Integration Patterns and Practices

Version 39.0, Spring '17



@salesforcedocs

Last updated: March 10, 2017

© Copyright 2000–2017 salesforce.com, inc. All rights reserved. Salesforce is a registered trademark of salesforce.com, inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

INTRODUCTION 1

Chapter 1: Integration Patterns Overview 1

Purpose and Scope 1
Pattern Template 1
Pattern Summary 2
Pattern Approach 3
Pattern Selection Guide 3
Middleware Terms and Definitions 5

DESIGN PATTERN CATALOG 9

Chapter 2: Remote Process Invocation—Request and Reply 9

Chapter 3: Remote Process Invocation—Fire and Forget 17

Chapter 4: Batch Data Synchronization 28

Chapter 5: Remote Call-In 34

Chapter 6: UI Update Based on Data Changes 43

APPENDICES 46

Appendix A: Resources—External 46

Appendix B: Resources—Salesforce 47

Appendix C: Security Considerations 48

INDEX 50

INTRODUCTION

CHAPTER 1 Integration Patterns Overview

When you implement Salesforce, you frequently need to integrate it with other applications. Although each integration scenario is unique, there are common requirements and issues that developers must resolve.

This document describes strategies (in the form of patterns) for these common integration scenarios. Each pattern describes the design and approach for a particular scenario rather than a specific implementation. In this document you'll find:

- A number of patterns that address key "archetype" integration scenarios
- A selection matrix to help you determine which pattern best fits your scenario
- Integration tips and best practices

Purpose and Scope

This document is for designers and architects who need to integrate the Force.com platform with their on-premises applications. This content is a distillation of many successful implementations by Salesforce architects and partners.

Read the pattern summary and selection matrix if you're considering large-scale adoption of Salesforce-based applications (or the Force.com or Database.com tool sets) so that you can become familiar with the integration capabilities and options available. Architects and developers should consider these pattern details and best practices during the design and implementation phase of a Salesforce integration project.

If implemented properly, these patterns enable you to get to production as fast as possible and have the most stable, scalable, and maintenance-free set of applications possible. Salesforce's own consulting architects use these patterns as reference points during architectural reviews and are actively engaged in maintaining and improving them.

As with all patterns, they cover the majority of integration scenarios, but not all. While Salesforce allows for user interface (UI) integration—mashups, for example—such integration is outside the scope of this document. If you feel that your requirements are outside the bounds of what these patterns describe, please speak with your Salesforce representative.

Pattern Template

Each integration pattern follows a consistent structure. This provides consistency in the information provided in each pattern and also makes it easier to compare patterns.

Name

The pattern identifier that also indicates the type of integration contained in the pattern.

Context

The overall integration scenario that the pattern addresses. Context provides information about what users are trying to accomplish and how the application will behave to support the requirements.

Problem

The scenario or problem (expressed as a question) that the pattern is designed to solve. When reviewing the patterns, read this section to quickly understand if the pattern is appropriate for your integration scenario.

Forces

The constraints and circumstances that make the stated scenario difficult to solve.

Solution

The recommended way to solve the integration scenario.

Sketch

A UML sequence diagram that shows you how the solution addresses the scenario.

Results

Explains the details of how to apply the solution to your integration scenario and how it resolves the forces associated with that scenario. This section also contains new challenges that can arise as a result of applying the pattern.

Sidebars

Additional sections related to the pattern that contain key technical issues, variations of the pattern, pattern-specific concerns, and so on.

Example

An end-to-end scenario that describes how the design pattern is used in a real-world Salesforce scenario. The example explains the integration goals and how to implement the pattern to achieve those goals.

Pattern Summary

The following table lists the integration patterns contained in this document.

List of Patterns

Pattern	Scenario
Remote Process Invocation—Request and Reply	Salesforce invokes a process on a remote system, waits for completion of that process, and then tracks state based on the response from the remote system.
Remote Process Invocation—Fire and Forget	Salesforce invokes a process in a remote system but doesn't wait for completion of the process. Instead, the remote process receives and acknowledges the request and then hands off control back to Salesforce.
Batch Data Synchronization	Data stored in Force.com should be created or refreshed to reflect updates from an external system, and when changes from Force.com should be sent to an external system. Updates in either direction are done in a batch manner.
Remote Call-In	Data stored in Force.com is created, retrieved, updated, or deleted by a remote system.
UI Update Based on Data Changes	The Salesforce user interface must be automatically updated as a result of changes to Salesforce data.

Pattern Approach

The integration patterns in this document are classified into two categories:

- Data Integration—These patterns address the requirement to synchronize data that resides in two or more systems so that both systems always contain timely and meaningful data. Data integration is often the simplest type of integration to implement, but requires proper information management techniques to make the solution sustainable and cost-effective. Such techniques often include aspects of Master Data Management (MDM), data governance, mastering, de-duplication, data flow design, and others.
- Process Integration—The patterns in this category address the need for a business process to leverage two or more applications to complete its task. When you implement a solution for this type of integration, the triggering application has to call across process boundaries to other applications. Usually, these patterns also include both orchestration (where one application is the central "controller") and choreography (where applications are multi-participants and there is no central "controller"). These types of integrations can often require complex design, testing, and exception handling requirements. Also, such composite applications are typically more demanding on the underlying systems because they often support long-running transactions, and the ability to report on and/or manage process state.

Choosing the best integration strategy for your system is not trivial. There are many aspects to take into consideration and many tools that can be used, with some tools being more appropriate than others for certain tasks. Each pattern addresses specific critical areas including the capabilities of each of the systems, volume of data, failure handling, and transactionality.

Pattern Selection Guide

The selection matrix table lists the patterns, along with key aspects, to help you determine the pattern that best fits your integration requirements. The patterns in the matrix are categorized using the following dimensions:

Aspect	Description
Source/Target	Specifies the requestor of the integration transaction along with the target(s) that provide the information. Note that the technical capabilities of the source and target systems, coupled with

Aspect	Description
	the type and timing of the integration, may require an additional middleware or integration solution. See the details of each pattern for additional details.
Type	Specifies the style of integration: Process or Data. • Process—Process-based integrations can be defined as ways to integrate the processing of functional flow across two or more applications. These integrations typically involve a higher level of abstraction and complexity, especially with regard to transactionality and rollback. • Data—Data integrations can be defined as the integration of the information used by applications. These integrations can range from a simple table insert or upsert, to complex data updates requiring referential integrity and complex translations.
Timing	Specifies the blocking (or non-blocking) nature of the integration. • Synchronous—Blocking or "near-real-time" requests can be defined as "request/response operation, and the result of the process is returned to the caller immediately via this operation."[1] • Asynchronous—Non-blocking, queue, or message-based requests are "invoked by a one-way operation and the results (and any faults) are returned by invoking other one-way operations."[2] The caller therefore makes the request and continues, without waiting for a response.

Note: An integration can require an external middleware or integration solution (for example, Enterprise Service Bus) depending on which aspects apply to your integration scenario.

Pattern Selection Matrix

The following table lists the patterns, along with key aspects, to help you determine the pattern that best fits your integration requirements.

Source/Target	Type		Timing		Key Pattern(s) to Consider
	Process Integration	Data Integration	Synchronous	Asynchronous	
Salesforce –> System (s)	X		X		Remote Process Invocation—Request and Reply
				X	Remote Process Invocation—Fire and Forget
		X	X		Remote Process Invocation—Request and Reply
				X	UI Update Based on Data Changes
System –> Salesforce	X		X		Remote Call-In
				X	Remote Call-In
		X	X		Remote Call-In

[1] "Synchronous and asynchronous processes," IBM Corporation, last accessed May 18, 2012, *http://publib.boulder.ibm.com/infocenter/adiehelp/v5r1m1/index.jsp?topic=%2Fcom.ibm.etools.ctc.flow.doc%2Fconcepts%2Fcsynchf.html*.

[2] Ibid.

Source/Target	Type		Timing		Key Pattern(s) to Consider
	Process Integration	Data Integration	Synchronous	Asynchronous	
				X	Batch Data Synchronization

Middleware Terms and Definitions

This table lists some key terms related to middleware and their definitions with regards to these patterns.

Term	Definition
Event handling	Event handling is the receipt of an identifiable occurrence at a designated receiver ("handler"). The key processes involved in event handling include: • Identifying where an event should be forwarded. • Executing that forwarding action. • Receiving a forwarded event. • Taking some kind of appropriate action in response, such as writing to a log, sending an error or recovery process, or sending an additional message. Note that the event handler might ultimately forward the event to an event consumer. Common uses of this feature with middleware can be extended to include the popular "publish/subscribe" or "pub/sub" capability. In a publish/subscribe scenario, the middleware routes requests or messages to active data-event subscribers from active data-event publishers. These consumers with active listeners can then retrieve the events as they are published. In Salesforce integrations using middleware, the control of event handling is assumed by the middleware layer; it collects all relevant events (synchronous or asynchronous) and manages distribution to all endpoints, including Salesforce.
Protocol conversion	Protocol conversion "is typically a software application that converts the standard or proprietary protocol of one device to the protocol suitable for another device to achieve interoperability. In the context of middleware, connectivity to a particular target system may be constrained by protocol. In such cases, the message format needs to be converted to or encapsulated within the format of the target system, where the payload can be extracted. This is also known as tunneling." [3] Salesforce doesn't support native protocol conversion, so it's assumed that any such requirements are met by either the middleware layer or the endpoint. See *http://searchsoa.techtarget.com/definition/event-handler*.
Translation and transformation	Transformation is the ability to map one data format to another to ensure interoperability between the various systems being integrated. Typically, this entails reformatting messages en route to match the requirements of the sender or recipient. In more complex cases, one application can send a message in its own native format, and two or more other applications might each receive a copy of the message in their own native format.

[3] Gregor Hohpe, and Bobby Woolf, *Enterprise Integration Patterns* (Boston: Addison-Wesley Professional, 2003).

Term	Definition
	Middleware translation and transformation tools often include the ability to create service facades for legacy or other non-standard endpoints; this allows those endpoints to appear to be service-addressable. With Salesforce integrations, it's assumed that any such requirements are met by either the middleware layer or the endpoint. Transformation of data can be coded in Apex, but we don't recommend it due to maintenance and performance considerations. See *http://en.wikipedia.org/wiki/Message-oriented_middleware*.
Queuing and buffering	Queuing and buffering generally rely on asynchronous message passing, as opposed to a request-response architecture. In asynchronous systems, message queues provide temporary storage when the destination program is busy or connectivity is compromised. In addition, most asynchronous middleware systems provide persistent storage to back up the message queue. The key benefit of an asynchronous message process is that if the receiver application fails for any reason, the senders can continue unaffected; the sent messages simply accumulate in the message queue for later processing when the receiver restarts. Salesforce provides only explicit queuing capability in the form of workflow-based outbound messaging. To provide true message queueing for other integration scenarios (including orchestration, process choreography, quality of service, and so on) a middleware solution is required. See *http://en.wikipedia.org/wiki/Message-oriented_middleware*.
Synchronous transport protocols	Synchronous transport protocols refer to protocols that support activities wherein a "single thread in the caller sends the request message, blocks to wait for the reply message, and then processes the reply....The request thread awaiting the response implies that there is only one outstanding request or that the reply channel for this request is private for this thread."[4]
Asynchronous transport protocols	Asynchronous transport protocols refer to protocols supporting activities wherein "one thread in the caller sends the request message and sets up a callback for the reply. A separate thread listens for reply messages. When a reply message arrives, the reply thread invokes the appropriate callback, which reestablishes the caller's context and processes the reply. This approach enables multiple outstanding requests to share a single reply thread."[5]
Mediation routing	Mediation routing is the specification of a complex "flow" of messages from component to component. For example, many middleware-based solutions depend on a message queue system. While some implementations permit routing logic to be provided by the messaging layer itself, others depend on client applications to provide routing information or allow for a mix of both paradigms. In such complex cases, mediation (on the part of middleware) simplifies development, integration, and validation). "Specifically, Mediators coordinate a group of objects so that they [do not] need to know how to coordinate with each other….Then, each consumer could focus on processing a particular kind of message, and the coordinator [Mediator] could make sure the right message gets to the right consumer."[6]

[4] Gregor Hohpe, and Bobby Woolf, *Enterprise Integration Patterns* (Boston: Addison-Wesley Professional, 2003).
[5] Ibid.
[6] Ibid.

