White Paper

Mulesoft + Appian

A Macedon Labs Exploration

By Brandon Sultana

MACEDON
TECHNOLOGIES

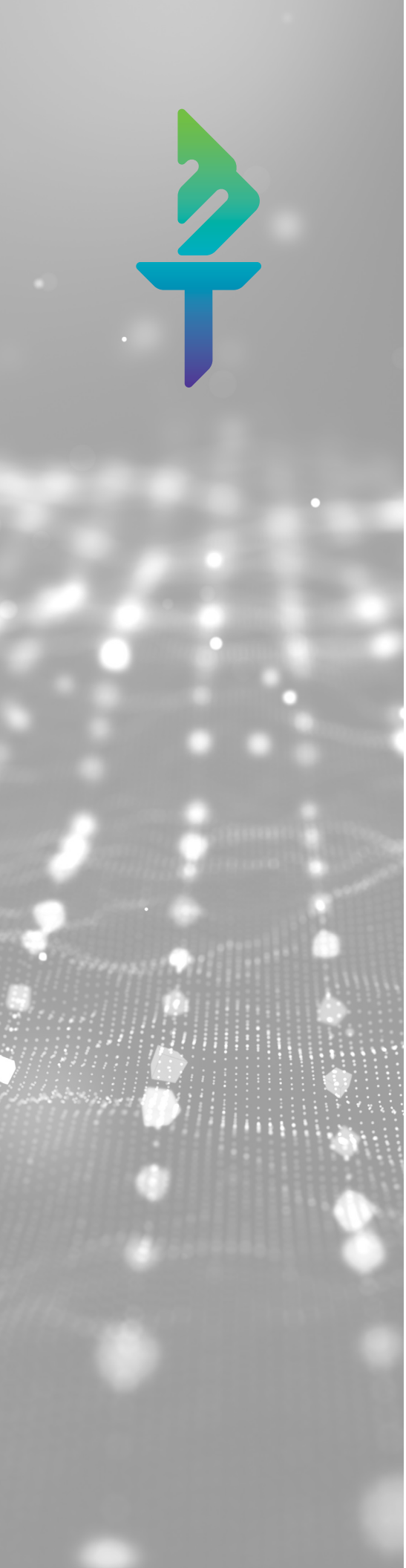


Table of Contents

Introduction

Macedon Technologies improves the way companies conduct business by leveraging innovative technologies that streamline processes. Appian has been a core part of that business model as a means to create frontend systems backed by Appian-hosted databases and processes. As part of our research into additional technologies, Macedon has found Mulesoft to be a powerful partner to augment our Appian digital transformation solutions.

The Enterprise Service Bus (ESB) is a rapidly-growing architecture design pattern that promotes high scalability through independent applications known as microservices. These applications are frequently linked together through an Application Programming Interface (API) in the ESB, which acts as a centralized hub for the entire system.

MuleSoft is an ESB owned by Salesforce that promises ease of development and reduced costs compared to other ESB systems. This is made possible by a platform that promotes reusable connectors to popular microservices, saving developers time in building out integrations. This streamlined build process, along with other time and cost-saving practices such as robust error handling and data scaling, make MuleSoft an essential platform for building applications that can scale with ease.

In this white paper, we will dive into what an ESB is and what it accomplishes, the tools MuleSoft brings to the table compared to other ESBs, how we combine our existing Appian knowledge with MuleSoft and dive into a case study with a previous client to demonstrate how Appian and MuleSoft together can provide real value to new and existing clients of Macedon.

