Spring Integration Specialist Study Notes

Voici un guide de révision permettant de préparer la **certification Spring Integration Specialist**.
Les fiches le composant reprennent une à une toutes les questions abordées dans l'***Enterprise Integration with Spring 1.x Certification Study Guide*** mis à disposition par SpringSource. J'ai essayé d'y répondre en m'appuyant sur le support de formation, les différents manuels de référence de Spring et le code source.
Au regard de la formation, lorsqu'il m'a semblé que des points importants n'avaient pas été abordés, j'ai ajouté des questions/réponses repérables par *leur police en italique et de couleur bleue*.

Topic	Subject	Question	Réponse
Remoting	Généralités	Les concepts proposés par Spring Remoting, côté serveur comme client	Remoting => appel synchrone de méthodes distantes Côté serveur : concept d'Exporter permettant d'exposer à distance un bean Spring (POJO). Côté client : un ProxyFactoryBean chargé de créer dynamiquement un proxy masquant les appels distants et gérant la plomberie technique (connexion, exceptions …)
		Les bénéfices de Spring Remoting par rapport aux techniques traditionnelles d'appel de méthodes distantes.	1. Faible couplage avec la technologie d'accès à distance : les exceptions vérifiées sont encapsulées par Spring, 2. Il n'est plus nécessaire d'étendre d'interfaces techniques (ex : Remote), 3. Les services métiers peuvent directement être exposés sans modification du code, 4. Spring s'occupe de l'enregistrement dans le registre (RMI) ou l'exposition du service en tant qu'endpoint (http). 5. Aucun couplage avec la technologie de remoting utilisée. 6. Changement de protocole possible par simple changement de configuration
		Les protocoles de Remoting supportés par Spring	• RMI(-IIOP) • HTTPInvoker : protocole d'échange propriétaire à Spring permettant de sérialiser des objets java sur HTTP • Hessian : protocole binaire basé sur HTTP conçu par Caucho • Burlap : alternative XML à Hessian)

			• JAX-RPC : remplacé par JAX-RS depuis Java EE 5 / Java 6 • EJB sans état
	RMI-based Spring Remoting	En quoi le remoting RMI avec Spring est moins invasif que le RMI natif ?	Côté serveur : - Exposition de services POJO (RmiServiceExporter) - Les interfaces des services exposés n'ont plus à étendre java.rmi.Remote - le binding dans le registre RMI est effectué automatiquement par Spring Côté client : - Spring convertit les exceptions vérifiées java.rmi.RemoteException en exceptions non vérifiées (runtime) de type RemoteAccessException - Plus besoin de stub RMI : Spring génère dynamiquement le proxy (RmiProxyFactoryBean). Attention : les classes échangées doivent toujours implémenter l'interface Serializable
	Spring HTTP Invoker	Comment le client et le serveur interagisse l'un avec l'autre ?	Protocole propriétaire utilisant la sérialisation Java pour les paramètres en entrée et en sortie. L'invocation de méthodes est réalisée à l'aide d'un POST HTTP. Utilisation au choix de l'API du JDK ou d'Apache Commons HttpClient.
		Comment exposer un service en HTTP ?	1. Déclarer le bean spring du service à exposer 2. Exporter le bean en utilisant le HttpInvokerServiceExporter <bean name="/monservice" class="o.s.r.h. HttpInvokerServiceExporter …> 3. Utiliser la DispatcherServlet de Spring MVC ou déclarer dans le web.xml une servlet par service exporté
Spring Web Service	Généralités	Différences entre les Web Services et le Remoting ou le Messaging	Couplage technologique lâche. On définit le contrat de service Document-Oriented entre les consommateurs et le fournisseur de service. Basés sur du XML, les web services sont interopérables avec d'autres plateformes que Java : .NET, C++, Ruby, PHP …
	Spring Web Services	L'approche supportée par Spring-WS pour construire des web services	Spring WS permet d'utiliser uniquement l'approche contract-first. Nécessite de commencer par écrire la XSD ou le WSDL (plutôt que d'annoter des méthodes). Le schéma décrit les messages échangés dans le corps de la requête SOAP. Possibilité d'utiliser des outils pour générer une XSD à partir de messages XML d'exemple, XSD qu'il est souvent nécessaire de retoucher, là encore à l'aide d'outils (Trang, XML Spy)

