ШКОЛА 2026 / ТРЕК DM

Парсинг с использованием Selenium и Playwright

Анна Иванова

Стажёр-исследователь Лаборатории поведенческой экономики и финансов НИУ ВШЭ

Преподаватель Департамента теоретической экономики НИУ ВШЭ (ДОЦ «Экономика»)

Методолог Лаборатории нейронаук и поведения человека Сбера

БАЗОВЫЕ ПОНЯТИЯ

Как связаны HTML, DOM и JavaScript

01

HTML

Текст разметки, который сервер присылает браузеру.

02

DOM

Дерево элементов страницы. Браузер строит его из этого текста и держит в памяти.

03

JavaScript

Код страницы. Он реагирует на действия, догружает данные и меняет дерево.

04

Страница

То, что мы видим на экране. Картинка текущего состояния дерева.

HTML `<div class="quotes"></div>`

JAVASCRIPT `fetch("/api/quotes")`

DOM `<div class="quote">…</div>`

Словом API называют адрес на сервере, по которому программа запрашивает готовые данные, чаще всего в формате JSON. Со страницы такой запрос обычно отправляет функция fetch().

ЧАСТЬ I

Данные, которых нет в ответе сервера

requests и BeautifulSoup читают только то, что сервер прислал сразу. Всё, что страница дорисовывает потом, для них не существует.

ЧАСТЬ I

Что умеют requests и BeautifulSoup

1. requests отправляет HTTP-запрос и возвращает ответ сервера.
2. BeautifulSoup разбирает этот ответ и строит из него дерево тегов.
3. По дереву мы ищем нужные элементы по тегу, классу или CSS-селектору.

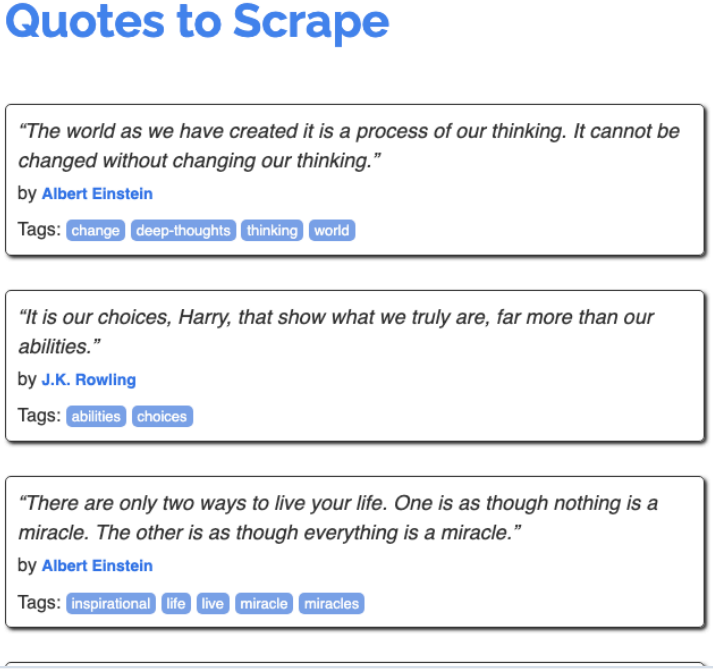
- JavaScript при этом не выполняется, потому что у библиотеки нет ни браузера, ни DOM, ни событий.
- Мы получаем ровно то, что сервер прислал в ответе, а не то, что страница показывает после работы скриптов.

> Нажмите Ctrl+U и посмотрите исходный код страницы. Если нужного текста нет и там, значит его дорисовывает JavaScript уже в браузере.

ЧАСТЬ I

Цитаты подгружаются отдельным запросом

СТРАНИЦА В БРАУЗЕРЕ



ИСХОДНЫЙ HTML / CTRL+U

```
<body>
  <div class="container">
    <h1>Quotes to Scrape</h1>
    <div class="quotes"></div>

    [JavaScript выполняется в браузере]
    var page = 1;
    var hasNextPage = true;

    function appendQuotes(quotes) {
      var html = $.map(quotes, function(d) {
        return "<div class='quote'>"
          + d['text']
          + "</div>";
      });
      $('.quotes').append(html);
```

Сервер отдаёт пустой контейнер div.quotes. Уже в браузере скрипт обращается к /api/quotes и вставляет цитаты в дерево страницы. Поэтому на экране текст есть, а в исходном HTML его нет.

