Clean architecture

Łukasz Andrzejewski

Definicje

- Reguły biznesowe definiują pojęcia i polityki niezbędne dla działania biznesu
- Proces biznesowy to seria powtarzalnych kroków, wykonywanych przez organizację, w celu uzyskania pożądanego efektu (celu biznesowego)
- Logika biznesowa to część aplikacji, odpowiedzialna za realizację przyjętych reguł biznesowych

Poprawna implementacja logiki biznesowej

- Opiera się o stosowanie praktyk, prowadzących do czystego kodu oraz czystej architektury
- Większość z nich daje się uogólnić do działań zapewniających niskie sprzężenie (low coupling) i wysoką spójność (high cohesion)

Kontrakt / protokół / interfejs / API

- Abstrakcja definiująca kontrakt miedzy współpracującymi elementami (obiektami, komponentami, modułami, usługami, mikroserwisami)
- Określa możliwe sposoby interakcji
- Ukrywa szczegóły implementacyjne

Inwersja kontroli (IoC)

- Odwrócenie sterowania wykonania programu / przeniesienie na zewnątrz odpowiedzialności za kontrolę wykonania

Wstrzykiwanie zależności

- Tworzenie, konfigurowanie i podawanie zależności „z zewnątrz”
- Zmniejsza sprzężenie między elementami aplikacji, co m.in. daje swobodę wymiany implementacji i ułatwia testowanie
- Przykłady realizacji:
  - Ręczne podawanie zależności np. przez konstruktor, metody
  - Wzorce projektowe np. Factory Method
  - Kontener zarządzający cyklem życia komponentów (Spring, CDI)

Programowanie aspektowe

- Uzupełnia paradygmat programowania obiektowego
- Umożliwia oddzielenie logiki biznesowej od dodatkowych zadań pobocznych, takich jak: transakcje, logowanie, bezpieczeństwo
- Często realizowane z wykorzystaniem wzorca Proxy

Programowanie przez zdarzenia

- Umożliwia rozluźnienie powiązań - komponenty mogą się komunikować mimo, że niewiele o sobie wiedzą
- Opiera się o wzorzec Observer

Domain Driven Design

- Podejście do tworzenia oprogramowania, które kładzie nacisk na to, aby obiekty i ich zachowania wiernie odzwierciedlały rzeczywistość / domenę problemu

Bounded Context

- Koncepcja zakładająca podział aplikacji na konteksty, definiujące własny model dziedzinowy, odwzorowujące konkretne potrzeby, warunki i procesy np. płatności, wyszukiwanie produktów, realizacja zamówień
- Komunikacja między kontekstami odbywa się za pośrednictwem dobrze określonego interfejsu, a zmiany modelu wewnętrznego lub logiki nie powinny mieć na nią wpływu

Anemiczny model domenowy

- Encje reprezentują wyłącznie struktury danych (obiektowa reprezentacja tabel bazy danych)
- DAO - umożliwiają operacje CRUD na encjach
- Services - realizują logikę biznesową, wykorzystują DAO do odczytu / zapisu Encji (Transaction Scripts)

Bogaty model domenowy DDD

- Większość logiki jest zlokalizowana w Agregatach, Encjach i Value Objects
- Repositories - umożliwiają operacje na Agregatach (jest ich mniej niż w przypadku modelu anemicznego)
- Services są dużo lżejsze, zawierają pozostałą cześć logiki