		Spring WS est à même de générer dynamiquement le WSDL à partir de la XSD.
	Les frameworks Object-to-XML supportés par Spring OXM	Pour manipuler les requêtes SOAP, Spring WS propose plusieurs techniques : • Bas niveau en parsant le XML à l'aide d'API XML : JDOM, XOM, Dom4J, TrAX, W3C DOM) • En utilisant le marshalling Object / XML (OXM) • Par binding XPath Spring OXM supporte : JAXB 1 et 2 (standard Java), Castor XML, XMLBeans, XStream, JiBX. Le marshaller peut être déclaré manuellement : <oxm:jaxb2-marshaller id="marshaller" contextPath="com.myapp.ws.dto"/> Spring WS permet de déclarer tous les beans d'infrastructure (y compris le marshaller JAXB2) par la balise : <ws:annotation-driven />
	Les différentes stratégies supportées pour mapper une requête à un Endpoint	Dans le jargon Spring WS, un Endpoint correspond au code métier traitant les messages SOAP. Côté serveur, le point d'entrée d'une requête SOAP est le MessageDispatcher. Celui-ci fait appel au EndpointMapping pour déterminer quel Endpoint doit être invoqué. Indirect, l'appel au Endpoint passe par un Endpoint Adapter qui assure, par exemple, l'unmarshalling XML. Pour déterminer quelle méthode de quel Endpoint invoquer, Spring WS peut utiliser plusieurs stratégies : • Nom de la balise racine du corps du message SOAP (Payload) • Balise action définie dans l'en-tête SOAP • WS-Adressing (basé sur les en-têtes SOAP Action, ReplyTo et To) • X-Path
	De ces stratégies, comme fonctionne précisément le @PayloadRoot ?	L'annotation **@PayloadRoot** permet de mapper la balise racine du corps de la requête SOAP (le Payload) sur une méthode d'un bean annoté avec @Endpoint. Le nom de la balise racine et son namespace doivent être précisés. Couplée aux annotations @ResponsePayload et @RequestPayload , elle assure l'unmarshalling des paramètres d'entrée et le marshalling XML du paramètre de sortie. Nécessite que le bean PayloadRootAnnotationMethodEndpointMapping soit

		enregistré. @PayloadRoot(localPart="helloRequest", namespace="http://myapp.com/schemas/hello") public @ResponsePayload Hello sayHello(@RequestPayload Personne personne)
	Les fonctionnalités proposées par le WebServiceTemplate	Simplifie l'appel aux web services Facilite l'envoie de requêtes et la réception des réponses Travaille directement avec le payload des messages SOAP Supporte le marshaling / unmarshalling Mécanisme de méthodes de rappel (callback) pour les appels bas niveau (ex : accès aux headers SOAP). Extensible par ajout d'intercepteurs permettant par exemple de valider la réponse SOAP au regard de la XSD Gestion des exceptions assurée par le SoapFautMessageResolver qui encapsule les erreurs dans une SoapFaultClientException. Possibilité de fournir son propre resolver. Permet d'utiliser plusieurs protocoles : HTTP, Mail, JMS, XMPP Exemple d'utilisation : Hello hello = (Hello) webServiceTemplate.marshallSendAndReceive(personne)

Web Services Security	Les implémentations sous-jacentes à WS-Security supportés par Spring-WS	La sécurisation des web services en termes de signature, d'authentification et de chiffrement est implémentée à l'aide d'intercepteurs.
		• XwsSecurityInterceptor basé sur le package de Sun XML and Web Services Security (XWSS) de Sun. Pré-requis : JDK de Sun/Oracle et l'implémentation de référence de SAAJ de Sun. Nécessite un *security policy file* pour opérer.
		• Wss4jSecurityInterceptor pour l'intégration d'Apache WSS4J implémentant les standards :
		  ○ SOAP Message Security 1.0 (OASIS)
		  ○ Username Token profile 1.0
		  ○ X.509 Token Profile 1.0

	Comment les key stores sont supportés par Spring WS pour être utilisés par WS-Security ?	La plupart des opérations de cryptographie nécessite un java.security.KeyStore standard. Le Keystore stocke 3 types d'éléments : 1. Clés privés : utilisée par WS-Security pour signer et déchiffrer

