This page last changed on Mar 08, 2011 by patrik_nordwall.

.

Sculptor Event Driven Tutorial

Introduction

Event-Driven Architecture (EDA) is a good complement to Domain-Driven Design. We think EDA is an important ingredient for building scalable systems. It is also an enabler for designing loosely coupled modules and bounded contexts (business components).

In the simplest form you might need Observer pattern to decouple modules, swap direction of dependencies. When something happens in the core domain a notification should occur in supporting module (e.g. statistics). It is desired that core is independent of support modules. For this Sculptor provides DomainEvent and a mechanism to publish and subscribe through a simple event bus.

You will often benefit from communicating between bounded contexts (business components) in a loosely coupled way. E.g. send and receive messages with JMS. For this Sculptor provides implementations of the event bus that integrates with Apache Camel and Spring Integration. Same API as the simple event bus. It is also possible to mix Sculptor event bus with usage of ordinary endpoints provided by Camel or Spring Integration.

The simple event bus use plain synchronuos invocations to notify subscribers. If you need asynchronous notifications that is easy to achieve with Camel or Spring Integration.

The event bus API is extremly simple and you can easily plug-in your own implementation of the bus. It should be easy to integrate with Akka, Redis, GAE Task Queues, or additional integration products.

Currently there is no implementation of the event bus for EJB3 target implementation.

In addition to publish/subscribe Sculptor also has support for CQRS and EventSourcing.

Sample Domain

In this tutorial we have used Martin Fowler's shipping system example.

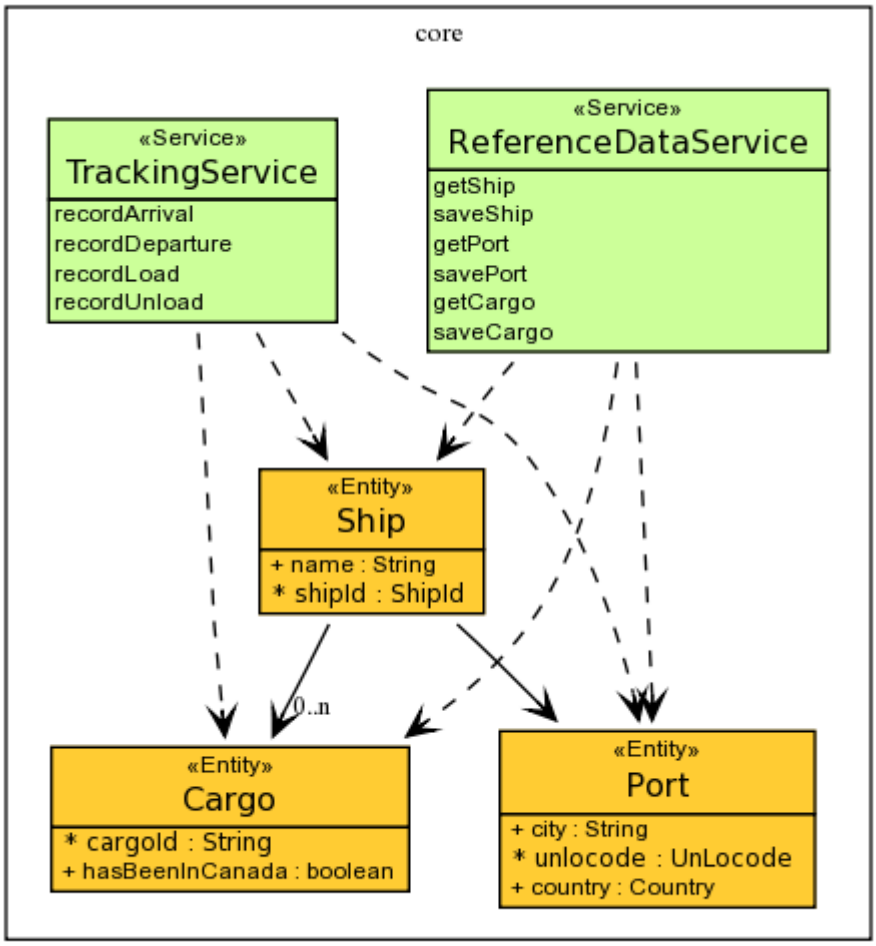
Sculptor design model (`model.btdesign`) of the core domain objects and services is defined like this:

```
Application Shipping {
   basePackage = org.sculptor.shipping
```

```
Module core {
   Service TrackingService {
      inject @ReferenceDataService
      inject @ShipRepository
      recordArrival(DateTime occurred, @Ship ship, @Port port);
      recordDeparture(DateTime occurred, @Ship ship, @Port port);
      recordLoad(DateTime occurred, @Ship ship, @Cargo cargo);
      recordUnload(DateTime occurred, @Ship ship, @Cargo cargo);
   }

   Service ReferenceDataService {
      getShip delegates to ShipRepository.findByKey;
      saveShip(@Ship ship) delegates to ShipRepository.save;
      getPort delegates to PortRepository.findByKey;
      savePort(@Port port) delegates to PortRepository.save;
      getCargo delegates to CargoRepository.findByKey;
      saveCargo(@Cargo cargo) delegates to CargoRepository.save;
   }

   Entity Ship {
      gap
      - ShipId shipId key
      String name
      - Port port nullable
      - Set<@Cargo> cargos

      Repository ShipRepository {
         gap
         inject @CargoRepository
         save;
         findByKey;
      }
   }

   BasicType ShipId {
      String identifier key
   }

   Entity Port {
      - UnLocode unlocode key;
      String city
      - Country country

      Repository PortRepository {
         save;
         findByKey;
      }
   }

   "United nations location code."
   BasicType UnLocode {
      String identifier key
   }

   enum Country {
      US,
      CANADA
   }

   Entity Cargo {
      gap
      String cargoId key
      boolean hasBeenInCanada

      Repository CargoRepository {
         save;
```

```
            findByKey;
        }
      }

    }
  }
```

