# WEB API and REST Controllers

### RESTful API

True RESTful API, is a web service must adhere to the following six REST architectural constraints

* + Use of a uniform interface (UI)
  + Client-server based
  + Stateless operations
  + RESTful resource caching
  + Layered system
  + Code on demand

### SOAP and RPC

**Simple Object Access Protocol** (SOAP)

* + Standardized protocol that **sends messages** using other protocols such as **HTTP** and **SMTP**
  + The SOAP specifications are official web standards, maintained and developed by the World Wide Web Consortium (W3C)

**Remote Procedure Call** (RPC)

* + A way to describe a mechanism that lets you **call a procedure in another process** and **exchange data by message passing**

### HTTP GET – retrieve data

* + Used to retrieve single data entities

GET /items/1 -> {id:1, ….}

* + Used to retrieve data arrays

GET /items -> [{id:1, …}, {id:2, …}, …]

### HTTP POST – save data

* + Used to save data

Request:

POST /items

{'name': 'Read Book', 'deadline': 1362268800000, 'categoryName': 'Work', 'enabled': false }

Response:

OK {'id': 1, 'name': 'Read Book', 'deadline': 1362268800000, 'categoryName': 'Work', 'enabled': false }

### HTTP PUT – Update data

PUT /items/1

{'id': 1, 'name': 'Read Book', 'deadline': 1362268800000, 'categoryName': 'Work', 'enabled': false }

### HTTP DELETE – Delete data

DELETE /items/delete/1

## REST with Spring

### Response Body On MVC Controller

@GetMapping('/info/{id}')

@ResponseBody

public Student getInfo(@PathVariable Long id){  
 ...

return new Student().setName(“Joro”);

}

### Response Status

@GetMapping('{id}/info')

@ResponseStatus(HttpStatus.OK)

public GameInfoView getInfo(@PathVariable Long id){  
 GameInfoView gameInfo = this.gameService.getInfoById(id);

return new Gson().toJson(gameInfo);

}

### REST Controllers

**@RestController** is essentially **@Controller** + **@ResponseBody**

@RestController

public class OrderController {

@GetMapping('{id}/info')

public ResponseEntity<Game> getGame(@PathVariable Long id){  
 ...

}

}

### Response Entity

* + Controlling the entire response object

@GetMapping('{id}/title')

public ResponseEntity<Game> getTitle(...){

...

return new ResponseEntity(gameService.getGame(id));

}

* + The **ResponseEntity<>** object allows you **to change the response body**, response headers and response code

### Spring Data REST

* + Maven Dependency

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

* + Spring Data REST **scans your project** and **provides REST API** for your application **using HAL** as media type

### Configuring Repositories

You can configure repository settings using the **@RepositoryRestResource** annotation:

**@RepositoryRestResource(path = 'gameIssues')**

public interface IssueRepository extends   
 JpaRepository<Issue, Long> {

Issue getById(@Param('id') Long id);

List<Issue> getAllByOrderByDateDesc();

}

### Rest Template

* + Accessing **a third-party REST service** inside a Spring application revolves around the use of the Spring **RestTemplate class**
  + Class is **designed to call REST services**
  + Its **main methods** are closely tied to **REST's underpinnings**, which are the **HTTP protocol's methods**: **HEAD**, **GET**, **POST**, **PUT**, **DELETE**
  + **Recommended** to use the non-blocking, **reactive WebClient**.
  + RestTemplate will be **deprecated in a future version**

### HTTP GET Method Example

* + getForObject(url, classType)
    1. Retrieves a representation by doing a GET on the URL.
    2. The response (if any) is unmarshalled to given class type and returned
  + getForEntity(url, responseType)
    1. Retrieve a representation as ResponseEntity by doing a GET on the URL
  + **exchange(requestEntity, responseType)**
    1. **Executes** the specified **request** and **returns** the response as **ResponseEntity**
  + **execute(url, httpMethod, requestCallback, responseExtractor)**
    1. **Executes** **the** **httpMethod** to the given URI template and **preparing the request** with the **RequestCallback**

### HTTP POST Method Example

* + **postForObject(url, request, classType)**
    1. POSTs the given object to the URL and returns the representation found in the response as given class type
  + **postForEntity(url, request, responseType)**
    1. POSTs the given object to the URL and returns the response as ResponseEntity
* **postForLocation(url, request, responseType)**
  + 1. POSTs the given object to the URL and returns the value of the Location header
* **exchange(url, requestEntity, responseType)**
* **execute(url, httpMethod, requestCallback, responseExtractor)**

