Real Life Aspects

Eberhard Wolff
http://ewolff.com
eberhard.wolff@gmail.com

pluralsight
hardcore developer training

Real Life Aspects

- AOP is a powerful
- Which challenges can be solved with AOP?

- Spring's aspect library contains some interesting aspect

- But can we do more?

- Aspects to solve everyday issues
- Inspiration for your own aspects

Retry

- Retry a method call if it fails
- Can help to resolve transient failures

```
@Aspect
@Component
public class RetryAspect {

  @Around("execution(* com.ewolff.retry..*Service.*(..))")
  public Object retry(ProceedingJoinPoint joinPoint)
   throws Throwable {
    try {
      return joinPoint.proceed();
    } catch (Throwable e) {
      return joinPoint.proceed();
    }
  }

}
```

Don't Use Retry in Real Life!

- **If a service is not accessible:**
  - Calls are buffered
  - Calls pile up
- **The service comes up again**
  - and is immediately swamped with request
  - and will probably go down again
- **Not very smart**
- **In particular if you are dealing with external systems**

Circuit Breaker

- From "Release It" by Michael T. Nygard

- If an error occurs the circuit breaker breaks the circuit
- i.e. service is not called any more
- Instead the exception is immediately forwarded to the caller
- After a while the service is called again
- If the call succeeds the circuit breaker is closed again

- Several policies possible when the server should be called again
- E.g. retry after some time
- E.g. retry after 10 calls

Circuit Breaker

```
@Component
@Scope("prototype")
@Aspect(
"perthis(circuitbreaker.CircuitBreakerAspect.circuitBreakerMethods())")
public class CircuitBreakerAspect {
   @Pointcut(
    "execution(@circuitbreaker.CircuitBreaker * *(..))")
   public void circuitBreakerMethods() {}

   private AtomicInteger counter = new AtomicInteger();
   private Throwable throwable;

}
```

All other aspect so far were singletons

This aspect store the number of failed calls per method

Expressed with perthis expression

Spring scope prototype – i.e. multiple instances possible

Circuit Breaker

```
@Component
@
@
pu
    (
    I
    I
    I
}
```

```
@Around(
"circuitbreaker.CircuitBreakerAspect.circuitBreakerMethods()"
)
public Object retry(ProceedingJoinPoint joinPoint) throws Throwable {
  try {
    if (counter.get() == 0) {
      return joinPoint.proceed();
    }
    if (counter.incrementAndGet() == 10) {
      Object result = joinPoint.proceed();
      counter.set(0);
      return result;
    }
   } catch (Throwable throwable) {
     this.throwable = throwable;
     counter.set(1);
   }
   throw this.throwable;
}
```

Circuit Breaker: Possible Improvements

- **After a failure slowly ramp up**

- **Create a STRATEGY**
- **Can decide how the Circuit Breaker should behave:**
  - isAvailable() : boolean
  - reportSuccess()
  - reportFailure()
  - STRATEGY can be implemented in many ways.

- **Use some kind of fallback**
  - Default value
  - Simplified service
  - Cache

JPA / JDBC

- Problem: JPA contains a cache
- i.e. changes are not immediately propagated to the database
- JDBC calls might see incorrect data
- Solution: Flush EntityManager every time a JDBC call happens

```
@Component
@Aspect
public class JPAStateSynchronizer {

  @PersistenceContext
  private EntityManager entityManager;

  @Before(
   "execution(* org.springframework.jdbc.core.JdbcTemplate.*(..) )")
  public void flush() {
    if (entityManager != null) {
      entityManager.flush();
    }
  }

}
```

Exception Handling

- Exception can be logged in a centralized place
- No more lost exceptions!

```
@Component
@Aspect
public class ExceptionHandling {
  Logger logger = LoggerFactory.getLogger(ExceptionHandling.class);

  @AfterThrowing(pointcut = "SystemArchitecture.Repository()",
   throwing = "ex")
  public void logDataAccessException(DataAccessException ex) {
    logger.error("Problem in Repositories", ex);
  }

  @AfterThrowing(pointcut =
   "SystemArchitecture.Repository() || SystemArchitecture.Service()",
    throwing = "ex")
   public void logRuntimeException(RuntimeException ex) {
     logger.error("RuntimeException", ex);
   }

}
```

