Software development and other square topics

Using RabbitMQ, Spring AMQP and Spring Integration

Published: 31 Aug 2010

I've recently done a proof of concept using RabbitMQ, Spring AMQP and Spring Integration. This post will describe how each of these open source technologies along with some enterprise integration patterns were used to replace legacy code making it easier to maintain, easier to debug and faster.

Background

The system I'll be talking about is composed of various processes which work together to perform hundreds of calculations on any number of bonds entered into the system on a daily basis. Some bonds are imported via a nightly process and some are entered manually by traders via a web application.

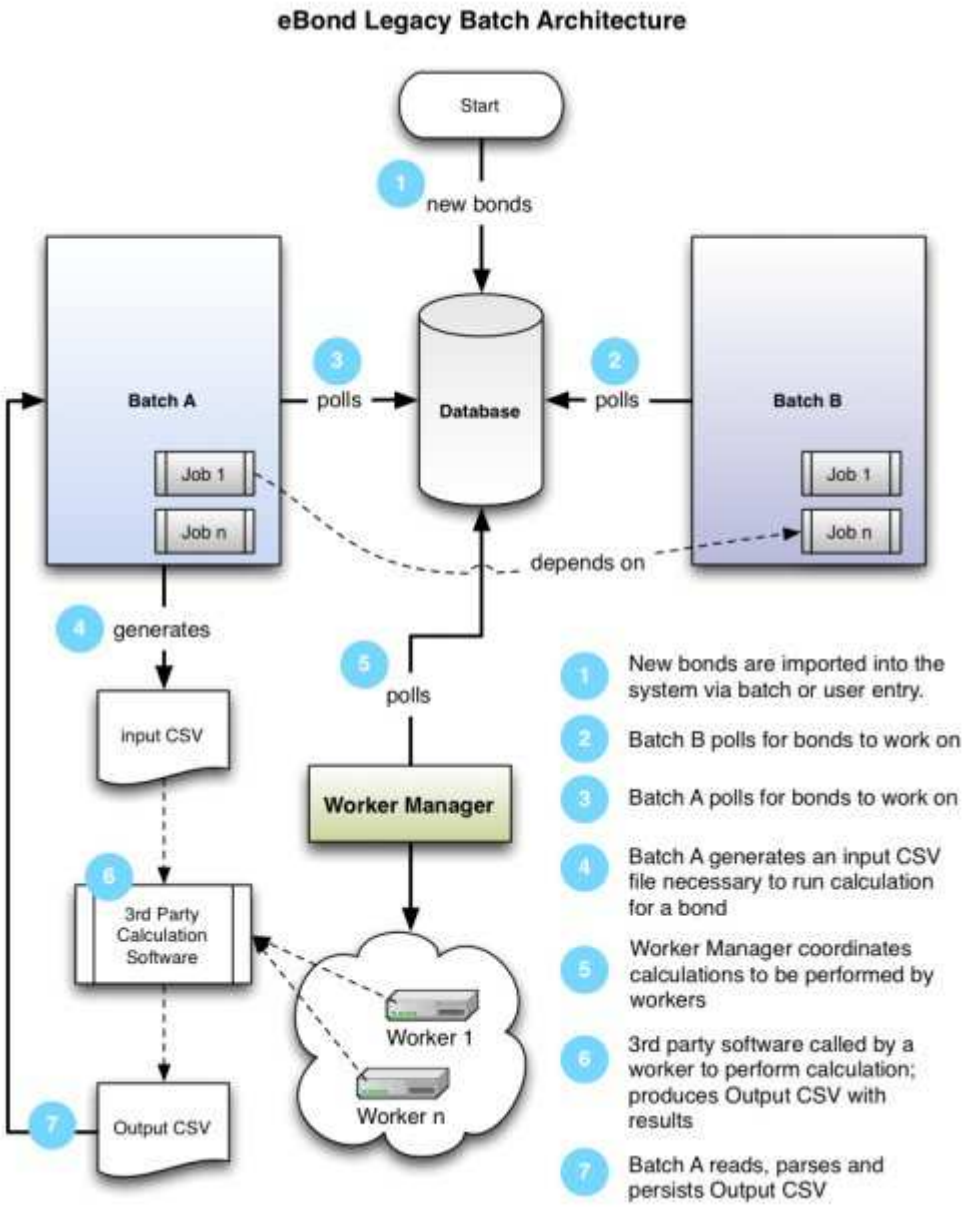
The idea for the proof of concept is to "prove" that the legacy code which has always been difficult to debug and equally difficult to test can be improved with newer technology and proven patterns. Over the years, the code has also become inefficient and needlessly taxing on other resources such as the primary database. The current state of this system is no one's fault and is unfortunately typical of a majority of enterprise software that becomes highly visible while incurring several years of fast iterations without the resources to properly test and evolve the architecture as needed.

