Több platform – egy kódbázis

Tanulságok a Tresorittól

Budai Péter, vezető fejlesztő

Miről lesz szó?

- A Tresorit szolgáltatás és platformjainak gyors bemutatása
- A Tresorit szoftver architektúrája
  - Hogyan épül fel?
  - Miért?
- Kihívások a köztes réteg fejlesztése során
  - Anekdoták
- A felhasználói felület és a köztes réteg kapcsolata
  - Belső API
  - Fejlesztési folyamat és szoftver életciklus
- Útravaló

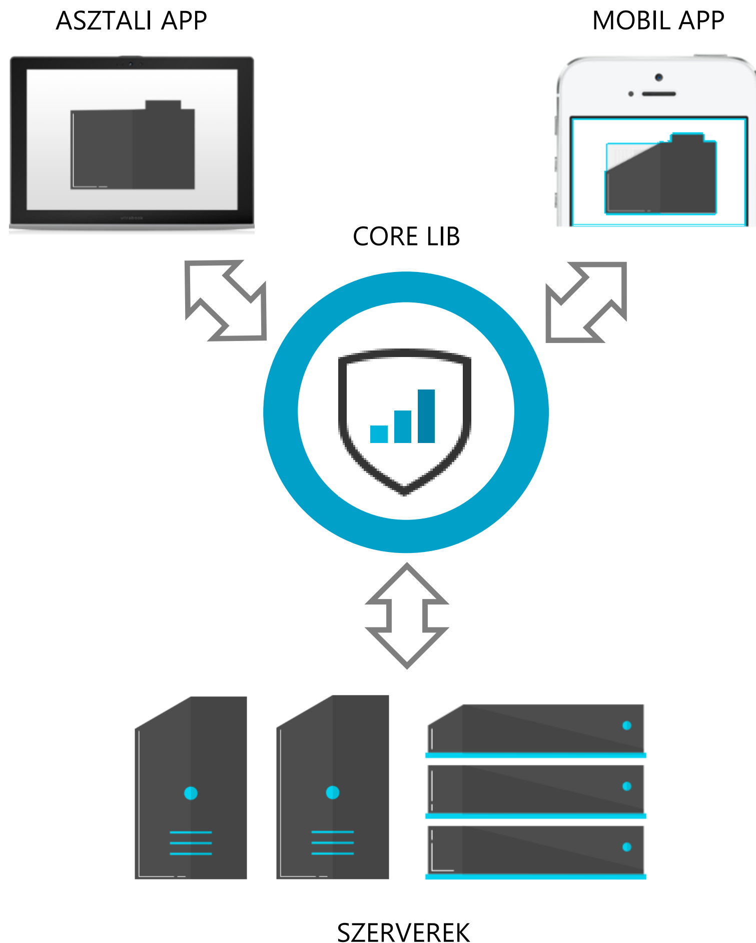
Tresorit - biztonságos kollaboráció 6 platformon

- Biztonságos kollaborációs szoftver
  – Fájlok tárolása, szinkronizációja és megosztása a felhőben
  – Magas szintű biztonság, kliens oldali titkosítás

- 6 platform
  – Windows
  – Mac OS X
  – Android
  – iOS
  – BlackBerry
  – Windows Phone

tresorit

A Tresorit háromszintű architektúrája



- Kliens alkalmazások
  - Platform “natív” eszközkészlet és nyelv
- Kriptográfiai köztes réteg
  - “Core” library
  - C++
  - Beépül a kliensekbe
  - Szerver elérés
- Tároló és kiszolgáló szerverek
  - Microsoft Azure szolgáltatások

tresorit

Miért? Titkosítás és gyorsabb fejlesztés

- Kliens oldali titkosítás
  - Számításigényes műveletek
    - Gépi kódra fordított kód hatékonyabb
    - Jobb sebesség és üzemidő
  - Több kiforrott C/C++ kriptográfiai könyvtár
- Fejlesztés gyorsítása
  - Jelentős kliensoldali logika
    - Csak egyszer kell ugyanazt megvalósítani
  - Hibalehetőségek minimalizálása
    - Biztonságkritikus szoftvernél elengedhetetlen
    - Kevesebb helyen kell ugyanazt implementálni

Miért? Kezelhetőség és csapat

- Könnyű kezelés és fejlesztés
  - Natív, megszokottabb felhasználói felület
    - A operációs rendszer összes szolgáltatása csak a platform natív nyelvéből érhető el
  - Nem igényel "univerzális" fejlesztőket
  - Szerverek csak egyféle kliensre készülnek