Term	Definition
Process choreography and service orchestration	Process choreography and service orchestration are each forms of "service composition" where any number of endpoints and capabilities are being coordinated. The difference between choreography and service orchestration is: • Choreography can be defined as "behavior resulting from a group of interacting individual entities with no central authority."[7] • Orchestration can be defined as "behavior resulting from a central conductor coordinating the behaviors of individual entities performing tasks independent of each other."[8] In addition, "an orchestration shows the complete behavior of each service whereas the choreography combines the interface behavior descriptions of each service."[9] Portions of business process choreographies can be built in Salesforce workflows or using Apex. We recommend that all complex orchestrations be implemented in the middleware layer because of Salesforce timeout values and governor limits (especially in solutions requiring true transaction handling).
Transactionality (encryption, signing, reliable delivery, transaction management)	Transactionality can be defined as the ability to support global transactions that encompass all necessary operations against each required resource. Transactionality implies the support of all four ACID (atomicity, consistency, isolation, durability) properties, where atomicity guarantees all-or-nothing outcomes for the unit of work (transaction). While Salesforce is transactional within itself, it's not able to participate in distributed transactions or transactions initiated outside of Salesforce. Therefore, it's assumed that for solutions requiring complex, multi-system transactions, transactionality (and associated roll-back/compensation mechanisms) be implemented at the middleware layer. See *http://en.wikipedia.org/wiki/Distributed_transaction*.
Routing	Routing can be defined as specifying the complex flow of messages from component to component. In modern services-based solutions, such message flows can be based in a number of criteria, including header, content type, rule, and priority. With Salesforce integrations, it's assumed that any such requirements are met by either the middleware layer or the endpoint. Message routing can be coded in Apex, but we don't recommend it due to maintenance and performance considerations.
Extract, transform, and load	Extract, transform, and load (ETL) refers to a process that involves: • Extracting data from the source systems. This typically involves data from a number of source systems, and both relational and non-relational structures. • Transforming the data to fit operational needs, which can include data quality levels. The transform stage usually applies a series of rules or functions to the extracted data from the source to derive the data for loading into the end target(s).

[7] "Choreography and Orchestration: A Software Perspective,"e-Zest, last accessed April 11, 2013, *http://www.e-zest.net/blog/choreography-and-orchestration-a-software-perspective/*.
[8] Ibid.
[9] "Orchestration vs. Choreography," Stack Overflow, last accessed April 11, 2013, *http://stackoverflow.com/questions/4127241/orchestration-vs-choreography*.

Term	Definition
	• Loading the data into the target system. The target system can vary widely from database, operational data store, data mart, data warehouse, or other operational systems. While not strictly necessary, most mature ETL tools provide a change data capture capability. This capability is where the tool identifies records in the source system that have changed since the last extract, thereby reducing the amount of record processing. See *http://en.wikipedia.org/wiki/Extract,_transform,_load* and *http://en.wikipedia.org/wiki/Change_data_capture*.

CHAPTER 2 Remote Process Invocation—Request and Reply

Context

You use Salesforce to track leads, manage your pipeline, create opportunities, and capture order details that convert leads to customers. However, the Salesforce system doesn't contain or process orders. After the order details are captured in Salesforce, the order is created in the remote system, which manages the order to conclusion.

When you implement this pattern, Salesforce calls the remote system to create the order and then waits for successful completion. If successful, the remote system synchronously replies with the order status and order number. As part of the same transaction, Salesforce updates the order number and status internally. The order number is used as a foreign key for subsequent updates to the remote system.

Problem

When an event occurs in Salesforce, how do you initiate a process in a remote system, pass the required information to that process, receive a response from the remote system, and then use that response data to make updates within Salesforce?

Forces

Consider the following forces when applying solutions based on this pattern.

- Does the call to the remote system require Salesforce to wait for a response before continuing processing? Is the call to the remote system a synchronous request-reply or an asynchronous request?
- If the call to the remote system is synchronous, does Salesforce have to process the response as part of the same transaction as the initial call?
- Is the message size small or large?
- Is the integration based on the occurrence of a specific event, such as a button click in the Salesforce user interface, or DML-based events?

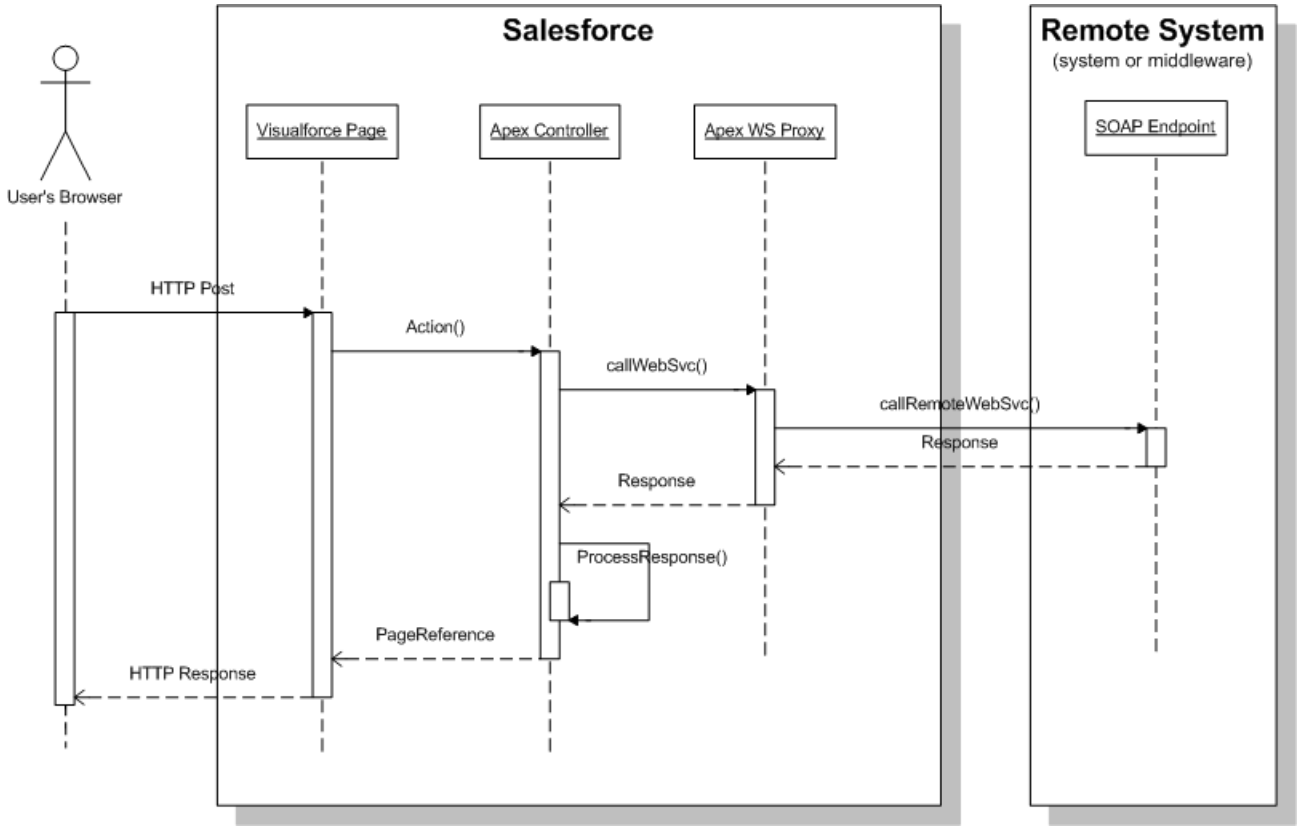
Solution

The following table contains solutions to this integration problem.

Solution	Fit	Comments
A custom Visualforce page or button initiates an Apex SOAP callout in a synchronous manner.	Best	Salesforce enables you to consume a WSDL and generate a resulting proxy Apex class. This class provides the necessary logic to call the remote service. A user-initiated action on a Visualforce page then calls an Apex controller action that then executes this proxy Apex class to perform the remote call. Visualforce pages require customization of the Salesforce application.
A custom Visualforce page or button initiates an Apex HTTP callout in a synchronous manner.	Best	Salesforce enables you to invoke HTTP services using standard GET, POST, PUT, and DELETE methods. You can use several HTTP classes to integrate with RESTful services. It's also possible to integrate to SOAP-based services by manually constructing the SOAP message. The latter is not recommended because it's possible for Salesforce to consume WSDLs to generate proxy classes. A user-initiated action on a Visualforce page then calls an Apex controller action that then executes this proxy Apex class to perform the remote call. Visualforce pages require customization of the Salesforce application.
A trigger that's invoked from Salesforce data changes performs an Apex SOAP or HTTP callout in a synchronous manner.	Suboptimal	You can use Apex triggers to perform automation based on record data changes. An Apex proxy class can be executed as the result of a DML operation by using an Apex trigger. However, all calls made from within the trigger context must execute asynchronously from the initiating event. Therefore, this solution isn't recommended for this integration problem. This solution is better suited for the Remote Process Invocation—Fire and Forget pattern.
A batch Apex job performs an Apex SOAP or HTTP callout in a synchronous manner.	Suboptimal	You can make calls to a remote system from a batch job. This solution allows batch remote process execution and processing of the response from the remote system in Salesforce. However, a given batch has limits to the number of calls. For more information, see Governor Limits. A given batch run can execute multiple transaction contexts (usually in intervals of 200 records). The governor limits are reset per transaction context.

Sketch

The following diagram illustrates a synchronous remote process invocation using Apex calls.



In this scenario:

1. You initiate an action on the Visualforce page (for example, clicking a button).
2. The browser performs an HTTP POST that in turn performs an action on the corresponding Apex controller.
3. The controller calls a previously generated Apex web service proxy class.
4. The proxy class performs the actual call to the remote web service.
5. The response from the remote system is returned to the Apex controller. The controller processes the response, updates data in Salesforce as required, and re-renders the page.

In cases where the subsequent state must be tracked, the remote system returns a unique identifier that's stored on the Salesforce record.

Results

The application of the solutions related to this pattern allows for event-initiated remote process invocations in which Salesforce handles the processing.

Calling Mechanisms

The calling mechanism depends on the solution chosen to implement this pattern.

Calling Mechanism	Description
Visualforce and Apex controllers	Used when the remote process is triggered as part of an end-to-end process involving the user interface, and the result must be displayed or updated in a Salesforce record. For example, the submission of a credit card payment to an external payment gateway and the payment results are immediately returned and displayed to the user.

Calling Mechanism	Description
	Integration that's triggered from user interface events usually requires the creation of custom Visualforce pages.
Apex triggers	Used primarily for invocation of remote processes using Apex callouts from DML-initiated events. For more information about this calling mechanism, see pattern Remote Process Invocation—Fire and Forget.
Apex batch classes	Used for invocation of remote processes in batch. For more information about this calling mechanism, see pattern Remote Process Invocation—Fire and Forget.

Error Handling and Recovery

It's important to include an error handling and recovery strategy as part of the overall solution.

- Error handling—When an error occurs (exceptions or error codes are returned to the caller), the caller manages error handling. For example, an error message displayed on the end-user's page or logged to a table requiring further action.
- Recovery—Changes aren't committed to Salesforce until the caller receives a successful response. For example, the order status isn't updated in the database until a response that indicates success is received. If necessary, the caller can retry the operation.

Idempotent Design Considerations

Idempotent capabilities guarantee that repeated invocations are safe. If idempotency isn't implemented, repeated invocations of the same message can have different results, potentially resulting in data integrity issues. Potential issues include the creation of duplicate records or duplicate processing of transactions.