			2. Clés symétriques (ou clé secrète) : client et serveur stockent la même clé. Cette dernière est utilisée à la fois pour chiffrer et déchiffrer. 3. Certificats de confiance (X509). WS-Security les utilise pour valider les certifications, vérifier la signature et le chiffrage. Les différentes classes XWSS de Spring WS référencent un bean keystore pouvant être créé à l'aide de la fabrique KeyStoreFactoryBean. Cette dernière a 2 propriétés : le chemin vers le keystore (ex : classphath:truststore.jks ou keystore.jks) et le mot de passe du keystore. Xwss repose sur un KeyStoreCallbackHandler référençant le bean d'un des keystores (le keystore dépend du type d'opérations). Pour gérer les certificats, WSS4J utilise un keystore dont le fichier est référencé par la classe CryptoFactoryBean.
RESTful services with Spring-MVC	Généralités	Les principes de REST	Style d'architecture basé sur http HTTP est utilisé comme protocole applicatif et non comme simple couche de transport comme avec SOAP. 5 concepts principaux : 1. Des ressources identifiables par leur URI (tout est ressource, ex : une entité métier) 2. Une interface d'accès aux ressources unifiée : seulement quelques verbes (opérations) : a. GET : permet d'accéder en lecture seule à la représentation d'une ressource. Pas d'effets de bord. Mise en cache possible côté client (en-têtes E-Tag ou Last-Modified => code 304 Not Modified) b. HEAD : similaire à un GET, sans corps ni mise en cache possible. c. POST : crée une nouvelle ressource. Opération non idempotente. d. PUT : met à jour une ressource ou créée une ressource identifiée par son URI (ex : PUT /personne/123). Idempotent. e. DELETE : supprime une ressource. Idempotent. 3. Une ressource peut avoir plusieurs représentations (text/html,

		image/png). Utilisation des en-têtes Accept et Content-Type pour indiquer au serveur quelle représentation le client sait interpréter. 4. Une conversation est sans état : le serveur ne maintient pas d'état. Le client peut conserver un état à l'aide de liens http. Architecture scalable. 5. Hypermedia : une ressource contient des liens **Idempotence** : opération maintenant le même état après une ou plusieurs invocations Sécurité possible avec HTTP Basic ou Digest + SSL. Utilisation possible de XML-DSIG et XML-Encryption. REST doit être écarté pour des transactions longues.
Support REST dans Spring-MVC	Spring-MVC est une alternative à JAX-RS, non une implémentation	JAX-RS est un standard : Java API for RESTful Web Services (JSR-311). Jersey et CXF en sont des implémentations. Ces frameworks supportent Spring. Annotations JAX-RS : @Path, @GET, @POST, @Produces, @PathParam JAX-RS 1.0 est cantonné à la partie serveur. Spring MVC apporte un support pour REST différent de JAX-RS : • Utilise les annotations Spring MVC : @RequestMapping, @PathVariable • Permet de définir des templates d'URI : "/client/{id}" comme JAX-RS • Permet de déclarer par annotation le code des réponses HTTP. Exemple d'un code 204 : @ResponseStatus(HttpStatus.NO_CONTENT) • Gère la négociation de contenu • Permet d'interroger côté client des services REST à l'aide du RestTemplate
	L'annotation @RequestMapping, incluant le support des URI template	L'annotation @RequestMapping permet de mapper une requête HTTP sur une classe et une méthode. Quelques propriétés : • value : chemin supportant le style ant (ex: "/myPath/*.do") et pouvant être templatisé avec des {}. La valeur extraite est convertie et passée en paramètre de la méthode (utilisation de @PathVariable). La regex par défaut du template [^\.]* peut être redéfinie (ex : /hotels/{hotel:\d+}) • method : verbe HTTP à mapper • params : paramètres de la requête HTTP à mapper. Exemples : myParam=myValue, myParam=!myValue, myParam ou !myParam

			• headers : en-têtes HTTP à mapper sur cette classe / méthode.
		Les annotations @RequestBody et @ResponseBody	Utilisée avec des POST ou des PUT, @RequestBody annote une méthode. Spring MVC convertit le corps de la requête vers le type du paramètre. Il s'aide de l'en-tête *Content-Type*. Positionnée sur une méthode, @ResponseBody indique à Spring MVC de convertir le paramètre de sortie en fonction de l'en-tête *Accept* de la requête (HTML, XML, JSON)
		Les fonctionnalités proposées par le RestTemplate	Le RestTemplate permet de faire appel à des services RESTful. • Supporte les templates d'URI • Détection automatique des frameworks disponibles dans le classpath ROME (Atom, RSS), Jackson, JAXB2 pour enregistrer les converters • Accès direct au corps des requêtes et des réponses • Permet d'utiliser une *HttpEntity* modélisant une requête / réponse http avec son corps et ses en-têtes Quelques méthodes : • getForObject(String url, Class<T> responseType, String urlVariables) • delete(String url, String… urlVariables) • postForLocation(String url, Object request, String… urlVariables) • put(String url, Object request, String… urlVariables) Le RestTemplate peut être configuré pour s'appuyer sur Apache Commons HTTP. Pour l'utiliser, il suffit de l'instancier : new RestTemplate() ou de le déclarer en tant que bean Spring.
JMS avec Spring	Généralités	De quelle manière les applications basées sur Spring JMS peuvent-elles récupérer leurs ressources JMS ?	2 cas de figures : le serveur d'application fournie les ressources JMS (ConnectionFactory et Queue) ou l'application s'interface directement avec le provider JMS. Spring peut donc accéder aux ressources JMS de 2 manières : 1. En les récupérant auprès du conteneur Java EE via un lookup JNDI <jee:jndi-lookup id="connectionFactory" jndi-name="jms/QueueCF"/>