Generated visualization looks like this:



Try with your own project

We encourage you to create your own sample project and try the different pieces hands-on as they are described in this tutorial. You create a new project with the archetype as described in Part 1 of Hello World Tutorial.

Thereafter you can copy the sample shipping domain model as defined in previous section and paste in `model.btdesign`.

Domain Events

In Domain Language Newsletter March 2010 we could read Eric Evans writeup on Domain Events.

> Over the last few years it has become clear that it is very useful to add a new pattern to the DDD "Building Blocks" (Entities, Value Objects, Services, etc.) – Domain Events. This pattern has been around for a long time. Martin Fowler wrote a good description .

DomainEvents are defined in Sculptor design model (`model.btdesign`) like this:

```
DomainEvent ShipHasArrived {
   - ShipId ship
   - UnLocode port
}

DomainEvent ShipHasDepartured {
   - ShipId ship
   - UnLocode port
}
```

DomainEvents may contain attributes and references in the same way as ValueObjects and Entities. DomainEvents are always immutable and not persistent.

Events are about something happening at a point in time, so it's natural for events to contain time information. Sculptor automatically adds two timestamps, occurred and recorded. The time the event occurred in the world and the time the event was noticed.

Publish

The easiest way to publish a DomainEvent is to mark a service or repository operation in `model.btdesign` like this:

```
Service TrackingService {
   @ShipHasArrived recordArrival(DateTime occurred, @Ship ship, @Port port)
      publish to shippingChannel;
}
```

The operation must return a DomainEvent or take a DomainEvent as parameter. That event is published to the defined topic of the event bus when the operation has been invoked.

As an alternative the DomainEvent instance can be created from the return value or parameters.
The DomainEvent must have a matching constructor. In the below sample code the save operation of CargoRepository returns a Cargo. The save method is defined with a publish side effect that will create an instance of CargoSavedEvent holding the saved Cargo entity.

```
Entity Cargo {
   gap
   String cargoId key
   boolean hasBeenInCanada

   Repository CargoRepository {
      save publish to shippingChannel;
      findByKey;
   }
}

DomainEvent CargoSavedEvent {
   - Cargo domainObject
}
```

The result of the above declarative way of publishing events is a generated annotation @Publish on the method. It will trigger the Spring AOP advice PublishAdvice that is part of Sculptor framework.

