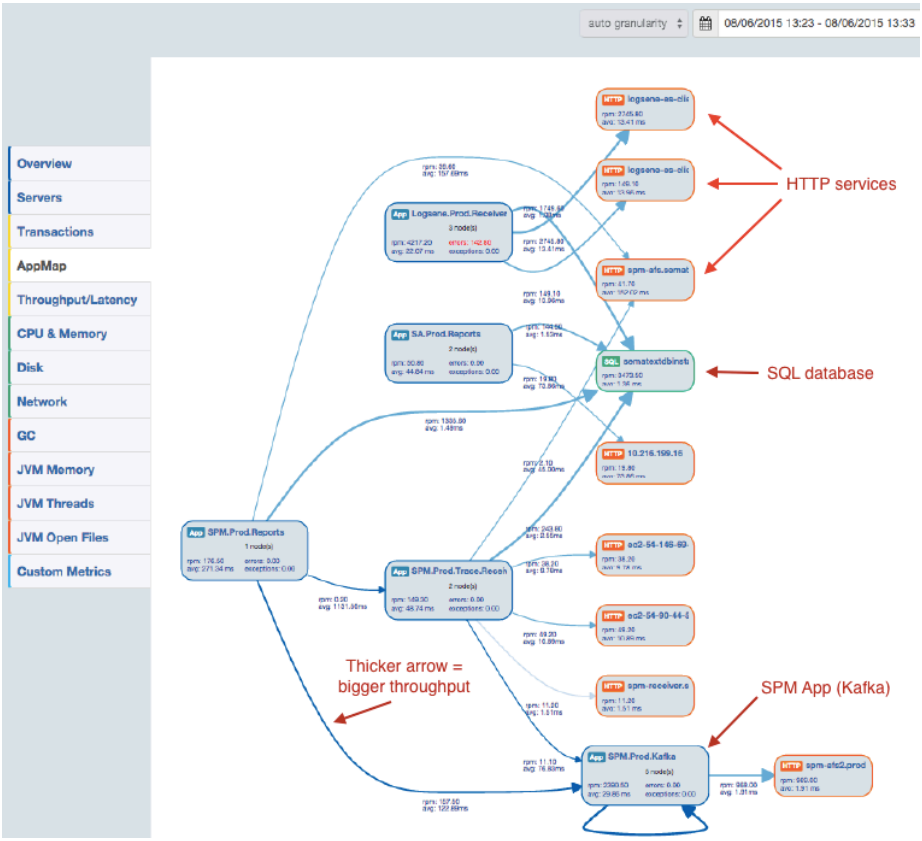

What is Transaction Tracing

Transaction tracing lets one trace code execution from beginning to end. It also enables Database Operations reporting. In case of SPM, transaction tracing can also cross applications, networks, and servers. For example, you can trace code execution from a beginning of an HTTP request made to a web application through any calls this web application makes to relational, NoSQL, or any other databases or backend servers and services like Elasticsearch or Solr or Kafka, etc., all the way to the response the application returns to the original caller. Transaction tracing is not limited to HTTP requests - one can also trace arbitrary applications, including backend apps, command-line apps, batch jobs like MapReduce, etc. Such tracing makes it possible to find performance bottlenecks in running code, whether in production or some other environment.

Starting with version 1.24.10, the SPM client provides ability to track application transactions, thus making it easier to find bottleneck in running applications and troubleshoot performance issues.

Notes:

- Transaction Tracing requires SPM monitor running in embedded mode (in-process/javaagent).
- Enabling/disabling the tracing agent requires SPM monitor restart, which means it requires the restart of the application running the embedded SPM monitor.
- Enabling transaction traces adds only about 1% CPU overhead.
- Transaction Tracing is different from On Demand Profiling.

Resources:

- http://blog.sematext.com/2015/08/03/transaction-tracing-performance-monitoring/
- http://blog.sematext.com/2015/08/05/transaction-tracing-reports/
- http://blog.sematext.com/2015/08/06/introducing-appmap/

Enable Tracing

To enable tracing edit the monitor configuration file - /opt/spm/spm-monitor/conf/spm-monitor-config-*token*−{jvm}.properties:

```
# enable tracing agent
SPM_MONITOR_TRACING_ENABLED=true

# capture only transaction that took longer than this many milliseconds
# default value is 50ms
# WARNING: setting this too low will increase the agent overhead
MIN_TRANSACTION_DURATION_RECORD_THRESHOLD=50
```

Supported Technologies

Currently, only Java applications can be traced. The agent is compiled with Java 1.6 target, so applications using Java 1.6 and up are supported.

