1. EAI and Spring Integration

* Enterprise Application Integration
  + Been around for as long as disparate computing paradigms have
* EAI is integration of services and / or data.
* Enterprise Application Integration platforms have changed names
  + Now known as: an Enterprise Service Bus (ESB)
* Types of EAI:
  + File Transfer
  + Shared Database
  + Remote Procedure Call
  + Messaging

2. Common ESB Capabilities

* Location Transparency
* Transport Conversion
* Message Transformation / Routing / Enhancement
* Security
  + Spring Security Integration’s on the roadmap
* Monitoring and management
  + you might use JMX for Spring Integration
* Process management (BPMs, orchestration)
* Complex Event Processing

3. Applicability of an ESB

* DO NOT USE:
  + The foundation of all SOA architectures
  + They encourage the creation of centralized silos of information and services
  + This is anti-SOA
  + Best used with an eye towards exposing hidden applications and services
  + Not the fastest solution since, by definition, they're a level of indirection
* USE:
  + You'd use an ESB to integrate applications and data that don't naturally fit well together.
  + Reduce architectural spaghetti

4 Java Solutions Use

* Messaging
* RPC
* Integration with Homogeneous Systems
* Integration with Heterogeneous Systems
* Mainframe systems?
* Security
* Flexible routing

5. Spring Integration

* New addition to the Spring Portfolio
* Provides support for SOA, EDA and EAI
  + It went live with a 1.0 late in 2008.
* Provides philosophical consistency with core Spring principles.
* Spring Integration is an API geared towards building ESB - centric solutions, not another name for the Spring remoting APIs.

6. The ESB Landscape

* Traditional EAI solutions from the likes of TIBCO, Axway, WebMethods
* Open source / Java-centric solutions like ServiceMix, PEtALS, OpenESB, JBossESB, Mule (and of course Spring Integration)
* The alternatives: solutions strung together with bailing wire and tape, a veritable Rube Goldberg machine \* Rube Goldberg solution --you’ve never built something like

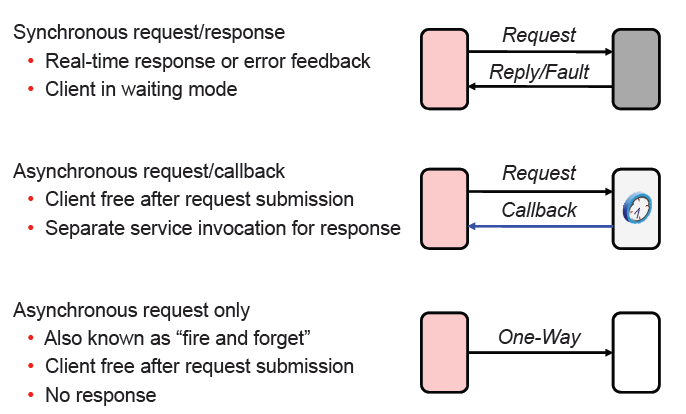
7. Spring Integration

* Uses standard Spring XML
* Idiomatic configuration: annotations, schema
* Spring Integration is embedded: deploy Spring
* Integration with your app - you don’t deploy your application to Spring Integration

8. Spring Integration

* You'll typically deal with three things in an integration solution
  + Channels
  + Endpoints
  + Adapters

9. Synchronous and Asynchronous Interactions



Overview of java.util.concurrent

1. Processes and Threads

* Two basic units of execution: *processes* and *threads*.
* In Java - concurrent programming is mostly concerned with threads, processes are also important.
* It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

2. Processes

* A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
* Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.
* Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a ProcessBuilder object.

3. Threads

* Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.
* Threads exist within a process - every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

4. We know multithreading

* Join
* Synchronization
* ThreadPool
* DeadLock
* Starvation
* LiveLock
  + they are simply too busy responding to each other to resume work.

5. High Level Concurrency Objects

* Lock objects support locking idioms that simplify many concurrent applications.
* Executors define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
* Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
* Atomic variables have features that minimize synchronization and help avoid memory consistency errors.

6. Starting Threads

* Executor framework

public interface Executor {

void execute(Runnable command);

}

7. Executor Example

class MyTask implements Runnable {

public void run() {…}

public static void main(…) {

Runnable task1 = new MyTask();

Executor exec = /\* \*/;

exec.execute(task1);

}

}

8. Thread/ Task

class OnePer implements Executor {

public void Execute(Runnable r){

new Thread(r).start();

}

}

9. Single Threaded

class Just1 implements Executor {

public void Execute(Runnable r){

r.run();

}

}

10. Provided Thread Pool Executors

* newFixedThreadPool
  + Bounded size
  + Replace if thread dies
* newCachedThreadPool
  + Demand driven variable size
* newSingleThreadExecutor
  + Just one thread
  + Replace if thread dies
* newScheduledThreadPool
  + Delayed and periodic task execution
  + Good replacement for class Timer

11. Executor Shut Down

* Executor is a service provider
* Executor abstracts thread management
* To maintain the abstraction, the service should generally provide methods for shutting down the service
* JVM cannot shut down until threads do

12. ExecutorService

public

interface ExecutorService extends Executor {

void shutdown();

List<Runnable> shutdownNow();

boolean isShutdown();

boolean isTerminated();

boolean

awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;

}

13. ExecutorService Notes

* Implies three states:
  + Running
  + Shutting down
  + Terminated
* shutdown() runs tasks in queue
* shutdownNow() returns unstarted tasks
* awaitTermination() blocks until the service is in the terminated state

14. ExecutorService implementation

* ExecutorService is an interface
* How do you implement shut down and cancellation?