It is of course also possible to publish events programatically:

```
    @Autowired
    @Qualifier("eventBus")
    private EventBus eventBus;

    public void recordArrival(DateTime occurred, Ship ship, Port port) {
        // process the arrival
        // ...
        // and then publish
        eventBus.publish("shippingChannel", new ShipHasArrived(occurred, ship.getShipId(),
port.getUnlocode()));
    }
```

Subscribe

Subscribers can be defined in `model.btdesign`. Any ordinary Service or Repository can become a subscriber of a topic like this:

```
        Service Statistics {
            subscribe to shippingChannel
            int getShipsInPort(@UnLocode port);
            reset;
        }
```

The result is that the Statistics service will implement the EventSubscriber interface and be marked with @Subscribe annotation. That means that the Statistics service will automatically be added as subscriber to chippingChannel of the event bus. It will be notified, `receive` method called, when events are published to that topic.

You need to manually implement the `receive` method of the EventSubscriber interface.

```
    @Override
    public void receive(Event event) {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("receive not implemented");
    }
```

Instead of writing ugly if-instanceof conditions you can add one method per event type and use the DynamicMethodDispatcher.

```
    @Override
    public void receive(Event event) {
        DynamicMethodDispatcher.dispatch(this, event, "consume");
    }

    public void consume(ShipHasArrived event) {
        UnLocode key = event.getPort();
        addShipInPort(key, 1);
    }

    public void consume(ShipHasDepartured event) {
        UnLocode key = event.getPort();
        addShipInPort(key, -1);
```

```
    }

    public void consume(Object any) {
        log.info("Ignored event: " + any);
    }
```

Not only Service and Repository can be subscribers. It is also possible to define a dedicated Consumer, which can only receive events. You can inject Services and Repositories to the Consumer, as usual.

```
Consumer EventCounter {
    inject @SomeService
    inject @SomeRepository
    subscribe to shippingChannel
}
```

It is of course also possible to subscribe with hand written code:

```
@Autowired
@Qualifier("eventBus")
private EventBus eventBus;

public void somewhere() {
    eventBus.subscribe("shippingChannel", new EventSubscriber() {
        @Override
        public void receive(Event event) {
            System.out.println("Received: " + event);
        }
    });
}
```

Switch Event Bus Implementation

The default event bus is a good start, but it is likely that you need to replace it with another implementation if you do real Event Driven Architecture. Sculptor provides two more implementations out of the box. One for Spring Integration and another for Apache Camel.

How to Use Spring Integration

Define the following property in `sculptor-generator.properties`:

```
integration.product=spring-integration
```

Add the following dependency in `pom.xml`

```
        <dependency>
            <groupId>org.springframework.integration</groupId>
            <artifactId>spring-integration-core</artifactId>
            <version>${spring-integration.version}</version>
        </dependency>
```

Add in properties section of `pom.xml`

```
        <spring-integration.version>2.0.0.RELEASE</spring-integration.version>
```

Re-generate!

Then you will have a new `spring-integration.xml` file in `src/main/resources`. This file is only generated once, and you can use it to add configuration for Spring Integration. Documentation is available at Spring Integration site.

There is also a `spring-integration-test.xml` in `src/test/resources` to make it possible to use different configuration when running JUnit tests.

