Enterprise Integration with Spring

Study Notes

These study notes were created by Lubos Krnac and are based on various Spring reference documentations.

Table of Contents

1 Tasks and Scheduling

1.1 Spring framework provides

~ uses `TaskExecutor` and `TaskScheduler` to abstract Java's asynchronous execution and scheduling
~ hides differences between Java SE 5, Java SE 6 and Java EE environments
~ features integration classes for scheduling

- `Timer` (part of JDK since 1.3)
- Quarz Scheduler

1.2 `TaskExecutor` abstraction

~ identical to `java.util.concurrent.Executor` interface

```
public interface TaskExecutor extends Executor {
      void execute(Runnable task);
}
```

~ created to give other Spring components an abstraction for thread pooling where needed
~ `TaskExecutor` implementations are used as simple JavaBeans within Spring context
~ out of the box implementations:

- `SimpleAsyncTaskExecutor`
  - doesn't use thread pool
  - support concurrency limit – will block further invocations until slot is freed up
- `SyncTaskExecutor`
  - doesn't execute invocation asynchronously (invocation takes place in calling thread)
- `ConcurrentTaskExecutor`
  - wrapper for a Java 5 `java.util.concurrent.Executor` interface
  - rarely used in comparison to `ThreadPoolTaskExecutor`
- `SimpleThreadPoolTaskExecutor`
  - subclass of Quartz's `SimpleThreadPool` which listens to Spring's lifecycle callbacks
- `ThreadPoolTaskExecutor`
  - wrapper for a Java 5 `java.util.concurrent.ThreadPoolExecutor`
- `TimerTaskExecutor`
  - uses a single `TimerTask`.
  - different from the `SyncTaskExecutor` - method invocations are executed in a separate thread, although they are synchronous in that thread
- `WorkManagerTaskExecutor`
  - convenience class for setting up a CommonJ `WorkManager` reference in a Spring context
  - implements the `WorkManager` interface and therefore can be used directly as a `WorkManager` as well.

1.3 `TaskScheduler` abstraction

~ variety of methods for scheduling tasks to run at some point in the future

```
public interface TaskScheduler {
      ScheduledFuture schedule(Runnable task, Trigger trigger);
      ScheduledFuture schedule(Runnable task, Date startTime);
```

```
    ScheduledFuture scheduleAtFixedRate(Runnable task, Date startTime, long period);
    ScheduledFuture scheduleAtFixedRate(Runnable task, long period);
    ScheduledFuture scheduleWithFixedDelay(Runnable task, Date startTime,long delay);
    ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);
}
```

  - fixed-rate – fixed start execution times
  - fixed-delay – fixed gap between executions

~ out of the box implementations:

- `TimerManagerTaskScheduler`
  - delegates to a CommonJ `TimerManager` instance, typically configured with a JNDI-lookup
- `ThreadPoolTaskScheduler`
  - can be used whenever external thread management is not a requirement
  - delegates to a `ScheduledExecutorServiceInstance`
  - implements Spring's `TaskExecutor` interface as well (can be used for immediate execution also)

1.4 `Trigger` interface

~ execution times may be determined based on past execution outcomes or even arbitrary conditions
~ single method – `Date nextExecutionTime(TriggerContext triggerContext)`
~ implementations:

- `CronTrigger` – scheduling based on Cron expressions
- `PeriodicTrigger` – accepts a fixed period, an optional initial delay value, and a `boolean` to indicate whether the period should be interpreted as a fixed-rate or a fixed-delay

~ `TriggerContext` interface is used to determine next execution time

1.5 Annotation Support

~ to enable annotation support:

- add `@EnableScheduling` and `@EnableAsync` to one of your `@Configuration` classes
- or use XML configuration:

```
<task:annotation-driven executor="myExecutor" scheduler="myScheduler"/>
<task:executor id="myExecutor" pool-size="5"/>
<task:scheduler id="myScheduler" pool-size="10"/>}
```

~ for more fine-grained control you can additionally implement the `SchedulingConfigurer` and/or `AsyncConfigurer` interfaces
~ `@Scheduled` annotation

- methods must have `void` returns
- methods must not expect any arguments
- annotation parameters
  - `fixedDelay`
  - `fixedRate`
  - `initialDelay`
  - `cron`

~ `@Async` annotation

- invocation of method will occur asynchronously
- methods can expect arguments, because they will be invoked in the "normal" way

by callers at run-time
- methods must be `void` or return `Future` type
- can't be used in conjunction with lifecycle callbacks (e.g. `@PostConstruct`)
- can have qualifier to override default configured task executor

```
@Async
Future<String> returnSomething(int i) {
	// this will be executed asynchronously
}
```

1.6 Task Namespace

~ `scheduler` element – create a `ThreadPoolTaskScheduler` instance with the specified thread pool size

```
<task:scheduler id="scheduler" pool-size="10"/>
```

~ `executor` element - create a `ThreadPoolTaskExecutor` instance

```
<task:executor id="executor" pool-size="10"/>
<task:executor id="executorWithPoolSizeRange" pool-size="5-25"
	queue-capacity="100"/>
```

~ `pool-size` – can have two value forms
- single value – specifies *core pool size*
  - number of threads to keep in the pool, even if they are idle
- range of values
  - first value – specifies *core pool size* (default is 1)
  - second value – specifies *maximum pool size* (default is `Integer.MAX_VALUE`)

~ `queue-capacity`
- capacity for the `ThreadPoolExecutor`'s `BlockingQueue`
- holds scheduled tasks when there isn't free thread in pool
- when capacity is reached, rejects to schedule new tasks
- default is `Integer.MAX_VALUE`
  - often not desirable, because `OutOfMemory` can occur

~ `scheduled-tasks` element - support for configuring tasks to be scheduled within a Spring Application Context

```
<task:scheduled-tasks scheduler="myScheduler">
	<task:scheduled ref="beanA" method="methodA" fixed-delay="5000"/>
	<task:scheduled ref="beanB" method="methodB" fixed-rate="5000"/>
	<task:scheduled ref="beanC" method="methodC" cron="*/5 * * * * MON-FRI"/>
</task:scheduled-tasks>
```

1.7 Application Servers

~ JEE apps shouldn't use threads directly
~ Spring can integrate with application servers via
- CommonJ
- JCA

~ Application server handles configuration

2 Spring Remoting

2.1 The concepts involved with Spring Remoting on both server- and client-side

~ Remoting – synchronous calls of remote methods

~ Spring Remoting goals
- Decouple from remoting specific code
- Declarative approach to configure and expose services
- Support various protocols

~ Server side: provides exporters to handle requests
- Binding to RMI registry or endpoint exposing

~ Client side: provides `FactoryBeans` that generate proxies
- Invoke methods on remote server
- Convert remote exceptions to runtime

2.2 The benefits of Spring Remoting over traditional remoting technologies

~ Configuration-based approach
- Server
  - Expose existing business services without code changes
- Client
  - Existing code doesn't have to be changed when invoking remote methods
  - Can use dependency injection

~ Exporters and proxy `FactoryBeans` provide consistent access via multiple protocols
- Server
  - Expose a single service over multiple protocols
- Client
  - Replace protocols easily
  - Switch between remote and local implementations

~ Hiding remoting infrastructure
- Server
  - No need to extend remoting interfaces (e.g. `Remote`)
- Client
  - `RemoteException` is translated into runtime

2.3 The remoting protocols supported by Spring

- Remote method invocation (RMI-IIOP)
- HTTPInvoker – Spring provides a special remoting strategy which allows for Java serialization via HTTP
- Hessian – lightweight binary HTTP-based protocol designed by Caucho
- Burlap – XML based alternative to Hessian
- JAX-RPC – replaced by JAX-WS from Java EE 5 / Java 6
- JMS

2.4 How Spring Remoting-based RMI is less invasive than plain RMI

~ Server side:
- exposing POJO services (via `RmiServiceExporter`)
- exposed service interfaces don't have to extend `java.rmi.Remote`
- the binding in the RMI registry is done automatically by Spring

~ Client side:
- Spring converts checked exceptions `java.rmi.RemoteException` into

unchecked (runtime) exceptions `RemoteAccessException`
- Spring provides factory (`RmiProxyFactoryBean`) dynamically generates the client-side proxy (no need to use traditional RMI stub).

~ Warning: classes exchanged must always implement the `Serializable` interface

2.5 How client and server interact with each other

~ Via remoting protocol
~ Server needs to expose service
- Must provide service implementation bean (*service* property below)
- Must provide service interface (*serviceInterface* property below)
- Must publish the service
  - RMI – using RMI registry

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
      <!-- does not necessarily have to be the same name as the bean to be
exported -->
      <property name="serviceName" value="AccountService" />
      <property name="service" ref="accountService" />
      <property name="serviceInterface" value="example.AccountService" />
      <!-- defaults to 1099 -->
      <property name="registryPort" value="1199" />
</bean>
```

  - HTTP invoker – various ways how to configure
    - Using servlet container + Spring MVC
      - register `DispatcherServlet` into servlet container
      - example of bean in specified in belonging context:

```
<bean name="/AccountService"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
      <property name="service" ref="accountService"/>
      <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

    - Using servlet container without Spring MVC support
      - register `HttpRequestHandlerServlet` into servlet container
      - example of bean in specified in belonging context:

```
<bean name="accountExporter"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
      <property name="service" ref="accountService"/>
      <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

    - Don't use servlet container at all
      - `SimpleHttpServerFactoryBean` + `SimpleHttpInvokerServiceExporter`

```
<bean name="accountExporter" class="org.springframework.remoting.
          httpinvoker.SimpleHttpInvokerServiceExporter">
      <property name="service" ref="accountService" />
      <property name="serviceInterface" value="example.AccountService" />
</bean>
<bean id="httpServer"
      class="org.springframework.remoting.support.SimpleHttpServerFactoryBean">
      <property name="contexts">
            <util:map>
                  <entry key="/remoting/AccountService" value-ref=
                        "accountExporter" />
            </util:map>
      </property>
```

```
      <property name="port" value="8080" />
</bean>
```
~ Client uses proxy to invoke service functionality (bean from example below can be used as if it was local bean)

- RMI example:

```
<bean id="accountService"
class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
      <property name="serviceUrl" value="rmi://HOST:1199/AccountService" />
      <property name="serviceInterface" value="example.AccountService" />
</bean>
```

- HTTP invoker example:

```
<bean id="accountService"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
      <property name="serviceInterface" value="example.AccountService"/>
      <property name="serviceUrl"
            value="http://HOST:8080//remoting/AccountService"/>
</bean>
```

3 Spring Web Services

3.1 How do Web Services compare to Remoting and Messaging

- Loose Coupling – we define document-oriented contract between service consumers and providers
- Interoperability – XML payload (is understood by all major platforms like Java. NET, C++, Ruby, PHP, Perl,...)

3.2 The approach to building web services that Spring-WS supports

~ Not using Contract Last approach where XSD/WSDL are generated from Java. Cons:

- XSD extensions – restrictions (e.g. regexp for string) can't be used because Java doesn't support it
- unportable types into XML
- cyclic graphs from Java are hard to represent in XML
- fragility – if contract (XSD and WSDL) is generated from Java (usually interface) can be generated differently -> contract can be changed more often
- performance – Java reference graph can easily become very big -> performance hit during conversion to XML
- versioning – change of contract is easier (e.g. by XSLT conversion from old version to new)

~ Spring WS uses Contract First approach only (start by writing the XSD/WSDL). Simple steps:

- create sample messages
- generate XSD (Trang, XML Spy)
- tweak resulting XSD to fit requirements
- Spring WS has ability to dynamically generate the WSDL from the XSD

3.3 The Object-to-XML frameworks supported by Spring-OXM

Note that Spring-OXM is now a module in Spring 3.0, not in Spring-WS, but for what you need to know that doesn't matter

- Low level techniques

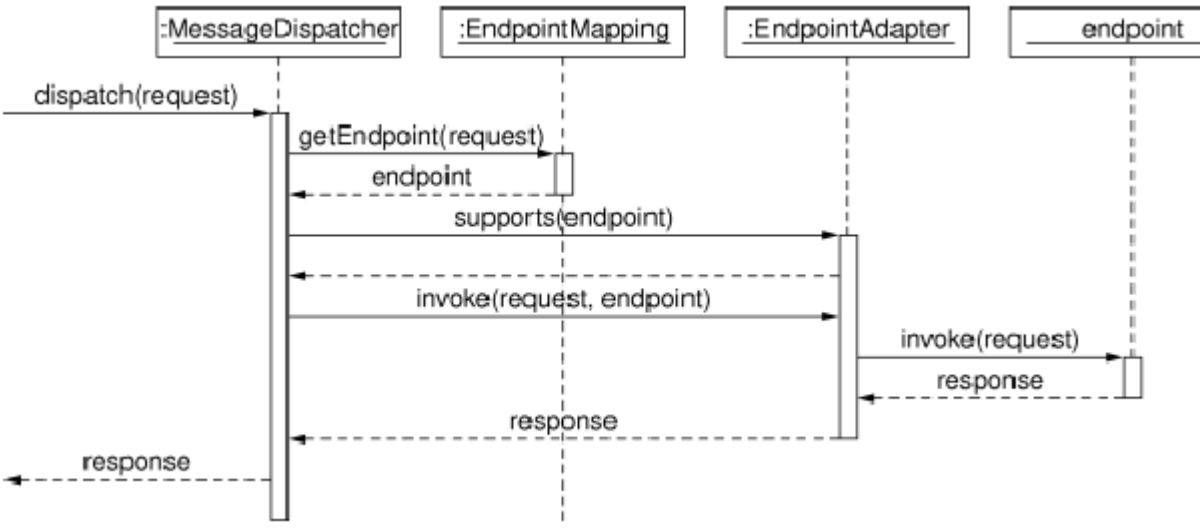
    - DOM family: JDOM, XOM, Dom4J, TrAX, W3C DOM
    - SAX
    - StAX
- Marshalling Object / XML (OXM)
    - JAXB 1 and 2 (standard Java)
    - Castor XML
    - XML Beans
    - XStream
    - JiBX
- Xpath argument binding

~ JAXB2 marshaller can be declared manually (as a part of Spring OXM):
<oxm:jaxb2-marshaller id="marshaller" contextPath="marshaller.package"/>
~ or Spring WS can register infrastructure beans for annotation driven (un)marshalling (including JABX2) by tag: <ws:annotation-driven/>

3.4 The strategies supported to map requests to endpoints

~ Endpoints (`@Endpoint`) provide access to the application behavior which is typically defined by a business service interface. An endpoint interprets the XML request message and uses that input to invoke a method on the business service (typically).
~ This list below describes the complete process a request goes through when handled by a `MessageDispatcher`:

1. An appropriate endpoint is searched for using the configured `EndpointMapping`s. If an endpoint is found, the invocation chain associated with the endpoint (pre-processors, post-processors, and endpoints) will be executed in order to create a response.
2. An appropriate adapter is searched for the endpoint. The `MessageDispatcher` delegates to this adapter to invoke the endpoint.
3. If a response is returned, it is sent on its way. If no response is returned (which could be due to a pre- or post-processor intercepting the request, for example, for security reasons), no response is sent.



~ Mapping techniques are based on:

- Message Payload (`@PayloadRoot`)
- SOAP Action Header (`@SoapAction`)
- WS-Addressing

- Xpath

~ Automatic publishing of generated WSDL