The calculation processes can best be described as several dozen "jobs" that get processed in single-threaded loops which run continuously. While this is a nice and simple solution, it starts to bog down when you have hundreds of jobs running in the same loop. To get an idea of what I'm talking about, here's a snippet:

```
public void run() throws Exception {
        setUp();
        while (run) {
            try {
                if (continueWithProcess()) {
                    if (!existingRequestsCleaned) {
                        cleanUpExistingRequests();
                    }
                    prepare();
                    poll();
                } else {
                    doRest();
                }
            }
            catch (Exception e) {
                handleException(e);
            }
            Thread.sleep(sleepInSec * 1000);
        }
    }
```

This Gist brought to you by GitHub. BatchManagerSupport.java view raw

For each job, the process is described by the following steps:



After looking at the above illustration, you might have noticed a couple things. First, all the major components of the system rely on the database in order to do anything. Polling occurs everywhere via SQL queries in order to determine if there's calculations to do. Second, there's no easy way to determine which "jobs" depend on other jobs which gets tricky during debugging sessions.

Beyond what I've described here, there's a lot more that goes on in order to accommodate failover scenarios, coordinating the workers that do the actual calculations and re-running of calculations in the event of something going wrong. Each of the jobs above may also have multiple dependency jobs that span across two or more applications. Most of the coordination between these jobs and processes is handled via SQL queries. The database acts as a huge state machine AND single point of failure.

The Problems to Solve

As you can imagine, a system like the one described above combined with years of fast iterations (1 production release per week on average) will lead to issues. To that end, here are *some* of the improvements we wanted to make:

1. **Make the code easier to test.** So how do you test just one job out of dozens if they're all running in a big loop? Good question. The team did make some headway toward testing but it's less than ideal and requires simulation code and use of mock objects. Although it's possible to test the code, it's far from ideal and takes a good deal of setup to do. Testing code shouldn't be that difficult.

2. **Ease Debugging.** Debugging the legacy code is what some call a "black art" and requires a good deal of patience. Debugging one job involves starting up all the required machinery for the whole process and running various SQL queries to manually verify data.

3. **Make the system more efficient without taxing resources.** A single process continuously looping through code while bombarding the database for data. Everything, in effect, is driven by SQL queries and uses the database as a state machine. There are a couple of scenarios that come to mind that illustrate the inefficiency of the system.

4. **Eliminate the single point of failure.** Because the database acts as the state machine for all the processes in all the supporting applications, it's a single point of failure.

While these are not an exhaustive list, they summarize nicely the top priorities that needed to be solved.

The Solution: Leverage an Event-driven Architecture with Messaging

For the proof of concept I put together, I assembled an end-to-end process to do a single calculation based on a "new bond event". Here's a summary of the technologies used:

RabbitMQ

The idea of leveraging messaging to make systems more efficient is not a new one so there are tons of available solutions out there. While my evaluation of messaging systems wasn't exhaustive, I were aware that the firm was already using RabbitMQ with success. After doing my own research, I quickly became familiar with the advantages and flexibility of RabbitMQ. At the time of this writing one of the things that really impressed me was hearing that the whole thing was initially written in less than 8K lines of Erlang code!

RabbitMQ brands itself as a "complete and highly reliable enterprise messaging system based on the emerging AMQP standard." AMQP is basically a wire protocol which means you can more easily create clients for the AMQP implementation (RabbitMQ in this case) in just about any language out there. RabbitMQ can also easily be installed on all the major platforms including Windows and various Linux distributions. This was nice since our development is done on Windows and we deploy to Linux servers.

RabbitMQ also has broker clustering available out of the box, for free. Although I haven't tried clustering as part of this proof of concept, I imagine the implementation will be cleaner than the proprietary code written to coordinate the failover of batch processes.

Spring Integration

With almost no messaging experience, I was able to easily digest the AMQP specification, install RabbitMQ and start writing code with the provided Java client jar. However, after a few days of hacking it became clear that I needed something to be able to define the routing and transformations needed for the various messages and queues. For that, I looked at Apache Camel and Spring Integration. While Camel seemed to fit the bill, I found that Spring Integration (SI) was a better fit in part due to the fact that the documentation was better and my team was already proficient with Spring.

Spring Integration is a top level project in the Spring "portfolio" which "extends the Spring programming model to support the well-known Enterprise Integration Patterns. I was already familiar with the Enterprise Integration Patterns book by Gregor Hohpe and pleasantly surprised to see that a lot of the patterns he presented were implemented in a Spring project.