# Spring Security

## Filters and Interceptors

## Filters

* A filter is an object used to **intercept** the HTTP **requests** and **responses** of your application
* We can perform two operations at two instances:
  + Before sending the **request** to the controller
  + Before sending a **response** to the client

### Filter Example

**@Component**

public class GreetingFilter implements Filter {

**@Override**

public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,

FilterChain filterChain) throws IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) servletRequest;

HttpServletResponse response = (HttpServletResponse) servletResponse;

request.getSession().setAttribute('name', 'Pesho');

filterChain.doFilter(request, response);

}

}

**@Controller**

public class HomeController {

**@GetMapping('/')**

public ModelAndView index(ModelAndView modelAndView, HttpSession session) {

modelAndView.setViewName('index');

modelAndView.addObject('name', session.getAttribute('name'));

return modelAndView;

}

}

## Interceptor

* A **Filter** is used in the **web layer only** as it is defined in web.xml. We can not use it out of web context
* While Spring **Interceptors** are defined in the Spring context
* The **interceptor** include **three** main **methods**:
  + **preHandle**: executed before the execution of the target resource
  + **afterCompletion:** executed after the execution of the target resource (after rendering the view)
  + **postHandle:** Intercept the execution of a handler

### Interceptor Example

public class LogingInteceptor implements HandlerInterceptor {

**@Override**

public boolean preHandle(HttpServletRequest request, HttpServletResponse response,

FilterChain filterChain, Object handler) throws IOException, ServletException {

//Log some information ...

retrun true;

}

}

### Register Interceptor in Configuration

To use interceptors we need to register them

**@Configuration**

public class WebConfiguration implements WebMvcConfigurer {

private final MyInterceptor myInterceptor;

public WebConfiguration(MyInterceptor myInterceptor) {

this.myInterceptor = myInterceptor;

}

**@Override**

public void addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(myInterceptor);

}

}

## What is Spring Security?

* A **powerful** and **highly customizable** authentication and access-control framework
* It is the de-facto **standard** for securing **Spring-based** applications
* Focuses on providing both **authentication** and **authorization** to Java applications
  1. **Authentication**

Who is logged in ?

* 1. **Authorization**

What you are allowed to do ?

### Security Context and Authentication

****

* At the heart of Spring Security's authentication model is the **SecurityContextHolder**
* It contains the **SecurityContext**

### Spring Security Maven/Gradle

* Adding **Spring Security**.

**Maven**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

**Gradle**

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-security'

}

### Spring Security Configuration

* We need to implement **UserDetails** interface

public interface UserDetails {

Collection<? extends GrantedAuthority> getAuthorities();

String getPassword();

String getUsername();

boolean isAccountNonExpired();

boolean isAccountNonLocked();

boolean isCredentialsNonExpired();

boolean isEnabled();

}

* Implementing the **GrantedAuthority** interface.

public class Role implements GrantedAuthority {

private String authority;

}

**SimpleGrantedAuthority**

* If we want, we can use **SimpleGrantedAuthority** instead of creating Role class
* It is a basic concrete **implementation** of a **GrantedAuthority**
* Stores a **String representation** of an **authority** granted to the Authentication object
* Implementing the **UserDetailsService** interface.

public class UserDetailsServiceImpl implements **UserDetailsService** {

public UserDetailsServiceImpl() {

…

}

**@Override**

public UserDetails loadUserByUserName(String userName) {

// get the user and map to **UserDetails**

}  
}

* Expose as beans

**@Configuration**

public class SecurityConfiguration {

**@Bean**

public UserDetailsService userDetailsService(UserRepository

userRepository) {

return new UserDetailsServiceImpl(userRepository);

}

**@Bean**

public PasswordEncoder passwordEncoder() {

return new Pbkdf2PasswordEncoder();

}

**@Bean**

public SecurityFilterChain filterChain(HttpSecurity httpSecurity){

http

.authorizeRequests()

.antMatchers('/', '/register').permitAll()

.anyRequest().authenticated();

…

return http.build();

}

}

### Principal

* This is the currently logged user

**Pre / Post Authorize**

* Grant Access to specific methods

### Cross-Site Request Forgery (CSRF)

**Spring CSFR Protection**

.csrf()

