ChatGPT

# Projekt i planowanie architektury

Projekt warto zacząć od **zdefiniowania wymagań** i podziału na etapy. Aplikacja wypożyczalni filmów powinna mieć co najmniej dwie **mikrousługi** (np. serwis rejestracji użytkowników oraz serwis obsługujący katalog i wypożyczenia filmów) komunikujące się ze sobą przez API lub kolejki wiadomości. Klasyczne podejście *monolityczne* staje się skomplikowane w rozwoju i wdrażaniu, dlatego **cloud-native** rekomenduje dzielenie funkcji na niezależne moduły (mikrousługi) [1]. Każdą usługę można rozwijać, testować i wdrażać osobno, co ułatwia szybkie aktualizacje (por. wieża Jengi). W Scrumie warto spisać backlog (w Jira) z zadaniami: np. "rejestracja użytkownika", "CRUD filmów", "uwierzytelnianie". Na początek opracuj ogólny **schemat architektury** – które elementy są niezależne, jakie API się udostępniają, jakimi danymi się wymieniają. Planując architekturę pamiętaj, że każda mikrousługa powinna mieć swoje własne środowisko uruchomieniowe (może to być osobna wirtualna maszyna lub kontener) oraz dostęp do współdzielonych zasobów (np. bazy danych czy kolejki).

## Tworzenie aplikacji w chmurze (Python + MongoDB)

Kod aplikacji piszemy tak samo jak zwykłą aplikację webową – np. w **Pythonie** przy użyciu frameworka (Flask, FastAPI, Django itp.). Możecie więc pracować lokalnie na komputerach, pisać kod w wybranym języku i upewnić się, że działa (testując operacje CRUD na MongoDB). Następnie przygotowujemy środowisko, by przenieść to do chmury. Chmurze natywne oznacza, że projekt tworzy się z myślą o **skorzystaniu z usług chmurowych** (np. skalowalne bazy, gotowe kontenery), ale samo kodowanie odbywa się normalnie – to raczej sposób wdrażania i uruchamiania jest cloud-native. Dzięki konteneryzacji możemy spakować aplikację i uruchomić ją *tak samo wszędzie – na laptopie, w CI/CD i w chmurze* [2]. Kod z Pythona i konfigurację zależności pakuje się do obrazu Docker. Ten obraz można potem przechowywać w rejestrze (np. Azure Container Registry) i wdrażać w Azure. Można też skorzystać z platformy Azure App Service lub Azure Container Instances, które pozwalają uruchomić gotowe kontenery w chmurze bez zarządzania VM. Warto także ustawić środowisko testowe w chmurze (np. niewielka instancja Ubuntu z Dockerem), żeby weryfikować działanie serwisów już na wczesnym etapie.

## Mikrousługi i konteneryzacja

*Przykładowa architektura mikrousług w chmurze Azure.* Aplikacja cloud-native dzieli się na **mikrousługi**, każda realizująca osobną funkcję biznesową (np. rejestracja, katalog filmów, wypożyczenie). Dzięki temu można każdą usługę rozwijać i wdrażać oddzielnie – zmiany jednej nie psują innych [1]. Mikrousługi komunikują się zazwyczaj przez HTTP/REST lub kolejki wiadomości (np. RabbitMQ/Azure Service Bus). Aby zapewnić przenośność i spójne środowisko, warto każdy serwis uruchomić w **kontenerze Docker**. Kontenery zawierają wszystko, czego aplikacja potrzebuje – kod, biblioteki i środowisko uruchomieniowe – więc działają identycznie niezależnie od miejsca uruchomienia [2]. Obraz kontenera tworzysz np. komendą `docker build`, a potem umieszczasz go w rejestrze (np. Azure Container Registry) [3]. Azure umożliwia hostowanie kontenerów na różne sposoby – od **Azure Kubernetes Service (AKS)**, przez **Azure Container Instances (ACI)**, po **App Service dla kontenerów** [4]. Na początek można użyć prostszego rozwiązania: np. uruchomienie kilku kontenerów na jednej maszynie lub w ACI. Ważne jest też skonfigurowanie wewnętrznego łącza między mikrousługami (np. prywatna sieć wirtualna w Azure albo Docker Compose dla lokalnego testu).