Wymagania niefunkcjonalne i ich wpływ na architekturę

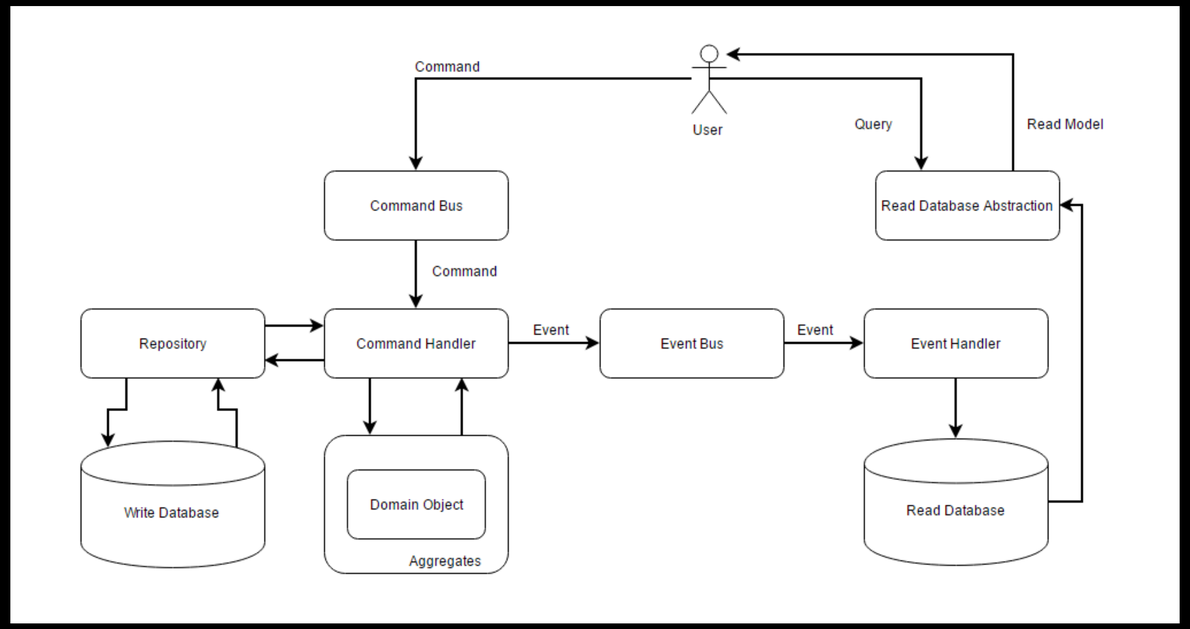
- Opisują kryteria jakościowe rozwiązania
- Najczęściej determinowane przez typ aplikacji, cele biznesowe, budżet czy grupę docelową
- Mogą mieć istotny wpływ na architekturę rozwiązania
- Przykłady:
  - aplikacja powinna być dostępna 24/7/365
  - system powinien być w stanie jednocześnie obsłużyć 1000 użytkowników
  - system powinien być w stanie przetworzyć 200 transakcji w ciągu minuty

Command Query Separation

- Zasada przedstawiona przez w 1986 roku przez Bertranda Meyera, mówiąca o tym, że każda metoda powinna być zaklasyfikowana do jednej z grup:
  - Command - metody, które zmieniają stan aplikacji i nic nie zwracają
  - Query - metody, które coś zwracają, ale nie zmieniają stanu aplikacji

CQRS - Command Query Responsibility Segregation

- Przeniesienie pomysłu CQS na poziom klas / komponentów / całego systemu, zaproponowane przez Greg Young oraz Udi Dahan



https://bulldogjob.pl/articles/122-cqrs-i-event-sourcing-czyli-latwa-droga-do-skalowalnosci-naszych-systemow

CQRS - elementy składowe

- Command - obiekt reprezentujący intencję użytkownika systemu np. UpdateItemQuantityCommand
- Command Bus - kolejka dla przychodzących komend, „router” przekazujący zadania do odpowiedniego handlera
- Command Handler - waliduje otrzymane komendy, tworzy lub zmienia stan obiektu domenowego, utrwala zmiany z pomocą repozytorium (write database) i przekazuje ewentualne zdarzenia do Event Bus
- Domain objects i Aggragates - model domenowy i logika biznesowa, zmiany na obiektach tego typu generują zdarzenia

CQRS - elementy składowe

- Event - obiekt reprezentujący zmiany, które zaszły w domenie
  np. ItemQuantityUpdatedEvent
- Event Bus - kolejka dla generowanych zdarzeń, „router” przekazujący zdarzenia do odpowiedniego event handlera
- Event Handler - utrwala zmiany stanu w bazie do odczytu (read database)
- Read Database Abstraction - abstrakcja / kontrakt umożliwiająca odczyt danych / stanu domeny