.csrfTokenRepository(csrfTokenRepository())

private CsrfTokenRepository csrfTokenRepository() {

HttpSessionCsrfTokenRepository repository = new HttpSessionCsrfTokenRepository();

repository.setSessionAttributeName("\_csrf");

return repository;

}

<input type='hidden' th:name='${\_csrf.parameterName}' th:value='${\_csrf.token}' />

### Thymeleaf Security

**Dependency**

<dependency>

<groupId>org.thymeleaf.extras</groupId>

<artifactId>thymeleaf-extras-springsecurity5</artifactId>

</dependency>

**Example**

<!DOCTYPE html>

<html lang='en'

xmlns:th='http://www.thymeleaf.org'

xmlns:sec='http://www.thymeleaf.org/extras/spring-security'>

<body>

<div sec:authentication='name'>

The value of the 'name' property of the authentication object should appear here.

</div>

</body>

</html>

<!DOCTYPE html>

<html lang='en'

xmlns:th='http://www.thymeleaf.org'

xmlns:sec='http://www.thymeleaf.org/extras/spring-security'>

<body>

<div sec:authorize='hasRole('ADMIN')'>

This content is only shown to administrators.

</div>

</body>

</html>

# Hypermedia As the Engine of Application State (HATEOAS)

### What is HATEOAS

* **HATEOAS** is a constraint of the REST application architecture
* Keeps the RESTful style architecture **unique from most other network application** architectures
* Uses **hypermedia** to describe what future actions are available to the client
* Allowable actions are derived in the API based on the current application state and returned to the client as a **collection of links**
* Client uses these **links to drive further** interactions with the API
* Tells the client what **options** are **available** at a given point in time.
* Doesn't tell them how each link should be used or exactly what information should be sent
* It is conceptually the same as a **web user browsing** through web pages by clicking the **relevant hyperlinks**to achieve a final goal

**HATEOAS Example**

* Simple response **without** using **HATEOAS**
* We have a simple REST controller that returns entity in JSON format to the client

{ "id" :2, "name": "Pesho", "age":12 }

**Using** **HATEOAS**

{ "id":2,

"name":"Pesho",

"age":12,  
 "\_links":{

"self":{"href":"http://localhost:8080/students/2"},

"delete":{"href":"http://localhost:8080/students/delete/2"},

"update":{"href":"http://localhost:8080/students/update/2"},

"orders":{"href":"http://localhost:8080/orders/allByStudentId/2"}

}

}

**Rel & Href**

* **rel** - describes the **relationship** between the Student resource and the URL
  + In example above – self, update, delete …
  + **describes** the **action** that's performed with the link
  + It's important that this **value** is intuitive as it **describes** the purpose of the link
* **href** - the **URL** used to perform the action described in rel

### Implement HATEOAS in Spring

* Adding hypermedia links to RESTful responses is something you could implement on your own, but …
* **Spring HATEOAS** makes it **very easy**

<dependency>

<groupId>org.springframework.hateoas</groupId>

<artifactId>spring-hateoas</artifactId>

<version>1.1.0.RELEASE</version>

</dependency>

### Example App DB

* Our example app have small base with some relations between entities
  + We have Students, Orders and Courses

**Prepare Controllers to Work**

* If we implementing **RepresentationModel <T>** we can added links directly to our entity
* We need two methods from **WebMvcLinkBuilder**, that’s why we must import them

import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;

import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;

**Main Work in Controller**

... // Without implementing RepresentationModel<T>

Optional<Student> studentOpt = this.studentRepository.findById(id);

return studentOpt  
 .map(s -> ResponseEntity.ok(  
 EntityModel.of(s, getStudentLinks(s))))  
 .orElse(ResponseEntity.notFound().build());

}

private Link[] getStudentLinks(Student student) {

Link self = linkTo(methodOn(StudentsController.class)

.getStudent(student.getId()))

.withSelfRel();

Link orders = linkTo(methodOn(StudentsController.class)

.getAllOrdersByStudentId(student.getId()))

.withRel("orders");

return List.of(self, orders).toArray(new Link[0]);

}

**... // Implementing RepresentationModel<T>**

Student student = this.studentService.findById(id);

student.add(linkTo(methodOn(StudentsController.class)

