Events, Scheduling Tasks and Caching

```
@Cacheable(value="your cache name",
key="name of key")
public void Method_name()
{
// logic goes here //
}
```

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#java-web

Table of Contents

Spring Events

What Are Events in Spring?

- **Events** are a mechanism used for communication between different parts of an **application**
- Spring provides an **event handling** mechanism that allows components within the application to interact without directly **coupling** them together
- This promotes **loose coupling** and improves the modularity and scalability of the application

Basic Spring Boot Events

- In **Spring**, events are published using the **ApplicationEventPublisher** interface
- Spring's **ApplicationContext** interface extends this, meaning any **Spring-managed** bean can publish events

```
public class UserRegistrationEvent extends ApplicationEvent {
    private String username;
    public UserRegistrationEvent(Object source, String username) {
        super(source);
        this.username = username;
    }
    public String getUsername() {
        return username;
    }
}
```

Basic Spring Boot Events

- **UserRegistrationEvent** is fired whenever a new user registers
- It carries the **username** as its **payload**
- To publish this event, we'll use the **ApplicationEventPublisher**

```java
@Service
public class UserRegistrationService {
    @Autowired
    private ApplicationEventPublisher eventPublisher;

    public void registerUser(String username) {
        // ... registration logic ...
        eventPublisher.publishEvent(new UserRegistrationEvent(this, username));
    }
}
```

Basic Spring Boot Events

- Create a listener by implementing **ApplicationListener** to listen for this event
- Whenever a **UserRegistrationEvent** is published, the onApplicationEvent method of our listener is called

```java
@Component
public class UserRegistrationListener implements
ApplicationListener<UserRegistrationEvent> {
    @Override
    public void onApplicationEvent(UserRegistrationEvent event) {
        // ... handle event ...
    }
}
```

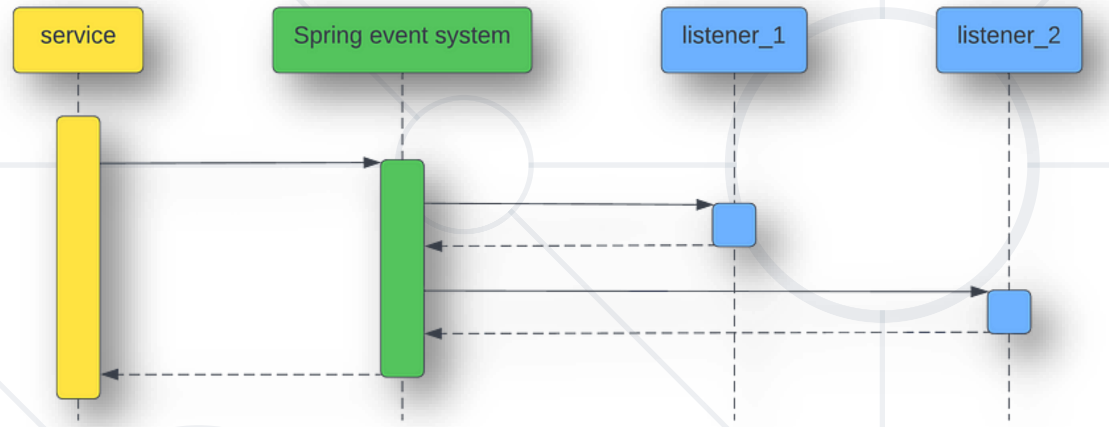
Using Annotations

- Spring also provides a more concise way of handling events using **annotations**
- With the **@EventListener** annotation, any method in a managed **bean** can act as an **event** listener

```
@Component
public class UserRegistrationListener {
    @EventListener
    public void handleUserRegistrationEvent(UserRegistrationEvent
event) {
        // ... handle event ...
    }
}
```

Synchronous Events

- By default, Spring events are **synchronous**
- When the event is published, the main **thread blocks** until all listeners have finished **processing** the event and then **continues**



```
@Component
public class UserRegistrationListener {
    @EventListener
    public void handleUserRegistrationEvent(UserRegistrationEvent event) {
        // ... handle event ...
    }
}
```