```
<ws:dynamic-wsdl id="transferDefinition" portTypeName="Transfers"
locationUri="http://somehost:8080/transferService/">
      <ws:xsd location="/WEB-INF/transfer.xsd"/>
</ws:dynamic-wsdl>
```

→ will expose WSDL to address:

`http://somehost:8080/transferService/transferDefinition.wsdl`

3.5 Of these strategies, how does `@PayloadRoot` work exactly?

~ `@Endpoint` (class annotation) – marks that class will be used as web services endpoint bean
~ `@PayloadRoot` (method annotation) - maps the root tag of the SOAP request body (the payload) on a bean method.
~ `localPart` (parameter of `@PayloadRoot` annotation) – specifies XSD/WSDL root element of the request payload
~ `namespace` (parameter of `@PayloadRoot` annotation) – specifies namespace URL
~ `@RequestPayload` – indicates that this method parameter should be mapped to the payload of request message
~ `@ResponsePayload` - indicates that the return value is used as the payload of the response message. If the return value is `void` (without annotation), no response is send

```
@PayloadRoot(localPart="helloRequest", namespace="http://myapp.com/hello")
public @ResponsePayload Hello sayHello (@RequestPayload Person onePerson){
```

3.6 The functionality offered by the `WebServiceTemplate`

~ Simplifies invoking web services
~ Works directly with XML payload
- of a SOAP message body
- or POX (Plain Old XML)

~ Supports marshalling/unmarshalling
~ Provides methods for sending and receiving messages
~ Mechanism callback methods (callback) calls for low level (eg access to SOAP headers)
~ `WebServiceTemplate` properties
- defaultUri
- marshaller
- unmarshaller

~ Expandable by adding interceptors (e.g. for validating)
~ Exception handling provided by the `SoapFautMessageResolver` that wraps errors in a `SoapFaultClientException`. Opportunity to provide their own resolver.
~ Allows multiple protocols: HTTP, Mail, JMS, XMPP
~ Example:

```
Hello response = (Hello) webServiceTemplate.marshalSendAndReceive(person);
```

3.7 The underlying WS-Security implementations supported by Spring-WS

~ Securing web services in terms of signature authentication and encryption is implemented using interceptors.
- `XwsSecurityInterceptor` based package Sun XML and Web Services Security

(XWSS) Sun. Prerequisites: Sun JDK / Oracle and the reference implementation of Sun SAAJ. Requires a security policy file to operate.

- `Wss4jSecurityInterceptor` for integrating Apache WSS4J implements standards:
  - SOAP Message Security 1.0 (OASIS)
  - Username Token Profile 1.0
  - X.509 Token Profile 1.0

3.8 How key stores are supported by Spring-WS for use with WS-Security

~ Most cryptographic operations requires a standard `java.security.KeyStore`. The keystore stores three types of elements:

1. Private Key: Used by WS-Security to sign and decrypt
2. Symmetric key (or secret key): client and server store the same key. The latter is used to both encrypt and decrypt.
3. Trust certificates (X509). WS-Security uses to validate certifications, verify the signature and encryption.

~ `KeyStoreFactoryBean` can be used to easily load keystores using Spring configuration. It has two properties:

- `location` to the keystore (eg classphath: truststore.jks or keystore.jks)
- `password` for the keystore.

~ When is security XWSS based, `KeyStoreCallbackHandler` is needed to handle various cryptographic callbacks. It has three parameters (exact stores used by the handler depend on the cryptographic operations that are to be performed by this handler):

- `keyStore`
- `trustStore`
- `symmetricStore`

~ To manage certificates, WSS4J uses a keystore file that is referenced by the `CryptoFactoryBean` class.

3.9 Additional chapters

3.9.1 Best practises for Spring Web Services

- Contract-first approach
- Don't use stubs and skeletons to promote separate evolution of contract and code
- Skipping validation (not necessarily good practise)

3.9.2 Error Handling

~ `EndpointExceptionResolver` interface converts exceptions thrown from endpoint methods to SOAP messages (replaces regular response messages with SOAP faults)

```
public interface EndpointExceptionResolver {
      boolean resolveException(MessageContext messageContext,
      Object endpoint, Exception ex);
}
```

~ `SimpleSoapExceptionResolver` – default implementation, that creates SOAP 1.1 Server or SOAP1.2 Receiver fault and uses the exception message as the fault string

~ `SoapFaultMappingExceptionResolver` – take the class name of any exception that might be thrown and map it to a SOAP Fault

```
<bean class="org.springframework.ws.
```

```
        soap.server.endpoint.SoapFaultMappingExceptionResolver">
        <property name="exceptionMappings">
            <value>
            java.lang.NumberFormatException=CLIENT, Invalid format
            </value>
        </property>
</bean>
```

~ `SoapFaultAnnotationExceptionResolver` – can annotate exception with `@SoapFault`, to indicate the SOAP Fault that should be returned whenever that exception is thrown (exception message -> SOAP fault string)

```
@SoapFault(faultCode = FaultCode.SERVER)
public class MyBusinessException extends Exception {
    public MyClientException(String message) {
        super(message);
    }
}
```

~ `SoapFaultMessageResolver`

- used on client side by `WebServiceTemplate`
- default implementation for fault response handling
- `WebServiceTemplate` throws `SoapFaultClientException` exception
  - wraps received SOAP error message and coverts into `SoapFault`
- can be replaced by custom implementation via property `faultMessageResolver` in `WebServiceTemplate` declaration

3.9.3 WS testing

~ Spring WS provides out of container integration testing with various expectations support

~ Server side

1. Create a `MockWebServiceClient` instance by calling its factory methods
2. Send request messages by calling `sendRequest` method (possibly use `RequestCreator` callback)
3. Set up response expectations by calling `andExpect` (possibly use `ResponseMatcher` callback)

```
mockClient = MockWebServiceClient.createClient(applicationContext);

Source requestPayload = new StringSource("...");
Source responsePayload = new StringSource("...");
mockClient.sendRequest(RequestCreators.withPayload(requestPayload))
    .andExpect(ResponseMatchers.payload(responsePayload));
```

~ Client side

1. Create a `MockWebServiceServer` instance by calling its factory methods
2. Set up request expectations by calling `expect` method (possibly use `RequestMatcher` callback)
3. Create an appropriate response message by calling `andRespond` (possibly use `ResponseCreator` callback)
4. Use the `WebServiceTemplate`
5. Call `verify` to check expectations

3.9.4 Interceptors

~ callbacks during message handling flow

~ `EndpointInterceptor` interface methods:

- `handleRequest` – before endpoint is called
- `handleResponse` – after endpoint was process without fault
- `handleFault` – after endpoint was processed with fault

~ with `false` return value can stop processing of invocation chain
~ methods can amend `MessageContext` request or response
~ Spring WS provides built-in implementations for
- logging (`SoapEnvelopeLoggingInterceptor`, `PayloadLoggingInterceptor`)
- payload validation (`PayloadValidatingInterceptor`)
- XSLT transformation (`PayloadTransformingInterceptor`)
- WS-Addressing support (`AddressingEndpointInterceptor`)
- security (`XwsSecurityInterceptor`, `Wss4jSecurityInterceptor`)

~ also provides support for client side interceptors (built-in interceptors for validation and security)

```
<ws:interceptors>
      <bean class="org.springframework.ws.server.
            endpoint.interceptor.PayloadLoggingInterceptor"/>
</ws:interceptors>
```

4 RESTful services with Spring-MVC

4.1 The main REST principles

~ Architectural style based on HTTP
~ HTTP is used as the application protocol (not as a transport layer in SOAP)
~ Five key concepts:

1. Identification of resources
   - everything is resource, e.g. a business entity
   - resources are represented as URIs
2. Uniform interface – contains operations to access resources
   - many resources (nouns)
   - few operations (verbs)
     - GET
       1. allows read-only access to the representation of a resource
       2. safe operation – no side effects
       3. cacheable (headers E-Tag or Last-Modified => code 304 Not Modified)
     - HEAD
       1. similar to a GET
       2. without body
       3. used when saving bandwidth
     - POST
       1. creates new resource
       2. `Location` header is send back to indicate URI of created resource
       3. non-idempotent operation.
     - PUT
       1. updates or create resource identified by a URI
       2. idempotent
       3. not safe (has side effects)
     - DELETE
       1. deletes a resource

        2. idempotent
        3. not safe
    3. Resource representations
        - resources are abstracted from its representations
        - resource can have various representations (text / html, image / png)
        - request uses header `Accept` to specify desired representation
        - response uses header `Content-Type` to indicate representation
    4. Stateless conversation
        - no client content is being stored on server
        - client can maintain state (via HTTP links)
        - scalable architecture – any server instance can serve the request
        - loose coupling – no shared session
        - enables easy scaling of server in production environment
    5. Hypermedia
        - resources contain hypermedia links
        - clients make state transitions only through actions that are dynamically identified by these links
        - no need to update client when server is changed
        - client has to conform some server semantics

~ Idempotency (not just for RESTful applications)
- multiple identical requests should have the same effect as a single request
- helps with integration (client can retry request without side effects)

~ Advantages of REST
- Scalability of component interactions
- Protocol support
    - Languages
    - Scripts
    - Browsers – only GET and POST through HTML
- Redirect
- Caching
- Different representations
- Simple Organizing resources
- Pluggable formats (e.g. XML, JSON, Atom,...)
- Simple load balancing
- Decouples client from server

~ Security
- possible with HTTP Basic or Digest
- every request must have authentication (REST is stateless!!!)
- transport level security – requires SSL
- message level security – use of XML-DSIG and XML-Encryption

~ HTTP (and therefore also REST) is not suitable for long-running transactions (use compensating transactions instead)

4.2 Spring MVC is an alternative to JAX-RS, not an implementation

~ Two options how to use Spring for REST (both are valid)
1. JAX-RS
2. Spring MVC 3.0 with REST

~ JAX-RS is a standard: Java API for RESTful Web Services (JSR-311)

- focuses mostly on app-2-app communication
- Jersey and CXF are implementations
- JAX-RS annotations: `@Path`, `@GET`, `@POST`, `@Produces`, `@PathParam`

```
import javax.ws.rs.*;
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Scope;

@Path("/customer/{id}")
@Component
@Scope("request")
public class CustomerService {
   @GET
   @Produces({"application/json"})
   public Customer getCustomer(@PathParam("id") String customerId) {
        return ...
   }
}
```

~ Spring MVC with REST (since Spring version 3.0):

- URI templates
- Content negotiation
- Declarative response status codes (no View is used)
- Client side with `RestTemplate`
- Message converters
- Easier for developers familiar with Spring MVC
- Supports browsers and REST clients

~ Example:

```
@Controller
@RequestMapping("/pets/{petId}")
public class PetController {
      @RequestMapping(method=RequestMethod.GET)
      public @ResponseBody Pet getPet(@PathVariable String petId) {
            return ...
      }
}
```

4.3 The `@RequestMapping` annotation, including URI template support

~ `@RequestMapping` annotation

- specifies how will be request mapped into controller methods
- can be used at
  - class level – typically specifies base URI (or URI pattern) for all methods handling the request
  - method level – narrowing the primary mapping for a specific HTTP method request method (GET, POST, etc.) or an HTTP request parameter condition
- parameters
  - `consumes` - consumable media types of the mapped request, narrowing the primary mapping
  - `headers` - headers of the mapped request, narrowing the primary mapping
  - `method` - HTTP request methods to map to, narrowing the primary mapping: GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE
  - `params` - parameters of the mapped request, narrowing the primary mapping
  - `produces` - producible media types of the mapped request, narrowing the

primary mapping
    - `value` - primary mapping expressed by this annotation

~ URI template patterns

- convenient access to selected parts of a URL in a `@RequestMapping` method

```
@RequestMapping(value="/pets/{petId}", method=RequestMethod.GET)
public @ResponseBody Pet getPet(@PathVariable String petId) {
      return ... }
```

- `@PathVariable` parameter is not necessary if the parameter name is the same as pattern
- when a `@PathVariable` annotation is used on a `Map<String, String>` argument, the map is populated with all URI template variables
- regular expressions can be used

```
@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\\d\\.\\d\\.\\d}
{extension:\\.[a-z]+}")
      public void handle(@PathVariable String version, @PathVariable String
            extension) {
            // ...
      }
}
```

~ Other tweaks with URI

- Path patterns
- Patterns with placeholders
- Matrix variables

~ Declarative response status codes (no View)

```
@RequestMapping(value="/pets", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public void createPet(HttpServletRequest req, HttpServletResponse resp) {
      ... }
```

~ Business exceptions can be annotated with `@ResponseStatus`

4.4 The `@RequestBody` and `@ResponseBody` annotations

~ `@ResponseBody`

- No View is used
- Return instance is converted into HTTP response body
- Accept header specifies converter / marshaller
    - XML via JAXB2, JSON via Jackson,...