It's important to ensure that the remote procedure being called is idempotent. It's almost impossible to guarantee that Salesforce only calls once, especially if the call is triggered from a user interface event. Even if Salesforce makes a single call, there's no guarantee that other processes (for example, middleware) do the same.

The most typical method of building an idempotent receiver is for it to track duplicates based on unique message identifiers sent by the consumer. Apex web service or REST calls must be customized to send a unique message ID.

In addition, operations that create records in the remote system must check for duplicates before inserting. Check by passing a unique record ID from Salesforce. If the record exists in the remote system, update the record. In most systems, this operation is termed as an upsert operation.

Security Considerations

Any call to a remote system must maintain the confidentiality, integrity, and availability of the request. The following security considerations are specific to using Apex SOAP and HTTP calls in this pattern.

- One-way SSL is enabled by default, but two-way SSL is supported with both self-signed and CA-signed certificates to maintain authenticity of both the client and server.
- Salesforce does not currently support WS-Security.
- Where necessary, consider using one-way hashes or digital signatures using the Apex Crypto class methods to ensure request integrity.
- The remote system must be protected by implementing the appropriate firewall mechanisms.

See Security Considerations.

Sidebars

Timeliness

Timeliness is of significant importance in this pattern. Usually:

- The request is typically invoked from the user interface, so the process must not keep the user waiting.
- Salesforce has a configurable timeout of up to 60 seconds for calls from Apex.
- Completion of the remote process is executed in a timely manner to conclude within the Salesforce timeout limit and within user expectations.

Data Volumes

This pattern is used primarily for small volume, real-time activities, due to the small timeout values and maximum size of the request or response for the Apex call solution. Do not use this pattern in batch processing activities in which the data payload is contained in the message.

Endpoint Capability and Standards Support

The capability and standards support for the endpoint depends on the solution that you choose.

Solution	Endpoint Considerations
Apex SOAP callouts	The endpoint must be able to receive a web service call via HTTP. Salesforce must be able to access the endpoint over the public Internet. This solution requires that the remote system is compatible with the standards supported by Salesforce. At the time of writing, the web service standards supported by Salesforce for Apex SOAP callouts are: • WSDL 1.1 • SOAP 1.1 • WSI-Basic Profile 1.1 • HTTP
Apex HTTP callouts	The endpoint must be able to receive HTTP calls. Salesforce must be able to access the endpoint over the public Internet. You can use Apex HTTP callouts to call RESTful services using the standard GET, POST, PUT, and DELETE methods.

State Management

When integrating systems, keys are important for ongoing state tracking. There are two options.

- Salesforce stores the remote system's primary or unique surrogate key for the remote record.
- The remote system stores the Salesforce unique record ID or some other unique surrogate key.

There are specific considerations for handling integration keys, depending on which system contains the master record, as shown in the following table.

Master System	Description
Salesforce	The remote system stores either the Salesforce RecordId or some other unique surrogate key from the record.
Remote system	The call to the remote process returns the unique key from the application, and Salesforce stores that key value in a unique record field.

Complex Integration Scenarios

In certain cases, the solution prescribed by this pattern can require the implementation of several complex integration scenarios. This is best served by using middleware or having Salesforce call a composite service. These scenarios include:

- Orchestration of business processes and rules involving complex flow logic
- Aggregation of calls and their results across calls to multiple systems
- Transformation of both inbound and outbound messages
- Maintaining transactional integrity across calls to multiple systems

Governor Limits

Due to the multitenant nature of the Salesforce platform, there are limits to Apex callouts have limits.

Description	Synchronous Limit	Asynchronous Limit
Total number of SOQL queries issued[1] (This limit doesn't apply to custom metadata types. In a single Apex transaction, custom metadata records can have unlimited SOQL queries.)	100	200
Total number of records retrieved by SOQL queries	50,000	
Total number of records retrieved by `Database.getQueryLocator`	10,000	
Total number of SOSL queries issued	20	
Total number of records retrieved by a single SOSL query	2,000	
Total number of DML statements issued[2]	150	
Total number of records processed as a result of DML statements, `Approval.process`, or `database.emptyRecycleBin`	10,000	
Total stack depth for any Apex invocation that recursively fires triggers due to `insert`, `update`, or `delete` statements[3]	16	
Total number of callouts (HTTP requests or Web services calls) in a transaction	100	
Maximum timeout for all callouts (HTTP requests or Web services calls) in a transaction	120 seconds	
Maximum number of methods with the `future` annotation allowed per Apex invocation	50	
Maximum number of Apex jobs added to the queue with `System.enqueueJob`	50	
Total number of `sendEmail` methods allowed	10	
Total heap size[4]	6 MB	12 MB
Maximum CPU time on the Salesforce servers[5]	10,000 milliseconds	60,000 milliseconds

Description	Synchronous Limit	Asynchronous Limit
Maximum execution time for each Apex transaction	10 minutes	
Maximum number of push notification method calls allowed per Apex transaction	10	
Maximum number of push notifications that can be sent in each push notification method call	2,000	

Middleware Capabilities

The following table highlights the desirable properties of a middleware system that participates in this pattern.

Property	Mandatory	Desirable	Not Required
Event handling		X	
Protocol conversion		X	
Translation and transformation		X	
Queuing and buffering		X	
Synchronous transport protocols	X		
Asynchronous transport protocols			X
Mediation routing		X	
Process choreography and service orchestration		X	
Transactionality (encryption, signing, reliable delivery, transaction management)		X	
Routing			X
Extract, transform, and load			X

Example

A utility company uses Salesforce and has a separate system that contains customer billing information. They want to display the billing history for a customer account without storing that data in Salesforce. They have an existing web service that returns a list of bills and the details for a given account, but can't display this data in a browser.

This requirement can be accomplished with the following approach.

1. Salesforce consumes the billing history service WSDL from an Apex proxy class.
2. Execute the Apex proxy class with the account number as the unique identifier by creating a Visualforce page and custom controller.
3. The custom controller then parses the return values from the Apex callout and the Visualforce page, and then renders the bill to the user.

This example demonstrates that:

- The state of the customer is tracked with an account number stored on the Salesforce account object

- Subsequent processing of the reply message by the caller

CHAPTER 3 Remote Process Invocation—Fire and Forget

Context

You use Salesforce to track leads, manage your pipeline, create opportunities, and capture order details that convert leads to customers. However, Salesforce isn't the system that holds or processes orders. After the order details are captured in Salesforce, an order must be created in the remote system, which manages the order through to its conclusion.

When you implement this pattern, Salesforce calls the remote system to create the order, but doesn't wait for the call's successful completion. The remote system can optionally update Salesforce with the new order number and status in a separate transaction.

Problem

When an event occurs in Salesforce, how do you initiate a process in a remote system and pass the required information to that process without waiting for a response from the remote system?

Forces

Consider the following forces when applying solutions based on this pattern.

- Does the call to the remote system require Salesforce to wait for a response before continuing processing? Is the call to the remote system synchronous request-reply or asynchronous?
- If the call to the remote system is synchronous, does the response need to be processed by Salesforce as part of the same transaction as the call?
- Is the message size small?
- Is the integration based on the occurrence of a specific event, such as a button click in the Salesforce user interface, or DML-based events?
- Is guaranteed message delivery from Salesforce to the remote system a requirement?
- Is the remote system able to participate in a contract-first integration in which Salesforce specifies the contract? In some solution variants (for example, outbound messaging), Salesforce specifies a contract that the remote system endpoint implements.
- Are declarative configuration methods preferred over custom Apex development? In this case, solutions such as outbound messaging are preferred over Apex callouts.

Solution

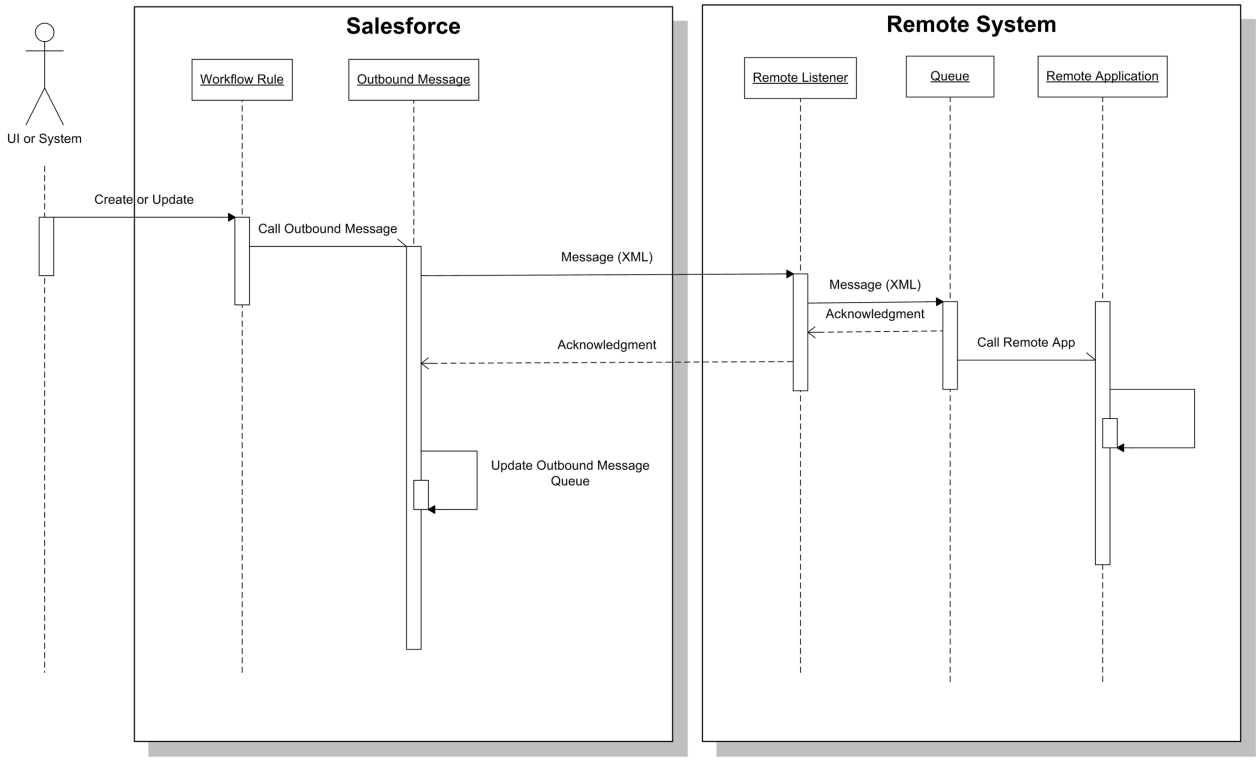
The following table contains solutions to this integration problem.

Solution	Fit	Comments
Workflow-driven outbound messaging	Best	No customization is required in Salesforce to implement outbound messaging. The recommended solution for this type of integration is when the remote process is invoked from an insert or update event. Salesforce provides a workflow-driven outbound messaging capability that allows sending SOAP messages to remote systems triggered by an insert or update operation in Salesforce. These messages are sent asynchronously and are independent of the Salesforce user interface. The outbound message is sent to a specific remote endpoint. The remote service must be able to participate in a contract-first integration where Salesforce provides the contract. On receipt of the message, if the remote service doesn't respond with a positive acknowledgment, Salesforce retries sending the message, providing a form of guaranteed delivery. When using middleware, this solution becomes a "first-mile" guarantee of delivery.
Outbound messaging and callbacks	Best	Callbacks provide a way to mitigate the impacts of out-of-sequence messaging. In addition, they handle these scenarios. • Idempotency— If an acknowledgment isn't received in a timely fashion, outbound messaging performs retries. Multiple messages can be sent to the target system. Using a callback ensures that the data retrieved is at a specific point in time rather than when the message was sent. • Retrieving more data—A single outbound message can send data only for a single object. A callback can be used to retrieve data from other related records, such as related lists associated with the parent object. The outbound message provides a unique SessionId that can be used as the authentication token. You can use the token to authenticate and authorize a callback using either the SOAP API or the REST API. The system performing the callback isn't required to separately authenticate to Salesforce. The standard methods of either API can then be used to perform the desired business functions. A typical use of this variant is the scenario in which Salesforce sends an outbound message to a remote system to create a record. The callback is used to update the original Salesforce record with the unique key of the record created in the remote system.

