



DATA INTEGRATION BEST PRACTICES

JACOB HORBULYK

DATA INTEGRATION BEST PRACTICES

JACOB HORBULYK



Meet the author



JACOB HORBULYK

PRE-SALES & PROFESSIONAL SERVICE SOFTWARE ENGINEER

Jacob Horbulyk is a pre-sales & professional service software engineer at elastic.io. Over the last two years, he has led a number of integration projects across a wide range of industries from education to e-commerce. During that time, he has made all the possible mistakes so that others won't have to, even if it makes others look smarter. Currently, he is doing what he can to make potentially bland and monotonous integration projects run as quickly and as seamlessly as possible by mastering how to

take apart an integration problem and put together a generic, reusable and correct solutions. On the side, he mentors students, helps them succeed, watches them when they fail and then builds the tools they need to be successful the next time around. He hopes that you can learn from those mistakes with the content that is inside this book.

When he isn't working at elastic.io, he is learning German so that he can fit into his new home in Bonn, taking road trips on the weekend and playing board games to stave off senility.

About elastic.io



elastic.io is an industry-first microservices-based hybrid integration platform as a service (iPaaS) that empowers IT organizations to accelerate enterprise digital transformation. In 2017, elastic.io became part of mVISE Group, a German public company with over 15 years of consultancy and project experience in IT.

elastic.io's primary motivation is to support large corporations and mid-size businesses alike in their digital strategy initiatives by helping them spend less time on gathering data together across the entire organization, and instead, have enough time and resources to focus on using this data to improve business operations or to develop new products and services.

Contents

Introduction	1
DATA INTEGRATION BEST PRACTICES	1
PART 1: INTEGRATION PROBLEMS	2
Chapter I	3
IDENTIFYING AND SOLVING INTEGRATION PROBLEMS	3
Technical mechanics vs business rules.....	4
Technical mechanics problem: Moving data in and out of systems	4
Chapter II	6
MECHANISMS TO DETECT DATA CHANGES	6
Why is detecting data changes important?	7
Polling by timestamp	7
Data push from place of change	8
Bulk extract with delta detection.....	8
Chapter III	10
CONNECTION POINTS WEB SERVER API vs DATABASE	10
Web server API.....	11
Database views/database tables	12
Chapter IV	13
DATA DUPLICATION AND ID LINKING	13
Point to point ID linking	14
Dedicated system for storing ID links	15
Universal ID	15
Chapter V	16
DATA TRANSFORMATION	16
Data transformation	17
Data mapping	18
Types of fields to map.....	18
One directional vs bi-directional transformations	19
Master schemas & master data systems.....	19
Chapter VI	21
HANDLING DATA INTEGRATION ERRORS	21

Maintaining consistency	22
Conflict resolution	23
Hierarchy preservation	24
PART 2: DIFFERENT TYPES OF INTEGRATION	25
Chapter VII.....	26
REQUEST-REPLY VS ASYNCHRONOUS INTEGRATION.....	26
Request-reply (synchronous)	27
Asynchronous integration	28
Chapter VIII	30
WHAT SYSTEMS MOVE THE DATA?.....	30
When systems move data without intermediaries	31
Integration layer / iPaaS option	32
Chapter IX.....	34
DIFFERENCES IN WHAT IS BEING INTEGRATED	34
Integration with a shared authentication mechanism	35
Integration between different parts of a system.....	36
Event propagation between systems.....	36
Data synchronization between systems.....	37
PART 3: INTEGRATION BEST PRACTICES MOVING FORWARD.....	39
Chapter X	40
INTEGRATION PROJECTS: THE REAL AND HIDDEN COSTS.....	40
Creation costs	41
Operational costs	42
Retirement costs.....	42
Chapter XI	43
HOW TO SET THE FRAMEWORK FOR DATA INTEGRATION PROJECTS.....	43
Describe integrations better	44
Consider using an iPaaS / integration layer	45
Chapter XII	47
SEPARATION OF ENVIRONMENTS AND LOG COLLECTION	47
Separation of environments	48
Log collection	49
Conclusion	51

Introduction

DATA INTEGRATION BEST PRACTICES

One would think that the topic of data integration and application integration has been so widely covered that there really should be no questions left. And yet, we are regularly asked questions both from the IT people and the business users. So, we decided to answer the most frequently asked questions in this e-book on data integration best practices.

This e-book will focus both on the technical side of the data integration as well as answering the questions that drive the business decisions regarding the selection of a suitable tool and an appropriate software vendor. However, it is equally suitable for those who prefer to handle all integration work on their own as well as for those who decide to work with a third-party solution.



PART 1

**INTEGRATION
PROBLEMS**



Chapter I

IDENTIFYING AND SOLVING INTEGRATION PROBLEMS



Before we can engage with the data integration best practices, it is important to understand what issues both IT specialists and business users are faced with when trying to integrate two or more systems, be it business applications, databases or even entire platforms. Hence, this is where we are going to start from.

TECHNICAL MECHANICS VS BUSINESS RULES

Fundamentally, all integration problems can be broken down into two main categories: the problems that concern the pure technical mechanics of integration and the problems that are related to the correct application of business rules. Technical problems are generally lower level in nature and answer the question of “How?”, while business problems are normally higher level in nature and answer the question of “What?”

To give you an example, the desire to have customer information copied from the CRM into the ERP would be a business problem. The technical problem is, on the other side, to identify and pick which APIs to interact with to solve the business problem.

It is important to keep in mind that the descriptions of the requirements associated with the problem on the business level may not necessarily line up with the technical details. For instance, a business expert may see and describe requirements involving contacts where contacts are associated with addresses. At the same time, from a technical standpoint, contacts and addresses may exist as different objects within the system.

In plain terms, an IT person will see contacts and addresses as two absolutely separate items, while a business person would associate a given contact with a given address, seeing them as one whole piece of information.

This doesn't mean, though, that one standpoint is better than the other. On the contrary, the individuals who can answer technical problems are in general not the individuals who also should answer business problems, and vice versa.

TECHNICAL MECHANICS PROBLEM: MOVING DATA IN AND OUT OF SYSTEMS


In the first two chapters of our e-book on data integration best practices, we are going to tackle the technical specifics of moving data in and out of business applications, databases, or platforms.

On a very high level of abstraction, one can sum up the process of data integration the following way: In order for data to move between systems at some point, the data will go from being inside the system to, obviously, not inside the system (or vice versa).

These operation can be categorized into read operations that don't change data and write operations that do change data, and there are multiple mechanics that can make such operations occur. Polling by a timestamp would be one example, bulk extra with delta deletion

– another. The selection of an implementation mechanic is, in general, independent of the business requirements that drive the particular integration need.

Ultimately, the selected mechanic must have the following properties:

- 
- It must be sufficiently performant
 - There must be a commitment from the software vendor that this mechanism will continue to exist after any future updates to the software
 - It must be allowed in the sense that your IT security team is not opposed to the permissions granted to that mechanism
 - It must be capable of reading and modifying the data as required by the applicable business rules
 - Last but not least, it must not break the actual business application / database / platform

In the next chapter we will be going through several mechanisms for the detection of data changes and the different technical sub-decisions that need to be made in this respect.

Chapter II

MECHANISMS TO DETECT DATA CHANGES

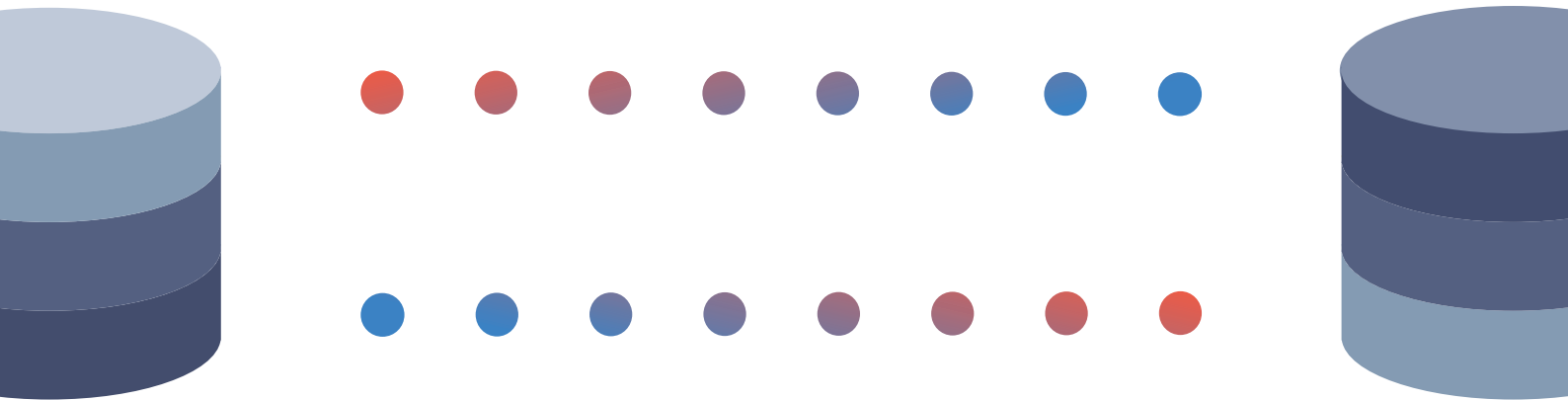


In the previous chapter of Data Integration Best Practices, we defined that all problems related to data integration can be essentially broken down into large categories: The ones that have something to do with technical mechanics and the ones that are connected to business rules.

So, in this chapter, we will review one of the purely technical questions, namely how to detect any changes in data. Within the scope of this topic, it is important to note that the technical decisions that will be covered in this and the next few chapters are not necessarily dictated by what type of system you're dealing with: cloud-based, on-premises systems or hybrid.

WHY IS DETECTING DATA CHANGES IMPORTANT?

When synchronising large amounts of data and/or working with dynamically changed data, it's highly valuable to only synchronize the data that was actually changed. Otherwise, re-syncing all data records every time might, for example, consume a lot of computing resources and eventually lead to an unnecessary system overload.



This means that detection of only the data that was changed is a crucial property of the systems. The following sections describe the different strategies to do such detection.