ЧАСТЬ I

Статическую страницу можно разобрать без браузера

```
import requests
from bs4 import BeautifulSoup

HEADERS = {"User-Agent": "SummerSchool2026/1.0 (educational scraping class)"}

resp = requests.get("https://books.toscrape.com/", headers=HEADERS, timeout=20)
resp.raise_for_status()

# Сервер объявляет ISO-8859-1, хотя meta указывает utf-8.
# Передаём байты, чтобы BeautifulSoup определил кодировку по разметке.
soup = BeautifulSoup(resp.content, "html.parser")
cards = soup.select("article.product_pod")

print("карточек на странице:", len(cards))
for card in cards[:5]:
    print(f"  {card.select_one('p.price_color').get_text(strip=True):>8}"
          f"  {card.h3.a['title'][:52]}")
```

ЧАСТЬ I

Что видит requests на динамической странице

```
html = requests.get("https://quotes.toscrape.com/js/", headers=HEADERS, timeout=20).text
soup = BeautifulSoup(html, "html.parser")

print("найдено цитат:", len(soup.select("div.quote")))
print("длина ответа :", len(html), "символов")
print("данные встроены в JavaScript:", "var data =" in html)

i = html.find("var data =")
print("\n" + html[i:i + 300])
```

ЧАСТЬ II

Как браузер строит страницу

Пять шагов от первого ответа сервера до готовой картинки на экране.

