# Ćwiczenie 8

# GitHub i Gitea
# Automatyzacja projektów

## 8.1. Rozpoczęcie pracy i konto GitHub

1. Otworzyć przeglądarkę i przejść na stronę `https://github.com/`. Przygotować login własnego konta do dalszych zadań.
2. Zalogować się do `https://github.com/` albo utworzyć konto, potwierdzić adres e-mail i zapisać własny login (dalej `LOGIN`). W ustawieniach *Password and authentication* skonfigurować uwierzytelnianie dwuskładnikowe (2FA) i zabezpieczyć kody odzyskiwania. W *Emails* wybrać adres do podpisywania commitów, także adres `noreply`, jeśli adres prywatny nie powinien być ujawniany.
3. Sprawdzić ustawienia Git (z ćwiczenia 6) i aktualne repozytoria:

```
git --version
git config --global user.name
git config --global user.email
git remote -v
```

   Ostatnie polecenie wykonać w jednym z wcześniejszych repozytoriów Git.

4. Wykorzystać wiadomości z ćwiczenia 7: wygenerować **osobny** klucz GitHub (albo użyć istniejącego klucza użytkownika, jeśli prowadzący dopuszcza):

```
ssh-keygen -t ed25519 -f ~/.ssh/ndp_github_ed25519 \
    -C "ndp-github"
cat ~/.ssh/ndp_github_ed25519.pub
```

   W serwisie GitHub otworzyć *Settings* → *SSH and GPG keys* → *New SSH key* i dodać **tylko plik publiczny**. Nie dodawać klucza prywatnego.

5. Do `~/.ssh/config` dopisać odpowiedni wpis, nie nadpisując aliasu serwera

```
Host github.com
    HostName github.com
    User git
    IdentityFile ~/.ssh/ndp_github_ed25519
    IdentitiesOnly yes
```

Wykonać `chmod 600 ~/.ssh/config` i `ssh -T git@github.com`. Przed zaakceptowaniem klucza hosta sprawdzić jego oficjalny odcisk w dokumentacji GitHub. Komunikat *successfully authenticated* nie oznacza dostępu do powłoki GitHub; zakończenie testu z kodem 1 może być prawidłowe.

## 8.2. Publiczne repozytorium i przeniesienie projektu

6. Utworzyć repozytorium `ndp-INDEKS` przez *New repository*; zamiast `INDEKS` wpisać własny numer indeksu. Wybrać *Public*, zainicjalizować `README.md`, `.gitignore` (szablon Bash albo Python, jeżeli dostępny) i licencję `MIT`. Numer indeksu w nazwie repozytorium i URL strony będzie publiczny. Można uzgodnić z prowadzącym inny jednoznaczny identyfikator. Widoczność publiczna nie oznacza zgody na publikację sekretów ani danych osób trzecich.
7. Otworzyć nowe repozytorium i sklonować je za pomocą adresu SSH, a następnie sprawdzić `origin`:

```
cd ~
git clone git@github.com:LOGIN/ndp-INDEKS.git
cd ndp-INDEKS
git remote -v
git status
```

   Zastąpić `LOGIN` i `INDEKS` rzeczywistymi wartościami. Porównać zapis `git@github.com:...` z adresem serwera Git z ćwiczenia 6.
8. Utworzyć katalog `scripts/` i skopiować do niego `deploy.sh` z ćwiczenia 7. **Przed publikacją** przejrzeć skrypt i usunąć wszelkie lokalnie wpisane adresy, hasła, tokeny i nazwy użytkowników; pozostawić wyłącznie parametryzację zmiennymi środowiskowymi. Nie umieszczamy tam plików `~/.ssh/`, `result.txt`, ani danych serwera.

```
mkdir -p scripts docs
cp ~/cw7/starter/deploy.sh scripts/deploy.sh
bash -n scripts/deploy.sh
```