POLLING BY TIMESTAMP

For this strategy to work, the system that stores data, e.g. a CRM application, also stores and maintains a last modified date along. This date describes when a given record or entity has last been modified including creation and possibly, even deletion. We also covered this feature in our somewhat older blog article [“6 Characteristics That Make APIs Fit for Application Integration”](#).

When this strategy applies, there is some integration flow somewhere that wakes up on a scheduled interval and looks for all records/entities that have been updated since this flow was last run. It then reports only those records as changed.

In order for this strategy to work, the integration flow that wakes up must either be able to store the information when it was last run or otherwise reliably learn when it was last run.

Pros:

- Wide support for the strategy
- Has built-in failure tolerance as queries can be repeated if the request has failed or was otherwise somehow interrupted

Cons:

- Not all systems support this feature
- Delay between modification occurring and change detection, unless the integration flow is scheduled to wake up every millisecond
- Not quite resource efficient as many checks for data changes will be empty if the dataset does not change often

DATA PUSH FROM PLACE OF CHANGE

In this strategy, the system that stores data can send modifications made to data to other systems as the modifications occur. For example, it could publish a REST/SOAP message over HTTP(S), send an email or publish some message to a service bus.

Pros:

- Data changes can be detected faster than with polling by timestamp
- Resources are used more efficiently as there will be no “blank” runs

Cons:

- Again, not all systems support this feature
- If a message to be published got lost, it cannot be repeated. Therefore, you need to know how to recover from failures in the push infrastructure

BULK EXTRACT WITH DELTA DETECTION

This strategy is the last resort strategy when the above two strategies can not be applied. In this strategy, in addition to all data records being stored inside the system, there exists a hashed copy of all records along with their IDs outside the system. A simple example would include data stored inside an CRM application and the copy of that data stored in some integration middleware – i.e. outside the CRM application.

At a regular interval, a scheduled integration flow will wake up, read all data records and compute their hash – in other words, their unique code. Then it will compare the value of each such hash with the stored hash for each corresponding record. If the hashes do not match, it will report the record as changed and store the new hash.

Pros:

- This feature is universally supported
- It might be the only way to detect deletion of a data record, because not many systems have a built-in mechanism for that
- Can be extended to detect field level changes, meaning that it will detect only the data changes that are of a particular interest / relevance for the given integration scenario

Cons:

- It's very resource intensive and, as a consequence, cannot be run as frequently
- Requires an additional datastore
- There might be a risk of falsely reporting deletions if the actual system is down for some reason and the integration flow is trying to read records at exactly the same moment

In order to perform any of the aforementioned “extractions”, there must obviously be a way to connect to a system in the first place. Indeed, even though we often talk about disparate systems and data silos in the context of application integration, no system is designed as a sealed cocoon.

So, in the next chapter, we'll briefly review the two ways to connect to a system, going through the pros and cons of each.

Chapter III

CONNECTION POINTS WEB SERVER API VS DATABASE



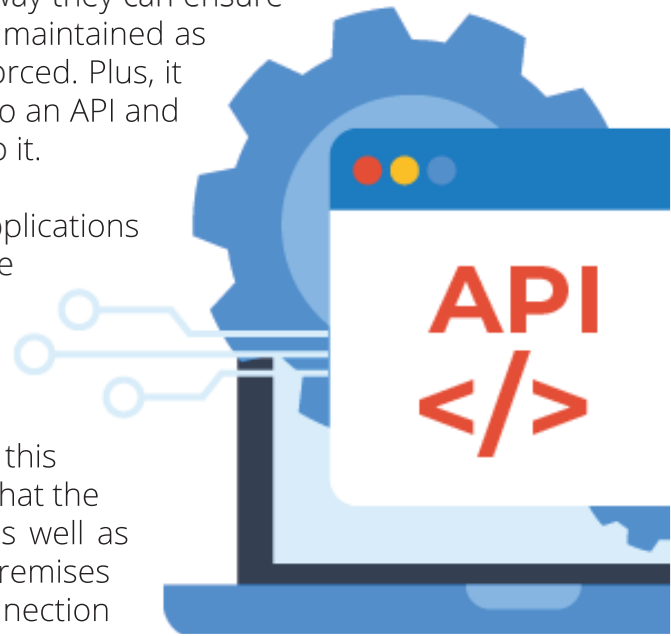
In the previous chapter, we went through various techniques to detect data changes in systems. This is essential if we want to have updates to data records as soon as they occur so that we can, for example, send orders to an address that is actually the current one. As mentioned before, in order for the data exchange to happen, there must be a way to connect to a system in the first place. Here there is a difference between on-premises and cloud-based systems.

Cloud-based software as a service (SaaS) providers usually only expose APIs on the web server, through which you can interact with a database. This way they can ensure with higher level of certainty that API behavior can be maintained as the product develops and that business rules are enforced. Plus, it is considerably easier to build a secure mechanism into an API and protect the database simply by not giving any access to it.

Then there are some custom built or specific purpose applications that only expose database connections – but not the API.

Last but not least, an on-premises system would often include both a web server that exposes an API and a corresponding database. And indeed, since this application resides on-premises, it is easier to ensure that the same level of security standards applies to the API as well as the database. Which means that in the case of on-premises systems, both API and database may provide connection opportunities.

Now let's review their pros and cons.



WEB SERVER API

Exposing a web server API is often the only way for web-based SaaS.

Pros:

- Enforces the largest set of business rules
- Provides the highest level of abstraction
- Enables the most granular permissioning
- Can have mechanisms to handle demand spikes
- The chances that a published web server API will work after one or two years are quite high – or at least you will be notified when the changes are made

Cons:

- This mechanism is built on top of HTTP(S) which is not exactly reliable, because an HTTP(S) call may fail. It is also pretty verbose, meaning that it requires more data to be transmitted than necessary
- The set of operations you can do through a web server API is rather limited compared to the set of all operations that are possible
- Has performance overhead

DATABASE VIEWS/DATABASE TABLES

Pros:

- Can be more performant
- Often database expertise is more plentiful than application expertise
- Often results in fewer bugs, because there are simply fewer layers of logic you're going through, and it is simple to the extent that it's hard not to understand it correctly
- Can be used when web server APIs don't exist or are otherwise not very good

Cons:

- Since most of the time, this is not an official API, it is subject to unexpected changes during version upgrades, updates, etc, which might result in a disruption of an integration flow.

It is also important to note that within this arrangement, database views provide more abstraction than database tables. To understand this, imagine you have a contacts table and an address table as separate tables. In order to read data from them, you have to know that there is a contacts table and an address table in the first place. Then you would need to know how they are related or do the join(s) yourself. Whereas with the database views, this might be several tables combined and presented as one.

In the next chapter, we will talk about yet another integration challenge related to technical mechanics. We will go through the techniques to effectively maintain and keep up-to-date the data that is "duplicated" across several various systems.

Chapter IV

DATA DUPLICATION AND ID LINKING



Let us first make it clear what ID linking is about. To do that, let's take a CRM system as an example.

In general, the possibility that there are more than one person in the system that have the same first and last names is quite high. And it is inevitable that sometimes, some pieces of information about these people will change. So, how would a CRM system "know" that it is the Jane Smith in Branchville, Virginia, whose billing address needs to be updated, and not the Jane Smith in Shoreline, Washington? Quite simple – the CRM has assigned a unique ID to each Jane Smith, by which it can "distinguish" one from the other.

But what if we have two CRM systems – one that is used in, say, a parent company and the other one in a daughter company? There can be a business scenario where both of them need to store the same information – both Jane Smiths – but each would assign its own ID just because they are designed this way.

This is where ID linking comes into play. When the same record is stored on more than one system, there must be some way to correlate these records between systems. This is essential in the case when you have data that can change, because then you need to find a way to distinguish between an existing data that has changed and a completely new data that is just somewhat similar to the existing data. And here are some techniques to solve these problems.

POINT-TO-POINT ID LINKING

In this solution, if data is replicated between systems, one (or more) of the systems is responsible for storing other systems' ID for the related object. The idea behind it is that, taking the scenario above, one CRM system would store the ID of another CRM. In addition to that, there is a separate system which has the sole purpose of keeping track of these ID links.

Pros:

- It's easy to set up and frequently available
- Provides a convenient mechanism for end user to modify links
- This is a natural way of storing links. After all, the life cycle of a link is logically the same as the life cycle of a linked object

Cons:

- Conceptual complexity grows quite quickly as use cases and additional systems are added
- This method requires adding write permissions to integration flows that otherwise wouldn't require it
- It also requires the systems to have available fields for foreign IDs – that is, for the IDs of other systems – or have configurable schemas. This implies having space to store this additional information. However, you may not have this capability in terms of the way how the system is designed.
- It is easy to inadvertently mess up data integrity
- It must be designed per pair of systems

DEDICATED SYSTEM FOR STORING ID LINKS

In this solution, a separate database is set up which is responsible for correlating IDs between objects.

Pros:

- It's conceptually simple
- Scales as number of systems grow

Cons:

- There is yet one more database to maintain
- It's difficult for users to visualize or modify linked IDs

UNIVERSAL ID

In this solution, when a record such as a new customer or a new product is created, they are assigned a globally unique ID that is referenced by all systems. For this technique to be possible, there must be just one system that is responsible for generating IDs for all records. It would also be responsible for every other system to learn these IDs.

Pros:

- It's conceptually simple

Cons:

- It requires LOTS of human coordination
- Eliminating duplicate or redundant information becomes hard
- All systems must be designed to accommodate a universal ID

In the next chapter of Data Integration Best Practices series, we will have a look at the problem that is related essentially to the question of business rules: Data transformation. We will review the three categories into which systems can be classified based on how they handle their schema. In addition to that, we will review what master schemata are and how they are related to master data systems.



Chapter V

DATA TRANSFORMATION



So far, we have addressed only the problems that concern the pure technical mechanics of integration. Now it's time to review the challenges that are related to the correct application of business rules. These are, as we stated in the first article of our blog series, are typically of higher level in nature and answer the question of "What?"