ЧАСТЬ II

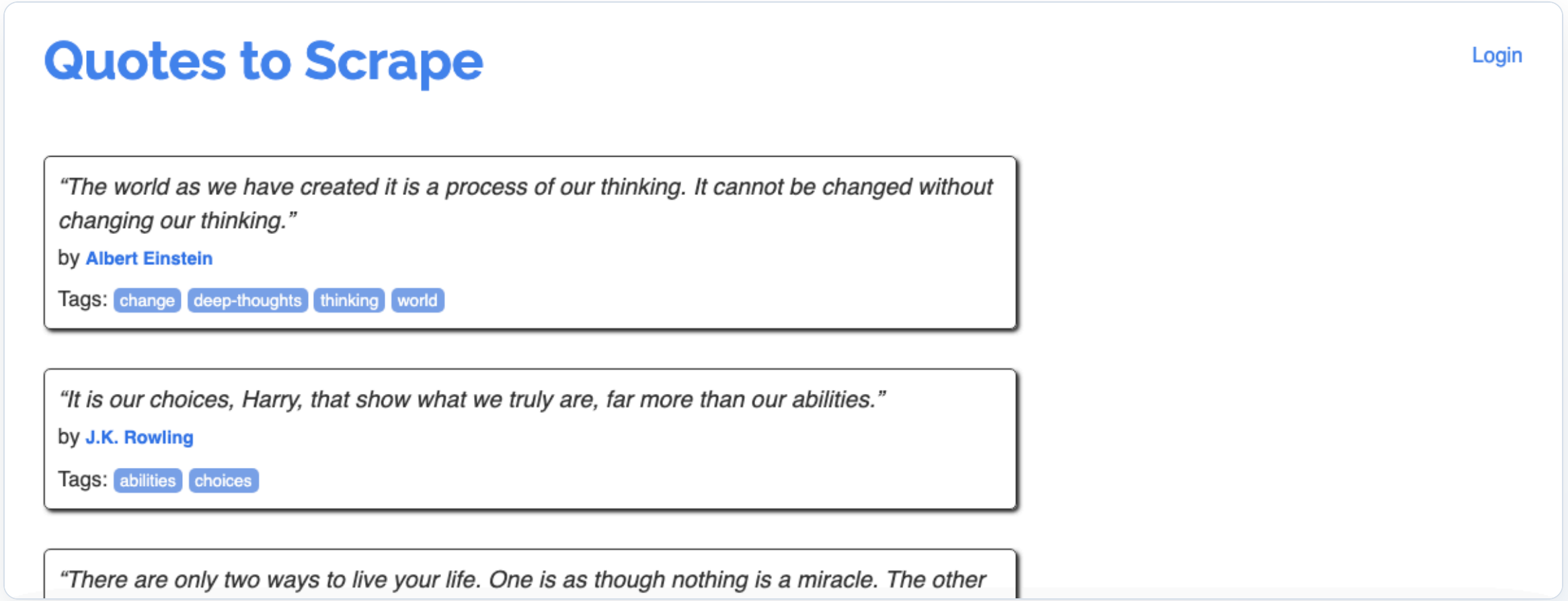
Этапы построения страницы

1. Браузер получает HTML. Часто это только каркас без содержимого.
2. Загружает стили и скрипты и запускает их.
3. Скрипты отправляют дополнительные запросы, XHR или fetch.
4. Пришедшие данные, обычно JSON, превращаются в элементы страницы.
5. Браузер рассчитывает разметку и рисует результат.

- requests останавливается на первом шаге.
- Selenium и Playwright запускают настоящий браузер, поэтому доходят до последнего.

ЧАСТЬ II

Network показывает, откуда пришли данные



Сразу после HTML браузер отдельно запрашивает /api/quotes?page=1. Из этого JSON и собираются карточки на странице.

ЧАСТЬ II

Прямой запрос к API

```
import time

def fetch_quotes_via_api(max_pages=20):
    out, page = [], 1
    while page <= max_pages:
        r = requests.get("https://quotes.toscrape.com/api/quotes",
                         params={"page": page}, headers=HEADERS, timeout=20)
        r.raise_for_status()
        payload = r.json()
        out.extend(payload["quotes"])
        if not payload["has_next"]:
            break
        page += 1
    return out

t0 = time.perf_counter()
quotes_api = fetch_quotes_via_api()
t_api = time.perf_counter() - t0
print(f"собрано цитат: {len(quotes_api)} за {t_api:.2f} с")
print("пример:", quotes_api[0]["author"]["name"], "|", quotes_api[0]["text"][:60])
```

ЧАСТЬ II

Когда без браузера не обойтись

- Содержимое появляется только после клика, наведения или раскрытия блока.
- Вход на сайт идёт через форму, переадресации и сохранение сессии.
- Параметры запроса или токен считает сам скрипт страницы.
- Нужен внешний вид, то есть скриншот, PDF, размеры и положение элементов.
- Нас интересует поведение интерфейса, а не только данные.

> Настоящий браузер съедает в разы больше памяти и времени, чем один HTTP-запрос. Запускать его стоит тогда, когда этого требует задача, а не на всякий случай.

ЧАСТЬ III

Selenium

Стандарт W3C WebDriver, зрелая экосистема и ожидания, которые нужно задавать самому.

ЧАСТЬ III

Как устроен Selenium

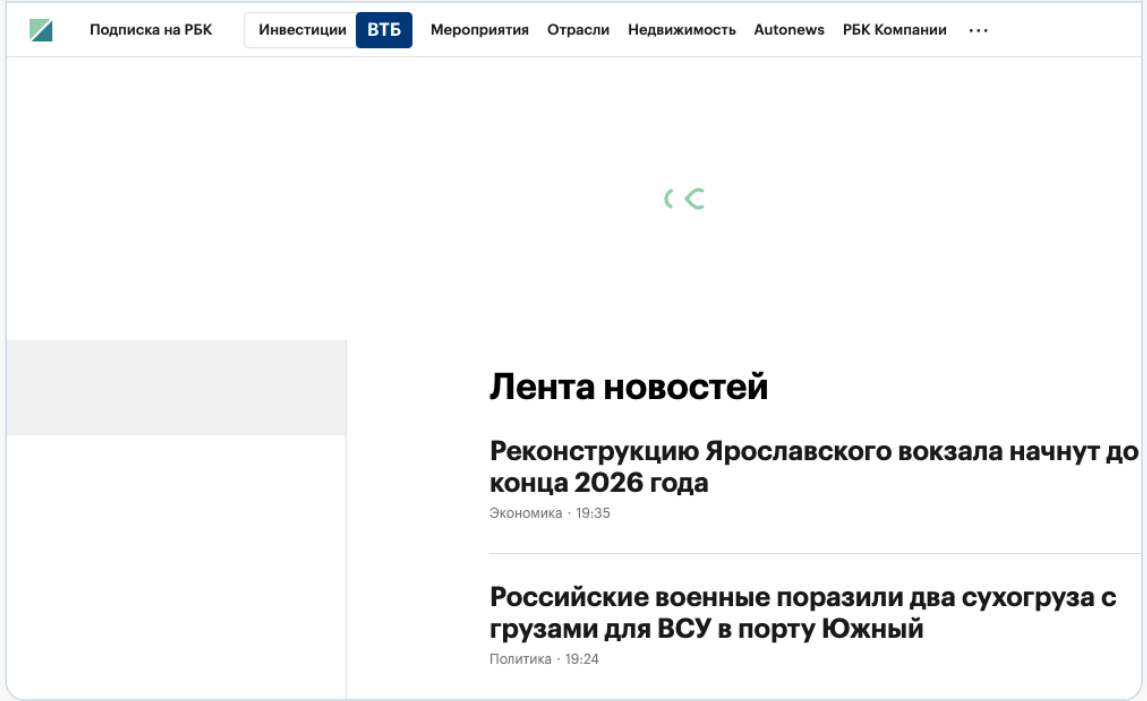
- Скрипт не управляет браузером напрямую. Команды идут через WebDriver, программу-посредника.
- Обмен построен на стандарте W3C WebDriver, поэтому один и тот же код работает с Chrome, Firefox, Safari и Edge.
- Каждая команда проходит через посредника, поэтому на мелких операциях накладные расходы заметнее.

