14 DE ABRIL DE 2013

Java Configuration with Spring Batch

In **the first post in this series**, I introduced Spring's Java configuration mechanism. There is the base configuration style, and - as we saw - there are annotations (like `@EnableWebMvc`) that turn on container-wide, and often conventions-oriented features. In the case of `@EnableWebMvc`, the annotation enables Spring MVC and turns on support for correctly exposing `@Controller`-annotated beans as HTTP endpoints. Java configuration APIs of the same style as `@EnableWebMvc` can often optionally implement interfaces that are used to contribute, change, or otherwise tailor the API. Most Spring projects today offer beta-or-better APIs that work in this same manner. In this post, we'll look at **Spring Batch's Java Configuration API** available in the upcoming 2.2 release.

Spring Batch provides a solution for batch processing. It lets you describe jobs, which themselves have steps, which read input from some source system (a large RDBMS dataset, a large CSV file or XML document, etc.), optionally process the input, and then write the output to a destination system like an RDBMS, a file, or a message queue.

```
@Configuration
@EnableBatchProcessing
public class BatchInfrastructureConfiguration   {

    @Bean
    public TaskScheduler taskScheduler() {
        return new ConcurrentTaskScheduler();
    }

    @Bean
    public PlatformTransactionManager transactionManager(
DataSource ds) {
        return new DataSourceTransactionManager(ds);
    }


    @Bean
    public DataSource dataSource(Environment environment)
 {

        String pw = environment.getProperty("dataSource.p
assword"),
                user = environment.getProperty("dataSourc
e.user"),
                url = environment.getProperty("dataSource
.url");
```

Blog Feeds

- **general**
- **java**
- **python, ruby, scripting, etc.**

(no results)

Link

- **Java Configuration with Spring Batch**

```
        Class classOfDs = environment.getPropertyAsClass(
"dataSource.driverClassName", Driver.class);

        SimpleDriverDataSource dataSource = new SimpleDri
verDataSource();
        dataSource.setPassword(pw);
        dataSource.setUrl(url);
        dataSource.setUsername(user);
        dataSource.setDriverClass(classOfDs);

        return dataSource;
    }
}
```

Once you've done this, you can start describing jobs using the Spring Batch configuration DSL. Here, we define a job named `flickrImportJob` which in turn has one step, `step1`, that in turn reads data using an `ItemReader` named `photoAlbumItemReader` and writes data using an `ItemWriter` named `photoAlbumItemWriter`.

```
@Configuration
@Scope(proxyMode = ScopedProxyMode.TARGET_CLASS)
@Import(BatchInfrastructureConfiguration.class)
public class BatchImporterConfiguration {

    @Bean(name = "flickrImportJob")
    public Job flickrImportJob(
            JobBuilderFactory jobs,
            @Qualifier("step1") Step s1
    ) {
        return jobs.get("flickrImportJob")
                .flow(s1)
                .end()
                .build();
    }


    @Bean(name = "step1")
    public Step step1(StepBuilderFactory stepBuilderFacto
ry,
                      @Qualifier("photoAlbumItemReader")
ItemReader ir,
                      @Qualifier("photoAlbumItemWriter")
ItemWriter iw
    ) {
        return stepBuilderFactory.get("step1")
                .chunk(10)
                .reader(ir)
                .writer(iw)
                .build();
    }
    // ... omitting definitions of ItemReader and ItemWri
ters
}
```

Posted 14 de Abril de 2013 by Josh Long in **java**
Permalink