15. Cancellation in Java

* *Cooperative*, not mandated
* Traditional method: *interruption*

16. IMPORTANT!

* Your code should follow protocol when interrupted
* If *interrupted()*,
  + Throw exception
  + Reset the flag for other code
* If *InterruptedException*
  + Pass it along
  + Reset the flag for other code
* Don’t swallow, unless your code is handles the interruption policy

17. Restoring Interrupted Status

catch(InterruptedException e) {

Thread.currentThread().interrupt();

}

// checks (AND CLEARS!) current thread

if (Thread.interrupted()) {

Thread.currentThread().interrupt();

}

18. Handling Interruption

* Long computations “break” to check interrupted status
* If interrupted, stop computation, throw *InterruptedException* or restore the interrupted status

19. Interrupting Non-Blocking Operations

* Socket read/writes do not support interruption!
  + If you detect interruption, close the socket causing read/write to throw an exception
* Locks (via synchronized) can not be interrupted
  + Explicit locks beyond scope of this lecture

20. Future

* Interface representing the *lifecycle* of a task

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();

V get() throws InterruptedException,

ExecutionException, CancellationException

V get() throws InterruptedException,

ExecutionException, CancellationException,

TimeoutException

}

21. Higher level concurrency strategies

* Rather than relying on implicitly obtained locks with synchronized code blocks, since JDK1.5 Java now allows use of explicit lock objects.
* Key classes are in java.util.concurrent.locks.
* *ReentrantLock* is key class to focus on.
* Does not enable new capabilities, just simplifies programming model.

22. Basics of *Lock* interface –example

private Lock aLock = new ReentrantLock();

myLock.lock();

try{

// synchronized code here

}

finally {

myLock.unlock();

}

* lock() method blocs until lock is available
* This is same as synchronized code block but attempt to acquire lock

is explicit.

* It is reentrant because a thread can reacquire a lock it already owns.

23. Condition Object

* Need a technique to mimic wait/notify using explicit locks.
* Java provides these as *condition variables.*
* e.g.
  + private Condition sufficientFunds = aLock.newCondition();

Followed by:

* + sufficientFunds.await();
  + sufficientFunds.signalAll();

24. ReadWriteLock

* java.util.concurrent.locks also contains a useful class *ReentrentReadWriteLock.*
* This provides a concurrency model when many threads read a resources but many fewer write to it.
* Those that read do not need synchronized access.

25. Synchronized data structures

* Often possible to use a data structure with built-in synchronization and avoid using locks or synchronized code blocks.
* A good example is a simple blocking queue.
* Blocking queues block threads when they try to add elements beyond capacity, or remove elements from an empty queue.
* See course example ProducerConsumerBlockingQueue for a simple example

26. Asynchronous computation – Callables

* Since JDK 1.5, Java allows the launching of asynchronous tasks that also allow return values.
* This is similar to launching a thread, but with threads the programmer cannot catch exceptions, cannot return values, and needs to use a relatively low level mechanism to coordinate with the thread at termination (e.g. join()).
* The Callable and Future classes bring this one level higher and simplify the programming model

27. Using Callable Objects

* Callable classes implement the Callable interface and are similar to threads:

Callable<T>{

public T call(){

}

}

* Rather than passed to Thread objects, Callable classes are launch using the submit(Callable) method of an ExecutorService, e.g.

ExecutorService pool = Executors.newFixedThreadPool(3);

Future f = pool.submit(myCallable);

And later …

retunVal = f.get(); //a blocking operation!

28. Task Execution and Scheduling

29. Introduction

* The Spring Framework provides abstractions for asynchronous execution and scheduling of tasks with the TaskExecutor and TaskScheduler interfaces, respectively.
* Spring also features implementations of those interfaces that support thread pools or delegation to CommonJ within an application server environment.
* Ultimately the use of these implementations behind the common interfaces abstracts away the differences between Java SE 5, Java SE 6 and Java EE environments.

30. TaskExecutor Types

* SimpleAsyncTaskExecutor
  + does not reuse any threads, rather it starts up a new thread for each invocation. However, it does support a concurrency limit which will block any invocations that are over the limit until a slot has been freed up.
* SyncTaskExecutor
  + doesn't execute invocations asynchronously. Instead, each invocation takes place in the calling thread. It is primarily used in situations where multithreading isn't necessary such as simple test cases.
* ConcurrentTaskExecutor
  + is a wrapper for a Java 5 java.util.concurrent.Executor. There is an alternative, ThreadPoolTaskExecutor, that exposes the Executor configuration parameters as bean properties. It is rare to need to use the ConcurrentTaskExecutor but if the ThreadPoolTaskExecutor isn't robust enough for your needs, the ConcurrentTaskExecutor is an alternative.

31. Task Executor Types

* SimpleThreadPoolTaskExecutor
  + a subclass of Quartz's SimpleThreadPool which listens to Spring's lifecycle callbacks. This is typically used when you have a thread pool that may need to be shared by both Quartz and non-Quartz components.
* ThreadPoolTaskExecutor
  + it is not possible to use any backport or alternate versions of the java.util.concurrent package with this implementation. Both Doug Lea's and Dawid Kurzyniec's implementations use different package structures which will prevent them from working correctly.
* TimerTaskExecutor
  + uses a single TimerTask as its backing implementation. It's different from the SyncTaskExecutor in that the method invocations are executed in a separate thread, although they are synchronous in that thread.

32. TaskScheduler

* For scheduling tasks to run at some point in the future.
  + 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);

33. Annotation Support for Scheduling and Asynchronous Execution

* To enable support for @Scheduled and @Async annotations add @EnableScheduling and @EnableAsync to one of your @Configuration classes:

34. @Scheduled

* Exactly one of the cron(), fixedDelay(), or fixedRate() attributes must be specified.

35. The @Async Annotation

* The @Async annotation can be provided on a method so that invocation of that method will occur asynchronously.
* In other words, the caller will return immediately upon invocation and the actual execution of the method will occur in a task that has been submitted to a Spring TaskExecutor.
* In the simplest case, the annotation may be applied to a void-returning method.

