Gradus Vol 12, No 1 (2025)
ISSN 2064-8014

JAVA SPRING KERETRENDSZER

JAVA SPRING FRAMEWORK

Subecz Zoltán [ORCID 0000-0001-5036-4679 1*]

[1] Informatika Tanszék, GAMF Műszaki és Informatikai Kar, Neumann János Egyetem, Magyarország
https://doi.org/10.47833/2025.1.CSC.002

Kulcsszavak:
Java programozás
Java Spring keretrendszer
Java Spring Core Container
Inversion of Control
Dependency Injection

Keywords:
Java programming
Java Spring framework
Java Spring Core Container
Inversion of Control
Dependency Injection

Cikktörténet:
Beérkezett 2024. november 20.
Átdolgozva 2025. március 24.
Elfogadva 2025. március 28.

Összefoglalás

A Java Spring keretrendszer egy nyílt forráskódú Java platform, amely átfogó támogatást nyújt robusztus Java alkalmazások egyszerű és gyors fejlesztéséhez, egy jó megoldás a vállalati használatra kész alkalmazások felépítéséhez [4]. Fő jellemzői az Inversion of Control és a Dependency Injection, amelyek leegyszerűsítik a moduláris és tesztelhető alkalmazások létrehozását. [8] A legfontosabb funkciói közé tartoznak még a Spring MVC a webfejlesztéshez, a Spring Boot az alkalmazások gyors beállításához és a Spring Security a robusztus autentikációhoz és autorizációhoz. A Spring Data az adatbázis műveletekhez és a Spring Cloud a felhőszolgáltatások kiépítéséhez kapcsolódó gazdag ökoszisztémájával a Spring méretezhető és rugalmas vállalati megoldásokat támogat [10]. A cikkben bemutatjuk a Java Spring keretrendszert és annak fő jellemzőit az Inversion of Control-t és a Dependency Injection-t.

Abstract

The Java Spring framework is an open-source Java platform that provides comprehensive support for developing robust Java applications quickly and easily and is a good solution for building enterprise-ready applications [4]. Its main features are Inversion of Control and Dependency Injection, which simplify the creation of modular and testable applications [8]. Other key features include Spring MVC for web development, Spring Boot for rapid application provisioning, and Spring Security for robust authentication and authorization. With its rich ecosystem of Spring Data for database operations and Spring Cloud for building cloud services, Spring supports scalable and flexible enterprise solutions [10]. In this article, we introduce the Java Spring framework and its main features, Inversion of Control and Dependency Injection.

1. Bevezetés

A Spring framework egy nyílt forráskódú, a legnépszerűbb alkalmazásfejlesztési keretrendszer a vállalati Java alkalmazások számára [3]. Világszerte fejlesztők milliói használják nagy teljesítményű, könnyen tesztelhető, újra felhasználható kód létrehozására. A Spring könnyű súlyú, ami a méretet és az átláthatóságot illeti, alapverziója körülbelül 2 MB méretű [5]. Alapvető funkciói

* Kapcsolattartó szerző.
E-mail cím: subecz.zoltan@nje.hu

bármely Java-alkalmazás fejlesztésében használhatók, de vannak bővítmények a webalkalmazások Java EE platformra történő építéséhez. A Spring keretrendszer célja a J2EE fejlesztés könnyebben használhatóbbá tétele és a jó programozási gyakorlat előmozdítása. Átfogó programozási és konfigurációs modellt biztosít a modern Java alapú vállalati alkalmazásokhoz bármilyen telepítési platformon [1]. A Spring kulcsfontosságú eleme az infrastrukturális támogatás az alkalmazások szintjén: a vállalati alkalmazások összeállítására összpontosít, így a fejlesztők az alkalmazásszintű üzleti logikára összpontosíthatnak [6].