A little sample of how to add a few topics and routes:

```
    <!-- publish-subscribe-channel corresponds to topic in event bus -->
    <publish-subscribe-channel id="shippingChannel" />
    <publish-subscribe-channel id="shippingStatistics" />
    <publish-subscribe-channel id="shippingStatistics2" />

    <!-- Invoke a method auditEvent on auditService when
         events are published to shippingChannel  -->
    <service-activator input-channel="shippingChannel"
         ref="auditService" method="auditEvent" />

    <!-- Route events from one topic to two other topics -->
    <recipient-list-router id="statisticsRouter" input-channel="shippingChannel">
         <recipient channel="shippingStatistics"/>
         <recipient channel="shippingStatistics2"/>
    </recipient-list-router>
```

How to use Apache Camel

Define the following property in `sculptor-generator.properties`:

```
integration.product=camel
```

Add the following dependencies in `pom.xml`

```
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-core</artifactId>
            <version>${camel.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-jms</artifactId>
            <version>${camel.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-spring</artifactId>
            <version>${camel.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.activemq</groupId>
```

```
            <artifactId>activemq-camel</artifactId>
            <version>5.3.2</version>
        </dependency>
        <!-- xbean is required for ActiveMQ broker configuration in the spring xml file -->
        <dependency>
            <groupId>org.apache.xbean</groupId>
            <artifactId>xbean-spring</artifactId>
            <version>3.7</version>
        </dependency>
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.1</version>
        </dependency>
```

Add in properties section of `pom.xml`

```
        <camel.version>2.6.0</camel.version>
```

Re-generate!

Then you will have a new `camel.xml` file in `src/main/resources`. This file is only generated once, and you can use it to add configuration for Camel. Documentation is available at Apache Camel site.

There is also a `camel-test.xml` in `src/test/resources` to make it possible to use different configuration when running JUnit tests.

Camel makes it possible to define endpoints and routes in either xml or Java DSL. It is a matter of taste. Here is a little sample of how to add a few endpoints and routes with the Java DSL:

```
public class Routes extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        // route from the shippingChannel queue to two different statistics endpoints,
        // via jms topic. Method invokation of auditService is also done on the way.

        from("direct:shippingChannel").log(LoggingLevel.DEBUG, "Processing: ${body}").to(
                "bean:auditService?method=auditEvent", "jms:topic:shippingEvent");
        from("jms:topic:shippingEvent").to("direct:shippingStatistics").to("direct:shippingStatistics2");
    }
}
```

How to use your own event bus

The interface of the event bus is simply three methods, `subscribe`, `unsubscribe`, and `publish`. You can easily write your own implementation. You can get inspiration by looking at the source code of `SimpleEventBusImpl`, `SpringIntegrationEventBusImpl`, or `CamelEventBusImpl`.

Thereafter you need to define that your bus is to be used. Define that in `src/main/resources/more.xml` like this:

```
    <bean id="myEventBusImpl" class="org.foo.MyEventBusImpl"/>
    <alias name="myEventBusImpl" alias="eventBus"/>
```

That will override the default event bus named "eventBus".

Command-Query Responsibility Segregation (CQRS)

- Where is my cargo now?
- Which route did my cargo take?
- What cargos have passed Vancouver during the last week?

... and dozens of more questions are of interest for different stakeholders.

It would be rather easy to develop support for those kind of queries in the master shipping domain, but problems will soon bubble up.

- Poor performance - the structure of the domain model is not optimized for all types of queries.
- Not scalable - single centralized database will become a bottleneck.
- Hard to change - too much functionality in one monolithic system.

Command-Query Responsibility Segregation (CQRS) comes to the rescue. Simplified it is about separating commands (that change the data) from the queries (that read the data). Separate subsystems take care of answering queries (reporting) and the domain for processing and storing updates can stay focused. The result of the commands are published to query subsystems, each optimized for its purpose.

There are a lot of interesting material to study related to CQRS. For example:

- Clarified CQRS
- Unshackle Your Domain

Let us look at a sample of a query subsystem that is responsible for answering "Where is my cargo now?" (and similar queries).

