Szakmai gyakorlat beszámoló

Boér Lehel  
Z8N953  
GE Healthcare

# Részletes feladatleírás

A szakmai gyakorlatom a GE Healthcarenél végeztem, ahol az Imaging Fabric SDK szoftvercsomagon dolgoztam. Az Imaging Fabric SDK orvosi képalkotó alkalmazások fejlesztéséhez tartalmaz backend oldali komponenseket. Ezek többek között DICOM képek különböző adatbázisokból való olvasásához, kereséséhez, megjelenítéséhez, autentikációhoz, tömörítéshez, logoláshoz, konfiguráció kezeléshez biztosítanak szolgáltatást.

Az SDK a fejlesztők számára készült, egy adott program vagy platform szolgáltatásainak használatát, az azon futó programok készítését lehetővé tevő készlet.  
  
A fejlesztőkészletek a feltétlenül szükséges dokumentáció és könyvtárfájlok mellett sokszor segédprogramokat, vagy akár komplett fejlesztőeszközöket is magukban foglalhatnak. Mivel SDK, cél, hogy ne egyetlen specifikus alkalmazáshoz készüljön, hanem a sok testreszabható komponensből álljon, és könnyen integrálhatóak legyenek számos alkalmazásba.

Magyarországon az Imaging Fabric SDK jelenleg négy csapat dolgozik, és mindegyik különböző komponensekért fele. Az én csapatom – a Troopers – az API Gateway, Configuration Adapter, a Logging és a VNA BulkDataLoader komponensekért felel. Én munkám során főleg az első kettővel foglalkoztam.

Mivel belsős információkat nem adhatok ki, így az egyes komponensek architektúráját és pontos működését sem írhatom le. Ehelyett egy általános leírást adok róluk, hogy mire valók és hogyan működnek, illetve bemutatom a fejlesztés során használt technológiákat.

A szakmai gyakorlatom során számomra új volt a Scrum szoftverfejlesztési módszertan, a multi környezet és a feladatok menedzselése, a meetingek, a mentorálás és a folyamatos kommunikáció. Így a következőkben ismertetett technológiák mellett ezekben is sokat fejlődtem.

# GE Healthcare

A GE több, mint 28 éve van jelen Magyarországon és nemcsak a legnagyobb amerikai befektető, de egyben az egyik legnagyobb magyar vállalat is.

A GE digitális beruházásaira, kiterjedt ellátási láncára, valamint az ország kiemelkedő gyártási és digitális képességeire támaszkodva magyarországi üzemeiben korszerű egészségügyi, olaj- és gázipari, légi közlekedési, és energiatermelési berendezéseket és gépeket gyárt, amelyek kiegészülnek a budapesti Globális Szolgáltató Központ és a Digitális Fejlesztési Központ szolgáltatásaival.

A GE közel 4 000 munkatársat foglalkoztat Magyarországon, ahol 4 gyár, 3 kutatás-fejlesztést végző központ (GE Power, GE Healthcare, GE Aviation) és egy regionális üzleti központ található.

A GE Healthcare gyökeresen alakítja át a technológiákat és a szolgáltatásokat annak érdekében, hogy a magasabb minőségű és megfizethető egészségügy iránt megnövekedett igényt szerte a világon kielégítse. A GE Healthcare egészségügyi szakemberek részére biztosít megoldásokat az orvosi képalkotó programoktól, informatikai és szoftver megoldástól, beteg felügyeleti rendszerektől kezdve a diagnosztikán, gyógyszerkutatáson át, egészen a biotechnológiai gyártási technológiákig, hogy azok kiváló egészségügyi szolgáltatást nyújthassanak betegeiknek.

# Imaging Fabric SDK

Ebben a fejezetben bemutatom a csapatom által fejlesztett SDK szoftverkomponenseket.

## API Gateway

Az API Gateway egy fordított proxy szerepét tölti be az SDK-ban és token alapú autentikációval biztosítja a komponensek közti kommunikáció biztonságát és hitelességét. Az API Gateway lehetővé teszi a szolgáltatások elérését a felhasználó számára biztonságos HTTPS és WebSocket illetve Secure WebSocket protokollokon keresztül.

A fordított proxy egy olyan proxy szerver (a számítógép-hálózatok működését gyorsító eszköz), ami a sima proxyval ellentétben nem a klienshez, hanem a szerverhez van közel, és a célja nem a hálózati forgalomnak, hanem a szerver terhelésének a csökkentése. A működése hasonló a sima proxyhoz: a kliens és a szerver között helyezkedik el, megjegyzi, hogy egy adott kérésre a szerver milyen választ adott, és ha később ugyanazt a kérést kapja, akkor nem küldi ismét tovább a szervernek, hanem a saját memóriájából válaszolja meg. Mivel ugyanaz kontrollálja, aki a szervert is, a sima proxynál kifinomultabb funkciókat is meg tud valósítani; például a szerver jelezni tudja a fordított proxynak, ha a tárolt tartalom idejétmúlttá válik, illetve lehetséges a válasznak egyes részeit a proxyn, más részeit a szerveren előállítani.

## Configuration Adapter

A Configuration Adapter külső konfigurációs adatbázisból képes konfigurációt generálni SDK-s alkalmazások számára. Transzformálás alatt egyrészt kiegészíti olyan biztonsági információkkal amit nem tárolhatunk SDK-n kívüli adatbázisban, másrészt megfelelő formára hozza azt (json, xml stb.). Képes a konfigurációs adatbázis tartalmát folyamatosan figyelni, és amint az megváltozik rögtön újragenerálja a konfigurációt.