36. Enable Support(*@EnableAsync* )

* ***annotation****– b*y default, *@EnableAsync* detects Spring’s *@Async* annotation and the EJB 3.1 *javax.ejb.Asynchronous*; this option can be used to detect other, user-defined annotation types as well
* ***mode***– indicates the type of *advice* that should be used – JDK proxy-based or AspectJ weaving
* ***proxyTargetClass***– indicates the type of *proxy*that should be used – CGLIB or JDK; this attribute has effect only if the ***mode*** is set to *AdviceMode.PROXY*
* ***order*** – sets the order in which *AsyncAnnotationBeanPostProcessor* should be applied; by default, it runs last, just so that it can take into account all existing proxies

37. *@Async* Annotation

* Must be applied to *public* methods only and Self-invocation – calling the async method from within the same class won’t work (why?)
* Can return
  + Void
  + Future
* Exception Handling
  + *Void -* exceptions will not be propagated to the calling thread.
  + *Future.get() -* method will throw the exception.

RESTful Web Services

Representational State Transfer is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, termed RESTful Web services, provide interoperability between computer systems on the Internet. [Wikipedia](https://en.wikipedia.org/wiki/Representational_state_transfer)

**BackGROUND Information**

**REPRESENTATIONAL STATE TRANSFER**

REST, or REpresentational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. REST-compliant systems, often called RESTful systems, are characterized by how they are stateless and separate the concerns of client and server. We will go into what these terms mean and why they are beneficial characteristics for services on the Web.

**SEPARATION OF CLIENT AND SERVER**

In the REST architectural style, the implementation of the client and the implementation of the server can be done independently without each knowing about the other. This means that the code on the client side can be changed at any time without affecting the operation of the server, and the code on the server side can be changed without affecting the operation of the client.

As long as each side knows what format of messages to send to the other, they can be kept modular and separate. Separating the user interface concerns from the data storage concerns, we improve the flexibility of the interface across platforms and improve scalability by simplifying the server components. Additionally, the separation allows each component the ability to evolve independently.

By using a REST interface, different clients hit the same REST endpoints, perform the same actions, and receive the same responses.

**STATELESSNESS**

Systems that follow the REST paradigm are stateless, meaning that the server does not need to know anything about what state the client is in and vice versa. In this way, both the server and the client can understand any message received, even without seeing previous messages. This constraint of statelessness is enforced through the use of *resources*, rather than *commands*. Resources are the nouns of the Web - they describe any object, document, or *thing* that you may need to store or send to other services.

Because REST systems interact through standard operations on resources, they do not rely on the implementation of interfaces.

These constraints help RESTful applications achieve reliability, quick performance, and scalability, as components that can be managed, updated, and reused without affecting the system as a whole, even during operation of the system.

Now, we’ll explore how the communication between the client and server actually happens when we are implementing a RESTful interface.

**COMMUNICATION BETWEEN CLIENT AND SERVER**

In the REST architecture, clients send requests to retrieve or modify resources, and servers send responses to these requests. Let’s take a look at the standard ways to make requests and send responses.

**MAKING REQUESTS**

REST requires that a client make a request to the server in order to retrieve or modify data on the server. A request generally consists of:

* an HTTP verb, which defines what kind of operation to perform
* a *header*, which allows the client to pass along information about the request
* a path to a resource
* an optional message body containing data

**HTTP VERBS**

There are 4 basic HTTP verbs we use in requests to interact with resources in a REST system:

* GET — retrieve a specific resource (by id) or a collection of resources
* POST — create a new resource
* PUT — update a specific resource (by id)
* DELETE — remove a specific resource by id

You can learn more about these HTTP verbs in the following Codecademy article:

* [What is CRUD?](https://www.codecademy.com/articles/what-is-crud)

**HEADERS AND ACCEPT PARAMETERS**

In the header of the request, the client sends the type of content that it is able to receive from the server. This is called the Accept field, and it ensures that the server does not send data that cannot be understood or processed by the client. The options for types of content are MIME Types (or Multipurpose Internet Mail Extensions, which you can read more about in the [MDN Web Docs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types).

MIME Types, used to specify the content types in the Accept field, consist of a type and a subtype. They are separated by a slash (/).

For example, a text file containing HTML would be specified with the type text/html. If this text file contained CSS instead, it would be specified as text/css. A generic text file would be denoted as text/plain. This default value, text/plain, is not a catch-all, however. If a client is expecting text/css and receives text/plain, it will not be able to recognize the content.

Other types and commonly used subtypes:

* image — image/png, image/jpeg, image/gif
* audio — audio/wav, image/mpeg
* video — video/mp4, video/ogg
* application — application/json, application/pdf, application/xml, application/octet-stream

For example, a client accessing a resource with id 23 in an articlesresource on a server might send a GET request like this:

GET /articles/23

Accept: text/html, application/xhtml

The Accept header field in this case is saying that the client will accept the content in text/html or application/xhtml.

**PATHS**

Requests must contain a path to a resource that the operation should be performed on. In RESTful APIs, paths should be designed to help the client know what is going on.

Conventionally, the first part of the path should be the plural form of the resource. This keeps nested paths simple to read and easy to understand.

A path like fashionboutique.com/customers/223/orders/12 is clear in what it points to, even if you’ve never seen this specific path before, because it is hierarchical and descriptive. We can see that we are accessing the order with id 12 for the customer with id 223.

Paths should contain the information necessary to locate a resource with the degree of specificity needed. When referring to a list or collection of resources, it is unnecessary to add an id to a POST request to a fashionboutique.com/customers path would not need an extra identifier, as the server will generate an id for the new object.

If we are trying to access a single resource, we would need to append an id to the path. For example: GET fashionboutique.com/customers/:id — retrieves the item in the customers resource with the id specified. DELETE fashionboutique.com/customers/:id — deletes the item in the customersresource with the id specified.