A Spring fő tulajdonságai [2]:

- Rugalmas Dependency injection
- Fejlett támogatás az aspektusorientált programozáshoz
- Első osztályú támogatás az általános nyílt forráskódú keretrendszerekhez, mint például a Hibernate
- Rugalmas webes keretrendszer RESTful MVC-alkalmazások és szolgáltatásvégpontok építéséhez

és fő felhasználási területei [9]**:**

- Microservices
- Cloud
- Web applications
- Serverless
- Batch processing

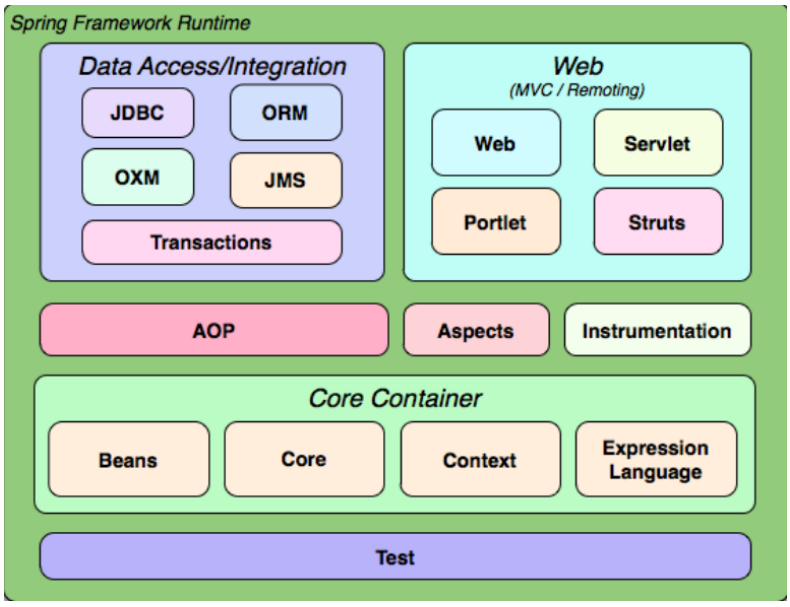
A Spring moduláris felépítésű, lehetővé téve az egyes részek, például a Core container vagy a JDBC támogatás fokozatos beépítését. Bár az összes Spring szolgáltatás tökéletesen illeszkedik a Spring Core containerhez, sok szolgáltatás programozott módon is használható a containeren kívül.

A Spring keretrendszernek az *Inversion of Control* (IoC, Vezérlés megfordítása) és *Dependency Injection* (DI, Függőség injektálása) funkciói adják az alapot a szolgáltatások és funkciók széles skálájához. Az IoC koncepciója az osztályok közötti kapcsolat lazítása. Ahelyett, hogy egy tagváltozót konkrét osztállyal inicializálnánk, egy interfész használható a szükséges osztálytípus jelzésére. A szoftverfejlesztésben a *Dependency Injection* olyan programozási technika, amelyben egy objektum vagy funkció más objektumokat vagy funkciókat kap, amelyekre szüksége van, nem pedig belső módon hozza létre. A *Dependency Injection* célja az objektumok létrehozásával és használatával kapcsolatos problémák szétválasztása, ami lazán csatolt objektumokhoz vezet. A DI minta biztosítja, hogy egy adott szolgáltatást használni kívánó objektumnak vagy függvénynek ne kelljen tudnia, hogyan kell ezeket a szolgáltatásokat létrehozni. Ehelyett a fogadó "klienst" külső kód (az "injektor") látja el függőségeivel, amelyeket a kliens nem ismer részletesen [11].

A konkrét osztály létrehozását a Spring kezeli és beinjektálja az osztályba. A beinjektált osztály vezérlésének feladata már nem a kliens osztályé, hanem a Springé, így a vezérlés megfordult, innen az Inversion of Control elnevezés. A Spring lehetővé teszi az IoC beállítását XML konfiguráció vagy a modernebb Java annotációk használatával is.

2. A Spring keretrendszer architektúrája és moduljai

A Spring keretrendszer moduláris, és 20 olyan modulból áll, amelyek különböző feladatokat látnak. Ezek a modulok a következő fő csoportokba vannak szervezve: Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation és Teszt. (*1. ábra*)



1. ábra. A Spring keretrendszer architektúrája [7]

A Spring kiemelt építőelemei a **Bean**-ek: olyan objektumok, amelyek az alkalmazás gerincét alkotják, amelyeket a Spring IoC Container futási időben példányosít és állít össze.