Event Sourcing

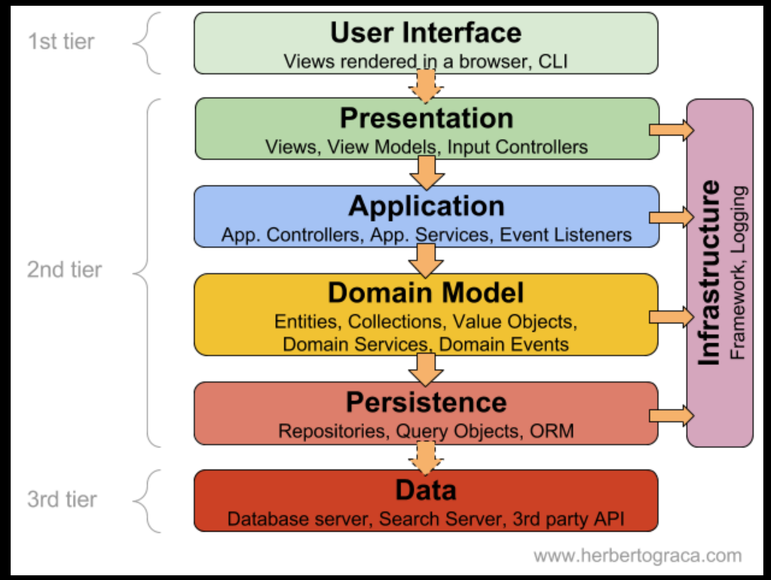
- W CQRS zdarzenia są używane jako mechanizm synchronizacji dwóch baz danych (read i write)
- Event Sourcing pozwala na odtworzenie aktualnego stanu aplikacji (obiektów domenowych) na podstawie zdarzeń składowanych w magazynie danych zwanym Event Store

Architektura

- Definiuje najważniejsze komponenty, zakres ich odpowiedzialności, a także wzajemne relacje
- Stanowi szablon rozwiązania
- Może być definiowana na różnym poziomie np. aplikacja, system

Architektura warstwowa (n-tier pattern)

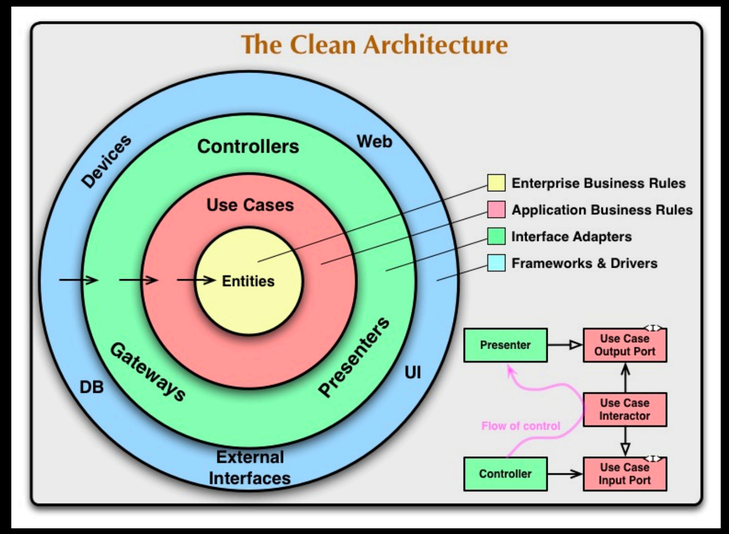
- Komponenty aplikacji organizowane są w warstwy pełniące określoną rolę np. prezentacja, logika, utrwalanie danych (podział techniczny)
- Komunikacja między warstwami odbywa się w jednym kierunku



https://herbertograca.com/2017/08/03/layered-architecture

Czysta architektura

- Centralnym elementem aplikacji jest logika biznesowa, zaimplementowana w sposób niezależny od bibliotek, frameworków czy użytej infrastruktury
- Wokół logiki biznesowej tworzone są kolejne, bardziej wysokopoziomowe warstwy m.in. warstwa adapterów umożliwiająca komunikację ze światem zewnętrznym



https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Monolit

- Aplikacja rozwijana, testowana i wdrażana jako całość (single artifact)
- Zalety (do pewnego rozmiaru projektu)
  - Łatwa implementacja nowych funkcjonalności biznesowych i pobocznych
  - Mała złożoność infrastrukturalna
- Wyzwania (od pewnego rozmiaru projektu)
  - Trudność utrzymania i rozwoju ze względu na rosnącą złożoność, zakres funkcjonalności i rozmiar aplikacji
  - Ograniczona skalowalność
  - Przywiązanie do określonych rozwiązań i technologii

