Taberna

Implement a RESTful testing strategy

Vladimir Ritz Bossicard (vbossica@gmail.com)

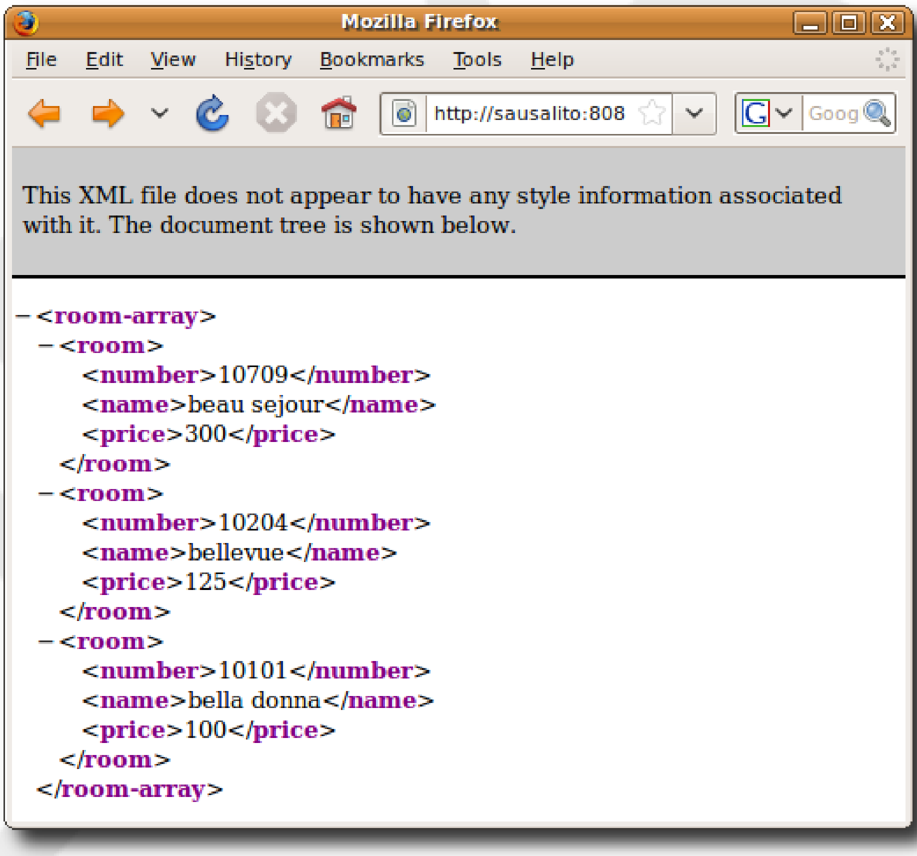
The REST interface is a very easy and powerful way to expose and interact with application data. Everyone is raving about this technology and promise that it will scale your regular Java EE enterprise application the same way it scaled the web.

The Spring framework has now a great support for REST and since the framework is the backbone of numerous application, it makes perfect sense to take a closer look. Instead of just verifying that exposing data via a REST interface is indeed trivial, let's see how this can improve the testing of the application.

Do you need some REST?

As base for the discussion, we'll consider a (simplistic) application that allows a customer to make a reservation for a room. The application is called Taberna[1] and exposes a few methods and objects via its REST interface[2]. Since this article is not about how to expose data via REST (there are plenty of documentation on the web) there's really not much to say than to show a nice screen shot:

