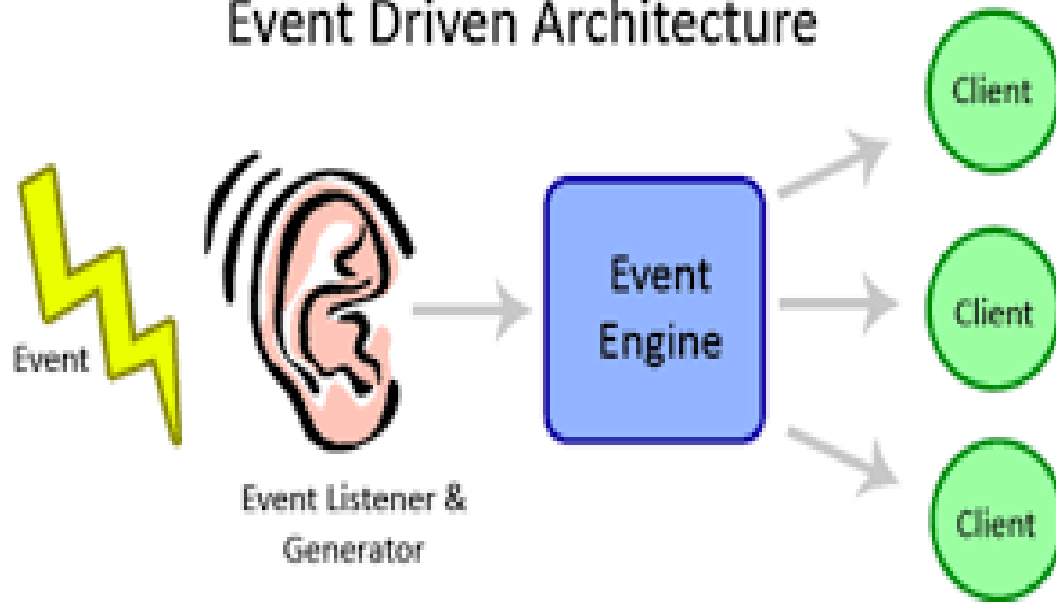


# Events in Spring

## Event Driven Architecture



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

# Table of Contents

1. What are Events
2. Build-in Events
3. Custom Events
4. Scheduling Tasks
5. Caching



sli.do

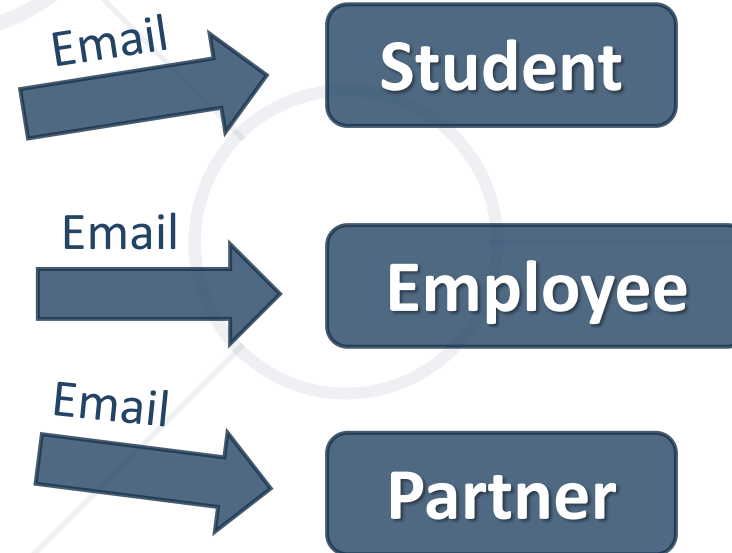
**#java-web**



**What Are the Events**

# Observer Pattern in JAVA

- Observer pattern is a **behavioral pattern**
- Provides **one object** with a loosely coupled method of **informing multiple objects** of property changes



# Events in Spring

- The core of Spring is the **ApplicationContext**, which manages the complete **life cycle** of the beans
- The ApplicationContext **publishes** certain types of **events** when **loading** the beans
- Spring's event handling is **single-threaded** so if an event is published, until all the receivers get the message, the **processes** are **blocked** and the flow will not continue



- **ContextRefreshedEvent**
  - published when the ApplicationContext is either initialized/refreshed
- **ContextStartedEvent**
  - published when the ApplicationContext is started using the **start()**
- **ContextStoppedEvent**
  - published when the ApplicationContext is stopped using the **stop()**