## CI/CD z Jenkinsem

Zarówno kod aplikacji, jak i konfiguracja infrastruktury warto zautomatyzować za pomocą **pipeline CI/CD**. Jenkins to popularne narzędzie do tego celu. Pipeline automatyzuje budowanie, testowanie i wdrażanie aplikacji przy każdej zmianie w kodzie. W skrócie: po zatwierdzeniu kodu Jenkins buduje obraz (np. przez `docker build`), uruchamia testy, a jeśli wszystko się powiedzie, wypycha obraz do rejestru i wdraża go na serwery. CI/CD pozwala skupiać się na kodzie, bo maszyna wykonuje rutynowe zadania (budowę, testy, deploy) [5]. Na przykład można w Jenkinsie ustawić potok: (1) **Checkout** – pobranie kodu z repozytorium Git; (2) **Test** – uruchomienie testów jednostkowych (np. `pytest` w Pythonie); (3) **Build** – zbudowanie obrazu Docker z aplikacją; (4) **Deploy** – wypchnięcie obrazu do Azure Container Registry i wdrożenie kontenera na docelowym środowisku (VM lub ACI). Dzięki Jenkinsowi możemy też zautomatyzować provisioning – np. uruchomić skrypty Terraform/Ansible na etapie pipeline'u. Ustawiając CI, zadbaj o **integrację testów jednostkowych i e2e**: jeśli testy (np. przy użyciu pytest czy Selenium) zakończą się niepowodzeniem, proces zostaje przerwany. Takie podejście (CI/CD) znacząco skraca cykle wprowadzania zmian i pozwala szybko wychwycić błędy [6] [5]. Jenkins umożliwia także bezproblemową integrację z Azure – np. przez wtyczki Azure CLI lub Azure App Service, co upraszcza push do rejestru ACR lub wdrożenie aplikacji webowych na Azure [7] [8].

## Infrastruktura: Terraform/Ansible i Azure

Do tworzenia infrastruktury (VM, sieci, zasobów bazy danych itp.) warto użyć narzędzi IaC, np. **Terraform**. Terraform umożliwia zdefiniowanie całej infrastruktury w plikach konfiguracyjnych (HCL) i automatyczne jej wdrożenie w Azure [9]. Możemy więc opisać: *grupę zasobów*, maszyny wirtualne z Dockerem, reguły sieciowe itp., a Terraform zadba o utworzenie tych zasobów. Analogicznie **Ansible** (playbook) może konfigurować maszyny (instalacja Dockera, pobranie obrazów). Dzięki temu jeden skrypt (pipeline) może zbudować środowisko testowe/produkcyjne w Azure. Np. Terraform tworzy VM z systemem Linux, Ansible instaluje i konfiguruje tam Docker, a Jenkins wypycha tam kontenery. Alternatywnie można od razu użyć usług chmurowych zamiast zarządzać VM – np. **Azure Container Instances** (uruchamia pojedyncze kontenery) lub **Azure App Service** (dla kontenerów webowych). Ważne, by proces tworzenia środowiska również podlegał kontroli wersji i pipeline'owi (Infrastructure as Code). Dzięki narzędziom DevOps obniżamy manualną pracę i możliwa jest powtarzalna, spójna konfiguracja w różnych środowiskach.