SI provided all the necessary pieces I needed to wire together services, files and other Spring applications in a uniform way which it implements in part through it's vast number of adapters. One adapter that was missing was one for AMQP! By being a general pain in the ass by asking lots of questions in the Spring forums, I found out that there was another new project coming out to serve this purpose, Spring AMQP!

Spring AMQP

I was able to start leveraging Spring AMQP before it was released by downloading from the Spring subversion repository. Mark Fisher, the lead for the project, was gracious and very helpful in addressing a couple of bugs I found including headers not getting passed through the inbound adapter. Once the initial bugs were ironed out, it was smooth sailing. Spring AMQP not only provided adapters for SI but also provided a nice abstraction on top of the RabbitMQ Java client library. I threw away much of the initial code I wrote for defining exchanges, queues and bindings in favor of the MUCH cleaner and concise Spring AMQP API using familiar Spring patterns. Here's an example of the configuration for consumers I wrote:

```
package spg.poly.consumer.conf;

import org.springframework.amqp.core.*;
import org.springframework.amqp.rabbit.config.AbstractRabbitConfiguration;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.connection.SingleConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;
import org.springframework.amqp.rabbit.listener.adapter.MessageListenerAdapter;
import org.springframework.amqp.support.converter.JsonMessageConverter;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import spg.poly.consumer.handler.PolyCalcHandler;
import spg.poly.core.domain.SpgExchange;
import spg.poly.core.domain.SpgQueue;

/**
 * @author mitosha Aug 13, 2010 10:56:15 AM
 */
@Configuration
@ImportResource("classpath:/polyConsumerPropertiesContext.xml")
public class ConsumerConfiguration extends AbstractRabbitConfiguration {

  @Value("${poly-consumer.version}")
  private String version;

  @Value("${poly-consumer.concurrentConsumers}")
  private int concurrentConsumers;

  @Value("${poly-consumer.rabbitHost}")
  private String rabbitHost;

  @Value("${poly-consumer.rabbitUser}")
  private String rabbitUser;

  @Value("${poly-consumer.rabbitPass}")
  private String rabbitPass;

  private static final String DBCMPLY_REQUEST_PREFIX = "dbcm-ply-request";

  public String getVersion() {
    return this.toString();
  }

  @Bean
  public SimpleMessageListenerContainer listenerContainer() {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer
    container.setConnectionFactory(connectionFactory());
    container.setQueues(dbcmPlyRequestQueue());
    container.setMessageListener(messageListenerAdapter());
    container.setConcurrentConsumers(concurrentConsumers);
    return container;
  }

  @Bean
  public MessageListenerAdapter messageListenerAdapter() {
    return new MessageListenerAdapter(polyCalcHandler(), jsonMessageConverter(
```

This class uses a combination of dependency injection, annotations and extends AMQP abstractions with reasonable defaults.