Solution	Fit	Comments
Custom Visualforce page that initiates an Apex SOAP or HTTP asynchronous callout	Good	This solution is typically used in user interface-based scenarios, but does require customization. In addition, the solution must handle guaranteed delivery of the message in the code. Similar to the solution for the Remote Process Invocation—Request and Reply pattern solution that specifies using a Visualforce page, together with an Apex callout. The difference is that in this pattern, Salesforce doesn't wait for the request to complete before handing off control to the user. After receiving the message, the remote system responds and indicates receipt of the message, then asynchronously processes the message. The remote system hands control back to Salesforce before it begins to process the message; therefore, Salesforce doesn't have to wait for processing to complete.
A trigger that's invoked from Salesforce data changes performs an Apex SOAP or HTTP asynchronous callout	Suboptimal	You can use Apex triggers to perform automation based on record data changes. An Apex proxy class can be executed as the result of a DML operation by using an Apex trigger. However, all calls made from within the trigger context must be executed asynchronously.
A batchApex job that performs an Apex SOAP or HTTP asynchronous callout	Suboptimal	Calls to a remote system can be performed from a batch job. This solution allows for batch remote process execution and for processing of the response from the remote system in Salesforce. However, there are limits to the number of calls for a given batch context. For more information, see the *Salesforce Limits Quick Reference Guide*.

Sketch

The following diagram illustrates a call from Salesforce to a remote system in which create or update operations on a record trigger the call.



In this scenario:

1. A DML update or insert occurs on a given set of records in Salesforce.
2. A Salesforce workflow rule triggers, based on a given set of conditions.
3. This workflow rule invokes a pre-configured outbound message that sends a SOAP-based message to a remote listener.
4. The remote listener receives the SOAP message, places the message on a local queue, and returns a positive acknowledgment to Salesforce as part of the same transaction context.
5. The queueing application forwards the message to the remote application for processing.
6. Salesforce receives the acknowledgment and completes the request, but doesn't wait for the remote application to process the message.
7. Salesforce waits for an acknowledgment from the remote system for up to 10 seconds. After 10 seconds, Salesforce retries sending the outbound message request for up to 24 hours.

In the case where the remote system must perform operations against Salesforce, you can implement an optional callback operation. The outbound message sends a SessionId that you can use in the callback to authenticate and authorize an API or web service call into Salesforce.

Results

The application of the solutions related to this pattern allows for:

- User interface–initiated remote process invocations in which the result of the transaction can be displayed to the end user
- DML event-initiated remote process invocations in which the result of the transaction can be processed by the calling process

Calling Mechanisms

The calling mechanism depends on the solution chosen to implement this pattern.

Calling Mechanism	Description
Visualforce and Apex controllers	Used to invoke a remote process asynchronously using an Apex callout.
Workflow rules	Used only for the outbound messaging solution. Create and update DML events trigger the Salesforce workflow rules, which can then send a message to a remote system.
Apex triggers	Used primarily for invocation of remote processes, using Apex callouts from DML-initiated events.
Apex batch classes	Used for invocation of remote processes in batch mode.

Error Handling and Recovery

An error handling and recovery strategy must be considered as part of the overall solution. The best method depends on the solution you choose.

Solution	Error Handling and Recovery Strategy
Apex callouts	• Error handling—The remote system hands off invocation of the end process, so the callout only handles exceptions in the initial invocation of the remote service. For example, a timeout event is triggered if no positive acknowledgment is received from the remote callout. The remote system must handle subsequent errors when the initial invocation is handed off for asynchronous processing. • Recovery—Recovery is more complex in this scenario. A custom retry mechanism must be created if quality of service requirements dictate it.
Outbound messaging	• Error handling—Because this pattern is asynchronous, the remote system handles error handling. For outbound messaging, Salesforce initiates a retry operation if no positive acknowledgment is received within the timeout period, for up to 24 hours. Error handling must be performed in the remote service because the message is effectively handed off to the remote system in a "fire-and-forget" manner. • Recovery—Because this pattern is asynchronous, the system must initiate retries based on the service's quality of service requirements. For outbound messaging, Salesforce initiates retries if no positive acknowledgment is received from the outbound listener within the timeout period, up to 24 hours. The retry interval increases exponentially over time, starting with 15-second intervals and ending with 60-minute intervals. Administrators must monitor this queue for any messages exceeding the 24-hour delivery period and retry manually, if necessary. For custom Apex callouts, a custom retry mechanism must be built if the quality of service requirements warrant it.

Idempotent Design Considerations

The idempotent design considerations in the Remote Process Invocation—Request and Reply pattern also apply to this pattern.

Idempotency is important for outbound messaging because it's asynchronous and retries are initiated when no positive acknowledgement is received. Therefore, the remote service must be able to handle messages from Salesforce in an idempotent fashion.

Outbound messaging sends a unique ID per message and this ID remains the same for any retries. The remote system can track duplicate messages based on this unique ID. The unique record ID for each record being updated is also sent, and can be used to prevent duplicate record creation.

Security Considerations

Any call to a remote system must maintain the confidentiality, integrity, and availability of the request. Different security considerations apply, depending on the solution you choose.

Solution	Security Considerations
Apex callouts	A call to a remote system must maintain the confidentiality, integrity, and availability of the request. The following are security considerations specific to using Apex SOAP and HTTP calls in this pattern. • One-way SSL is enabled by default, but two-way SSL is supported with both self-signed and CA-signed certificates to maintain authenticity of both the client and server. • Salesforce does not support WS-Security when generating the Apex proxy class. • Where necessary, consider using one-way hashes or digital signatures using the Apex Crypto class methods to ensure the integrity of the request. • The remote system must be protected by implementing the appropriate firewall mechanisms.
Outbound Messaging	For outbound messaging, one-way SSL is enabled by default. However, two-way SSL can be used together with the Salesforce outbound messaging certificate. The following are some additional security considerations. • Whitelist Salesforce server IP ranges for remote integration servers. • Protect the remote system by implementing the appropriate firewall mechanisms.

See Security Considerations.

Sidebars

Timeliness

Timeliness is less of a factor with the fire-and-forget pattern. Control is handed back to the client either immediately or after positive acknowledgment of a successful hand-off in the remote system. With Salesforce outbound messaging, the acknowledgment must occur within 24 hours; otherwise, the message expires.

Data Volumes

Data volume considerations depend on which solution you choose and communication type.

Solution	Communication Type	Limits
Apex callouts from a batch or a trigger	Asynchronous or batch	It's possible to perform Apex callouts from an Apex batch class, but there are a limited number of callouts per batch execution (a single batch run is made up of multiple small batch executions). Therefore, callouts in batch must be able to support multiple records in a given request rather than being treated as atomic transactions.

Solution	Communication Type	Limits
		In addition, asynchronous Apex callouts must be invoked through triggers that fire due to DML record changes. DML operations in Salesforce can occur for more than one record and a single trigger invocation can consist of up to 200 records. Therefore, any DML-initiated or batch-initiated integration can't be implemented on a single record per message basis given the previous limitations.
Outbound messaging	Asynchronous	A single outbound message can contain up to 100 records. Therefore, it's important that the remote listener can handle more than a single record in its implementation.

Endpoint Capability and Standards Support

The capability and standards support for the endpoint depends on the solution that you choose.

Solution	Endpoint Considerations
Apex SOAP callouts	The endpoint must be able to process a web service call via HTTP. Salesforce must be able to access the endpoint over the public Internet. This solution requires that the remote system is compatible with the standards supported by Salesforce. At the time of writing, the web service standards supported by Salesforce for Apex SOAP callouts are: • WSDL 1.1 • SOAP 1.1 • WSI-Basic Profile 1.1 • HTTP
Apex HTTP callouts	The endpoint must be able to receive HTTP calls and be accessible over the public internet by Salesforce. Apex HTTP callouts can be used to call RESTful services using the standard GET, POST, PUT, and DELETE methods.
Outbound message	• The endpoint must be able to implement a listener that can receive SOAP messages in predefined format sent from Salesforce. • The remote listener must participate in a contract-first implementation, where the contract is supplied by Salesforce. • Each outbound message has its own predefined WSDL.

State Management

When integrating systems, unique record identifiers are important for ongoing state tracking. For example, if a record is created in the remote system, you have two options.

- Salesforce stores the remote system's primary or unique surrogate key for the remote record.
- The remote system stores the Salesforce unique record ID or some other unique surrogate key.

The following table lists considerations for state management in this pattern.

Master System	Description
Salesforce	The remote system must store either the Salesforce RecordId or some other unique surrogate key in the Salesforce record.
Remote system	Salesforce must store a reference to the unique identifier in the remote system. Because the process is asynchronous, storing this unique identifier can't be part of the original transaction. Salesforce must provide a unique ID in the call to the remote process. The remote system must then call back to Salesforce to update the record in Salesforce with the remote system's unique identifier, using the Salesforce unique ID. The callback implies specific state handling in the remote application to store the Salesforce unique identifier for that transaction to use for the callback when processing is complete, or the Salesforce unique identifier is stored on the remote system's record.

Complex Integration Scenarios

Each solution in this pattern has different considerations for complex integration scenarios, including transformation, process orchestration, and so on.

Solution	Considerations
Apex callouts	In certain cases, solutions prescribed by this pattern require implementing several complex integration scenarios best served using middleware or having Salesforce call a composite service. These scenarios include: • Orchestration of business processes and rules involving complex flow logic • Aggregation of calls and their results across calls to multiple systems • Transformation of both inbound and outbound messages • Maintaining transactional integrity across calls to multiple systems
Outbound messaging	Given the static, declarative nature of the outbound message, no complex integration scenarios, such as aggregation, orchestration, or transformation, can be performed in Salesforce. The remote system or middleware must handle these types of operations .

Governor Limits

Due to the multitenant nature of the Salesforce platform, there are limits to outbound callouts. Limits depend on the type of outbound call and the timing of the call. There are no governor limits for outbound messaging.

Description	Synchronous Limit	Asynchronous Limit
Total number of SOQL queries issued[1] (This limit doesn't apply to custom metadata types. In a single Apex transaction, custom metadata records can have unlimited SOQL queries.)	100	200
Total number of records retrieved by SOQL queries	50,000	
Total number of records retrieved by `Database.getQueryLocator`	10,000	
Total number of SOSL queries issued	20	

Description	Synchronous Limit	Asynchronous Limit
Total number of records retrieved by a single SOSL query	2,000	
Total number of DML statements issued[2]	150	
Total number of records processed as a result of DML statements, `Approval.process`, or `database.emptyRecycleBin`	10,000	
Total stack depth for any Apex invocation that recursively fires triggers due to `insert`, `update`, or `delete` statements[3]	16	
Total number of callouts (HTTP requests or Web services calls) in a transaction	100	
Maximum timeout for all callouts (HTTP requests or Web services calls) in a transaction	120 seconds	
Maximum number of methods with the `future` annotation allowed per Apex invocation	50	
Maximum number of Apex jobs added to the queue with `System.enqueueJob`	50	
Total number of `sendEmail` methods allowed	10	
Total heap size[4]	6 MB	12 MB
Maximum CPU time on the Salesforce servers[5]	10,000 milliseconds	60,000 milliseconds
Maximum execution time for each Apex transaction	10 minutes	
Maximum number of push notification method calls allowed per Apex transaction	10	
Maximum number of push notifications that can be sent in each push notification method call	2,000	

Reliable Messaging

Reliable messaging attempts to resolve the issue of guaranteeing the delivery of a message to a remote system in which the individual components are unreliable. The method of ensuring receipt of a message by the remote system depends on the solution you choose.

