Запознавање со Spring рамката

Веб програмирање

Што е Spring?

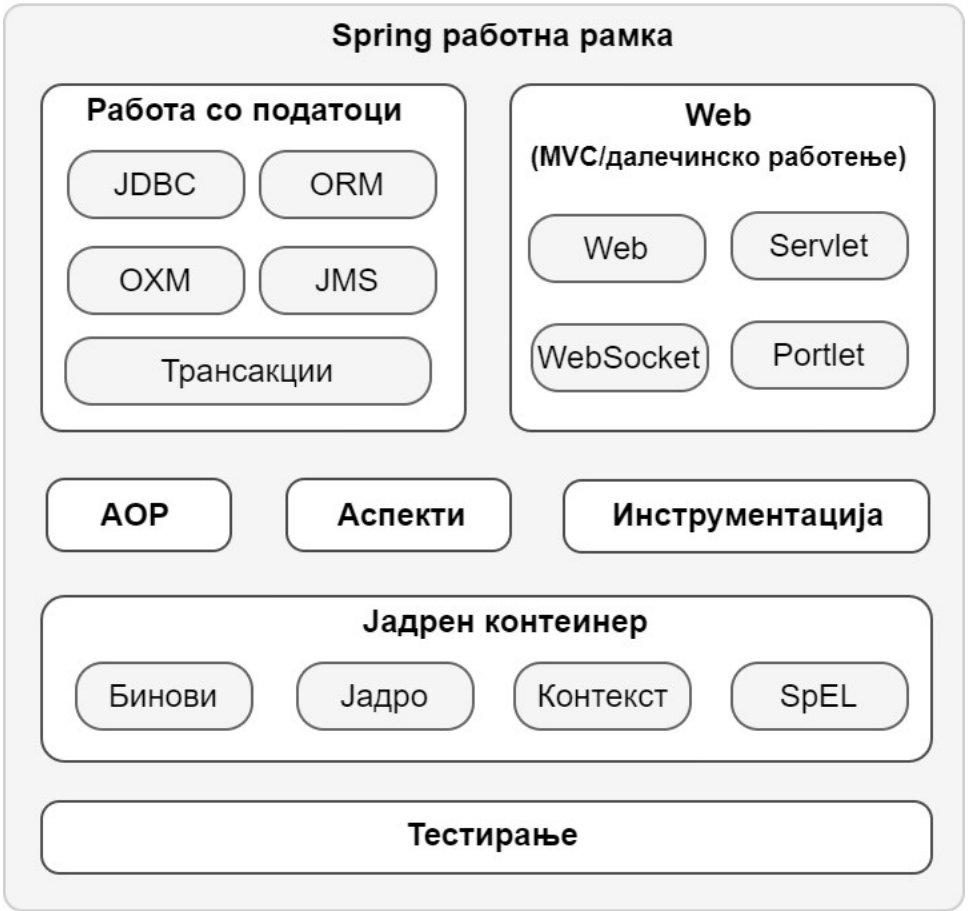
- Spring рамката е дизајнирана за да го олесни креирањето на комплексни јава апликации
  - Брз и флексибилен развој
  - Најдобри практики за градење на сигурни и одржливи апликации
  - Абстрахира голем дел од деталите
- Отворен код (open source)
- Голема и активна заедница
  - континуирана подршка за различни реални сценариа
  - помага Spring рамката да еволуира континуирано и да ги подржува потребите на реалните апликации

Зошто Spring?

- Примарна цел е да се намалат зависностите, па дури и да се воведат негативни зависности
- Дел од причината што е толку брзо усвоена
- Докажано е дека кодот на Spring е добро структуриран (можеби и најдобро)
- Без циклуси на зависност