- Egyéb tényezők
  - Kezdeti fejlesztőcsapat összetétele
    - Nem elhanyagolható egy startupnál ☺
  - C++11 szabvány elterjedése
    - Sokkal könnyebb fejlesztés, fel sem merült később a váltás

Napi kihívások a köztes rétegben

- C++ fordítás több platformra
  - 1 kódbázis
  - 5 fordító
    - MSVC 18 C++ és C++/CLI módban (Visual Studio 2013), Clang 3.5 (Xcode 6.1), GCC 4.8, GCC 4.9 (Android NDK 10c)
  - 3 standard library
    - Visual C++ runtime, libstdc++, libc++
  - 4 platform
    - Windows, Mac OS X, Android, iOS
  - 4 architektúra
    - x86, x86-64, ARMv5te, ARMv7
  - 11 külső könyvtárat is fordítani kell
    - OpenSSL, libcurl, ICU4C, SQLite és még további 7

tresorit

Kihívások#1 – fordítók és standard libraryk

- Fordítók
  - MSVC
    - Tipikusan megenged nem szabványos kódot is, ami aztán nem fordul a többi platformon
    - C++ és C++/CLI módban más hívási konvenciót fordított, de csak egyetlen függvénynél!
  - Clang
    - Ugyanazt a fájlt fordítva, elsőre összeomlik a fordító, másodszorra hibátlanul lefordítja
    - Néha egy-egy plusz sort bele kell írni, hogy ne omoljon össze fordítás közben
- Standard libraryk
  - Bizonyos STL osztályok nincsenek, vagy hibásan vannak megvalósítva az egyes implementációkban
    - std::thread, std::future, emplace_back(), std::shared_mutex
    - C++ allokátorok

Kihívások#2 – platformok és külső könyvtárak

- Platformok
  - Fájlkezelés
    - Kardinális probléma fájlszinkronizáció során
    - Nem megengedett karakterek Windows fájlnevekben
    - Mac OS X és iOS normalizálja a fájlneveket, a Windows és Android nem
    - Fájl lockolás kezelése
  - C++ kivételkezelés hibás megvalósítása véletlen összeomlásokat okozott OS X alatt
- Külső könyvtárak
  - Próbálkoztunk például boost-tal is, kevés sikerrel
    - boost::thread ugyanúgy hibás Androidon
    - boost::filesystem ugyanúgy platformfüggően működik, mintha natív API-t használnánk

Core - kliens kapcsolat: a belső API

- Belső, objektum-orientált C++ API
  - **A cél:** köztes réteg objektumai = felhasználói felület modelljei
  - API osztályok logikai elemekhez
    - Felhasználói profil, trezorok, fájlok, stb.
  - „Natív” wrapper minden platformhoz
    - Androidon Java JNI
    - OS X-en és iOS-en Objective-C++
    - Windowson .NET C++/CLI
  - Kliens alkalmazások “natív” nyelven

Előnyök, hátrányok – bevált, de sokat tanultunk

- Előnyök
  - Egyelőre bevált
    - Objektum-orientált felépítés jól illeszkedik a GUI logikához
    - Átlátható, strukturált API, eseményvezérelt működés
- Mire kell figyelni?
  - Memóriakezelés
    - A szemétgyűjtés vs, manuális memóriakezelés
  - Többszálúság
    - A belső objektumokat a felhasználói felület felől is lehet olvasni, manipulálni
    - Alaposabban fel kell készülni a többszálú elérésre
- Hátrányok
  - A funkciók bővülésével összetettebbé válik

Fejlesztési folyamat a gyakorlatban

- Igazi erősségek a gyakorlatban
  - Köztes réteg + kliensek külön fejleszthetők
  - A kliens oldali logika (köztes réteg) külön tesztelhető
  - Nem kell egyidejű release minden platformon
- Saját életciklus
  - Külön fejlesztőcsapat, külön verziószám
  - Folyamatos integráció
    - Minden módosítás csak review után kerülhet a főágba
    - Automatikus éjszakai tesztek és verziózott build
  - A kliens bármikor adoptálhatja az újabb Core verziókat
    - Kimaradhat Core verzió, több GUI is megjelenhet ugyanazzal a Core verzióval, stb.

Útravaló

- Mit csinálnánk mégegyszer ugyanígy?
  – A különválasztott GUI és köztes réteg a mi esetünkben jó döntésnek bizonyult
  – Újabb platform adoptálása egyre kevesebb ráadás erőfeszítést igényel
- Mit csinálnánk máshogy?
  – Félig aszinkron API korlátozó tényező
  – Teljesen szétcsatolt, aszinkron, üzenettovábbításon alapuló API

Köszönöm a figyelmet! Kérdések?