**SENDING RESPONSES**

**CONTENT TYPES**

In cases where the server is sending a data payload to the client, the server must include a content-type in the header of the response. This content-type header field alerts the client to the type of data it is sending in the response body. These content types are MIME Types, just as they are in the accept field of the request header. The content-type that the server sends back in the response should be one of the options that the client specified in the accept field of the request.

For example, when a client is accessing a resource with id 23 in an articles resource with this GET Request:

GET /articles/23 HTTP/1.1

Accept: text/html, application/xhtml

The server might send back the content with the response header:

HTTP/1.1 200 (OK)

Content-Type: text/html

This would signify that the content requested is being returning in the response body with a content-type of text/html, which the client said it would be able to accept.

**RESPONSE CODES**

Responses from the server contain status codes to alert the client to information about the success of the operation. As a developer, you do not need to know every status code (there are [many](http://www.restapitutorial.com/httpstatuscodes.html) of them), but you should know the most common ones and how they are used:

| **Status code** | **Meaning** |
| --- | --- |
| 200 (OK) | This is the standard response for successful HTTP requests. |
| 201 (CREATED) | This is the standard response for an HTTP request that resulted in an item being successfully created. |
| 204 (NO CONTENT) | This is the standard response for successful HTTP requests, where nothing is being returned in the response body. |
| 400 (BAD REQUEST) | The request cannot be processed because of bad request syntax, excessive size, or another client error. |
| 403 (FORBIDDEN) | The client does not have permission to access this resource. |
| 404 (NOT FOUND) | The resource could not be found at this time. It is possible it was deleted, or does not exist yet. |
| 500 (INTERNAL SERVER ERROR) | The generic answer for an unexpected failure if there is no more specific information available. |

For each HTTP verb, there are expected status codes a server should return upon success:

* GET — return 200 (OK)
* POST — return 201 (CREATED)
* PUT — return 200 (OK)
* DELETE — return 204 (NO CONTENT) If the operation fails, return the most specific status code possible corresponding to the problem that was encountered.

**EXAMPLES OF REQUESTS AND RESPONSES**

Let's say we have an application that allows you to view, create, edit, and delete customers and orders for a small clothing store hosted at fashionboutique.com. We could create an HTTP API that allows a client to perform these functions:

If we wanted to view all customers, the request would look like this:

GET http://fashionboutique.com/customers

Accept: application/json

A possible response header would look like:

Status Code: 200 (OK)

Content-type: application/json

followed by the customers data requested in application/json format.

Create a new customer by posting the data:

POST http://fashionboutique.com/customers

Body:

{

“customer”: {

“name” = “Scylla Buss”

“email” = “scylla.buss@codecademy.org”

}

}

The server then generates an id for that object and returns it back to the client, with a header like:

201 (CREATED)

Content-type: application/json

To view a single customer we *GET* it by specifying that customer’s id:

GET http://fashionboutique.com/customers/123

Accept: application/json

A possible response header would look like:

Status Code: 200 (OK)

Content-type: application/json

followed by the data for the customer resource with id 23 in application/json format.

We can update that customer by \_PUT\_ting the new data:

PUT http://fashionboutique.com/customers/123

Body:

{

“customer”: {

“name” = “Scylla Buss”

“email” = “scyllabuss1@codecademy.com”

}

}

A possible response header would have Status Code: 200 (OK), to notify the client that the item with id 123 has been modified.

We can also *DELETE* that customer by specifying its id:

DELETE http://fashionboutique.com/customers/123

The response would have a header containing Status Code: 204 (NO CONTENT), notifying the client that the item with id 123 has been deleted, and nothing in the body.

**PRACTICE WITH REST**

Let’s imagine we are building a photo-collection site for a different want to make an API to keep track of users, venues, and photos of those venues. This site has an index.html and a style.css. Each user has a username and a password. Each photo has a venue and an owner (i.e. the user who took the picture). Each venue has a name and street address. Can you design a REST system that would accommodate:

* storing users, photos, and venues
* accessing venues and accessing certain photos of a certain venue

Start by writing out:

* what kinds of requests we would want to make
* what responses the server should return
* what the content-type of each response should be

**POSSIBLE SOLUTION - MODELS**

{

“user”: {

"id": <Integer>,

“username”: <String>,

“password”: <String>

}

}

{

“photo”: {

"id": <Integer>,

“venue\_id”: <Integer>,

“author\_id”: <Integer>

}

}

{

“venue”: {

"id": <Integer>,

“name”: <String>,

“address”: <String>

}

}

**POSSIBLE SOLUTION - REQUESTS/RESPONSES**

**GET REQUESTS**

Request- GET /index.html Accept: text/html Response- 200 (OK) Content-type: text/html

Request- GET /style.css Accept: text/css Response- 200 (OK) Content-type: text/css

Request- GET /venues Accept:application/json Response- 200 (OK) Content-type: application/json

Request- GET /venues/:id Accept: application/json Response- 200 (OK) Content-type: application/json

Request- GET /venues/:id/photos/:id Accept: application/json Response- 200 (OK) Content-type: image/png

**POST REQUESTS**

Request- POST /users Response- 201 (CREATED) Content-type: application/json

Request- POST /venues Response- 201 (CREATED) Content-type: application/json

Request- POST /venues/:id/photos Response- 201 (CREATED) Content-type: application/json

**PUT REQUESTS**

Request- PUT /users/:id Response- 200 (OK)

Request- PUT /venues/:id Response- 200 (OK)

Request- PUT /venues/:id/photos/:id Response- 200 (OK)

**DELETE REQUESTS**

Request- DELETE /venues/:id Response- 204 (NO CONTENT)

Request- DELETE /venues/:id/photos/:id Response- 204 (NO CONTENT)

1. Low Level Client Access with HttpClient

* HttpRequest/HttpResponse
* Doing CRUD Operation using HTTPClient
* SetHeader
* Sending Form Params
* Different Mime Types

2. Abstract Client Access with the RestTemplate

3. CONNEG - Content Negotiation

* You can use the RESTful @ResponseBody approach and HTTP message converters, typically to return data-formats like JSON or XML. Programmatic clients, mobile apps and AJAX enabled browsers are the usual clients.
* Alternatively you may use view resolution. Although views are perfectly capable of generating JSON and XML if you wish, views are normally used to generate presentation formats like HTML for a traditional web-application.