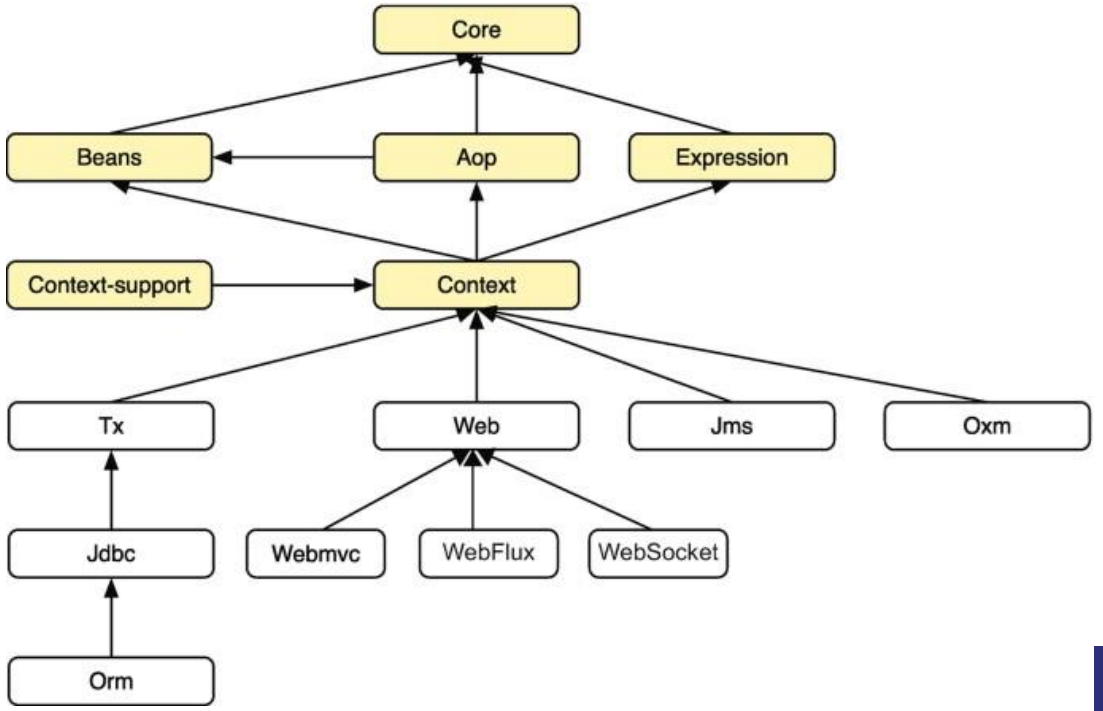
Spring решенија

- Решенија кои ги адресираат најзначајните J2EE предизвици:
  - Развој на веб апликации (MVC)
  - Enterprise Java Beans (EJB, JNDI)
  - Пристап до базата на податоци (JDBC, ORM)
  - Управување со трансакции (JTA, Hibernate, JDBC)
  - Далечински пристап (Web Services, RMI)
- Секое решение се надоврзува на основната архитектура
- Решенијата поттикнуваат интеграција, тие не го измислуваат тркалото повторно



Што може да направиме со Spring?

- Веб апликации
- Микросервиси
- Реактивни апликации
- Процесирање на настани во реално време
- Процесирање на големи количини на податоци



Модули од Spring рамката

Spring Boot

Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.

Spring Framework

Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.

Spring Cloud Data Flow

Provides an orchestration service for composable data microservice applications on modern runtimes.

Spring Security

Protects your application with comprehensive and extensible authentication and authorization support.

Spring Data

Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.

Spring Cloud

Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.

Spring Session

Provides an API and implementations for managing a user's session information.

Spring Integration

Supports the well-known Enterprise Integration Patterns through lightweight messaging and declarative adapters.

Модули од Spring рамката

Spring HATEOAS

Simplifies creating REST representations that follow the HATEOAS principle.

Spring REST Docs

Lets you document RESTful services by combining hand-written documentation with auto-generated snippets produced with Spring MVC Test or REST Assured.

Spring for Android

Provides key Spring components for use in developing Android applications.

Spring LDAP

Simplifies the development of applications that use LDAP by using Spring's familiar template-based approach.

Spring Batch

Simplifies and optimizes the work of processing high-volume batch operations.

Spring AMQP

Applies core Spring concepts to the development of AMQP-based messaging solutions.

Spring Flo

Provides a JavaScript library that offers a basic embeddable HTML5 visual builder for pipelines and simple graphs.

Spring for Apache Kafka

Provides Familiar Spring Abstractions for Apache Kafka.

Spring подршка за веб апликации

- Подржува различни сценарија за стартување на веб апликации
  - Стартување на апликации во рамки на веб апликациски сервер кој е независен од апликацијата
  - Стартување на апликацијата со вгнезден (embedded) сервер
  - Стартување на апликација без користење на сервер (executable jar)

Дизајн филозофија

- Избор на секое ниво
- Подршка на различни перспективи
- Одржување на компатибилноста наназад
- Грижа за дизајнот на интерфејсите (API)
- Поставување на високи стандарди за квалитет на код