DATA TRANSFORMATION

Almost all systems have some sort of schema, by which we mean the structure of the information that exists in a system. In general, schema includes two parts:

- A list of object types within the system
- A set of rules which describe the structure of a given object of a given type

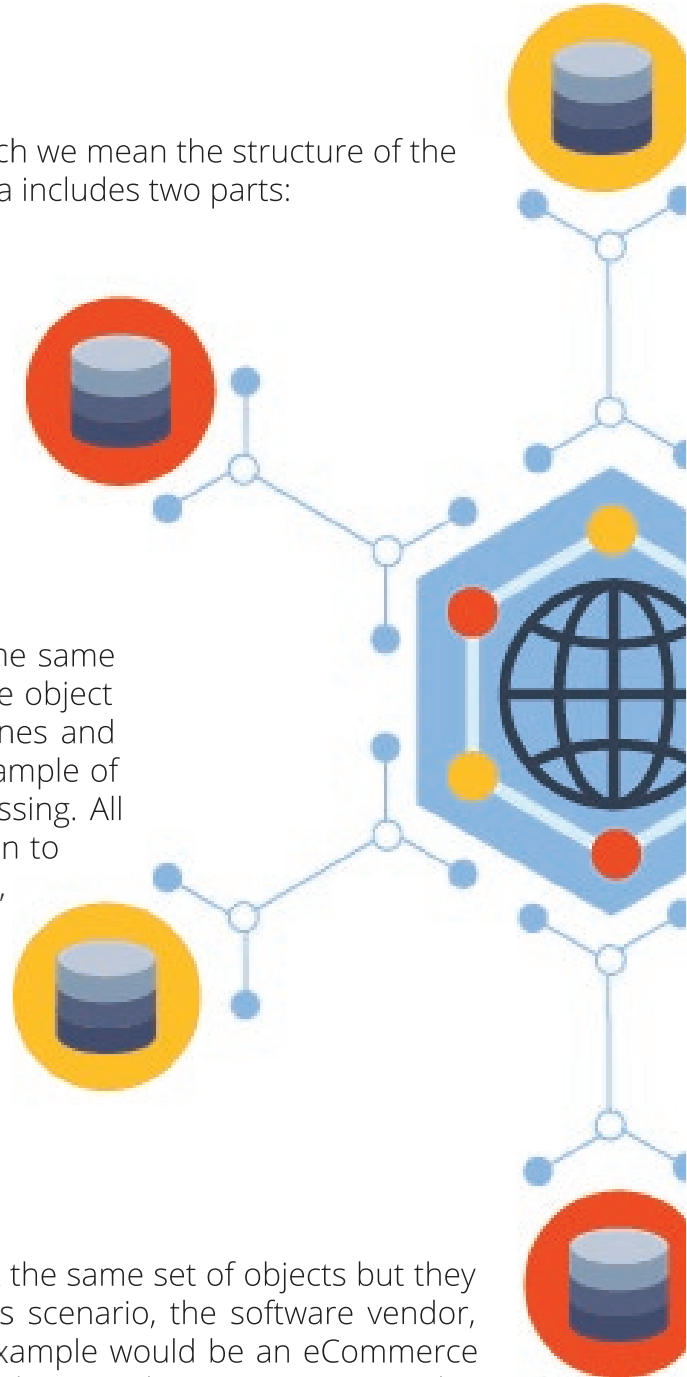
Generally, systems can be classified into one of three categories based on how they handle their schema.

ZERO CUSTOMIZATION

In the first category, all customers of a system get the same set of objects and the same set of rules that describe object structure. In this scenario, the software vendor defines and delivers this objects list and set of rules. A typical example of such a system would be credit card payment processing. All credit cards must contain the same set of information to allow for transactions between different banks. Hence, it doesn't matter what system you use for credit card payment processing. You are never going to have something other than these specific objects and you'll never have any customizations to those objects, such as "process this custom credit card".

PARTIAL CUSTOMIZATION

In the second category, all customers of a system get the same set of objects but they can customize the structure of these objects. In this scenario, the software vendor, too, is the one which sets this objects list. A good example would be an eCommerce application. Here, you will have such objects as products, orders, customers, and so on. There is no particular reason to invent some other type of object. However, you can customize certain information related to, say, the product object, depending on what you are selling. For example, a shop selling computer parts would have different adjectives to describe their products than a shop that sells soap. These adjectives will be part of the product information.



FULL CUSTOMIZATION

Last but not least, in the third category, customers have the ability to customize everything: the set of objects as well as the structures for these objects. An example of that would be a CRM system that a travel company initially bought for customer management. Since the software vendor ensures high levels of customization, the company decides to adapt it to its travel management needs as well. And so, they will create new objects such as a trip object, a destination object, and a ticket object.

DATA MAPPING

When we need to sync data between systems that have been developed independently of each other, the schemas for these systems will be different. This means that some transformation needs to occur to the structure as it moves between systems.

Understanding how this data needs to be transformed – in other words, what fields in the first schema need to map to the fields in the second schema – is generally a business level problem. Furthermore, this problem needs to be solved per pair of systems.

Additionally, if either system does not fall into the ‘zero customization’ category described above and it has undergone some customizations, then the generated mapping rules will be applicable only to these specific systems purchased by this one specific company. In other words, no other company will be able to use the same mapping rules unless they customize the systems in question in exactly the same way.

TYPES OF FIELDS TO MAP

One of the things that is often overlooked when describing the requirements for a mapping is that not all fields can be mapped as “simple” fields. By simple fields we mean information that is a series of strings, numbers, booleans, timestamps, and so on, that exist as pieces of information in their own right, and could be held in structures or arrays. An example of such information would be Name, Email or Address on a contact in a CRM system.

Now, in addition to the aforementioned “simple” fields, let’s consider the following types of information that could be stored as part of an object:

- **Static Enums:** These contain information that indicates a selection from a fixed list of values. For example, consider Gender or Office Location Code on a contact. Knowing that a contact is “M” and works in the office “DE01” already provides some information about the contact. However, knowing the list of options and what options exist provides useful context.
- **Dynamic Enums:** These are information that isn’t fully free form but selected from a range of values with the possibility of adding new values. Following the same example use case, on a contact in a CRM system, this could be Department or Job Title.
- **The ID of the object itself:** For example, Contact ID.

- The ID of related objects: For example, Manager ID, Company ID, Active Subscription(s) ID(s), and so on.

THE MAPPING RULES

We've already previously talked about the ID linking problem so we will focus on mapping rules for simple fields, static enums and dynamics enums.

- Simple fields are easiest to map: one can either move the data as is or transform the data based on some fixed rules that don't require additional inputs.
- Static enums often require an additional dictionary to perform the mapping. This dictionary is used to convert the value of one possibility in one system to the value in the other system. A simple example would be converting "Female" to "Ms."
- Dynamic Enums are often transferred as they are, but this transformation is slightly more tricky because of the fact that sometimes one needs to add new options to the range of existing valid values before transferring data. For example, imagine we are moving profession information into a system where each assignable profession must be explicitly added to a list of allowed options. So far, that system has only seen "engineer", "accountant" and "product manager". If we now want to add a contact with profession "lawyer", we must first add "lawyer" to the list of allowed professions.

ONE DIRECTIONAL VS BI-DIRECTIONAL TRANSFORMATIONS

If data flows in only one direction, then the transformation logic has to be defined in only one direction. A business case example would be an SMS push after the dispatchment of an order.

If data flows in two directions, then the transformation logic has to be defined, of course, in both directions. Moreover, the transformations described in the logic must be reversible. Otherwise, there is only one way in which you can transport data. A very simple example of that would be two CRM systems. One of them accepts German letters and another one doesn't. For a German address to be able to sync between these two, the transformation logic for the address field must convert the letter "ß" in the German word "Strasse" (= "street") into double "ss" and backwards.

MASTER SCHEMAS & MASTER DATA SYSTEMS

When the number of data systems being synchronized is larger than 2, the number of connection pairs increases exponentially because of the many possible pairing combinations between these systems. In order to keep the number of pairs under control, one solution is to create a master schema or add a master data system. This technique requires each schema to be mapped to the master schema only once, instead of being mapped to each other system.

The idea behind a master schema is that one defines a schema that meaningfully represents objects in all systems. Transformations are then written between a given systems schema and

the master schema. This reduces the number of transformation pairs considerably.

A step beyond master schema is to create a master data system. The idea behind a master data system is that in addition to data being stored in systems that manipulate or otherwise use the data, the data is also stored in the master data system. All integrations are then written to work between each system and the master data system.

So far, we have covered the problems that concern the pure technical mechanics of integration and the problems that depend on the correct application of business rules. In the next chapter, we will have a look at the problems that combine both. To be more precise, we will focus on how to maintain data consistency. We will review the types of errors that can occur during an integration and the ways how to potentially handle them.

Chapter VI

HANDLING DATA INTEGRATION ERRORS



As we mentioned at the outset, all integration problems can be broken down into two main categories: problems relating to the technical mechanics of integration and problems relating to the correct application of business rules.

There are, however, certain types of problems that originate in both. In this chapter, which is also closing part one of our e-book, we are going to review them and suggest a few ways to deal with them.

MAINTAINING CONSISTENCY

If we receive a large number of errors, we need to find a way to sort them out depending on the type. The following errors can occur during an integration.

AUTHENTICATION PROBLEMS

These errors occur when the credentials provided to perform an integration are invalid. This can happen to a previously valid account as well when a token has expired, a password changed, or a system user who is trying to access it doesn't exist in the system anymore. In order to solve these problems, the user will need to provide new credentials, or the permissions associated with a credential need to be expanded.

AUTHORIZATION PROBLEMS

These errors occur when the credentials provided to perform an integration lack the permissions to do the required operations. When systems support the concept of user specific permissions or user roles, API access usually happens in the context of a user or based on a list of granted/not granted permissions. This means that actions done through the API will be allowed or disallowed based on the permissions that the API user has or that are on a permission whitelist. In general, based on the principle of least access, this list of permissions should be as small as possible. However, if the permissions granted are insufficient, then authorization problems like this will occur. The resolution to this problem is to expand these permissions.