First Failure Data Capture (FFDC)

- Idea: Get as much data from the first exception as possible
- Store all information about the call stack in a ThreadLocal

```
@Component
@Aspect
public class FirstFailureDataCaptureAspect {

  private ThreadLocal<CallContext> callContext =
   new ThreadLocal<CallContext>();

  private CallContext getCallContext() {
    CallContext result = callContext.get();
    if (result == null) {
      callContext.set(new CallContext());
      result = callContext.get();
     }
     return result;
  }

}
```

First Failure Data Capture (FFDC)

- Idea: Get as much data from the first exception as possible
- Store all information about the call stack in a ThreadLocal
- Delegate all Advices to the ThreadLocal

```
@Component
@Aspect
public class

  private Thr
   new Threa

  private Cal
    CallConte
    if (result =
      callConte
      result = c
    }
    return re
  }

}
```

```
@AfterThrowing(value = "SystemArchitecture.Service()",
 throwing = "ex")
public void afterThrowing(Throwable ex) {
  getCallContext().afterThrowing(ex);
}

@Before("SystemArchitecture.Service()")
public void before(JoinPoint joinPoint) {
  getCallContext().before(joinPoint);
}
            @AfterReturning(pointcut="SystemArchitecture.Service()",
  returning="result")
public void afterReturning(Object result) {
  getCallContext().afterReturning(result);
}
```

FFDC: Design

- Aspect delegates to other objects

- Aspect can used differently scoped object to store data
- E.g. ThreadLocal

- Object easier to test
- E.g. using unit tests
- Might also be used without aspects

Similar Ideas to FFDC

- **FFDC records each call and does something**
- **i.e. FFDC information used to understand a possible exception**
- **Might be used for other purposes**
  - Send information from a system to another to mirror activities
  - Auditing: Who did what when?
  - Undo/Redo in GUIs
  - …
- **Application specific monitoring**
  - Based on business data
  - Specific methods only
  - …

Filter

- Customer may only see his Account
- Account is a domain object – no Spring Bean
- How can you solve it using Spring AOP?

```
@Aspect
@Component
public class AccountFilterAspect {

  private ThreadLocal<Customer> currentCustomer =
   new ThreadLocal<Customer>();

  private Customer getCurrentCustomer() {
    return currentCustomer.get();
  }

  public void setCurrentCustomer(Customer customer) {
    currentCustomer.set(customer);
  }

  public void clearCurrentCustomer() {
    currentCustomer.set(null);
  }
```

Filter

- Customer may only see his Account
- Account is a domain object – no Spring Bean
- How can you solve it using Spring AOP?

```
@Aspect
@Com
public
  priva
   new
  priva
    retu
  }
  publi
    curr
  }
  publi
    curr
  }
```

```
@Around("execution(com.ewolff.domain.Account *(..))")
public Object filterAccount(ProceedingJoinPoint joinPoint)
 throws Throwable {
  Customer customer = getCurrentCustomer();
  if (customer == null) {
    return null;
  }
  Account result = (Account) joinPoint.proceed();
  if (customer.getFirstname().equals(result.getFirstname())
    && customer.getName().equals(result.getName())) {
   return result;
  } else {
    return null;
  }
}
```

Similar Ideas to Filter

- **Spring Security supports this approach**
- **General modifications to return values**
  - Transform results
  - …
- **Or filter / manipulate argument**
- **A way to manipulate domain objects using Spring AOP**

Summary

- **Can manipulate how a call is executed**
  - Retry
  - Circuit Breaker
  - Cache
- **Exception Handling**
- **Record calls**
  - First Failure Data Capture
  - Auditing etc
- **Filter / manipulate parameters / return value**
- **Offers a lot of interesting possibilities**
- **Some patterns can be implemented using aspects http://hannemann.pbworks.com/w/page/16577895/Design%20Patterns#Downloads**