Solution	Reliable Messaging Considerations
Apex callouts	Salesforce doesn't provide explicit support for any reliable messaging protocols (for example, WS-ReliableMessaging). We recommend that the remote endpoint receiving the Salesforce message implement a reliable messaging system, like JMS or MQ. This system ensures full end-to-end guaranteed delivery to the remote system that ultimately processes the message. However, this system doesn't ensure guaranteed delivery from Salesforce to the remote endpoint that it calls. Guaranteed delivery must be handled through customizations to Salesforce. Specific techniques, such as processing a positive acknowledgment from the remote endpoint in addition to custom retry logic, must be implemented.

Solution	Reliable Messaging Considerations
Outbound messaging	Outbound messaging provides a form of reliable messaging. If no positive acknowledgment is received from the remote system, the process retries for up to 24 hours. This process guarantees delivery only to the point of the remote listener. In most implementations, the remote listener calls another remote service. Ideally, the invocation of this remote service through a reliable messaging system ensures full end-to-end guaranteed delivery. The positive acknowledgement to the Salesforce outbound message occurs after the remote listener has successfully placed its own message on its local queue.

Middleware Capabilities

The following table highlights the desirable properties of a middleware system that participates in this pattern.

Property	Mandatory	Desirable	Not Required
Event handling		X	
Protocol conversion		X	
Translation and transformation		X	
Queuing and buffering	X		
Synchronous transport protocols			X
Asynchronous transport protocols	X		
Mediation routing		X	
Process choreography and service orchestration		X	
Transactionality (encryption, signing, reliable delivery, transaction management)	X		
Routing			X
Extract, transform, and load			X

Solution Variant—Outbound Messaging and Message Sequencing

Salesforce outbound messaging can't guarantee the sequence of delivery for its messages because a single message can be retrieved over a 24-hour period. There are multiple methods for handling message sequencing in the remote system.

- Salesforce sends a unique message ID for each instance of an outbound message. The remote system discards any messages that have a duplicate message ID.
- Salesforce sends only the RecordId. The remote system makes a callback to Salesforce to obtain the necessary data to process the request.

Solution Variant—Outbound Messaging and Deletes

Salesforce workflow rules can't track deletion of a record. The rules can track only the insert or update of a record. Therefore, it's not possible to directly initiate an outbound message from the deletion of a record. You can initiate a message indirectly with the following process.

1. Create a custom object to store key information from the deleted records.
2. Create an Apex trigger, fired by the deletion of the base record, to store information such as the unique identifier in the custom object.
3. Implement a workflow rule to initiate an outbound message based on the creation of the custom object record.

It's important that state tracking is enabled by storing the remote system's unique identifier in Salesforce or Salesforce's unique identifier in the remote system.

Example

A telecommunications company wants to use Salesforce as a front end for creating accounts using the Lead to Opportunity process. The creation of an order is initiated in Salesforce when the opportunity is closed and won, but the back-end ERP system is the data master. The order must be saved to the Salesforce opportunity record, and the opportunity status changed to indicate that the order was created.

The following constraints apply.

- The ERP system must be able to participate in a contract-first integration in which its service implements a Salesforce WSDL interface.
- There can't be any custom development in Salesforce.
- You don't require immediate notification of the order number after the opportunity converts to an order.

This example is best implemented using Salesforce outbound messaging, but does require the implementation of a proxy service by the remote system.

On the Salesforce side:

- Create a workflow rule to initiate the outbound message (for example, when the opportunity status changes to "Close-Won").
- Create an outbound message that sends only the opportunity RecordId and a SessionId for a subsequent callback.

On the remote system side:

- Create a proxy service that can implement the Salesforce outbound message WSDL interface.
- The service receives one or more notifications indicating that the opportunity is to be converted to an order.
- The service transforms and places the message on a local message queue. On notification of receipt, the service replies with a positive acknowledgment back to the Salesforce outbound message.
- The local message queue forwards the message to the back-end ERP system so that the order can be created.
- After the order is successfully created, a separate thread calls back to Salesforce using the SessionId as the authentication token. The callback updates the opportunity with the order number and status. This callback can be done using previously documented pattern solutions, such as the Salesforce SOAP API, REST API, or an Apex web service.

This example demonstrates the following.

- Implementation of a remote process invoked asynchronously
- End-to-end guaranteed delivery
- Subsequent callback to Salesforce to update the state of the record

CHAPTER 4 Batch Data Synchronization

Context

You're moving your CRM implementation to Salesforce and want to:

- Extract and transform accounts, contacts, and opportunities from the current CRM system and load the data into Salesforce (initial data import).
- Extract, transform, and load customer billing data into Salesforce from a remote system on a weekly basis (ongoing).
- Extract customer activity information from Salesforce and import it into an on-premises data warehouse on a weekly basis (ongoing).

Problem

How do you import data into Salesforce and export data out of Salesforce, taking into consideration that these imports and exports can interfere with end-user operations during business hours, and involve large amounts of data?

Forces

There are various forces to consider when applying solutions based on this pattern:

- Should the data be stored in Salesforce? If not, there are other integration options an architect can and should consider (mashups, for example).
- If the data should be stored in Salesforce, should the data be refreshed in response to an event in the remote system?
- Should the data be refreshed on a scheduled basis?
- Does the data support primary business processes?
- Are there analytics (reporting) requirements that are impacted by the availability of this data in Salesforce?

Solution

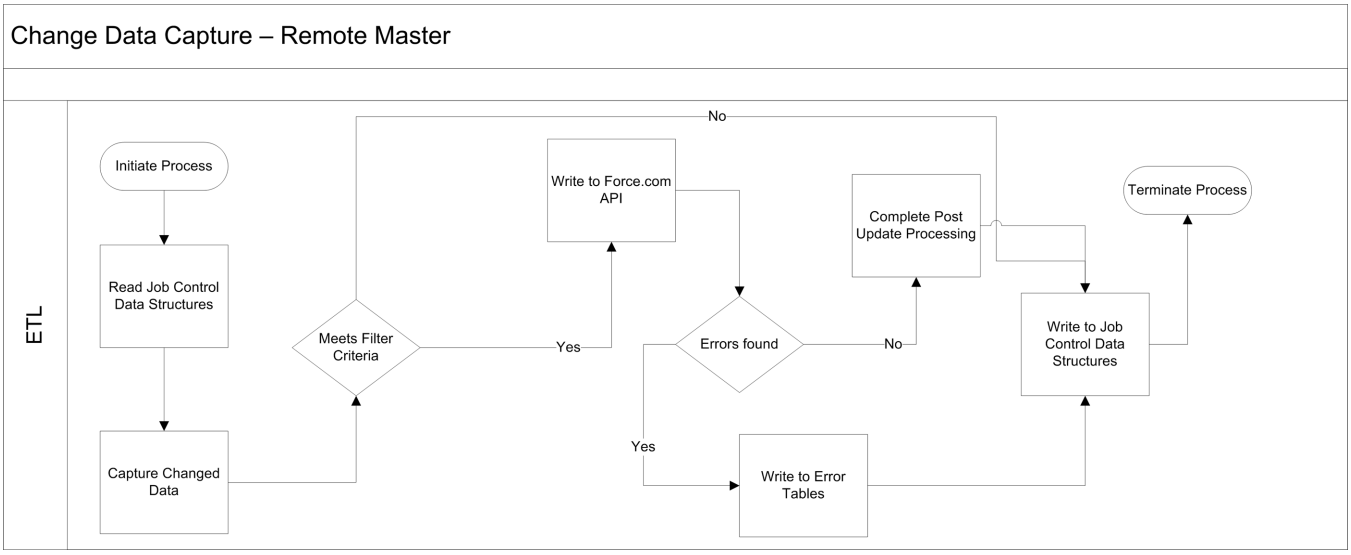
The following table contains various solutions to this integration problem.

Solution	Fit	Data master	Comments
Change data capture	Best	Remote system	Leverage a third-party ETL tool that allows you to run change data capture against source data. The tool reacts to changes in the source data set, transforms the data, and then calls Salesforce Bulk API to issue DML statements. This can also be implemented using the Salesforce SOAP API.

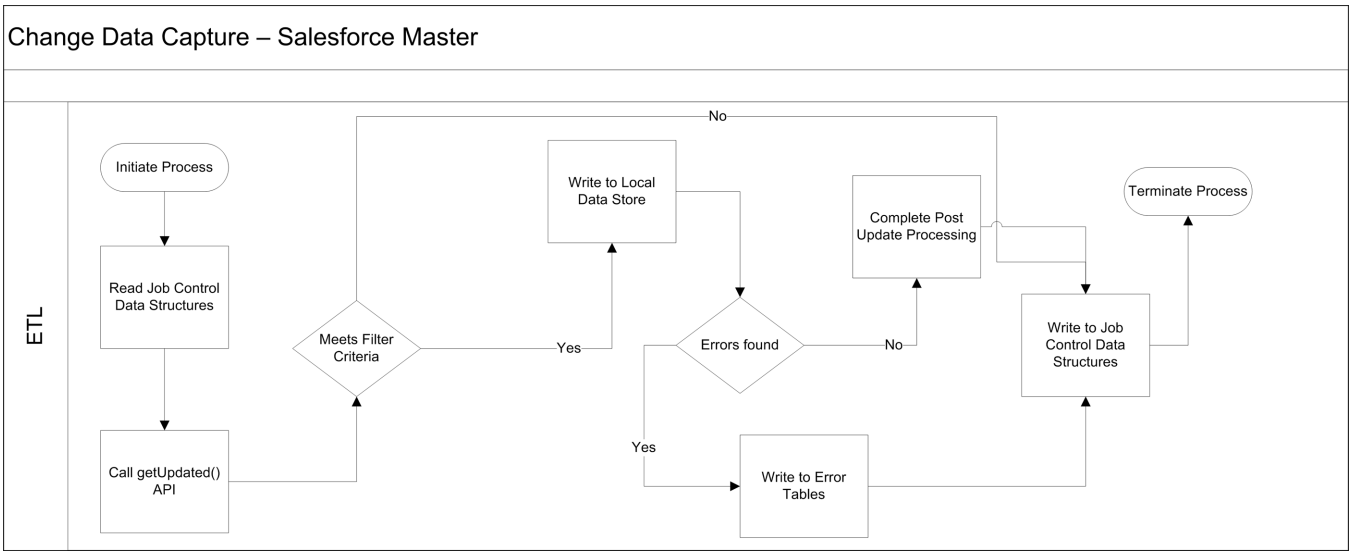
Solution	Fit	Data master	Comments
Change data capture	Best	Salesforce	Leverage a third-party ETL tool that allows you to run change data capture against ERP and Salesforce data sets. In this solution, Salesforce is the data source, and you can use time/status information on individual rows to query the data and filter the target result set. This can be implemented by using SOQL together with SOAP API and the `query()` method, or by the using SOAP API and the `getUpdated()` method.
Remote call-in	Suboptimal	Remote system	It's possible for a remote system to call into Salesforce by using one of the APIs and perform updates to data as they occur. However, this causes considerable on-going traffic between the two systems. It also requires that greater emphasis be placed on error handling and locking because this pattern has the potential for causing continual updates, which has the potential to impact performance for end-users.
Remote process invocation	Suboptimal	Salesforce	It's possible for Salesforce to call into a remote system and perform updates to data as they occur. However, this causes considerable on-going traffic between the two systems. It also requires that greater emphasis be placed on error handling and locking because this pattern has the potential for causing continual updates, which has the potential to impact performance for end-users.

Sketch

The following diagram illustrates the sequence of events in this pattern, where the remote system is the data master.



The following diagram illustrates the sequence of events in this pattern, where Salesforce is the data master.



Results

You can integrate data that's sourced externally with Salesforce under the following scenarios:

- External system is the data master—Salesforce is a consumer of data provided by a single source system or multiple systems. In this scenario, it's common to have a data warehouse or data mart that aggregates the data before the data is imported into Salesforce.
- Salesforce is the data master—Salesforce is the system of record for certain entities.

In a typical Salesforce integration scenario, the implementation team does one of the following:

- Implement change data capture on the source data set.
- Implement a set of supporting database structures, known as control tables, in an intermediate, on-premises database.