SOLID принципи

- **Single Responsibility (Единствена одговорност)**
  - Секоја класа треба да има само една одговорност (само една причина за промена)
    - Помалку сценарија за **тестирање**
    - **Помало преплетување** – Помалку функционалности => помалку зависности
    - **Организација**– Помали, добро организирани класи се полесни за пребарување
- **Open/Closed (Отворени за проширување, затворени за менување)**
  - Модификација на постоечки код => потенцијални нови багови
    - Ако работи, не го менуваме ☺
- **Liskov Substitution**
  - Ако класата **А** е **подтип** на класата **Б**, треба да можеме да го **замениме Б со А** без да го нарушиме однесувањето на нашата програма
- **Interface Segregation (Поделба на интерфејсите)**
  - Поголемите интерфејси треба да се поделат на повеќе помали
- **Dependency Inversion (Инверзија на контролата)**
  - Се однесува на раздвојување на софтверски модули
  - Наместо модулите на високо ниво зависни од модулите на ниско ниво, и двата ќе зависат од апстракции (најчесто интерфејси)

https://www.baeldung.com/solid-principles

Вметнување на зависности

Што се зависности?

```
public class NewsServiceImpl implements NewsService {
    private final NewsRepository newsRepository;
    protected NewsCategoryRepository categoryRepository;
```

- Својствата на класите
  - `newsRepository`
  - `categoryRepository`
- Како да видиме од што зависи некоја класа?
  - Зависи од модулите наведени со **import** изразите

Вметнување на зависности

Како може да вметнеме зависност?

- Преку конструктор (constructor injection)
- Преку поставување на својство (property injection)
- Преку метод за поставување вредност (setter injection)
- Преку метод за креирање инстанца (factory method injection)

```
public class NewsServiceImpl implements NewsService {
    private final NewsRepository newsRepository;
    protected NewsCategoryRepository categoryRepository;

    public static NewsService create(NewsRepository newsRepository,
                                     NewsCategoryRepository categoryRepository) {
        NewsServiceImpl instance =  new NewsServiceImpl(newsRepository,
         ↪  categoryRepository);
        // do some further initialization or configuration
        return instance;
    }

    public NewsServiceImpl(NewsRepository newsRepository,
                           NewsCategoryRepository categoryRepository) {
        this.newsRepository = newsRepository;
        this.categoryRepository = categoryRepostiory;
    }

    public setCategoryRepository(NewsCategoryRepository categoryRepository)
     ↪  {
        this.categoryRepository = categoryRepository;
    }

    // the rest of the code is omitted
}
```

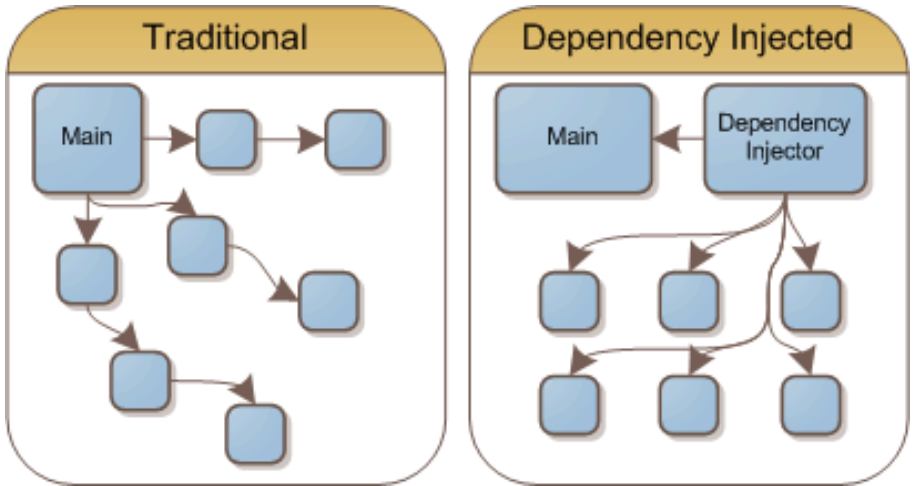
Вметнување на зависности
Поедноставно тестирање

- Не ги користиме вистинските имплементации на зависностите
  - Не се поврзуваме со база
    - Нема потреба од инсталација на база за тестот
    - Нема потреба од иницијализација на податоци
  - Тестот е повторлив
  - Тестот се извршува брзо
- Ја проверуваме бизнис логиката од интерес, независно од зависностите

```
public class NewsServiceImplTest {

    NewsService instance;

    @Before
    public init() {
        // instantiate dependencies
        NewsRepository nr=Mockito.mock(NewsRepository.class);
        NewsCategoryRepository
         ↪  cr=Mockito.mock(NewsCategoryRepository.class);

        // constructor injection
        instance = new NewsServiceImpl(nr, cr);

        // property injection
        instance.categoryRepository = cr;

        // setter injection
        instance.setCategoryRepository(cr);

        // factory method injection
        NewsService factoryInstance = NewsServiceImpl.create(nr, cr);

    }
}
```