		2. Par déclaration d'un provider JMS standalone : <bean id="connectionFactory" class="o.a.a.ActiveMQConnectionFactory"> <property name="brokerURL" value="tcp://localhost:4444"/> </bean>
	Les fonctionnalités offertes par le conteneur de listeners JMS de Spring, incluant l'utilisation du MessageListenerAdapter à travers l'attribut 'method' de l'élément <jms:listener/>	Alternative aux MDB nécessitant un conteneur EJB, les conteneurs de listeners JMS de Spring permettent de recevoir des messages de manière asynchrone. 2 conteneurs sont proposés : 1. SimpleMessageListenerContainer : approche bas niveau car utilisation requise de l'API JMS. Nombre fixe de sessions JMS. 2. **DefaultMessageListenerContainter** : support des transactions, scaling dynamique, support des workmanagers Le namespace jms simplifie la déclaration d'un conteneur contenant une liste de listeners JMS : <jms:listener-container connection-fatory="jmsCF"> <jms:listener destination="queue.order" ref="orderListener"/> </jms:listener-container> Dans cet exemple, le bean orderListener doit implémenter MessageListener ou SessionAwareMessageListener. Il est possible de filtrer les messages en utilisant un sélecteur JMS. L'élément <jms:listener-container> est hautement paramétrable : task-executor, message-converter, concurrency, transaction-manager, acknowledge, cache … Transparente lors de l'utilisation du namespace jms, la classe **MessageListenerAdapter** permet de déclarer n'importe qu'elle classe en Message Driven POJO (MDP) : <jms:listener ref="orderService" method="placeOrder"/> Un MessageConverter est chargé de convertir le message JMS en paramètre d'entrée de la méthode placeOrder. Lorsque la méthode du MDP retourne un paramètre, il est possible de configurer le listener avec l'attribut response-destination pour le convertir en un message JMS et le déposer dans une file de réponse.

	Les fonctionnalités proposées par le JmsTemplate	Le **JmsTemplate** simplifie l'utilisation de l'API JMS 1 : • Diminution du code technique redondant • Gestion robuste des erreurs • Encapsulation des JMSException explicites dans des exceptions non vérifiées (runtime) • Gestion transparente des ressources JMS • Réutilisation des sessions et des connexions JMS à l'aide du CachingConnectionFactory faisant office de pool (à configurer) • Conversion implicite d'objets en message (ex : String en TextMessage) et possibilité de définir son propre MessageConverter, par exemple pour assurer le marshalling XML • Sélection dynamique de la destination sur laquelle émettre le message. Utilisation par défaut du DynamicDestinationResolver, et possibilité d'implémenter JndiDestinationResolver • Réception synchrone de message (méthodes receiveXXX()). Appel bloquant avec possibilité de spécifier un timeout. • Envoi simplifié de messages (méthodes convertAndSend(destination, message)). Utilisation de callbacks pour une utilisation avancée nécessitant de manipuler l'API JMS : MessagePostProcessor, MessageCreator, ProducerCallback, SessionCallback. jmsTemplate.execute(new SessionCallback() { public Object doInJms(Session session) throws JMSException { …. } }

Transactions	Transactions JMS Locales avec Spring	Comment activer les transactions JMS locale lors de l'utilisation du conteneur de listeners JMS de Spring ?	Le conteneur de listener JMS de Spring permet d'utiliser l'un des 3 modes d'acquittement de JMS ou bien une transaction locale. <jms:listener-container acknowledge="transacted"> • **transacted** : transaction s'appliquant uniquement à la ressource JMS. La transaction démarre lorsque le message est reçu. • **auto** : dès la réception du message JMS, ce dernier est acquitté (retiré) de la file. En cas d'erreur de traitement, le broker JMS est donc dans l'incapacité redélivrer le message. Opération unitaire pouvant être plus lente que d'autres modes. • **client** : l'application cliente est responsable de l'acquittement du message JMS. L'appel de la méthode Message.acknowledge() peut être effectué après le traitement dans la même Session JMS de plusieurs messages. En cas d'erreur, on demande au broker de redélivrer les messages par l'appel de Message.recover(). Le client peut donc être amené à traiter des doublons. • **dups_ok** : le drivers JMS assure l'acquittement des messages en mode lazy, ce qui lui permet des optimisations. Le système doit être à même de savoir traiter des messages JMS dupliqués.
		Comment une transaction JMS locale est-elle rendue disponible au JmsTemplate ?	Utilisée dans un **listener en mode "transacted"**, le JmsTemplate utilise la même session JMS et participe donc à la **transaction initiée par le listener**. Lors de la déclaration d'un JmsTemplate, il est possible de spécifier le mode transactionnel par défaut via les propriétés sessionTransacted et sessionAcnowledgeMode. Ces paramètres sont ignorés lorsqu'une Session JMS active est déjà en cours. En interne, JmsTemplate fait appel à ma méthode ConnectionFactoryUtils.doGetTransactionalSession(…) pour réutiliser la session en cours.
		Comment Spring cherche à synchroniser une transaction JMS locale et une transaction base de données locale ?	Spring applique la stratégie dite du **"best effort"** : 1. Les commits base de données précèdent les commits JMS a. Permet de ne perdre aucun message b. Provoque des messages dupliqués lorsque le commit JMS échoue

