Spring Base vs pure Java Examples

במסמך הבא ניתן למצוא מדריך לשתי דוגמאות ספרינג הגדולות הנמצאות בפרויקט.
המדריך יתאר את היתרונות של השימוש בספרינג, הפשטות, העזרה למפתח להתמקד אך ורק בלוגיקה העסקית ולא במניהול משאבי המערכת ויצירת אובייקטים מורכבים.

הקונספט הראשי של ספרינג, הוא לרכז במקום אחד את כל תהליך יצירת האובייקטים ולייצר אותם בזמן עליית המערכת ולא בזמן ריצה.

*במהלך המדריך נעבור על קבצי הקונפיגורציה ועל כל אחד מהבינים בקובץ הקונפיגורציה, נשווה את יצירת הבין ליצירת האובייקט הג׳וואי בפרויקט שאינו ספרינג ונסביר את היתרונות.
*ניתן למצוא את הדוגמאות תחת התיקייה Medium_Spring_Examples
ב GitHub שלנו וגם ב GoogleDrive.

דוגמא 1:

Wheather_Service

רקע לדוגמא:
מערכת קטנה שנותנת שירות על מזג האוויר, ניתן לדרוש מהמערכת לקבל את מצב התחזית היומי של עיר ספציפית בעולם, אפשר לבקש תחזית של כמה ימים קדימה עבור עיר בעולם, ואפשר לקבל תחזית יומית של כל הערים שהמערכת תומכת בהם.
המערכת מוציאה דוחות תחזית לפי בקשה, לכל דוח יש את חתימת הזמן שלו ומספר מזהה (דוחות לא נעלמים, הם ממשיכים להיות קיימים גם כשיוצא דוח חדש).

נתחיל במעבר על הבינים בקבצי הקונפיגורציה.
נשים לב שהפרויקט מחולק לשני packages שונים.
עבור כל package נממש קובץ קונפיגורציה נפרד, ונדאג שקובץ הקונפיגורציה הראשי יכיר וימשוך את התוכן של קובץ הקונפיגורציה המשויך ל-package.
חשוב לציין – במידה ומחלקה תחת ה-package weatherReports תרצה לכתוב בינים לקובץ קונפיגורציה והמחקלות תחתיה ירצו למשוך בינים מקובץ קונפיגורציה היא תהיה חייבת לכתוב אותם תחת הקובץ קונפיגורציה המשויך ל-package.

עבור קובץ הקונפיגורציה בתוך ה-package שנקרא WeatherReportsConfiguration קיים רק בין אחד, בעצם רשימה של WeatherDetails מאוד פשוטה.
כן, גם רשימות של אובייקט של המערכת נמיר לבינים, הסיבה היא קודם כל ביצועים, ספרינג הוכיחו שיצירת אובייקטים בצורה שהם מציעים יעילה יותר מיצירת אובייקטים רגילים בג׳אווה, כאשר המספרים נמוכים אין לכך משמעות אך במערכות גדולות ומרובות אובייקטים יש לכך משמעות רבה.
השימוש ברשימה זו של המחלקות ForecastDetails, WeatherSummary הינה עניין ידוע מראש, ולכן אין כל סיבה בזמן ריצה להתחיל ולייצר רשימה מסוימת. נכון שכרגע הלוגיקה של יצירת הרשימה הוא מאוד פשוט ומסתכם בשורה אחת בלבד, אך למען הקונספט, למען הרחבה עתידית של אתחול הרשימה שאולי יתארך ויורחב בהמשך, הוצאנו את היצירה של המשתנה הזה להיות בין, בצם להיות מנוהל תחת המסגרת של ספרינג. וכמובן שהמחלקות עצמן עכשיו מושכות את הבין הזה (אובייקט שכבר קיים ונוצר בזמן עליית המערכת ונשלח בעת דרישה).