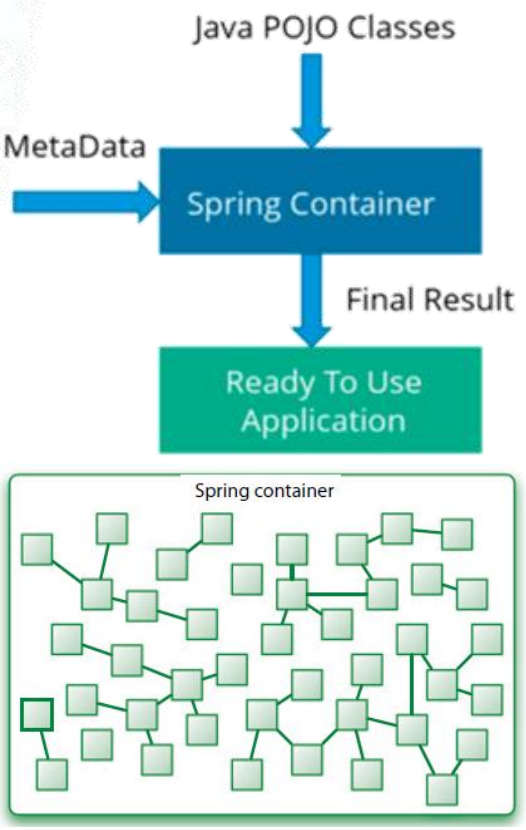
Инверзија на контролата (Inversion of Control – IoC)

- Во реални апликации често имаме **десетици**, па и **стотици** објекти кои треба да ги креираме и да им ги вметнеме зависностите
  - здодевен и долготраен процес
- **Инверзија на контролата** овозможува **декларативно означување** на **зависностите** и нивно **имплицитно поврзување** во процесот на креирање на објектите
  - **Spring** рамката го поедноставува процесот на креирање објекти со инверзија на контролата



Контејнер за инверзија на контролата

- Spring container is the heart of the Spring Framework, and performs the following tasks:
  - Instantiating the beans
  - Wiring the beans together
  - Configuring the beans
  - Managing the bean's entire life-cycle
- The container receives metadata from either
  - an XML file,
  - Java annotations, or
  - Java code
- IoC adds the flexibility and control of application, and provides a central place of configuration management for Plain Old Java Objects (POJO) of our application.



Имплементација на Spring IoC контејнер

- Секоја инстанца од класа која е зачувана во Spring IoC контејнерот се нарекува **bean**
- **BeanFactory**
  - Ги дефинира напредни механизми за конфигурација, кои овозможуваат управување со сите типови на објекти.
- **ApplicationContext**
  - Изведен од BeanFactory и ги додава следните својства:
    - Полесна интеграција со функционалностите за **аспект ориентирано програмирање (AOP)** од Spring рамката
    - Справување со ресурси за **интернационализација**
    - **Објавување на настани** (Event publishing)
    - Воведува подршка за **специфични контексти**, како што е WebApplicationContext, кој се користи кај веб апликациите
- **BeanFactory** ги дефинира механизмите за **конфигурација** и основните функционалности на Spring рамката, додека пак **ApplicationContext** додава специфични функционалности кои се потребни за бизнис апликациите

Конфигурација на Spring IoC контејнер

- Spring контејнерот е тој што најпрво треба да ги **пронајде компонентите** и да **креира бинови** од нив.
- Најчести начини за регистрација на бинови во Spring се
  - XML конфиуграција
    - Експлицитно ги наведува класите кои ќе бидат вклучени во апликацискиот контекст и нивните зависности во XML датотека која се користи за креирање на ApplicationContext
  - Јава конфигурација
    - Експлицитно се наведуваат класите и нивните зависности, но во јава код
  - Скенирање на анотациите на компонентите (component scan)
    - Се користи механизмот за скенирање на класи анотирани со анотации кои се изведени од **@Component**, и потоа се креираат бинови од нив
      - @ControllerAdvice, @Controller, @Repository, @JsonComponent, @TestComponent, @Service, @Configuation

XML конфигурација

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- scan the content of a package-->
    <component-scan package="mk.ukim.finki.wp.repository"/>
    <!-- services -->
    <bean id="newsService"
          class="mk.ukim.finki.wp.service.NewsServiceImpl">
        <constructor-arg index="0" ref="newsRepository"/>
        <constructor-arg index="1" ref="categoryRepository"/>
    </bean>

    <!-- enable spring annotation processing for DI -->
    <context:annotation-config/>

    <!-- more bean definitions can go here -->

</beans>
```

Јава конфигурација

```
@Configuration
@ComponentScan("mk.ukim.finki.wp.repository")
public class AppConfig {