> С версии 4.6 Selenium Manager сам находит и ставит подходящий драйвер. Скачивать chromedriver вручную обычно не нужно.

ЧАСТЬ III

Лента РБК после выполнения JavaScript

СТРАНИЦА В БРАУЗЕРЕ



ИСХОДНЫЙ HTML / CTRL+U

```
HTTP/1.1 401 Unauthorized

<html>
  <head>
    <meta charset="utf-8">
    [защитный JavaScript Qrator]
  </head>
  <body></body>
</html>

BeautifulSoup:
  a:        0
  article:  0
  h2:       0

После загрузки в браузере:
```

Обычный HTTP-запрос получает здесь 401 и пустое тело. Браузер выполняет скрипты сайта, и лента появляется. Мы просто открываем публичную страницу. Ни CAPTCHA, ни подмены отпечатка браузера, ни других обходов защиты в примере нет.

ЧАСТЬ III

Заголовки РБК в CSV, скриншот и PDF

```
import base64
import csv
import time
from pathlib import Path
from urllib.parse import urljoin
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

URL = "https://www.rbc.ru/short_news/"
t0 = time.perf_counter()

def progress(step, text):
    print(f"[{step}/8] {time.perf_counter() - t0:5.1f} c  {text}", flush=True)

opts = Options()
opts.add_argument("--window-size=1440,1000")   # headless НЕ включаем, окно должно быть видно

progress(1, "Запуск Chrome, окно откроется на экране")
driver = webdriver.Chrome(options=opts)
try:
    progress(2, "Открытие публичной ленты РБК")
    driver.get(URL)
```

ЧАСТЬ III

Заголовки РБК в CSV, скриншот и PDF

```
progress(3, "Ожидание элементов article")
cards = WebDriverWait(driver, 20).until(
    EC.presence_of_all_elements_located((By.CSS_SELECTOR, "article")))
progress(4, f"Найдено карточек: {len(cards)}")

rows = []
for card in cards[:30]:
    links = card.find_elements(By.CSS_SELECTOR, "a")
    text = " ".join(card.text.split())
    href = links[0].get_attribute("href") if links else None
    if text and href:
        rows.append({"text": text, "url": urljoin(URL, href)})

progress(5, f"Собрано уникальных строк: {len(rows)}")
with open("rbc_headlines.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=["text", "url"])
    writer.writeheader()
    writer.writerows(rows)

progress(6, "CSV записан, сохранение полного PNG")
shot = driver.execute_cdp_cmd("Page.captureScreenshot",
                              {"captureBeyondViewport": True, "fromSurface": True})
Path("rbc_news.png").write_bytes(base64.b64decode(shot["data"]))
```

ЧАСТЬ III

Заголовки РБК в CSV, скриншот и PDF

```
    progress(7, "PNG сохранён, формирование PDF")
    Path("rbc_news.pdf").write_bytes(base64.b64decode(driver.print_page()))
finally:
    driver.quit()

progress(8, "Готово, CSV, PNG и PDF сохранены")
print("\nРезультат")
print("заголовков:", len(rows))
print("файлы: rbc_headlines.csv, rbc_news.png, rbc_news.pdf")
for row in rows[:5]:
    print("-", row["text"][:95])
```

Это один заход на публичную страницу, без параллельных сессий и без обхода защиты. Прежде чем наращивать сбор, посмотрите условия использования сайта и допустимую частоту обращений.