		2. Système de synchronisation rapprochant le plus possible les commits bases de données et JMS Seules les transactions distribuées XA assurent une synchronisation 100% garantie.
	La fonctionnalité offerte par le JmsTransactionManager	Le **JmsTransactionManager** attache au thread courant une paire de Connection/Session JMS récupérée de la ConnectionFactory. Le JmsTemplate auto-détecte une Session attachée au thread et y participe automatiquement. Le JmsTransationManager permet d'utiliser une CachingConnectionFactory qui utilise une unique Connection JMS pour tous ses accès (gains en performance). Toutes les Sessions appartiennent à la même Connection.
L'usage avec Spring de JTA et des commit à 2 phases	Que garantie JTA contrairement aux transactions locales ?	Plus que JTA, c'est l'utilisation de XA qui : • Garantie l'ACIDité des transactions distribuées / globales sur plusieurs ressources • Coordonne les commits sur plusieurs ressources • Ecarte tout traitement de messages dupliqués : les messages sont délivrés une et une seule fois.
	Comment basculer d'une transaction locale à une transaction JTA globale ?	La bascule se fait par re-configuration. Le code ne change pas. Nécessite de remplacer le JmsTransactionManager par le JtaTransactionManager (ou l'une de ses classes filles spécifiques aux serveurs d'applis). Toutes les 2 héritent de PlatformTransationManager. JtaTransactionManager n'implémente pas JTA mais permet d'intégrer un gestionnaire de transaction JTA tiers. L'utilisation du tag <tx:jta-transaction-manager /> simplifie encore la déclaration du gestionnaire de transaction JTA. Le conteneur de listeners JMS doit être configuré avec un transaction manager JTA : <jms:listener-container transaction-manager="jtaTransactionManager"/> Des frameworks tiers comme Hibernate doivent être configurés spécifiquement pour JTA.
	D'où peut-on récupérer un gestionnaire de transaction JTA ?	Lorsque l'application est déployée dans un serveur Java EE, Spring récupère le gestionnaire de transaction JTA du serveur par un lookup JNDI. La déclaration <tx:jta-transaction-manager /> indique à Spring de détecter le serveur d'application et de créer le bean spring transationManager avec le

			meilleur gestionnaire de transaction (le plus spécifique au serveur d'application). Chaque ressource transactionnelle XA (dataSource, connectionFactory JMS) peut être récupérée par un <jee:jndi-lookup … /> Pour les applications stand-alone, nécessité de définir manuellement un bean transactionManager et de spécifier ses 2 propriétés transactionManager et userTransaction à l'aide d'implémentation JTA comme Atomikos.
Traitements par lots avec Spring Batch	Généralités	Les principaux concepts : Job, Step, Job Instance, Job Execution, Step Execution …	• Job : entité encapsulant l'ensemble des traitements d'un batch • Step : un batch est composé d'étapes successives • Job Instance = Job + Job Parameters : exécution logique d'un Job. Peut • être redémarré après une erreur • Job Execution : tentative physique d'exécution d'un Job Instance • Step Execution : tentative physique d'exécution d'une étape • JobLauncher : interface permettant de lancer un Job avec un ensemble de Job Parameters.
		Les interfaces typiquement utilisées pour implémenter des Step chunk-oriented	Généralement, un traitement par morceaux (chunk) s'appuie sur un ItemReader, un ItemProcessor (optionnel) et un ItemWriter. Spring Batch met à disposition plusieurs implémentations prêtes à l'emploi par simple configuration XML : • JDBC (curseur et pagination), JPA, iBatis, Hibernate, Procédure Stockée • Fichiers plats (CSV ou à taille fixe) • Fichiers XML (basé sur StAX) • JMS
		Comment et où les états peuvent-ils être persistés ?	L'interface JobRepository offre un mécanisme de persistance proposant des opérations CRUD pour le JobLauncher, les Job et les Step. Persiste le statut de l'exécution des jobs. Spring Batch propose 2 implémentations : mémoire (Map) ou base relationnelle. Le namespace batch permet de déclarer un repository : <batch:job-repository id="jobRepository" /> Le code applicatif et les readers / writers statefull peuvent persister des données à l'aide de l'ExecutionContext. Utile pour le monitoring, la reprise sur erreur et le