4. Http Header

* There are three situations where we need to know what type of data-format to send in the HTTP response:
  + HttpMessageConverters: Determine the right converter to use.
  + Request Mappings: Map an incoming HTTP request to different methods that return different formats.
  + View Resolution: Pick the right view to use.
* Determining what format the user has requested relies on a ContentNegotationStrategy (either out of the box / custom)

5. Enabling Content Negotiation in Spring MVC

* Supported Options
  + URL suffixes and/or a URL parameter. These work alongside the use of Accept headers. As a result, the content-type can be requested in any of three ways. By default they are checked in this order

1 - Add a path extension (suffix) in the URL.

* + - For url like <http://.../list.html> - HTML is required.
    - For http://../list.xls. The suffix to media-type mapping is automatically defined via the JavaBeans Activation Framework or JAF (so activation.jar must be on the class path).

6. Enabling Content Negotiation in Spring MVC

* 2 - A URL parameter like this: http://myserver/myapp/accounts/list?format=xls. The name of the parameter is format by default, but this may be changed. Using a parameter is disabled by default, but when enabled, it is checked second.
* Finally the Accept HTTP header property is checked. This is how HTTP is actually defined to work, but, as previously mentioned, it can be problematic to use.

7. Java Configuration

@Configuration

@EnableWebMvc

public class WebConfig extends WebMvcConfigurerAdapter {

/\* Setup a simple strategy: use all the defaults and return XML by default when not sure. \*/

@Override

public void configureContentNegotiation (ContentNegotiationConfigurer configurer) {

configurer.defaultContentType(MediaType.APPLICATION\_XML);

}

}

8. Content Negotiation

@Override

public void configureContentNegotiation(

ContentNegotiationConfigurer configurer) {

// Simple strategy: only path extension is taken into account

configurer.favorPathExtension(true).

ignoreAcceptHeader(true).

useJaf(false).

defaultContentType(MediaType.TEXT\_HTML).

mediaType("html", MediaType.TEXT\_HTML).

mediaType("xml", MediaType.APPLICATION\_XML).

mediaType("json", MediaType.APPLICATION\_JSON);

}

9. Combining Data and Presentation Formats

* @RequestMapping(value="/accounts", produces={"application/xml", "application/json"})

10. Exchanging Headers

* In
  + RestTemplate
  + HTTPClient

JMS – Java Messaging Services

Background

## Overview of the JMS API

This overview defines the concept of messaging, describes the JMS API and when it can be used, and explains how the JMS API works within the Java EE platform.

### What Is Messaging?

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is **loosely coupled**. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which message format and which destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application’s methods.

Messaging also differs from electronic mail (email), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

### What Is the JMS API?

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.

The JMS API enables communication that is not only loosely coupled but also:

* **Asynchronous**: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
* **Reliable**: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The current version of the JMS specification is Version 1.1. You can download a copy of the specification from the JMS web site: <http://www.oracle.com/technetwork/java/index-jsp-142945.html>.

### When Can You Use the JMS API?

An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as a remote procedure call (RPC), under the following circumstances.

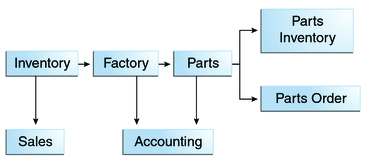
* The provider wants the components not to depend on information about other components’ interfaces, so components can be easily replaced.
* The provider wants the application to run whether or not all components are up and running simultaneously.
* The application business model allows a component to send information to another and to continue to operate without receiving an immediate response.

For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like these:

* The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so the factory can make more cars.
* The factory component can send a message to the parts components so the factory can assemble the parts it needs.
* The parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers.
* Both the factory and the parts components can send messages to the accounting component to update budget numbers.
* The business can publish updated catalog items to its sales force.

Using messaging for these tasks allows the various components to interact with one another efficiently, without tying up network or other resources. [Figure 47-1](https://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html#bncdv) illustrates how this simple example might work.

**Figure 47-1 Messaging in an Enterprise Application**



Manufacturing is only one example of how an enterprise can use the JMS API. Retail applications, financial services applications, health services applications, and many others can make use of messaging.

### How Does the JMS API Work with the Java EE Platform?

When the JMS API was introduced in 1998, its most important purpose was to allow Java applications to access existing messaging-oriented middleware (MOM) systems, such as MQSeries from IBM. Since that time, many vendors have adopted and implemented the JMS API, so a JMS product can now provide a complete messaging capability for an enterprise.

Beginning with the 1.3 release of the Java EE platform, the JMS API has been an integral part of the platform, and application developers have been able to use messaging with Java EE components.

The JMS API in the Java EE platform has the following features.

* Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously. (Applets, however, are not required to support the JMS API.)
* Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages. A JMS provider can optionally implement concurrent processing of messages by message-driven beans.
* Message send and receive operations can participate in distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.

The JMS API enhances the Java EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among Java EE components and legacy systems capable of messaging. A developer can easily add new behavior to a Java EE application that has existing business events by adding a new message-driven bean to operate on specific business events. The Java EE platform, moreover, enhances the JMS API by providing support for distributed transactions and allowing for the concurrent consumption of messages. For more information, see the Enterprise JavaBeans specification, v3.1.

The JMS provider can be integrated with the application server using the Java EE Connector architecture. You access the JMS provider through a resource adapter. This capability allows vendors to create JMS providers that can be plugged in to multiple application servers, and it allows application servers to support multiple JMS providers. For more information, see the Java EE Connector architecture specification, v1.6.