## Bezpieczeństwo danych

W projekcie należy wdrożyć bezpieczny transfer i składowanie danych. **HTTPS/TLS:** wszystkie endpointy API wrażliwe na dane (logowanie, rejestracja) muszą działać przez HTTPS. Azure oferuje certyfikaty SSL/TLS (np. darmowe Let's Encrypt, **Key Vault** do przechowywania certyfikatów). Protokół HTTPS szyfruje przesyłane dane, zapewniając integralność i uwierzytelnienie serwera [10]. Poza tym warto zwrócić uwagę na **szyfrowanie danych „at rest"** – np. MongoDB w Azure (Cosmos DB) może mieć włączone szyfrowanie dysków. Według praktyk bezpieczeństwa powinno się szyfrować dane zarówno w tranzycie, jak i w stanie spoczynku [11] [12]. **Hashowanie haseł:** nigdy nie przechowujemy haseł w postaci niezaszyfrowanej. Zamiast tego stosujemy hashing (jednokierunkowa funkcja), najlepiej z solą i mocnym algorytmem (bcrypt, Argon2) [13]. Dzięki temu nawet jeśli baza zostanie skompromitowana, atakujący nie odzyska prostych haseł. OWASP podkreśla, że hasła **zawsze powinny być hashowane** – hashowanie (nie odwracalne), w odróżnieniu od szyfrowania, uniemożliwia odzyskanie hasła z bazy danych [13]. Dodatkowo stosujemy solidne **uwierzytelnianie** (np. JWT w nagłówkach) i kontrolę dostępu do API. Zadbajcie także o regularne aktualizacje serwerów i zależności – niezaktualizowane biblioteki to częsta przyczyna luk bezpieczeństwa.

## Testy (jednostkowe i end-to-end)

W trakcie developmentu równolegle twórz testy. **Testy jednostkowe** (np. `pytest` dla Pythona) sprawdzą pojedyncze funkcje i endpointy. **Testy end-to-end (E2E)** symulują cały przepływ – np. rejestracja użytkownika i wypożyczenie filmu z front-endu lub przy użyciu REST. Testy E2E można realizować narzędziami takimi jak Selenium czy Postman/Newman. Wszystkie testy warto integrować z pipeline CI: Jenkins powinien uruchamiać testy przed każdym wdrożeniem. Dzięki temu błędy wykryjemy szybko, a zmiany w kodzie nie przypadkowo nie popsują istniejącej logiki. To standardowa praktyka DevOps – CI automatyzuje proces testowania, aby programista szybko otrzymywał feedback i minimalizował ryzyko błędów w produkcji [6] [5].

## Pierwsze kroki – co robić najpierw

1. **Sporządź plan projektu:** Na backlogu w Jira utwórz zadania (user stories) dla kluczowych funkcji (rejestracja, logowanie, dodawanie filmów, wypożyczanie itp.). Określ mikrousługi i zależności między nimi.
2. **Skonfiguruj repozytorium Git:** Utwórz repozytorium (GitHub/Bitbucket) i podziel projekt na moduły. Ustal strukturę folderów (np. foldery dla każdej mikrousługi).
3. **Utwórz szkielet aplikacji:** Zacznij od najprostszej funkcjonalności – np. mikrousługi użytkowników. Wybierz framework (np. Flask/FastAPI) i utwórz proste API do rejestracji/logowania. Przetestuj lokalnie z MongoDB (może to być lokalny Docker z MongoDB albo instancja Cosmos DB na Azure).
4. **Dockerfile i kontener:** Stwórz Dockerfile dla tej usługi (bazowany na Python:3.x). Upewnij się, że aplikacja działa w kontenerze (`docker run`). Obraz wypchnij do Azure Container Registry lub Docker Hub.
5. **Dodaj CI:** Skonfiguruj Jenkins (może być uruchomiony jako VM w Azure lub kontener) i napisz prosty Pipeline: checkout, budowanie obrazu, uruchamianie testów. Na początku przetestuj pipeline na tej jednej usłudze.
6. **Dalsze mikrousługi:** Powtórz kroki 3–5 dla drugiej usługi (np. serwisu filmów). Ustal sposób komunikacji między nimi (np. REST: użytkownik pobiera token, film serwis weryfikuje je).
7. **Provision infrastruktury:** Użyj Terraform/Ansible, aby zautomatyzować tworzenie zasobów w Azure. Zacznij od małego sandboxu (jedna VM z Dockerem lub ACI). W ramach pipeline'a Jenkins możesz odpalać komendy Terraform (np. `terraform apply`) przed wdrożeniem.
8. **Wdrażanie w Azure:** Połącz się z Azure CLI/portalem – wdrożenie powinno tworzyć DNS, certyfikaty TLS/SSL (App Service) i uruchamiać kontenery w wybranym środowisku. Sprawdź, czy adres jest dostępny przez HTTPS (np. https://TwojSerwis.azurewebsites.net).
9. **Zabezpiecz system:** Dodaj hash i sól do haseł w kodzie aplikacji (biblioteki bcrypt dla Pythona). Włącz HTTPS. Przetestuj, czy hasła w bazie to skróty, a nie plaintext.
10. **Pisz testy:** Twórz testy w miarę pisania funkcji. Dodaj testy e2e do pipeline'u Jenkins. Upewnij się, że każde nowe wydanie przechodzi wszystkie testy.