```
@RequestMapping(value="/pets/{petId}", method=RequestMethod.GET)
public @ResponseBody Pet getPet(@PathVariable String petId) {
      return ... }
```

~ `@RequestBody`

- Used with POST, PUT
- HTTP request body is converted to method parameter
- `Content-Type` of request specifies converter

```
@RequestMapping(value="/pets/{petId}", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updatePet(@RequestBody Pet updatePet,
      @PathVariable("petId") long id) {
      // process updated order data and return empty response
      ...
}
```

4.5 The functionality offered by the `RestTemplate`

- Provides client access to RESTful services
- Also supports URI templates
- HTTP message conversion
- Manipulating HTTP headers and content via `HttpEntity` (since version 3.0.2)

HTTP method	RestTemplate methods
DELETE	**delete**(String url, String... urlVariables)
GET	**getForObject**(String url, Class<T> responseType, String... urlVariables) **getForEntity**(String url, Class<T> responseType, String... urlVariables)
HEAD	**headForHeaders**(String url, String... urlVariables)
OPTIONS	**optionsForAllow**(String url, String... urlVariables)
POST	**postForLocation**(String url, Object request, String... urlVariables) **postForObject**(String url, Object request, Class<T> responseType, String... uriVariables) **postForEntity**(String url, Object request, Class<T> responseType, String... uriVariables)
PUT	**put**(String url, Object request, String...urlVariables)

~ `RestTemplate` usage

1. Creation via constructor

```
RestTemplate template = new RestTemplate();
```

   - Various out of the box HTTP message converters (like on server)

2. XML configuration

Client side usage examples:

```
RestTemplate template = new RestTemplate();
String uri = "http://petshop.com/pets/categories/{category}";
// GET all pets from category dogs:
Pet[] pets = template.getForObject(uri, Pet[].class, "dogs");
// POST to create pet
URI itemLocation = template.postForLocation(uri, pet, "dogs");
// PUT to update the pet
pet.setName("Rexo");
template.put(itemLocation, pet);
// DELETE to remove the pet
template.delete(itemLocation);

HttpEntity<String> request = new HttpEntity<String>("Do you have a dog?",
MediaType.TEXT_PLAIN);
URI location = template.postForLocation("http://petshop.com/", request);
// wait for response reception
ResponseEntity<String> response =
template.getForEntity("http://petshop.com", String.class);
HttpStatus status = response.getStatusCode();
MediaType contentType = response.getHeaders().getContentType();
String body = response.getBody();
```

5 Spring JMS

5.1 Where can Spring-JMS applications obtain their JMS resources from

~ Spring JMS is designed to decouple application from JMS infrastructure
- Spring instantiates JMS resources
- Application uses Spring API (e.g. `JmsTemplate`)

~ Spring JMS also provides various deployment choices
- Standalone JMS provider
  - `ConnectionFactory` implementation

```
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:8082"/>
</bean>
```

  - Destination implementation (`Queue` or `Topic`)

```
<bean id="defaultQueue" class="org.apache.activemq.command.ActiveMQQueue">
      <constructor-arg value="queue.name"/>
</bean>
```

- Pick up JMS implementation from JEE application server via JNDI
  - `ConnectionFactory` implementation

```
<jee:jndi-lookup id="connectionFactory" jndi-name="jms/ConnectionFactory"/>
```

  - Destination implementation (`Queue` or `Topic`)

```
<jee:jndi-lookup id="defaultQueue" jndi-name="jms/DefaultQueue"/>
```

~ `ConnectionFactory`, `Queue`, `Topic` interfaces are part of JMS specification (JMS resources)

5.2 The functionality offered by the `JmsTemplate`

~ `JmsTemplate` is central class of Spring JMS and handles creation and release of JMS resources during
- message production
- synchronous message consumption

~ For asynchronous message consumption is used `MessageListenerContainer`

~ Advantages of `JmsTemplate`
- Reduces amount of JMS specific code
- Handles creation and release of JMS resources transparently
- Has better mechanisms to handle exceptions
- No need to catch JMS checked exceptions – converts into runtime
- Provides handier methods and `JdbcTemplate` like callbacks

~ `JmsTemplate` delegates some responsibilities to instances of following interfaces
- `DestinationResolver` – resolution of a destination name to a JMS destination object
- `MessageConverter` – message conversion process

~ `MessageConverter` interface – defines a simple contract to convert between Java objects and JMS messages
- `SimpleMessageConverter`
  - default implementation
  - handles conversion between
    - `String` and `TextMessage`

        - `byte[]` and `BytesMesssage`
        - `java.util.Map` and `MapMessage`
        - `Serializable` and `ObjectMessage`
    - `MapMessageConverter` – uses reflection to convert between a Java bean and a `MapMessage`
    - Custom implementation is suitable sometimes for
        - XML marshalling into `TextMessage` – JAXB, Castor, XMLBeans, XStream
        - Steps for custom implementation:
            - Implement

```
public interface MessageConverter {
      Message toMessage(Object object, Session session) throws JMSException,
                  MessageConversionException;
      Object fromMessage(Message message) throws JMSException,
                  MessageConversionException;
}
```

            - Pass the implementation into `JmsTemplate`

~ `DestinationResolver` interface

- Only one method

```
public interface DestinationResolver {
      Destination resolveDestinationName(Session session, String
            destinationName, boolean pubSubDomain) throws JMSException;
}
```

- `JndiDestinationResolver` – service locator for destinations contained in JNDI
- `DynamicDestinationResolver`
    - default implementation
    - accommodates resolving dynamic destinations
    - may then also create a physical destination

~ Creation of `JmsTemplate`

- mandatory parameter `connectionFactory` to obtain JMS Connections from
- main optional parameters:
    - `messageConverter` - resolve `Object` parameters to `convertAndSend` methods and `Object` results from `receiveAndConvert` methods
    - `destinationResolver` – resolving destination names from simple `String`s to actual `Destination` implementation instances
    - `defaultDestination` – destination to be used on send/receive operations that do not have a destination parameter

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
      <property name="connectionFactory" ref="connectionFactory" />
      <property name="defaultDestination" ref="defaultDestination" />
</bean>
```

~ JMS resource caching

`JmsTemplate` assumes that JMS resources are being cached by JMS provider – closes and reopens them. But this is not effective without caching. So there is possibility to wrap `ConnectionFacotry` implementation into `CachingConnectionFactory` which is provided by Spring JMS and does caching of JMS resources.

~ Sending messages – `JmsTemplate` contains many convenience methods

- those that specify destination using `javax.jms.Destination`
- those that specify destination using spring for use in JNDI
- those that do not specify destination – use template's default destination
- those that leverage template's message converter
- callbacks that reveal JMS resources for more control

- those that specify `MessagePostProcessor`

Some examples:

```
public void convertAndSend(Destination destination, Object message)
public void send(MessageCreator messageCreator)
public <T> T execute(Destination destination, ProducerCallback<T> action)
public <T> T execute(SessionCallback<T> action, boolean startConnection)
```

~ Synchronous receiving messages

- similar convenience methods combinations as for sending
- methods with blocking of caller thread

```
public Message receive()
public Message receive(Destination destination)
public Message receive(String destinationName)
```

- leverage of `MessageConverter`

```
public Object receiveAndConvert(Destination destination)
```

5.3 The functionality offered by Spring's JMS message listener container, including the use of a `MessageListenerAdapter` through the '`method`' attribute in the `<jms:listener/>` element

~ Message listener container

- Used for asynchronous message reception from JMS message queue and drive `MessageListener` that is injected into it
- Dispatches messages into `MessageListener`
- Usually this mechanism is used by EJB container (Message driven POJO)
- Provides support for scheduling and endpoint management
- Provides two message listener container implementations
  - `SimpleMessageListenerContainer`
    - creates a fixed number of JMS sessions and consumers at startup
    - registers the listener using the standard JMS `MessageConsumer.setMessageListener()` method
    - leaves it up the JMS provider to perform listener callbacks
    - does not allow for dynamic adaption to runtime demands
    - does not allow participation in externally managed transactions
    - not compatible with Java EE's JMS restrictions
  - `DefaultMessageListenerContainer`
    - does allow for dynamic adaption to runtime demands
    - is able to participate in externally managed transactions
    - each received message is registered with an XA transaction when configured with a `JtaTransactionManager`

~ Listener bean

- can implement
  - `MessageListener` – JMS interface with one method that should be implemented by application

```
void onMessage(Message message)
```

  - `SessionAwareMessageListener` – Spring JMS listener interface that allows tweaking with JMS `Session`
- or can be used any bean directly via `MessageListenerAdapter`

```
<jms:listener destination="queue.orders" ref="orderService" method="placeOrder"
response-destination="queue.confirmations" />
```

~ Definition of `MessageListenerContainer`

```
<jms:listener-container connection-factory="connectionFactory">
    <jms:listener destination="queue.orders" ref="orderService"
      method="placeOrder"/>
    <jms:listener destination="queue.confirmations" ref="confirmationListener"/>
</jms:listener-container>
```

- Allows configuration of
  - task execution strategy
  - concurrency
  - container type
  - transaction manager
  - …

```
<jms:listener-container connection-factory="myConnectionFactory"
                                    task-executor="myTaskExecutor"
                                    destination-resolver="myDestinationResolver"
                                    transaction-manager="myTransactionManager"
                                    concurrency="10">
      <jms:listener destination="queue.orders" ref="orderService"
            method="placeOrder" />
      <jms:listener destination="queue.confirmations" ref="confirmationLogger"
            method="log" />
</jms:listener-container>
```

6 Local JMS Transactions with Spring

6.1 How to enable local JMS transactions with Spring's message listener container

1. When message listener container parameter `acknowledge` is set to value `transacted`

```
<jms:listener-container acknowledge="transacted">
      ...
</jms:listener-container>
```

   - Other possible values:
     - `auto`
       - default value
       - no transaction
       - acknowledge immediately after successful reception
       - slower
       - message can be lost
     - `client`
       - no transaction
       - client must acknowledge reception explicitly
       - dupes can occur if used properly (acknowledge after DB TX commit)
     - `dupes-ok`
       - no transaction
       - acknowledging is handled by JMS provider and dupes can occur
       - faster then `auto`
2. When transaction manager is used by message listener container
   - Usually for JTA

~ Transaction is created after reception when no JTA TX is in progress

6.2 If and if so, how is a local JMS transaction made available to the JmsTemplate

~ `JmsTemplate` parameters for `Session` can be specified
- `sessionTransacted`
- `sessionAcknowledgeMode` – will be ignored if `sessionTransacted` is specified

~ Same `Session` instance is used for receiving and sending

6.3 How does Spring attempt to synchronize a local JMS transaction and a local database transaction

~ User can use 'best effort' strategies
- Commit database before JMS commit on message reception
  - Messages wouldn't be lost
  - But can duplicate messages if error occurs during JMS commit
- Invoke commits close to each other to reduce failure risk (best approach is to invoke JMS commit straight after DB commit)

~ Only using of XA and JTA distributed transactions can ensure transactions synchronizing
~ Handling duplicates
- No problem if processing is idempotent
- If not, duplicity check is needed
  - At first, check if current reception is redelivery (this is crucial because avoids DB calls in most cases)

`message.getJMSRedelivered()`
  - No → process message
  - Yes → check if message was already processed (most probably check the DB)

6.4 The functionality offered by the JmsTransactionManager

~ `JmsTransactionManager`
- Performs local resource transactions, binding a JMS `Connection/Session` pair from the specified `ConnectionFactory` to the thread
- The `JmsTemplate` auto-detects an attached thread and participates automatically with `Session`
- The `JmsTransationManager` allows a `CachingConnectionFactory` that uses a single connection for all JMS access (performance gains). All Sessions belong to the same connection

7 JTA and Two-phased commit transactions with Spring

7.1 What guarantees does JTA provide that local transactions do not provide

~ XA (X/Open XA) is specification for distributed transaction processing
~ JTA (Java transaction API) is Java implementation that enables handling of transactions across multiple XA resources
~ More than JTA, it is the use of XA which:
- Guarantee ACID distributed / global transactions
- Coordinates commits of several transactional resources

- Avoids duplicate messages – messages are delivered once and only once.