BUSINESS VALIDATION ERROR

This type of error happens when a system rejects a request because such a request would violate the enforced business rules. These are often the result of incorrectly formulated business rules. Another reason can be that system A has different validation requirements than system B. Let's take as an example a fictional address. If you enter such an address into your CRM system, there is a high probability that it would accept this address because it doesn't have the means to check its validity. It would look quite differently, though, if you were to enter the same address in, say, your navigation app.

One way to solve this problem is to ensure that all systems storing data have the same validity requirements. Another way is to only try to write data to a system once a record passes the

validity requirements.

SYSTEM AVAILABILITY PROBLEMS

This type of error occurs when an integration can't be completed because a system goes down. In order to solve this problem, the system that is down must be, obviously, brought back online. Some other possible causes of this include DNS, SSL and Disk space problems. These need to be handled accordingly.

TRANSIENT PROBLEMS

In this case, some sort of temporary issue appears that is self-corrected. You can compare it with accessing a website when you send your request and it times out. If you try sending the same request a minute later, you are most likely to succeed. In general, it is possible to deal with this type of problems by setting up some sort of an auto-retry for such cases.

SYSTEM THRESHOLD PROBLEMS

Most systems have a finite capacity, either in terms of a number of allowable API calls or computer resources. If those resources are exceeded, then integrations will stop working.

DEVELOPER BUGS

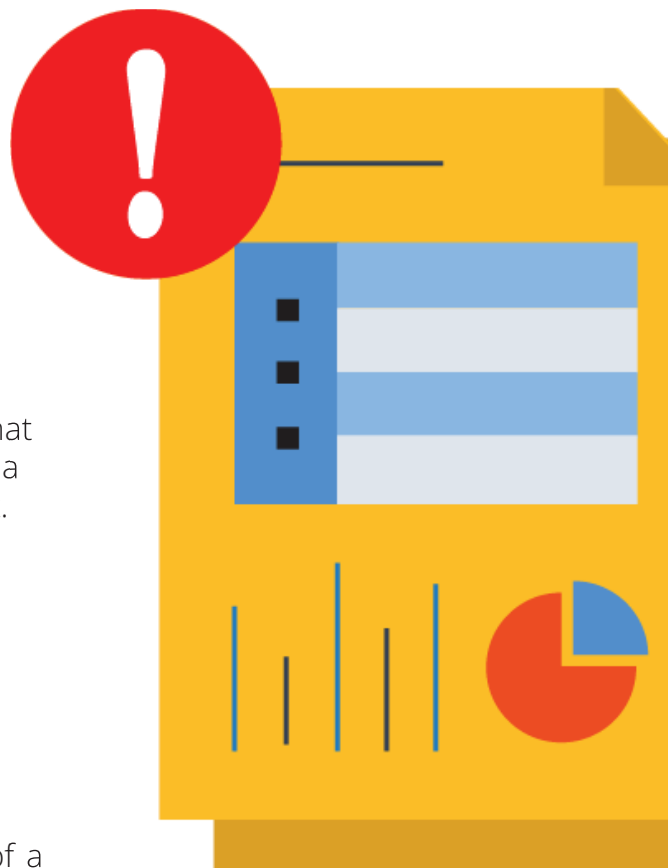
As the title says, such errors occur because the developer failed to take into account some edge cases, or in other words, an unusual but a valid set of inputs.

CONFLICT RESOLUTION

Sometimes you would need to configure two systems to mirror each other, so that if you enter certain data, it will be automatically copied into another system. In such a scenario, it is possible that users make conflicting edits in these two systems.

There are a number of ways to avoid such errors. One way is to set system A to be the source of truth and system B to be the mirror. In other words, if there is any discrepancy between the systems, then system A is considered accurate. It is possible to set this for either the entire record or you can set this for each field on the record.

Another option is to allow both to be the source of truth. However, if conflicting changes



are then created, you would need to find a technique to resolve the conflict. This can be complicated by the fact that two systems may record semantically equivalent data in different forms. Unfortunately, there is no generic solution to this potential problem.

HIERARCHY PRESERVATION

Often there is linking between objects in a system. For example, in a CRM, a contact would likely be linked to a company. This means that in order to create a customer that is linked to a company, the company must be created before or at the same time as the customer. It is important that this is considered into the design of any integration solution. In general, this can be solved by understanding that the relationships between objects can be described as a tree and that all “parents” must be synced before “children”.

At this point, we have covered the main problems you would typically face when trying to integrate and sync data between different systems. The next part of our e-book will be all about different types of integration. In the first chapter, we will have a look at the request-reply and async integrations, their pros and cons.



PART 2

DIFFERENT TYPES OF INTEGRATION



Chapter VII

REQUEST-REPLY VS ASYNCHRONOUS INTEGRATION



Moving on to the next part of our e-book on Data Integration Best Practices, in this chapter we're going to explore the main two types of integrations: asynchronous integration and synchronous (also known as request-reply) integration.

Whether you need a fast response from the systems involved or the results to be delivered further in the future, both request-reply and asynchronous integration offer their own pros and cons and could be used for different purposes.

REQUEST-REPLY (SYNCHRONOUS)

Request-reply integrations require interactions with an external system to occur before an operation is complete. In general, we need these types of integrations for rendering information in front of a user in real time.

For example, a system needs to display a list of courses to a volunteer based on their profile while the list of courses is stored in a different system. In order for this to happen, the system requiring information must request information from the system containing the information. And the system containing information must reply to that request.

A common protocol for this exchange is HTTP(S) but other protocols are possible. These requests may be 'read' operations that do not have any so called side effects – that is to say "has an observable effect besides returning a value". However, they may also be 'write' operations or other operations with side effects.

In order for the requesting system to operate, the system where the requests go to must be up and available.

When building request-reply integrations, there are two strategies as follows, each with their own pros and cons.

DIRECT REQUESTS

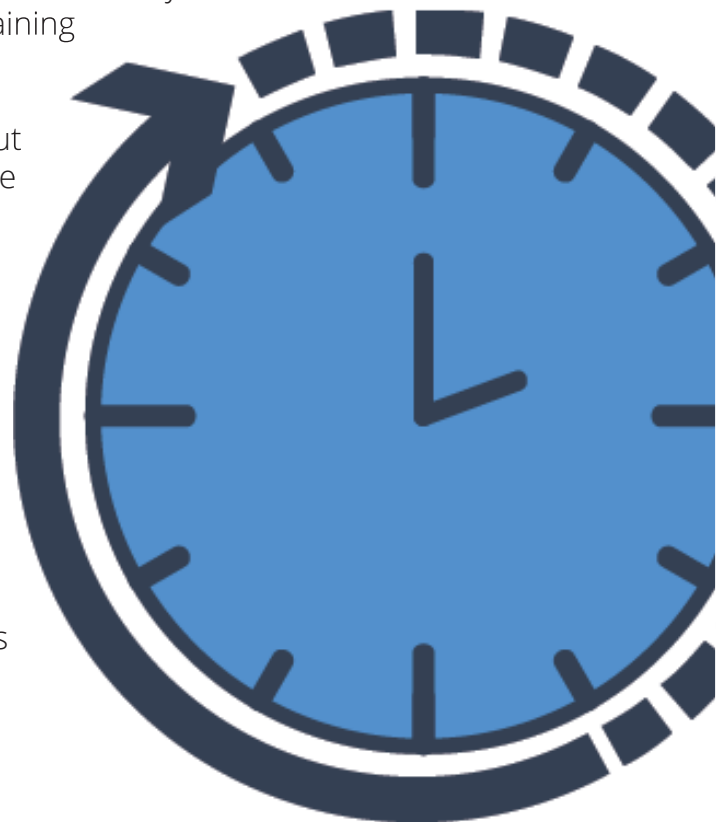
As the name suggests, these are requests from one system to another without any intermediary system.

Pros:

- Data exchange happens fast
- It is reasonably simple to set up
- There are fewer failure points as there are only two systems in play

Cons:

- Error tracing is difficult because you wouldn't usually have a visual interface or built-in monitoring tools
- Credential management is difficult too



REQUESTS THROUGH AN INTEGRATION LAYER

In this case, we have a third system between the requesting and the receiving system, which plays a role of intermediary. This can be a third-party or in-house solution.

Pros:

- This third system can act as mediator by modifying or enriching requests and responses
- It can also act as a caching mechanism when there is no built-in one
- It can also have a system health tool role providing system monitoring. After all, you have two systems talking to each other directly, it's hard to know if there is a problem with that "conversation". With this third, intermediate layer, you can check if this communication is OK or not.
- It can provide centralized logging, error management and credential management
- Additionally, it can act as an abstraction layer. For instance, to collect information from several similar sources such as email providers where the actual source is not relevant but the information is.

Cons:

- With an additional system now in play, this adds yet another failure point
- Has some performance overhead since the third system requires some time to process and sort the information it receives and sends

ASYNCHRONOUS INTEGRATION

Asynchronous Integration is integration where the data does not have to be moved immediately but can be moved at a later point in time. This means that the system sending a request doesn't have to wait for a reply in order to continue operating.

This type of integration is especially useful when we have large volumes of data to process or when we don't expect any immediate response. For instance, when we set up a regular data sync between a CRM and ERP systems. If a sales employee enters a new account, the CRM system would spot the change and push it, e.g. into a queue on an integration middleware. In doing so, CRM has done its part of the deal. From there, it will be at some point picked up by or pushed to the ERP system. It doesn't really matter either for the employee or for the CRM when the update occurs – immediately or in an hour – as long as it occurs eventually.