Itt részletesen bemutatjuk az előző ábrán megadott összetevőket, modulokat.

2.1. A Core Container csoport

A Core Container biztosítja a Spring keretrendszer alapvető funkcióit, a következő modulokat tartalmazza:

- **Spring Core**: Ez a modul szolgáltatja a Spring keretrendszer alapvető funkcióit, beleértve az *Inversion of Control* konténert és a *Dependency Injection*-t. Az IoC-konténer a Spring Framework szíve, amely a JavaBeans példányok létrehozásáért és kezeléséért felelős és Dependency Injection-t használ a bean-ek összerendezéséhez.
- **Spring Beans**: Ez a modul biztosítja a BeanFactory-t, amely az IoC konténer alapvető építőeleme, és a BeanWrappert, amely a bean életciklusának szabályozásáért felelős. A Bean Factory az IoC konténer elérésének alapvető interfésze, módszereket biztosít a bean-ek kezelésére.
- **Spring Context**: Ez a modul biztosítja az ApplicationContext-et, amely a BeanFactory továbbfejlesztett változata, és további szolgáltatásokat biztosít, mint például a nemzetköziesítés és az erőforrások betöltése, valamint az események közzétételének és felhasználásának lehetősége.
- **Spring Expression Language (SpEL)**: Ez a modul hatékony nyelvet biztosít az objektumok gráfjának lekérdezéséhez és módosításához futás közben. A szolgáltatások széles skáláját támogatja, beleértve a tulajdonság-hozzáférést, a metódusok meghívását, a ciklusokat és a típuskonverziót. Támogatja továbbá az alkalmazás-kontextusban definiált változók és függvények elérését, valamint az egyéni függvények és változók meghatározását.

2.2. A Data Access/Integration csoport

Az Data Access/Integration csoport támogatja az adatbázisokkal és más adatforrásokkal való integrációt. A következő modulokat tartalmazza:

- **Spring JDBC**: Ez a modul egy egyszerű JDBC absztrakciós réteget biztosít, amely csökkenti a JDBC-vel való munkához szükséges alapkód mennyiségét. Támogatja a tranzakciókezelést, lehetővé téve a fejlesztők számára, hogy deklaratív módon kezeljék az adatbázis-tranzakciókat.

- **Spring ORM**: Ez a modul integrációt biztosít az Object-Relational Mapping (ORM) keretrendszerekkel, mint például a Hibernate és a JPA. Magasabb szintű absztrakciós réteget biztosít az ORM-keretrendszerek felett, lehetővé téve a fejlesztők számára, hogy kevesebb alapkódot írjanak, és könnyebben integrálják az ORM-technológiákat a Spring egyéb szolgáltatásaival, például a tranzakciókezeléssel és a gyorsítótárazással.
- **Spring Data**: Ez a modul konzisztens és könnyen használható programozási modellt biztosít az adatelérési technológiákkal való munkavégzéshez, beleértve az adatbázisokat, a NoSQL-t és a felhőalapú adatszolgáltatásokat. Szolgáltatások széles skáláját kínálja, beleértve az automatikus CRUD (Create, Read, Update, Delete) műveleteket, a metódusnevekből lekérdezések generálását, a lapozás és a rendezés támogatását, a Spring tranzakciókezelésével való integrációt. Ezenkívül támogatja az általános adathozzáférési mintákat, például a repository-kat és az adatelérési objektumokat (DAO).
- **Spring Transaction**: Ez a modul támogatja a deklaratív tranzakciókezelést a Spring alkalmazásokban. Különféle tranzakcióterjedési és elkülönítési szintekhez nyújt támogatást, lehetővé téve a fejlesztők számára, hogy a tranzakciókat különböző részletességi szinteken kezeljék. Ezenkívül különféle tranzakciókezelési stratégiákhoz nyújt támogatást, mint például a JTA tranzakciókezelő vagy az egyszerű JDBC tranzakciókezelő használata.