    // If not defined explicitly,
    // the bean id is the name of the method
    @Bean
    public NewsService newsService(
                    NewsRepository newsRepository,
                    NewsCategoryRepository categoryRepository) {
        return new NewsServiceImpl(newsRepository, categoryRepository);
    }
}
```

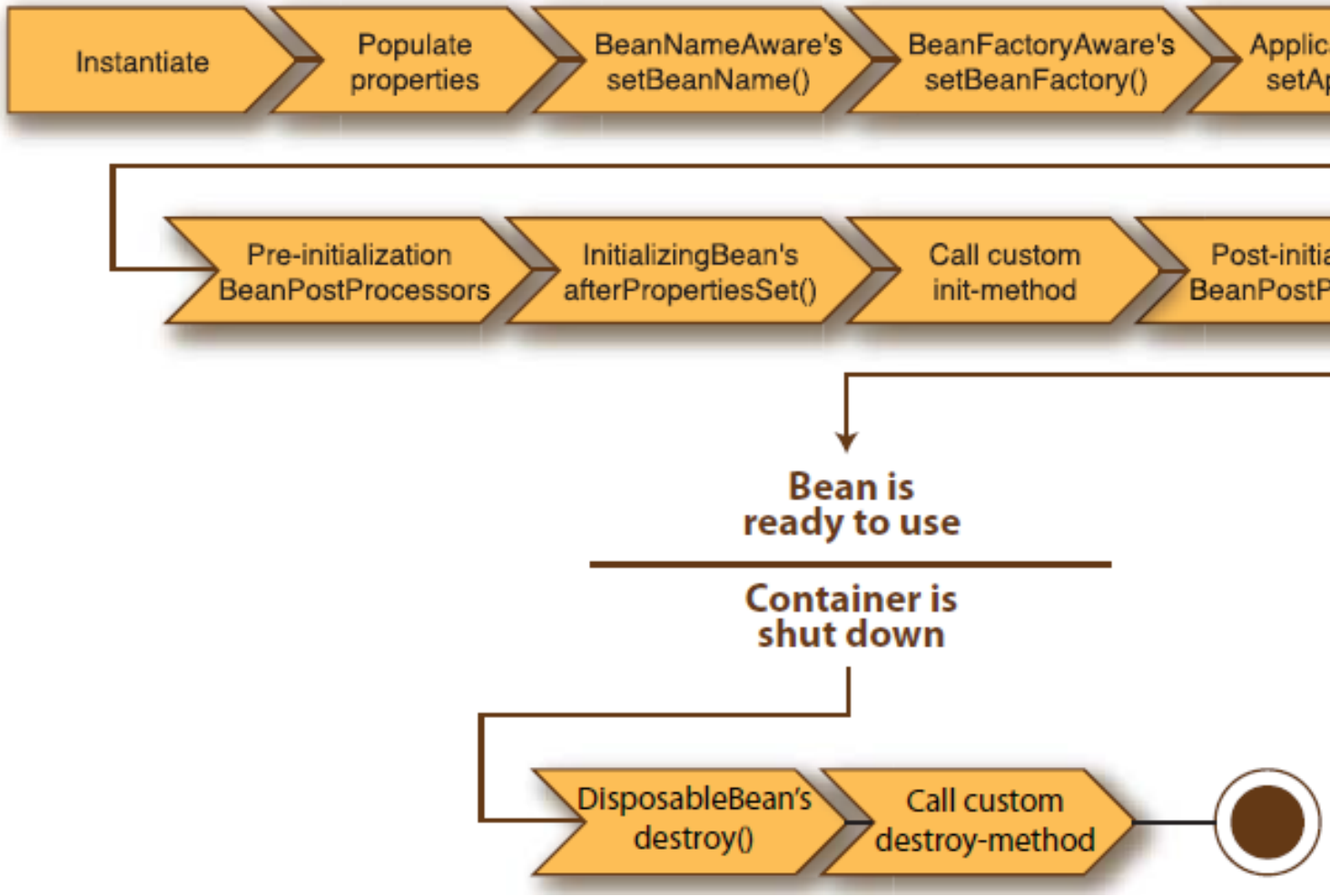
Креирање на Spring IoC контејнр со XML и јава конфигурација

```
public static void main(String[] args) {
  ApplicationContext javaCtx = new AnnotationConfigApplicationContext(
                                                    AppConfig.class);

  ApplicationContext xmlCtx = new ClassPathXmlApplicationContext(
                                                    "app-config.xml");