   Jeśli `deploy.sh` wymaga modyfikacji, zadbać, aby bez podanych zmiennych wyświetlał czytelny błąd, a nie próbował łączyć się z cudzym serwerem.
9. W katalogu `scripts/` utworzyć niewielki test lokalny `check.sh`: Przykład skryptu testowego jest dołączony do PDF: `check.sh`. Skopiować go do `scripts/check.sh`. Sprawdza on składnię i odmowę uruchomienia bez wymaganych zmiennych, ale jeszcze bez nawiązywania połączenia z serwerem. Zapisać plik i uruchomić `bash scripts/check.sh`.
   **Automatyczne zakończenia wierszy LF/CRLF.** W katalogu głównym repozytorium utworzyć `.gitattributes`. Git będzie zapisywał wskazane pliki tekstowe jako LF w repozytorium, również gdy edytor w Windows używa CRLF. Pliki binarne pozostają niezmienione:

```
cat > .gitattributes <<'EOF'
*.sh text eol=lf
*.yml text eol=lf
*.yaml text eol=lf
*.md text eol=lf
*.pdf binary
*.png binary
*.jpg binary
```

```
EOF
git add .gitattributes scripts/
git add --renormalize .
git ls-files --eol scripts/
bash scripts/check.sh
```

Sprawdzić, czy dla skryptów Git pokazuje `i/lf` i `w/lf`. Jeżeli pojawia się `w/crlf`, ustawić LF w edytorze i zapisać plik ponownie; `git add --renormalize .` normalizuje indeks Git, ale nie musi zmieniać bajtów pliku roboczego. W Git Bash można sprawdzić CRLF bez modyfikacji danych:

```
python -c "from pathlib import Path; \
p=Path('scripts/deploy.sh'); \
print('CRLF' if b'\r\n' in p.read_bytes() else 'no CRLF')"
```

Nie stosować zamiany końców wierszy do PDF, grafik, archiwów, kluczy i innych plików binarnych. Zatwierdzić zmianę:

```
git commit -m "Add SSH example and LF policy"
git push origin main
```

## 8.3. Dokumentacja i pliki organizacyjne

10. Rozbudować `README.md`: opisać cel projektu, wymagania `bash`, `ssh`, `scp`, sposób uruchomienia skryptu, jego zmienne środowiskowe, wymagania bezpieczeństwa oraz odnośnik do dokumentacji. Nie dodawać danych rzeczywistego serwera. Kompletny, zwięzły przykład: `README-example.md`. Dostosować opis do własnego projektu i wskazać sposób uruchomienia testów.
11. Utworzyć plik `AGENTS.md`, definiujący zakres projektu, zasady wprowadzania zmian i testowanie przez agenty programistyczne. Jest to dokument dla narzędzi obsługujących tę konwencję, a nie automatyczny mechanizm egzekwowania zasad. Pełny przykład pliku jako załącznik PDF: `AGENTS.md`. Zachować krótkie, jednoznaczne reguły walidacji i bezpieczeństwa.
12. Utworzyć `ROADMAP.md` z trzema etapami w postaci list kontrolnych: przygotowanie repozytorium, uruchomienie testów i publikacja strony, konfiguracja własnego hostingu. Zaznaczać wykonane zadania zgodnie z faktycznym postępem. Porównać plan z *Issues* i *Milestones*.
13. Utworzyć dokumenty `ARCHITECTURE.md`, `CHANGELOG.md`, `CONTRIBUTING.md` i `SECURITY.md`. W pierwszym opisać przepływ *lokalny skrypt* → *SSH* → *serwer* → *wynik*, w drugim datowane zmiany i sekcję `Unreleased`, w trzecim pracę przez gałęzie i PR, a w ostatnim bezpieczną metodę zgłaszania podatności (bez publikacji danych dostępowych).
14. Sprawdzić istniejącą licencję `LICENSE`: wyjaśnić, co przyznaje MIT i dlaczego publiczny kod bez jawnej licencji nie oznacza automatycznie zgody na dowolne ponowne wykorzystanie.
15. Do `.gitignore` dopisać:

```
.env
.env.*
*.log
*.key
*.pem
*.ppk
```

```
id_*
*_ed25519
*_rsa
result.txt
```

Wyjaśnić, że reguły `.gitignore` nie działają na już śledzone pliki (`git rm --cached` usuwa je tylko z indeksu) i dlaczego `.gitignore` nie usuwa sekretów już zapisanych w historii Git i dlaczego nawet przypadkowe zatwierdzenie klucza wymaga unieważnienia klucza.