2.3. A Web csoport

A Web csoport támogatást nyújt webalkalmazások készítéséhez. A következő modulokat tartalmazza:

- **Spring MVC**: Ez a modul Model-View-Controller (MVC) keretrendszert biztosít webes alkalmazások építéséhez. Számos szolgáltatást kínál, beleértve a HTTP-kérések és válaszok kezelésének támogatását, az űrlapkezelést, az adatkötést, a validációt és. Támogatja a különböző nézet (view) technológiákat is, mint például a JSP (JavaServer Pages), a Thymeleaf és a Velocity, így a fejlesztők kiválaszthatják az igényeiknek leginkább megfelelő nézet technológiát.
- **Spring WebFlux**: Ez a modul reaktív programozási modellt biztosít nagy párhuzamosságot és méretezhetőséget igénylő webalkalmazások készítéséhez. Támogatást nyújt reaktív webalkalmazások létrehozásához számos technológia segítségével, például Netty, Undertow és Servlet 3.1+ konténerek használatával. Ezenkívül számos reaktív szolgáltatást kínál, beleértve az adathozzáférést, az adatfolyam-feldolgozást és a HTTP-klienseket.
- **Spring Web Services**: Ez a modul támogatja a SOAP alapú és RESTful webszolgáltatások felépítését. Támogatja a WSDL (Web Services Description Language) létrehozását Java osztályokból, valamint a Java osztályok WSDL-ből történő előállítását. Ez lehetővé teszi a fejlesztők számára, hogy WSDL használatával meghatározzák webszolgáltatásuk szerződését (vagyis a felületet), és a WSDL-ből generálják a webszolgáltatást megvalósító Java osztályokat.

2.4. Egyebek

- Az **AOP** modul az aspektus-orientált programozási megvalósítást biztosítja.
- Az **Aspects** modul integrációt biztosít az AspectJ-vel ami ismét egy erőteljes és érett aspektus-orientált programozási (AOP) keretrendszer.
- A **Instrumentation** modul osztály-kezelési támogatást és osztály-betöltési implementációkat biztosít, amelyeket alkalmazás-szerverekben lehet felhasználni.
- A **Test** modul támogatja a Spring tesztelését JUnit vagy TestNG keretrendszerekkel.

3. Az Inversion of Control és a Dependency Injection

A Core Spring az Inversion of Control (IoC) és a Dependency Injection (DI) alapelveire összpontosít. Ezek a koncepciók kulcsfontosságúak az összetevők szétválasztásához és az objektumok életciklusainak kezeléséhez.

3.1. Inversion of Control

A Spring IoC (Inversion of Control) Container a Spring keretrendszer magja. Létrehozza az objektumokat, konfigurálja és összeállítja a függőségeiket, kezeli a teljes életciklusukat. A Container a Dependency Injection segítségével kezeli az alkalmazást alkotó összetevőket, a bean-eket. Mivel a Java objektumok és életciklusuk vezérlését nem a fejlesztők végzik, hanem a Spring, innen ered a Vezérlés megfordítása (Inversion Of Control) elnevezés.

Az IoC tároló két fajtája: BeanFactory és ApplicationContext. A BeanFactory a legegyszerűbb konténer, amely alapvető támogatást nyújt a Dependency Injection számára. Az ApplicationContext pedig a BeanFactory szolgáltatásait bővíti ki.

3.2. Dependency Injection

A Dependency Injection (DI) az IoC által biztosított fő funkció. Komplex Java-alkalmazás írásakor az alkalmazás-osztályoknak a lehető legfüggetlenebbeknek kell lenniük a többi Java osztálytól, azért, hogy növeljük ezen osztályok újra felhasználásának lehetőségét és hogy más osztályoktól függetlenül lehessen ezeket tesztelni. Az IoC tervezési elve hangsúlyozza, hogy a Java osztályok maradjanak függetlenek egymástól, ezt alkalmazva az IoC felszabadítja az osztályokat az objektumok létrehozásának és karbantartásának feladatától. A Dependency Injection biztosítja az osztályok közötti laza csatolást, segíti ezen osztályok egymáshoz kapcsolását, közben függetlennek tartva azokat.