נעבור לקובץ הקונפיגורציה השני הקובץ הראשי, תחילה נשים לב לשורה הבאה:
@Import(WeatherReportsConfiguration.class)
משמעות השורה היא לייבא מקובץ הקונפיגורציה המצוין את כל הבינים שלו, כלומר למרות שה-package ניהל בעצמו בצורה פנימית קובץ קונפיגורציה, עכשיו אנחנו רוצים לצרוך את התכולה שלו בקובץ קונפיגורציה הראשי.
בהמשך נראה איך זה משרת אותנו.

נתחיל עם הבין הראשון – weatherManager
נתבונן תחילה בפרויקט ללא ספרינג על המחלקה Main ונשים לב לדרך יצירת האובייקט ללא ספרינג.
המנג׳ר הוא אובייקט שתלוי באובייקט אחר – הסרוויס, והסרוויס תלוי באובייקט אחר שהוא הרפוסיטורי.
אנחנו מתמודדים עם מקרה של יצירת אובייקט מורכב, בפרויקט הלא ספרינגי עלינו להתחיל ולייצר מלמטה למעלה את האובייקטים ובכל פעם לבצע השמה או לשלוח בקונסטרקטור את האובייקט התלוי.
תסכימו איתי שבצורה שמייצרים את המנג׳ר בתוך המחלקה המיין היא צורה מסורבלת.
בסופו של דבר מבחינת הלוגיקה הנדרשת באותו רגע – אני רוצה לייצר מנג׳ר ולהריץ את המתודה run שלו.
נחזור שוב לפרויקט הספרינג.
נייצר כל אובייקט בנפרד, לכל אחד את הלוגיקה הנדרשת שלו.
וכעת יצירת המנג׳ר הופכת פשוטה, נייצר אותו ונשלח לו בקונסטרקטור את האובייקט שהוא צריך – הסרוויס (כמובן כבין), לוגיקת יצירת הסרוויס קובעת שיש צורך ברפוסיטורי.
בסופו של דבר במחלקה Main בפרויקט הספרינגי, נמשוך את האובייקט היחידי שמעניין אותנו – במקרה הזה WeatherManager הקוד שלנו לא ״יתלכלך״ בשורות קוד שמטרתן היחידה היא לאפשר לאובייקט WeatherManagr להיווצר ונוכל להתמקד אך ורק במודל העסקי – הרצת המתודה run של המנג׳ר.

נעבור לבין הבא – weatherService
אז כמו הבין הקודם גם הוא תלוי באובייקט אחר ולכן נזריק לו בקונסטרקטור קריאה של הבין הדרוש לו.
בנוסף נשים לב שתחת האנוטציה המצהירה עליו שהוא בין קיימת אנוטציה נוספת – Lazy.
המשמעות שלה – ישנם מספר מצבים בהם נרצה להגדיר לבין מסוים לא להיווצר בצורה הדיפולטית כמו שאר הבינים – בזמן עליית האפליקציה אלא להיווצר בצורה ״עצלנית״ בזמן ריצה אך ורק כשמגיעה דרישה לשימוש בבין הזה.
במקרה שלנו, הסיבה לכך היא שהמחלקה weatherService בקונסטרקטור שלה פונה למתודה סטטית של המחלקה Main.
המחלקה Main איננה בין, זוהי מחלקה רגילה של ג׳אווה.
המחלקות הרגילות של ג׳אווה אינן נטענות בזמן עליית המערכת אלא בזמן ריצת המערכת, ולכן ללא שימוש ב-lazy במקרה הזה, הבין weatherService ינסה להיטען בזמן עליית המערכת, ינסה לפנות למתודה הסטטית של המחלקה Main אך יגלה שבזמן הזה המחלקה הזאת עדיין לא קיימת וכמובן שאנחנו נעוף על שגיאת זמן ריצה.
השימוש ב-Lazy פותר כאן את העניין ובעצם גורם לזה ש-WeatherService יפנה למחלקה Main רק בזמן ריצה ובזמן שהאפליקציה כבר רצה והמחלקה קיימת.

weatherRepository
זהו בין פשוט שמייצר אובייקט מהמחלקה WeatherRepository, מוזרק ל-WeatherSerivce.