1. Some Definitions

* **Messaging Client**
  + Messaging is peer-to-peer. Every peer is called a client
* **Message Producer**
  + Message Sender is called a producer
* **Message Consumer**
  + Message Receiver is called a consumer
* ***A Client may be a Producer, a Consumer, or both***
* **Durable Message Delivery**
  + Router holds the message until the time client becomes active and receives message
* **Persistent Message Delivery**
  + If Messaging Server goes down before a message is delivered, the delivery is still guaranteed

2. Communication Models

* **Point-To-Point (PTP)**
  + Used if there is one and only one receiver for each message
* **Publish-Subscribe (PS)**
  + Used for general broadcast kind of applications

3. Point-To-Point Features

* Producer sends Messages to the queue
* Consumer receives message from the queue
* Multiple senders may send messages to same queue
* One and only one consumer for every message
* System may provide multiple queues.

4.PS Features

* Producer publishes messages to a topic
* Consumer subscribes to interested topic
* A message may be consumed by multiple subscribers
* Multiple publishers and subscribers to a single topic

5. Message Delivery Model

* **Synchronous**
  + Consumer is blocked until
    - Message is received
    - Timeout set by consumer expires
    - Consumer closes
* **Asynchronous**
  + Messaging system responsible for callback on consumer

6. Message Object

* + Header
    - JMS defined fields
  + Properties
    - User defined Name/Value pairs
  + Body
    - User specified data

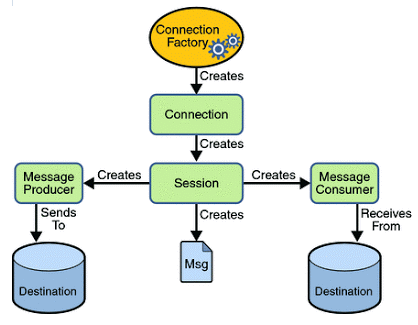
7. Message Types

* TextMessage
  + Represents String

messages

* MapMessage
  + Provides Name/Value pairs
* BytesMessage
  + Stream of bytes
* StreamMessage
  + Series of primitive data types
* ObjectMessage
  + Serialized Java object

8 Messaging Model

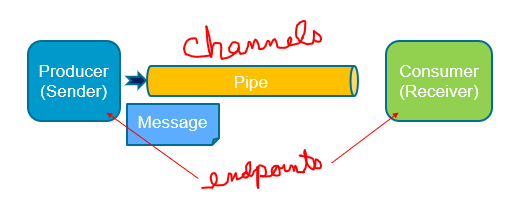


9.

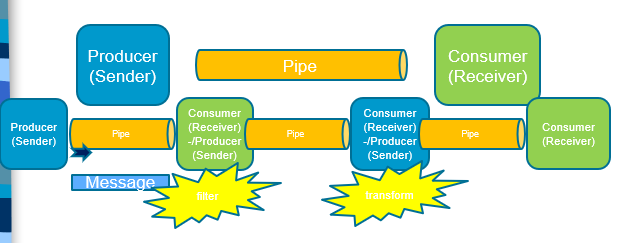
Spring Integration

Background - Refer to gids09pptmaster-090906114703-phpapp02.pdf

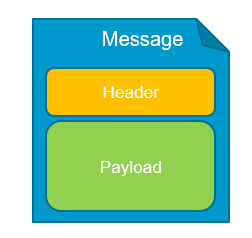
1. Spring Integration Applications



2. Spring Integration Applications



3. Messages



4. Message EndPoints

* Adapters (connect your channel to some other system)
* Filter (remove some messages from channels based on header, content, etc.)
* Transformer (convert a message content or structure)
* Enricher (add content to the message header or payload)
* Service activator (invoke service operations based on the arrival of a message)
* Gateway (connect your channels without SI coupling)

5. Message Channels

* Two general classifications of message channels
  + **Pollable Channel**
  + **Subscribable Channel**
* While there are many subtypes, they all implement at least one of these SI channel interfaces
  + see http://docs.spring.io/spring-integration/reference/html/messaging-channels-section.html

6. Pollable Channels

* May buffer its messages
  + Requires a queue to hold the messages
  + The queue has a designated capacity
* Waits for the consumer to get the messages
  + Consumers actively poll to receive messages
* Typically a point-to-point channel
  + Only one receiver of a message in the channel
* Usually used for sending information or “document” messages between endpoints

7. Subscribable

* Allows multiple subscribers (or consumers) to register for its messages.
  + Messages are delivered to all registered subscribers on message arrival
  + It has to manage a list or registry of subscribers.
* Doesn’t buffer its messages
* Usually used for “event” messages
  + Notifying the subscribers that something happened and to take appropriate action.

8. Channel Adapters (Adapters)

* A Channel Adapter is a Message Endpoint that enables connecting a single sender or receiver to a Message Channel. Spring Integration provides a number of adapters out of the box to support various transports, such as JMS, File, HTTP, Web Services, and Mail.
* The simple but flexible Method-invoking Channel Adapter support. There are both inbound and outbound adapters, and each may be configured with XML elements provided in the core namespace.

9. Channel Types

* PollableChannel
* SubscribableChannel
* PublishSubscribeChannel
* QueueChannel
* PriorityChannel
* DirectChannel
* ExecutorChannel
* Scoped Channel

10. Adapters

* A Channel Adapter is a Message Endpoint that enables connecting a single sender or receiver to a Message Channel. Spring Integration provides a number of adapters out of the box to support various transports, such as JMS(MessageTemplate), File, HTTP, Web Services, Mail, and more.