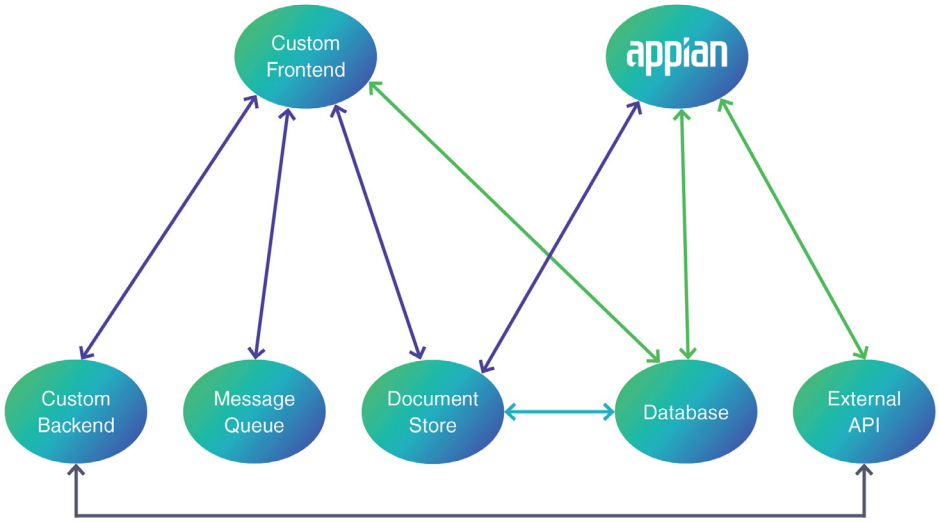
ESBs

What is an ESB?

An Enterprise Service Bus (ESB) is an application that is designed to link systems together using a set of rules defined by a development team. In the age of microservices, ESBs can play a critical role in delivering data from one application to another. ESBs provide a centralized hub for various different platforms to communicate with each other. This concept allows individual applications to be more independent while the ESB handles the heavy lifting of routing connections between those applications.

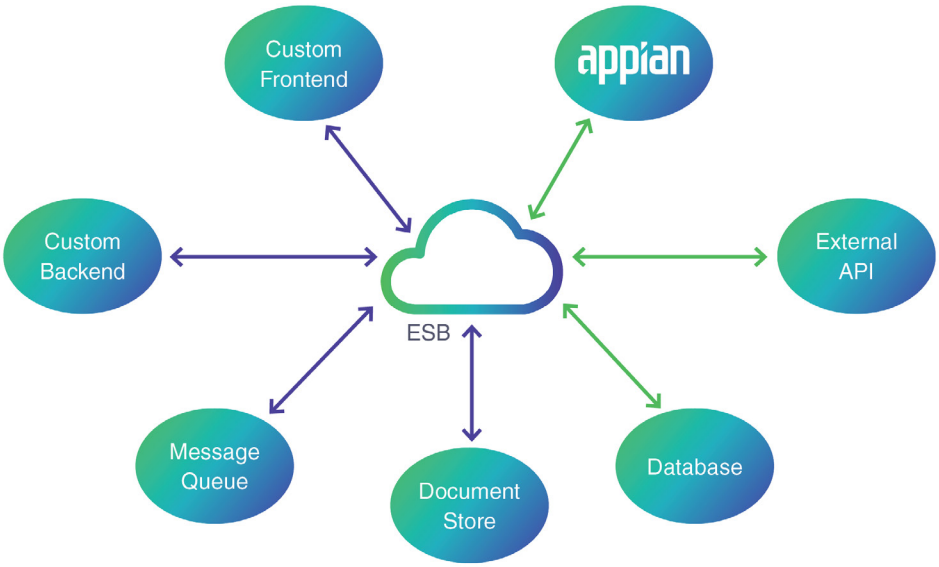
ESBs are most useful at high scale with many different interconnected services. For a system with two applications that communicate with each other, it makes sense to have these connections to be direct. However, if we expand this model out to a whole host of applications, each with its own list of applications that they communicate with, writing the connections needed for all of these applications to function gets significantly more complicated. Each application needs dedicated logic to handle each connection that it maintains. With an ESB, unified calls to each system are written, so it becomes substantially easier to add new applications to a system. If we add a new application to a