Figure 1. Tabera REST invocation



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <room-array>
  - <room>
      <number>10709</number>
      <name>beau sejour</name>
      <price>300</price>
    </room>
  - <room>
      <number>10204</number>
      <name>bellevue</name>
      <price>125</price>
    </room>
  - <room>
      <number>10101</number>
      <name>bella donna</name>
      <price>100</price>
    </room>
  </room-array>
```

[1] taberna is the latin translation for “inn”
[2] taberna, inn, rest... got it? :-)

Because REST primarily focuses on directly accessing the business logic of an application (without going through the often convoluted GUI) it seems quite well suited for testing that particular layer. This is especially true if the application is mainly focused on processing data.

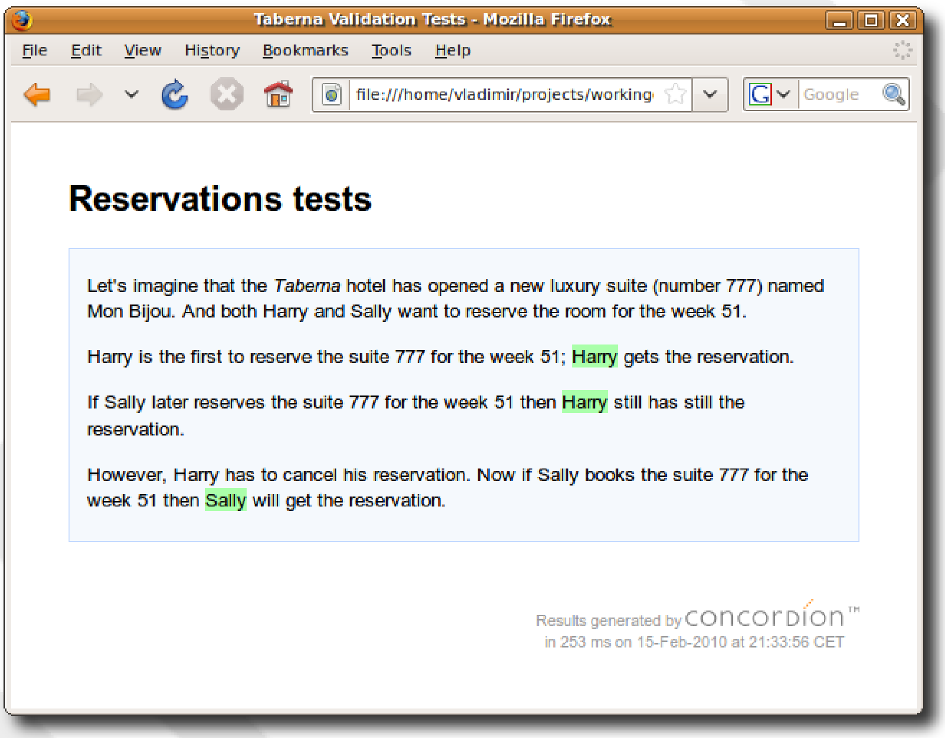
So why not using this interface to *test* our application business logic?

Testing, the RESTful way

Of course it would be complete nonsense to write unit tests or integration tests with REST but business tests could be well suited for this purpose. The idea is quite simple: using a framework like Fitness or Concordion, business tests access the application via its REST interface and verify that the deployed application behaves correctly. Of course, the application may have to be started in a *testing* mode (meaning that external systems are mocked) but this is not too difficult to do[3].

As an example, the following screen shot shows a Concordion test accessing Taberna via REST. I won't go into the details but the code is trivial.

Figure 2. Concordion tests


Taberna Validation Tests - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///home/vladimir/projects/working
Google

Reservations tests

Let's imagine that the *Taberna* hotel has opened a new luxury suite (number 777) named Mon Bijou. And both Harry and Sally want to reserve the room for the week 51.

Harry is the first to reserve the suite 777 for the week 51; Harry gets the reservation.

If Sally later reserves the suite 777 for the week 51 then Harry still has still the reservation.

However, Harry has to cancel his reservation. Now if Sally books the suite 777 for the week 51 then Sally will get the reservation.

Results generated by concordion™
in 253 ms on 15-Feb-2010 at 21:33:56 CET

To facilitate the tests, a custom REST client has also been developed. The fact that we are in control of both the server and the client implementations has the big advantage of keeping the test definitions quite stable. This can make a huge difference when comes the time to do some big refactoring...

Another advantage is that the tests can be shipped alongside the application and used to verify that an application has been correctly installed:

```
java -Durl=http://sausalito:8080/taberna -Doutdir=output
    org.testng.TestNG -testclass org.workingonit.taberna.CustomerTest
```

[3]another pet project, *Modulus* is experimenting with this very idea

Business Events

This is all nice and shiny with a trivial application but things starts to get a little bit messier when manual processes or third party applications are involved. The solution is to somehow streamline the processing so that interruptions in the process can be removed during testing.

One elegant solution is to leverage Spring's event model to dispatch business event at the join points between automatic and manual processes. In our example, imagine that mails must be sent after the reservation is booked:

```
public class BookingEngine implements ApplicationEventPublisherAware {

    private ApplicationEventPublisher publisher;

    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
    }

    public void book(Integer number, Reservation reservation) {
        // ...
        this.publisher.publishEvent(new ReservationBookedEvent(number, reservation));
    }

}
```

The real listener will of course send the email (maybe during the night), but for our testing purposes we will simply mock the behavior:

```
public class MockReservationPublisherListener
        implements ApplicationListener<ReservationBookedEvent> {

    public void onApplicationEvent(ReservationBookedEvent event) {
        Reservation reservation = reservations.get(event.getNumber(), event.getWeek());
        reservation.setStatus(PUBLISHED);
    }

}
```

The main advantage of this approach is that the processing is decoupled and the architecture is suddenly more concerned about the *what* (the reservation has been booked) than the *how* (an email has been sent).