11. Filters

* <int:filter ref="selector" output-channel="outboundChannel" input-channel="inboundChannel"/>
* <bean id="selector" class="com.intertech.lab3.FileSelector"/>
* public class FileSelector implements MessageSelector {

public boolean accept(Message<?> message) {

if (message.getPayload() instanceof File

&& ((File) message.getPayload()).getName().startsWith("msg")) {

return false;

}

return true;

}

}

12. Splitters and Aggregators

* The Splitter is a component whose role is to partition a message in several parts, and send the resulting messages to be processed independently. Aggregator is a type of Message Handler that receives multiple Messages and combines them into a single Message

Spring Batch

Background

# Spring Batch Introduction

Many applications within the enterprise domain require bulk processing to perform business operations in mission critical environments. These business operations include automated, complex processing of large volumes of information that is most efficiently processed without user interaction. These operations typically include time based events (e.g. month-end calculations, notices or correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g. Insurance benefit determination or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Batch processing is used to process billions of transactions every day for enterprises.

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. Spring Batch builds upon the productivity, POJO-based development approach, and general ease of use capabilities people have come to know from the Spring Framework, while making it easy for developers to access and leverage more advance enterprise services when necessary. Spring Batch is not a scheduling framework. There are many good enterprise schedulers available in both the commercial and open source spaces such as Quartz, Tivoli, Control-M, etc. It is intended to work in conjunction with a scheduler, not replace a scheduler.

Spring Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It also provides more advance technical services and features that will enable extremely high-volume and high performance batch jobs though optimization and partitioning techniques. Simple as well as complex, high-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information.

## 1.1 Background

While open source software projects and associated communities have focused greater attention on web-based and SOA messaging-based architecture frameworks, there has been a notable lack of focus on reusable architecture frameworks to accommodate Java-based batch processing needs, despite continued needs to handle such processing within enterprise IT environments. The lack of a standard, reusable batch architecture has resulted in the proliferation of many one-off, in-house solutions developed within client enterprise IT functions.

SpringSource and Accenture have collaborated to change this. Accenture's hands-on industry and technical experience in implementing batch architectures, SpringSource's depth of technical experience, and Spring's proven programming model together mark a natural and powerful partnership to create high-quality, market relevant software aimed at filling an important gap in enterprise Java. Both companies are also currently working with a number of clients solving similar problems developing Spring-based batch architecture solutions. This has provided some useful additional detail and real-life constraints helping to ensure the solution can be applied to the real-world problems posed by clients. For these reasons and many more, SpringSource and Accenture have teamed to collaborate on the development of Spring Batch.

Accenture has contributed previously proprietary batch processing architecture frameworks, based upon decades worth of experience in building batch architectures with the last several generations of platforms, (i.e., COBOL/Mainframe, C++/Unix, and now Java/anywhere) to the Spring Batch project along with committer resources to drive support, enhancements, and the future roadmap.

The collaborative effort between Accenture and SpringSource aims to promote the standardization of software processing approaches, frameworks, and tools that can be consistently leveraged by enterprise users when creating batch applications. Companies and government agencies desiring to deliver standard, proven solutions to their enterprise IT environments will benefit from Spring Batch.

## 1.2 Usage Scenarios

A typical batch program generally reads a large number of records from a database, file, or queue, processes the data in some fashion, and then writes back data in a modified form. Spring Batch automates this basic batch iteration, providing the capability to process similar transactions as a set, typically in an offline environment without any user interaction. Batch jobs are part of most IT projects and Spring Batch is the only open source framework that provides a robust, enterprise-scale solution.

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.

## 1.3 Spring Batch Architecture

Spring Batch is designed with extensibility and a diverse group of end users in mind. The figure below shows a sketch of the layered architecture that supports the extensibility and ease of use for end-user developers.

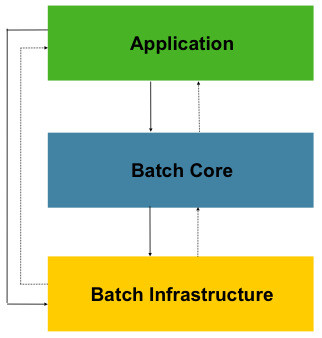


Figure 1.1: Spring Batch Layered Architecture

This layered architecture highlights three major high level components: Application, Core, and Infrastructure. The application contains all batch jobs and custom code written by developers using Spring Batch. The Batch Core contains the core runtime classes necessary to launch and control a batch job. It includes things such as aJobLauncher, Job, and Step implementations. Both Application and Core are built on top of a common infrastructure. This infrastructure contains common readers and writers, and services such as the RetryTemplate, which are used both by application developers(ItemReader and ItemWriter) and the core framework itself. (retry)

## 1.4 General Batch Principles and Guidelines

The following are a number of key principles, guidelines, and general considerations to take into consideration when building a batch solution.

* A batch architecture typically affects on-line architecture and vice versa. Design with both architectures and environments in mind using common building blocks when possible.
* Simplify as much as possible and avoid building complex logical structures in single batch applications.
* Process data as close to where the data physically resides as possible or vice versa (i.e., keep your data where your processing occurs).
* Minimize system resource use, especially I/O. Perform as many operations as possible in internal memory.
* Review application I/O (analyze SQL statements) to ensure that unnecessary physical I/O is avoided. In particular, the following four common flaws need to be looked for:
  + Reading data for every transaction when the data could be read once and kept cached or in the working storage;
  + Rereading data for a transaction where the data was read earlier in the same transaction;
  + Causing unnecessary table or index scans;
  + Not specifying key values in the WHERE clause of an SQL statement.
* Do not do things twice in a batch run. For instance, if you need data summarization for reporting purposes, increment stored totals if possible when data is being initially processed, so your reporting application does not have to reprocess the same data.
* Allocate enough memory at the beginning of a batch application to avoid time-consuming reallocation during the process.
* Always assume the worst with regard to data integrity. Insert adequate checks and record validation to maintain data integrity.
* Implement checksums for internal validation where possible. For example, flat files should have a trailer record telling the total of records in the file and an aggregate of the key fields.
* Plan and execute stress tests as early as possible in a production-like environment with realistic data volumes.
* In large batch systems backups can be challenging, especially if the system is running concurrent with on-line on a 24-7 basis. Database backups are typically well taken care of in the on-line design, but file backups should be considered to be just as important. If the system depends on flat files, file backup procedures should not only be in place and documented, but regularly tested as well.