## Logging

Általános java logging lib SDK-s alkalmazások számára. Ezt dependenciaként húzzák be. Különböző események logolásához különböző logging kontextusokat kell felépíteni, a lib gondoskodik ezután a logok megfelelő formájú kiírásáról.

## VNA BulkDataLoader

Viszonylag új projekt a csapatunkban, Vendor Neutral Archive típusú adatbázisokból képes DICOM képeket betölteni WADO-RS végpontokon keresztül az SDK számára.

# Fejlesztési környezet

A fejlesztés során egy Windows operációs rendszerű számítógépen és egy Linuxos virtuális gépen dolgoztam. Az Imaging Fabric SDK Windows-on saját installerrel, Linuxon Docker fölött fut. A két külöböző telepítési és futási mód teljesítménybeli okok miatt szükséges. Ugyan Windowsra is lehet Dockert telepíteni, mérések szerint jobb megoldás volt a windows szolgáltatásokba csomagolt java alkalmazások használata. Emiatt a fejlesztés ugyan több munkát igényel, de a szoftver is nagyobb teljesítményre képes.

# Scrum

A Scrum egy olyan keretrendszer, amelynek segítségével emberek komplex problémákat tudnak adaptív módon kezelni úgy, hogy közben termelékenyen és kreatívan szállítják le a lehető legértékesebb termékeket.

A Scrum:

* Egyszerű
* Könnyen érthető
* Rendkívül nehezen művelhető mesteri szinten

A Scrum egy folyamat-keretrendszer, amit az 1990-es évek eleje óta használnak komplex termékek fejlesztésére. Nem egy termékek létrehozására kitalált folyamat vagy technika; sokkal inkább egy olyan keretrendszer, melyen belül különböző folyamatokat és technikákat lehet alkalmazni.

A Scrum láthatóvá teszi a termék menedzsmentjének és a fejlesztési gyakorlatainak relatív hatékonyságát, így elősegíti annak tökéletesítését. A Scrum keretrendszer a Scrum Csapatokból, valamint a hozzájuk rendelt szerepekből, eseményekből, munkaanyagokból (artifacts) és szabályokból áll. A keretrendszeren belül minden egyes komponens meghatározott célt szolgál, és mindegyik alapvetően szükséges a Scrum sikeréhez és használatához. A Scrum szabályai kapcsolják össze az eseményeket, szerepköröket és a munkaanyagokat, meghatározva a köztük lévő viszonyokat és kölcsönhatásokat. A Scrum szabályait e dokumentum törzse ismerteti. A Scrum keretrendszer használatának sajátos technikái, taktikái eltérőek lehetnek és más forrásokban találhatók meg.

# Microservice architektúra

A mikroszervizek kialakításánál a tervezők megoldást próbáltak találni az összetett alkalmazások fejlesztésekor jelentkező szervezési problémákra, melyek az egyes komponensek fejlesztési ciklusainak összekapcsolódásából adódnak, és a szoftverfejlesztők munkáját nagyban megnehezítik.

Egy monolitikus szolgáltatás orientált architektúra esetén minden kis változtatás a teljes alkalmazás újrafordítását jelentheti. Mivel a fejlesztés jellemzően sok kis változtatás sorozatából áll, az újrafordítás nagyon gyakori, ami ha elhúzódik az a fejlesztők nem hatékony munkájához, frusztrációjához vezet. Ennél még komolyabb probléma, hogy az összefüggések miatt a fejlesztőknek a változtatásokat egymással koordináltan kell végrehajtaniuk. Minél összetettebb az alkalmazás, és minél többen dolgoznak rajta, annál nehezebb és a hibákra is érzékenyebb ez a feladat.

A mikroszerviz architektúra, a monolitikus megközelítéssel ellentétben, lehetővé teszi minden egyes mikroszerviznek egy egyedi folyamat futtatását és a saját adatbázisának menedzselését. Ez a fejlesztő csapatoknak lehetőséget biztosít az alkalmazás decentralizált tervezésére, valamint lehetővé teszi a komponensek egymástól független menedzselését.

Mivel a mikroszervizek meglehetősen széles és összetett területét fedik le az informatikának, úgy döntöttünk készítünk egy cikket a látogatóinknak, hogy elmagyarázzunk néhány általános fogalmat és tippet az ilyen rendszerekről.

A mikroszervízek inkább fogalamak, mint használt technológiák gyűjteménye. Nem más, mint egy szoftver design ajánlás lazán összekapcsolt, könnyen változtatható és skálázható szolgáltatás architektúrák készítésére.

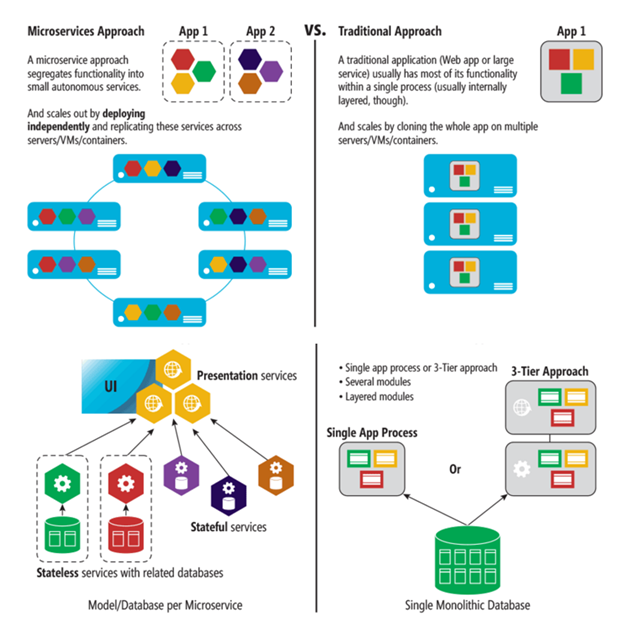
A monolitikus szolgáltatások hátrányai általában:

* meredek tanulási görbét kell a fejlesztőknek bejárniuk
* egyszerre az egész szoftverrel és környezettel kell foglalkozni
* nagyon nagy kódbázisok bonyolult ütközéseinek megoldása a feladat
* egy hatalmas szoftverben a hibák terjedése több modulra is kihathat
* rendkívül nehéz az egyes részek új üzleti igényeknek vagy új technológiáknak megfelelő megváltoztatása, újraírása (refactoring)

A szoftvernek a mikroszerviz elvek mentén kisebb részekre való feldarabolása több szempontból előnyös lehet:

* az egyes komponensek fejlesztését 2-3 fős független csapatokhoz lehet rendelni, akik tudásukat el tudják mélyíteni az adott területen és felelősséget tudnak így vállalni az adott komponens teljes életciklusára
* könnyebben érthető kódok, mivel a kódbázis sokkal kisebb
* sokkal kevesebb merge konfliktus mivel kevesebb ember dolgozik ugyanazon a kódon
* a leghatékonyabb technológiákat lehet alkalmazni a komponens feladataira
* a hardware erőforrásokat az adott feladatokhoz lehet külön-külön optimalizálni
* lazán csatolt, nyelvfüggetlen interfészek a komponensek között

A következő ábrán összehasonlítva áttekinthetőek a mikroszerviz megközelítés valamint a hagyományos monolit megközelítés közötti lényegesebb különbségek:



# A megismert technológiák

## Docker

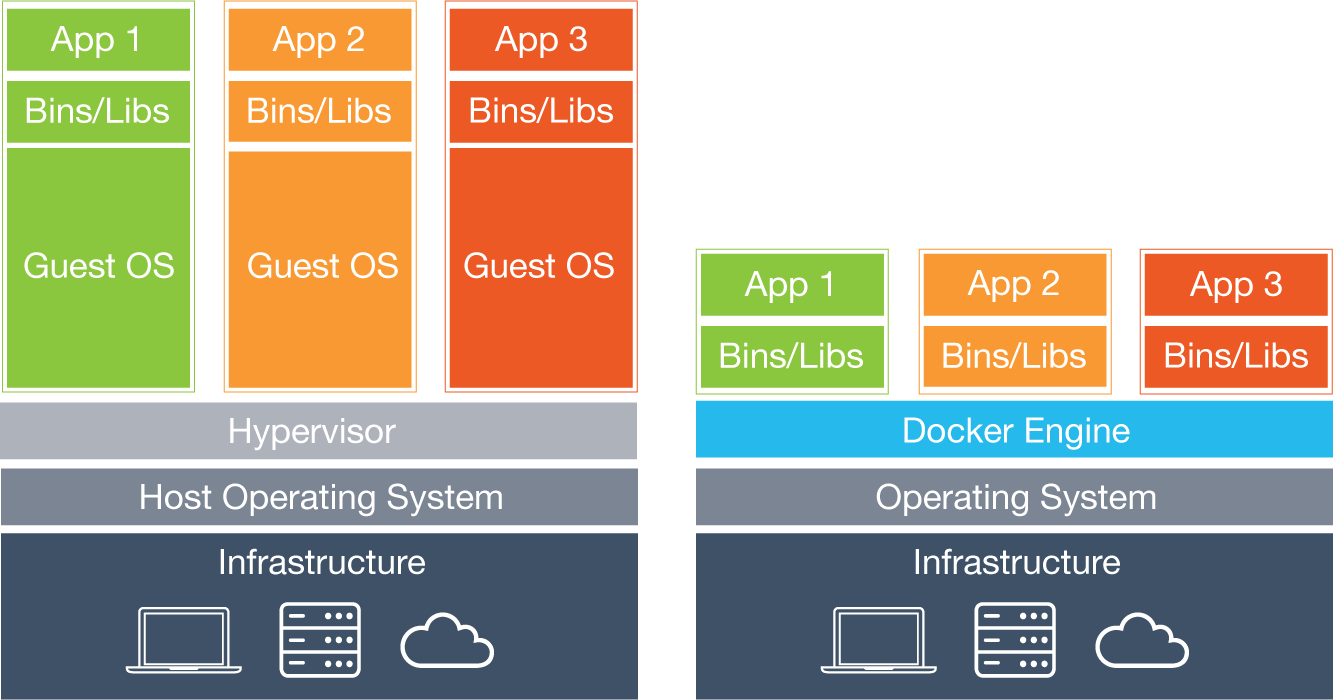
**A virtuális gépek:**

A containerek és a VM-ek gyakorlatilag ugyanarra a problémára adnak megoldást: hogyan lehet izolálni egy applikációt és annak összes függőségét egy bárhol futtatható egységbe. Azonban a két megoldás architektúrája meglehetősen különböző; első körben tekintsük át ezt az aspektust. A virtuális gép egy valódi számítógép emulációját takarja, és szinte pontosan úgy hajt végre bármilyen feladatot, mint egy valódi, fizikailag létező számítógép. A VM a gazdaszámítógép erőforrásait (RAM, CPU, stb.) terheli, így ha több VM-et futtatunk egyszerre, mind ezeken kell osztozniuk. Ezt az ún. hypervisor réteg segíti, ez biztosít átjárást a fizikai gép operációs rendszere és a VM-ek között.

Fontos látni, hogy minden egyes VM **tartalmazza a teljes szoftver-, és hardver stacket**, amin az alkalmazások futnak, így a virtualizált hardvereket (hálózati adapter, CPU, merevlemezek, stb.), a teljes operációs rendszert, és a felhasználói területet.