  // we can also use xmlCtx for finding a bean that implements
  // the NewsService interface
  NewsService newsService = javaCtx.getBean(NewsService.class);
  // newsService.doStuff();
}
```

Разлика помеѓу @Bean и @Component

- @Component
  - Анотација наменета за означување на **класи** за кои Spring IoC контејнерот имплицитно ќе креира и регистрира бинови. Овие анотации се процесираат во рамки на скенирањето на компонентите (component scan).
- @Bean
  - Анотација наменета за означување на **методи** чии резултат ќе биде експлицитно креиран бин кој ќе се регистрира во рамките на IoC контејнерот.

Животен циклус на биновите



Автоматско вметнување на зависности (Autowiring)

- Автоматското вметнување на зависности
  - Вметнување на зависности анотирани со **@Autowired** анотацијата,
  - Вметнување на аргументи на **конструкторот** за компоненти од кои се креираат бинови и
  - Вметнување на аргументи на методите анотирани со **@Bean**

Модели за автоматско вметнување на зависности

- no:
  - Не се користи автоматско вметнување на зависностите. Сите зависности мора експлицитно да се вметнат. Ова е предефинираниот модел.
- byName:
  - Во овој модел се вметнуваат зависностите според името на бинот. Во овој случај, името на својството и името на бинот треба да бидат исти.
- byType:
  - Во овој модел се вметнуваат зависностите според типот, при што е дозволено својството и бинот да имаат различни имиња.
- constructor:
  - Во овој модел се вметнуваат зависностите со повик на конструкторот на класата:
- autodetect:
  - Во овој модел, прво се прави обид за вметнување на зависностите со constructor моделот. Доколку не успее, тогаш се прави обид со byType моделот.

Правила за вметнување на зависности во спринг

- Ако нема ниту една инстанца од типот во IoC контејнерот, ќе биде фрлена NoSuchBeanException грешката
- Ако има само една инстанца од типот во IoC контејнерот, ќе биде вметната
- Ако има повеќе инстанци од типот во IoC контејнерот
  - Ако има **@Qualifier("beanName")** анотација на својството, ќе се вметне инстанцата регистрирана со името "beanName"
  - Ако имаме инстанца регистрирана со **името на променливата**, ќе се вметне таа инстанца
  - Ако има инстанца анотирана со **@Primary**, ќе се вметне таа инстанца
  - Ќе се фрли исклучок **NonUniqueBeanException**

Зошто се препорачува вметнување на зависности преку конструктор?

- Јасно се идентификуваат зависностите.
- Не постои начин да се заборави зависност при тестирање
  - Не би можело да се креира објектот без проследување на вредност за зависноста во конструкторот.
- Не е потребна рефлексија за да се постават зависностите.
  - Може да се користат симулирани имплементации и експлицитно да се вметнат преку конструкторот за полесно тестирање.
- Со секоја нова зависност треба да модифицираат тестовите за компонентата.
  - Придонесува да имаме тековни тестови за имплементацијата.
- Голем број на зависности ги прави конструкторите тешки за користење.
  - **Ова е индикација дека се нарушуваат single responsibility и interface segregation принципите од SOLID шаблонот.**
  - Во вакви случаи е препорачливо иницијалната компонента да се подели во повеќе помали компоненти со јасни и специфични задачи.
- Се препорачува зависностите да бидат final, што помага за робусност и сигурност на нишките (thread-safety)

@Scope во Spring

- **@Scope** анотацијата во Spring се користи за да се дефинира животниот век на еден бин во контекстот на апликацијата.
- Оваа анотација има за цел да укаже **колку долго ќе опстане бинот во контекстот**, кога и како ќе се креира и уништува.

- Типови на @Scope
  - **Singleton - @Scope("singleton")**
    - Стандардната вредност.
    - Бинот е еденствен во рамките на целиот контекст на апликацијата.
    - Креира се еднаш, ако не е специфицирано поинаку.
  - **Prototype - @Scope("prototype")**
    - Бинот се креира за секоја барање.
    - Не е еденствен и може да има повеќе инстанци во различни делови на апликацијата.
  - **Request - @Scope("request")**
    - Бинот е важечки само за време на HTTP Request.
    - Креира се при секое барање на HTTP Request.
  - **Session - @Scope("session")**
    - Бинот е важечки само за време на HTTP Session.
    - Креира се при почетокот на нова HTTP Session.

Вметнување на вредности со @Value

- @Value анотацијата во Spring е моќна алатка за вметнување на конфигурациски вредности во биновите.
- Што е @Value анотацијата?
  - @Value анотацијата во Spring овозможува вметнување на вредности во биновите од различни извори.

