Spring Boot Middleware: Cheat Sheet-Aryan

1. Introduction to Middleware in Spring Boot

1.1 What is Middleware?

Middleware in **Spring Boot** is a powerful mechanism for intercepting and processing requests and responses between different components of an application. It enables implementation of cross-cutting concerns such as:

- Logging
- Authentication
- Security
- Request processing

Middleware ensures clean code by abstracting these concerns away from the core application logic.

1.2 Key Middleware Concepts

Concept	Description
Interceptors	Components that can modify or interrupt the request-response cycle.
Filters	Low-level servlet container mechanisms for request preprocessing.
Handlers	Custom request processing components for business logic execution.
Advice	AOP (Aspect-Oriented Programming) mechanisms for handling cross-cutting tasks.

---

2. Types of Middleware in Spring Boot

2.1 Servlet Filters

Servlet filters operate at the **servlet container level**, making them the most fundamental type of middleware.

2.1.1 Characteristics

- Implements `javax.servlet.Filter` interface

- Executes **before** a request reaches the dispatcher servlet
- Can **modify** request and response objects
- Ideal for low-level preprocessing tasks (e.g., logging, compression, security)

2.1.2 Example Implementation

```
@Component
public class CustomAuthenticationFilter implements Filter {
   @Override
   public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
         throws IOException, ServletException {
      HttpServletRequest httpRequest = (HttpServletRequest) request;

      // Pre-processing logic (e.g., authentication or logging)
      System.out.println("Request intercepted: " + httpRequest.getRequestURI());

      chain.doFilter(request, response); // Continue filter chain

      // Post-processing logic
      System.out.println("Response processed.");
   }
}
```

2.2 Spring Interceptors

Spring MVC Interceptors provide **higher-level control** and operate within the Spring MVC framework.

2.2.1 Key Features

- Implements `HandlerInterceptor` interface
- Operates **before** and **after** handler execution
- Context-aware: has access to handler and request information

---

2.2.2 Interceptor Lifecycle Methods

Method	Execution Point
`preHandle()`	Before the handler/controller method execution.
`postHandle()`	After the handler method execution but before rendering the view.

`afterCompletion()`	After the complete request processing.

2.2.3 Example Configuration

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
   @Override
   public void addInterceptors(InterceptorRegistry registry) {
      registry.addInterceptor(new CustomInterceptor())
            .addPathPatterns("/api/**")         // Include paths
            .excludePathPatterns("/public/**");  // Exclude paths
   }
}

public class CustomInterceptor implements HandlerInterceptor {
   @Override
     public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) {
      System.out.println("Interceptor: Before handler execution");
      return true; // Proceed with request
   }
}
```

2.3 Spring Boot Actuator Middleware

Spring Boot Actuator simplifies application **monitoring and management**.

2.3.1 Middleware Features

- **Health Checks**: Monitor application health via `/actuator/health`
- **Metrics Collection**: Access performance and usage metrics
- **Custom Endpoints**: Extend Actuator with custom management endpoints

---

2.4 AOP-Based Middleware

Aspect-Oriented Programming (AOP) enables middleware functionality via **aspects** and **advice**.

2.4.1 Types of Advice

Advice	Execution

`@Before`	Before method execution.
`@After`	After method execution.
`@Around`	Wraps the entire method execution.
`@AfterReturning`	After successful method completion.
`@AfterThrowing`	After a method throws an exception.

2.4.2 Example Aspect

```
@Aspect
@Component
public class LoggingAspect {
   @Before("execution(* com.example.service.*.*(..))")
   public void logBefore(JoinPoint joinPoint) {
      System.out.println("Executing method: " + joinPoint.getSignature());
   }
}
```

---

3. Middleware Processing Flow

3.1 Request Processing Sequence

1. Servlet Filters
2. Spring Interceptors (preHandle)
3. Controller/Handler Method Execution
4. Spring Interceptors (postHandle)
5. View Rendering
6. Spring Interceptors (afterCompletion)
7. Servlet Filters (response filters)

---

4. Best Practices

4.1 Middleware Design Principles

- **Single Responsibility**: Focus middleware on one task.
- **Performance Efficiency**: Avoid blocking operations.

- **Error Handling**: Ensure middleware handles exceptions gracefully.
- **Logging**: Implement consistent logging for traceability.

4.2 Common Use Cases

- **Authentication/Authorization**: Validating user access.
- **Request Logging**: Capturing incoming and outgoing requests.
- **Rate Limiting**: Preventing abuse through API throttling.
- **Cross-Origin Resource Sharing (CORS)**: Managing cross-domain requests.
- **Performance Monitoring**: Tracking response times and performance metrics.

---

5. Advanced Middleware Techniques

5.1 Dynamic Middleware Registration

Register middleware dynamically using Spring's `FilterRegistrationBean`.

```
@Bean
public FilterRegistrationBean<CustomFilter> registerFilter() {
   FilterRegistrationBean<CustomFilter> registrationBean = new FilterRegistrationBean<>();
   registrationBean.setFilter(new CustomFilter());
   registrationBean.addUrlPatterns("/api/*");
   registrationBean.setOrder(Ordered.HIGHEST_PRECEDENCE);
   return registrationBean;
}
```

5.2 Conditional Middleware

Leverage `@ConditionalOnProperty` to activate middleware based on environment properties.

```
@ConditionalOnProperty(name = "custom.filter.enabled", havingValue = "true")
public class ConditionalFilter implements Filter {
   // Custom logic
}
```

---

6. Performance Considerations

6.1 Middleware Overhead

- Avoid complex processing in middleware
- Implement non-blocking operations
- Use caching for repetitive tasks

6.2 Execution Order

Control middleware execution using:

- `@Order` annotation
- `setOrder()` method in filter configuration

---

7. Security Implications

7.1 Security Middleware Patterns

- **JWT Token Validation**: Validate JSON Web Tokens in requests.
- **CSRF Protection**: Ensure secure POST operations.
- **Input Sanitization**: Protect against injection attacks.
- **CORS**: Configure cross-origin resource sharing.

---

8. Troubleshooting Middleware

8.1 Common Issues

- Middleware chain interruption
- Unintended modifications to request/response
- Performance degradation

8.2 Debugging Techniques

- Enable detailed logging for middleware components
- Use Spring Boot Actuator for real-time monitoring
- Leverage profilers for performance insights