ЧАСТЬ III

Живая демонстрация в настоящем Chrome

```
import csv
import re
import time
import requests
from pathlib import Path
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By

OUT = Path("демонстрация")
OUT.mkdir(exist_ok=True)

opts = Options()
opts.add_argument("--window-size=1280,900")   # headless НЕ включаем, окно должно быть видно

def scroll(driver, steps=6, step=400, pause=0.9):
    """Плавно прокручиваем страницу, чтобы за движением можно было следить."""
    for _ in range(steps):
        driver.execute_script(f"window.scrollBy({{top: {step}, behavior: 'smooth'}})")
        time.sleep(pause)

driver = webdriver.Chrome(options=opts)
try:
    driver.get("https://lenta.ru/")
    time.sleep(2)
```

ЧАСТЬ III

Живая демонстрация в настоящем Chrome

```
print("1. Chrome открыт, главная страница загружена", flush=True)

scroll(driver)
print("2. Пролистали главную", flush=True)

button = driver.find_element(By.XPATH, "//button[contains(., 'Новые материалы')]")
driver.execute_script("arguments[0].scrollIntoView({block: 'center', behavior: 'smooth'})", button)
time.sleep(1.5)
driver.execute_script("arguments[0].click()", button)
time.sleep(2.5)
print("3. Нажали кнопку «Новые материалы»", flush=True)

link = driver.find_element(By.CSS_SELECTOR, "a[href*='/rubrics/economics/']")
driver.execute_script("arguments[0].scrollIntoView({block: 'center', behavior: 'smooth'})", link)
time.sleep(1.5)
driver.execute_script("arguments[0].click()", link)
time.sleep(3)
print("4. Перешли в раздел «Экономика»:", driver.current_url, flush=True)

scroll(driver, steps=5)
print("5. Пролистали раздел", flush=True)

rows, seen = [], set()
for a in driver.find_elements(By.CSS_SELECTOR, "a[href^='/news/']"):
    href = a.get_attribute("href")
```

ЧАСТЬ III

Живая демонстрация в настоящем Chrome

```
        text = re.sub(r"\s*\d{2}:\d{2}$", "", " ".join(a.text.split()))
        if len(text) > 25 and href not in seen:
            seen.add(href)
            rows.append({"заголовок": text, "ссылка": href})

with open(OUT / "новости.csv", "w", newline="", encoding="utf-8-sig") as f:
    writer = csv.DictWriter(f, fieldnames=["заголовок", "ссылка"])
    writer.writeheader()
    writer.writerows(rows)
print(f"6. Собрано заголовков: {len(rows)}, записан новости.csv", flush=True)
for row in rows[:3]:
    print("   -", row["заголовок"][:70], flush=True)

first = driver.find_element(By.CSS_SELECTOR, "a[href^='/news/']")
driver.execute_script("arguments[0].scrollIntoView({block: 'center', behavior: 'smooth'})", first)
time.sleep(1.5)
driver.execute_script("arguments[0].click()", first)
time.sleep(3)
print("7. Открыли новость:", driver.title[:60], flush=True)

img = driver.find_element(By.CSS_SELECTOR, "figure img")
data = requests.get(img.get_attribute("src"), timeout=20).content
(OUT / "картинка.jpg").write_bytes(data)
print(f"8. Картинка скачана: {len(data) / 1024:.1f} КБ", flush=True)
```

ЧАСТЬ III

Живая демонстрация в настоящем Chrome

```
    driver.save_screenshot(str(OUT / "страница.png"))
    print("9. Скриншот сохранён", flush=True)
    time.sleep(2)
finally:
    driver.quit()

print("\nФайлы в папке", OUT.resolve())
for f in sorted(OUT.iterdir()):
    print(f"  {f.name:14} {f.stat().st_size / 1024:8.1f} КБ")
```

Прокрутка и клики намеренно замедлены, чтобы за ними можно было следить. Мы открываем публичную ленту как обычный читатель, robots.txt Ленты главную и раздел новостей не закрывает.