It should be defined in a separate subsystem with its own database. It might be defined like this in `model.btdesign`:

```
Module tracking {
   Service CargoTracker {
      locateCargo delegates to CargoInfoRepository.findByKey;
   }

   Entity CargoInfo {
      String cargoId key;
      - UnLocode location;
      DateTime locationTime;
      - ShipId loadedOnShip;
      boolean atSea;

      Repository CargoInfoRepository {
         subscribe to shippingChannel
         findByKey;
         protected List<@CargoInfo> cargosOnShip;
         save;
      }
   }
}
```

Note that it subscribes to the shippingChannel. When the master shipping subsystem has processed the commands it publishes DomainEvents to this topic. It can be JMS messages, maybe serialized with Protobuf, but that is technical details handled in the integration product and hidden from the business domain.

1. recordLoad publishes CargoLoaded, received by tracking subsystem, which updates loadedOnShip for corresponding CargoInfo
2. recordDeparture publishes ShipHasDepartured, received by tracking subsystem, which updates atSea for all CargoInfo loaded on that ship
3. recordArrival publishes ShipHasArrived, received by tracking subsystem, which updates atSea, location, locationTime for all CargoInfo loaded on that ship

4. recordUnload publishes CargoUnloaded, received by tracking subsystem, which updates loadedOnShip for corresponding CargoInfo

The actual public query, locateCargo in the CargoTracker Service, is a plain findByKey without any joins, i.e. super fast.

Does it strike you that a relational database might not be necessary, nor optimal, for all types of query stores? I think you should select the best tool for the job. Key-value store, document-oriented database, and RDBMS with SQL all have their strengths and weaknesses for different scenarios.

Sculptor has support for MongoDB.

As you can see the same publish/subscribe mechanisms as described before can be used for an architecture based on CQRS.

CommandEvent

Sculptor also supports CQRS by having possibility to define CommandEvent, which is something that the system is asked to perform, as opposed to DomainEvent, which is something that has happened. This means that you can make the commands explicit in the model, e.g.

```
CommandEvent RecordArrival {
   - ShipId ship
   - UnLocode port
}

CommandEvent RecordLoad {
   - ShipId ship
   String cargoId
}
```

There is a separate command event bus instance for processing CommandEvents. When subscribing to the command bus it must be specified:

```
Service ShippingCommandHandler {
   subscribe to handleShippingCommand eventBus=commandBus
```

Event Sourcing

A domain model typically holds current state of the world. Event Sourcing makes it possible to see how we got to this state and query how the state looked liked in the past. Essentially it means that we have to *capture all changes to an application state as a sequence of events*. These events can be used to reconstruct current and past states.

Good descriptions of Event Sourcing can be found here:

- Martin Fowler's description of Event Sourcing, see also Parallel Model
- CQRS and Event Sourcing
- Why use Event Sourcing?

Note that CQRS and EventSourcing are two different patterns. One can be used without the other, but they also play very well together.

DomainEvents are by default not persistent, but it is easy to make them persistent so that they can be stored and loaded in the same way as Entities and ValueObjects, i.e. with Repository

```
abstract DomainEvent ShipEvent {
        persistent
        - ShipId ship
        Long aggregateVersion nullable
        Long changeSequence nullable

        Repository ShipEventRepository {
                List<@ShipEvent> findAllForShip(@ShipId shipId);
                save publish to shippingChannel;
                protected findByCondition;
        }
}

DomainEvent ShipHasArrived extends @ShipEvent {
    - Port port
}
```

Using the ordinary building blocks of Sculptor it is rather straightforward to implement EventSourcing. It is described in this blog post how to do it. You might need to add snapshot mechanism also.

MongoDB is a good data store for events, as explained in the blog. The MongoDB Tutorial explains how to setup a project with MongoDB persistence. If you don't use MongoDB then you can use any other supported database (JPA), since it is using the ordinary persistence mechanisms of Sculptor. Note that you can have one subsystem responsible for the EventSourcing only and use MongoDB for that, and have another subsystem storing current state of the domain model using RDBMS, and another for queries.

Source

The complete source code for this tutorial is available in Subversion.

Web Access (read only):
http://fisheye3.cenqua.com/browse/fornax/trunk/cartridges/sculptor/mongodb-sample/sculptor-shipping

Anonymous Access (read only):
https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/mongodb-sample/sculptor-shipping/