		passage d'états d'une étape à l'autre. Le Job ExecutionContext est commité à la fin de chaque Step. Le Step ExecutionContext est commité à la fin de chaque chunk Le listener StepExecutionListener et l'annotation @BeforeStep permettent d'accéder au context d'exécution et de lire / écrire des données : int position = executionContext.getInt("position", 0) ; executionContext.put("position", position) ; Destinée aux ItemReader et ItemWriter, l'interface ItemStream définie un contrat permettant de sauvegarder / restaurer des états en cas d'erreur. Les méhodes open() et update() prennent en paramètre un ExecutionContext
	Qu'est-ce qu'un paramètre de job et comment sont-ils utilisés ?	Paramètres passés pour exécuter un job (ex : nom d'un fichier, date du jour). Uniques pour chaque Job Instance (exception JobInstanceAlreadyCompletedException) Pour que chaque Job Instance soit unique, possibilité d'utiliser le JobParametersIncrementer. Lors de l'exécution d'un batch à l'aide du CommandLineJobRunner, il est possible de passer les paramètres en utilisant la syntaxe key(type)=value avec type = string, date ou long (ex : schedule.date(date)=2012/07/26 Au sein de bean de portée scope, une SpEL peut être utilisée pour accéder à la valeur d'un paramètre (ex : #{jobParameters['input.file.name']})
	Qu'est-ce qu'un FieldSetMapper et à quoi servent-ils ?	Le FieldSetMapper est utilisé par le FlatFileItemReader : il permet de mapper une ligne d'un fichier plat dans un objet du domaine. Le reader commence par décomposer une ligne en token, puis il fait appel à la méthode T mapFieldSet(FieldSet fieldSet) throws BindException; de FieldSetMapper<T> pour parser les données. Un FieldSet est l'équivalant du ResultSet JDBC. Il permet d'accéder aux champs d'une ligne d'un fichier plat par leurs noms ou leurs indexes, et cela de manière fortement typé (ex : int readInt(String name)) Spring Batch fournie 2 implémentations de FieldSetMapper :

			• PassThroughFieldSetMapper : renvoie tel quel le FieldSet • BeanWrapperFieldSetMapper : utilise l'introspection en se basant sur le nom des propriétés du bean et le nom des champs du FieldSet.
Spring Integration (SI)	Généralités	Les principaux concepts (Messages, Channels, Endpoint types) Faites particulièrement attention aux différents types de Endpoints et de quelle manière ils sont utilisés.	Permet la mise en œuvre d'une architecture orientée évènement (API Message) Encourage le faible couplage et la séparation des préoccupations (ex : parsing vs traitement métier) Caractéristiques principales d'un **Message** : - possède un corps (payload) et des en-têtes (MessageHeaders) optionnelles - est immuable - possède un identifiant unique Les **Endpoints** connectent le code applicatif au système de messages de Spring Integration, et cela de manière non invasive. Les **Channels** connectent les Endpoints. Ils participent au faible couplage. Par défaut, un Channel est en mémoire (simple bean), mais possibilité de les faire persister via JMS ou JDBC. <u>Différents types de Endpoints</u> : 1. **Filter** : décide de faire passer ou non un message vers le output channel. Par défaut, un message filtré est supprimé. Autre configuration possible : levée d'une exception (attribut throw-exception-on-rejection) ou message routé dans un discard-channel. Exemple d'utilisation : lorsque plusieurs consommateurs sont abonnés à un pub-sub channel, ce endpoint permet de filtrer les messages à traiter en fonction de critères bien précis. <filter input-channel="input" output-channel="output" ref="filterBean" method="filter" /> 2. **Router** : décide dynamiquement vers quel(s) channel(s) un message doit

être envoyé. La décision est généralement fonction des en-têtes ou du contenu. Une SpEL peut également être utilisée.
Quelques implémentations sont disponibles : RecipientListRouter, HeaderValueRouter