**Konténerek:**

A legnagyobb különbség a containerek és a virtuális gépek között az, hogy **minden container osztozik a gazda kerneljén a többi containerrel**, így ebben az esetben a virtualizált hardver és az OS nem része a “csomagnak”, csupán az alkalmazásunkhoz kötődő könyvtárak, binárisok és a felhasználó terület. Könnyen látható, hogy ezáltal a container nagyságrendekkel kisebb overheaddel bír a VM-ekhez képest, mondhatni egy “lightweight” technológia az alkalmazások futtatási környezettel együtt történő becsomagolására.



**Docker:**

A Docker jelenleg az egyik legelterjedtebb container framework, amelynek implementációja erősen támaszkodik a Linux kernel nyújtotta szeparációs lehetőségekre (namespacek, control groupok). Azonban a technológia már a Docker előtt is létezett – a Google pl. régóta használja -, így felmerülhet kérdés, hogy miért éppen a Docker lett az a megoldás, amely futótűzként terjedt el?

A legkézenfekvőbb válasz erre a**könnyű használhatóság**. A Docker fejlesztők és sysadminok számára egyaránt egy nagyon egyszerű lehetőséget biztosít hordozható alkalmazások létrehozására és menedzselésére. Segítségével bárki fejleszthet és becsomagolhat egy alkalmazást például a saját laptopján, és biztos lehet benne, hogy az egy teljesen másik környezetben, tipikusan egy cloud szerveren futtatva is ugyanúgy működni fog.

A másik komoly előny a **sebesség**. A technológiából fakadóan a Docker containerek kicsik és és gyorsak, hiszen tulajdonképpen csak egy, a kernelen futó sandbox környezetről van szó, amely kevés erőforrást emészt fel.

További két tényező, amelyet érdemes megemlíteni, az **a modularitás és a skálázhatóság**. A Dockerrel nagyon egyszerűen “feldarabolhatjuk” az alkalmazásunkat különböző containerekbe, például egy containerben csücsül a MongoDB adatbázisunk, egy másikban egy Redis szerver, és egy harmadik containerben pedig az ezekhez kapcsolódó Node.js app.

A Docker alapvetően három részből áll: a daemon, a kliens, és egy REST API. A daemon a host gépen fut és parancsokat fogad a klienstől, az API segítségével pedig közvetlenül kommunikálhatunk a daemonnal. Mi felhasználóként persze elsősorban a klienst fogjuk használni, ennek adunk ki különböző parancsokat.

A containerek definiálására az ún. dockerfile szolgál, ez instrukciókat tartalmaz arra vonatkozóan, hogyan épüljön fel a container. Ezekkel a parancsokkal például előírhatjuk az operációs rendszer verzióját, bizonyos Linux packagek installációját, portok megnyitását, és még rengeteg egyéb dolgot. A részletek ismertetése nélkül így néz ki például egy dockerfile, ez konkrétan egy nagyon alap Apache szerver definícióját tartalmazza:

# A basic apache server. To use either add or bind mount content under /var/www

FROM ubuntu:12.04

MAINTAINER Kimbro Staken version: 0.1

RUN apt-get update && apt-get install -y apache2 && apt-get clean && rm -rf /var/lib/apt/lists/\*

ENV APACHE\_RUN\_USER www-data

ENV APACHE\_RUN\_GROUP www-data

ENV APACHE\_LOG\_DIR /var/log/apache2

EXPOSE 80

CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]

Forrás: [*https://github.com/kstaken/dockerfile-examples*](https://github.com/kstaken/dockerfile-examples)

A dockerfile elkészülte után jöhet a build lépés, ami egy Docker image-t fog létrehozni. A Docker image-re úgy kell gondolnunk, mint egy kizárólag olvasható template-re, ami a fájlban meghatározottak alapján létrejövő környezetet tartalmazza. Ha ehhez hozzácsatoljuk az alkalmazásunkat és egy írható/olvasható fájlrendszert, létrejön a Docker container. Amint ez kész, az alkalmazásunk bármely környezetben változtatás nélkül futtathatóvá válik.

## Spring

A Spring egy nyílt forráskódú, inversion of controlt megvalósító Java alkalmazás keretrendszer.

A Spring keretrendszer magját képező szolgáltatásokat főként Java alkalmazás fejlesztésére használják a programozók. Ugyanakkor a Java EE platformra is elérhetők a bővítményei, amelyek web-alkalmazás fejlesztését segítik elő. Nem rendelkezik külön specifikált fejlesztési modellel, hanem az Enterprise JavaBean (EJB) modell kiegészítése-, helyettesítője-, vagy alternatívájaként vált népszerűvé a Java fejlesztők között.

A Spring keretrendszer több önálló modulból épül fel, amelyek az alábbi szolgáltatásokat nyújtják a fejlesztők számára:

* [Inversion of control](https://hu.wikipedia.org/wiki/Inversion_of_control) konténer: a Java objektumok életciklusának kezelése és az alkalmazás-komponensek testreszabása.
* Aspektus orientált [programozási paradigma](https://hu.wikipedia.org/wiki/Programoz%C3%A1si_paradigma) követésének lehetősége.
* Adatelérés: lehetőség van [relációs adatbázis-kezelő](https://hu.wikipedia.org/wiki/Rel%C3%A1ci%C3%B3s_adatb%C3%A1zis-kezel%C5%91) rendszerek [JDBC](https://hu.wikipedia.org/wiki/JDBC) segítségével történő elérésre, és [objektum-relációs leképzések](https://hu.wikipedia.org/wiki/Objektum-rel%C3%A1ci%C3%B3s_lek%C3%A9pz%C3%A9s), NoSQL integrálására.
* Tranzakciókezelés: többféle tranzakció kezelő API-t tartalmaz.
* [Modell-nézet-vezérlő](https://hu.wikipedia.org/wiki/Modell-n%C3%A9zet-vez%C3%A9rl%C5%91) szabvány: egy [HTTP](https://hu.wikipedia.org/wiki/HTTP)- és [servlet](https://hu.wikipedia.org/wiki/Servlet) alapú keretrendszer segítségével valósítható meg, amelyet arra fejlesztettek ki, hogy bővíthetők és személyre szabhatóak legyenek a [webszolgáltatások](https://hu.wikipedia.org/wiki/Webszolg%C3%A1ltat%C3%A1s)
* Távoli eljáráshívás kezelő keretrendszer: biztosítja a [RPC](https://hu.wikipedia.org/wiki/Remote_procedure_call) alapú, hálózaton keresztül történő Java objektum importokat és exportokat. További támogatást nyújt a [RMI](https://hu.wikipedia.org/wiki/Java_remote_method_invocation), a [CORBA](https://hu.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture) és [HTTP](https://hu.wikipedia.org/wiki/HTTP) alapú protokollok használatára, beleértve a [webszolgáltatásokat](https://hu.wikipedia.org/wiki/Webszolg%C3%A1ltat%C3%A1s) ([SOAP](https://hu.wikipedia.org/wiki/SOAP)) is.
* Kötegelési eljárás támogatása.
* Azonosítás és azonosságkezelés: biztonsági folyamatok konfigurálása, melyet a Spring projekthez tartozó, Spring Security alprojekt tesz lehetővé a különféle protokollok és módszerek biztosításával.
* Üzenetkezelés: a [JMS API](https://hu.wikipedia.org/wiki/Java_Message_Service)-n keresztül történő általános üzenetkezelés továbbfejlesztése érhető el.
* [Tesztelés](https://hu.wikipedia.org/wiki/Szoftvertesztel%C3%A9s): segítséget nyújt a unit- és az integrációs teszt írására.

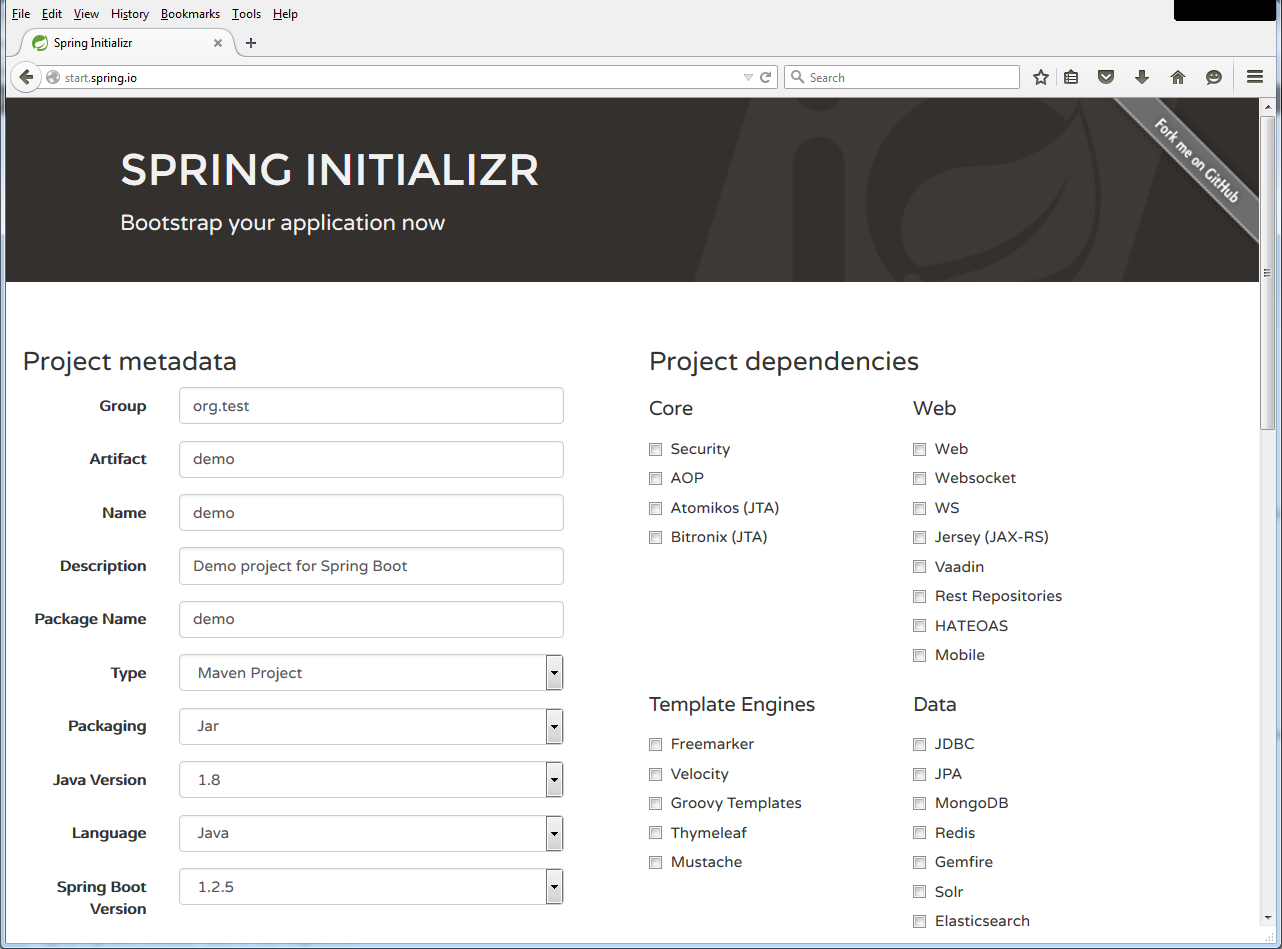
## Spring Boot

A Spring Boot a Spring „convention-over-configuration” megoldása stand-alone futtatható Spring-alapú alkalmazások létrehozására. Ezek automatikusan konfiguráltak a Spring csapat ajánlása szerint, illetve dependenciaként használják a Spring Platform egyes részeit és egyéb 3rd party könyvtárakat.

