УДК 004

Написание собственного стартера загрузки Spring

Еровлева Регина Викторовна
Приамурский государственный университет имени Шолом-Алейхема
Студент

Еровлев Павел Андреевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данной статье будут рассмотрены возможности создания собственного стартера загрузки. Использоваться будет технология Spring Boot.
Ключевые слова: Starter, Spring, Spring boot

Writing your own Spring boot starter

Erovleva Regina Viktorovna
Sholom-Aleichem Priamursky State University
Student

Erovlev Pavel Andreevich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article will explore the possibilities of creating your own boot starter. Spring Boot technology will be used.
Keywords: Starter, Spring, Spring boot

Стартеры автоматически настраивают все и запускают приложение. Это позволяет быстрее приступить к работе и не беспокоиться о конфигурации, которая обычно является копипастом из чего-то еще. Приятно то, что нет ограничений на уже существующий стартер загрузки Spring, так как также можно написать собственные стартеры.

Цель статьи – написать собственный стартер для загрузки приложений Spring.

С.В. Мельников провел обзор на работу с отладочным интерфейсом Java и методом модификации функциональности приложения, не изменяющий его бинарные файлы [1]. А.А.Шейн, Д.Г.Залевский, С.В. Автайкин, С.В.Карташев, С.А.Скороход разработали и описали действия программы предназначенной для автоматического создания набора классов для представления объектов модели Decode в виде нативных объектов языка

Java [2]. Н.Н. Глибовец проанализировала в своей работе особенности агентных технологий и перспективы их использования для разработки сложных многопользовательских программных систем [3]. В своей работе М.К.Ермаков, С.П.Вартанов рассмотрели вопросы проведения анализа программ интерпретируемых языков программирования [4]. Так же А.А. Птицын, Н.Л. Подколодный, Д.А.Григорович, С.В. Лаврюшев разработали молекулярно-биологический сервер и на его базе создали ряд информационно-вычислительных систем, для изучения регуляции экспрессии генов с использованием новейших технологий Java [5].

Первое, добавляем зависимости «spring-boot-dependencies» в управление зависимостями. Это позволяет использовать любую зависимость Spring, которую хотели бы, без необходимости импортировать всю стартовую спецификацию и без необходимости самостоятельно указывать номера версий.

Следующим шагом будет добавление следующих двух зависимостей (рис.1).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

Рисунок 1 – Добавление зависимостей

Первая зависимость позволяет использовать правильные аннотации для создания собственных классов конфигурации, в то время как вторая зависимость позволяет генерировать правильные метаданные для свойств конфигурации, чтобы IDE могла подбирать их в «intellisense».

Для создания настраиваемого стартера необходимо создать собственный класс и аннотировать его с помощью «@ConfigurationProperties» (рис.2).

```yaml
monitoring:
  path: /prometheus
  interval: 10000
  matchers:
    - name: api.user
      method: GET
      matcher: /api/user
    - name: api.token
      method: GET
      matcher: /api/token
    - name: prometheus
      method: GET
      matcher: /prometheus
    - name: metrics
      method: GET
      matcher: /metrics
```

Рисунок 2 – Создание настроек

Далее создаем классы (рис.3-4).

```java
@ConfigurationProperties("monitoring")
public class MonitoringProperties {
    private String path = "/prometheus";
    private int interval = 10;
    private List<MonitoringPathMatcher> matchers;


}
```

Рисунок 3 – Класс мониторинга

```
public class MonitoringPathMatcher {
    private static final AntPathMatcher MATCHER = new AntPathMatcher();
    private String name;
    private HttpMethod method;
    private String matcher;



    public boolean isMatching(HttpServletRequest request) {
        return getMethod().matches(request.getMethod()) && MATCHER.match(getMatcher(), request.getServletPath());
    }
}
```

Рисунок 4 – Класс мониторинга

«@ConfigurationProperties» указывает на то, что этот POJO будет использоваться для всех свойств конфигурации с префиксом мониторинга. С другой стороны, имена свойств должны иметь то же имя, что и в POJO. Таким образом, в этом случае «monitoring.path» будет соответствовать «path» свойство в «MonitoringProperties» классе.

Если использовать «camelcase» в именах свойств, то можно использовать «monitoring.camelCase=» или «monitoring.camel-case=» для сопоставления.

Чтобы повысить удобство использования библиотеки, можно добавить комментарии к свойствам конфигурации.

Следующим шагом будет создание собственного класса авто-конфигурации. Для этого начнем с создания простого класса и аннотирования его «@Configuration» аннотацией (рис.5).

```
@Configuration
public class MonitorAutoConfiguration {
    // ...
}
```

Рисунок 5 -Класс конфигурации

После этого необходимо включить свойства конфигурации, которые создали ранее, добавив «@EnableConfigurationProperties» аннотацию.