Modularny monolit

- Aplikacja rozwijana, testowana i wdrażana jako zbiór modułów, które:
  - są od siebie niezależne
  - mają dobrze zdefiniowaną odpowiedzialność
  - posiadają publiczny interfejs / kontrakt
  - mogą być reużywane w innych aplikacjach

Architektura zorientowana na usługi (SOA)

- Rozwiązanie w postaci rozproszonych usług / komponentów, które:
  - mają dobrze zdefiniowaną odpowiedzialność
  - posiadają publiczny interfejs / kontrakt
  - mogą współdzielić dane
  - są komponowane / integrowane z wykorzystaniem ESB i standardowych protokołów, aby realizować założone cele biznesowe

Architektura oparta o mikrosierwisy

- Rozwiązanie w postaci rozproszonych mikro usług (mini aplikacji), które:
  - są od siebie niezależne
  - mają dobrze zdefiniowaną odpowiedzialność (najczęściej w oparciu o domenę biznesową)
  - posiadają publiczny interfejs / kontrakt
  - nie współdzielą danych i komunikują się wyłącznie przez publiczne API

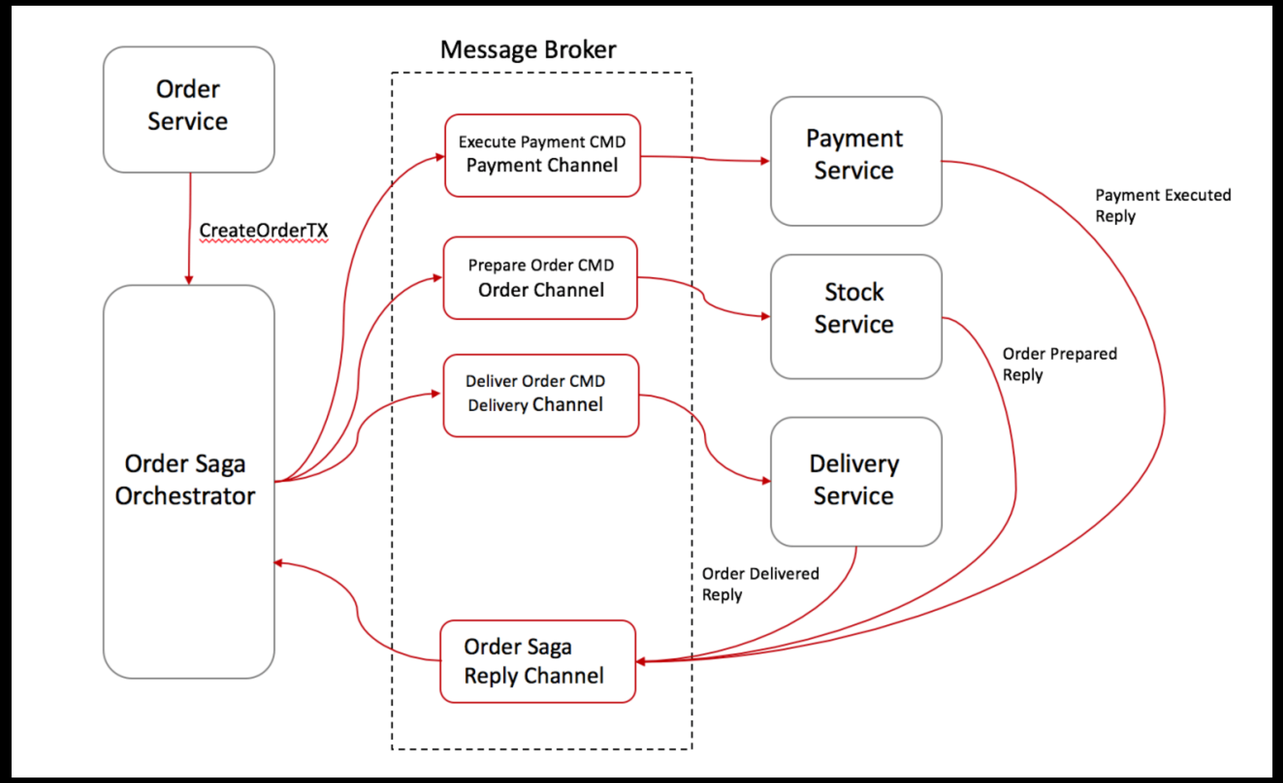
Mikroserwisy - zalety

- Modularność
- Reużywalność
- Skalowalność
- Optymalne zużycie zasobów
- Niezawodność i wysoka dostępność
- Łatwość tworzenia / utrzymania na poziomie pojedynczych usług