Előnyei:

* Stand-alone Spring alkalmazások készítése.
* Beágyazott Tomcat vagy Jetty (nincs szükség WAR fájlok telepítésére).
* Starter POM.xml-t biztosít mavenhez a szükséges dependenciákkal, így kevesebb konfigurációra van szükség.
* Automatikusan konfigurálja a Springet ahol csak lehetséges.
* Production-ready szolgáltatásokat biztosít, mint metrikák, health check és külső konfiguráció
* Nincs kódgenerálás és nincs szükség XML leírókra.

Új projekt lérehozásánál érdemes a Spring honlapjáról kiindulni:

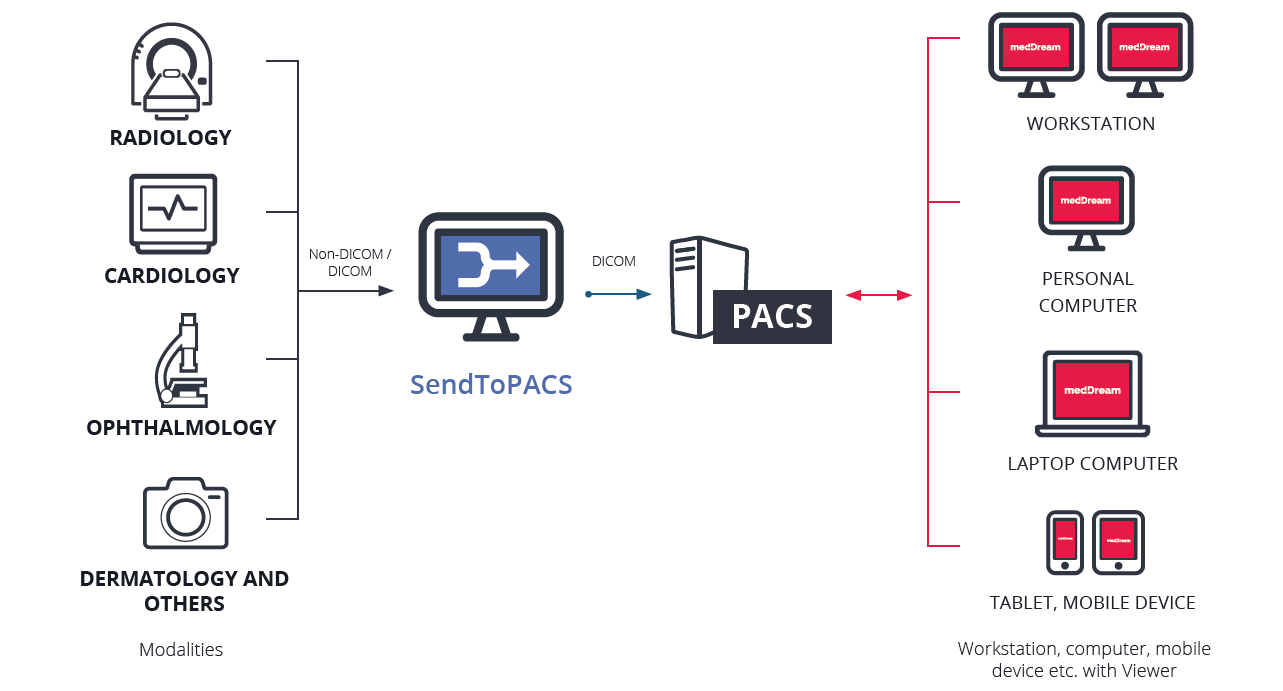


## Vert.x

Az Eclipse Vert.x egy polyglot eseményvezérelt toolkit Java alapú reaktív alkalmazások fejlesztéséhez. A Vert.X nem blokkoló és eseményvezérelt, így az alkalmazás kevés szál esetén is jól skálázódik és konkurrens módon működik.

## DICOM

A Digital Imaging and Communications in Medicine (DICOM), azaz Digitális képkezelés és kommunikáció az orvostudományban szabványok átfogó halmaza információ kezelésére, tárolására és továbbítására orvosi képekben. Fájlformátum definíciót és hálózati kommunikációs protokollt tartalmaz. A protokoll egy alkalmazás protokoll, mely TCP/IP használatával teremt kapcsolatot a gépek között.



A DICOM-ot úgy tervezték, hogy lehetővé tegye több gyártótól származó szkennerek, szerverek, munkaállomások és hálózati hardverek integrációját egyetlen képarchiváló és kommunikációs rendszerré. A DICOM-ot széles körben alkalmazzák a kórházak, és egyre jobban elterjed a kisebb alkalmazásokban is, mint a fogorvosi és körzeti orvosi rendelőkben.