ЧАСТЬ IV

Playwright

Постоянное соединение с браузером, ожидания из коробки и доступ к сетевым запросам страницы.

ЧАСТЬ IV

Архитектура Playwright

- Playwright держит с браузером постоянное соединение, поэтому не тратит отдельный HTTP-вызов на каждое действие и видит события страницы.
- Browser context это отдельная сессия внутри одного браузера, со своими cookies, хранилищем и разрешениями.
- Установка занимает две команды, pip install playwright и playwright install chromium.

> Для последовательного сценария хватает sync_api. async_api нужен, когда сценарии идут параллельно или приложение уже работает асинхронно.

ЧАСТЬ IV

Что Playwright проверяет перед действием

- Элемент есть в дереве и виден.
- Он перестал двигаться, анимация закончилась.
- Он принимает события и ничем не закрыт.
- Он включён, а поле доступно для ввода.

> Эти проверки снимают с вас синхронизацию с загрузкой. Паузу вы ставите там, где сами задаёте темп, а смысловые условия пишете через expect.

ЧАСТЬ IV

Базовый сценарий целиком

```
from playwright.sync_api import sync_playwright, expect

t0 = time.perf_counter()
with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)
    ctx  = browser.new_context(locale="ru-RU")
    page = ctx.new_page()

    page.goto("https://quotes.toscrape.com/js/", wait_until="domcontentloaded")
    page.locator("div.quote").first.wait_for()      # ожидаем первую карточку

    for q in page.locator("div.quote").all()[:3]:
        print("  ", q.locator("small.author").inner_text())
    n_pw = page.locator("div.quote").count()
    print("всего цитат:", n_pw)
    browser.close()

t_pw = time.perf_counter() - t0
print(f"Playwright: {t_pw:.2f} c")
```

ЧАСТЬ IV

Ожидание внутри expect

```
with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)
    page = browser.new_page()
    page.goto("https://quotes.toscrape.com/js/")

    expect(page.locator("div.quote")).to_have_count(10)
    expect(page.locator("div.quote").first).to_be_visible()
    print("обе проверки прошли")

    browser.close()
```

ЧАСТЬ IV

JSON через перехват сети

```
collected = []

with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)
    page = browser.new_page()

    def on_response(resp):
        if "/api/quotes" in resp.url and resp.status == 200:
            try:
                collected.extend(resp.json()["quotes"])
            except Exception:
                pass                        # не всякий ответ — JSON

    page.on("response", on_response)        # подписка ДО goto
    page.goto("https://quotes.toscrape.com/scroll")

    prev, stable = -1, 0
    for _ in range(30):
        page.mouse.wheel(0, 15000)
        page.wait_for_timeout(500)
        if len(collected) == prev:
            stable += 1
            if stable >= 2:
                break
        else:
```

ЧАСТЬ IV

JSON через перехват сети

```
            stable = 0
        prev = len(collected)
    browser.close()

print("получено объектов из сети:", len(collected))
print("пример:", collected[0]["author"]["name"], "|", collected[0]["text"][:60])
```

Браузер сам проходит авторизацию и собирает запрос. Нам остаётся забрать готовый JSON из ответа, не вытаскивая данные обратно из вёрстки.

ЧАСТЬ IV

Блокировка лишних ресурсов

```
import statistics

BLOCK = {"image", "media", "font"}

def measure(block: bool, n: int = 3):
    """Медиана по n прогонам снижает влияние холодного старта."""
    times, reqs = [], []
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=True)
        for _ in range(n):
            ctx = browser.new_context()
            if block:
                ctx.route("**/*", lambda route:
                          route.abort() if route.request.resource_type in BLOCK
                          else route.continue_())
            page = ctx.new_page()
            seen = {"n": 0}
            page.on("response", lambda r: seen.__setitem__("n", seen["n"] + 1))
            t = time.perf_counter()
            page.goto("https://books.toscrape.com/", wait_until="load")
            times.append(time.perf_counter() - t)
            reqs.append(seen["n"])
            ctx.close()
        browser.close()
    return statistics.median(times), statistics.median(reqs)
```

ЧАСТЬ IV

Блокировка лишних ресурсов

```
t_slow, n_slow = measure(False)
t_fast, n_fast = measure(True)
print(f"без блокировки: {t_slow:.2f} с, запросов {n_slow:.0f}")
print(f"с блокировкой : {t_fast:.2f} с, запросов {n_fast:.0f}")
print(f"запросов меньше в {n_slow / n_fast:.1f} раза, время x{t_slow / t_fast:.2f}")
```

> Число запросов падает предсказуемо, а время зависит от страницы, сети и состояния кэша. Смотрите в первую очередь на число запросов.