weatherInfoGenerator
זהו בין פשוט שמייצר אובייקט מהמחלקה weatherInfoGenerator.

weatherDetails, forecastDetails, weatherSummary
שלושת האובייקטים הנ״ל מיוצרים בשיטת prototype, כלומר בכל פעם שתתבצע משיכה של בין מהסוג של האובייקטים האלה ייווצר אובייקט חדש.
נשים לב לדבר מעניין, שלושת הבינים הללו משתמשים בבין סינגלטוני על מנת לייצר מספר מזהה, חשוב שהבין שיוצר את המספרים המזהים יהיה סינגלטוני על מנת להימנע ממצב ששני דוחות שונים מקבלים את אותו מספר מזהה.
יש לנו כאן מצב שבין מסוג prototype, שנוצר מספר רב של פעמים פונה לבין סינגלטוני.
יחס רבים ליחיד.
בנוסף נשים לב שביצירה של weatherSummary, foreastDetails שהם דוחות שמורכבים ממספר תחזיות, אנחנו משתמשים בבין המגיע מקובץ קונפיגורציה אחר, קובץ הקונפיגורציה ששייך ל-Package weatherReports.
על מנת להיות יכולים לפנות לבין שנמצא בקובץ קונפיגורציה שונה עלינו להגדיר data member של קובץ הקונפיגורציה הנדרש.
ולכן בתחילת המחלקה אנחנו משתמשים ב-Autowired על מנת להזריק אוטומאטית את קובץ הקונפיגורציה של ה-package ושמתמשים בבין מקובץ אחר.
בנוסף, מבצעים השמה של חותמת זמן יצירת הדוח.

דוגמא 2:

Hash_password

רקע לדוגמא:
מערכת קטנה לרישום משתמשים.
נניח שלחברה גדולה מסוימת יש שירות אינטרנטי גדול.
החברה החליטה שאת כל תהליך רישום המשתמשים היא תבצע במערכת נפרדת, שאינה קשורה לשירות שהחברה מספקת.
הגיוני בסך הכל, אם אתה משתמש רשום פשוט תיכנס עם שם המשתמש והסיסמא שלך לשירות.
אם אתה משתמש שעדיין אינו רשום, תפנה למערכת חיצונית שמנהלת אך ורק את רישום המשתמשים, תבצע שם את התהליך ולאחר מכן תפנה לשירות האמיתי ותיכנס עם שם המשתמש והסיסמא שהמערכת החיצונית סיפקה לך.

המערכת מבקשת מהמשתמש שם משתמש וסיסמא, מוודאת ששם המשתמש לא קיים ורושמת אותך למערכת.
רגע לפני שהמערכת שומרת את המשתמש לבסיס הנתונים, היא מבצעת hashing על הסיסמא.
המערכת תומכת בשלושה סוגי hash:

- SHA512
- Adler32
- Crc32

ניתן לבחור בעזרת קובץ קונפיגורציה את שיטת ה-hashing הרצויה.

נתחיל במעבר על הבינים בקבצי הקונפיגורציה.
קיים רק קובץ קונפיגורציה אחד ללא package שנקרא MainConfiguration.
נשים לב תחילה לאנוטציה המעניינת –

@PropertySource

זוהי בעצם הפניה של המפתח שאומר לספרינג – יש לי קובץ properties אני מבקש ממך להתייחס אליו ולקחת אותו בחשבון בפרויקט, כמובן שנותנים גם את ה-path המתאים.
בהמשך נראה איך זה משרת אותנו.