Mikroserwisy - wyzwania

- Złożoność na poziomie makro (infrastruktura, utrzymanie, implementacja pobocznych funkcjonalności, synchronizacja stanu / procesów)
- Koszty (zasoby ludzkie, sprzęt, utrzymanie)
- Mniejsza wydajność i większa podatność na awarie (komunikacja przez sieć)

Wzorzec Saga i orkiestracja logiki

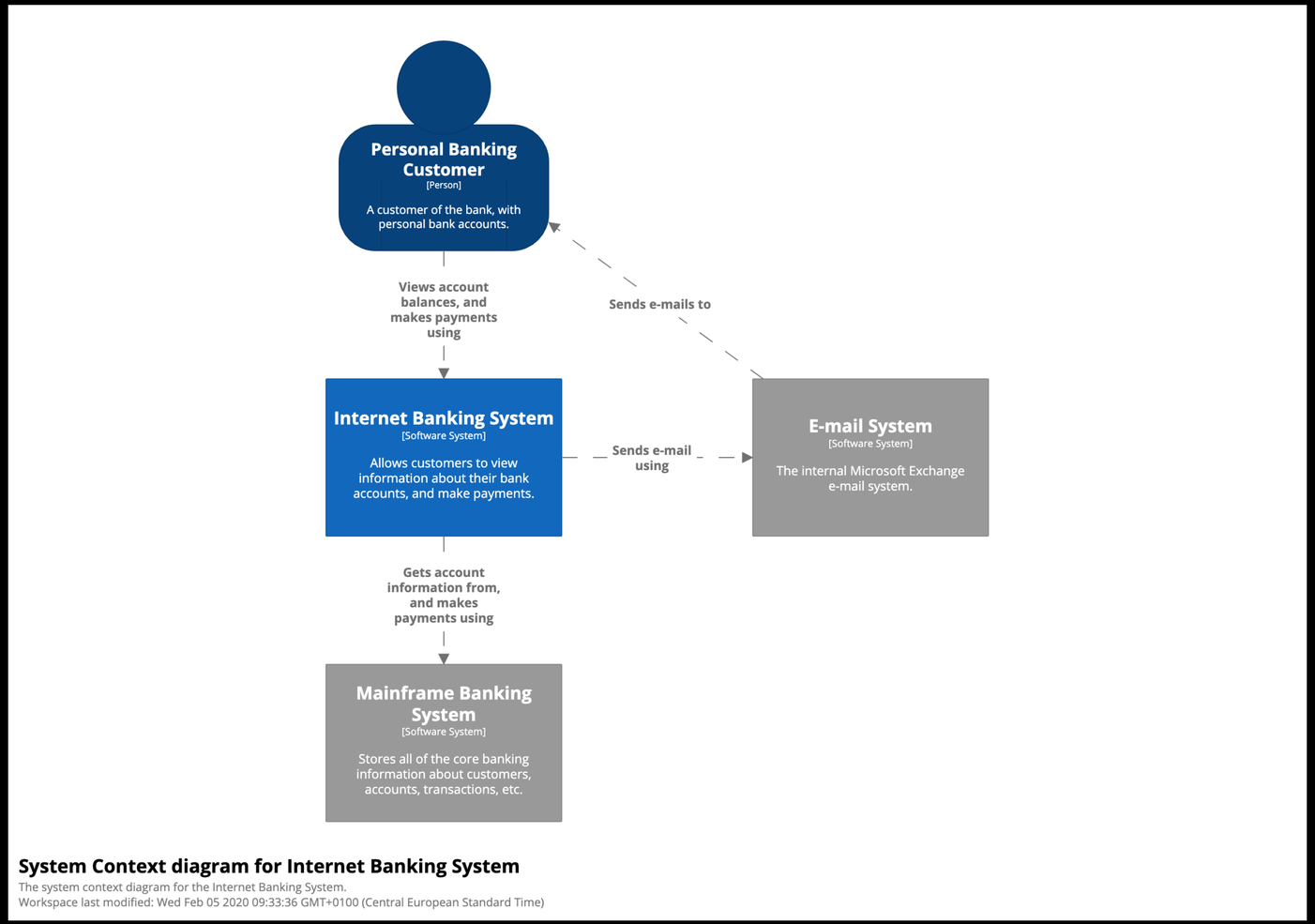


https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part-2

Model C4

- Umożliwia opisywanie / przedstawienie informacji na temat architektury i ułatwia dzielenie się wiedzą na jej temat
- Pokazuje rozwiązanie na różnych poziomach szczegółowość