<router input-channel="input" ref="routerBean" method="route" />

3. **Splitter** : découpe un message en plusieurs messages. Typiquement, cela permet de segmenter le traitement d'un payload « composite ».
Afin de pouvoir être regroupés, le slipper spécifie dans les en-têtes : CORRELATION_ID (par défaut à partir du MESSAGE_ID), SEQUENCE_SIZE et SEQUENCE_NUMBER

<splitter input-channel="input" output-channel="output" ref="splitterBean" method="split" />

4. **Agregator** : recompose en un seul message plusieurs messages. Un agrégateur doit être capable d'identifier les messages d'un même lot (Correlation strategy) et de savoir s'il a reçu tous les messages du même lot (Release strategy). Cet endpoint avec état repose sur un MessageStore.
Par défaut, Agregator et Splitter fonctionnent de concert.

<agregator input-channel="input" output-channel="output" ref="agregatorBean" correlation-strategy="correlationBean" release-strategy-expression="#this.size() gt 10"/>

5. **Service Activator** : endpoint générique permettant de connecter un service (métier) au système de messagerie de SI. L'opération d'un service est invoquée pour traiter le message reçu sur l'input-channel. La réponse du service est encapsulée dans un message émis sur le output-channel ou le replyChannel.
Aucun message n'est retourné en réponse de méthodes retournant void

	ou null. Equivaut à utiliser un <outbound-channel-adaptor>. Le flag requires-reply="true" permet de lever une exception. Lors de la déclaration d'un Service Activator, il n'est pas nécessaire de spécifier la méthode à appeler si le service ne contient qu'une seule méthode publique ou si une méthode est annotée avec @ServiceActivator. <service-activator input-channel="input" ref="someService" method="someMethod"/> 6. **Transformer** : déclinaison du Service Activator dédiée à la conversion du payload et/ou à l'enrichissement du payload ou de l'en-ête. Exemples d'utilisation : Object -> JSON, XML → Object, Map → Object <transformer input-channel="input" output-channel="output" ref="transformerBean" method="transform" /> 7. **Channel Adapter** : connecte un Message Channel avec des systèmes ou des services de transports externes. Il y'a 2 types de Channel Adapter : *inbound* pour les messages entrant dans l'application (mails, soap, fichier => messages) et *outbound* pour les messages sortant (messages => mail, jms, rest). Un Channel Adapter est uni-directionnel (one-way). <int-file:inboud-channel-adapter id="filesIn" channel="incomingFiles directory="file:C:/inputFiles" /> <int-jdbc:outbound-channel-adapter query="insert into event (id, name) values (:headers[id], :payload[name]) data-source="dataSource" channel="input" />
Quel usage fait-on des Gateway ?	Le principal but d'une Gateway est de masquer l'API de messaging fournie par SI. Le code applicatif travaille alors uniquement avec des interfaces. La Gateway fait office de proxy. Une *inbound* Gateway fait rentrer des messages dans l'application et attend la réponse. Une *outbound* Gateway fait appel à un système externe et renvoie la réponse dans l'application (sous forme de Message).

		<int:gateway id="cafeService" service-interface="org.cafetaria.ICafeService" default-request-channel="request" default-reply-channel="reply" /> L'attribut *default-reply-channel* est facultatif. SI crée alors un Channel temporaire à usage unique.
	Comment créer de nouveaux Messages par programmation ?	La classe **MessageBuilder** peut être utilisée pour créer des messages par programmation : Message<String> msg = MessageBuilder.withPayload("test").setHeader("foo", "bar").build(); On peut également faire appel au constructeur du message par un **new GenericMessage**(payload, headers); Une fois instancié, un message est immuable. Chaque message possède un identifiant unique (UUID.randomUUID()) L'en-tête est une simple Map<String, Object>
	Utilisation des Chains et des Bridges	Les **Chains** permettent d'alléger la configuration d'endpoints travaillant les uns à la suite des autres (message en sortie de l'un = message en entrée du suivant). Le Chain spécifie un input-channel et un output-channel (optionnel) et tous les endpoints déclarés à l'intérieur n'ont plus besoin de se soucier sur quel Channel travailler. <chain input-channel="input" output-channel="output"> <filter ref="someSelector"/> <header-enricher> <header name="foo" value="bar"/> </header-enricher> <service-activator ref="someService" method="someMethod"/> </ chain> Tous les Endpoints chainés le sont avec des DirectChannels. Lorsque le dernier Endpoint retourne une valeur, un output-channel ou un replyChannel doivent être spécifiés. Les **Bridges** permettent de connecter 2 Message Channels ou 2 Channel Adapters. Ils permettent par exemple de connecter un PollableChannel vers un SubscribableChannel . L'utilisation d'un poller permet de cadencer les messages.