7.2 How to switch from local to global JTA transactions

~ No code change needed, switch can be done via configuration
~ Just replace local implementation of transaction manager (`PlatformTransactionManager`) with `JtaTransactionManager` (or some of its application server vendor specific children that Spring provides)
~ `JtaTransactionManager` doesn't provide JTA support, it only integrates local transactions with external (application server) JTA transaction manager
~ If application server specific subclass is used, it allows usage of features that are not in JTA specification (e.g. transaction suspension)
~ Instance can be created via tx namespace in Spring's XML configuration

```
<tx:jta-transaction-manager/>
```

- Than for DB or synchronous JMS access can be used via `@Transactional` with `<tx:annotation-driven/>` or `<tx:advice/>`
- For asynchronous JMS reception can be used as parameter for JMS listener container:

```
<jms:listener-container transaction-manager="transactionManager">
    <jms:listener destination="queue.orders" ref="orderService"
      method="placeOrder"/>
    <jms:listener destination="queue.confirmations" ref="confirmationListener"/>
</jms:listener-container>
```

~ Third party frameworks like Hibernate must be configured specifically for JTA
~ JTA is requirement for EJB transaction handling (even with single transaction resource)
~ It is optional when Spring manages transactions (make sense only for more than one transaction resources)
~ Switching from local to global transaction handling is very easy with Spring – only little configuration changes

7.3 Where can you obtain a JTA transaction manager from

~ User can use Spring JTA support in two ways

1. Integrate with JEE application server
2. Stand alone usage of JTA

~ If you use JTA in a Java EE container then you use a container `DataSource`, obtained through JNDI, in conjunction with Spring's `JtaTransactionManager`

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>
<bean id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

- `JtaTransactionManager` does not need to know about the `DataSource`, or any other specific resources, because it uses the container's global transaction management infrastructure
- `JtaTransactionManager` class can optionally perform a JNDI lookup for the JTA
- `UserTransaction` and `TransactionManager` objects and autodetect the location for the latter object, which varies by application server (allows for enhanced transaction semantics, in particular supporting transaction suspension)
- Each XA transactional resource (`dataSource` for DB access, `connectionFactory` for JMS access) can be retrieved by a `<jee:jndi-lookup … />`

~ For stand-alone usage

- user needs to manually define a bean `transactionManager` bean and specify its

two properties `transactionManager` and `userTransaction` applications using JTA implementation (e.g. Atomikos, JOTM, Jboss Transactions (former Arjuna))

- When the bean is named `transactionManager`, Spring will automatically pick it up

```
<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager">
      <property name="transactionManager" ref="atomikosTransactionManager" />
      <property name="userTransaction" ref="atomikosUserTransaction" />
</bean>
```

7.4 Additional topics

7.4.1 Declarative transaction demarcation

- Little bit off the topic but crucial
- Recommended and non-invasive transaction handling in Spring applications
- Replaces explicit transaction demarcation API with an AOP transaction interceptor
- This interceptor can be configured via XML or Java annotations (preferred)
- Allows you to keep business services free of repetitive transaction demarcation code and to focus on adding business logic
- Two ways how to configure it
  - `@Transactional` annotation with `<tx:annotation-driven/>`

```
@Transactional(readOnly = true)
public List<Product> findAllProducts() {
      return this.productDao.findAllProducts();
}
```

  - XML configuration via `<tx:advice>`

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
      <property name="sessionFactory" ref="sessionFactory" />
</bean>
<aop:config>
      <aop:pointcut id="productServiceMethods"
            expression="execution(* product.ProductService.*(..))" />
      <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods" />
</aop:config>
<tx:advice id="txAdvice" transaction-manager="myTxManager">
      <tx:attributes>
            <tx:method name="increasePrice*" propagation="REQUIRED" />
            <tx:method name="someBusinessMethod" propagation="REQUIRES_NEW" />
            <tx:method name="*" propagation="SUPPORTS" read-only="true" />
      </tx:attributes>
</tx:advice>
```

  - Asynchronous JMS reception transaction demarcation
    - Local transactions

```
<jms:listener-container acknowledge="transacted">
      <jms:listener ref="jmsListener" destination="jms.queue"/>
</jms:listener-container>
```

    - Global transactions
      - use `transaction-manager="transactionManager"` instead of `acknowledge`

8 Spring Integration

8.1 Main concepts (Messages, Channels, Endpoint types)

Pay special attention to the various Endpoint types and how they're used!

~ Spring integration is implementation of Enterprise Integration Patterns

~ Principles

- Components should be loosely coupled for modularity and testability
- The framework should enforce separation of concerns between business logic and integration logic
- Extension points should be abstract in nature but within well-defined boundaries to promote reuse and portability

~ Spring Integration uses declarative adapters to

- Connect applications with external systems via various protocols
- Lightweight messaging within Spring-based applications
- Decouple application components from integration infrastructure
  - Converts external events into internal messages
  - Application components process only messages (typically only payload)
  - Can convert Internal message into external event

~ Main Components

- `Message`
- `MessageChannel`
- `MessageEndpoint`

8.1.1 `Message`

```
public interface Message<T> {
      MessageHeaders getHeaders();
      T getPayload();
}
```

- Is sent or received by `MessageEndpoint`
- `Message` parts
  - Headers (key/value pairs)
    - user defined
      - typically used to store metadata
    - pre-defined
      - `ID` (`java.util.UUID`) – unique identifier
      - `CORRELATION_ID` (`java.util.Object`)
      - `REPLY_CHANNEL` (`String` or `MessageChannel`)
      - ...
  - Payload (Java object)
- Immutable
- Created by
  - automatically by Spring Integration framework
  - or by `MessageBuilder` helper class

8.1.2 `MessageEndpoint`

- Receives/sends `Message`s from/into `MessageChannel`
- Primary role is to connect application code to the messaging framework in a non-

invasive manner
- `MessageEndpoint`s are mapped to `MessageChannel`s to isolate application code from the infrastructure
- Annotation support
  - `@MessageEndpoint` class level annotation indicates that class is used as message endpoint
  - Method level annotation specifies type of the message endpoint
  - Method parameters can be annotated with `@Header("headerParameter")` annotation – Spring injects particular header value into that parameter

8.1.2.1 Channel Adapter

- Connects a `MessageChannel` to some other system or transport
- Typically, the Channel Adapter will do some mapping between the `Message` and object of other system (File, HTTP Request, JMS Message, etc).
- Spring Integration provides a number of Channel Adapters implementations
- *Output channel parameter is optional, since each Message may also provide its own 'Return Address' header. This same rule applies for all consumer endpoints.*
- Types
  - Inbound
    - Can use poller to trigger input message
  - Outbound
- Uni-directional

```
<int:inbound-channel-adapter ref="source2" method="method2" channel="channel2">
      <int:poller cron="30 * 9-17 * * MON-FRI"/>
</int:channel-adapter>
<int:outbound-channel-adapter channel="channel" method="handle">
      <beans:bean class="org.Foo"/>
</int:outbound-channel-adapter>
<int-file:inbound-channel-adapter id="filesIn2"
      directory="file:${input.directory}" filter="customFilterBean" />
<int-jdbc:outbound-channel-adapter
      query="insert into foos (id, status, name) values (:headers[id], 0,
:payload[foo])" data-source="dataSource" channel="input" />
```

8.1.2.2 Messaging Gateway

- Inbound
- Primary purpose of a Gateway is to hide Spring Integration messaging API - clients code interacts with a simple interface only
- It acts as proxy
- Bi-directional
- Typically Gateway will auto-create a temporary, anonymous reply channel, where it will listen for the reply
- Sometimes may prompt you to define a `default-reply-channel` (or `reply-channel` with adapter gateways such as HTTP, JMS, etc.)

```
<int:gateway id="cafeService"
      service-interface="org.cafeteria.Cafe"
      default-request-channel="requestChannel"
      default-reply-channel="replyChannel"/>
```

- Return values of gateway interface method
  - Other type than `Future` – Synchronous

        - Caller of gateway interface is blocked
    - `Future` - Asynchronous
        - Caller isn't blocked
        - Interface method returns `Future`
    - `void`
        - Acts as passive inbound adapter
- Gateway interface methods and its parameters can be annotated with `@Gateway` / `@Header`

```
public interface FileWriter {
      @Gateway(requestChannel="filesOut")
      void write(byte[] content, @Header(FileHeaders.FILENAME) String
                  filename);
}
```

- Apart from above inbound Messaging Gateway (which maps interface onto messaging infrastructure) Spring provides Integration adapters gateways of two types for various protocols (JMS, JPA, HTTP, TCP, RMI, Web Services, ...):
    - Inbound gateway – receives external request, send if for internal processing and sends reply via same protocol / interface
    - Outbound gateway – sends request to external system and waits for response
    - These adapters are alternative API to Spring components like Spring MVC for REST, Spring WS, Spring RMI,...

8.1.2.3 Service Activator

- Generic endpoint for connecting a service instance to the messaging system
- Input Message Channel must be configured
- If the service method to be invoked is capable of returning a value, an output Message Channel may also be provided
- Invokes an operation on some service object (bean) to process the request `Message`
- Extracts the request Message's payload and converts if necessary
- `method` attribute specifies which method would be invoked
- Or `@ServiceActivator` annotation can be used for that purpose
- Arguments may also have `@Header` or `@Headers`
- `void` and `null` return values are supported (means no response), but Outbound Channel Adapter is more suitable for such case
- If inbound gateway expects reply – can set parameter `request-reply` to `true` (throws exception when null)
- Similar endpoint to Gateway, but responsibilities are different
    - Gateway – to map external system into internal messaging infrastructure
    - Service Activator – to map internal application service into internal messaging infrastructure

```
<int:service-activator input-channel="exampleChannel"
      output-channel="replyChannel" ref="somePojo" method="someMethod"/>
```

8.1.2.4 Message Transformer

- Converting a `Message`'s content or structure and returns the modified `Message`
- In fact service activator with specific role
- Commonly used as
    - payload convertor

    - header enricher
    - header filter
- Creates new message, because `Message` is immutable

```
<int:transformer id="testTransformer" ref="testTransformerBean"
input-channel="inChannel" method="transform" output-channel="outChannel"/>
```

- Annotation configuration

```
@Transformer
Order generateOrder(String productId, @Header("customerName") String customer) {
      return new Order(productId, customer);
}
```

8.1.2.5 Filter

- Determines whether a `Message` should be passed to an output channel at all
- Return `boolean` value
- Check for a particular payload content type, a property value, the presence of a header, etc
- Default behaviour is silent discard
- If `Message` is not accepted
    - can be dropped
    - exception can be thrown
- It can support discard channel (acts as simple router)

```
<int:filter input-channel="input" ref="selector"
      output-channel="output" throw-exception-on-rejection="true"/>
<int:filter input-channel="input" ref="selector"
      output-channel="output" discard-channel="rejectedMessages"/>
```

- Annotation configuration

```
@Filter
public boolean dogsOnly(String input) {...}
```

8.1.2.6 Router

- Deciding what channel or channels should receive the Message next (if any)
- Typically the decision is based upon the Message's content and/or metadata available in the Message Headers
- Implementations
    - `PayloadTypeRouter`
    - `HeaderValueRouter`
    - `RecipientListRouter`
    - XPath Router
- Or user can specify custom router

```
<int:router input-channel="input" ref="somePojo" method="someMethod"/>
```

- Output channel parameter is optional
- Annotation configuration (String return value is channel name/s)

```
@Router
public MessageChannel route(Message message) {@Header ...}
@Router
public List<MessageChannel> route(Message message) {...}
@Router
public String route(Foo payload) {...}
@Router
public List<String> route(Foo payload) {...}
```

8.1.2.7 Splitter

- Splits received Message into multiple Messages and sends each of those to its output channel
- Often, they are upstream producers in a pipeline that includes an Aggregator
- `AbstractMessageSplitter` sub-classes or any POJO (method accepts single argument and return a value) can be configured as splitter
- `AbstractMessageSplitter` fills appropriate message headers
  - `CORRELATION_ID`
  - `SEQUENCE_SIZE`
  - `SEQUENCE_NUMBER`

```
<int:splitter id="testSplitter" input-channel="inChannel"
      method="split" output-channel="outChannel">
      <beans:bean class="org.foo.TestSplitter" />
</int:spliter>
```

- Annotation configuration

```
@Splitter
List<LineItem> extractItems(Order order) {
      return order.getItems();
}
```

8.1.2.8 Aggregator

- Receives multiple Messages and combines them into a single Message
- Often downstream consumers in a pipeline that includes a Splitter
- Technically, the Aggregator is more complex than a Splitter, because it is required to maintain state to decide when the complete group of Messages is available and to timeout if necessary
- In case of a timeout, it needs to know whether to send the partial results or to discard them to a separate channel
- Correlates and stores messages, until the group is complete
- At that point, Aggregator combines gathered messages and sends single output message
- Uses there strategies (user can provide custom implementations)
  - `CorrelationStrategy`
    - Default logic is based on `CORRELATION_ID`
    - Custom implementation can be specified via parameters `correlation-method` or `correlation-expression`
  - `ReleaseStrategy`
    - By default will release a group when all `Message`s included in a sequence are present, based on the `SEQUENCE_SIZE` header
    - Custom implementation can be specified via parameters `release-strategy-method` or `release-strategy-expression`

```
aggregator input-channel="input" method="sum" output-channel="output">
      <beans:bean class="org.foo.PojoAggregator"/>
</aggregator>
```

- Annotation configuration

```
public class Waiter {
      ...
      @Aggregator
      public Delivery aggregatingMethod(List<OrderItem> items) {
      ...
```

```
        }
        @ReleaseStrategy
        public boolean releaseChecker(List<Message<?>> messages) {
        ...
        }
        @CorrelationStrategy
        public String correlateBy(OrderItem item) {
        ...
        }
}
```

8.1.3 Error Handling

- Synchronous error handling
  - When error occurs in consumer during synchronous handoff
    - message is failed
    - error is wrapped into `MessageHandlingException`
    - and propagated to sender
- Asynchronous error handling
  - Exception can't be propagated to sender
  - Therefore it is sent to error channel
    - `errorChannel` specified in message header
    - if header is missing in message, global `errorChannel` is used
  - Global `errorChannel`
    - internally created `PublishSubscribeChannel` for sending error messages
    - may be overridden with a custom configuration
  - It is possible to specify router based on exceptions type (`ErrorMessageExceptionTypeRouter`)

```
<int:exception-type-router input-channel="inputChannel"
      default-output-channel="defaultChannel">
      <int:mapping exception-type="java.lang.IllegalArgumentException"
            channel="illegalChannel" />
      <int:mapping exception-type="java.lang.NullPointerException"
            channel="npeChannel" />
</int:exception-type-router>
```

8.1.4 SpEL Expressions

- A lot of endpoints support expression language attribute
- To enable trivial in-line logic

```
<int:router input-channel="inChannel" expression="payload + 'Channel'"/>
<int:filter input-channel="input" expression="payload.equals('nonsense')"/>
```

8.2 How to programmatically create new Messages

~ `MessageBuilder`

- Message interface doesn't provide setters for payload and headers
- `MessageBuilder` is used for constructing messages
- Two factory methods:

```
Message<String> message1 = MessageBuilder.withPayload("test")
                        .setHeader("foo", "bar")
                        .build();
Message<String> message2 = MessageBuilder.fromMessage(message1).build();
Message<String> message3 = MessageBuilder.withPayload("test3")
            .copyHeaders(message1.getHeaders())
```

```
            .build();
Message<String> message4 = MessageBuilder.withPayload("test4")
            .setHeader("foo", 123)
            .copyHeadersIfAbsent(message1.getHeaders())
            .build();
Message<Integer> importantMessage = MessageBuilder.withPayload(99)
            .setPriority(5)
            .build();
Message<Integer> lessImportantMessage =
MessageBuilder.fromMessage(importantMessage)
            .setHeaderIfAbsent(MessageHeaders.PRIORITY, 2)
            .build();
```

~ `MessagingTemplate`

- Provides sending / receiving support
- Also conversion support
- More invasive as usage of Messaging gateway
- Sometimes `MessagingTemplate` is handier (e.g. in unit test)
- `sendTimeout` and `receiveTimeout` properties may also be set on the template

```
MessagingTemplate template = new MessagingTemplate();
Message reply = template.sendAndReceive(someChannel, new GenericMessage("f"));
```

- Some functions provided

```
public boolean send(final MessageChannel channel, final Message<?> message) {
...}
public Message<?> sendAndReceive(final MessageChannel channel, final Message<?
> request) {...}
public Message<?> receive(final PollableChannel<?> channel) {...}
```

8.3 Using chains and bridges

~ Chain

- Chains various endpoints together
- Creates multiple anonymous channels to tie them together
- Chain only requires to specify single input-channel and single output-channel eliminating the need to define channels for each individual component
- Endpoints except last one have to return output (`null` value is accepted)
- Last element in chain have to define `output-channel` or message should have `replyChannel` header
- Reply channel header will not be taken into account within the chain: only after the last handler

```
<int:chain input-channel="input" output-channel="output">
      <int:filter ref="someSelector" throw-exception-on-rejection="true" />
      <int:header-enricher>
            <int:header name="foo" value="bar" />
      </int:header-enricher>
      <int:service-activator ref="someService" method="someMethod" />
</int:chain>
```

- 'black-box' consumer of the message flow

```
<int:chain input-channel="input">
      <si-xml:marshalling-transformer
            marshaller="marshaller" result-type="StringResult" />
      <int:service-activator ref="someService" method="someMethod" />
      <int:header-enricher>
            <int:header name="foo" value="bar" />
      </int:header-enricher>
      <int:logging-channel-adapter level="INFO" log-full-message="true" />
```

```
</int:chain>
```

- Disallowed Attributes and Elements
  - Disallowed attributes on components within the chain
    - order
    - input-channel
    - poller
- Nested chain can be specified in chain with usage of Messaging gateway

~ Bridge

- simply connects two Message Channels or Channel Adapters
- can throttle inbound Messages by providing an intermediary `poller` between two channels
  - `poller`'s trigger will determine the rate at which messages arrive on the second channel
  - `poller`'s "`maxMessagesPerPoll`" property will enforce a limit on the throughput
- another valid use for a Messaging Bridge is to connect two different systems
  - more common is Transformer between the two systems to translate between their formats
- Connecting channels

```
<int:bridge input-channel="pollable" output-channel="subscribable">
      <int:poller max-messages-per-poll="10" fixed-rate="5000"/>
</int:bridge>
```

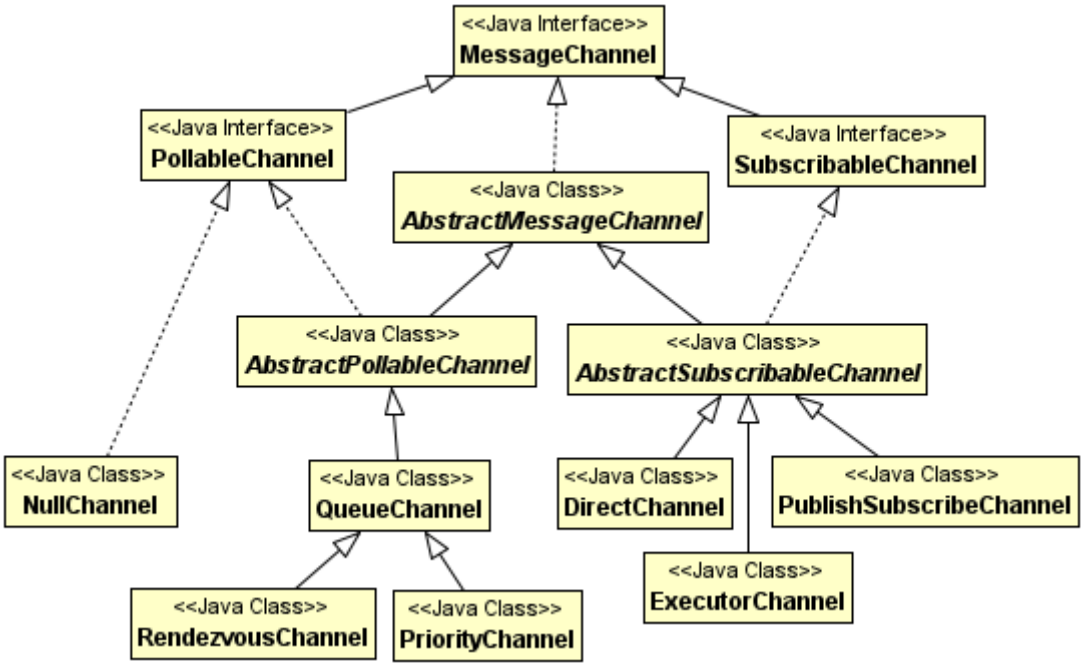
- Connecting channel adapters

```
<int-stream:stdin-channel-adapter id="stdin"/>
<int-stream:stdout-channel-adapter id="stdout"/>
<int:bridge id="echo" input-channel="stdin" output-channel="stdout"/>
```

8.4 The different Channel types and how each of them should be used

- Connects Message Endpoints
- Decouples messaging components
- Point of interception and monitoring of message flow
- Channel is passive component
- Two semantic types:
  - Point-to-Point - one consumer
  - Publish-Subscribe - multiple consumers
- Two handoff types:
  - Synchronous
    - Consumer is registered into channel and triggered when message is being sent
    - Message is immediately received using sender's thread
    - Sender is blocked while message is being processed by consumer endpoint
    - Transaction is spread into consumer
    - Exception is propagated to sender
    - Consumer uses security context of sender
    - Channels:
      - `RendezvousChannel`
      - Synchronous `PublishSubscribeChannel`
      - `DirectChannel`

- Asynchronous
  - Consumer actively requests the message in separate thread/s
  - Consumer can't use transaction and security context of sender
  - Exceptions (typically) can't be propagated to the sender
  - To specify that channel is asynchronous, just specifying of XML sub-element is needed
    - `queue`
    - `task-executor`
  - Channels
    - `QueueChannel`
    - Asynchronous `PublishSubscribeChannel`
    - `ExecutorChannel`
    - `PriorityChannel`
- Simple Spring bean
  - no additional infrastructure needed (e.g. Broker)
  - can be optionally persisted (by JMS or JDBC)
- Implemantations



  - `DirectChannel`
    - Point-to-Point semantics - single consumer
    - Synchronous handoff
    - Dispatches messages directly to subscriber
    - Dispatcher can have `load-balancer` strategy if multiple consumers are subscribed (default strategy is “round-robin”)
    - Dispatcher have also `failover` property
    - Enables a single thread to perform the operations on "both sides" of the channel – this behavior is to support transactions
    - Message producer is blocked by consumers processing – synchronous

```
<int:channel id="directChannel"/>
```

  - `QueueChannel`

        - Point-to-point semantics
        - Asynchronous handoff
        - Stores messages in internal queue until capacity is reached (default value is `Integer.MAX_VALUE`)
        - Enforces first-in/first-out (FIFO) ordering
        - If capacity is reached, sender is blocked until some room is available or timeout is reached
        - Can be configured with persistence store for messages
        - Receiver needs to poll from separate thread
            - `Message` flow will get stuck if there wouldn't be actively consuming `MessageEndpoint` (with `poller`)

```
<int:channel id="queueChannel">
      <queue capacity="25"/>
</int:channel>
```

    - `PriorityChannel`
        - `QueueChannel` sub-class
        - Processes classes based on `priority` attribute in message header
        - Comparator type custom logic can be specified via constructor

```
<int:channel id="priorityChannel" datatype="example.Widget">
      <int:priority-queue comparator="widgetComparator" capacity="10"/>
</int:channel>
```

    - `RendezvousChannel`
        - `QueueChannel` sub-class
        - Direct handoff scenario – sender is blocked until poller invoked `receive()`
        - It uses a `SynchronousQueue` (a zero-capacity implementation of `BlockingQueue`)
        - Sender knows that some receiver has accepted the message

```
<int:channel id="rendezvousChannel"/>
      <int:rendezvous-queue/>
</int:channel>
```

    - `PublishSubscribeChannel`
        - Broadcast message to all subscribers
        - Is intended for sending only
        - Consumers can't poll for messages
        - Have `apply-sequence` property for implementing *Resequencer* or *Aggregator* EI patterns
        - Handoff types
            - synchronous – publishes message in the sender's thread

```
<int:publish-subscribe-channel id="pubsubChannel"/>
```

            - asynchronous – using `TaskExecutor`

```
<int:publish-subscribe-channel id="pubsubChannel" task-executor="someExecutor"/>
```

    - `ExecutorChannel`
        - Point-to-point semantics
        - Asynchronous handoff
        - Supports same dispatcher configuration (`load-balancer` and `failover` properties) as `DirectChannel`
        - Delegates to `TaskExecutor` to perform dispatch – handler invocation doesn't block senders thread
        - Does not support transaction spanning between sender and receiver

```
<int:channel id="executorChannelWithoutFailover">
```

```
      <int:dispatcher task-executor="someExecutor" failover="false"/>
</int:channel>
```

8.4.1 ChannelInterceptor

- Can be configured
  - Individually for each channel

```
public interface ChannelInterceptor {
      Message<?> preSend(Message<?> message, MessageChannel channel);
      void postSend(Message<?> message, MessageChannel channel, boolean sent);
      boolean preReceive(MessageChannel channel);
      Message<?> postReceive(Message<?> message, MessageChannel channel);
}
```

    - Methods with return type `Message<?>`
      - Can be used for transforming messages
      - Or with `null` value prevent further processing (of course any method can throw `RuntimeException`)
    - `preReceive` can return `false` to prevent receive operation to process
  - Globally for all channels
    - User can specify pattern to match channels to intercept
    - `order` attribute allows ordering in case of multiple global interceptors

```
<int:channel-interceptor pattern="input*, bar*, foo" order="3">
      <bean class="foo.barSampleInterceptor"/>
</int:channel-interceptor>
```

    - WireTab
      - EI pattern
      - Interceptor that sends the Message to another channel without altering existing flow
      - Useful for debugging and monitoring

```
<int:channel id="in">
      <int:interceptors>
            <int:wire-tap channel="logger" />
      </int:interceptors>
</int:channel>
<int:logging-channel-adapter id="logger" level="DEBUG" />
```

    - Global Wire Tap

```
<int:wire-tap pattern="input*, bar*, foo" order="3" channel="wiretapChannel"/>
```

8.4.2 Special Channels

- `nullChannel`
  - logs any `Message` sent to it at DEBUG level and returning immediately
  - name `nullChannel` is reserved in application context
- `errorChannel`
  - internally created `PublishSubscribeChannel` for sending error messages
  - may be overridden with a custom configuration

8.4.3 Temporary reply channels

- Are used by
  - Inbound gateways, that doesn't specify `reply-channel` explicitly
    - anonymous
    - point-to-point

        - message header `replyChannel` is created automatically
    - Producer components without output channel specified
        - message is converted into output
- Automatically disposed after processing message
- Good practice is to specify channel explicitly only when it is needed
    - intercepting
    - publish-subscribe

8.4.4 Point-to-Point Dispatcher

- Can be configured for point-to-point `SubscridableChannel`s
- Message is delivered only to one handler
- But there can be more subscribers
- Dispatcher has load-balancer with fail-over support
- Default implementation is round-robin
- Each handler can have order for fail-over specified
- User can disable fail-over / load-balancing
- Synchronous – `DirectChannel`
    - Exception is propagated immediately if fail-over is disabled
- Asynchronous – `ExecutorChannel`
    - Sender of the message is not blocked
    - Message delivery is processed in different thread

8.5 The corresponding effects on things like transactions and security

- Two handoff types:
    - Synchronous
        - Transaction is spread into consumer
        - Exception is propagated to sender
        - Consumer uses security context of sender
        - Low overhead
        - No scaling possible
    - Asynchronous
        - Consumer can't use transaction and security context of sender
        - Exceptions can't (typically) be propagated to the sender
        - sending method can not assume anything about the relative timing of the method `send()` returns, and the delivery and processing of the message
        - To specify that channel is asynchronous, just specifying of XML sub-element is needed
            - `queue`
            - `task-executor`

8.6 The need for active polling and how to configure that

~ `MessageEndpoint` is by default passive component – just waits to invoked
~ Consumer endpoints can be changed to active component by using of `poller`

- Is needed for retrieval of messages from `PollableChannel`
- Is used as sub-element in XML configuration of message endpoint
- It is polling in one thread by default

- Poller is using trigger for polling
  - standard implementations via properties
    - `cron`
    - `fixed-delay` - fixed gap between executions
    - `fixed-rate` - fixed start execution times
  - or can use trigger reference via property `trigger`