Application servers:

- Apache Tomcat - 6.0.x, 7.0.x, 8.0.x
- Jetty - 7.x, 8.x, 9.x
- Generic servlet container (without cross-application tracing support)

Technologies:

- Generic JDBC driver support
- JPA
- Spring
- Elasticsearch TransportClient
- java.net.URL
- Apache HttpClient 3.x, 4.x
- JAX-RS

Custom Pointcuts

The built-in extension mechanism can be used to instrument custom method calls that are not handled by the SPM tracing agent out of the box. An extension is set of pointcuts described in XML files placed in /opt/spm/spm-monitor/ext/tracing/ directory. The tracing agent scans all .xml files in that directory when it starts.

Below is the description of the XML format:

```
<instrumentation-descriptor name="descriptor-name">
  <pointcuts>
     <pointcut name="pointcut-1" [entry-point="true"] [transaction-name="custom-transaction-
         <class name="com.company.example.ClassName"/>
         <class name="com.company.example.ClassName$InnerClassName"/>
         <constructor signature="com.company.example.Service(int intParam, java.lang.String
         <method signature="java.lang.String com.company.example.Service#getUserName(com.co
     </pointcut>
  </pointcuts>
</instrumentation-descriptor>
```

An extension represents a set of pointcuts. Each pointcut defines a set of rules to match joinpoints which will be instrumented. A joinpoint is a particular method/constructor that will be instrumented if it satisfies given rules.

A pointcut has two optional parameters:

- **entry-point** - if set to true a new transaction will be created for each joinpoint.
- **transaction-name** - transaction name will inherit the joinpoint name unless this parameter is specified

The following rules are supported:

- **class** - all non-static methods of this class. The 'name' parameter should be a fully qualified java class name (see http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)
- **constructor** - constructor for a given class which matches given parameter types. The 'signature' should specify a fully qualified java class name followed by a list of comma-separated parameters inside parenthesis (e.g. (int foo, java.lang.String bar))
- **method** - method for a given class and all its subclasses whose methods match the specified method name, return type, and parameters.

Examples

An extension that enables instrumentation of *all* methods in given classes.

```
<instrumentation-descriptor name="spring-petclinic-extension">
  <pointcuts>
    <pointcut name="jpa-repository">
      <class name="org.springframework.samples.petclinic.repository.jpa.JpaOwnerRepositoryIm
      <class name="org.springframework.samples.petclinic.repository.jpa.JpaPetRepositoryImpl
      <class name="org.springframework.samples.petclinic.repository.jpa.JpaVetRepositoryImpl
      <class name="org.springframework.samples.petclinic.repository.jpa.JpaVisitRepositoryIm
    </pointcut>
    <pointcut name="jdbc-repository">
      <class name="org.springframework.samples.petclinic.repository.jdbc.JdbcOwnerRepository
      <class name="org.springframework.samples.petclinic.repository.jdbc.JdbcPetRepositoryIm
      <class name="org.springframework.samples.petclinic.repository.jdbc.JdbcVetRepositoryIm
      <class name="org.springframework.samples.petclinic.repository.jdbc.JdbcVisitRepository
    </pointcut>
  </pointcuts>
</instrumentation-descriptor>
```

In the extension below a new transaction is created each time methods CustomDescriptionTracerTest$Job\#doJob and CustomDescriptionTracerTest$CustomJob\#doJob are invoked. For the former the transaction name will be 'com.sematext.spm.tracing.agent.tracer.CustomDescrip while for the latter we set it to 'CustomTransactionName' using the transaction-name attribute.

```
<instrumentation-descriptor name="custom-description-tracer-test">
  <pointcuts>
    <pointcut name="method-pointcut-test">
      <method signature="void com.sematext.spm.tracing.agent.tracer.CustomDescriptionTracerT
      <method signature="void com.sematext.spm.tracing.agent.tracer.CustomDescriptionTracerT
    </pointcut>
    <pointcut name="whole-class-test">
      <class name="com.sematext.spm.tracing.agent.tracer.CustomDescriptionTracerTest$Service
    </pointcut>
    <pointcut name="constructor-test">
      <constructor signature="com.sematext.spm.tracing.agent.tracer.CustomDescriptionTracerT
      <constructor signature="com.sematext.spm.tracing.agent.tracer.CustomDescriptionTracerT
      <constructor signature="com.sematext.spm.tracing.agent.tracer.CustomDescriptionTracerT
    </pointcut>
    <pointcut name="job" entry-point="true">
      <method signature="void com.sematext.spm.tracing.agent.tracer.CustomDescriptionTracerT
    </pointcut>
    <pointcut name="custom-named-job" entry-point="true" transaction-name="CustomTransactior
      <method signature="void com.sematext.spm.tracing.agent.tracer.CustomDescriptionTracerT
    </pointcut>
  </pointcuts>
```

```
</instrumentation-descriptor>
```