		<bridge input-channel="input" output-channel="output"> <poller max-messages-per-poll="5" fixed-rate="200"/> </bridge>
	Les intercepteurs de Channel et le pattern Wire Tap comme exemple d'utilisation	Des **intercepteurs** peuvent être positionnés individuellement sur chaque Channel ou de manière globale (utilisation possible d'un pattern pour sélectionner les channels sur lesquels il s'applique). Implémenté à l'aide d'un intercepteur, le pattern EIP **Wire Tap** permet de recopier dans un autre Channel les messages déposés dans le Channel intercepté. Particulièrement utile pour le debuggage et le monitoring, il est souvent utilisé conjointement avec le *logging channel adapte*r. SI permet de configurer un Wire Tap de manière transverse (globale).
Traitement des messages synchrones vs asynchrones	Les différents types de Channel et comment chacun doit être utilisé	Deux grandes familles de Channel : **1 Point-to-Point : un seul consommateur** 1.a) DirectChannel : envoi bloquant, synchrone, réception dans le même thread. Possibilité d'avoir plusieurs abonnés en mode failover ou load-balancer (configuration d'un dispatcher). 1.b) ExecutorChannel : envoi non bloquant, asynchrone, les messages sont consommés par un seul thread 1.c) QueueChannel : envoi non bloquant, asynchrone, file FIFO, réception dans un (ou plusieurs) thread(s) séparé(s) à base d'un mécanisme de polling (implémente PollableChannel). File d'attentes spécialisées : PriorityChannel (header priority), Rendezvous Channel **2. Publish-Subscribe : plusieurs consommateurs** (similitudes aux topics JMS) 2.a) Appels séquentiels dans le même thread (synchrone) 2.b) Ou utilisation possible d'un TaskExecutor pour paralléliser les notifications (asynchrone)
	Les effets bords possibles, par exemple sur les transactions et la sécurité	L'ajout d'un Executor sur un DirectChannel ou un PublishSubscribeChannel ajoute de l'asynchronisme. Synchrone => assimilé à des appels de méthodes : - Contextes transactionnels et de sécurité disponibles (ThreadLocal) - Les exceptions sont retournées naturellement à l'appelant (mais wrappées)

		- Faible overhead - Pas scalable Asynchrone : - Les récepteurs reçoivent le message dans un autre thread - Les contextes de sécurité et de transaction sont perdus - Les exceptions ne sont pas systématiquement propagées à l'appelant - Délai de traitement du message inconnu. Permet néanmoins de se mettre en attente du message de réponse Une ligne de configuration permet de passer du mode synchrone au mode asynchrone : 1. <queue capacity="5"/> sur les channel 2. Référence à un task-executor sur les publish-subscribe-channel
	Le besoin de polling actif et comment le configurer	Le **polling** est nécessaire pour activer la consommation de messages déposés dans un PollableChannel (les Channels sont passifs) Par défaut, un unique thread assure le polling, mais il est possible d'utiliser un TaskExecutor. On peut définir un poller qui sera utilisé par défaut pour lire les messages déposés dans les PollableChannel: <poller default="true" task-executor="pool" fixed-delay="200" /> Chaque Endpoint peut redéfinir un poller : <service-activator …> <poller task-executor="otherpool" fixed-rate="500" /> </service-activator> Les pollers peuvent être déclarés comme transactionnel afin que le traitement d'un message soit atomique. 2 pré-conditions : 1. Le traitement doit être géré par un seul thread 2. La transaction englobe l'appel à la méthode receive() du PollableChannel <service-activator …> <poller fixed-rate="500"> <transactional /> </poller>

		</service-activator>
Mécanismes avancées	*Dispatching des messages point-à-point*	Les Channel de type point-à-point (ex : DirectChannel) peuvent avoir plusieurs abonnés. Par défaut, SI utilise l'algorithme round-robin pour répartir alternativement les messages entre les différents abonnés. Lorsqu'une exception est remontée, il fait appel au prochain abonné. Ce comportement est configurable : <channel id="input"> <dispatcher failover="true" load-balancer="none"/> </channel> Lorsque le load-balancing est désactivé, les abonnés sont toujours appelés dans le même ordre, ce qui permet d'avoir un premier abonné pour le cas nominal et un second abonné pour le mode dégradé. Le dispatching peut également fonctionner en asynchrone. L'émetteur n'est alors pas bloqué. Par contre, les abonnés sont appelés dans un autre et même thread.
	Support XML ?	XPath (spliting, transforming, routing, header enricher, filtering), XSLT (transformation), OXM (JAXB 1 & 2, JiBX, XMLBeans, XStream, Castor), Filtres de validation