- **ContextClosedEvent**
  - published when the `ApplicationContext` is closed using the **`close()`**
- **RequestHandledEvent**
  - Web-specific event telling all beans that an HTTP request has been serviced





# Listening for Events

- There are ways to listen for events in Spring:
  - **Implement** the **ApplicationListener** interface
    - Which has just one method **onApplicationEvent()**
  - Use **@EventListener()**
    - Annotate on a method
- Some of the events are **published too early** for a listener to be found via annotations and the application context. Then you must **register them** in the Spring Application instance

- Implementing **ApplicationListener** interface

```
@Component
public class EventsListener implements
ApplicationListener<SpringApplicationEvent> {
    @Override
    public void onApplicationEvent(SpringApplicationEvent e) {
        System.out.printf("Event-%s!\n",
            e.getClass().getSimpleName());
    }
}
```

- Use **@EventListener()** with specific event class

```
@EventListener(ApplicationStartingEvent.class)
public void startEvent(){
    System.out.println("Starting Event!"); }

```

```
@EventListener(RequestHandledEvent.class)
public void requestHandler(){
    System.out.println("Request Handler event!");
}

```

- Use `@EventListener(classes = {EventOne.class, EventTwo.class})` to listen for multiple events

```
@EventListener(classes = {MyEventOne.class,  
MyEventTwo.class})  
public void handleTwoEvents(){  
    System.out.println("Listens for two events!");  
}
```

- Using **lambda expressions** with specific event class

**@SpringBootApplication**

```
public class DemoForCustomEventsApplication {  
    public static void main(String[] args) {  
        SpringApplication springApp = new SpringApplication  
            (DemoForCustomEventsApplication.class);  
        springApp.addListeners((ApplicationContextInitializedEvent e) -> {  
            System.out.println("App context init event");  
        });  
        springApp.run(args);  
    }  
}
```

# Register Events in Spring Application

- **Remember** that for some event is published too early for a listener to be found and needs to **be added**



```
@SpringBootApplication
```

```
...
```

```
    springApp.addListeners(new MyEventsClass());  
    springApp.run(args);
```

```
...
```

- The listener of an event to a **phase** of the **transaction**
- Transaction **phases** :
  - **AFTER\_COMMIT** - The default, used to fire the event if the transaction has **completed successfully**
  - **AFTER\_ROLLBACK** - when transaction has **rolled back**
  - **AFTER\_COMPLETION** - when transaction has **completed**
  - **BEFORE\_COMMIT** - used to fire the event right **before** transaction **commit**



# Transaction Bound Events (2)

- An example of Transaction Bound Event, that will fire before transaction commit

```
@TransactionalEventListener(phase =  
    TransactionPhase.BEFORE_COMMIT)  
public void transactionEventListener  
    (MyCustomEvent event){  
    System.out.println("Hit before transaction commit!");  
}
```



**Creating Custom Event**

- To create and publish our custom event, there is some steps that we need to follow:
  - **Create** our custom **event class** that **extends** **ApplicationEvent** class
  - **Create publisher**, that publish our new event
  - **Add event listener**, that listens for our new event

# Create Our Custom Event Class

- Create our event class, that extends `ApplicationEvent`

```
public class MyCustomEvent extends ApplicationEvent {  
    private String msg;  
    public MyCustomEvent(Object source, String msg) {  
        super(source);  
        this.msg = msg;  
    }  
    ... }  
}
```

- Create a publisher that publish our custom event and inject in him the ApplicationEventPublisher object

```
@Component
public class MyPublisher {
    @Autowired // It is better to inject in constructor
    private ApplicationEventPublisher appEventPublisher;
    public void publishEvent(String message) {
        MyCustomEvent myEvent = new MyCustomEvent(this, message);
        appEventPublisher.publishEvent(myCustomEvent);
    } } ;
```

- Create listeners, already explain the different ways

```
@Component
public class Listeners {
    @EventListener(MyCustomEvent.class)
    public void listener(MyCustomEvent myCustomEvent) {
        System.out.printf("Custom event listeners with message\n%s!\n", myCustomEvent.getMsg());
    }
}
```



# Scheduling Tasks

- **Scheduling** is a process of executing the tasks for the **specific time** period
- Spring Boot provides a good support to write a scheduler on the Spring applications
- We can specify the time period by different ways:
  - Using **Cron**
  - Using **Fixed Rate**
  - Using **Fixed Delay**



