**Springboot Distributed transactions management with multiple resources using Best Effort 1 Phase Commit - ChainedTransactionManager**

We commonly use the Java Transaction API and the XA protocol for distributed transactions in Spring, which provides the best guarantees to recover in case of an outage. Spring JtaTransactionManager synchronizes the transaction using 2 phase commit to ensure that all resources agree on the outcome of a transaction before it ends. This is an expensive operation. Spring also supports other mechanisms for distributed transaction management depending upon the types of resources used by the application and the program specific requirements of performance, safety, reliability, and data integrity. In this post, we will see an implementation of the Best Efforts 1 Phase Commit approach, which is utilized by many high volume, high-throughput transaction-processing systems to improve performance.

**Best Efforts 1 Phase Commit Pattern**

This pattern requires us to delay the commit of all resources as late as possible in a transaction so that the only thing that can go wrong is an infrastructure failure (not a business-processing error). Systems that implement this pattern tolerate the rare chance of an infrastructure failure in order to achieve better performance. Correctness of the data is guaranteed by designing idempotent services.

**ChainedTransactionManager**

Spring’s ChainedTransactionManager orchestrates transaction creation, commits and rollbacks to a list of delegating PlatformTransactionManagers.

[ChainedTransactionManager](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/transaction/ChainedTransactionManager.html#ChainedTransactionManager-org.springframework.transaction.PlatformTransactionManager...-)([PlatformTransactionManager](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/transaction/PlatformTransactionManager.html?is-external=true)... transactionManagers)

The configured instances will start transactions in the order given and commit/rollback in reverse order, which means the [PlatformTransactionManager](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/transaction/PlatformTransactionManager.html?is-external=true" \o "class or interface in org.springframework.transaction) most likely to break the transaction should be the last in the list configured. A [PlatformTransactionManager](https://docs.spring.io/spring/docs/5.0.x/javadoc-api/org/springframework/transaction/PlatformTransactionManager.html?is-external=true" \o "class or interface in org.springframework.transaction) throwing an exception during commit will automatically cause the remaining transaction managers to roll back instead of committing.

In this post, we will save information into two databases in a distributed transaction using a ChainedTransactionManager.

Here are the steps: -

1. Create a Springboot project.
2. Define data model.
3. Add datasource configurations to the properties file.
4. Create data repositories.
5. Configure datasources, transaction managers and data repositories for each database.
6. Create a ChainedTransactionManager passing all the transaction managers.
7. Create a Service that saves the information into multiple databases in a single transaction.
8. Write a Rest Controller to invoke the service method.
9. **Create a Springboot project**

<parent>

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

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

<version>1.3.5.RELEASE</version>

</parent>

1. **Define data model**

Our data model comprises of an EventMessage, and an EventFuture as follows: -

@Entity

@Table(name = "eventmessage")

**public** **class** EventMessage {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** Integer id;

@Column(name = "description", nullable = **false**, length=255)

**private** String description;

@Entity

@Table(name = "eventfuture")

**public** **class** EventFuture {

@Id

**private** Integer id;

@Column(name = "action", nullable = **false**, length=10)

**private** String action;

We would save both the entities into two separate databases.

1. **Add datasource configurations to the properties file**

eventmessage.datasource.url=jdbc:h2:mem:eventmessage

eventmessage.datasource.username=sa

eventmessage.datasource.password=

eventfuture.datasource.url=jdbc:h2:mem:eventfuture

eventfuture.datasource.username=sa

eventfuture.datasource.password=

1. **Create data repositories**

**public** **interface** EventMessageRepository **extends** JpaRepository<EventMessage, Integer> {

}

**public** **interface** EventFutureRepository **extends** JpaRepository<EventFuture, Integer> {

}

1. **Configure datasources, transaction managers and data repositories for each database**

@Configuration

@EnableJpaRepositories(basePackages = "com.example.repository.eventMessage", entityManagerFactoryRef = "eventMessageEntityManager", transactionManagerRef = "eventMessagePlatformTransactionManager")

@EnableConfigurationProperties(EventMessageDatasourceProperties.**class**)

**public** **class** EventMessageConfig {

@Autowired

**private** JpaVendorAdapter jpaVendorAdapter;

@Autowired

**private** EventMessageDatasourceProperties eventMessageDatasourceProperties;

@Primary

@Bean(name = "eventMessageDataSource")

**public** DataSource eventMessageDataSource() {

DataSourceBuilder dataSourceBuilder = DataSourceBuilder.*create*();

dataSourceBuilder.driverClassName("org.h2.Driver");

dataSourceBuilder.url(eventMessageDatasourceProperties.getUrl());

dataSourceBuilder.username(eventMessageDatasourceProperties.getUsername());

dataSourceBuilder.password(eventMessageDatasourceProperties.getPassword());

**return** dataSourceBuilder.build();

}

@Primary

@Bean(name = "eventMessageEntityManager")

**public** LocalContainerEntityManagerFactoryBean eventMessageEntityManager() **throws** Throwable {

LocalContainerEntityManagerFactoryBean entityManager = **new** LocalContainerEntityManagerFactoryBean();

entityManager.setDataSource(eventMessageDataSource());

entityManager.setJpaVendorAdapter(jpaVendorAdapter);

entityManager.setPackagesToScan("com.example.domain.eventMessage");

entityManager.setPersistenceUnitName("eventMessagePersistenceUnit");

**return** entityManager;

}

@Bean(name="eventMessagePlatformTransactionManager")

**public** PlatformTransactionManager eventMessagePlatformTransactionManager(@Qualifier("eventMessageEntityManager") EntityManagerFactory entityManagerFactory) {