```
<int:poller id="defaultPoller" max-messages-per-poll="5" fixed-rate="3000"/>
```

- Can use various threads from pool when using `task-executor`
- `max-messages-per-poll` – maximum number of messages to pick up per one poll operation
- `receive-timeout`
  - amount of time the poller should wait if no messages are available when it invokes the receive operation
  - receiver thread is blocked during timeout interval but is able to retrieve message immediately while waiting
- Long polling technique can emulate event-driven behaviour when interval trigger is short (e.g. 50 ms) and `receive-timeout` is long (e.g. 5 s)

~ Global default poller

- single top level poller with default `property` set to `true`
- applies for consumer endpoint with `PollableChannel` as `input-channel`
- each such endpoint can override this by local poller

```
<int:poller id="defaultPoller" default="true" fixed-rate="3000"/>
```

~ Transaction support for poller

- each receive-and-forward operation can be performed as an atomic unit-of-work
- simply add the `transactional` sub-element
- transaction includes `receive()` call on `PollableChannel`

```
<int:poller fixed-delay="1000">
      <int:transactional transaction-manager="txManager" propagation="REQUIRED"
            isolation="REPEATABLE_READ" timeout="10000" read-only="false" />
</int:poller>
```

9 Spring Batch

9.1 Main concepts (Job, Step, Job Instance, Job Execution, Step Execution, etc.)

~ Spring Batch is a lightweight, comprehensive batch framework designed to enable the development of robust batch applications vital for the daily operations of enterprise systems

~ Provides reusable functions that are essential in processing

- large volumes of records (usually not suitable for short transactions)
- time based events (month-end calculations, notices or correspondence)
- without user interaction (often handling restarts, error, retries, ...)
- long-running jobs (usually in off peak hours/days)

~ Business Scenarios

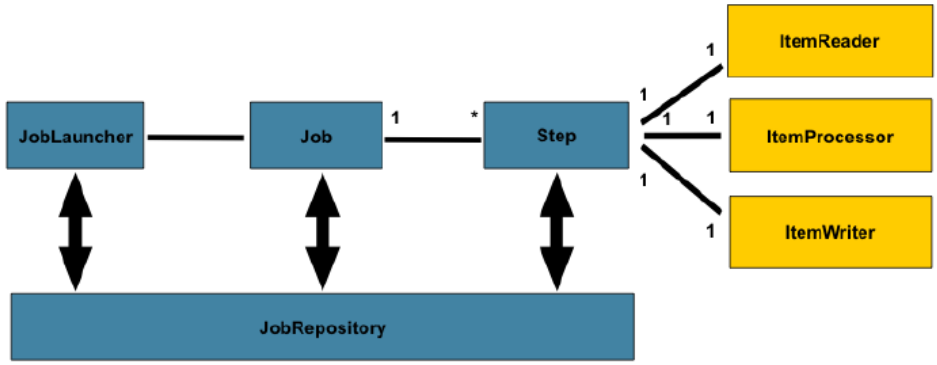
- Commit batch process periodically
- Concurrent batch processing: parallel processing of a job
- Staged, enterprise message-driven processing
- Massively parallel batch processing

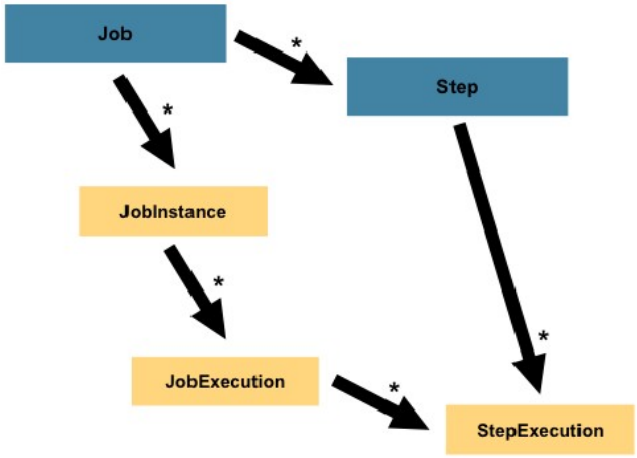
- Manual or scheduled restart after failure
- Sequential processing of dependent steps (with extensions to workflow-driven batches)
- Partial processing: skip records (e.g. on rollback)
- Whole-batch transaction: for cases with a small batch size or existing stored procedures/scripts

~ Technical Objectives

- Batch developers use the Spring programming model: concentrate on business logic; let the framework take care of infrastructure.
- Clear separation of concerns between the infrastructure, the batch execution environment, and the batch application.
- Provide common, core execution services as interfaces that all projects can implement.
- Provide simple and default implementations of the core execution interfaces that can be used ‘out of the box’.
- Easy to configure, customize, and extend services, by leveraging the spring framework in all layers.
- All existing core services should be easy to replace or extend, without any impact to the infrastructure layer.
- Provide a simple deployment model, with the architecture JARs completely separate from the application, built using Maven.

~ Spring Batch Domain





9.1.1 Job

- Entity that encapsulates an entire batch process
- Container for `Step`s
- Combines multiple `Step`s that belong logically together in a flow
- Allows for configuration of properties global to all steps, such as restartability
- `Job` configuration contains
  - The simple name of the `Job`
  - Definition and ordering of `Step`s
  - Whether or not the `Job` is restartable
- Default Implementation is `SimpleJob`
  - Creates standard functionality on top of `Job`
  - Batch namespace abstracts away the need to instantiate it directly
- Configuring a `Job`
  - Example of job configuration

```
<job id="footballJob" job-repository="specialRepository">
      <step id="playerload" parent="s1" next="gameLoad" />
      <step id="gameLoad" parent="s2" next="playerSummarization" />
      <step id="playerSummarization" parent="s3" />
      <listeners>
            <listener ref="sampleListener" />
      </listeners>
</job>
```

  - Three required parameters/dependencies
    - name / ID
    - `JobRepository` instance
      - if `job-repository` is not defined, expects bean with name

jobRepository
        - list of Steps
    - Optional parameters
        - restartable
            - launching of a Job is considered to be a 'restart' if a JobExecution already exists for the particular JobInstance
            - default value is true
            - Restart can start
                - Where job ended up last execution - need to persist ExecutionContext intance (for Job and Step instances)
                - From the beginning if the state is not persisted
            - *Entirely up to the developer to ensure that a new JobInstance is created in this scenario*
        - listeners
            - specifies list of Job's interceptors

```
public interface JobExecutionListener {
	void beforeJob(JobExecution jobExecution);
	void afterJob(JobExecution jobExecution);
}
```

            - Annotations @BeforeJob and @AfterJob
        - split – for Steps parrallelisation

```
<split id="split1" next="step4">
	<flow>
		<step id="step1" parent="s1" next="step2" />
		<step id="step2" parent="s2" />
	</flow>
	<flow>
		<step id="step3" parent="s3" />
	</flow>
</split>
<step id="step4" parent="s4" />
```

        - decision – for declarative flow control
        - flow – for externalization of flow definitions and reusability

```
<job id="job">
	<flow id="job1.flow1" parent="flow1" next="step3" />
	<step id="step3" parent="s3" />
</job>
<flow id="flow1">
	<step id="step1" parent="s1" next="step2" />
	<step id="step2" parent="s2" />
</flow>
```

- Job Inheritance
    - concrete Jobs may inherit properties
    - "child" Job will combine its elements and attributes with the parent's
    - child will override any of the parent's properties by default
    - merge can be used to combine configurations

```
<job id="baseJob" abstract="true">
	<listeners>
		<listener ref="listenerOne" />
	</listeners>
</job>
<job id="job1" parent="baseJob">
	<step id="step1" parent="standaloneStep" />
```

```
        <listeners merge="true">
                <listener ref="listenerTwo" />
        </listeners>
</job>
```

9.1.2 JobInstance

- each individual run of the `Job` must be tracked separately
- `JobInstance` represents logical job run
- each `JobInstance` can have multiple executions (`JobExecution`)
- only one `JobInstance` corresponding to a particular `Job` and identified by `JobParameters` can be running at a given time
- reusing the same `JobInstance` will determine, the 'state' (i.e. the `ExecutionContext`) from previous executions will be used
- Using a new `JobInstance` will mean 'start from the beginning'

9.1.3 JobLauncher

- Simple interface for launching a `Job` with a given set of `JobParameters`
- It is expected that implementations will obtain a valid `JobExecution` from the `JobRepository` and execute the `Job`

```
public interface JobLauncher {
        public JobExecution run(Job job, JobParameters jobParameters) throws
                JobExecutionAlreadyRunningException, JobRestartException,
                JobInstanceAlreadyCompleteException, JobParametersInvalidException;
}
```

- Two invocation behaviours
  - Synchronous
    - Client (invoker) is blocked until the end of processing
    - `JobExecution` contains `ExitStatus.FINISHED` or `ExitStatus.FAILED`
    - Ususaly started from scheduler
  - Asynchronous
    - `run()` method does not block the client
    - `TaskExecutor` is configured for `JobLauncher`
    - `JobExecution` contains `ExitStatus.UNKNOWN`
    - Usefull for HTTP request

```
<bean id="jobLauncher"
        class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
        <property name="jobRepository" ref="jobRepository" />
</bean>
```

- Running a `Job`
  - From command line
    - Java launcher class `CommandLineJobRunner`
      - Command line parameters
        - jobPath – spring context XML configuration
        - jobName – name/ID of the job to run
        - Optional job parameters – key(type)=value pairs

```
bash$ java CommandLineJobRunner endOfDayJob.xml endOfDay
schedule.date(date)=2007/05/05
```

      - `ExitCodeMapper` converts `ExitStatus` returned as part of `JobExecution` from `JobLauncher` into process exit code return value

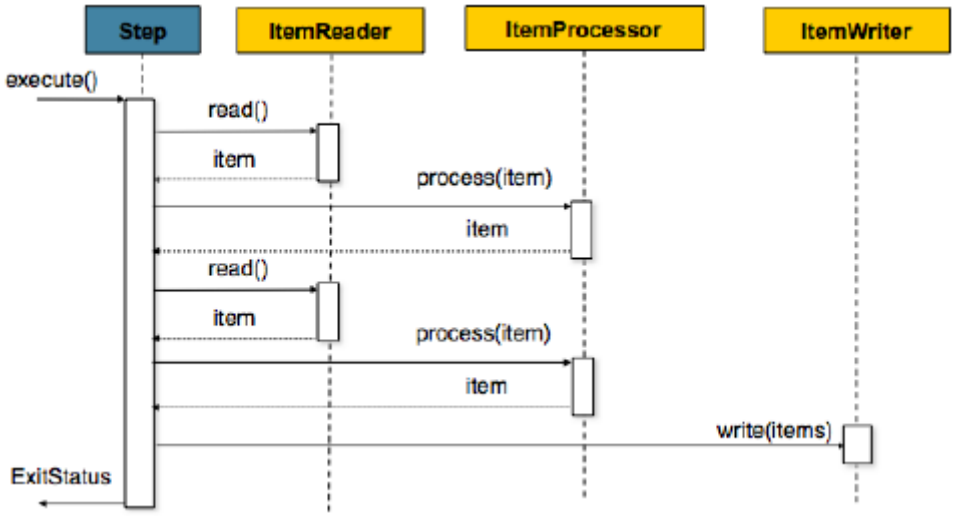
(0=success,...)

- From Spring driven application
  - Autowire or inject `jobLauncher` bean into client spring bean

9.2 The interfaces typically used to implement a chunk-oriented Step

9.2.1 Step

- Encapsulates an independent, sequential phase of a batch job
- Contains all of the information necessary to define and control the actual batch processing
- Can be as simple or complex as the developer desires
- Uses Chunk Oriented processing style
  - One item is read in from an `ItemReader`
  - Optionally handed to an `ItemProcessor`,
  - Aggregated
  - Entire chunk is written out via the `ItemWriter` when number of items read equals the commit interval
  - Transaction is committed



9.2.2 Configuring a `Step`

```
<job id="sampleJob" job-repository="jobRepository">
	<step id="step1">
		<tasklet transaction-manager="transactionManager">
			<chunk reader="itemReader" writer="itemWriter" commit-
				interval="10" />
		</tasklet>
	</step>
</job>
```

- Namespace parameters
  - Required
    - `reader`
    - `writer`
    - `transaction-manager` (defaults to `transactionManager`)
    - `job-repository` (defaults to `jobRepository`)
    - `commit-interval`
  - Optional

- `start-limit` (on `tasklet` tag) – how many times can be `Step` executed
- `allow-start-if-complete`
  - Allow re-run previously completed step when job is restarting
  - Default is `false`

```
<step id="gameLoad" next="playerSummarization">
      <tasklet allow-start-if-complete="true">
            <chunk reader="gameFileItemReader" writer="gameWriter"
                  commit-interval="10" />
      </tasklet>
</step>
<step id="playerSummarization">
      <tasklet start-limit="3">
            <chunk reader="playerSummarizationSource" writer="summaryWriter"
                  commit-interval="10" />
      </tasklet>
</step>
```

- `skip-limit` (on `chunk` element) – skip the chunk for particular exceptions (specified by `skippable-exception-classes` (`chunk` sub-element))
  - In many scenarions processing shouldn't be terminated by error
  - It may be suitable to skip certain amount of errornous data
  - exceptions thrown from the `ItemReader` will not cause a rollback in this scenario

```
<step id="step1">
      <tasklet>
            <chunk reader="flatFileItemReader" writer="itemWriter"
                  commit-interval="10" skip-limit="10">
                  <skippable-exception-classes>
                        <include class="java.lang.Exception" />
                        <exclude class="java.io.FileNotFoundException" />
                  </skippable-exception-classes>
            </chunk>
      </tasklet>
</step>
```

- `retry-limit`
  - retry the chunk for particular exceptions
  - useful when error is transient and retry might succeed
  - supports also `exclude`