# Scheduled Task Using Cron

- Java **Cron expressions** are used to configure the instances of **CronTrigger**
- The *cron* expression consists of **six fields**:

<second><minute><hour><day-of-month><month><day-of-week>

```
@Scheduled(cron = "0 5 * * * ?")  
public void showTimeWithCron(){  
    System.out.println(LocalDateTime.now());  
}
```

# Scheduled Task Using Fixed Rate

- **Fixed Rate** scheduler is used to execute the tasks at the **specific time**
- It **does not wait** for the completion of **previous task**
- The values should be in **milliseconds**

```
@Scheduled(fixedRate = 5000)
    public void showTimeWithFixedRate() {
        System.out.println(LocalDateTime.now());
    }
```

# Scheduled Task Using Fixed Delay

- **FixedDelay** is the time between tasks
- The **initialDelay** is the time after which the task will be executed the first time after the initial delay value
- It **wait** for the completion of **previous task**

```
@Scheduled(fixedDelay = 5000, initialDelay = 10000)
public void showTimeWithFixedDelay() {
    System.out.println(LocalDateTime.now());
}
```

- The **@EnableScheduling** annotation is used to **enable** the **scheduler** for your application.
- This annotation should be added into the main Spring Boot application class file.

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



**Caching Data**

- If you using Spring Boot, then simply use the **spring-boot-starter-cache** dependency
- Under the hood, the starter brings the spring-context-support module

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-cache</artifactId>  
</dependency>
```

- When using Spring Boot, the **@EnableCaching** annotation would register the **ConcurrentMapCacheManager**
- No need for separate Bean declaration
- Simply adding the **@EnableCaching** annotation to any of the configuration classes

```
@Configuration
@EnableCaching
public class MyConfig {
    // Some configurations
}
```

- Use **@Cacheable** to demarcate methods that are cacheable
- Result is **stored** in the **cache** and on subsequent invocations (with the same arguments), the value in the cache is returned **without** having to actually **execute** the method
- the **findAllStudents** method is associated with the cache named **students**

```
@Cacheable("students")  
public List<Student> findAllStudents() { //... }
```



- Custom Cache Resolution

```
@Cacheable("students", cacheManager = "myCacheManager")  
public List<Student> findAllStudents() { //... }
```

- Conditional Caching

```
@Cacheable("student", condition = "#avg > 4")  
public List<Student> findStudentsByAvgScore(Double avg) {  
    //...  
}
```

- When the **cache** needs to be **updated** without interfering with the method execution
- The **method** is **always executed** and its result is placed into the cache
- It supports the same options as **@Cacheable**

```
@CachePut("students")  
public List<Student> findAll() {  
    //...  
}
```

- This process is useful for **removing** stale or unused data from the cache
- Using the **allEntries** attribute to evict all entries from the cache