16. Zatwierdzić dokumenty i wysłać je do publicznego repozytorium. Następnie otworzyć podgląd Markdown w edytorze i przeglądarce. Sprawdzić obecność `README.md`, `AGENTS.md`, `ROADMAP.md`, `ARCHITECTURE.md`, `CHANGELOG.md`, `CONTRIBUTING.md`, `SECURITY.md`, `LICENSE` i `.gitignore`.

## 8.4. Issues, gałęzie i Pull Requests

W modelu *GitHub Flow* każdą niezależną zmianę przygotowuje się na osobnej gałęzi utworzonej od `main`, a następnie zgłasza do przeglądu przez *Pull Request*. GitHub nie narzuca jednego schematu nazw, dlatego na zajęciach należy stosować `typ/krotki-opis`: `docs/opis-ssh` (dokumentacja), `feat/test-bash` (nowa funkcja), `fix/sciezka-deploy` (poprawka błędu) oraz `ci/walidacja-bash` (automatyzacja testów). Nazwy powinny być krótkie, zapisane małymi literami bez polskich znaków, ze słowami rozdzielonymi myślnikiem. Jedna gałąź dotyczy jednej logicznej zmiany; po zatwierdzeniu PR scala się ją do `main` i można usunąć. Jest to konwencja projektu, nie obowiązkowy standard GitHub.

17. Utworzyć dwa zgłoszenia w *Issues*: *Automated Bash validation* oraz *Publish project page*. Dodać odpowiednie etykiety, a następnie *Milestone* o nazwie `v1.0.0`. Porównać funkcję zgłoszenia, kamienia milowego i pliku `ROADMAP.md`.

18. Utworzyć gałąź roboczą i zmodyfikować dokumentację:

```
git switch -c docs/ssh-example
# uzupelnij README.md o przyklad uzycia
# (bez prawdziwych danych serwera)
git add README.md
git commit -m "Document SSH example"
git push -u origin docs/ssh-example
```

Utworzyć *Pull Request* do gałęzi `main`, dodać opis i powiązanie z zadaniem. Przejrzeć zakładkę *Files changed*.

19. Przeprowadzić przegląd zmian (samodzielnie albo w parze) i scalić PR po zaakceptowaniu. Na komputerze wykonać `git switch main` i `git pull --ff-only`. Zwrócić uwagę na zależność między lokalnym branchem, zdalną gałęzią a historią zatwierdzeń.

## 8.5. GitHub Actions i sekrety repozytorium

20. W katalogu `.github/workflows/` utworzyć `ci.yml`. Początkowo zastosować ręczne uruchamianie, aby kontrolować liczbę zadań: Wzorzec pliku workflow stanowi załącznik PDF: `ci.yml`. Stosuje minimalne uprawnienia, limit czasu i aktualną główną wersję akcji checkout. Umieścić go na gałęzi `main`, aby było dostępne ręczne uruchomienie `workflow_dispatch`. **Nie uruchamiać `deploy.sh` w GitHub Actions.** Test CI ma sprawdzać kod statycznie, bez prób połączenia z

serwerem uczelni.

21. W repozytorium (po zatwierdzeniu workflow na gałęzi domyślnej) otworzyć *Actions*, włączyć workflowy, jeśli jest to wymagane, i uruchomić *NDP Bash CI* przez *Run workflow*. Prześledzić status zadania, logi i kod zakończenia. Jeśli zajęcia odbywają się przy limicie bezpłatnych minut równym zero, nie uruchamiać *GitHub-hosted runners*; wykonać `bash scripts/check.sh` lokalnie i pokazać konfigurację workflowu bez uruchamiania.

22. W *Settings* → *Secrets and variables* → *Actions* utworzyć sekret o nazwie `DEMO_MESSAGE`, wpisując dowolny tekst demonstracyjny (niehasło i niedane serwera). Uzupełnić `ci.yml` o krok:

```
      - name: Check configured secret
        env:
          DEMO_MESSAGE: ${{ secrets.DEMO_MESSAGE }}
        shell: bash
        run: |
          test -n "$DEMO_MESSAGE"
          echo "Sekret jest ustawiony"
```

    Nie wypisywać wartości sekretu do logów. Wprowadzić zmianę w nowej gałęzi i utworzyć Pull Request; po akceptacji i scaleniu wykonać jeden kontrolowany przebieg ręczny, o ile pozwalają na to limity usługi.