Теперь можно настроить стартер, чтобы Spring вызывал авто-конфигурацию. В данном случае, необходимо вызвать ее после загрузки репозитория метрик загрузочного привода Spring. Для этого используем «@AutoConfigureAfter» аннотацию.

Теперь некоторые стартеры загрузки Spring вызываются только при соблюдении определенных зависимостей или других вещей. Например, авто-конфигурация JPA, вероятно, сработает только тогда, когда драйвер находится в пути к классам, в то время как некоторые авто-конфигурации будут срабатывать только при установке определенного свойства.

В данном случае необходимо быть уверенными, что классы «GaugeService» и «CounterService» классы находятся в пути к классам. Можно сделать это, добавив «@ConditionalOnClass» аннотацию. Кроме того,

нужно вызывать авто-конфигурацию только в том случае, если проект является веб-приложением. Если проект не является веб-приложением, то ничего не можем сделать с конечной точкой «Prometheus». Для этого можно использовать «@ConditionalOnWebApplication» аннотацию (рис.6).

```
@Configuration
@AutoConfigureAfter(MetricRepositoryAutoConfiguration.class)
@ConditionalOnWebApplication
@ConditionalOnClass(name = {"org.springframework.boot.actuate.metrics.CounterService", "org.springframework.boot.actuate.metrics.GaugeService"})
@EnableConfigurationProperties(MonitoringProperties.class)
public class MonitorAutoConfiguration {
    // ...
}
```

Рисунок 6 – Класс стартера

Приятная вещь в «@ConditionalOnClass» аннотации заключается в том, что можно предоставлять классы в виде строк, поэтому не нужно каким-то образом добавлять эти классы в путь к классам.

Теперь, когда правильно аннотировали класс конфигурации, можно начать добавлять к нему bean-компоненты, необходимые для работы авто-конфигурации (рис.7).

```
@Bean
public SpringBootMetricsCollector springBootMetricsCollector(Collection<PublicMetrics> publicMetrics) {
    SpringBootMetricsCollector springBootMetricsCollector = new SpringBootMetricsCollector(publicMetrics);
    springBootMetricsCollector.register();
    return springBootMetricsCollector;
}

@Bean
public ServletRegistrationBean servletRegistrationBean(MonitoringProperties properties) {
    DefaultExports.initialize();
    return new ServletRegistrationBean(new MetricsServlet(), properties.getPath());
}

@Bean
public MonitoringFilter filter(CounterService counterService, GaugeService gaugeService, MonitoringProperties monitoringProperties) {
    return new MonitoringFilter(counterService, gaugeService, monitoringProperties);
}
```

Рисунок 7 – Добавление Bean-компонентов

Прежде чем включать загрузчик Spring в другие проекты, нужно сделать еще одну вещь. Чтобы приложение подхватило стартер загрузки Spring, нужно создать специальный файл свойств с именем «spring.factories». Этот файл должен быть помещен в папку «src/main/resources/META-INF» и должен содержать «org.springframework.boot.autoconfigure.EnableAutoConfiguration» свойство (рис.8).

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=be.g00glen00b.monitor.MonitorAutoConfiguration
```

Рисунок 8 – Добавление свойств

Это свойство должно содержать полное имя класса авто-конфигурации.

Теперь можно включить библиотеку в другие проекты, определив ее как зависимость. При этом авто-конфигурация будет автоматически вызвана, и будут созданы все bean-компоненты.

В данной статье была рассмотрена возможность создания собственного стартера для загрузки приложения в Spring Boot.

Библиографический список

1. Шейн А.А., Залевский Д.Г., Автайкин С.В., Карташев С.В., Скороход С.А. Генератор исходного кода на языке java по описанию бортовых компонентов decode (decode java generator 0.2) // Вестник волжского университета им. в.н. татищева. 2019. №3. С. 26-32.
2. Мельников С.В. Обзор и применение отладочного интерфейса java (jdi) для обратимой модификации программных продуктов // Современные проблемы науки и образования. 2018. №8. С. 8-19.
3. Глибовец Н.Н. Использование jade (java agent development environment) для разработки компьютерных систем поддержки дистанционного обучения агентного типа // Заметки по информатике и математике. 2019. №10. С. 15-20.
4. Ермаков М.К., Вартанов С.П. Подход к проведению динамического анализа java-программ методом модификации виртуальной машины java // Научные труды Винницкого национального технологического университета. 2018. №6. С. 10-17.
5. Птицын А.А., Подколодный Н.Л., Григорович Д.А., Лаврюшев С.В. Создание молекулярно-биологического сервера www с использованием новейших технологий java // Заметки по информатике и математике. 2020. №1. С. 11-20.