Model C4 - Context diagram

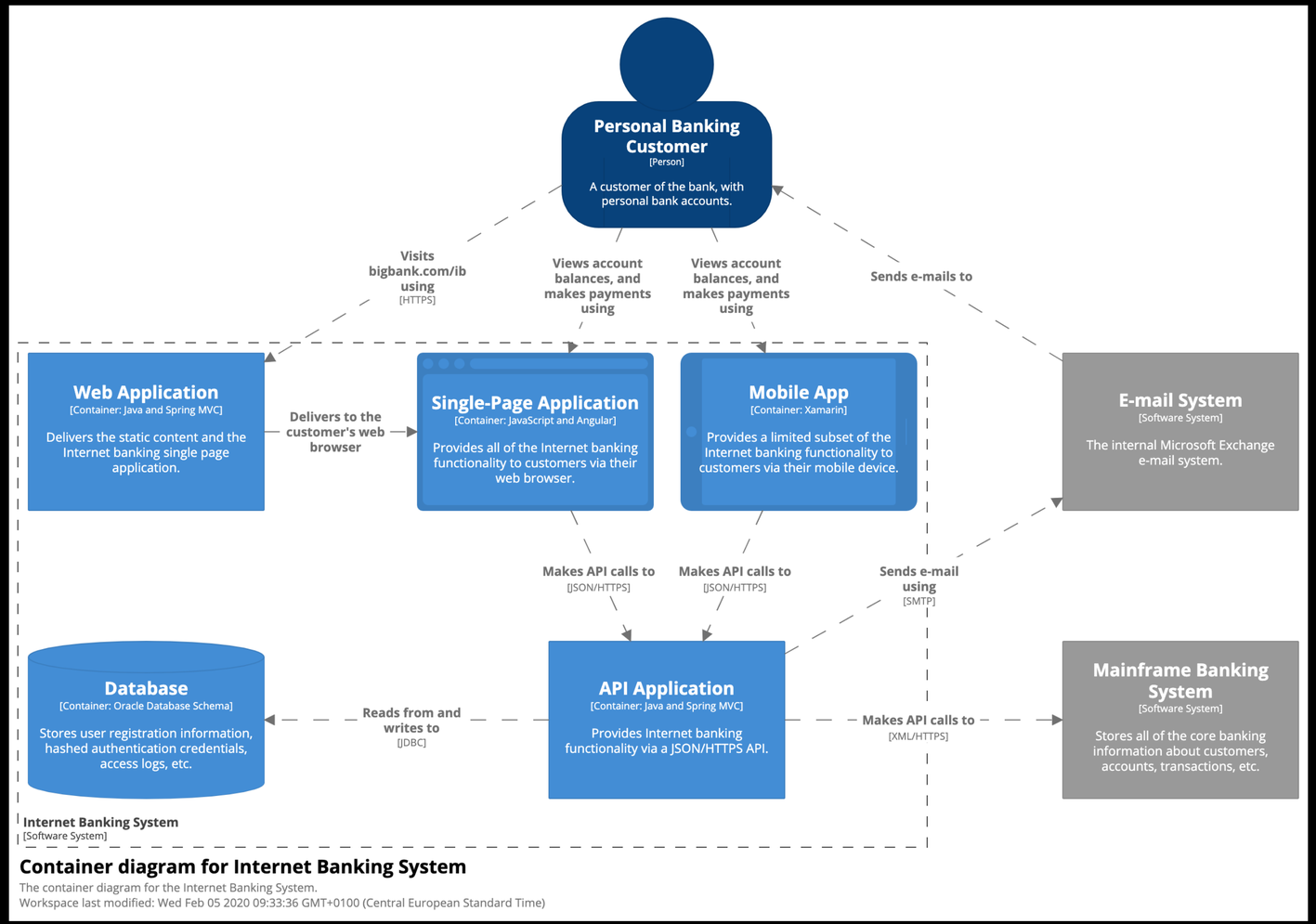


System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

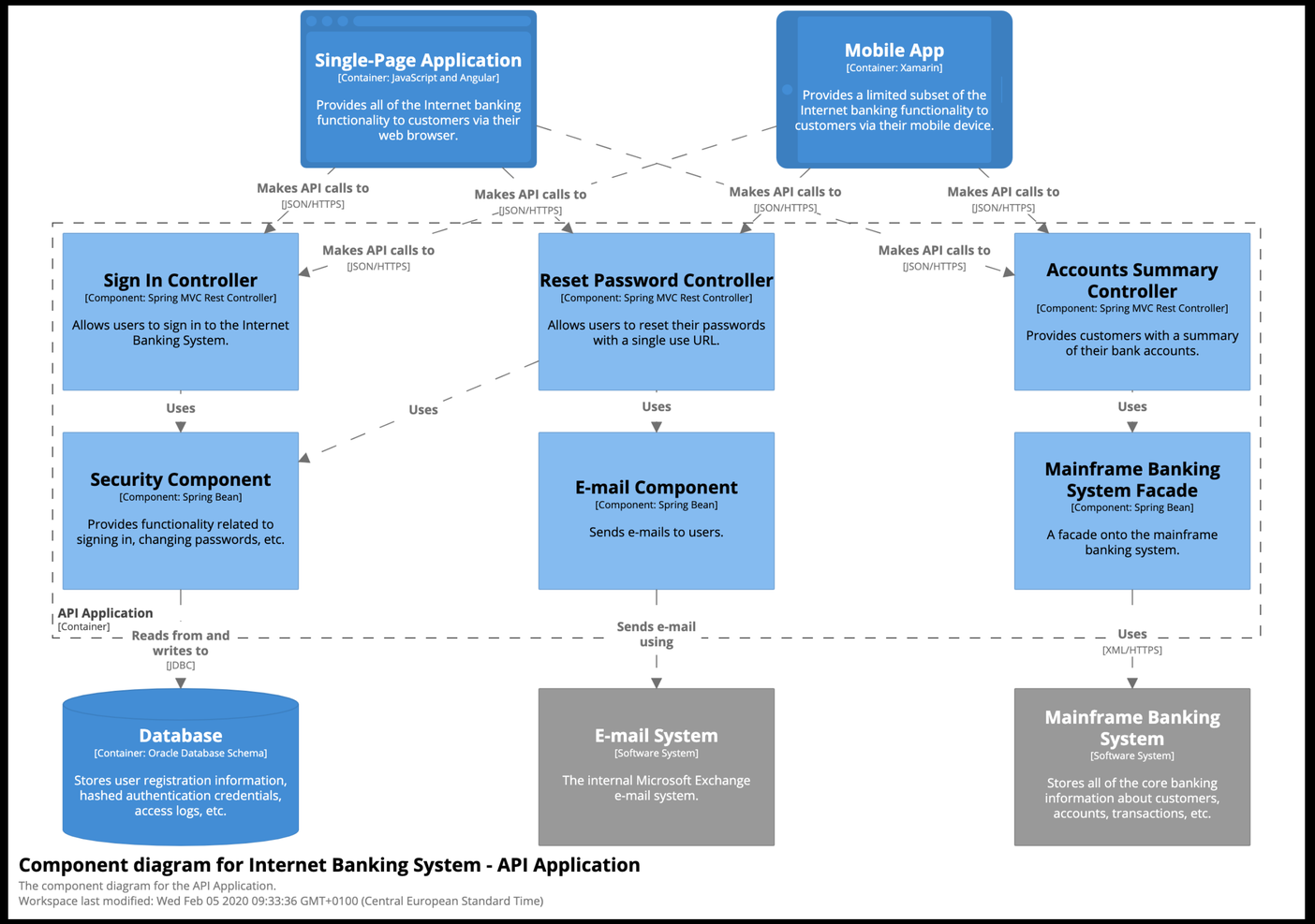
https://c4model.com