Asynchronous Events

- **Asynchronous events** in Spring provide a mechanism for **decoupling** the handling of events from the source of those events, allowing for more efficient handling of tasks that are **not time-sensitive** or **CPU-intensive**
- This approach improves the **responsiveness** and **scalability** of the application
- Way to achieve this is by using **ApplicationEventMulticaster** and **TaskExecutor**

Asynchronous Events

- **ApplicationEventMulticaster**
  - **ApplicationEventMulticaster** interface is responsible for multicasting events to the appropriate listeners
  - It defines methods like **addApplicationListener**, **removeApplicationListener** and **multicastEvent** to manage listeners and propagate events
- **TaskExecutor**
  - Spring's **TaskExecutor** abstraction provides a way to execute tasks asynchronously
  - It defines a single method, **execute**, which takes a **Runnable** task and executes it **asynchronously**

Asynchronous Events

- Combining these two components, you can achieve **asynchronous** event handling in Spring by configuring an **ApplicationEventMulticaster** to use a **TaskExecutor** for event propagation

```
@Configuration
public class AppConfig {
class MyCustomEvent {
    private final String message;

    public MyCustomEvent(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

Asynchronous Events

- **MyEventListener** is an event listener that will handle **MyCustomEvent** asynchronously

```java
public class MyEventListener implements
ApplicationListener<MyCustomEvent> {

    @Override
    public void onApplicationEvent(MyCustomEvent event) {
        // Handle the event asynchronously
        System.out.println("Handling event asynchronously: "
        + event.getMessage());
    }
}
```

Asynchronous Events

- **SimpleApplicationEventMulticaster** is configured as a bean using the **@Bean** annotation, and a **SimpleAsyncTaskExecutor** is set as the task executor

```
// Define the ApplicationEventMulticaster bean
      @Bean
    public SimpleApplicationEventMulticaster applicationEventMulticaster() {
        SimpleApplicationEventMulticaster eventMulticaster = new
      SimpleApplicationEventMulticaster();
        eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
        return eventMulticaster;
    }
// Register the event listener as a bean
      @Bean
    public MyEventListener myEventListener() {
        return new MyEventListener();
    }
```

Asynchronous Events

- The **publishCustomEvent** method can be invoked **anywhere** in the application to **publish** an event
- This setup allows you to use **asynchronous** event handling in your Spring application with the **ApplicationEventMulticaster** configured as a bean

```
public void publishCustomEvent(String message) {
        MyCustomEvent event = new MyCustomEvent(message);
        applicationEventMulticaster().multicastEvent(event);
   }
```

Transactional Events

- **Spring** allows for events to be tied into the application's transaction **management** system, leading to **"transactional events"**
- **Transactional** events are only published after the **successful** completion of a transaction
- To publish a transactional event, you need to use the **TransactionalApplicationEventPublisher**

Transactional Events

```
@Service
public class UserRegistrationService {
    @Autowired
    private TransactionalApplicationEventPublisher eventPublisher;

    @Transactional
    public void registerUser(String username) {
        // ... registration logic ...
        eventPublisher.publishEvent(new
UserRegistrationEvent(this, username));
    }
}
```

Transactional Events

- In Spring, the **@TransactionalEventListener** annotation is used to **listen** for transactional events and perform actions based on the outcome of the transaction
- Binding is **possible** to the following transaction phases:
  - **AFTER_COMMIT** (default) is used to fire the event if the transaction has **completed successfully**
  - **AFTER_ROLLBACK** – if the transaction has **rolled back**
  - **AFTER_COMPLETION** – if the transaction has **completed** (an alias for AFTER_COMMIT and AFTER_ROLLBACK)
  - **BEFORE_COMMIT** is used to fire the event **right before** transaction commit

Transactional Events

- **After Commit Event Handling:**
  - This event is triggered after a transaction has been **successfully** committed to the database
  - Methods annotated with **AFTER_COMMIT** are executed within the same transactional scope as the operation that triggered the event