Miért hasznos a Dependency Injection? Tegyük fel, hogy az Első nevű osztálynak szüksége van a Második osztály objektumára egy metódus példányosításához vagy működtetéséhez. Ilyenkor az Első osztály a Második osztálytól függ, ami sok problémához vezethet, ezért az ilyen függőséget el kell kerülni. A Spring IOC feloldja az ilyen függőségeket a Dependency Injection segítségével, amely megkönnyíti a kód tesztelését és újra-felhasználását is.

Az osztályok közötti laza csatolás lehetséges interfészek definiálásával, és az injektor példányosítja a szükséges objektumot és beinjektálja a kliensbe. Az objektumok példányosításának feladatát a konténer végzi a fejlesztő által megadott konfigurációk szerint.

A Spring-Core modul felelős a függőségek beinjektálásáért, amit Constructor vagy Setter metódusokon keresztül tud megtenni. Nézzünk egy példát a konstruktor injektálásra, ahol az Első osztálynak szüksége van a Második osztály objektumára. A hagyományos kód így néz ki:

```
public class Első {
 private Második második;
 public Első() {
       második = new Második();
       ……
 }
}
```

Itt függőséget hoztunk létre az Első és a Második osztály között. Az Inversion of Control módszer esetén ehelyett valami ilyesmit teszünk:

```
public class Első {
 private Második második;
 @Autowired
 public Első(Második második) {
       this.második = második;
       ……
 }
}
```

@Autowired annotáció: Lehetővé teszi a Spring számára, hogy feloldja és beinjektálja az együttműködő bean-t az osztályunkba.

Itt az Első osztálynak nem kell foglalkozni a Második implementációjával. A Második függetlenül lesz implementálva és az Első rendelkezésére fog állni az Első példányosításakor, és ezt az egész eljárást a Spring keretrendszer irányítja. Eltávolítottuk a vezérlést az Elsőből, és máshol adjuk meg meg (pl. egy XML konfigurációs fájlban, vagy annotációval) és a függőség (a Második osztály) az Első osztályba kerül az osztálykonstruktoron keresztül. Így az irányítás folyamata megfordult a Dependency Injection segítségével, mivel hatékonyan delegáltunk függőségeket egy külső rendszer számára.

A Dependency Injection másik módja a Setter injektálás, ahol az injektálás egy setter metóduson kelresztül történik. Erre egy példa az előző osztályok esetén:

```
public class Első {
 private Második második;
 @Autowired
 public setMásodik(Második második) {
       this.második = második;
       ……
 }
}
```

3.2.1. Dependency Injection és a tesztelés

A Dependency Injection egyik fő előnye, hogy a kódnak sokkal kevésbé kell függenie a konténerektől, mint a hagyományos J2EE fejlesztésben. Ha követjük a Spring körüli architektúra-ajánlásokat, akkor rá fogunk jönni, hogy a kódbázis ebből eredő tiszta rétegezése és komponensezése megkönnyíti az egységtesztelést. Például tesztelhetjük az objektumokat anélkül, hogy állandó adatokhoz kellene hozzáférnie az egységtesztek futtatása közben.

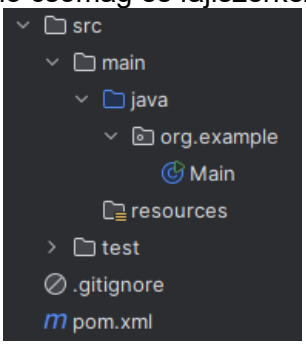
Például az @Autowired annotáció használható függőségek automatikus beinjektálására a Spring komponensekbe. Ez különösen hasznos lehet azoknál a teszteknél, ahol a függőség valódi megvalósítását kell beilleszteni a szolgáltatásba vagy a vezérlőbe.

A JUnit korábbi verzióiban a tesztkonstruktorok vagy metódusok nem rendelkeztek paraméterekkel. A JUnit 5 egyik fő változásaként mostantól mind a tesztkonstruktorok, mind a metódusok rendelkezhetnek paraméterekkel. Ez nagyobb rugalmasságot tesz lehetővé, és lehetővé teszi a függőségi injektálást a konstruktorok és metódusok számára.