```
<step id="step1">
      <tasklet>
            <chunk reader="itemReader" writer="itemWriter" commit-interval="2"
                  retry-limit="3">
                  <retryable-exception-classes>
                        <include class="org.springframework.
                              dao.DeadlockLoserDataAccessException"/>
                  </retryable-exception-classes>
            </chunk>
      </tasklet>
</step>
```

- `no-rollback-exception-classes` (`tasklet` sub-element) - exceptions thrown from the `ItemWriter` do not cause a rollback
- `is-reader-transactional-queue` – if there is needed rollback/commit for input queue (e.g. JMS input queue) – default is `false`
- `transaction-attributes` (`tasklet` sub-element)

```
<transaction-attributes isolation="DEFAULT" propagation="REQUIRED"
      timeout="30"/>
```

- `processor-transactional` (on `chunk` element) – turns transactions on for `ItemProcessor`
- `streams` (`chunk` sub-element) - for opening, closing resources, updating state into `ExecutionContext` instance

9.2.3 Inheritance + abstract Step

- default behaviour is overidding parent tags
- can merge lists
- parent step can be defined as abstract

```
<step id="listenersParentStep" abstract="true">
      <listeners>
            <listener ref="listenerOne" />
      <listeners>
</step>
<step id="concreteStep3" parent="listenersParentStep">
      <tasklet>
            <chunk reader="itemReader" writer="itemWriter" commit-interval="5"/>
      </tasklet>
      <listeners merge="true">
            <listener ref="listenerTwo" />
      <listeners>
</step>
```

9.2.4 Intercepting `Step` execution

- Each interceptor type can be configured via `liteners` / `listener` parameter of `Step` (see above)
- Can register callbacks for monitoring, logging, error or state handling
- `StepExecutionListener`

```
public interface StepExecutionListener extends StepListener {
      void beforeStep(StepExecution stepExecution);
      ExitStatus afterStep(StepExecution stepExecution);
}
```

  - `@BeforeStep`
  - `@AfterStep`

- `ChunkListener`

```
public interface ChunkListener extends StepListener {
      void beforeChunk();
      void afterChunk();
}
```

  - `@BeforeChunk`
  - `@AfterChunk`

- `ItemReadListener`

```
public interface ItemReadListener<T> extends StepListener {
      void beforeRead();
      void afterRead(T item);
      void onReadError(Exception ex);
}
```

  - `@BeforeRead`
  - `@AfterRead`
  - `@OnReadError`

- `ItemProcessListener`

```
public interface ItemProcessListener<T, S> extends StepListener {
      void beforeProcess(T item);
```

```
      void afterProcess(T item, S result);
      void onProcessError(T item, Exception e);
}
```

- `@BeforeProcess`
- `@AfterProcess`
- `@OnProcessError`

- `ItemWriteListener`

```
public interface ItemWriteListener<S> extends StepListener {
      void beforeWrite(List<? extends S> items);
      void afterWrite(List<? extends S> items);
      void onWriteError(Exception exception, List<? extends S> items);
}
```

- `@BeforeWrite`
- `@AfterWrite`
- `@OnWriteError`

- `SkipListener`
  - can intercept skipping single item on error
  - called right before commiting the transaction after chunk processing

```
public interface SkipListener<T,S> extends StepListener {
      void onSkipInRead(Throwable t);
      void onSkipInWrite(S item, Throwable t);
      void onSkipInProcess(T item, Throwable t);
}
```

  - To cooperate properly with transactions
    - appropriate skip method (depending on when the error happened) will only be called once per item
    - will always be called just before the transaction is committed to ensure that any transactional resources call by the listener are not rolled back by a failure within the `ItemWriter`
  - `@OnSkipInRead`
  - `@OnSkipInWrite`
  - `@OnSkipInProcess`

9.2.5 `TaskletStep`

- alternative to Chunk-oriented processing
- one atomic execution scenario
- `Tasklet.execute()` will be called releatedlly by `TeskletStep` until it either returns `RepeatStatus.FINISHED` or throws an exception to signal a failure

```
<step id="step1">
      <tasklet ref="myTasklet"/>
</step>
```

9.2.6 Controlling `Step` flow

- Control how the job 'flows' from one step to another
- Failure of a `Step` doesn't necessarily mean that the `Job` should fail
- May be more than one type of 'success' which determines which Step should be executed next
- Depending upon how a group of Steps is configured, certain steps may not even be processed at all
- Sequential flow

```
<job id="job">
```

```
      <step id="stepA" parent="s1" next="stepB" />
      <step id="stepB" parent="s2" next="stepC" />
      <step id="stepC" parent="s3" />
</job>
```

- Conditional flow

```
<job id="job">
      <step id="stepA" parent="s1">
            <next on="*" to="stepB" />
            <next on="FAILED" to="stepC" />
      </step>
      <step id="stepB" parent="s2" next="stepC" />
      <step id="stepC" parent="s3" />
</job>
```

- Configuring for Stop
  - 'End' Element
    - instructs a `Job` to stop with a `BatchStatus` of `COMPLETED`

```
<step id="step1" parent="s1" next="step2">
<step id="step2" parent="s2">
      <end on="FAILED" />
      <next on="*" to="step3" />
</step>
<step id="step3" parent="s3">
```

  - 'Fail' Element
    - instructs a `Job` to stop with a `BatchStatus` of `FAILED`

```
<step id="step1" parent="s1" next="step2">
<step id="step2" parent="s2">
      <fail on="FAILED" exit-code="EARLY TERMINATION" />
      <next on="*" to="step3" />
</step>
<step id="step3" parent="s3">
```

  - 'Stop' Element
    - instructs a `Job` to stop with a `BatchStatus` of `STOPPED`

```
<step id="step1" parent="s1">
      <stop on="COMPLETED" restart="step2"/>
</step>
<step id="step2" parent="s2"/>
```

- Programmatic flow control
  - implementing interface `JobExecutionDecider`
  - and use it in Job configuration:

```
<job id="job">
      <step id="step1" parent="s1" next="decision" />
      <decision id="decision" decider="decider">
            <next on="FAILED" to="step2" />
            <next on="COMPLETED" to="step3" />
      </decision>
      <step id="step2" parent="s2" next="step3" />
      <step id="step3" parent="s3" />
</job>
```

9.2.7 `ItemReader`

- abstraction that represents the retrieval of input for a `Step`
- one item at a time
- `null` return value indicates that no more items are left
- expected that item will be mapped into domain object (generic type T below)

```
public interface ItemReader<T> {
```

```
      T read() throws Exception, UnexpectedInputException, ParseException,
                  NonTransientResourceException;
}
```

9.2.8 `ItemProcessor`

- optional part of the `Step`
- abstraction that represents the business processing of an item
- one item at a time
- it is expected to transform one object to another (interface with two generic types)
- returning `null` indicates that the item should not be written out

```
public interface ItemProcessor<I, O> {
      O process(I item) throws Exception;
}
```

9.2.9 `ItemWriter`

- abstraction that represents the output of a `Step`
- one batch or chunk of items at a time (list of items)
- has no knowledge of the input it will receive next

```
public interface ItemWriter<T> {
      void write(List<? extends T> items) throws Exception;
}
```

9.2.10 Stateful item processing

- `ExecutionContext` – represents a collection of key/value pairs that are persisted and controlled by the framework in order to allow developers a place to store persistent state that is scoped to a `StepExecution` or `JobExecution`

```
executionContext.putLong(getKey(LINES_READ_COUNT), reader.getPosition());
ExecutionContext ecStep = stepExecution.getExecutionContext();
ExecutionContext ecJob = jobExecution.getExecutionContext();
```

- State can be passed between steps when `ItemReader`/`ItemWriter`
  1. Implements `StepExecutionListener` (more info in section “Intercepting `Step` execution”)
     - Can use `ExecutionContext` instance store/get state in reader/writer
  2. Uses `@BeforeStep` and `@AfterStep`
     - Same as above
  3. Implements `ItemStream`
     - Lifecycle
       - `open` – invoked before `read`
       - `update` – invoked right before commit of each `chunk`
       - `close` – invoked at the end of the `Step`

```
public interface ItemStream {
      void open(ExecutionContext executionContext) throws ItemStreamException;
      void update(ExecutionContext executionContext) throws
ItemStreamException;
      void close() throws ItemStreamException;
}
```

- Care must be taken in multithreaded scenarios

9.2.11 `ExecutionContextPromotionListener`

- Used for passing data to future `Step`s

- Data are "promoted" to the `Job`'s `ExecutionContext` after the step has finished
- Listener must be configured with the keys related to the data in the `ExecutionContext` that must be promoted
- Optionally can be configured with a list of exit code patterns for which the promotion should occur ("`COMPLETED`" is the default)

```
public class SavingItemWriter implements ItemWriter<Object> {
	private StepExecution stepExecution;

	public void write(List<? extends Object> items) throws Exception {
		// ...

		ExecutionContext stepContext = this.stepExecution
				.getExecutionContext();
		stepContext.put("someKey", someObject);
	}

	@BeforeStep
	public void saveStepExecution(StepExecution stepExecution) {
		this.stepExecution = stepExecution;
	}
}
```

```
<job id="job1">
	<step id="step1">
		<tasklet>
			<chunk reader="reader" writer="savingWriter"
				commit-interval="10" />
		</tasklet>
		<listeners>
			<listener ref="promotionListener" />
		</listeners>
	</step>
	<step id="step2">
		...
	</step>
</job>
```

```
public class RetrievingItemWriter implements ItemWriter<Object> {
	private Object someObject;

	public void write(List<? extends Object> items) throws Exception {
		// ...
	}

	@BeforeStep
	public void retrieveInterstepData(StepExecution stepExecution) {
		JobExecution jobExecution = stepExecution.getJobExecution();
		ExecutionContext jobContext = jobExecution.getExecutionContext();
		this.someObject = jobContext.get("someKey");
	}
}
```

9.2.12 Late binding of `Job` and `Step` attributes

- It is possible to parameterize some attribute in the `JobParameters` with late binding

```
<bean id="flatFileItemReader" scope="step"
	class="org.springframework.batch.item.file.FlatFileItemReader">
```

```
      <property name="resource" value="#{jobParameters['input.file.name']}"
/>
</bean>
```

- Both the `JobExecution` and `StepExecution` level `ExecutionContext` can be accessed in the same way

```
<property name="resource" value="#{jobExecutionContext['input.file.name']}" />
<property name="resource" value="#{stepExecutionContext['input.file.name']}" />
```

- `Step` scope
  - Required in order to use late binding since the bean cannot actually be instantiated until the `Step` starts, which allows the attributes to be found

9.2.13 Implementations of `ItemReader` and `ItemWriter`

Spring Batch provides various implementations of `ItemReader` and `ItemWriter` for

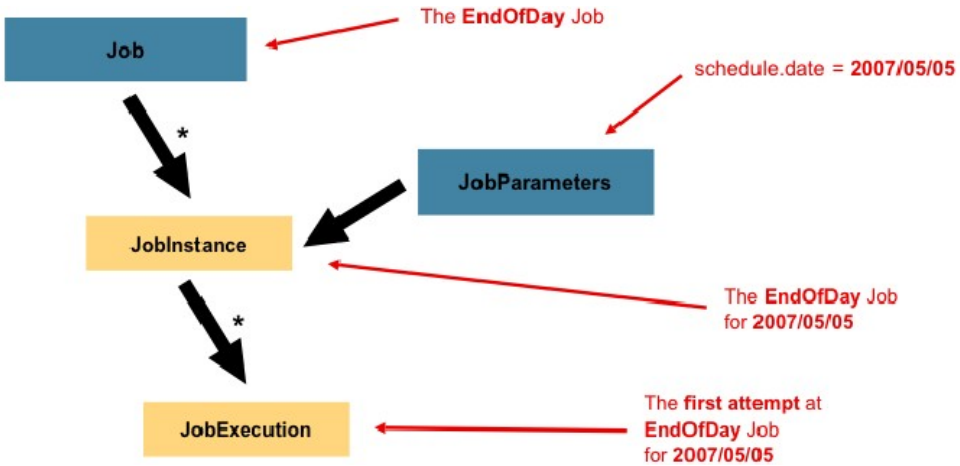
- Flat files
- JDBC
- Hibernate
- JMS
- XML

9.3 How and where state can be stored

~ Persistence mechanism for storing batch meta-data
~ Provides CRUD operations of the various persisted domain objects within Spring Batch, such as `JobExecution` and `StepExecution`
~ Required by many of the major framework features, such as the `JobLauncher`, `Job`, and `Step`
~ Values in following example are defaults (no need to specify them)

```
<job-repository id="jobRepository"
      data-source="dataSource"
      transaction-manager="transactionManager"
      isolation-level-for-create="SERIALIZABLE"
      table-prefix="BATCH_" max-varchar-length="1000" />
```

9.4 What are job parameters and how are they used



~ Set of parameters used to start a batch job
~ They can be used for identification or even as reference data of `JobInstance` during the run
~ Framework allows the submission of a Job with parameters that do not contribute to the identity of a `JobInstance` as well
~ Given a `JobParameters` object, it will return the 'next' `JobParameters` object by incrementing any necessary values it may contain

9.5 What is a FieldSetMapper and what is it used for

~ Spring Batch contains implementations of readers/writers from/to files
~ File reading is stateful, because `ItemReader` needs to know next line to read
~ Anyone reading flat file must understand ahead of time exactly how the file is structured
~ Two types of files

- Delimited
- Fixed length

9.5.1 `FieldSet` interface

- Abstraction for enabling the binding of fields from a file resource
- Conceptually very similar to a JDBC `ResultSet`
- Immutable
- Enables consistent behaviour
  - when handling errors caused by a format exception
  - when doing simple data conversions