.getStudent(student.getId()))  
 .withSelfRel());

student.add(linkTo(methodOn(StudentsController.class)

.deleteStudent(student.getId()))  
 .withRel("delete"));

student.add(linkTo(methodOn(StudentsController.class)

.updateStudent(student.getId(), student))  
 .withRel("update"));

student.add(linkTo(methodOn(OrdersController.class)

.findAllOrdersByUserId(student.getId()))

.withRel("orders"));

return ResponseEntity.ok(student);

### Benefits of Using HATEOAS

* **URL structure** of the API can be **changed without affecting** clients
* If the URL structure is changed in the service, clients will automatically pick up the new URL structure   
  via hypermedia
* Hypermedia APIs are **explorable**
* Guiding clients toward the next step in the workflow by **providing** only the **links** that are **relevant** based on the current application state

### Negatives of Using HATEOAS

* Adds **extra complexity** to the API, which affects to:
  1. **developer** needs to handle the **extra work** of adding links to each response
  2. **more complex** to **build** and **test** than a vanilla CRUD REST API
  3. **clients** also have to deal with the **extra complexity** of **hypermedia**

### HAL Explorer

* + To use **HAL Explorer** we need to add **dependency**

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-rest-hal-explorer</artifactId>

</dependency>

# Error Handling

### What is error handling?

* **Error handling** refers to:
  + The **anticipation**, **detection** and **resolution** of programming errors
  + The response & recovery procedures from error conditions
* Error handling is necessary!
  + Improves **user experience**
  + **Optimizes** debugging
  + Facilitates **code maintenance**
  + Ensures **product quality**

### Error Handling in Spring

* Spring MVC offers **no default** (fall-back) **error page** out of the box, however Spring Boot does
* At start-up, Spring Boot **tries to find** a mapping for **/error**
* Spring MVC provides **several approaches** to error handling
  + Per exception
  + Per controller
  + Globally
* Each option has its own use cases and circumstances
* You can use:
  + **Response-annotated** custom exceptions
  + **Controller-based** handlers on specified actions
  + **@ControllerAdvise** annotated classes for global handlers

### Custom error page

* To disable the default **White label error page**   
  for a Spring Boot application:
  + We must save **error.html** file in resources/templates directory, it'll automatically be picked up by the default Spring

### ErrorController Interface

* Spring Boot maps **/error** to **BasicErrorController** which populates model with error attributes   
  and then returns 'error' as the view name
* To replace BasicErrorController with our own custom controller which can map to /error, we need to **implement** **ErrorController** interface

@Controller

public class MyErrorController implements ErrorController {

@RequestMapping

@ResponseBody

public String handle(HttpServletRequest request){

//Some code ...

}

}

### HTTP Status Codes

* Unhandled exceptions during a request produce HTTP 500 response
* Any custom exception can be annotated with **@ResponseStatus**
  + Supports all HTTP status codes
  + When thrown and unhandled – produces error page with   
    appropriate response

@ResponseStatus(value = HttpStatus.NOT\_FOUND, reason = "Product was not found.")

public class ProductNotFoundException extends RuntimeException {

*// Exception definition*

}

* And the controller action, throwing the exception

@GetMapping("/products/details/{id}")

public ModelAndView productDetails(@PathVariable String id, ModelAndView modelAndView) {

Product product = this.productRepository.findProductById(id);

if(product == null) throw new ProductNotFoundException(id);

modelAndView.addObject("product", product);

return this.view("product/details", modelAndView);

}

### Controller-Based Error Handling

* You can define Controller-specific Exception Handlers
  + Annotated with **@ExceptionHandler** annotation
  + They work **only** for the **Controller** they are defined in
  + Can be annotated with **@ResponseStatus** to convert HTTP status
  + Can accept the **caught** **exception** as a **parameter**
  + Can return **ModelAndView** or **String** (view name)
  + Can catch **multiple** exception types

@ExceptionHandler({PersistenceException.class, TransactionException.class})

public ModelAndView handleDbExceptions(DatabaseException e) {

ModelAndView modelAndView = new ModelAndView("error");

modelAndView.addObject("message", e.getMessage());

return modelAndView;

}

View-------------------------------------------------------------------------------------------