Pros:

- Asynchronous integration is considered to be more reliable than synchronous integrations. Due to the fact that systems don't wait for each other, none of them will have a timeout.
- Derived from the previous point, asynchronous integration lead to higher services availability as asynchronous integration can wait for systems to recover
- More efficient use of machine resources
- The system can offload data when it is not busy
- Batching becomes possible

Cons:

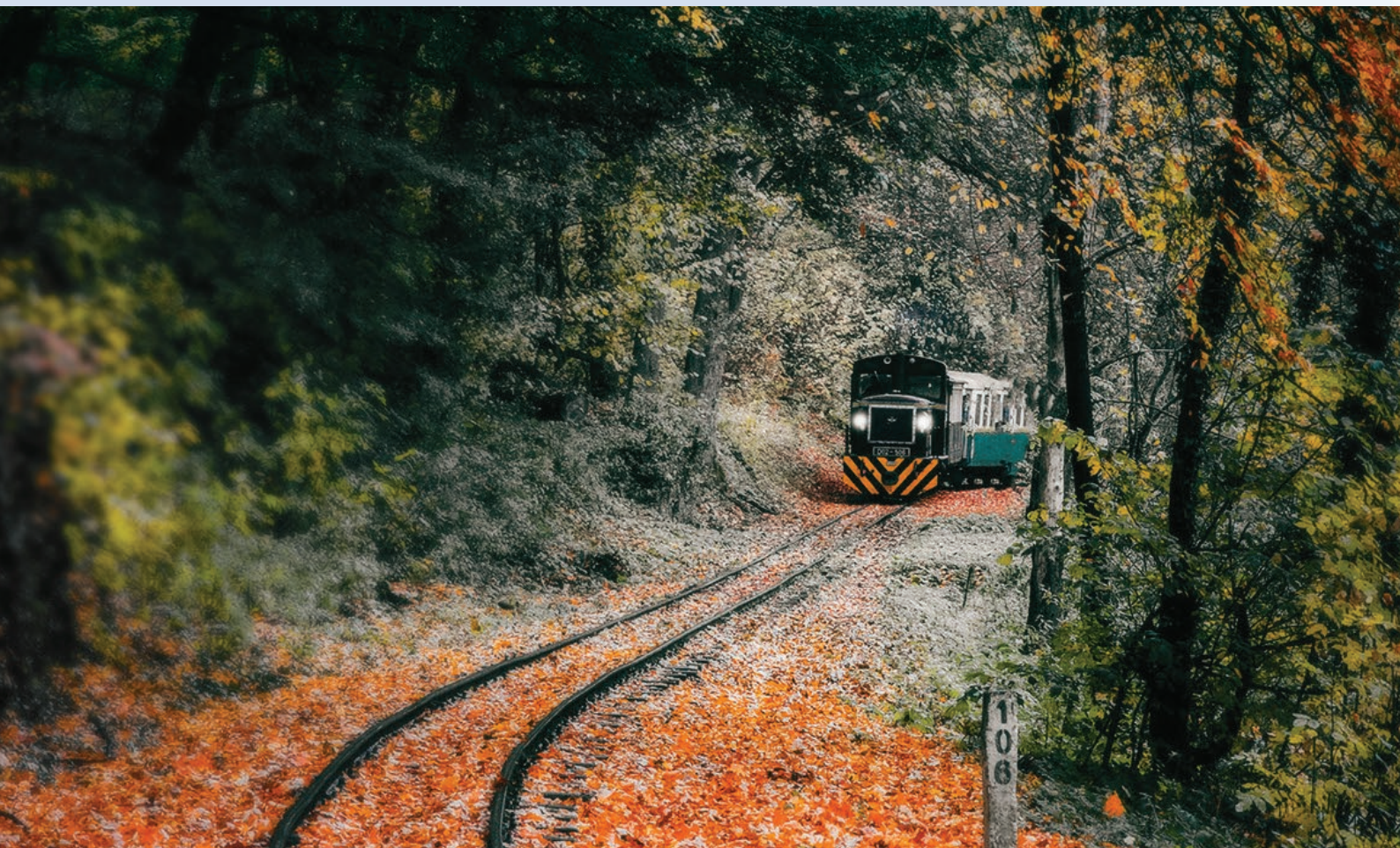
- The fire and forget model means that there is a risk that these events are forgotten
- Immediate feedback is not possible
- Lag may lead users to assume that the system is broken

So, as you can see, it really depends on your integration scenario which type of integration you would need. Both synchronous integration (request-reply) and asynchronous integration have their specific application cases, and their pros and cons, too. Sometimes, however, you have no choice but to choose one over the other. That's why you can consider the cons listed above as something to keep in mind rather than something to try to avoid. To learn more about different types of these integrations, please refer to the following article on our blog "[3 Message Exchange Patterns in Application Integration You Should Know About \(with Examples\)](#)".

In our next chapter, we'll have a look at different systems that enable these synchronous and asynchronous integrations in the first place. In other words, we'll check out the systems that move data

Chapter VIII

WHAT SYSTEMS MOVE THE DATA?



Obviously, data can not move itself. A processor somewhere must pick up and move data somewhere else. So, while in the [previous chapter](#) of our e-book on Data Integration Best Practices we talked about HOW we can move data, this time we are going to see WHAT moves data between systems and the design choices that we need to consider for that.

WHEN SYSTEMS MOVE DATA WITHOUT INTERMEDIARIES

Here we are basically talking about a system that either has some import / export / synchronization capabilities by design or allows the user to define or add such capabilities.

Some forms of this include SQL Server Push/Pull capabilities, cron jobs – i.e. jobs that “wake up” on a specified schedule – which push or pull data between systems, or cron jobs that run import or export tasks to write data to or from certain files. Alternatively, an application might have already been designed by its manufacturer to interact with other specific systems, which is common to larger software suites such as Sage or Microsoft 360. Last but not least, there may be customer plugins, extensions or customizations that allow one application to interact with each other – something one would find often in the eCommerce space, for example, with Magento or Shopware.

Like with any other methods we have covered so far, this one, too, has its own advantages and disadvantages.

Pros:

- This way of how we move data can be more performant. Especially, if system pairs are optimized to talk to each other
- If this is a standard pairing, then setup of the integration can be quite fast
- There is no need to add and manage a third system. There is also, by definition, no vendor lock-in with respect to this third system
- There is no single point of failure for all integrations

Cons:

- As the system architecture grows, it becomes more and more difficult to visualize and document it. Maintaining the configurations becomes more difficult as well.
- Building integrations is generally a long and expensive process
- You cannot replicate the integrations. Every time you need to reuse an integration, you need to do that from scratch
- Missing standardisation in integrations requires a know-how owner to be present in case of changes
- Logging and monitoring mechanisms are spread across systems. It is, therefore, difficult to tell which jobs are running on which system, and when lifecycle management of integration logic becomes difficult
- The number of credential pairs you'd have to manage will grow exponentially

- You would need to rediscover and re-implement the particular specifics of each system per system pair, instead of per system.
- Integration errors become hard to manage

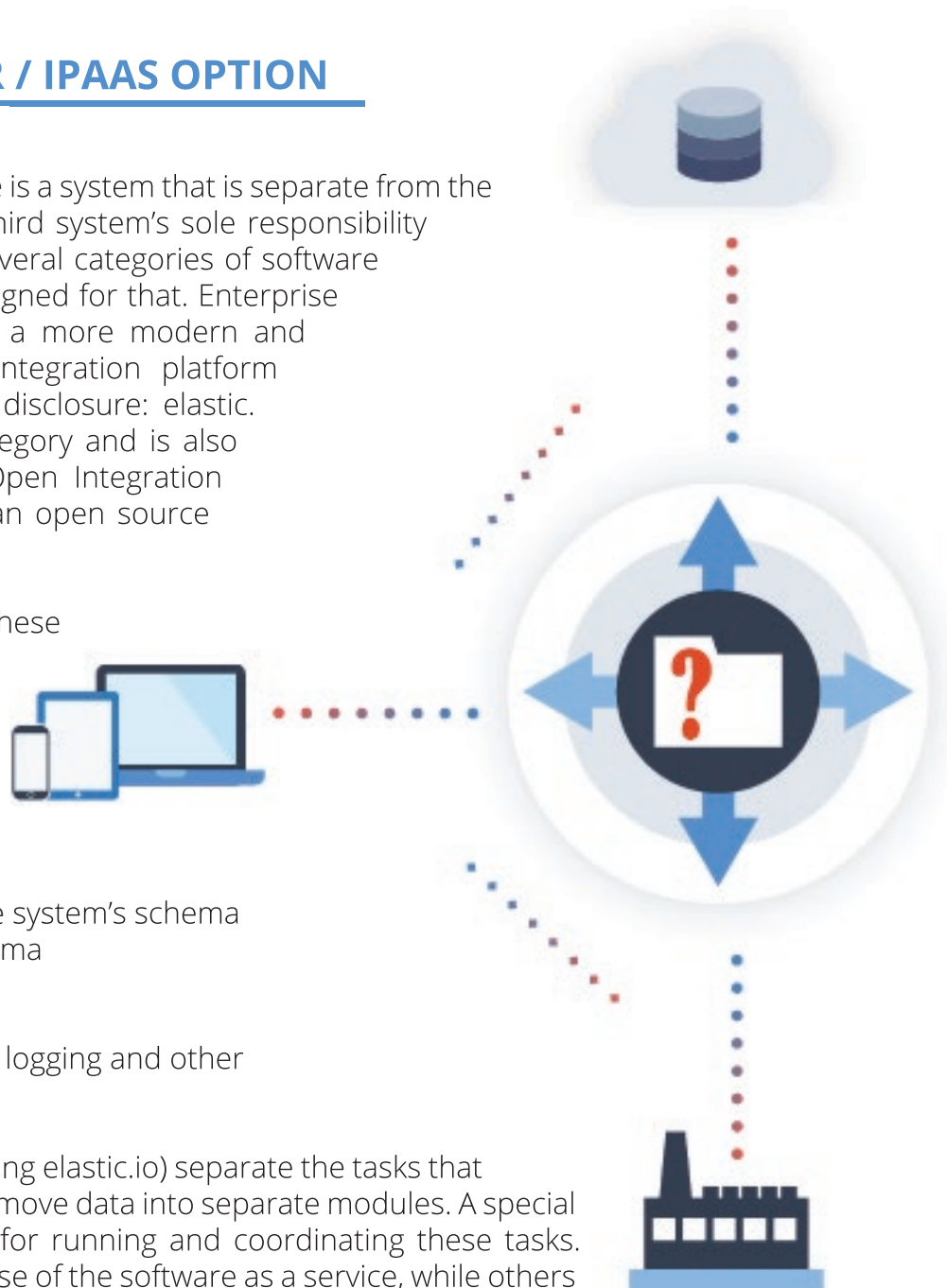
INTEGRATION LAYER / IPAAS OPTION

The principle here is that there is a system that is separate from the systems with the data. This third system's sole responsibility is to move data. There are several categories of software product or service that is designed for that. Enterprise Service Bus is one of them; a more modern and lightweight solution is call integration platform as a service – or iPaaS. Full disclosure: elastic.io belongs to the second category and is also currently working with the Open Integration Hub foundation to produce an open source version of our iPaaS.