The ETL tool is then used to create programs that will:

1. Read a control table to determine the last run time of the job and extract any other control values needed.
2. Use the above control values as filters and query the source data set.
3. Apply predefined processing rules, including validation, enrichment, and so on.
4. Use available connectors/transformation capabilities of the ETL tool to create the destination data set.
5. Write the data set to Salesforce objects.
6. If processing is successful, update the control values in the control table.
7. If processing fails, update the control tables with values that enable a restart and exit.

Note: We recommend that you create the control tables and associated data structures in an environment that the ETL tool has access to even if access to Salesforce isn't available. This provides adequate levels of resilience. Salesforce should be treated as a spoke in this process and the ETL infrastructure is the hub.

For an ETL tool to gain maximum benefit from data synchronization capabilities, consider the following:

- Chain and sequence the ETL jobs to provide a cohesive process.
- Use primary keys from both systems to match incoming data.
- Use specific API methods to extract only updated data.

- If importing child records in a master-detail or lookup relationship, group the imported data using its parent key at the source to avoid locking. For example, if you're importing contact data, be sure to group the contact data by the parent account key so the maximum number of contacts for a single account can be loaded in one API call. Failure to group the imported data usually results in the first contact record being loaded and subsequent contact records for that account to fail in the context of the API call.
- Any post-import processing, such as triggers, should only process data selectively.
- If your scenario involves large data volumes, follow the best practices in the white paper *Best Practices for Deployments with Large Data Volumes*.

Error Handling and Recovery

An error handling and recovery strategy must be considered as part of the overall solution. The best method depends on the solution you choose.

Error location	Error handling and recovery strategy
Read from Salesforce	• *Error handling*—If an error occurs during a read operation, implement a retry for errors that aren't infrastructure-related. In the event of repeated failure, standard processing using control tables/error tables should be implemented in the context of an ETL operation to: – Log the error – Retry the read operation – Terminate if unsuccessful – Send a notification • *Recovery*—Restart the ETL process to recover from a failed read operation. If the operation succeeds but there are failed records, an immediate restart or subsequent execution of the job should address the issue. In this case, a delayed restart might be a better solution because it allows time to triage and correct data that might be causing the errors.
Write to Salesforce	• *Error handling*—Errors that occur during a write operation can result from a combination of factors in the application. The API calls return a result set that consists of the information listed below. This information should be used to retry the write operation (if necessary). – Record identifying information – Success/failure notification – A collection of errors for each record • *Recovery*—Restart the ETL process to recover from a failed read operation. If the operation succeeds but there are failed records, an immediate restart or subsequent execution of the job should address the issue. In this case, a delayed restart might be a better solution because it allows time to triage and correct data that might be causing the errors.
External master system	Errors should be handled in accordance with the best practices of the master system.

Security Considerations

Any call to a remote system must maintain the confidentiality, integrity, and availability of the request. Different security considerations apply, depending on the solution you choose.

- A Force.com license is required to allow authenticated API access to the Salesforce API.

- We recommend that standard encryption be used to keep password access secure.
- Use the HTTPS protocol when making calls to the Salesforce APIs. You can also proxy traffic to the Salesforce APIs through an on-premises security solution, if necessary.

See Security Considerations.

Sidebars

Timeliness

Timeliness isn't of significant importance in this pattern. However, care must be taken to design the interfaces so that all of the batch processes complete in a designated batch window.

As with all batch-oriented operations, we strongly recommend that you take care to insulate the source and target systems during batch processing windows. Loading batches during business hours might result in some contention, resulting in either a user's update failing, or more significantly, a batch load (or partial batch load) failing.

For organizations that have global operations, it might not be feasible to run all batch processes at the same time because the system might continually be in use. Data segmentation techniques using record types and other filtering criteria can be used to avoid data contention in these cases.

State Management

You can implement state management by using surrogate keys between the two systems. If you need any type of transaction management across Salesforce entities, we recommend that you use the Remote Call-In pattern using Apex.

Standard optimistic record locking occurs on the platform, and any updates made using the API require the user, who is editing the record, to refresh the record and initiate their transaction. In the context of the Salesforce API, optimistic locking refers to a process where:

- Salesforce doesn't maintain the state of a record being edited by a specific user.
- Upon read, it records the time when the data was extracted.
- If the user updates the record and saves it, Salesforce checks to see if another user has updated the record in the interim.
- If the record has been updated, the system notifies the user that an update was made and the user should retrieve the latest version of the record before proceeding with their updates.

Middleware Capabilities

The most effective external technologies used to implement this pattern are traditional ETL tools. It's important that the middleware tools chosen support the Salesforce Bulk API.

It's helpful, but not critical, that the middleware tools support the `getUpdated()` function. This function provides the closest implementation to standard change data capture capability on the Salesforce platform.

The following table highlights the desirable properties of a middleware system that participates in this pattern.

Property	Mandatory	Desirable	Not required
Event handling		X	
Protocol conversion		X	
Translation and transformation	X	X	
Queuing and buffering	X		
Synchronous transport protocols			X
Asynchronous transport protocols	X		

Property	Mandatory	Desirable	Not required
Mediation routing		X	
Process choreography and service orchestration	X		
Transactionality (encryption, signing, reliable delivery, transaction management)		X	
Routing		X	
Extract, transform, and load	X		

Example

A utility company uses a mainframe-based batch process that assigns prospects to individual sales reps and teams. This information needs to be imported into Salesforce on a nightly basis.

The customer has decided to implement change data capture on the source tables using a commercially available ETL tool.

The solution works as follows:

- A cron-like scheduler executes a batch job that assigns prospects to users and teams.
- After the batch job runs and updates the data, the ETL tool recognizes these changes using change data capture. The ETL tool collates the changes from the data store.
- The ETL connector uses the Salesforce SOAP API to load the changes into Salesforce.

CHAPTER 5 Remote Call-In

Context

You use Salesforce to track leads, manage your pipeline, create opportunities, and capture order details that convert leads to customers. However, Salesforce isn't the system that contains or processes orders. Orders are managed by an external (remote) system that needs to update the order status in Salesforce as the order passes through its processing stages.

Problem

How does a remote system connect and authenticate with Salesforce and update existing records?

Forces

There are various forces to consider when applying solutions based on this pattern:

- Does the call to Salesforce require the remote process to wait for a response before continuing processing? Remote calls to Salesforce are always synchronous request-reply, although the remote process can discard the response if it's not needed to simulate an asynchronous call.
- What is the format of the message (for example, SOAP or REST, or both over HTTP)?
- Is the message size relatively small or large?
- In the case of a SOAP-capable remote system, is the remote system able to participate in a contract-first approach, where Salesforce dictates the contract? This is required where our standard Web service APIs are used, for which a predefined WSDL is supplied.
- Is transaction processing required?
- What is the extent to which you are tolerant of customization in the Salesforce application?

Solution

The following table contains various solutions to this integration problem.

Solution	Fit	Comments
SOAP API	Best	• *Accessibility*—Salesforce provides a SOAP API that remote systems can use to: – Query data in your organization – Create, update, and delete data – Obtain metadata about your organization

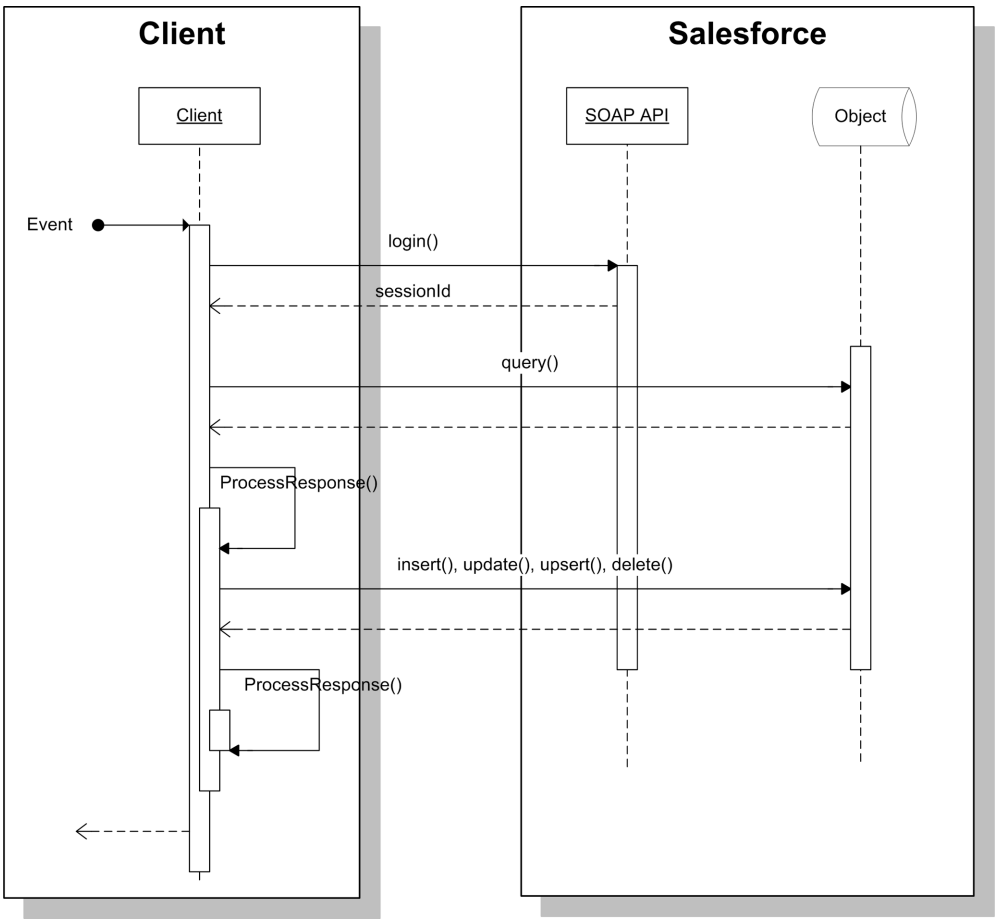
Solution	Fit	Comments
		– Run utilities to perform administrative tasks • *Synchronous API*—After the API call is made, the remote client application waits until it receives a response from the service. Asynchronous calls to Salesforce aren't supported. • *Generated WSDL*—Salesforce provides two WSDLs for remote systems: – Enterprise WSDL—Provides a strongly-typed WSDL that's specific to a Salesforce organization. – Partner WSDL—Contains a loosely-typed WSDL that's not specific to a Salesforce organization. • *Security*—The client executing SOAP API must have a valid login and obtain a session to perform any API calls. The API respects object-level and field-level security configured in the application based on the logged in user's profile. • *Transaction/Commit Behavior*—By default, each API call allows for partial success if some records are marked with errors. This can be changed to an "all or nothing" behavior where all the results are rolled back if any error occurs. It's not possible to span a transaction across multiple API calls. To overcome this limitation, it's possible for a single API call to affect multiple objects. • *Bulk Data*—For bulk data operations (more than 500,000 records), use the REST-based Bulk API.
REST API	Best	• *Accessibility*—Salesforce provides a REST API that remote systems can use to: – Query data in your organization – Create, update, and delete data – Obtain metadata about your organization – Run utilities to perform administrative tasks • *Synchronous API*—After the API call is made, the remote client application waits until it receives a response from the service. Asynchronous calls to Salesforce aren't supported. • *REST API vs. SOAP API*—REST exposes resources (entities/objects) as URIs and uses HTTP verbs to define CRUD operations on these resources. Unlike SOAP, the REST API requires no predefined contract, utilizes XML and JSON for responses, and has loose typing. REST API is lightweight and provides a simple method for interacting with Salesforce. Its advantages include ease of integration

Solution	Fit	Comments
		and development, and it's an excellent choice for use with mobile applications and Web 2.0 projects. • *Security*—The client executing REST API must have a valid login and obtain a session to perform any API calls. The API respects object-level and field-level security configured in the application based on the logged in user's profile. • *Transaction/Commit Behavior*—By default, each record is treated as a separate transaction and committed separately. Failure of one record change doesn't cause rollback of other record changes. This can be altered to an "all or nothing" behavior. • *Bulk Data*—For bulk data operations (more than 500,000 records), use the REST-based Bulk API.
Apex Web services	Suboptimal	Apex class methods can be exposed as Web service methods to external applications. This is an alternative to SOAP API, and is typically only used where the following additional requirements must be met: • Full transactional support is required (for example, create an account, contact, and opportunity all in one transaction). • Custom logic must be applied on the Salesforce side before committing. The benefit of using an Apex Web service must be weighed against the additional code that needs to be maintained in Salesforce.
Apex REST service	Suboptimal	An Apex class can be exposed as REST resources mapped to specific URIs with an HTTP verb defined against it (for example, POST or GET). Unlike SOAP, there is no need for the client to consume a service definition/contract (WSDL) and generate client stubs. The remote system requires only the ability to form an HTTP request and process the returned results (XML or JSON).
Bulk API	Optimal for bulk operations	Bulk API is based on REST principles, and is optimized for loading or deleting large sets of data. It has the same accessibility and security behavior as REST API. Bulk API allows the client application to query, insert, update, upsert, or delete a large number of records asynchronously by submitting a number of batches, which are processed in the background by Salesforce. In contrast, SOAP API is optimized for real-time client applications that update small numbers of records at a time. Although SOAP API can also be used for processing large numbers of records, when the data sets contain hundreds of

Solution	Fit	Comments
		thousands to millions of records, it becomes less practical. This is due to its relatively high overhead and lower performance characteristics.