<html>

<head>...</head>

<body>

<h1>An error occurred while processing your request!</h1>

<p th:text="|Error: ${message}|"></p>

</body>  
</html>

* Handler methods have **flexible signatures**
  + You can pass in servlet-related objects as parameters
    - **HttpServletRequest**
    - **HttpServletResponse**
    - **HttpSession**
    - **Principal**
* The **Model** or **ModelAndView** cannot be a parameter though
  + Instead of passing it, you have to setup it inside the method
  + Nevertheless, this is not an issue because the **IoC** **container** would have done the same (pass an **empty instance**)
* It is not a good practice for full error **stacktraces** to be exposed
  + Your users don't need to see ugly exception web-pages
  + You may even have security policies which **strictly forbid** any public exception info
  + Hide as **much information** as **possible** and present **User-friendly** error pages
  + For **testing** purposes you may view details
    - This may need an **environment** setup

### Global Application Exception Handling

* There is a way to achieve Global exception handling in Spring
  + This is done through the **@ControllerAdvise** annotation
* Any class annotated with **@ControllerAdvise** turns into an **interceptor-like** controller:
  + Enables **global exception handling**
  + Enables **model enhancement** methods
* In **@ControllerAdvice** classes you still use **@ExceptionHandler**
  + However, this time it refers to the whole application
  + The error handling is not limited only to a specific controller

**@ControllerAdvice**

**public class GlobalExceptionHandler {**

**@ExceptionHandler({TransactionException.class, PersistenceException.class})**

**public ModelAndView handleDatabaseErrors(DatabaseException e) {**

**ModelAndView modelAndView = new ModelAndView("index");**

**modelAndView.addObject("message", e.getMessage());**

**modelAndView.addObject("stack", {...} */\* Formatted Stack Trace \*/*);**

**return modelAndView;**

**}}**

### Global Exception Handling (REST)

* **RESTful requests** may also generate unexpected exceptions
  + HTTP Error response codes are a good choice
  + However sometimes you might need more than just a status
    - **Customized Error Object**, which can be presented on the Client
    - **Limited Information** returned to the Client
* You can customize the **Error Response** by introducing a class
  + The **Error Handler** itself remains the same as in casual web apps

public class ErrorInfo {

public final String url;

public final String ex;

public ErrorInfo(String url, Exception ex) {

this.url = url;

this.ex = ex.getLocalizedMessage();

}

}

@ControllerAdvice

public class GlobalRESTExceptionHandler {  
 @ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)

@ExceptionHandler({TransactionException.class,

PersistenceException.class})

public @ResponseBody ErrorInfo handleRESTErrors(HttpServletRequest req,

DbException e) {

return new ErrorInfo(req.getRequestURL(), ex);

}

}

### Exception Techniques Use Cases

**What to Use When?**

* Spring offers **many** choices, when it comes to **error** handling
* Be **careful** **mixing** too **many** of these
  + You may not get the behavior you wanted
* There are some semantics, that should be followed, though
  + For custom exceptions, consider adding **@ResponseStatus** to then
  + For Controller-specific exceptions, **@ExceptionHandler** methods should be added alongside the actions
  + For all other exceptions, **@ExceptionHandler** methods in **@ControllerAdvice** classes should be implemented

# Events in Spring

## 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

### Spring Build-in Events

* **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 received

## Listening for Events

### Listening for Events

* There are ways to listens 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

### Using ApplicationListeners Example

* 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());  
 }}

### Using @EventListener

* 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!");

}

### Listening for Multiple Events

* 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 When Adding Listener

* 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);

...

### Transaction Bound Events

* + 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**
* 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

### 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

public class MyCustomEvent extends ApplicationEvent {

private String msg;

public MyCustomEvent(Object source, String msg) {

super(source);

this.msg = msg;

}

... }

### Create Publisher

* 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 Listener

* 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 -%s!%n", myCustomEvent.getMsg());

}

}

## Scheduling Tasks

### 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());

}

### Enable Scheduling

* 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

### Caching

* 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>

### Enable Caching

* 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 }}

### @Cacheable

* 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) {

//...

}

### @CachePut

* 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() { //... }**

### @CacheEvict

* 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() { *//...* }

### Customize The auto-configured CacheManager

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

**@Component**

public class MyCacheCustomizer implements

CacheManagerCustomizer<ConcurrentMapCacheManager> {

**@Override**

public void customize(ConcurrentMapCacheManager cacheM){

cacheM.setCacheNames(asList("students","courses"));

} }