```
String[] tokens = new String[]{"foo", "1", "true"};
FieldSet fs = new DefaultFieldSet(tokens);
String name = fs.readString(0);
int value = fs.readInt(1);
boolean booleanValue = fs.readBoolean(2);
```

~ Three basic steps are required when reading a file

1. Read one line from the file
2. Pass the string line into the `LineTokenizer#tokenize()` method, in order to retrieve a `FieldSet` (see `LineTokenizer` section below)
3. Pass the `FieldSet` returned from tokenizing to a `FieldSetMapper`, returning the result from the `ItemReader#read()` method (see `FieldSetMapper` section below)

9.5.2 `FlatFileItemReader`

- Out of the box implementation of `ItemReader`
- Provides basic functionality for reading and parsing flat files
- Two important dependencies
  - `Resource`
    - Represents a Spring Core `Resource`

```
Resource resource = new FileSystemResource("resources/trades.csv");
```

  - `LineMapper`
    - Basic contract is that, given the current line and the line number with which it is associated, the mapper should return a resulting domain object

```
public interface LineMapper<T> {
	T mapLine(String line, int lineNumber) throws Exception;
}
```

9.5.3 `LineTokenizer`

- Abstraction for turning a line into a `FieldSet`
- Contract of a `LineTokenizer` is such that, given a line of input (in theory the String could encompass more than one line), a `FieldSet` representing the line will be returned

```
public interface LineTokenizer {
	FieldSet tokenize(String line);
}
```

- Implementations
  - `DelmitedLineTokenizer` – fields in a record are separated by a delimiter
  - `FixedLengthTokenizer` – fields in a record are each a 'fixed width'
  - `PatternMatchingCompositeLineTokenizer` – delimited against a pattern

9.5.4 `FieldSetMapper`

- Takes a `FieldSet` object and maps its contents to an object (e.g. custom DTO, a domain object, or a simple array)
- Used in conjunction with the `LineTokenizer` to translate a line of data from a resource into an object of the desired type

```
public interface FieldSetMapper<T> {
	T mapFieldSet(FieldSet fieldSet) throws BindException;
}
```

- Out of the box implementations
  - `PassthroughFieldSetMapper`
    - returns unchanged `FieldSet` instance
    - no domain model (direct copy) scenario
  - `BeanWrapperFieldSetMapper`
    - `FieldSetMapper` that automatically maps fields by matching a field name with a setter on the object using the `JavaBean` specification
    - field names are used convert into prototype object
    - `TypeConverter` is used for binding

```
<bean id="fieldSetMapper"
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
	<property name="prototypeBeanName" value="player" />
</bean>
<bean id="player" class="org.springframework.batch.sample.domain.Player"
	scope="prototype" />
```

9.5.5 DefaultLineMapper

- Default iplementation of 2nd and 3rd step of reading files
- Most users will use it

```
public class DefaultLineMapper<T> implements LineMapper<T>, InitializingBean{
	private LineTokenizer tokenizer;
	private FieldSetMapper<T> fieldSetMapper;

	public T mapLine(String line, int lineNumber) throws Exception {
		return fieldSetMapper.mapFieldSet(tokenizer.tokenize(line));
	}

	public void setLineTokenizer(LineTokenizer tokenizer) {
		this.tokenizer = tokenizer;
	}

	public void setFieldSetMapper(FieldSetMapper<T> fieldSetMapper) {
		this.fieldSetMapper = fieldSetMapper;
	}
}
```

9.5.6 Mapping Fields by Name

```
tokenizer.setNames(new String[] {"ID", "birthYear"}
...
public class PlayerMapper implements FieldSetMapper<Player> {
	public Player mapFieldSet(FieldSet fs) {
		if (fs == null) {
			return null;
		}
		Player player = new Player();
		player.setID(fs.readString("ID"));
		player.setBirthYear(fs.readInt("birthYear"));
	}
}
```

9.5.7 Multiple Record Types within a Single File

- File

```
USER;Smith;Peter;;T;20014539;F
LINEA;1044391041ABC037.49G201XX1383.12H
LINEB;2134776319DEF422.99M005LI
```

- Parsing configuration

```
<bean id="orderFileLineMapper"
	class="org.spr...PatternMatchingCompositeLineMapper">
	<property name="tokenizers">
		<map>
			<entry key="USER*" value-ref="userTokenizer" />
			<entry key="LINEA*" value-ref="lineATokenizer" />
			<entry key="LINEB*" value-ref="lineBTokenizer" />
		</map>
	</property>
	<property name="fieldSetMappers">
		<map>
			<entry key="USER*" value-ref="userFieldSetMapper" />
			<entry key="LINE*" value-ref="lineFieldSetMapper" />
		</map>
	</property>
</bean>
```

9.6 Additional topics

9.6.1 DB `ItemReaders`

~ Cursor Based `ItemReaders`

- `RowMapper` callback with `JdbcTemplate` are loading all data during the DB query
  - Not efficient
- Generally the default approach of most batch developers
- Java `ResultSet` class is essentially an object orientated mechanism for manipulating a cursor
- `ResultSet` maintains a cursor to the current row of data
- Calling next on a `ResultSet` moves this cursor to the next row
- Close method will then be called to ensure all resources are freed up
- Various implementations possible
  - `JdbcCursorItemReader`
    - works directly with a `ResultSet`
    - Requires a SQL statement to run against a connection obtained from a `DataSource`
    - Similar usage to `JdbcTemplate`

```
<bean id="itemReader" class="org.spr...JdbcCursorItemReader">
      <property name="dataSource" ref="dataSource" />
      <property name="sql" value="select ID, NAME, CREDIT from CUSTOMER" />
      <property name="rowMapper">
            <bean
      class="org.springframework.batch.sample.domain.CustomerCreditRowMapper" />
      </property>
</bean>
```

  - `HibernateCursorItemReader`
  - `StoredProcedureItemReader`

~ Paging `ItemReaders`

- Executing multiple queries where each query is bringing back a page of the results
- Each query that is executed must specify the starting row number and the number of rows that we want returned for the page
- Implementations
  - `JdbcPagingItemReader`
    - Each database has its own strategy for providing paging support
    - We need to use a different PagingQueryProvider for each supported database type
    - There is also the `SqlPagingQueryProviderFactoryBean` that will auto-detect the database – recommended best practise
    - Requires that you specify a select clause and a from clause.
    - You can also provide an optional where clause
    - These clauses will be used to build an SQL statement combined with the required `sortKey` (required to be unique)
    - It will pass back one item per call to read in the same basic fashion as any other `ItemReader`
    - The paging happens behind the scenes when additional rows are needed

```
<bean id="itemReader" class="org.spr...JdbcPagingItemReader">
      <property name="dataSource" ref="dataSource" />
      <property name="queryProvider">
            <bean class="org.spr...SqlPagingQueryProviderFactoryBean">
```

```
                <property name="selectClause" value="select id,name,credit"/>
                <property name="fromClause" value="from customer" />
                <property name="whereClause" value="where status=:status" />
                <property name="sortKey" value="id" />
            </bean>
        </property>
        <property name="parameterValues">
            <map>
                <entry key="status" value="NEW" />
            </map>
        </property>
        <property name="pageSize" value="1000" />
        <property name="rowMapper" ref="customerMapper" />
</bean>
```

- `JpaPagingItemReader`
- `IbatisPagingItemReader`

9.6.2 Repeat

~ Repetitive actions are common batch processing concern

```
public interface RepeatOperations {
    RepeatStatus iterate(RepeatCallback callback) throws RepeatException;
}
```

~ Callback can contain bussiness logic that will be repeated

```
public interface RepeatCallback {
    RepeatStatus doInIteration(RepeatContext context) throws Exception;
}
```

~ Return value

- Based on this value is decided if iteration should end
- `RepeatStatus.CONTINUABLE` – iteration should continue
- `RepeatStatus.FINISHED` – iteration should finish

~ Simplest implementation of `RepeatOperations` is `RepeatTemplate`

~ Spring Batch can iterate over the input

- `RepeatTemplate` with `RepeatCallback` are used
- Chunked step uses `RepeatCallback` implementation to call `ItemReader.read()`
- So user doesn't have to interact with `RepeatTemplate` directly
- Iteration ends when `ItemReader.read()` returns `null`

9.6.3 Scaling and parallel processing

~ First check if simplest implementation meets your needs first

~ Use concurrent processing only if necessary

~ Two modes of parallel processing

- Single process, multi-threaded
  - Multi-threaded `Step`
  - Parallel `Step`s
  - Partitioning a `Step`
- Multi-process
  - Remote Chunking of `Step`
  - Partitioning a `Step`

~ Multi-threaded `Step`

- Add a `TaskExecutor` to your `Step` configuration

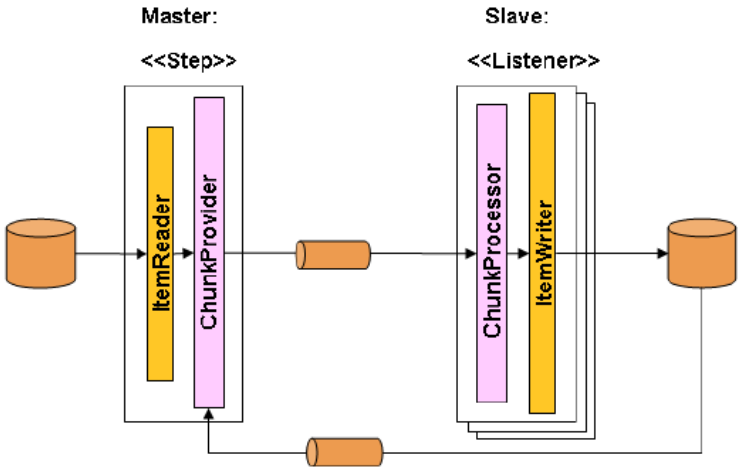
- `Step` parts has to be stateless or thread-safe
- A lot of out of the box implementations of `ItemReader`s/`ItemWriter`s aren't thread-safe
- There is a throttle limit in the tasklet configuration which defaults to 4
- May need to increase this to ensure that a thread pool is fully utilised

```
<step id="loading">
      <tasklet task-executor="taskExecutor" throttle-limit="20">...</tasklet>
</step>
```

~ Parallel `Step`s

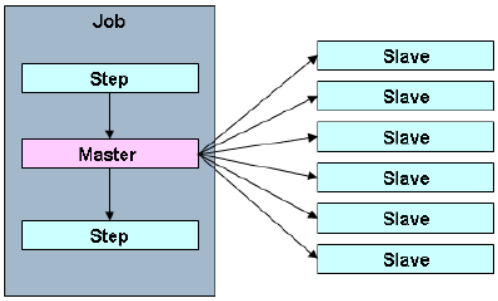
```
<job id="job1">
      <split id="split1" task-executor="taskExecutor" next="step4">
            <flow>
                  <step id="step1" parent="s1" next="step2" />
                  <step id="step2" parent="s2" />
            </flow>
            <flow>
                  <step id="step3" parent="s3" />
            </flow>
      </split>
      <step id="step4" parent="s4" />
</job>
```

~ Remote Chunking of `Step`

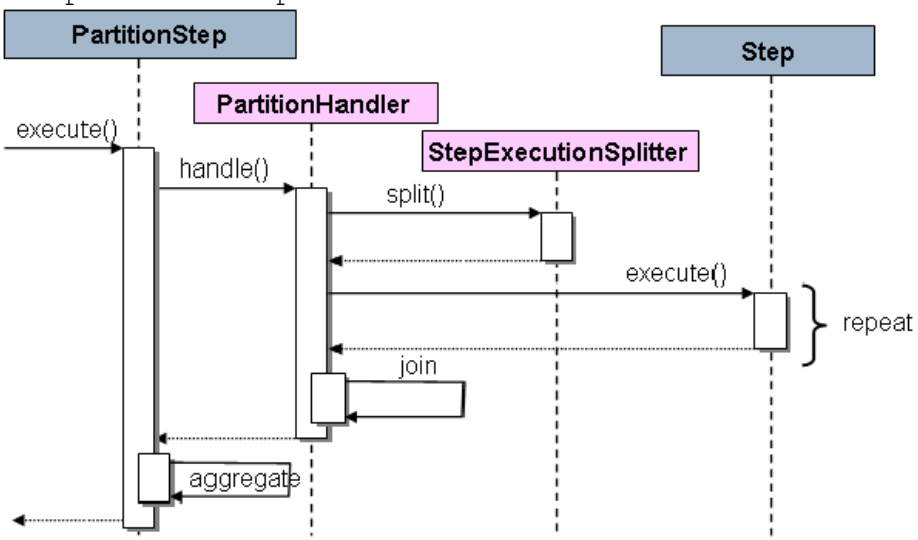


- Step processing is split across multiple processes
- Communicating with each other through some middleware
- Master component is a single process
- Slaves are multiple remote processes
- Works best if the Master is not a bottleneck – processing must be more expensive than the reading of items
- Spring Batch has a sister project Spring Batch Admin, which provides(amongst other things) implementations of various patterns like this one using Spring Integration. These are implemented in a module called Spring Batch Integration

~ Partitioning a `Step`



- Slaves can be
  - remote services (Multi-process scenario)
  - local threads of execution (Single process, multi-threaded scenario)
- Each `PatritionStep`
  - has it's own `ExecutionContext` instance
  - will run only once
- SPI (Service provider interface – API intended to be implemented or extended by a third party) in Spring Batch consists of
  - Special implementation of `Step` (`PartitionStep`)
  - Two strategy interfaces that need to be implemented for the specific environment
    - `PartitionHandler`
    - `StepExecutionSplitter`




```
<step id="step1.master">
      <partition step="step1" partitioner="partitioner">
            <handler grid-size="10" task-executor="taskExecutor" />
      </partition>
</step>
```

  - Spring Batch creates step executions for the partitions called "step1:partition0", etc.
  - `Partitioner` – generates execution contexts as input parameters for new step executions only (no need to worry about restarts)

```
public interface Partitioner {
      Map<String, ExecutionContext> partition(int gridSize);
}
```