Model C4 - Contaner diagram



https://c4model.com

Model C4 - Component diagram

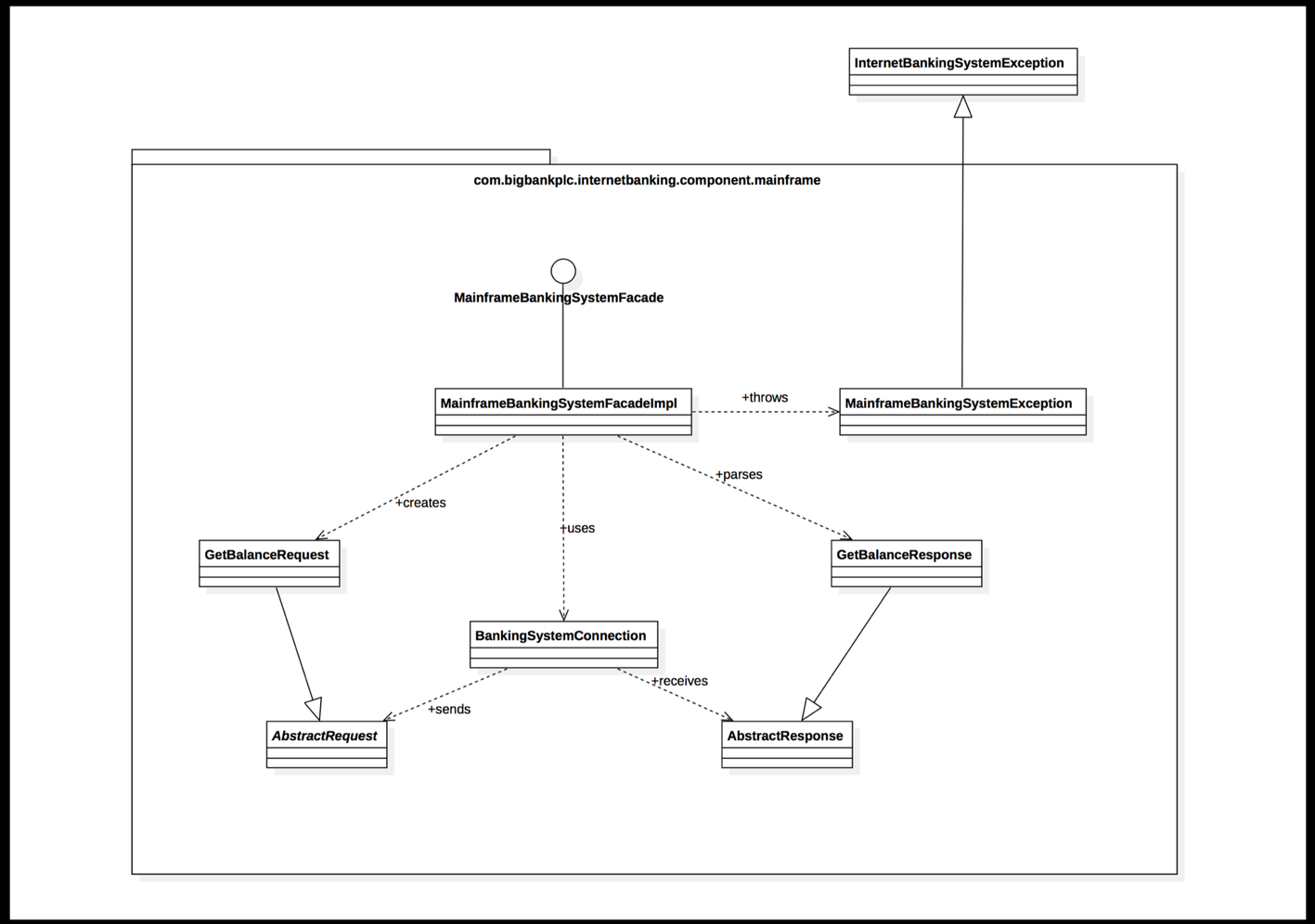


Component diagram for Internet Banking System - API Application

The component diagram for the API Application.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

https://c4model.com

Model C4 - Code / implementation diagram



https://c4model.com