**Narzędzia:** W projekcie wykorzystacie *Azure* (chmura do hostowania VM/kontenerów), *Docker* (konteneryzacja mikrousług), *Jenkins* (CI/CD), *Terraform/Ansible* (za pomocą IaC przygotujecie VM), *MongoDB* (np. na Azure Cosmos DB z API Mongo lub własny kontener Mongo). Jira posłuży do zarządzania zadaniami. W Pythonie możecie użyć popularnych bibliotek (Flask, FastAPI, PyMongo, pytest). Wszystkie te technologie współpracują dobrze w ekosystemie DevOps – np. Docker dostarcza obrazy, które ACI/AKS uruchomi, a Jenkins w pełni zautomatyzuje budowę i wdrożenie [3] [9].

**Podsumowanie:** Pisząc aplikację w chmurze, koduje się ją jak zwykle (wybranym językiem na komputerze), a potem pakuje do kontenerów i wdraża na platformę (tu Azure). Konteneryzacja oznacza „zapakowanie" aplikacji i środowiska w przenośny obraz, który działa tak samo wszędzie [2] [3]. CI/CD

(Jenkins) pozwala automatyzować budowę, testowanie i wdrażanie tych kontenerów. Ważne jest również uwzględnienie bezpieczeństwa (HTTPS, hash haseł) oraz testów od samego początku. Najpierw skupcie się na core funkcjach i jednej mikrousłudze – dopiero gdy ta działa lokalnie i w kontenerze, rozbudowujcie projekt o kolejne usługi i warstwy infrastruktury.

**Źródła:** Opisane koncepcje opierają się na dokumentacji i praktykach DevOps i cloud-native [1] [2] [3] [5] [9] [13] [11]. Każdy krok procesu powinien być udokumentowany i zautomatyzowany w CI/CD, co zapewni spójne, powtarzalne wdrażanie.

---


[1] [6] Co to jest cloud native? | OVHcloud Polska
https://www.ovhcloud.com/pl/learn/what-is-cloud-native/

[2] Docker - podstawy konteneryzacji - UProgramisty
https://uprogramisty.pl/docker-podstawy-konteneryzacji/

[3] [4] Wdrażanie kontenerów na platformie Azure - .NET | Microsoft Learn
https://learn.microsoft.com/pl-pl/dotnet/architecture/cloud-native/deploy-containers-azure

[5] CI/CD for Python Application Using Jenkins and Docker
https://www.fosstechnix.com/ci-cd-for-python-application-using-jenkins/

[7] [8] CI/CD with Jenkins Pipeline and Azure
https://www.jenkins.io/blog/2017/08/10/kubernetes-with-pipeline-acs/

[9] Quickstart: Use Terraform to create a Linux VM - Azure Virtual Machines | Microsoft Learn
https://learn.microsoft.com/en-us/azure/virtual-machines/linux/quick-create-terraform

[10] [11] Metody szyfrowania danych w chmurze. Poradnik Orange
https://www.orange.pl/poradnik-dla-firm/cloud/jakie-sa-najnowsze-metody-szyfrowania-danych-w-chmurze/

[12] Bezpieczeństwo danych w chmurze - odpowiednio o nie dbasz?
https://focustelecom.pl/blog/bezpieczenstwo-danych-w-chmurze/

[13] Password Storage - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html