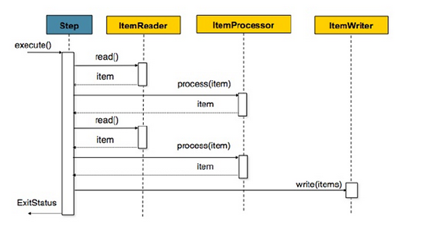
1. Batch Process?

* + Bulk Processessing
  + Long Running Processes
  + Mostly Sequential Processes
  + Routine - onetime, daily, monthly, yearly, ...

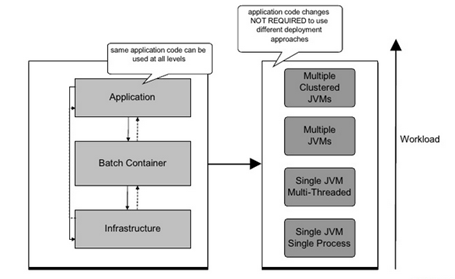
*"The lack of a standard, reusable batch architecture has resulted in the proliferation of many one-off, in-house solutions developed within client enterprise IT functions."*

- spring batch documentation

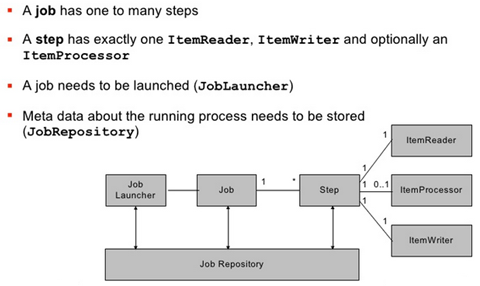
2. Item Oriented Processing



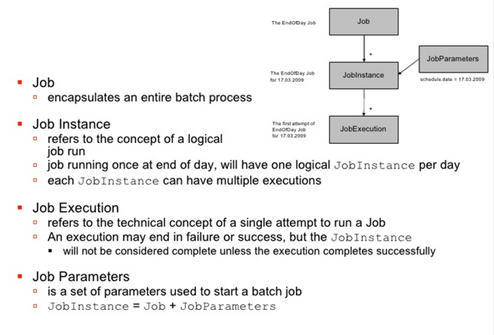
3. Layered Architecture



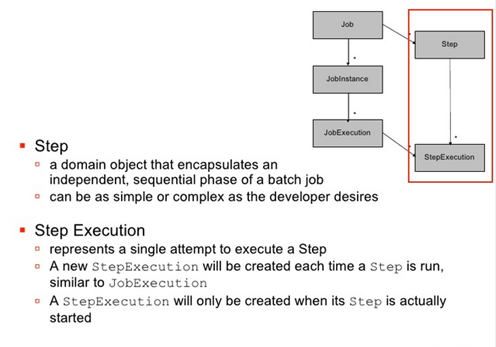
4. Domain Language



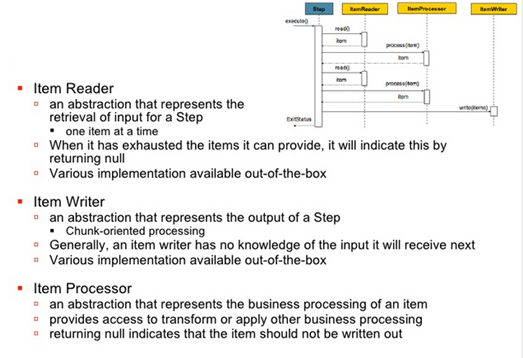
5. Domain Language



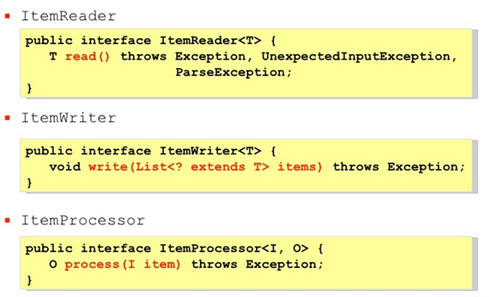
6. Domain Language of Batch



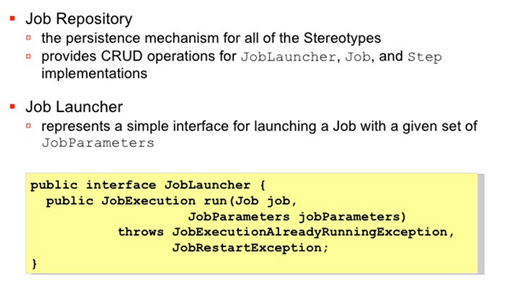
7. Domain Language of Batch



8. Domain Language of Batch



9. Domain Language for Batch



MBeans and MBeanExporter

Background

## Introduction - The JMX technology provides the tools for building distributed, Web-based, modular and dynamic solutions for managing and monitoring devices, applications, and service-driven networks. By design, this standard is suitable for adapting legacy systems, implementing new management and monitoring solutions, and plugging into those of the future.  Starting with the J2SE platform 5.0, JMX technology is included in the Java SE platform

## Spring and JMX

The JMX support in Spring provides you with the features to easily and transparently integrate your Spring application into a JMX infrastructure.

Specifically, Spring's JMX support provides four core features:

* The automatic registration of any Spring bean as a JMX MBean
* A flexible mechanism for controlling the management interface of your beans
* The declarative exposure of MBeans over remote, JSR-160 connectors
* The simple proxying of both local and remote MBean resources

These features are designed to work without coupling your application components to either Spring or JMX interfaces and classes. Indeed, for the most part your application classes need not be aware of either Spring or JMX in order to take advantage of the Spring JMX features

Reference - <https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch22s02.html>