The premise behind these services/products is that all integrations must solve the following problems:

- Move data from inside the system to outside of the system
- Transform data from one system's schema to another system's schema
- Allow ID Linking
- Execute the tasks
- Monitoring, error collect, logging and other operational concerns

Many of these systems (including elastic.io) separate the tasks that transform data and tasks that move data into separate modules. A special software is then responsible for running and coordinating these tasks. Some of the vendors sell the use of the software as a service, while others sell a license to software that you must run and host yourself.



Pros:

- Even if the number of integration grows considerably, you still have one place to overview them all
- Most if not all such systems are designed to ensure reusability of integrations
- Many integration layer solutions provide so-called connectors, which are responsible for connecting with an application without having to deal with the actual code of this application
- Logging and monitoring is provided in one place. For instance, this allows to quickly find the source of errors and the reason for them
- You have a centralized place to control integration processes, and
- A centralized place to manage connections to other systems

Cons:

- Additional cost on integration project that is recouped over time
- Just as with any other software as a service, there is a risk of a vendor lock-in
- If the integration layer solution is down, then all integrations fail by default
- Having a third system adds some performance overhead since it requires time to process and sort the information it receives and sends

So, when would you choose one approach over the other? Using the inherent integration capabilities of applications to integrate them directly makes sense when you have only a handful of them. As soon as your business or the number of automated business processes start to grow, having this kind of point-to-point integration will result in more of a tape spaghetti than anything else. It doesn't mean, though, that you have to buy expensive integration suits from the start. There are many services out there that fit various integration needs, from very little and simple to heavily complex ones.

In our next chapter, we'll talk about the differences in what is being integrated. For instance, we'll have a look at the specifics of integration with a shared authentication mechanism or the difference between event propagation and data synchronization.

Chapter IX

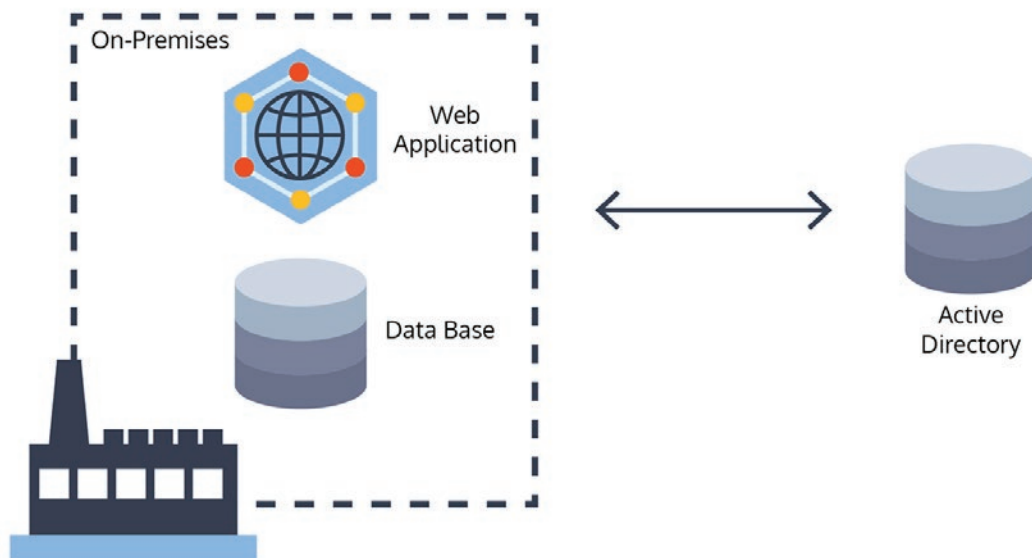
DIFFERENCES IN WHAT IS BEING INTEGRATED



In the part 2 of our e-book, we have already talked about different communication types for data integration – synchronous vs. asynchronous. We have also talked about the main difference between the types of systems that move data – direct data synchronization vs. an integration layer. In this chapter, we would like to take a step back and review some differences in what exactly we are going to integrate.

As an exception, we are going to use the word “integration” here in a more high-level sense. Some systems are designed to interact over a fixed protocol – in which case we are talking about integration of permissions. Others are subsystems of a larger system, meaning that here, we deal with integration in the sense of configuration. There are yet other systems that by design, have no integration capabilities with each other.

INTEGRATION WITH A SHARED AUTHENTICATION MECHANISM



In this situation, you typically have a system that should solve a problem. In other words, a common business application such as Microsoft AX or Navision. In order to use this system, users need authentication and authorization. Usually, you can configure such a system to keep track of its users with its own username/password mechanism.

However, these systems are also designed to exist within a corporate ecosystem and as a result, to connect to one or more single sign-on systems. Common protocols for connecting between individual applications and a sign-on system include:

- LDAP (Active Directory is an implementation of LDAP)
- OpenID / OpenID Connect (OAuth-based)
- SAML

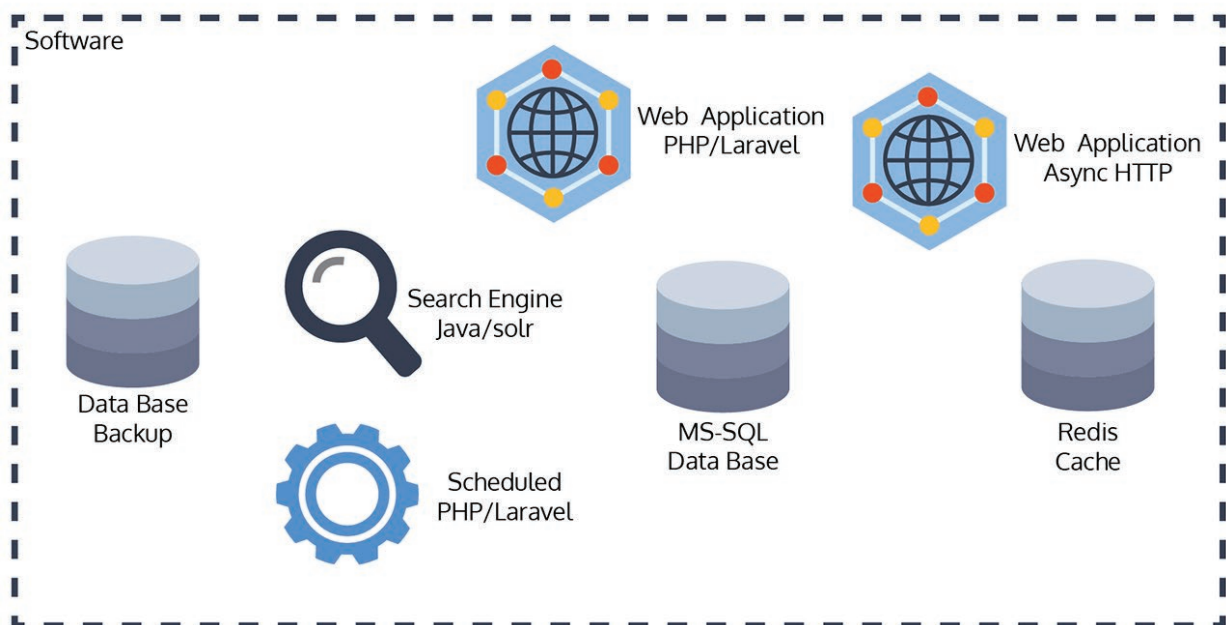
In general, the application should work with one of these protocols. In most cases, it is not possible for an outside provider to add these abilities if they don't exist. For instance, if all your business applications use LDAP and you want to buy a software that doesn't support this protocol but uses OpenID instead, it will be nearly impossible, and also impractical, to try to map between the new application and the existing ones. So, you should find out in advance

what protocols your desired application supports.

Generally, if properly configured and implemented, these protocols can define error handling by design.

However, shared authentication mechanisms introduce a data synchronization problem. Systems that work with such mechanisms fetch information about their users from a single authorization service. This means that this service will receive the user information from the protocol upon his sign-up. However, if the user was changed or deleted, the system wouldn't learn of that change unless we have set up a certain data integration flow to detect and apply that change.

INTEGRATION BETWEEN DIFFERENT PARTS OF A SYSTEM



In this case, you have one application or system which consists of smaller sub-systems (see the picture above). And so, by default and by design, the application is optimized to work with each subsystem being connected in the way this application “needs” it to be. Consequently, these different sub-systems are responsible for connecting to each other and ensuring that this connectivity remains. Considering this, it would make no sense for an outside provider to interfere with this structure for whatever reason. You know what they say - “If it isn't broken, don't fix it”.