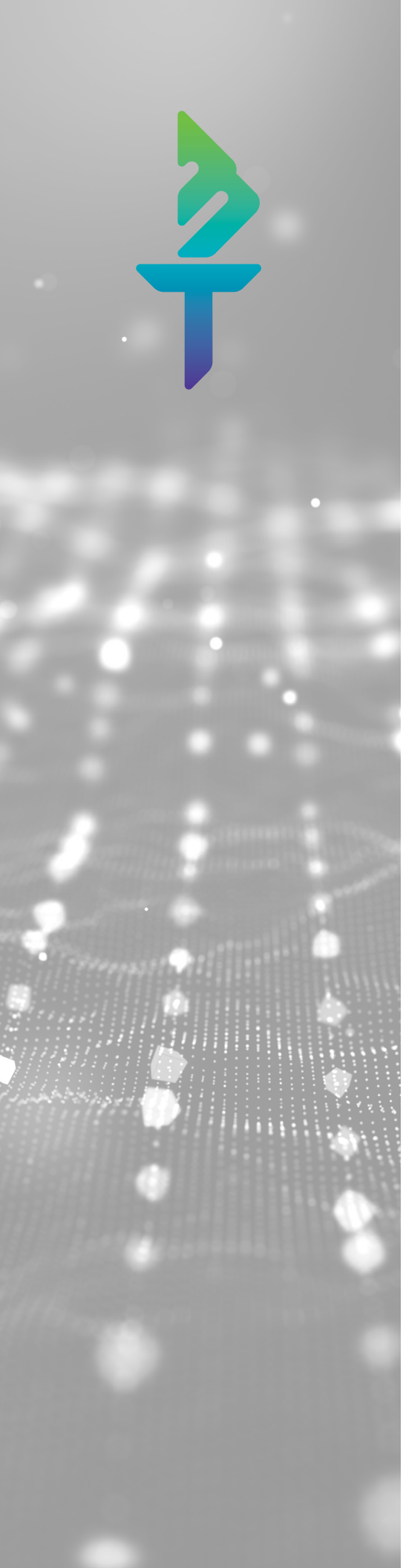
system with an ESB that requires data from 4 different applications, we no longer need to update those 4 applications to talk to the new application. Instead, we simply configure the new application to talk to the ESB and get the data provided by those applications through that connection with the ESB.



ESB Features and Benefits

ESBs can potentially receive high amounts of throughput for high-value systems, so it is important for the ESB to be scalable and have systems in place to manage and queue incoming traffic so that nothing is missed. Many modern ESBs, such as MuleSoft, are built to scale up or down depending on the load the system is currently experiencing. This, in the long run, can save on resources as the company doesn't need to pay for architecture built to handle worst-case traffic loads at all times.[1]





The centralized nature of ESBs also feeds into easier error handling and tracking. With a unified error handling system in the ESB, applications can receive standardized error codes that can be planned around, rather than having each application build out complex error handling structures. MuleSoft features a robust suite of error handling options that can have flows return custom error codes and messages depending on the results of the flow or the internal error that has been thrown