ExecutorCreatorFactory
תחילה, נשים לב שבפרויקט הלא ספרינגי יצירת האובייקט מתחילה במשיכת שדה מקובץ קונפיגורציה, לאחר מכן מייצרים את המפעל ומבצעים השמה לשיטת ההאש, מפה של כל ה-executers.
דבר דומה מתבצע ביצירת הבין בקובץ הקונפיגורציה.
אנסה להסביר על ההבדל בין המקרים ועל היתרונות של ספרינג במקרה הזה.
נסתכל על המחלקה RegisterManager בשני הפרויקטים.
האובייקט הנ״ל הוא אובייקט שתלוי באובייקט המפעל.
בשביל ללכת ולבצע האשינג לסיסמאות הוא חייב להחזיק את המפעל שיעניק לו את המבצע הנכון.
בסופו של דבר מה מעניין מבחינת לוגית את האובייקט RegisterManager מעניין אותו אך ורק להחזיק מפעל כזה, כל הלוגיקה של היצירה שלו, של האתחול שלו ממש לא מעניין את המנג׳ר.
אז נכון שיצירת האובייקט של המפעל כתובה באותה צורה במחלקה registerManager בפרויקט הלא ספרינגי ובקובץ הקונפיגורציה בפרויקט הספרינגי.
אך המשמעות היא החשובה, עכשיו המחקה RegisterManager נקייה מכל הלוגיקה של איך יוצרים את המפעל, איך מאתחלים אותו, איך המפה שלו נראית וכו׳.
ריכזנו במקום אחד את כל הלוגיקה של איך יוצרים את האובייקטים, הוצאנו את הלוגיקה הזאת אל מחוץ למחלקות כי זה מאוד לא מעניין וקשור למחלקות עצמן.

RegisterManager
כעת ניתן להבחין שאכן שכשאנחנו מייצרים את המנג׳ר, שתלוי במפעל וברפסיטורי של המשתמשים אנחנו מזריקים לו דרך הקונסטרקטור את כל אחד מהבינים שהוא צריך.
ועכשיו המחלקה Main בפרויקט הספרינגי תהיה נקייה הרבה יותר, אין צורך לייצר את היוזר רפוסיטורי, יש לנו צורך אך ורק ב-RegisterManger במידה והוא אובייקט מורכב והוא תלוי באובייקטים אחרים, הם יוזרקו בעת יצירתו.
כשאני אדרוש אותו בזמן ריצה הוא כבר יהיה מוכן, טעון, בנוי, יחד עם כל האובייקטים בהם הוא תלוי.

UserRepository
זוהי יצירה פשוטה של בין, מוזרק אל תוך RegisterManager בעת יצירתו.

Sha512Hash, adler32Hash, crc32Hash
אלו הם בעצם האובייקטים שבמצעים את ההאשינג, כל אחד מהם הוא סוג של HashExecutor ויודע לבצע האשינג כל אחד בשיטתו.
וגם כאן נשים לב שהוצאנו את הקוד והלוגיקה של יצירת האובייקטים האלה, שהמפעל תלוי בהם מחוץ למחלקה של המנג׳ר.
כל התלויות בין האובייקטים, כל הלוגיקה של איך יוצרים כל אחד ואחד מהאובייקטים כעת מיושם בקובץ הקונפיגורציה שמגדיר את הבינים.

נתבונן במחלקה RegisterManager
אנחנו זוכרים שבקובץ הקונפיגורציה הוספנו אנוטציה מעניינת PropertySource
נסביר איך זה בא לידי ביטוי.

בפרויקט הלא ספרינגי, החזקנו קובץ קונפיגורציה וייבאנו את השדות שם בעזרת מתודה מסורבלת וארוכה (שורה 82 בפרויקט הלא ספרינגי).
כעת ספרינג מאפשר לנו להצהיר על קובץ הוא PropertySource ואז במחלקות השונות לבצע Autowired לערך שמעניין אותנו

```
@Autowired
public void initProperty(@Value("${hash.method}") String property) {
    this.hashMethod = property;
}
```

בעזרת אנוטציה נוספת של @Value עם שם השדה המתאים מקובץ properties
נוכל לקבל בזמן ריצה ללא כל מאמץ את הערך הרלוונטי מהקובץ.
וככה אנחנו מאפשרים למשתמש לבחור בדיוק באיזו שיטת האשינג הוא רוצה שהמערכת תשתמש בצורה מאוד פשוטה לעומת הפרויקט הלא ספרינגי.