4. A Dependency Injection bemutatása egy részletes példán keresztül

Az előző fejezetben megismertük a Dependency Injection alapjait rövid példákon keresztül. Most nézzük meg, hogy működik ez a valóságban egy részletes példa segítségével, ahol annotáció alapú konfigurációt használunk.

Készítsünk egy Java Maven projektet egy integrált fejlesztői környezettel (pl. IntelliJ, Eclipse, NetBeans). A Maven projektnek megfelelő csomag és fájlszerkezet jön létre (*2. ábra*):



2. ábra. A Maven projekt csomag és fájlszerkezete

Maven dependency

Az IoC-hez szükséges a *spring-context* dependency, amit a *Maven repository*-ból fogunk letölteni a *pom.xml* fájl segítségével. A pom.xml fájlhoz adjuk a *spring-context* legfrissebb változatát, ami jelenleg a 6.2.0 A legfrissebb változatot megkereshetjük a https://mvnrepository.com/artifact/org.springframework/spring-context oldalon.

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.2.0</version>
    </dependency>
</dependencies>
```

Csomagok készítése

Készítsük el a következő csomagokat a kiinduló org.example csomagba: *szolgáltatások, fogyasztó, konfiguráció, teszt*. A következő osztályokat ezekbe a csomagokba fogjuk elkészíteni.

Szolgáltatás osztályok

Tegyük fel, hogy e-mail üzenetet és Twitter üzenetet szeretnénk küldeni a felhasználóknak. A Dependency Injection-hoz szükségünk van egy alaposztályra a szolgáltatásokhoz. Ehhez készítünk egy *ÜzenetSzolgáltatás* interfészt egyetlen metódus deklarációval az üzenetküldéshez:

```
package org.example.szolgáltatások;
public interface ÜzenetSzolgáltatás {
    boolean üzenetKüld(String üzenet, String cím);
}
```

Ez alapján implementálunk osztályokat e-mailek és Twitter üzenetek küldésére:

```
package org.example.szolgáltatások;
public class EmailSzolgáltatás implements ÜzenetSzolgáltatás {
    public boolean üzenetKüld(String üzenet, String cím) {
        System.out.println("Email küldve: "+ cím + " Üzenet="+ üzenet);
        return true;
    }
}

package org.example.szolgáltatások;
public class TwitterSzolgáltatás implements ÜzenetSzolgáltatás {
    public boolean üzenetKüld(String üzenet, String cím) {
        System.out.println("Twitter üzenet küldve: "+ cím + " Üzenet="+ üzenet);
        return true;
    }
}
```

Most, hogy szolgáltatásaink készen állnak, áttérhetünk a szolgáltatást igénybe vevő komponens osztályokra.

Komponens osztályok

Írjunk fogyasztói osztályt a fenti szolgáltatásokhoz, amiben az *@Autowired* Spring annotációt használjuk, amiről már tudjuk, hogy lehetővé teszi a Spring számára, hogy feloldja és beinjektálja az együttműködő bean-t az osztályunkba.

```
package org.example.fogyasztó;
import org.example.szolgáltatások.ÜzenetSzolgáltatás;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Component;
@Component
public class Alkalmazás {
    private ÜzenetSzolgáltatás szolgáltatás;
    @Autowired
    public void setSzolgáltatás(ÜzenetSzolgáltatás üzenetSzolgáltatás){
        this.szolgáltatás =üzenetSzolgáltatás;
    }
    public boolean üzenetFeldolgoz(String üzenet, String cím){
        return this.szolgáltatás.üzenetKüld(üzenet, cím);
    }
}
```

Néhány megjegyzés az Alkalmazás fogyasztó osztállyal kapcsolatban:

- a *@Component* annotáció hatására, amikor a Spring framework megkeresi a komponenseket, ez az osztály komponensként lesz kezelve. A komponensek keresése futási időben történik
- a példában a Setter injektálás van megvalósítva

Konfiguráció annotációval

Meg kell írnunk egy konfigurációs osztályt, amelyet arra használunk, hogy a tényleges implementációs bean-t beinjektáljuk a *Component* osztályba.