```
@Component
public class MyTransactionalEventListener {

    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    public void handleAfterCommit(MyTransactionEvent event) {
        // Perform actions after the transaction has been successfully committed
        System.out.println("Transaction committed successfully: " +
event.getTransactionId());
    }
}
```

Transactional Events

- **After Rollback Event Handling:**
  - After a transaction is rolled back, you might want to perform certain **cleanup** or **logging** operations to handle the failure gracefully
  - This is useful for scenarios where you need to **undo** changes made during the transaction or notify administrators about the failure

```java
@Component
public class TransactionEventListener {

    @TransactionalEventListener(phase = TransactionPhase.AFTER_ROLLBACK)
    public void handleFailedTransaction(FailedTransactionEvent event) {
        // Log the details of the failed transaction
        System.out.println("Transaction failed: " + event.getTransactionId());
    }
}
```

Transactional Events

- **After Completion Event Handling:**
  - The **AFTER_COMPLETION** event is triggered after the transaction has been completed, **regardless** of whether it was committed or rolled back
  - This method will be invoked after **each transaction** completes

```
@Component
public class MyTransactionListener {

    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMPLETION)
    public void handleAfterCompletion(MyCustomEvent event) {
        // Logic to execute after the transaction completes
    }
}
```

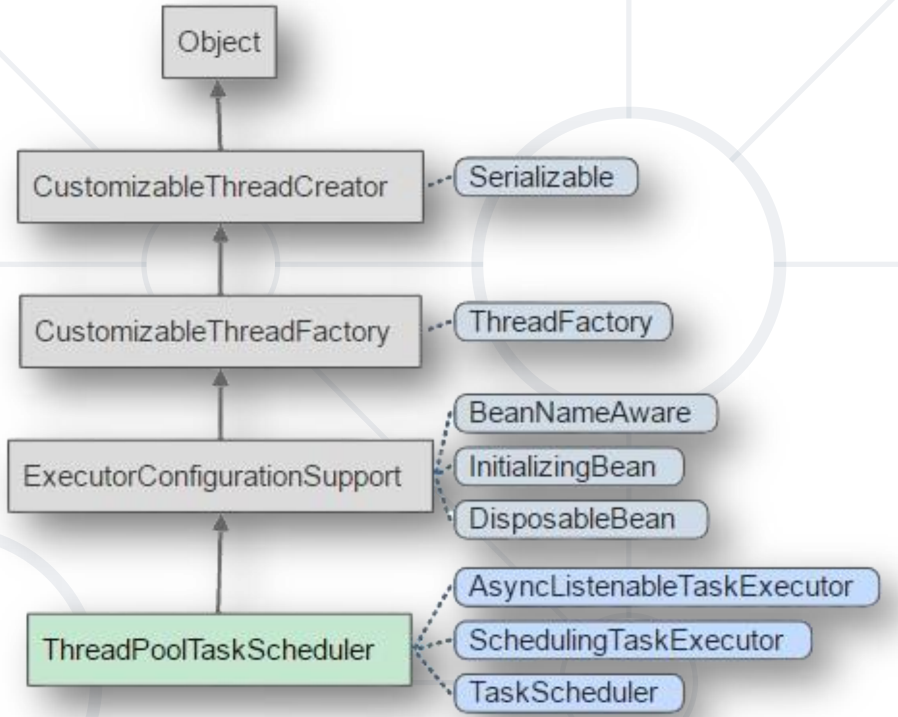
Transactional Events

```
@Component
public class MyService {

    @Autowired
    private TransactionalApplicationEventPublisher eventPublisher;