```
@Value("${app.api.url}")
private String apiUrl;
```

  - Се користи за вчитување вредности од property фајлови, системски променливи, аргументи на командната линија итн.
- Предности на користење @Value
  - Флексибилност: Овозможува динамичко конфигурирање на апликацијата без потреба за промена во кодот.
  - Лесност на управување со конфигурацијата: Вредностите можат лесно да се менуваат без рестартување на апликацијата.
- Овој механизам зголемува флексибилност и управување со апликацијата, што го прави идеален избор за конфигурација во Spring проекти.

Надворешни конфигурации

- Надворешни конфигурации во Spring се техники што овозможуваат конфигурација на апликацијата преку внешни извори како што се **property фајлови** (пр. resources/application.properties), **environment** променливи, **конзолни аргументи**, и сл.
- Ова овозможува динамичка и безбедна конфигурација без потреба за промена во изворниот код.
- Предности на Надворешните Конфигурации
  - Флексибилност: Можете да ја менувате конфигурацијата без рестартување на апликацијата.
  - Безбедност: Осетливи информации како лозинки можат да се чуваат безбедно во property фајлови.
- Надворешните конфигурации во Spring се силно средство за **динамичка** и **безбедна конфигурација** на апликациите.
- Апликацијата станува по-флексибилна и по-лесна за управување.

Извори на Надворешни Конфигурации во Spring

- **Глобални поставувања на Devtools** во вашиот домашен директориум (~/.spring-boot-devtools.properties кога devtools е активен).
- **@TestPropertySource** анотации во вашите тестови.
- **properties** атрибут во вашиот **тест**. Достапно во @SpringBootTest и тест анотациите за тестирање на одреден дел од вашата апликација.
- Аргументи од **командната линија**.
- Properties од **SPRING_APPLICATION_JSON** (вградени JSON во околинска променлива или системска својство).
- **ServletConfig** init параметри.
- **ServletContext** init параметри.
- **Java системски својства** (System.getProperties()).
- **OS environment** променливи.
- Апликациски својства специфични за **профилот надвор од пакуваниот JAR** (application-{profile}.properties и YAML варијанти).
- Апликациски својства специфични за **профилот пакувани во JAR архивата** (application-{profile}.properties и YAML варијанти).
- Апликациски **својства надвор од пакуваниот JAR** (application.properties и YAML варијанти).
- Апликациски **својства пакувани во JAR архивата** (application.properties и YAML варијанти).
- **@PropertySource** анотации во **@Configuration** класите.
- **Default својства** (поставени со поставување SpringApplication.setDefaultProperties).

Извор:
https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/boot-features-external-config.html

@Profile во Spring

- **@Profile** анотацијата во Spring овозможува конфигурација на биновите во зависност од активниот профил на апликацијата.
  - Бинови во профил