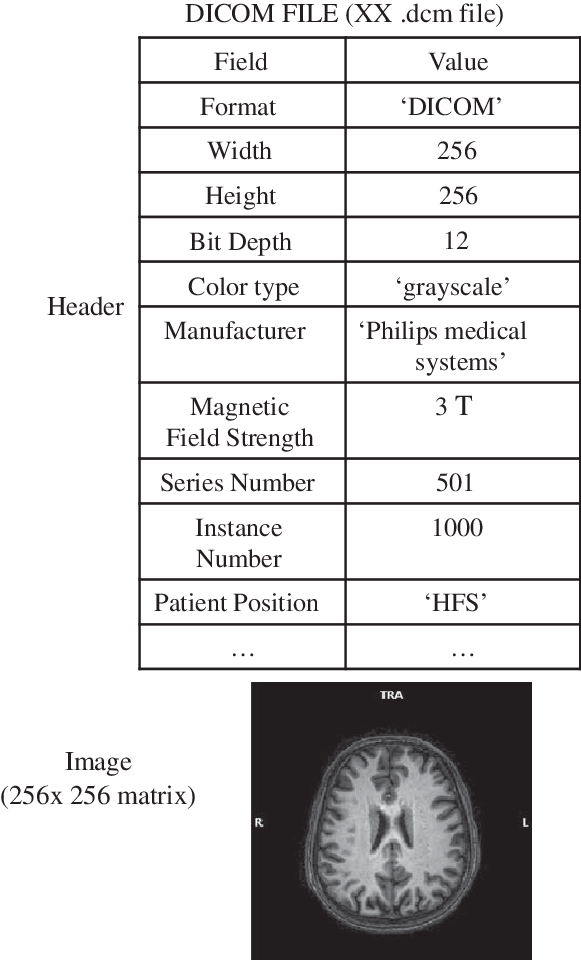
A DICOM két átviteli módra bontható:

* DICOM Store – Feladata képek küldése PACS (digitális képfeldolgozó és archiváló) rendszereknek vagy munkaállomásoknak.
* DICOM Print – Feladata képek küldése DICOM (száraz lézer) nyomtatókra "röntgenfilm" nyomtatása céljából.

A legtöbb gyártó egyedi licencet kér, hogy ezeket az átviteleket végre lehessen hajtani. A szabvány lehetővé teszi a küldő számára a képek minőségének beállítását.

### DICOM formátum

A DICOM fájlok szabványos és szabad felhasználású mezőkből álló fejrészből és a képi adattartalomból épülnek fel. Egyetlen DICOM fájl egy vagy több képet tartalmazhat, kötegek vagy animációk tárolását. A képi adat tömöríthető több szabványnak megfelelően is, többek között lehet JPEG, LZW vagy RLE is a kódolás.



A DICOM abban különbözik a többi adatformátumtól, hogy az információt adathalmazokba csoportosítja. Ez azt jelenti, hogy egy páciens mellkasának röntgenfelvétele egy fájlban helyezkedik el a páciens azonosítójával, így a kép nem keverhető el véletlenül az adatoktól.

A DICOM specifikáció részletei a NEMA honlapján megtalálhatók, valamint több szabad felhasználású és nyílt forráskódú implementáció is létezik.

## Cucumber

A Cucumber egy olyan szoftver tool amely támogatja a BDD-t (Behavior-Driven Development). A Gherkin nyelvet használja és segítségével tesztesetekhez üzleti nyelvű leírásokat rendelhetünk.

A tesztek Feature fájlokban helyezkednek el. Ezeken belül ún. Scneariok, amelyeken belül pedig Stepek vannak. Az egyes Stepekhez mögöttes kód tartozik. A modern idék támogatják az intelligens navigációt ezek között.

Példa:

**Feature:** Withdraw Money from ATM

A user with an account at a bank would like to withdraw money from an ATM.

Provided he has a valid account and debit or credit card, he should be allowed to make the transaction. The ATM will tend the requested amount of money, return his card, and subtract amount of the withdrawal from the user's account.

**Scenario Outline:** A user withdraws money from an ATM

**Given** <Name> has a valid Credit or Debit card

**And** their account balance is <OriginalBalance>

**When** they insert their card

**And** withdraw <WithdrawalAmount>

**Then** the ATM should return <WithdrawalAmount>

**And** their account balance is <NewBalance>

**Examples:**

**|** Name **|** OriginalBalance **|** WithdrawalAmount **|** NewBalance **|**

**|** Eric **|** 100 **|** 45 **|** 55 **|**

**|** Gaurav **|** 100 **|** 40 **|** 60 **|**

**|** Ed **|** 1000 **|** 200 **|** 800 **|**

A tesztek paraméterezhetők. A példa Scenario alatti Example tábla tartalmazza, hogy az milyen paraméterekkel fusson le.

Cucumberben én a stepek újrafelhasználását és az olvashatóságot, megérthetőséget látom a legnagyobb előnynek.

# Saját feladatok

Mivel Scrumban dolgozunk, minden sprintben a Product Owner által kért vagy legmagasabb priorítással rendelkező User Storyk formájában dolgozunk a terméken. Ebben a fejezetben bemutatok pár érdekesebb User Storyt példának, amelyekkel a legtöbbet foglalkoztam.

## A fejlesztési folyamat

A fejlesztés során verziókezelőnek Gitet, online repositorynak GitLabot, feladatkezelőnek Rallyt, Continous Integration céljából pedig Jenkinst használtunk.