    @Transactional
    public void doSomething() {
        // Business logic
        eventPublisher.publishEvent(new MyCustomEvent(this));
    }
}
```

Scheduling Tasks

What Is Scheduling?

- **What is Scheduling in Spring**
  - It is a process of executing tasks or jobs at specified intervals or times
  - Spring Boot provides a good support for **scheduling** tasks using **annotations** or **XML** configuration, making it easy to **schedule** background jobs within your application
- **Task Execution Context**
  - Spring Boot provides a **TaskScheduler** interface and various implementations to manage scheduled tasks
  - By default, Spring Boot uses a **ThreadPoolTaskScheduler** to execute **scheduled** tasks in a **multi-threaded** environment



Enabling Scheduling

- **Enabling Scheduling**
  - in your Spring Boot **application**, you need to annotate your **main** application class with **@EnableScheduling**
  - This annotation enables Spring's **scheduling** capabilities in the **application context**

```
@SpringBootApplication
@EnableScheduling
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

@Scheduled Annotation

- **@Scheduled Annotation**
  - Spring Boot allows you to **schedule** tasks using the **@Scheduled** annotation
  - This annotation specifies when the annotated methods should be **executed**
  - It supports various **attributes** to define the scheduling **behavior**, such as fixed **delay**, fixed **rate**, or **cron** expressions

```java
@Component
public class MyScheduledTasks {

    @Scheduled(fixedRate = 5000) // Execute every 5 seconds
    public void doTask() {
        // Task logic
    }
}
```

Fixed Delay and Fixed Rate

- **Fixed Delay**
  - The **fixedDelay** attribute of the **@Scheduled** annotation specifies the **time** (in milliseconds) to wait after the completion of the **previous** execution before starting the **next** execution
- **Fixed Rate**
  - The **fixedRate** attribute of the @Scheduled annotation specifies the time (in milliseconds) between the **start** times of **each** execution
  - It **ensures** that the task **runs** at a fixed interval, **regardless** of the execution time of the previous task

Cron Expressions

- **What is Cron**
  - **Cron** is a basic utility available on Unix-based systems and enables users to **schedule** tasks to run periodically at a **specified** date/time
- **Cron Expressions**
  - For more complex scheduling requirements, you can use **cron** expressions with the **@Scheduled** annotation
  - Cron **expressions** allow you to define **flexible** schedules based on specific dates, times, and **recurring** patterns

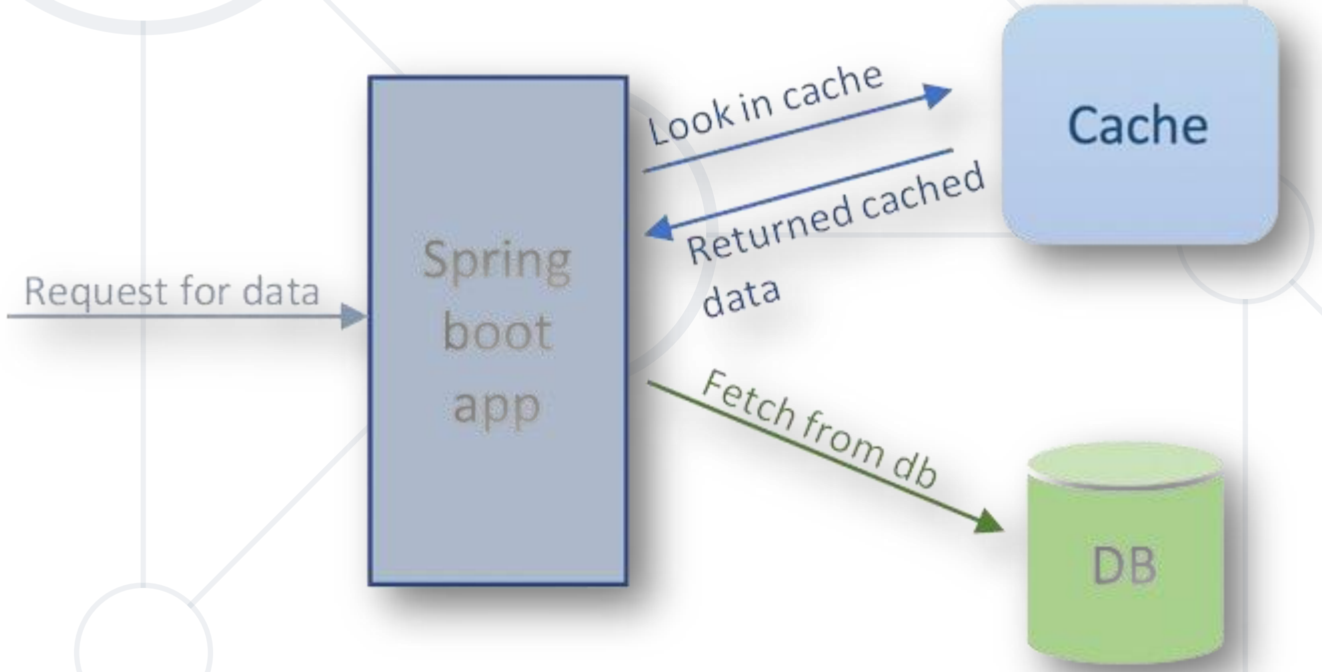
```java
@Scheduled(cron = "0 * * * * ?")
    // Execute every minute
public void doTask() {
    // Task logic
}
```