23. Sprawdzić zakładkę *Actions* i porównać: plik YAML, zdarzenie `workflow_dispatch`, zadanie (*job*), krok (*step*), runner, zmienną środowiskową i sekret. Zastanowić się, czemu użytkownik z dostępem do zmian workflowu i wykonania kodu może stanowić zagrożenie dla sekretów repozytorium.

## 8.6. Publikacja GitHub Pages

24. Utworzyć `docs/index.html` z prostą, nieinteraktywną stroną opisującą projekt:

```
<!doctype html>
<html lang="pl">
<head><meta charset="utf-8"><title>NDP</title></head>
<body>
  <h1>Projekt NDP</h1>
  <p>GitHub Pages oraz zdalna praca przez SSH.</p>
</body>
</html>
```

    Dodać plik do Git, zatwierdzić i przesłać na `main`.

25. W *Settings* → *Pages* ustawić *Build and deployment* → *Deploy from a branch*, gałąź `main`, katalog `/docs`. Sprawdzić opublikowany adres `https://LOGIN.github.io/ndp-INDEKS/`, zastępując wartości własnymi danymi. Publikacja może wymagać krótkiego czasu; status sprawdzić w zakładce *Actions*.

26. Uaktualnić `README.md` o rzeczywisty adres strony i opisać, w jaki sposób aktualizacja pliku `docs/index.html` prowadzi do ponownej publikacji. W `ROADMAP.md` zaznaczyć ukończone etapy.

## 8.7. Lokalny runner GitHub Actions (osobne środowisko)

27. Zapoznać się z różnicą między *GitHub-hosted runner* i *self-hosted runner*. Runner wykonuje dowolny kod wskazany w workflowie. **Nie rejestrować go w publicznym repozytorium.** Dla tej części utworzyć **prywatne** repozytorium `ndp-runner-INDEKS` i użyć izolowanej, jednorazowej VM Linux bez kluczy osobistych.
28. W repozytorium otworzyć *Settings* → *Actions* → *Runners* → *New self-hosted runner*. Wybrać platformę i architekturę. W VM utworzyć katalog `~/actions-runner`, pobrać wydanie runnera, sprawdzić jego sumę SHA-256 i przeprowadzić rejestrację **zgodnie z aktualnymi poleceniami wygenerowanymi przez GitHub**.
29. Uruchomić runner w VM poleceniem `./run.sh`; w osobnym terminalu obserwować status. W **prywatnym** repozytorium utworzyć `.github/workflows/runner.yml`:

```
name: Local runner demo
on: workflow_dispatch
permissions:
  contents: read
jobs:
  demo:
    runs-on: [self-hosted, linux]
    timeout-minutes: 3
    steps:
      - name: Test isolated worker
        run: |
          echo "Runner lokalny OK"
          uname -a
```

    Ręcznie uruchomić zadanie i potwierdzić jego wykonanie na VM. Zapewnić, że tylko zaufane workflowy mogą korzystać z runnera.
30. Po demonstracji zatrzymać `./run.sh` poleceniem `Ctrl+C`, wyrejestrować runner za pomocą opcji *Remove* w ustawieniach prywatnego repozytorium i poleceń wskazanych przez GitHub, a następnie usunąć VM. Nie instalować usług runnera na serwerze współdzielonym i nie dopuszczać kodu z niezaufanych PR do stałego systemu.

## 8.8. Gitea na własnym serwerze Linux

Serwer Gitea może działać także na komputerze kolegi (zamiast własnej VM), za jego zgodą i przy dozwolonej komunikacji w sieci lokalnej. Właściciel instaluje i uruchamia Docker/Gitea na **swoim** komputerze zgodnie z poniższymi punktami; drugi student nie potrzebuje uprawnień administracyjnych. Korzystając z osobnego konta SSH utworzonego w ćwiczeniu 7, na **komputerze klienta** wykonać:

```
ssh -N -L 3000:127.0.0.1:3000 ndp-guest@ADRES_IP
```

Otworzyć `http://localhost:3000/`. Port 3000 w konfiguracji Gitea ma pozostać związany z `127.0.0.1` na komputerze serwera. Nie udostępniać Gitea w sieci publicznej. W dalszych poleceniach `HOST_VM` może oznaczać również alias komputera kolegi. Jeżeli klient używa już portu 3000, zastosować wariant z portem 13000 opisany niżej.