**return** **new** JpaTransactionManager(entityManagerFactory);

}

}

@Configuration

@EnableJpaRepositories(basePackages = "com.example.repository.eventFuture", entityManagerFactoryRef = "eventFutureEntityManager", transactionManagerRef = "eventFuturePlatformTransactionManager")

@EnableConfigurationProperties(EventFutureDatasourceProperties.**class**)

**public** **class** EventFutureConfig {

@Autowired

**private** JpaVendorAdapter jpaVendorAdapter;

@Autowired

**private** EventFutureDatasourceProperties eventFutureDatasourceProperties;

@Bean(name = "eventFutureDataSource")

**public** DataSource eventFutureDataSource() {

DataSourceBuilder dataSourceBuilder = DataSourceBuilder.*create*();

dataSourceBuilder.driverClassName("org.h2.Driver");

dataSourceBuilder.url(eventFutureDatasourceProperties.getUrl());

dataSourceBuilder.username(eventFutureDatasourceProperties.getUsername());

dataSourceBuilder.password(eventFutureDatasourceProperties.getPassword());

**return** dataSourceBuilder.build();

}

@Bean(name = "eventFutureEntityManager")

**public** LocalContainerEntityManagerFactoryBean eventFutureEntityManager() **throws** Throwable {

LocalContainerEntityManagerFactoryBean entityManager = **new** LocalContainerEntityManagerFactoryBean();

entityManager.setDataSource(eventFutureDataSource());

entityManager.setJpaVendorAdapter(jpaVendorAdapter);

entityManager.setPackagesToScan("com.example.domain.eventFuture");

entityManager.setPersistenceUnitName("eventFuturePersistenceUnit");

**return** entityManager;

}

@Bean(name="eventFuturePlatformTransactionManager")

**public** PlatformTransactionManager eventFuturePlatformTransactionManager(@Qualifier("eventFutureEntityManager") EntityManagerFactory entityManagerFactory) {

**return** **new** JpaTransactionManager(entityManagerFactory);

}

}

1. **Create a ChainedTransactionManager passing all the transaction managers**

@Primary

@Bean(name="chainedTransactinManager")

**public** ChainedTransactionManager chainedTransactionManager(@Qualifier("eventMessagePlatformTransactionManager") PlatformTransactionManager ptmEventMessage,

@Qualifier("eventFuturePlatformTransactionManager") PlatformTransactionManager ptmEventFuture) {

**return** **new** ChainedTransactionManager(ptmEventMessage, ptmEventFuture);

}

1. **Create a Service that saves the information into multiple databases in a single transaction**

@Transactional(rollbackFor = Exception.**class**)

@Override

**public** **void** insert(EventMessage eventMessage, EventFuture eventFuture) **throws** Exception {

eventMessageRepository.save(eventMessage);

eventFuture.setId(eventMessage.getId());

eventFutureRepository.save(eventFuture);

}

1. **Write a Rest Controller to invoke the service method**

@Autowired EventService eventService;

@ResponseBody

@RequestMapping(value = "/event/{msg}", method = RequestMethod.***GET***)

**public** Map<String, Object> event(@PathVariable("msg") String msg) {

Map<String, Object> result = **new** HashMap<String, Object>();

**try** {

EventMessage m = **new** EventMessage();

m.setDescription(msg);

EventFuture f = **new** EventFuture();

f.setAction(m.getDescription().toUpperCase().startsWith("ERROR")?"TO-BE-DETERMINED":"SCHEDULED");

eventService.insert(m, f);

Assert.*notNull*(m.getId());

Assert.*notNull*(f.getId());

result.put("success", "true");

} **catch** (Exception e) {

e.printStackTrace();

result.put("success", "false");

result.put("exception", e.getMessage());

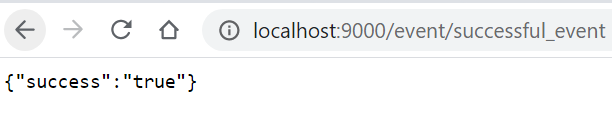
}

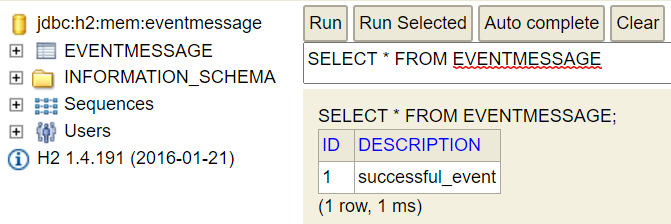
**return** result;

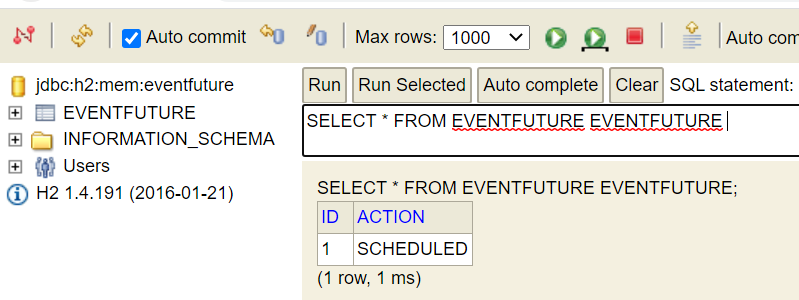
}

}

**Here is the output of a successfully committed transaction.**







**Here is the output of a failed transaction. If we send a message that starts with “ERROR”, we will see a database exception and the transaction will be rolled back.**