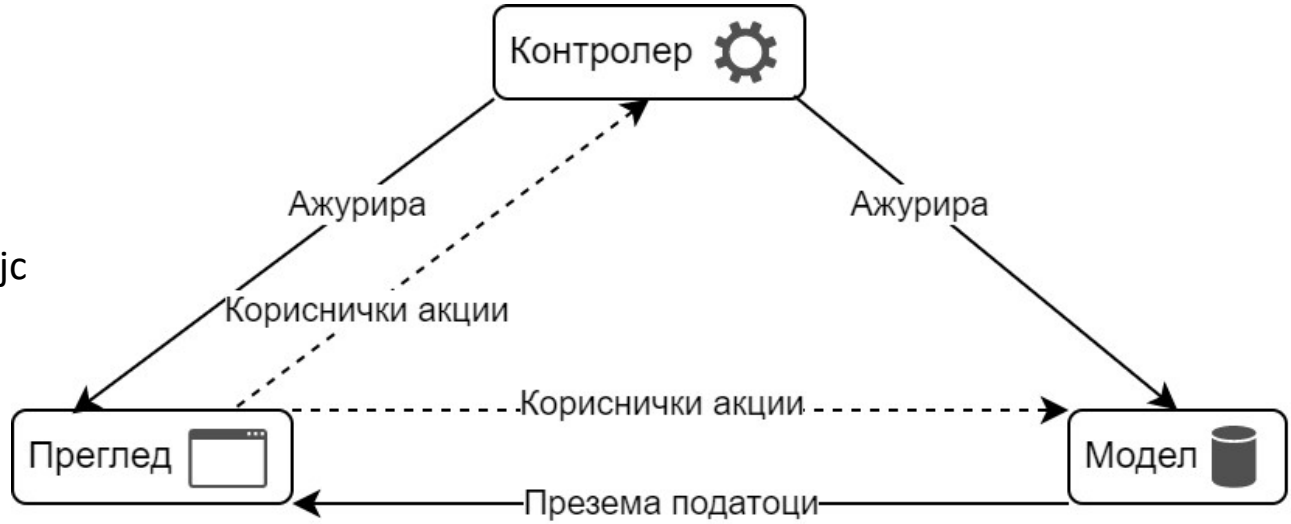
    ```
    @Profile("development")
    @Component
    ```
  - Активирање на профил во application.properties
    - spring.profiles.active=development
- Ова овозможува креирање на различни конфигурации за различни околности (профили) без потреба за промена во изворниот код.
- Предности на @Profile
  - Разграничување на Околините: Овозможува конфигурација специфична за околината.
  - Лесно управување со конфигурациите: Олеснува поддржувањето на различни околини без комликација.
- Оваа техника овозможува голема флексибилност и елегантно управување со конфигурациите во различни околини.

Клучни шаблони кои се користат во Spring

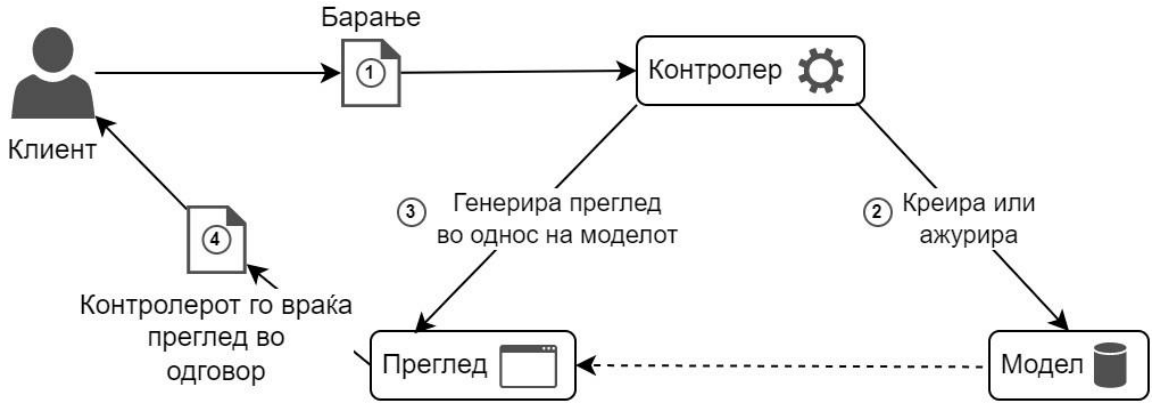
- MVC шаблон
- MVC шаблон за веб апликации
- Слоевита архитектура
- Слоевит MVC шаблон за веб апликации

Модел – Преглед – Контролер (MVC) шаблон

- Иницијално креиран за desktop апликации
- Секоја компонента има посебна одговорност и улога
  - поголема читливост на кодот
  - полесно одржување и тестирање
  - Модел
    - податоците кои се прикажуваат во прегледот
  - Преглед
    - дефиниции на елементите од корисничкиот интерфејс
  - Контролер
    - справување со акциите на корисниците
    - најчесто го менува моделот
    - може да го промени и прегледот кој се користи (навигација помеѓу различни прегледи)
- MVC шаблонот подразбира циклично извршување на акции и исцртување на прегледот



MVC шаблон за веб апликации



- Стандардниот MVC шаблон не е применлив во веб околините, заради природата на HTTP протоколот.
- Крајниот корисник комуницира директно со контролерот преку HTTP протоколот
- Улогата на моделот останува непроменета
- **Контролерот** е одговорен за:
  - извлекување на податоците добиени во HTTP барањето
  - креирање, вчитување и зачувување на моделот
  - извршување на бизнис логиката и промена на моделот
  - селекција на прегледот
  - генерирање на содржината на HTTP одговорот со комбинирање на селектираниот поглед и модифицираниот модел
- **Прегледот** најчесто е темплејт кој се користи за генерирање на корисничкиот интерфејс.

Слоевита архитектура

- Контролерот во MVC шаблонот за веб апликациите има голем број на одговорности.
  - нарушува поголем дел од SOLID практиките
    - Нема една одговорност на компонентата
    - Отвореност за проширување - Затвореност за менување,
      - целата логика е на исто место => промена директно во контролерот.
- Организацијата на кодот се практикува воведување на слоеви
  - Абстракција на однесувањето
  - Подолба на одговорностите
  - Лесна замена на компоненти од еден слој со нови понапредни и поефикасни компоненти
  - Поедноставно тестирање преку симулирање на повиците од слојот кој се користи

Најчести слоеви кај веб апликации

- Модел
- Кориснички интерфејс (преглед)
- Презентациски слој (веб слој)
- Сервисен слој (Service)
- Слој за податочен пристап (Repository)

Слоевит MVC шаблон за веб апликации