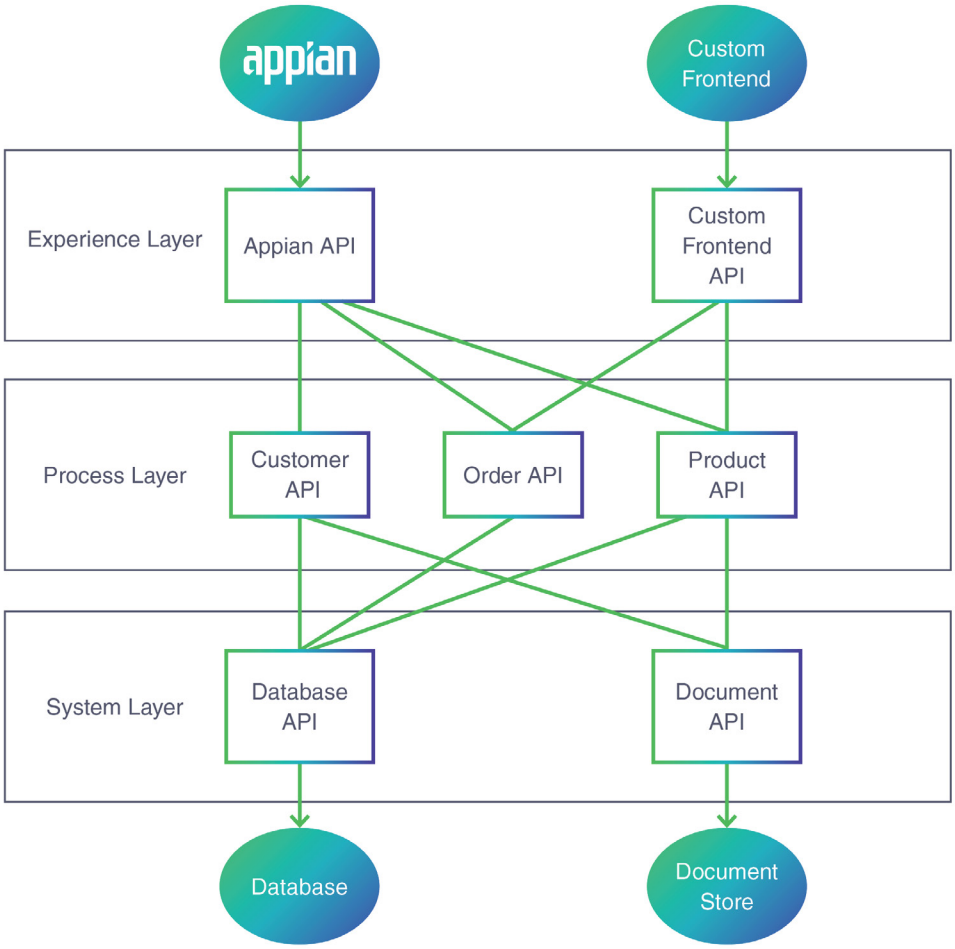
Crucially, a mature ESB practice can greatly reduce the development cost of adding new applications and features to an existing system. With each application operating with greater independence, more time can be focused on fleshing out the features and efficiency of the application while the ESB handles and standardizes many of the connectivity and error-handling protocols necessary for the system to function. Additionally, the independence of each application reduced the risks incurred when migrating or upgrading an application to a new technology. Replacing a system that operates fairly independently requires significantly fewer resources than one that is entrenched with many other applications and requires less overhead and coordination with other teams.[2]

MuleSoft

MuleSoft is a software company headquartered in San Francisco, California, and founded by Ross Mason in 2006. Their namesake product is a highly respected ESB platform used by companies such as eBay, Coca-Cola, Anheuser-Busch, Netflix, and Target. MuleSoft operates off of connectors designed to connect to MuleSoft development is a simple process to pick up but offers endless depth and possibilities for those looking for more nuanced features.

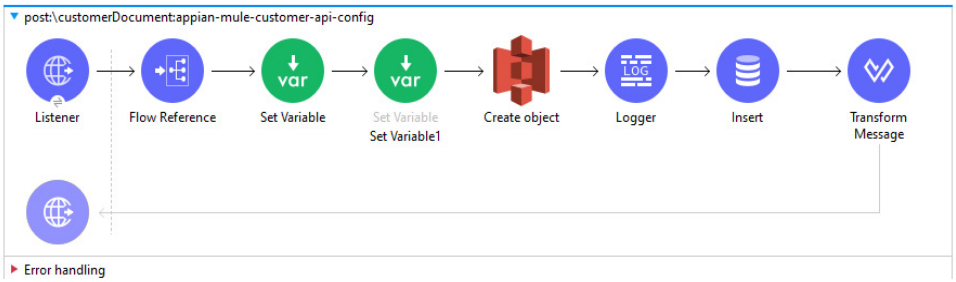
MuleSoft ESBs are built upon a multi-layered API approach, with each layer becoming more and more abstract. This approach allows developers to first create APIs specifically for each frontend application that needs to use the ESB in the Experience layer, then create an additional layer of APIs to transform data in the Process Layer, and finally have a bottom layer to communicate with backend systems called the System layer.

Modules are not limited to what is built and supplied by MuleSoft. Through Java, users can write and develop their own modules complete with any number of objects. These modules can be used to package custom functionality or even connect to services that MuleSoft doesn't support. These modules can then be uploaded to the AnyPoint Exchange site, where they can be accessed either by other users within the user's organization or throughout the broader MuleSoft community. Features like this expedite the addition of features by allowing users to easily create new functionality to a system without significant development overhead.



MuleSoft development is done in Anypoint Studio, an extension of Eclipse that allows for the creation of API flows using drag-and-drop nodes called operations. A flow can be entered either by an HTTP Listener at the start of the flow, which looks for incoming connections on the specified endpoint, or by another flow calling that sub-flow. Each flow contains a set of operations that are executed in sequence until the final operation is executed when the payload is then sent back to the original caller of the flow.