Sketch

The following diagram illustrates the sequence of events when you implement this pattern using SOAP API. The sequence of events is the same when using REST API.



Results

The application of the solutions related to this pattern allows for:

- Remote system to call the Salesforce APIs to query the database and execute single-object operations (create, update, delete, and so on).
- Remote system to call custom-built Salesforce APIs (services) that can support multi-object transactional operations and custom pre/post processing logic.

Calling Mechanisms

The calling mechanism depends on the solution chosen to implement this pattern.

Calling mechanism	Description
SOAP API	The remote system uses the Salesforce Enterprise or Partner WSDL to generate client stubs that are in turn used to invoke the standard SOAP API.
REST API	The remote system has to authenticate before accessing any Apex REST service. The remote system can use OAuth 2.0 or username/password authentication. In either case, the client must set the authorization HTTP header with the appropriate value (an OAuth access token or a session ID can be acquired via a login call to SOAP API). The remote system then generates REST invocations (HTTP requests) with the appropriate verbs and processes the results returned (JSON and XML data formats are supported).
Apex Web service	The remote system consumes the custom Apex Web Service WSDL to generate client stubs that are in turn used to invoke the custom Apex Web service.
Apex REST service	As per REST API, the resource URI and applicable verbs are defined using the `@RestResource`, `@HttpGet`, and `@HttpPost` annotations.
Bulk API	Bulk API is a REST-based API, so the same calling mechanisms as REST API apply.

Error Handling and Recovery

An error handling and recovery strategy must be considered as part of the overall solution.

- *Error handling*—All the remote call-in methods, standard or custom APIs, will require the remote system to handle any subsequent errors, such as timeouts and the management of retries. Middleware can be used to provide the logic for error handling and recovery.
- *Recovery*—A custom retry mechanism needs to be created if quality of service requirements dictate it. In this case, it's important to ensure idempotent design characteristics.

Idempotent Design Considerations

Idempotent capabilities guarantee that repeated invocations are safe and will have no negative effect. If idempotency isn't implemented, then repeated invocations of the same message can have different results, potentially resulting in data integrity issues, for example, creation of duplicate records, duplicate processing of transactions, and so on.

The remote system must manage multiple (duplicate) calls, in the case of errors or timeouts, to avoid duplicate inserts and redundant updates (especially if downstream triggers and workflow rules fire). While it's possible to manage some of these situations within Salesforce (particularly in the case of custom SOAP and REST services), we recommend that the remote system (or middleware) manages error handling and idempotent design.

Security Considerations

Different security considerations apply, depending on the pattern solution chosen. In all cases, the platform uses the logged-in user's access rights (for example, profile settings, sharing rules, permission sets, and so on). Additionally, profile IP restrictions can be used to restrict access to the API for a specific IP address range.

Solution	Security considerations
SOAP API	Salesforce supports Secure Sockets Layer v3 (SSL) and the Transport Layer Security (TLS) protocols. Ciphers must have a key length of at least 128 bits. The remote system must log in using valid credentials in order to obtain a session ID. If the remote system already has a valid session ID, then it can call the API without an explicit login. This is used

Solution	Security considerations
	in call-back patterns covered earlier in this document, where a preceding Salesforce outbound message contained a session ID and record ID for subsequent use. We recommend that clients that call SOAP API cache and reuse the session ID to maximize performance, rather than obtaining a new session ID for each call.
REST API	We recommend that the remote system establish an OAuth trust for authorization. REST calls can then be made on specific resources using HTTP verbs. It's also possible to make REST calls with a valid session ID that might have been obtained by other means (for example, retrieved by calling SOAP API or provided via an outbound message). We recommend that clients that call the REST API cache and reuse the session ID to maximize performance, rather than obtaining a new session ID for each call.
Apex Web service	The same security considerations as SOAP API apply.
Apex REST service	The same security considerations as REST API apply.
Bulk API	The same security considerations as REST API apply.

See Security Considerations.

Sidebars

Timeliness

SOAP API and Apex Web service API are synchronous. The following timeouts apply:

- Session timeout — The session will time out if there's no activity based on the Salesforce organization's session timeout setting.
- Query timeout — Each SOQL query has an individual timeout limit of 120 seconds.

Data Volumes

Data volume considerations depend on which solution and communication type you choose.

Solution	Communication type	Limits
SOAP API or REST API	Synchronous	• *Login*—The login request size is limited to 10 KB or less. • *Create, Update, Delete*—The remote system can create, update, or delete up to 200 records at a time. Multiple calls can be made to process more than a total of 200 records, but each request is limited to 200 records in size. • Query Results Size — By default, the number of rows returned in the query result object (batch size), returned in a `query()` or `queryMore()` call is set to 500. Where the number of rows to be returned exceeds the batch size, use the `queryMore()` API call to iterate through multiple batches. The maximum batch size is 2,000 records. However, additional rules might apply, so for more information, see *Salesforce Limits Quick Reference Guide*.

Solution	Communication type	Limits
Bulk API	Synchronous	Bulk API is optimized for importing and exporting large sets of data asynchronously. Bulk API is synchronous when submitting the batch request and associated data. The actual processing of the data occurs asynchronously in the background. For more information about API and batch processing limits, see *Bulk API Limits*. • Up to 2,000 batches can be submitted per rolling 24–hour period. • A batch can contain a maximum of 10,000 records.

Endpoint Capability and Standards Support

The capability and standards support for the endpoint depends on the solution that you choose.

Solution	Endpoint considerations
SOAP API	The remote system must be capable of implementing a client that can call the Salesforce SOAP API, based on a message format predefined by Salesforce. The remote system (client) must participate in a contract-first implementation where the contract is supplied by Salesforce (for example, Enterprise or Partner WSDL).
REST API	The remote system must be capable of implementing a REST client that invokes Salesforce—defined REST services, and processes the XML or JSON results.
Apex Web service	The remote system must be capable of implementing a client that can invoke SOAP messages of a predefined format, as defined by Salesforce. The remote system must participate in a code-first implementation, where the contract is supplied by Salesforce after the Apex Web service is implemented. Each Apex Web service has its own WSDL.
Apex REST service	The same endpoint considerations as REST API apply.

State Management

When integrating systems, keys are important for on-going state tracking, for example, if a record gets created in the remote system, in order to support ongoing updates to that record. There are two options:

- Salesforce stores the remote system's primary or unique surrogate key for the remote record.
- The remote system stores the Salesforce unique record ID or some other unique surrogate key.

There are specific considerations for handling integration keys in this synchronous pattern.

Master system	Description
Salesforce	In this scenario, the remote system should store either the Salesforce RecordId or some other unique surrogate key from the record.
Remote system	In this scenario, Salesforce must store a reference to the unique identifier in the remote system. Because the process is synchronous, the key can be provided as part of the same transaction using external ID fields.

Complex Integration Scenarios

Each solution in this pattern has different considerations, when handling complex integration scenarios such as transformation and process orchestration.

Solution	Considerations
SOAP API or REST API	SOAP API and REST API provide for simple transactions on objects. Complex integration scenarios, such as aggregation, orchestration, and transformation, can't be performed in Salesforce. These scenarios will need to be handled by the remote system or middleware, with middleware as the preferred method.
Apex Web service or Apex REST service	Custom Web services can provide for cross-object functionality, custom logic, and more complex transaction support. This solution should be used with care, and you should always consider the suitability of middleware for any transformation, orchestration, and error handling logic.

Governor Limits

Due to the multi-tenant nature of the Salesforce platform, there are limits when using the APIs.

Solution	Limits
SOAP API, REST API, and custom Apex APIs	• *API request limits*—Salesforce applies a limit on the number of API calls per 24–hour period. The limit is based on the Salesforce edition type and number of licenses. For example, Unlimited Edition provides 5,000 API requests per Salesforce or Force.com license per 24 hours. For more information, see *Salesforce Limits Quick Reference Guide*. • *API query cursor limits*—A user can have up to 10 query cursors open at a time. Otherwise, the oldest of the 10 cursors is released. If the remote application attempts to open the released query cursor, an error results. For example, if sharing integration user credentials, the maximum query cursors need to be considered. Middleware may need to execute requests across multiple users in a "round robin" fashion. • *Call limits*—See Data Volumes sidebar for create, update, and query limits.
Bulk API	See Data Volumes sidebar for more information.

Reliable Messaging

Reliable messaging attempts to resolve the issue of guaranteeing the delivery of a message to a remote system where the individual components themselves might be unreliable. The Salesforce SOAP API and REST API are synchronous and don't provide explicit support for any reliable messaging protocols, per se (for example, WS-ReliableMessaging).

We recommend that the remote system implement a reliable messaging system to ensure that error and timeout scenarios are successfully managed.

Middleware Capabilities

The following table highlights the desirable properties of a middleware system that participates in this pattern:

Property	Mandatory	Desirable	Not required
Event handling		X	

Property	Mandatory	Desirable	Not required
Protocol conversion		X	
Translation and transformation		X	
Queuing and buffering	X		
Synchronous transport protocols	X		
Asynchronous transport protocols			X
Mediation routing		X	
Process choreography and service orchestration		X	
Transactionality (encryption, signing, reliable delivery, and transaction management)	X		
Routing			X
Extract, transform, and load		X (for bulk/batches)	

Example

A printing supplies and services company uses Salesforce as a front-end to create and manage accounts and opportunities. Opportunities on existing accounts are updated with printing usage statistics from the on-premises Printer Management System (PMS), which regularly monitors printers on client sites. Upon creation of an opportunity, an outbound message is sent to the PMS to register the new opportunity. The PMS stores the Salesforce ID (Salesforce is the opportunity record master).

The following constraints apply:

- The PMS is capable of participating in a contract-first integration, where Salesforce provides the contract and the PMS acts as a client (consumer) of the Salesforce service (defined via the Enterprise or Partner WSDL).
- There should be no custom development in Salesforce.

This example is best implemented using the Salesforce SOAP API or REST API.

In Salesforce:

- Download the Enterprise or Partner WSDL and provide it to the remote system.

In the remote system:

- Create a client stub from the Enterprise or Partner WSDL.
- Log in to the API using the integration user's credentials (or the opportunity owner who created the record, assuming the session ID is provided in the initial outbound message).
- Call the update operation on the Salesforce record ID provided in the outbound message and pass in the relevant field updates (usage statistics).

This example demonstrates the following:

- Implementation of a Salesforce synchronous API client (consumer).
- A callback to Salesforce to update a record (aligned with previously covered request/reply outbound patterns).

CHAPTER 6 UI Update Based on Data Changes

Context

You use Salesforce to manage customer cases. A customer service rep is on the phone with a customer working on a case. The customer makes a payment, and the customer service rep needs to see a real-time update in Salesforce from the payment processing application, indicating that the customer has successfully paid the order's outstanding amount.