* + - User Story értelmezése
    - Grooming ha vannak tisztázatlan kérdések
    - Taskok létrehozása amennyiben lehetséges
    - Ha valami blokkolja az adott taskot vagy sztorit, annak blokkoltra állítása
    - A sztori Rallyban In Progressbe állítása
  1. Fejlesztés
     + Mindig saját branchen folyik a fejlesztés, nem kommitolunk a masterre
     + Remote branch létrehozása
     + Merge Request készítése
     + MR template kitöltése
     + Reviewerek hozzáadása
     + Review javítása
     + Unit teszt Jenkins pipeline legyen zöld (ne legyenek failelő unit tesztek)
  2. Mergelés
  3. A hivatalos funkcionális tesztek egy részhalmaza, azaz a smoke tesztek lefutnak egy virtuális gépre telepített SDK-n. Ha a futtatás sikeres, következik a hivatalos környezetbeli tesztelés.
  4. Hivatalos tesztek lefutnak egy hivatalos környezetben (virtuális gépen). Ezeknek a teszteknek sem szabad failelnie.
     + User Story completedbe állítása Rallyban.
     + DOD táblázat kitöltése

## Configuration Adapter - initializer mód hozzáadása

A Configuration Adapter Consul-ból olvas fel konfigurációt, ezt kiegészíti biztonsági információkkal, transzformálja, majd a megfelelő szolgáltatásoknak átadja a friss konfigurációt.

Megtörténhetett egyes esetekben – például ha a Consul nem futott a háttérben, vagy ha valamilyen okból nem volt kapcsolat a Consul és a Configuration Adapter között -, hogy egyes szolgáltatások régebbi konfigurációval indultak el.

Ennek kiküszöbölésére implementáltam egy initializer módot, ami egyszer elindítja a Configuration Adaptert, megfigyeli, hogy megkapja-e a legfrissebb konfigurációkat a Consulból, logolja majd leáll.

Jelenleg a windows installeres telepítő is lefuttatja az initializer módot, és jelzi a felhasználónak, ha a Consul tartalma nem elérhető.

Természetesen a Consul elindítása után a Configuration Adapter rögtön frissíti a konfigurációkat, nem igényel újraindítást vagy újratelepítst a folyamat.

## Docker – Java Corretto base docker imagek létrehozása

Mivel az SDK Linuxon Dockeren fut, az egyes alkalmazások Dockerfájlokat is tartalmaznak, ami leírja a Docker image részleteit: azaz a futtató környezetet és az alkalmazás elindítását. Mivel az Oracle megszünteti a régebbi Java verziók hosszú távú támogatását és az SDK hivatalosan Java 8-at használ, áttértünk az Amazon által fejlesztett és támogatott Java Corretto-ra, ami hosszú távú Java 8 supportot biztosít.

Ehhez módosítani kellett az alkalmazások base image-eit olyan módon, hogy a futtató operációs rendszer ne változzon, de a futtatott Java verziót Corretto-8 legyen. A feladatot nehezítette, hogy az Amazon által kiadott hivatalos correttos docker imagek mögött amazonlinux fut, ezért kézzel fabrikált docker imageket kellett készítenem.

## API Gateway – Undertows tesztelés Linuxon

A teszteléshez Java Cucumbert használtunk. Cél az összes komponens teszt Java Cucumberbe való migrációja, hogy a CI folyamat gyorsaságát segítsük. Mivel minden git pushnál lefutnak a unit tesztek, ha ezek között sok a komponens teszt, ami például elindít egy SpringBoot alkalmazást, a pipeline lassú lesz.

Ebből a célból készült egy Undertows projekt, amely segítségével az API Gateway mögötti komponenseket tudjuk mockolni.

Az Undertow egy mock szerver, amely arra jó, hogy könnyen tudjunk tesztelni WS és HTTP hívásokat általunk létrehozott endpointokon általunk létrehozott válaszokkal.

A másik igény az volt, hogy teszteléshez az API Gatewayt könnyedén elindíthassuk custom konfigurációval. Ehhez Windows és Linuxos környezeten is egy új szolgáltatást indítunk. Míg Windowsos esetben ez csupán egy jar fájl megfelelő felparamétezése a Linuxos eset ennél jóval bonyolultabb.

Linuxon az új szolgáltatást be kell kapcsolni egy docker servicebe úgy, hogy a docker konténer és a benne futó alkalmazás is megfelelő paraméterekkel fusson. Ez a fejlesztési környezetben, saját virtuális gépen, és a Jenkins által futtatott hivatalos tesztek esetén különbözik, mert míg saját környezetben a Java Cucumbert lokális futtatjuk, hivatalos környezetben az is docker konténerben fut.

## Developer Guide

Foglalkoztam dokumentáció írással is. Az egyes szolgáltatásoknak frissíteni kellett a Readme-jét, Developer Guide-ot kellett írni hozzá vagy a részletes fejlesztői dokumentációt kibővíteni. Ez is hozzátartozik a szoftverfejlesztéshez, fontos, hogy felülvizsgálat esetén minden rendben legyen.

## Biztonsági rések ellenőrzése

Egy User Storyban a feladatom az egyes szolgáltatások által használt Redis verzióhoz tartozó GitHub repositoryban található Issuek átnézése és kategorizálása. Ha érint minket fenyegetés, annak és hatásának elemzése az adott szolgáltatásra nézve. Általános dolog, hogy folyamatosan új és új biztonsági résekről kapunk feladatokat, ezekre ellenőriznünk kell, hogy sebezhető-e a szoftverünk, ha nem akkor javítanunk kell.

## Irodalomjegyzék

1. <https://ithub.hu/blog/post/Docker_bevezetes_a_containerek_vilagaba/>
2. <http://www.gehealthcare.hu/>
3. <https://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-HU.pdf>
4. <https://ithub.hu/blog/post/A_microservice_architekturarol_diohejban/>
5. <https://hu.wikipedia.org/wiki/Spring_keretrendszer>
6. <https://hu.wikipedia.org/wiki/Digital_Imaging_and_Communications_in_Medicine>