Operations are designed to fulfill a specific action. These operations are grouped together into a module to fulfill similar tasks. MuleSoft comes with several pre-installed modules, such as a Database Connector module with operations for selecting, inserting, or deleting data from a database. MuleSoft provides many different connectors to popular microservices such as AWS, Kafka, Salesforce, and many more for users to download off of the Anypoint Exchange site. Additionally, organizations can develop and publish their own modules to this site to be made available either to the whole organization or everyone on Exchange, granting limitless development possibilities.



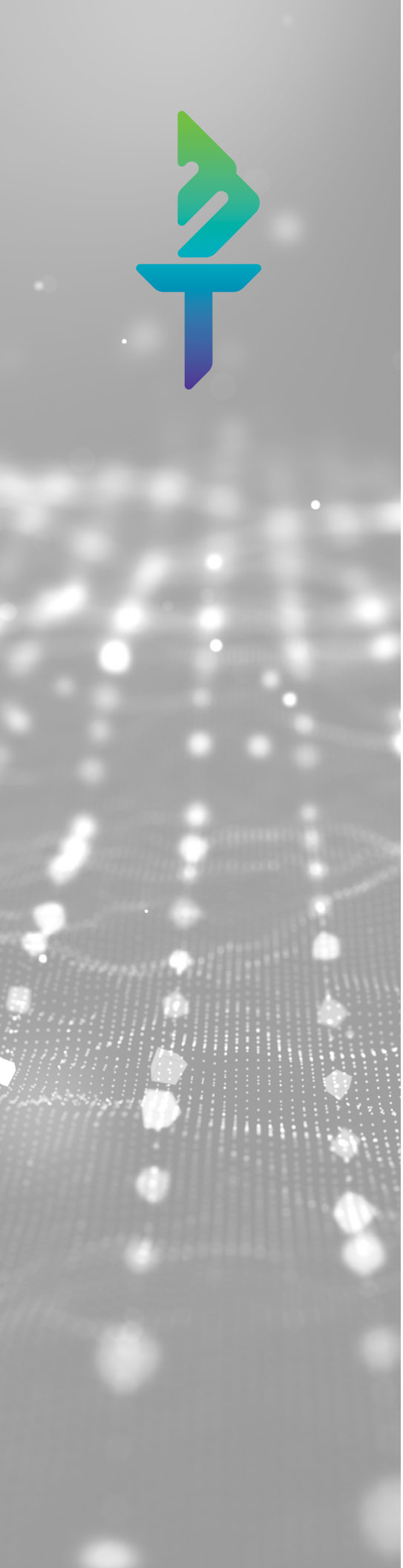
Example of an API flow. Contains an HTTP listener as an entry point and then a series of nodes executed in sequence that perform actions such as setting local variables, transforming data, or connecting to other systems to create or read data

As an alternative to creating each API flow individually, Anypoint Studio can connect to Anypoint Platform's API builder, which is accessible through the AnyPoint Website. This interface allows developers to create API specifications using a RESTful API Modeling Language (RAML) file, defining the routes, methods, query parameters, and responses associated with an API. Object type files can also be added, detailing the kinds of JSON objects that are expected to be consumed or sent out by a given anypoint. Anypoint Studio can then digest these RAML files and automatically build out flows for each of the API endpoints to accelerate development.

The API Specification that is created in Anypoint Studio will include a generic HTTP listener that listens for all requests. Then a flow for each API endpoint is generated, using example inputs and outputs that are laid out in the specification. Each of these flows will be routed to by the main listener depending on the request method and URL that is received. Developers can then go in and add additional nodes to each flow to further transform data, connect to additional servers, call subprocesses/APIs, or many other actions.

Once an API specification has been created, documentation for the individual endpoints and methods can be defined by developers to be viewed on the AnyPoint Exchange platform. This online platform even supports test calls to the API directly from the browser, giving users tangible examples of how the flows work. These calls support a variety of HTTP methods and can include query parameters, headers, and custom HTTP request bodies.