31. Wrócić do umiejętności z ćwiczenia 7. Sprawdzić architekturę i Git przez SSH:

```
ssh HOST_VM 'uname -m; git --version'
```

`HOST_VM` oznacza alias z `~/.ssh/config`; w wariancie w parach użyć `ndp-guest@ADRES_IP`. Obecność Dockera i Compose v2 weryfikuje **właściciel na serwerze**:

```
docker --version
docker compose version
```

Nie przydzielać kontu `ndp-guest` dostępu do Docker Engine ani grupy `docker`, która daje uprawnienia zbliżone do administratora.

32. Na wybranym serwerze utworzyć katalog `~/gitea-lab` oraz plik `compose.yaml`. Poniższy przykład przeznaczony jest wyłącznie do testów, a usługa HTTP jest nasłuchiwana na interfejsie loopback serwera: Przykładowy plik Compose dołączono do PDF: `compose.yaml`. Skopiować go na wybrany serwer, a następnie dostosować UID/GID i wersję obrazu do swojego środowiska. Plik ustawia `ROOT_URL` na lokalny port 3000; adres w przeglądarce i w konfiguratorze aplikacji musi być identyczny. Przed uruchomieniem sprawdzić aktualną stabilną wersję obrazu w oficjalnej dokumentacji i w razie potrzeby zmienić tag. UID/GID dostosować do konta właściciela na wybranym serwerze, zapewniając prawa do katalogu `data`. Ten wariant nie publikuje serwera w Internecie.

33. Uruchomić usługę i sprawdzić jej stan na wybranym serwerze (przez właściciela komputera):

```
cd ~/gitea-lab
docker compose config
docker compose up -d
docker compose ps
docker compose logs --tail=30
```

Gdy użytkownik nie ma prawa do komunikacji z Docker Engine, administrator VM musi nadać właściwe uprawnienia. Członkostwo w grupie `docker` jest w praktyce uprawnieniem uprzywilejowanym, dlatego nie przydzielać go na współdzielonym hoście.

34. Z komputera lokalnego zestawić tunel SSH, nawiązując do ćwiczenia 7:

```
ssh -N -o ExitOnForwardFailure=yes \
  -L 3000:127.0.0.1:3000 HOST_VM
```

W przeglądarce otworzyć `http://localhost:3000/`. Jeżeli port lokalny 3000 jest zajęty, zwolnić go lub użyć 13000 **oraz** zmienić `ROOT_URL` i *Gitea Base URL* na `http://localhost:13000/`. Po zmianie Compose odtworzyć kontener przez `docker compose up -d` i tunelować `-L 13000:127.0.0.1:3000`. Utrzymać tę samą wartość przy kolejnych uruchomieniach. W kreatorze instalacji wybrać *SQLite3* oraz utworzyć administratora. Nie używać hasła do serwera uczelnianego jako hasła do Gitea.

35. Po pierwszym logowaniu wyłączyć publiczną rejestrację, np. dodając do `compose.yaml` zmienną środowiskową `GITEA__service__DISABLE_REGISTRATION: "true"` i ponownie uruchamiając usługę za pomocą `docker compose up -d`. Zweryfikować w interfejsie, czy niezalogowany użytkownik nie może utworzyć konta. Włączyć 2FA dla administratora i przejrzeć ustawienia uprawnień.

36. Utworzyć prywatne repozytorium `ndp-gitea-INDEKS` i zainicjalizować je plikiem `README.md`. Skopiować lokalny URL HTTP (np. `http://localhost:3000/LOGIN/ndp.git`), sklonować je

**przez lokalny tunel SSH**, dodać `README.md` i wykonać `git push`. Nie stosować trwałego `credential.helper store` na stanowisku współdzielonym. Przy uwierzytelnianiu użyć menedżera poświadczeń lub tokenu o ograniczonym zakresie, jeśli wymaga tego konfiguracja; nigdy nie wpisywać tokenu w adresie repozytorium ani w pliku `README.md`.