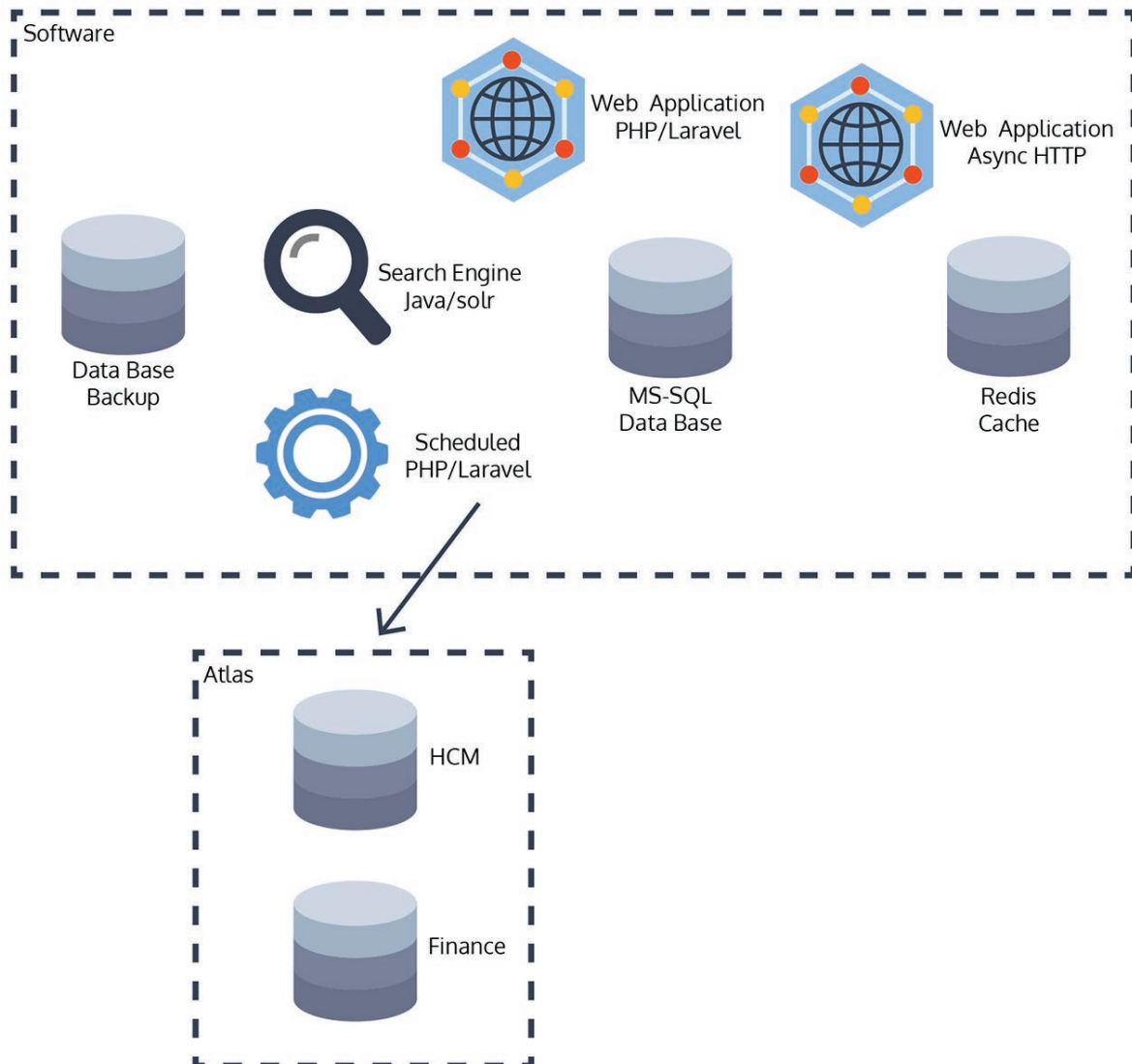
EVENT PROPAGATION BETWEEN SYSTEMS

This scenario assumes that you have two different systems that were designed independently, likely by different vendors to solve two different problems. As a result, there is no “native” way for these systems to talk to each other. And so if an event occurs in one system, the other

system needs to become aware of this event.

This scenario is similar to the data synchronization case bellow, except that it is simpler and lower-level. Once an event has happened, the information about it doesn't change. For example, once you received an email, its contents stay the same.

DATA SYNCHRONIZATION BETWEEN SYSTEMS



Just like with the previous scenario, we have here two independent systems, likely from different vendors, that solve two different problems. And just like previously, these systems don't speak to each other by default. Therefore, if some data appears in one system, the other system needs to learn about that. The same applies to any updates that happen to this data. For instance, data could be all customers who have bought a particular product. While buying a product is an event (see the section above), the up-to-date information about all these customers is data. To get back to the data integration topic, often, you would care more about synchronizing data and not events.

With this chapter, we have closed the part 2 of our e-book – where we described the different

types of integrations and integration systems. The next and the last part our e-book is about best practices moving forward.

And in the very first chapter of the final part, we will touch upon the topic, that can give many headaches to IT decision-makers – the question of costs. We will talk about costs calculations for building, operating and maintaining integration projects – something to think about when deciding for or against a third-party integration solution.



PART 3

INTEGRATION BEST PRACTICES MOVING FORWARD



Chapter X

INTEGRATION PROJECTS: THE REAL AND HIDDEN COSTS



So far, in our e-book on Data Integration Best practices, we have covered the different types of high-level and low-level problems occurring in data integration projects. We have also addressed the different types of integration and the systems that move data. Nine chapters later, we arrived at best practices moving forward.

In this last part, we are going to talk about some tips that revolve around preparing for and running an integration project. And the very first aspect that we are going to cover is the pricing aspect of an integration project.

While defining a data integration project we should consider what we want to integrate, when and how we want to integrate it. Once we have done this, we can move on with the last step of the plan: the costs. The billion-dollar question... how deep should we dig into our pockets?

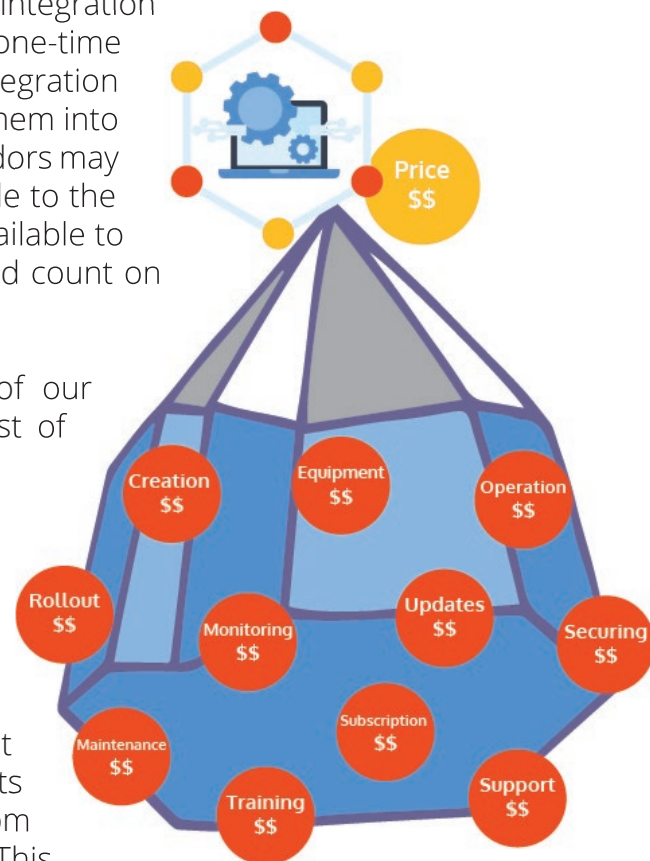
If we want to have a realistic idea of the cost of our integration project, we should keep in mind that it is not a “one-time payment” thing. Building, operating and maintaining integration projects incur various costs and we should take all of them into account before we embark on a project. Of course, vendors may allow for trade-offs in these costs if the pricing available to the purchaser is substantially different from the pricing available to the vendor. However, this is not something one should count on by default.

So, in order to determine an accurate total cost of our integration project, we should calculate the total cost of ownership (TCO), which includes the following criteria:

CREATION COSTS

These are also known as research and development costs. As the name suggests, they represent the costs present at the beginning of the project, starting from the research all the way down to the implementation. This includes:

- Consulting and /or implementation costs
- Installation and rollout costs
- License costs if you decide to use an off-the-shelf product, or
- A tailor-made plan, which may include regular costs, such as:
 - Perpetual license costs or subscription costs if it's a SaaS product
 - Maintenance and support costs
 - Upgrades and updates costs (if any)
- Operation, administration and training costs



OPERATIONAL COSTS

These are the expenses you would have to keep the integration project running. Such expenses include:

- Integration, monitoring, and administration costs
- Updates, upgrades and maintenance costs

Additionally, some other operational costs depend on what kind of product you have. For on-premises or perpetual products you need to take into account:

- Hardware operational costs for PCs and/or virtual machines that you will require to run the on-premises product
- Administrative costs on operating, securing, backing-up and ensuring the availability of the product

For SaaS or cloud-based solutions you will quite likely have to deal with increasing traffic costs. This means that you might want to pay special attention to this aspect when selecting a vendor, because depending on how data should ultimately be passing between your applications, it can become very expensive very quickly.

RETIREMENT COSTS

Last but not least, these are the remaining expenses that we will have to endure if we decide to leave the project. An example of such would be remaining license costs if we, for some reason, terminate the contract before the end of the license agreement.

An integration project could be an iceberg of hidden costs, therefore before implementing it, we should carefully consider each possibility in order to avoid unpleasant surprises and have a more accurate budget planning.

In the next chapter, we will talk about how to describe integrations in such a way that everybody is on the same page: the IT team, business management and any other stakeholders.

Chapter XI

HOW TO SET THE FRAMEWORK FOR DATA INTEGRATION PROJECTS



Even though it's IT who is responsible for connecting applications to ensure an efficient data exchange between them, it's often the business users upon whose request an integration project is initiated. Say, the marketing department needs to synchronize a new content management system with the company's CRM. When there is such a request, – or need, if you like, – it is important to communicate the requirements clearly and accurately.

On the other hand, the IT team should also document their integrations accurately, so that other parties such as technology partners or IT teams from other divisions can make sense of it in future projects. And this is what we are dealing with in this chapter.

DESCRIBE INTEGRATIONS BETTER

DESCRIBE BUSINESS LOGIC AND THEN MECHANICS

There is a tendency to look at the available mechanics first, and then define the business logic in terms of the mechanics. However, a far better approach is to clearly describe the business logic and then look at the available mechanics to implement that business logic.

For example, we have a SOAP API that can do a number of actions. The “mechanics first” approach would be to document that we are going to connect action A to action B, and then action D to action C. Easy to understand for a developer, harder so for non-techies.

The “business logic” approach first, on the other hand, looks at what you want to have happened. For instance, you want to update a contact in the application X and then you want to see this update automatically appear in the application Y. After you define that – then you can look at whether a particular API can support this particular use case.

USE VERB + NOUN DESCRIPTIONS

When describing an interaction with a system, business requirements focus on the noun over the verb. For instance, a business user might submit a request “I need Salesforce contacts”. However, this is not exactly helpful to understand what it is that they want to achieve with Salesforce and contacts.