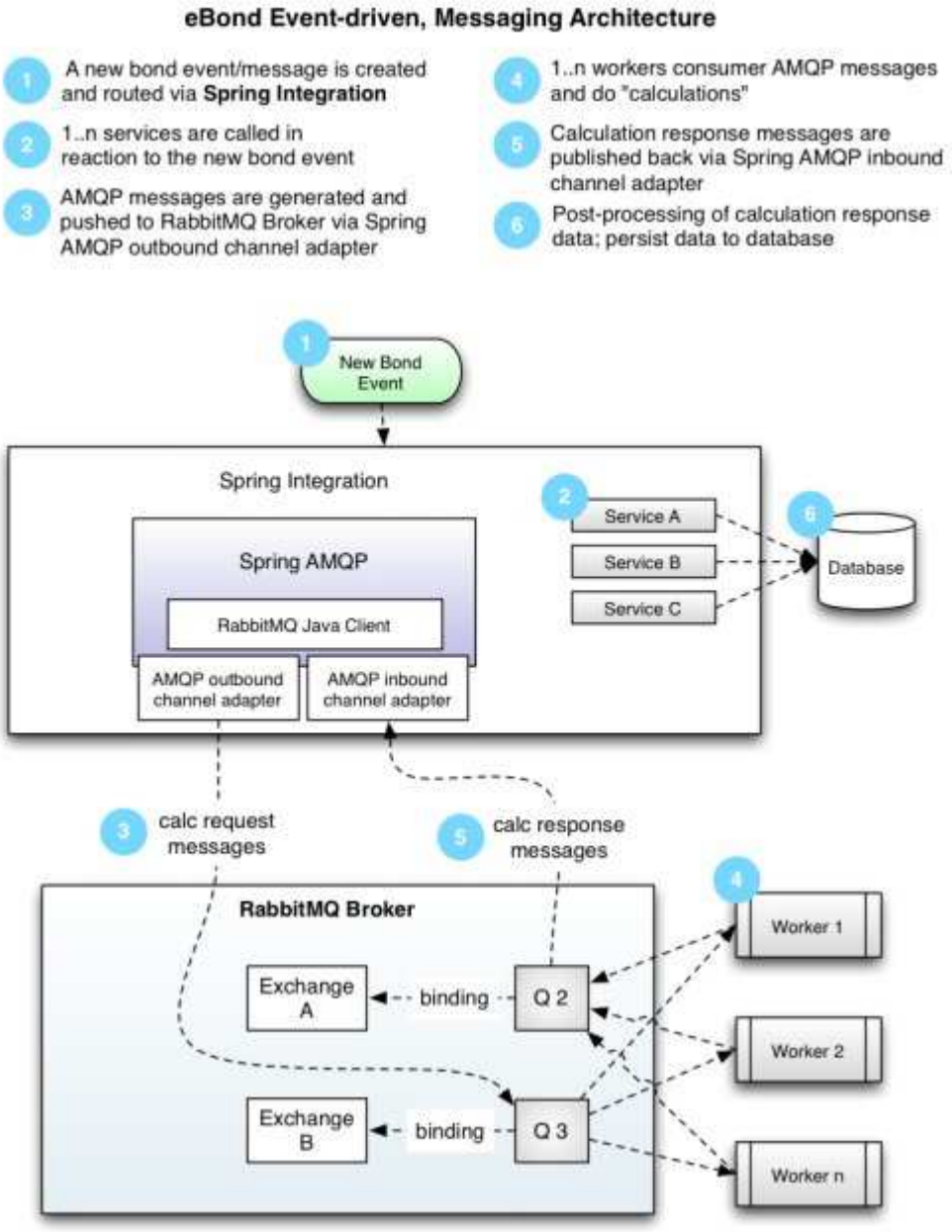
The AMQP adapters used in my Spring Integration context xml look like this:

```
<amqp:outbound-channel-adapter channel="amqpPolyRequestChannel"
                               exchange-name="dbcm-fanout-ex"
                               amqp-template="rabbitTemplate"/>

<amqp:inbound-channel-adapter channel="amqpPolyResponseChannel"
                              queue-name="dbcm-ply-response-q"
                              connection-factory="connectionFactory"/>
```


Summary

Below is an illustration of the proof of concept. The most noticeable change is that there's no longer any polling, anywhere. The end-to-end calculations on bonds are kicked off by an application event. From there, simple messages are created, transformed and routed via explicit rules instead of implicitly through a series of disconnected SQL queries. Overall, moving to a messaging system almost forces you to have loosely coupled code. Loosely coupled code means easier testing. Easier testing generally means you have the ability to more easily debug when things go wrong. The workers are consumers of a fanout exchange which means any number of the workers can go down without necessarily impacting the ability to do calculations. The RabbitMQ broker can be easily be clustered across any number of machines providing a higher level of fault tolerance.



Next Steps

The initial proof of concept has been successful but has not been without issues due in part because I'm using pre-release code and it's been a learning process. Going forward, there are more requirements to address such as clustering, monitoring and prioritizing messages based on the time of day. Once these and other issues are squashed, I'm confident the entire system will be more fault tolerant, easier to maintain, and have the flexibility to scale well into the future. More fun for subsequent posts!

3 people liked this.

Add New Comment

Type your comment here.

Post as ...

Showing 1 comments

Sort by Popular now Subscribe by email Subscribe by RSS

Simon Pettman 3 months ago

This is very similar to what I am attempting to achieve.

The way I see it is; if one process was going to do everything (a ridiculous notion made for the purposes of discussion) that the entire task can be very cleanly decomposed into Spring Integrations implementation of 'Enterprise Integration Patterns'.
But realistic systems are going to be disconnected by processes, machines etc.. So the 'work flow' for want of a better word needs to segregated into blocks that an 'execute together',using Spring Integration, and 'jumps. These 'jumps', at the logical disconnection points, will involve hoping on and off Rabbit MQ which can be accessed by Spring AMPQ.

My concern is whether Integration concepts such at the MessageHistory implementation will work when hopping about in the Rabbit world.

Like Reply

Trackback URL http://disqus.com/forums/

blog comments powered by **DISQUS**

Personal blog of **Shane Witbeck**. Most topics of this blog are about software development and documented solutions to problems.

PERSONALITY

INTJ - "Mastermind". Introverted intellectual with a preference for finding certainty. A builder of systems and the applier of theoretical models. 2.1% of total population.

PROJECTS

- buildr-plugin - an IntelliJ IDEA plug-in for Apache Buildr. {code}
- DigitalSanctum.com - personal blog. {code}
- DropPhotos.com - web image optimization service.
- rttwt - real-time Twitter client. {code}
- school-planner - home school management app. [code]
- TaskPlannr - todo list app. [code]

PRESENCE

- 

-