ЧАСТЬ IV

Повторное использование сессии

```
with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)

    ctx = browser.new_context()
    page = ctx.new_page()
    page.goto("https://quotes.toscrape.com/login")
    page.fill("#username", "test")
    page.fill("#password", "test")
    page.click("input[type='submit']")
    page.wait_for_url("https://quotes.toscrape.com/")
    ctx.storage_state(path="state.json")
    ctx.close()

    ctx2 = browser.new_context(storage_state="state.json")
    page2 = ctx2.new_page()
    page2.goto("https://quotes.toscrape.com/")
    print("сессия восстановлена:", page2.locator("a[href='/logout']").count() > 0)
    browser.close()
```

ЧАСТЬ IV

Запись действий и разбор ошибок

- Команда playwright codegen <url> открывает браузер, записывает ваши действия и сразу показывает готовый код с локаторами.
- Trace Viewer сохраняет шаги сценария, снимки дерева и сетевые запросы. Включаем через ctx.tracing.start(...), выключаем ctx.tracing.stop(path="trace.zip"), смотрим командой playwright show-trace trace.zip.

> Трасса особенно помогает с ошибками, которые повторяются через раз и не воспроизводятся руками.

ЧАСТЬ V

Надёжность и ограничения

Скорость, воспроизводимость, правовые рамки и нагрузка, которую мы создаём на чужой сервер.

ЧАСТЬ V

Четыре правила устойчивого сбора

Сохраняйте сырые ответы

HTML и JSON кладите на диск как есть, а разбор делайте отдельным шагом.

Ставьте контрольные точки

После сбоя работа продолжится с последней сохранённой записи, а не с начала.

Повторяйте с растущей паузой

1, 2, 4, 8 секунд плюс ограничение на число попыток.

Отличайте пустоту от сбоя

Ноль элементов и упавший запрос это разные ситуации, и обрабатывать их надо по-разному.

ЧАСТЬ V

Правовые и этические рамки

- Перед запуском посмотрите robots.txt и условия использования сайта.
- Ограничивайте частоту запросов и число одновременных сессий.
- Страница открыта всем, но это не отменяет правил работы с персональными данными.
- В User-Agent укажите, кто вы. Для исследовательского сбора полезно оставить контакт.
- Кэшируйте. То, что уже скачано, второй раз запрашивать незачем.

> Проверенные условия использования, фактическую частоту запросов и число параллельных сессий.

ЧАСТЬ V

Сравнение Selenium и Playwright

	Selenium	Playwright
Управление браузером	стандарт W3C WebDriver	постоянное двустороннее соединение
Ожидания	пишем сами через WebDriverWait	встроены в действия
Накладные расходы на команду	обычно выше	обычно ниже
Работа с сетью	WebDriver BiDi или CDP	**route, request, response**
Изоляция сессий	отдельные сессии или профили	**browser contexts**
Запись действий	Selenium IDE	**codegen**
Разбор ошибок	логи, скриншоты, BiDi	**Trace Viewer и сетевые события**
Возраст	с 2004 года	с 2020 года
Браузеры	широкий охват через драйверы	Chromium, Firefox, WebKit

> На новом проекте Playwright обычно требует меньше кода на синхронизацию и разбор ошибок. Selenium выигрывает там, где уже есть инфраструктура WebDriver или нужен более широкий набор браузеров и конфигураций.

ЧАСТЬ V

Прямой запрос против браузера

```
need = {"t_api": "«Прямой запрос к API»", "n_pw": "«Базовый сценарий целиком»"}
missing = [name for var, name in need.items() if var not in globals()]

if missing:
    print("Сначала выполните код на слайдах:")
    for name in missing:
        print("  ", name)
else:
    print(f"{'прямой запрос к API':24} {t_api:6.2f} с   {len(quotes_api):3} цитат")
    print(f"{'браузер, Playwright':24} {t_pw:6.2f} с   {n_pw:3} цитат")
    print("\nсценарии собрали разное, поэтому считаем цену одной цитаты")
    print(f"  прямой запрос {t_api / len(quotes_api):.3f} с")
    print(f"  браузер       {t_pw / n_pw:.3f} с")
```

ЧАСТЬ VI

Нагрузочные проверки с Playwright

Небольшая нагрузка через настоящий браузер. Проверяем сценарий пользователя, время и ошибки.