In addition to that, there are other problems with such approach. For one, when we are talking about describing the integration needs in the context of an API design, adding or changing nouns is lower cost than adding or changing verbs. In concrete terms, if we have ‘read contact’, it's easier to create ‘read order’ than it is to create ‘write contact’. It has to do with how most APIs are designed – one action is created to be similar to other actions.



Hence, 'reading a contact' and 'reading an order' are pretty similar, whereas 'reading a contact' and "writing a contact" are not.

In addition to that, verbs indicate the direction of the data transfer and can better describe dependencies. That is why it is necessary to keep in mind that the most important part of the description is the verb (e.g. read, write) and only then the noun (e.g. contact, order).

CATEGORIZE INTEGRATIONS

For each integration in your ecosystem, you should categorize it based on the following axes:

- Type of integration. Is it shared authentication or integration between parts of a system? Or maybe event propagation vs. data synchronization?
- Type of communication. Does the integration require the request-reply or async communication pattern?
- Identify the system moving the data
- Describe each interaction with a system using the verb + (if necessary) adjective + noun pattern

FORMALIZE ID LINKING STRATEGIES

Last but not least, make it clear how the IDs of related records are linked. This is something we've covered in more depth in our earlier article on data duplication and ID linking, so no need to repeat it here.

CONSIDER USING AN IPAAS / INTEGRATION LAYER

It might sound like a very predictable recommendation coming from an iPaaS vendor, but let's look at this objectively. The more systems you add to your IT infrastructure, the larger the company grows, the more sophisticated the business needs become – the more "interactions" between various systems, applications, databases and what not across the whole organization, and maybe even beyond it, take place.

First it was, say, simply automating the order fulfillment. Then came synchronizing customer and purchase data between your online and physical shop. Next thing you know, you need the notorious 360° view of the customer, you have some intelligent algorithms running to meet your customers' subconscious desires, and your network of partners is rapidly growing. All that needs consistent data exchange and synchronization.

As the number of systems and the complexity of the interactions between them increases, trying to manage them the old-fashioned way will quickly lead to bottlenecks in IT projects, hardly traceable bugs and errors, unclear or inconsistent data, and so on, and so forth.

Eventually, you will need some kind of an integration layer – be it iPaaS or any other comparable system – to be able to keep track of integrations and integrated systems, to find and resolve errors quickly, to add new integrations within a reasonable timeframe and to enable teams across the organization and beyond to work on the same data integration projects.

And here's the deal: You can regard this recommendation as coming not from an iPaaS vendor, but from a company that has accumulated extensive insights into the complexity and challenges of integration projects.

In our last chapter of this e-book, we continue looking at some best practices regarding the overall implementation of such projects. Specifically, we will go through three types of project environments and what part of the project each type includes. In addition to that, we will review some best practices for log collection.

Chapter XII

SEPARATION OF ENVIRONMENTS AND LOG COLLECTION

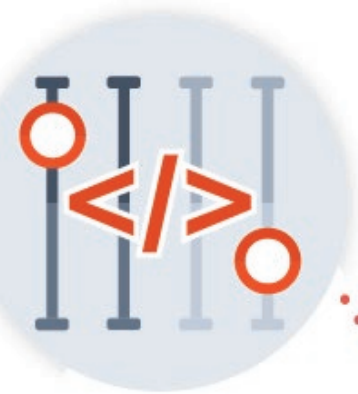


In the previous chapter, we took a look at how to describe integrations in such a way that everybody – from developers to business users – understands the requirements correctly or can follow what exactly has been done within the scope of this or that integration project. We also discussed why you eventually might need some type of an integration layer to keep your integration projects under control.

In this chapter, we continue reviewing some tips that revolve around preparing for and running an integration project. As the title suggests, we will start with three types of project environments and what part of the project each type includes, and finish with some best practices for log collection.

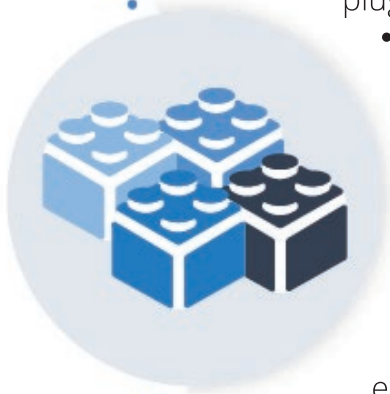
SEPARATION OF ENVIRONMENTS

In general, for each software system, there should be three types of environments: development, staging and production. The same is also true for an integration project. So, let's dive deeper into what you need – or don't need – to consider at each stage.



DEVELOPMENT

- In general, at this stage in an integration project, you are building the interactions to move data from inside the system to outside the system and vice versa. As a result, the system should not be connected to other systems in the environment. One possible exception would be connecting to a staging or production authentication mechanism.
 - The database should be mostly empty and resettable
 - This environment should be used to test new integrations, plugins, schema changes or other configuration tweaks
 - As the data isn't real, there is no need to be worried about data anonymization



STAGING

- In this environment, opposite to the development one, a system should be connected only to other staging systems. One possible exception is being connected to a production authentication mechanism.
 - The database should be full of test records with comparable structure and volume to production systems
 - In this environment, you can have more liberal access policies for external vendors, developers, or any other “parties” involved
- You should use this environment to verify the impact of changes on other systems
- You also should be able to copy the configuration to production
- As the data isn't real, there is no need to be worried about data anonymization.



PRODUCTION

- At this stage, admin access should be restricted only to a few trusted people. Should you need to grant an additional access, it should be temporary and monitored
- At this stage, a system should only talk to other production systems

LOG COLLECTION

In an environment with multiple systems, each system will produce its own logs. Considering this, it is particularly important to pay special attention to log management as part of data integration best practices. Here are a few key points you should keep in mind.

FILE-BASED VS DATABASE-BASED LOGGING

Computer systems generally produce log information in discrete log statements. Such statements can be written sequentially to a file or placed in a logging database. Log files are simpler and generally more reliable but have some drawbacks. They are harder to search, especially when there is a log statement per machine in a service.

Logging databases, on the other hand, are easier to search. In addition to that, they support anonymization capabilities, even though this requires more setup. It is also possible to extract +log files into a logging database – i.e. a database which sole purpose is to store logs.

LOG FORMATTING

Many systems support the ability to have their logs formatted in several different formats. Since you will most likely need to integrate logs from various providers, you need to take care of configuring the systems you use to produce their log statements with the same format. This will make integrating the logs from various different systems a lot easier.

LOG EXTRACTION

In integration projects, a fair share of troubleshooting revolves around the problem of some information to be expected to leave one system and arrive at another while it doesn't. To find out why, you would typically pull the logs from both systems to check if the data left the first system at all and if yes, why it didn't reach the destination system.

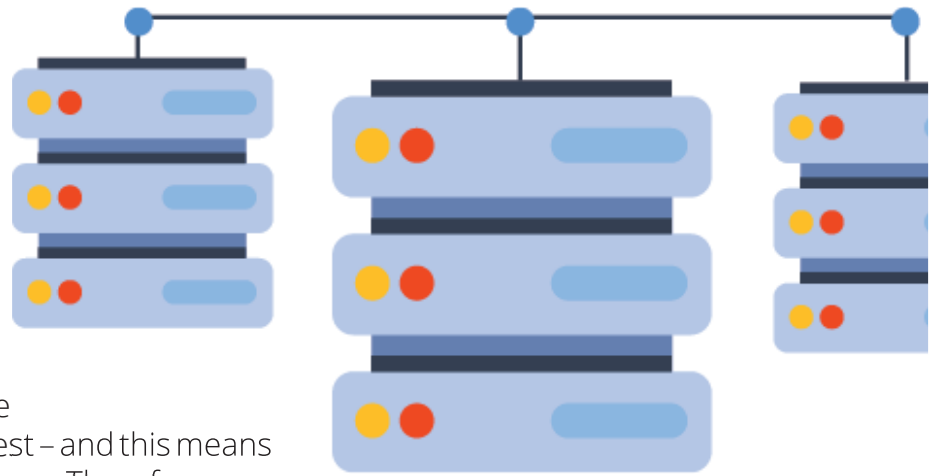
If you can aggregate all those logs in a central location, it becomes considerably easier to search through them. Such aggregation is implicitly available in logging databases, since such a database is generally shared between servers and systems. For log files, you would need to do some extra setup to pull them from a server that produces them and push to a central store where you can manage them more easily. It is also possible to extract log files into a logging database, such as Graylog.

LOG ANONYMIZATION

Occasionally, logs might contain sensitive information. If you place all logs in a central place, it can become a security risk. In order to reduce this risk, you can either: Refrain from logging sensitive information in the first place, or configure your logging database to anonymize data

LOG RETENTION

Logs take up space, and space costs money... The General Data Protection Regulation (GDPR) also stipulates that personal information needs to be deleted at some point, whether after some predefined time period or upon request – and this means deleted completely, including from logs. Therefore, you need a clearly defined strategy for log retention. Often organizations have automated processes to delete certain logged information after a specific period of time.



With this chapter, our e-book on Data Integration Best Practices has come to an end. To summarize, we have covered the different types of various problems that can occur in data integration projects. We have also addressed the different types of integration, the systems that move data and even the pricing aspect of such a project. Last but not least, we reviewed some practical tips for preparing and running an integration project. There is only one thing to be added, so jump over to the conclusion page.

Conclusion

Nobody really reads conclusions, so we'll be concise. Of course, we haven't been able to cover every single aspect of running an integration project successfully since each project is different – different systems, different environments, different starting conditions, different business needs.

Nevertheless, we hope that this e-book will help you prepare and run an integration project at least a little bit better than before – be it by giving you a new perspective or highlighting some important points you might have not yet considered. With all that said and done, we wish you good luck.

And of course, you are always welcome to try and see if our own [elastic.io integration platform as a service](#) can help you achieve your integration goals faster ;-)

DATA INTEGRATION BEST PRACTICES

JACOB HORBULYK



Rabinstrasse 4, 53111 Bonn, Germany | +49 (0) 228 53444221
info@elastic.io | www.elastic.io