```
package org.example.konfiguráció;
import org.example.szolgáltatások.EmailSzolgáltatás;
import org.example.szolgáltatások.ÜzenetSzolgáltatás;
import org.example.szolgáltatások.TwitterSzolgáltatás;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import java.util.Scanner;
@Configuration
@ComponentScan(value={"org.example.fogyasztó"})
public class DIKonfiguráció {
    @Bean
    public ÜzenetSzolgáltatás getÜzenetSzolgáltatás(){
        Scanner olvas = new Scanner(System.in);
        System.out.println("Adja meg a szolgáltatás típusát!");
        System.out.println("1: Email");
        System.out.println("2: Twitter");
        int típus = olvas.nextInt();
        if(típus==1)
            return new EmailSzolgáltatás();
        else
            return new TwitterSzolgáltatás();
    }
}
```

Néhány fontos jellemző a fenti osztályhoz kapcsolódóan:

- A *@Configuration* annotációval adjuk meg a Springnek, hogy ez egy konfigurációs osztály.
- A *@ComponentScan* annotáció a @Configuration annotációval együtt arra szolgál, hogy meghatározza azokat a csomagokat, amelyekben *Component* osztályokat kell keresni.
- *A @Bean* annotáció segítségével tudatjuk a Spring keretrendszerrel, hogy ezt a metódust kell használni a bean implementáció Component osztályokba való beinjektálására.
- Billentyűzetről választjuk ki a szolgáltatás típusát futási időben.

Írjunk egy egyszerű programot az annotáció alapú Spring *Dependency Injection* példánk tesztelésére

```
package org.example.teszt;
import org.example.konfiguráció.DIKonfiguráció;
import org.example.fogyasztó.Alkalmazás;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class KliensAlkalmazás {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(DIKonfiguráció.class);
        Alkalmazás alkalmazásom = context.getBean(Alkalmazás.class);
        alkalmazásom.üzenetFeldolgoz("Jó napot Ferenc!", "ferenc@mail.com");
        context.close();
    }
}
```

Az *AnnotationConfigApplicationContext* az *AbstractApplicationContext* absztrakt osztály implementálása, és a szolgáltatások automatikus bekötésére szolgál a komponensekbe annotációk használatával. Az AnnotationConfigApplicationContext konstruktor azt az osztályt használja argumentumként, amely a bean implementáció komponensosztályokba való beinjektálására szolgál.

A getBean(Class) metódus visszaadja a Component objektumot, és a konfigurációt használja az objektumok automatikus összeépítésére.

context.close(): a Context objektumok erőforrás-igényesek, ezért be kell zárnunk őket, ha végeztünk velük.

Futtatáskor a következő kimenetet kapjuk pl. az Email szolgáltatás választásakor:

```
Adja meg a szolgáltatás típusát!
1: Email
2: Twitter
1
Email küldve: ferenc@mail.com Üzenet=Jó napot Ferenc!
```

Az objektumorientált programozásban eddig is megszokott gyakorlat, hogy egy metódus paramétere egy interfész, és minden olyan osztály lehet bemeneti paraméter, ami implementálja az interfészt. De amíg ott a main(…) metódusból kiinduló vezérlésben adjuk meg, hogy milyen osztállyal implementáljuk az interfészt, addig a Dependency Injecton-nél a main(…) metódusból kiinduló vezérlésben nem derül ki, hogy melyik osztállyal implementáljuk az interfészt. Ezt a DIKonfiguráció osztályban adjuk meg, ami független a főprogramtól és később is bármikor módosítható.

Összefoglalás

A cikkben bemutattuk a Java Spring keretrendszer előnyös tulajdonságait és fő moduljait, valamint példákon keresztül két fő összetevőjét az Inversion of Control-t és a Dependency Injection-t. Láttuk, hogy a keretrendszer előnyös tulajdonságai segítségével támogatást nyújt robusztus Java alkalmazások egyszerű és gyors fejlesztéséhez, így egy jó megoldás a vállalati használatra kész alkalmazások felépítéséhez.

Irodalomjegyzék

[1] C. Basu and K. Singh, "The Role of Spring Boot in the Era of Cloud Native Java," 2019 IEEE International Conference on Cloud Computing (CLOUD), Milan, Italy, 2019, pp. 400-404, doi: 10.1109/CLOUD.2019.00069.

[2] Alexandru Marius Bonteanu, Cătălin Tudose: Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA, Hibernate, Spring Data JPA. MDPI Journal, Applied Sciences, Basel, Switzerland, 2024 https://doi.org/10.3390/app14072743

[3] Choma Dominik , Chwaleba Kinga , Dzieńkowski Mariusz: The efficiency and reliability of backend technologies: Express, Django, and Spring Boot, Wydawnictwo Politechniki Lubelskiej, ul. Nadbystrzycka 36C/309D, 20-618 Lublin, 2023. http://dx.doi.org/10.35784/iapgos.4279
[4] Downey, T. (2021). Spring Framework. In: Guide to Web Development with Java. Texts in Computer Science. Springer, Cham. https://doi.org/10.1007/978-3-030-62274-9_4 ISBN: 978-3-030-62273-2, 2021
[5] Michal Gajewski; Wojciech Zabierowski: Analysis and Comparison of the Spring Framework and Play Framework Performance, Used to Create Web Applications in Java, 2019 IEEE XVth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), ISBN: 978-1-7281-4030-8 DOI: https://doi.org/10.1109/MEMSTECH.2019.8817390
[6] M. S. Islam et al., "A Comparative Study on Implementing Microservices Architecture with Spring Boot and ASP.NET Core," 2020 International Conference on Computer, Communication, Chemical, Materials and Electronic Engineering (IC4ME2), Khulna, Bangladesh, 2020, pp. 1-6, 2020, doi: 10.1109/IC4ME248511.2020.9292276
[7] Rod Johnson et al.: Spring Framework Reference Documentation https://docs.spring.io/spring-framework/docs/3.2.17.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf
[8] S. S. M. M. Kamal, M. A. Hossain and M. A. R. Amin, "Performance Evaluation of Spring Boot for Scalable Web Applications," 2018 4th International Conference on Advances in Electrical Engineering (ICAEE), Dhaka, 2018, pp. 1-6, doi: 10.1109/ICAEE.2018.8629004
[9] M. Kim, Q. Xin, S. Sinha, and A. Orso, “Automated test generation for REST APIs: no time to rest yet,” Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022), Association for Computing Machinery, New York, NY, USA, 2022, pp. 289–301, doi: https://doi.org/10.1145/3533767.3534401
[10] Darshak Mota, Neel Zadafiya, Jinan Fiaidhi: Spring Framework for Testing, TechRxiv. April 11, 2020. https://doi.org/10.36227/techrxiv.12094230.v1
[11] Maya Retno Ayu Setyautami, Daya Adianto, Ade Azurat and Eko Kuswardono Budiardjo: A PROPOSED JAVA WEB FRAMEWORK TO SUPPORT SOFTWARE PRODUCT LINE ENGINEERING, ICIC Express Letters ICIC International 2024 ISSN 1881-803X Volume 18, Number 3, March 2024 pp. 293–301, DOI: 10.24507/icicel.18.03.293
[12] S. Singh, R. Singh and S. S. Saini, "A Comparative Study of Spring Boot and Node.js for Developing Microservices," 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC), Erode, India, 2019, pp. 605-610, doi: 10.1109/ICCMC.2019.8712615
[13] Wicha, M., & Pańczyk, B. (2023). Performance analysis of REST API technologies using Spring and Express.js examples. Journal of Computer Sciences Institute, 29, 352–359. https://doi.org/10.35784/jcsi.3796
[14] C. Zepeda-Núñez et al., "Performance evaluation of Spring Boot and Vert.x in microservices," 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA), Abu Dhabi, United Arab Emirates, 2019, pp. 1-8, doi: 10.1109/AICCSA47632.2019.8812115.