ЧАСТЬ VI

Что показывает такой тест

- Проверяется весь путь пользователя, от загрузки страницы и работы скриптов до действий с интерфейсом и появления результата.
- Несколько одновременных сессий вскрывают тайм-ауты, ошибки отрисовки, зависшие запросы и рост времени.
- Реальный масштаб это единицы или десятки виртуальных пользователей. Для сотен и тысяч берут специальные генераторы нагрузки.

> Каждый пользователь здесь это настоящий браузер и сотни мегабайт памяти. Нагрузку на API давайте через k6, Locust или JMeter, а Playwright оставьте для ключевых сценариев интерфейса.

ЧАСТЬ VI

Нагрузочный тест на локальном стенде

```
import asyncio
import statistics
import time
from playwright.async_api import async_playwright

TARGET = "http://127.0.0.1:8777/"
VUS = 4                 # параллельные виртуальные пользователи
ITERATIONS = 3          # сценария на каждого пользователя

async def load_test():
    durations, errors = [], []
    completed = 0
    total = VUS * ITERATIONS
    print(f"Старт: {VUS} пользователей × {ITERATIONS} итерации = {total} сценариев")

    async with async_playwright() as p:
        browser = await p.chromium.launch(headless=True)

        async def virtual_user(user_id):
            nonlocal completed
            context = await browser.new_context()
            page = await context.new_page()

            for iteration in range(1, ITERATIONS + 1):
                started = time.perf_counter()
```

ЧАСТЬ VI

Нагрузочный тест на локальном стенде

```
            try:
                response = await page.goto(
                    f"{TARGET}?vu={user_id}&run={iteration}",
                    wait_until="domcontentloaded",
                    timeout=15_000)
                await page.locator("#stage").wait_for(state="visible")
                if response is None or not response.ok:
                    raise RuntimeError(f"HTTP {response.status if response else 'нет ответа'}")
                elapsed = time.perf_counter() - started
                durations.append(elapsed)
                status = "OK"
            except Exception as exc:
                elapsed = time.perf_counter() - started
                errors.append(type(exc).__name__)
                status = "ERROR"

            completed += 1
            print(f"[{completed:02d}/{total}] VU {user_id:02d}  "
                  f"итерация {iteration}  {elapsed:5.2f} с  {status}", flush=True)

        await context.close()

    await asyncio.gather(*(virtual_user(i) for i in range(1, VUS + 1)))
    await browser.close()
```

ЧАСТЬ VI

Нагрузочный тест на локальном стенде

```
    if durations:
        p95 = (statistics.quantiles(durations, n=100, method="inclusive")[94]
               if len(durations) > 1 else durations[0])
        print("\nИтоги")
        print(f"успешно: {len(durations)}/{total}")
        print(f"ошибок : {len(errors)}/{total}")
        print(f"median : {statistics.median(durations):.3f} c")
        print(f"p95    : {p95:.3f} c")
    if errors:
        print("типы ошибок:", {e: errors.count(e) for e in set(errors)})

asyncio.run(load_test())
```

> Код обращается только к локальному стенду. Внешний сервис так можно проверять лишь с разрешения владельца и на заранее согласованной нагрузке.

ЧАСТЬ VI

Playwright и специализированные инструменты

Задача	Playwright	k6 или Locust по HTTP
Настоящий браузер и JavaScript	полноценная страница	**нет**
Клики и внешний вид	есть	**не проверяются**
Десятки сессий браузера	можно, если следить за ресурсами	не их режим
Сотни и тысячи пользователей	слишком дорого	основной сценарий
Нагрузка на API	можно, но избыточно	основной сценарий
Разбор ошибок	трасса, скриншоты, сетевые события	метрики и логи

> Основную нагрузку создаём на уровне протокола. Параллельно несколько сценариев Playwright проверяют, что человек по-прежнему может пройти ключевой путь в интерфейсе.

Вопросы

Анна Иванова / Летняя школа 2026, трек DM