for the curious: https://crontab.guru

Caching

What is Caching?

- **Caching** is a mechanism aimed at enhancing the **performance** of any kind of application
- It relies on a **cache**, which can be seen as a temporary **fast access** software or hardware component that stores data to **reduce** the time required to **serve future requests** related to the same data



Spring Boot Cache Abstraction

- The **Spring Boot Cache Abstraction** does not come with the framework natively but requires a few **dependencies**
- You can easily install all of them by adding the **spring-boot-starter-cache** to your dependencies

Caching Annotations

- **@EnableCaching**
  - To enable the Spring Boot caching feature, you need to add the **@EnableCaching** annotation to any of your classes annotated with **@Configuration** or to the boot application class annotated with **@SpringBootApplication**

```
@SpringBootApplication
@EnableCaching
public class SpringBootCachingApplication {
  public static void main(String[] args) {
    SpringApplication.run(SpringBootCachingApplication.class, args);
  }
}
```

Caching Annotations

- **@Cacheable**
  - This **method-level** annotation lets Spring Boot know that the return value of the annotated method can be **cached**
  - Each time a method marked with this **@Cacheable** is called, the caching behavior will be applied

```
@Cacheable("authors")
public List<Author> getAuthors(List<Int> ids) { ... }
```

Caching Annotations

- **@Cacheable**
  - You can also specify how the key that uniquely identifies each entry in the cache should be generated by harnessing the **key** attribute

```
@Cacheable(value="book", key="#isbn")
public Book findBookByISBN(String isbn) { ... }

@Cacheable(value="books", key="#author.id")
public Books findBooksByAuthor(Author author) { ... }
```

Caching Annotations

- **@CachePut**
  - This annotation is used to **update** the cache with the result of the annotated method, regardless of whether the method result is already cached or **not**
  - It **forces** the method execution and then **updates** the cache with the new result
  - The main difference between **@Cacheable** and **@CachePut** is that the first might avoid **executing** the method, while the second will **run** the method and put its results in the cache, **even** if there is already an existing key associated with the given parameters

Caching Annotations

- **@CacheEvict**
  - This **method-level** annotation allows you to **remove** (evict) data previously stored in the cache
  - By annotating a method with **@CacheEvict** you can specify the removal of **one** or **all** values so that fresh values can be loaded into the cache again
  - If you want to remove a **specific** value, you should pass the cache key as an **argument** to the annotation

```
@CacheEvict(value="authors", key="#authorId")
public void evictSingleAuthor(Int authorId) { ... }
```

Caching Annotations

- **@CacheEvict**
  - If you want to clear an entire cache you must use the parameter **allEntries** in conjunction with the name of cache to be cleared

```
@CacheEvict(value="authors", allEntries=true)
public String evictAllAuthorsCached() { ... }
```