```
@CacheEvict(cacheNames="books", allEntries=true)
public void loadStudents() {
    //...
}
```

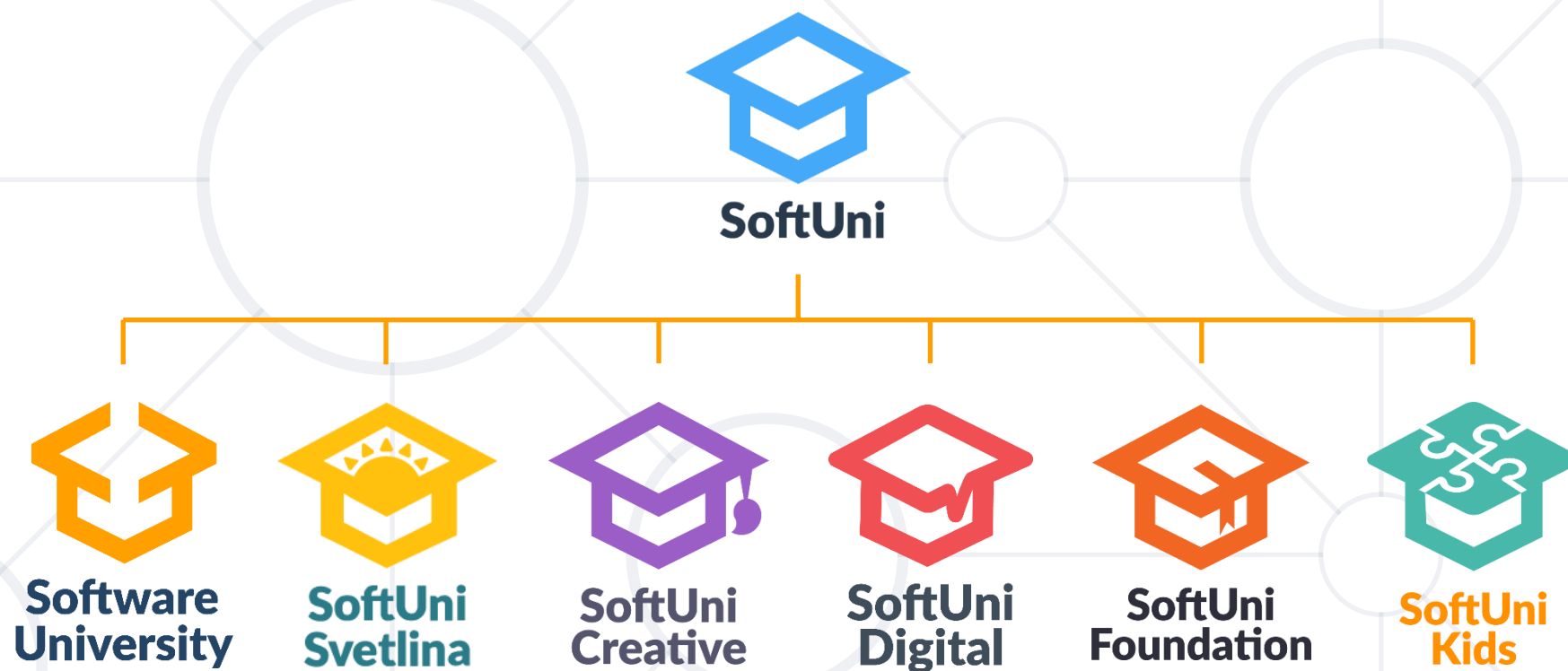
- To customize the CacheManager we must implement **CacheManagerCustomizer<ConcurrentMapCacheManager>**
- Create Bean CacheManager that returns new **ConcurrentMapCacheManager**

```
@Component
public class MyCacheCustomizer implements
    CacheManagerCustomizer<ConcurrentMapCacheManager> {
    @Override
    public void customize(ConcurrentMapCacheManager cacheM){
        cacheM.setCacheNames(asList("students", "courses"));
    }
}
```

- What are the **build-in** Events in Spring
  - How easy to use them
- How to make **listeners** for Events
  - Different ways to implement it
- How to create and use our **custom Events**
- Creating **Scheduled tasks** and **Caching** data



# Questions?



# SoftUni Diamond Partners

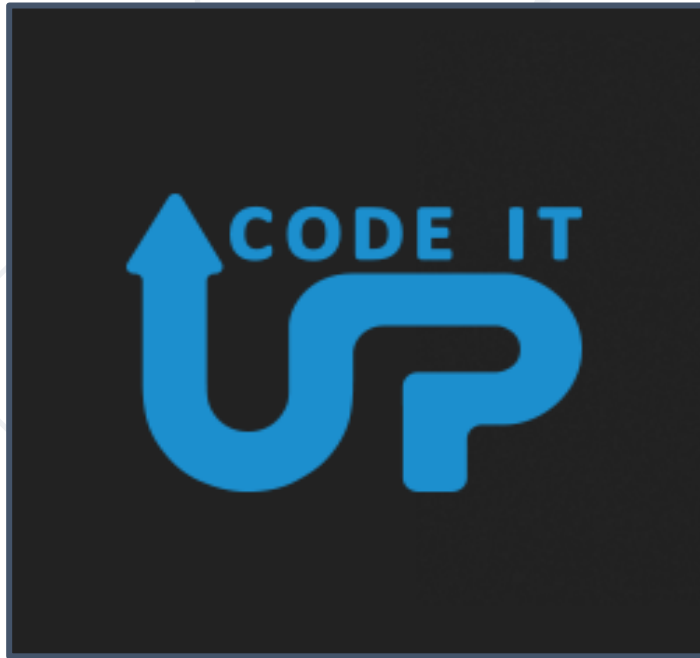


**SCHWARZ**



**SUPER  
HOSTING  
.BG**





**VIRTUAL RACING SCHOOL**





- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