To disable a pointcut definition from the /opt/spm/spm-monitor/ext/tracing/ directory simply rename the file so it doesn't have the .xml extension. For example, you might rename foo.xml to foo.xml.disabled. After that, restart your application.

IMPORTANT: instrumenting methods that are invoked frequently and execute quickly can and will increase the agent overhead.

Transactions Naming

To name web transactions (i.e., transactions triggered by an HTTP request) the SPM Java tracing agent uses method signature of entry point as the transaction name. Entry point methods can be last filter/servlet executed in a chain, or a Spring handler method name (i.e., a method name with the '@RequestHandler' annotation). Alternatively, transaction names can be redefined using servlet config. For example, here we name them "WorkerTransaction":

```
<servlet>
  <servlet-name>WorkerServlet</servlet-name>
  <servlet-class>phi.WorkerServlet</servlet-class>
  <init-param>
    <param-name>com.sematext.spm.tracing.agent.TRANSACTION_NAME</param-name>
    <param-value>WorkerTransaction</param-value>
  </init-param>
</servlet>
```

Precedence rules for resolving transaction names:

Precedence

Naming

3

Servlet config parameter

2

Spring handler

1

Servlet/Filter

Non-web transactions can be (re)named using the transaction-name attribute as described in Custom Pointcuts section.

Tracing Scala Apps

Yes, you can trace transactions for Scala apps, too! In order to define custom pointcuts for a Scala app you need to follow the convention Scala uses to generate JVM classes. Below you can find a toy example that covers all basic cases:

```
trait UserService {
  def getUsers(): List[String]
}
class UserServiceImpl extends UserService {
  def getUsers(): List[String] =
    "Max_Rockatansky"::"Nux"::"Joe"::"Furiosa"::Nil
}
sealed trait Type {
  def typeName: String
}
case object Original extends Type {
  val typeName: String = "original"
}
case object Retweet extends Type  {
  val typeName: String = "retweet"
}
case class Tweet(text: String, t: Type = Original) {
  def vowelsCount: Int = {
    text.toLowerCase.filter("aeiou".toCharArray.contains).size
  }
}
class TweetService {
  def getTweets(user: String): List[Tweet] =
    Tweet("What a lovely day!")::Nil
}
object StatisticsService extends App {
  val userService = new UserServiceImpl
  val tweetService = new TweetService
  def vowelsCount(): Int = {
    val count = for {
      user <- userService.getUsers()
      tweet <- tweetService.getTweets(user) if tweet.t.typeName == "original"
    } yield tweet.vowelsCount
    count.sum
  }
  def serve(): Unit = {
    println(vowelsCount())
    Thread.sleep(1000)
    serve()
  }
```

```
  serve()
}
```

Custom pointcuts definition:

```
<instrumentation-descriptor name="scala">
  <pointcuts>
    <pointcut name="foo" entry-point="true">
      <method signature="int StatisticsService$#vowelsCount()"/>
    </pointcut>
    <pointcut name="userService">
      <method signature="scala.collection.immutable.List UserService#getUsers()"/>
    </pointcut>
    <pointcut name="tweetService">
      <method signature="scala.collection.immutable.List TweetService#getTweets(java.lang.St
    </pointcut>
    <pointcut name="type">
      <method signature="java.lang.String Original$#typeName()"/>
      <method signature="java.lang.String Retweet$#typeName()"/>
    </pointcut>
  </pointcuts>
</instrumentation-descriptor>
```

Non-JVM Transaction Tracing

TBD