Caching Annotations

- **@Caching**
  - @Caching allows multiple nested **@Cacheable**, **@CachePut** and **@CacheEvict** to be used on the same method

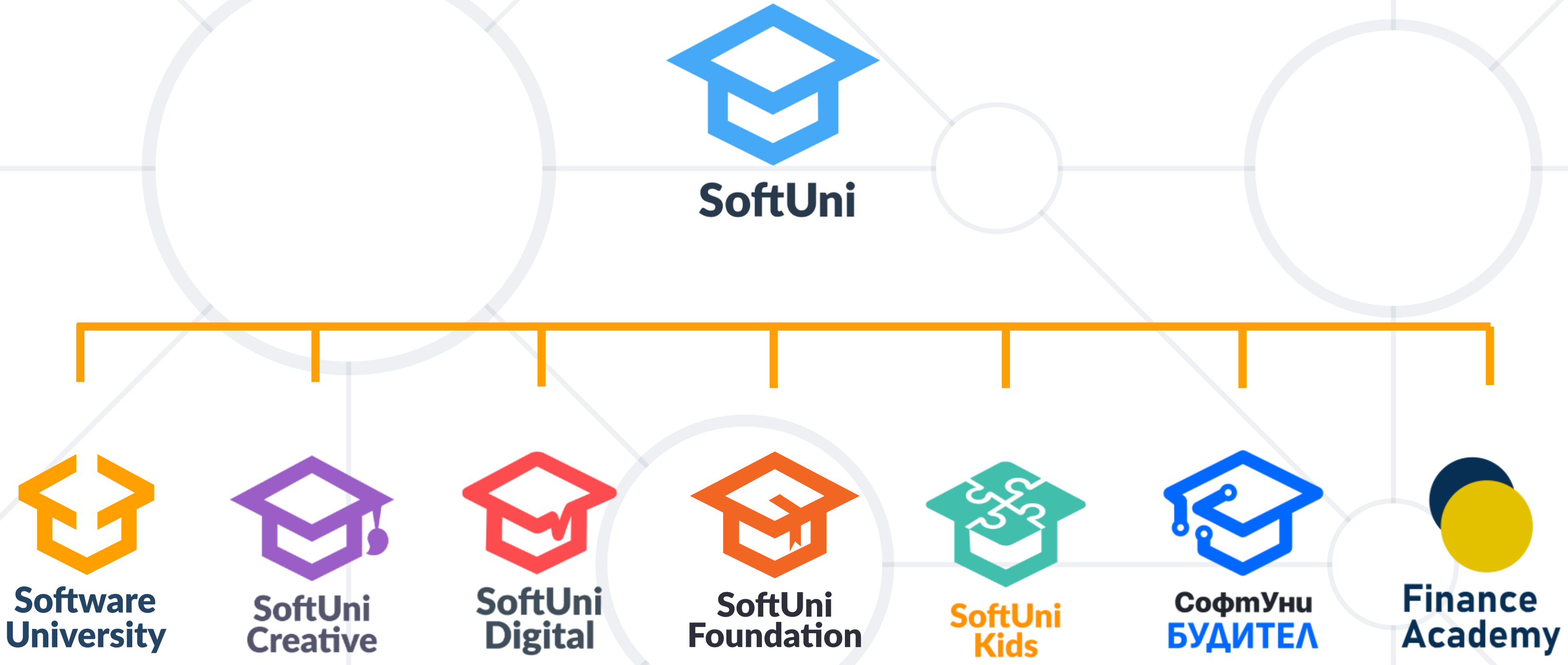
```
@Caching(evict = { @CacheEvict("primary"),
@CacheEvict(value = "secondary",
key = "#p0") })

public Book importBooks(String deposit, Date date)
```

Summary

- Spring Events
- Scheduling Tasks
- Caching

Software University

Questions?



SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License