Problem

When an event occurs in Salesforce, how can the user be notified in the Salesforce user interface without having to refresh their screen and potentially losing work?

Forces

There are various forces to consider when applying solutions based on this pattern:

- Does the data being acted on need to be stored in Salesforce?
- Can a custom user interface layer be built for viewing this data?
- Will the user have access for invoking the custom user interface?

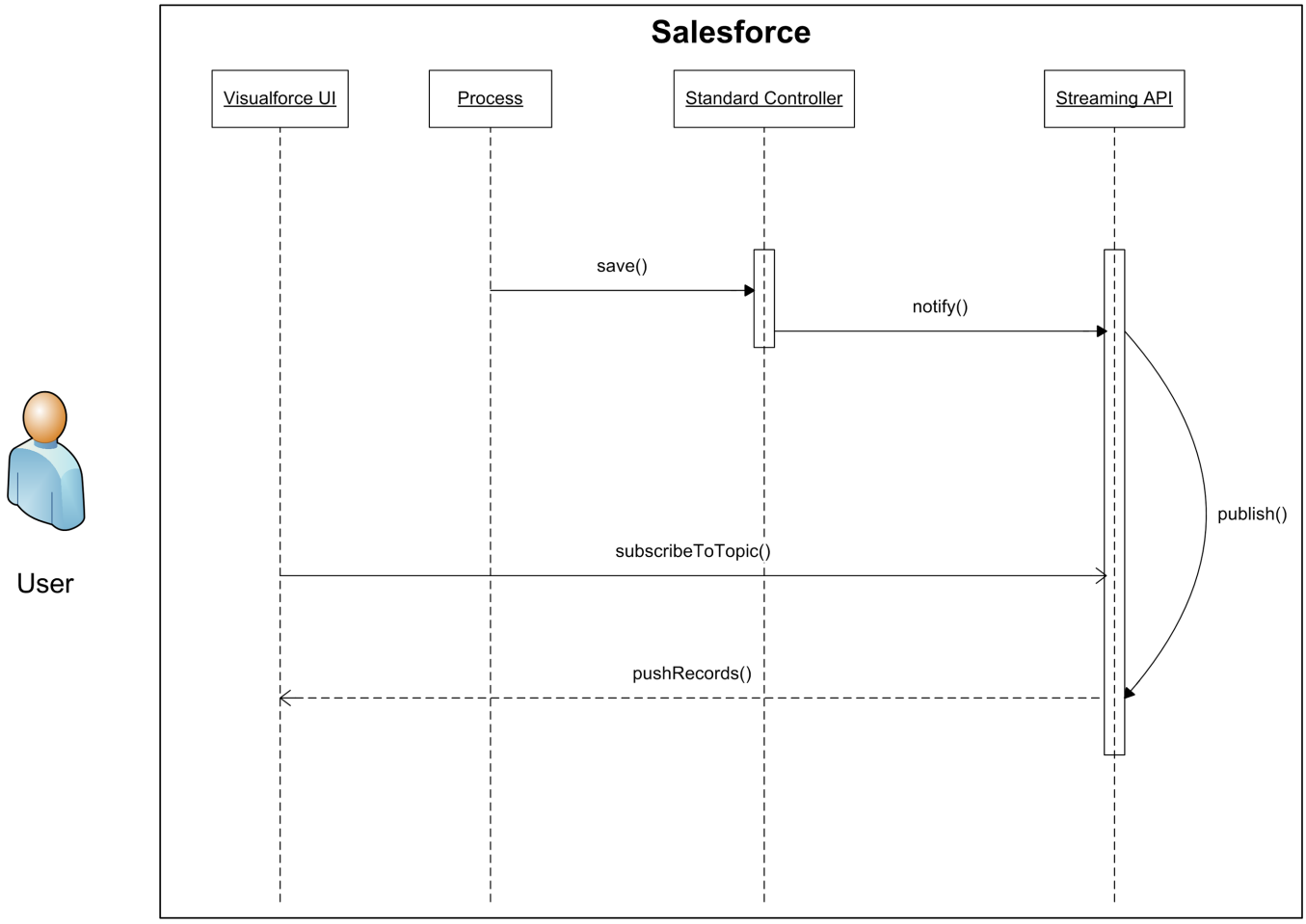
Solution

The recommended solution to this integration problem is to use the Salesforce Streaming API. This solution is comprised of the following components:

- A PushTopic with a query definition that allows you to:
  - Specify what events trigger an update
  - Select what data to include in the notification
- A JavaScript-based implementation of the *Bayeux* protocol (currently *CometD*) that can be used by the user interface
- A Visualforce page
- A JavaScript library included as a static resource

Sketch

The following diagram illustrates how Streaming API can be implemented to stream notifications to the Salesforce user interface. These notifications are triggered by record changes in Salesforce.



Results

Benefits

The application of the solution related to this pattern has the following benefits:

- Eliminates the need for writing custom polling mechanisms
- Eliminates the need for a user-initiated feedback loop

Unsupported Requirements

The solution has the following limitations:

- Delivery of notifications isn't guaranteed.
- Order of notifications isn't guaranteed.
- Notifications aren't generated from record changes made by Bulk API.

Security Considerations

Standard Salesforce organization-level security is adhered to. It's recommended you use the HTTPS protocol to connect to Streaming API. See Security Considerations.

Sidebars

The optimal solution involves creating a custom user interface in Salesforce. It's imperative that you account for an appropriate user interface container that can be used for rendering the custom user interface. Supported browsers are listed in the Streaming API *documentation*.

Example

A telecommunications company uses Salesforce to manage customer cases. The customer service managers want to be notified automatically when a case is successfully closed by one of their customer service reps.

Implementing the solution prescribed by this pattern, the customer should:

- Create a PushTopic that sends a notification when a case is saved with a Status of "Closed" and Resolution of "Successful."
- Create a custom user interface available to customer service managers. This user interface subscribes to the PushTopic channel.
- Implement logic in the custom user interface that shows alerts generated by that manager's customer service reps.

APPENDICES

APPENDIX A Resources—External

1. Hohpe, Gregor, and Bobby Woolf. *Enterprise Integration Patterns*. Boston: Addison-Wesley Professional, 2003.
2. Microsoft Corporation. *Integration Patterns (Patterns & Practices)*. Redmond: Microsoft Press, 2004.
3. IBM Corporation. *Application Integration Patterns*. IBM Corporation, 2004.
4. "Synchronous and asynchronous processes," IBM Corporation, last accessed March 18, 2013, *http://publib.boulder.ibm.com/infocenter/adiehelp/v5r1m1/index.jsp?topic=%2Fcom.ibm.etools.ctc.flow.doc%2Fconcepts%2Fcsynchf.html*.
5. "Hub and Spoke [or] Zen and the Art of Message Broker Maintenance," Enterprise Integration Patterns, last accessed March 18, 2013, *http://www.eaipatterns.com/ramblings/03_hubandspoke.html*.

APPENDIX B Resources—Salesforce

Developer Documentation

- *SOAP API Developer Guide*
- *Force.com REST API Developer Guide*
- *Salesforce Streaming API Developer's Guide*
- *Bulk API Developer Guide*
- *Apex Developer Guide*
- *Salesforce Object Query Language (SOQL) Reference*
- *Salesforce Object Search Language (SOSL) Reference*
- *Salesforce Limits Quick Reference Guide*

White Papers, Presentations, Websites

- *Best Practices for Deployments with Large Data Volumes*
- *Force.com Connect: Five Paths to Integration Success*
- *Consulting Resource Center: Large Data Volumes*

APPENDIX C Security Considerations

To be effective members of the enterprise portfolio, all applications must be created and integrated with relevant security mechanisms. Modern IT strategies employ a combination of on-premises and cloud-based services.

While integrating cloud-to-cloud services typically focuses on Web services and associated authorization, connecting on-premises and cloud services often introduces increased complexity. This section describes security tools, techniques, and Salesforce-specific considerations.

Reverse Proxy Server

"A reverse proxy server is a security device that is usually deployed in a network DMZ to protect HTTP servers . . . on a corporate intranet by performing security functions that protect the internal servers from attacks by users on the Internet."[10]

It's "a type of proxy server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client as though they originated from the proxy server itself. [A] reverse proxy acts as an intermediary for its (usually nearby) associated server(s) and only returns resources provided by those associated server(s)."[11]

In Salesforce implementations, such a service is typically provided via an external gateway product. For example, open source options such as Apache HTTP, lighttpd, and nginix can be used. Commercial products include IBM WebSeal and Computer Associates SiteMinder. These products can be easily configured to proxy and manage all outbound Salesforce requests on behalf of the internal requestor.

Encryption

Some enterprises require selected transactions or data fields to be encrypted between a combination of on-premises and cloud-based applications. If your organization must adhere to additional compliance requirements, you can implement alternatives, including:

- On-premises commercial encryption gateway services, including Salesforce's own, CipherCloud, IBM DataPower, Computer Associates. For each solution, the encryption engine or gateway is invoked at the transaction boundary by sending and receiving an encrypted payload or when encrypting or decrypting specific data fields before the HTTP(S) request executes.
- Cloud-based options, such as Salesforce Shield Platform Encryption. Shield Platform Encryption gives your data a whole new layer of security while preserving critical platform functionality. The data you select is encrypted at rest using an advanced key derivation system. You can protect data more securely than ever before. Refer to the Salesforce online help for more information.

Specialized WS-* Protocol Support

To address the requirements of security protocols (such as WS-*), we recommend these alternatives.

---

[10] "What is a reverse proxy server?," IBM Corporation, last accessed April 11, 2012, *http://publib.boulder.ibm.com/infocenter/sametime/v8r5/index.jsp?topic=%2Fcom.ibm.help.sametime.v851.doc%2Fconfig%2Fst_adm_port_rvprxy_overview_chtml*.

[11] "Reverse proxy," Wikipedia, last accessed April 11, 2012, *http://en.wikipedia.org/wiki/Reverse_proxy*.

- Security/XML gateway—Inject WS-Security credentials (IBM WebSeal or Datapower, Layer7, TIBCO, and so on) into the transaction stream itself. This approach requires no changes to application-level Web services or Web service invocations from Salesforce. You can also reuse this approach across the Salesforce installation. However, it requires additional design, configuration, testing, and maintenance to manage the appropriate WS-Security injection into the existing security gateway approach.
- Transport-level encryption—Encrypt the communication channel using two-way SSL and IP restrictions. While this approach doesn't directly implement the WS-* protocol by itself, it secures the communication channel between the on-premises applications and Salesforce without passing a username and password. It also doesn't require changes to Salesforce-generated classes. However, some on-premises Web services modifications might be required (at either the application itself or at the middleware/ESB layer).
- Salesforce custom development—Add WS-Security headers to the outbound SOAP request via the WSDL2Apex utility. This generates a Java-like Apex class from the WSDL file used to invoke the internal service. While this requires no changes to back-end Web services or additional components in the DMZ, it does require:
  - an increased build and test effort
  - a relatively complex and manual process to hand-code the WS-Security attributes (including XML serialization within the Apex code)
  - a higher long-term maintenance effort

  Note: The last option isn't recommended due to its complexity and the risk that such integrations need periodic reviews based on regular updates to Salesforce.

INDEX

B

Batch data synchronization pattern 28

C

Categories of patterns 3

M

Middleware terms and definitions 5

O

Overview of integration patterns 1

P

Pattern
- approach 3
- overview 1
- selection guide 3
- summary 2
- template 1

Patterns
- batch data synchronization 28
- remote call-in 34
- remote process invocation—fire and forget 17

Patterns *(continued)*
- remote process invocation—request and reply 9
- UI update based on data changes 43

Purpose and scope 1

R

Remote call-in pattern 34
Remote process invocation
- fire and forget pattern 17
- request and reply pattern 9

Resources
- external 46
- Salesforce 47

S

Security considerations 48
Summary of patterns 2

T

Template, pattern 1

U

UI update based on data changes pattern 43