37. Utworzyć zgłoszenie *Configure documentation*, gałąź roboczą i *Pull Request* w Gitea. Porównać funkcje *Issues* i *Pull Requests* z GitHub. Wyjaśnić, dlaczego Instancja Gitea nie wykonuje sama zadań: jej Gitea Actions wymaga włączonej funkcji oraz osobno skonfigurowanego Gitea Runner.

38. Skonfigurować Gitea pod własną domeną **wyłącznie na VM przeznaczonej do publikacji** (jeśli ma być dostępna z Internetu). Przygotować rekord DNS `git.example.org`, działające porty 80/443 oraz serwer reverse proxy, np. Caddy. Ustawić w `compose.yaml` wartości `GITEA__server__DOMAIN: "git.example.org"` oraz `GITEA__server__ROOT_URL: "https://git.example.org/"`. Na własnym serwerze z zainstalowanym Caddy dopisać do `/etc/caddy/Caddyfile`:

```
git.example.org {
    reverse_proxy 127.0.0.1:3000
}
```

Po weryfikacji DNS i składni Caddy użyć `sudo systemctl reload caddy` oraz `docker compose up -d` w katalogu Gitea. Sprawdzić certyfikat HTTPS i zgodność adresów generowanych przez aplikację. Nie udostępniać publicznie portu 3000; w przypadku braku własnej domeny pozostać przy tunelu SSH, nie zastępując HTTPS niezabezpieczonym HTTP w Internecie. Szczegóły: `https://docs.gitea.com/administration/reverse-proxies/`.

39. Wykonać **testową kopię zapasową** środowiska SQLite po zatrzymaniu usługi, zachowując dane i konfigurację:

```
cd ~/gitea-lab
docker compose down
umask 077
sudo tar -czf "$HOME/gitea-backup.tgz" data/
sudo chown "$(id -u):$(id -g)" "$HOME/gitea-backup.tgz"
tar -tzf "$HOME/gitea-backup.tgz" | head
docker compose up -d
```

Archiwum przechowywać poza publicznym repozytorium. Przemyśl, sposób okresowego odtwarzania kopii i aktualizacji obrazu.

## 8.9. Zadania dodatkowe

40. **Gitea Actions i lokalny runner.** Na oddzielnej, izolowanej VM skonfigurować *Gitea Runner* na podstawie `https://docs.gitea.com/runner/`. Włączyć Actions dla **prywatnego** repozytorium, pobrać oficjalny runner, zweryfikować wydanie i zarejestrować go tokenem przypisanym do tego repozytorium. Uruchomić ręcznie wyzwalany workflow `workflow_dispatch` wypisujący `Gitea Runner OK`; nie montować gniazda Docker z hosta produkcyjnego i nie dopuszczać niezaufanych workflowów. Po sprawdzeniu wyrejestrować runner i wyłączyć VM. Porównać mechanizmy Gitea Runner i GitHub Actions runner.

41. **Mirror GitHub do Gitea.** Skorzystać z funkcji *New Migration* na serwerze Gitea i utworzyć **nowe** prywatne repozytorium lustrzane własnego publicznego repozytorium GitHub z opcją

*This repository will be a mirror*. Ustawić jednokierunkowe pobieranie zmian *GitHub* → *Gitea*, wykonać nowy commit w repozytorium źródłowym, ręcznie zsynchronizować lustro i porównać SHA zatwierdzeń. W raporcie z zadania wymienić elementy, których samo mirrorowanie nie obejmuje, np. sekrety, uruchomienia CI i konfigurację GitHub Pages. Nie uruchamiać *push mirror* w kierunku źródła.

## 8.10. Zakończenie ćwiczenia

42. Przedstawić adres publicznego repozytorium GitHub i strony GitHub Pages, historię zatwierdzeń, Pull Request, pliki `README.md`, `AGENTS.md`, `ROADMAP.md`, `ARCHITECTURE.md`, `CHANGELOG.md`, `CONTRIBUTING.md`, `SECURITY.md`, `LICENSE`, `.gitignore` oraz wynik walidacji `bash scripts/check.sh`. Jeżeli dostępne były darmowe wykonania CI, pokazać również zakończony workflow. Tylko w przypadku realizacji rozszerzenia na własnej VM zademonstrować Gitea i testową kopię zapasową oraz usunąć tymczasowe środowiska runnera.