AnyPoint development involves dragging, dropping, and configuring operations into a chain to generate a flow. This design process is simple to pick up and doesn't require in-depth coding knowledge to get started. For more experienced developers, however, custom functionality can be created using a Java Connector to write new features in Java. MuleSoft provides a variety of tools to fit a variety of development styles and experience levels.



Anypoint CloudHub

A MuleSoft ESB can be self-hosted, either on-premise or through a cloud service managed by an enterprise such as Amazon EC2. However, MuleSoft strongly promotes its own CloudHub platform for hosting, which provides additional useful functionality for managing a MuleSoft ESB.

MuleSoft implements scaling through horizontal and vertical means. An ESB is hosted across a series of workers, with each worker having a number of virtual cores (vCores) that represent the processing power of the worker. The architecture can be scaled vertically to increase the number of vCores for each worker or horizontally to increase the total number of workers. Scaling up vertically is useful if the amount of requests coming into the system is very high, as more workers can handle the requests. Scaling up horizontally is useful if the actions taken by the workers require high processing power. CloudHub supports granular auto-scaling rules to be configured for the instance that will automatically scale the system up or down horizontally or vertically if conditions defined by the user are met. A previous problem with older ESBs is their rigidity had the potential to lead to bottleneck issues. The safeguards and features provided by MuleSoft, mean the system can scale up or down to ensure integrations are reliably meeting SLAs.[3]

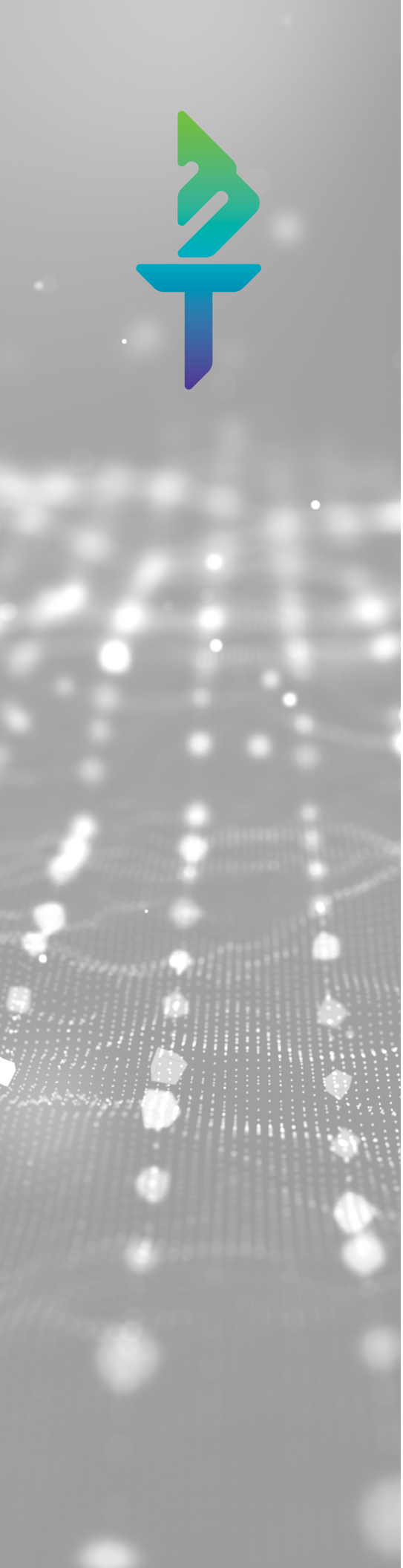
Anypoint also contains robust authentication options. Client Ids provided to customers for authentication to the MuleSoft APIs can be grouped into custom policies, which can have granular rules attached to them. For example, users in a policy titled “Silver” may only be able to access the API once a minute, while a “Gold” policy may allow users to connect 10 times a minute.

APIs that are hosted on the Anypoint Exchange platform can contain custom documentation, as well as testing directly within the page, allowing users to quickly understand the purpose of an endpoint and the standard inputs and outputs. On the page for an API, a user can view each endpoint for the API and the associated methods for that API. Then, clicking on that method displays further information, such as the headers and query parameters that can be input into the method, as well as a sample output of the data.

Appian + MuleSoft

Premise

At Macedon Technologies, we’re the world leaders in Appian development, with over a decade of experience under our belts in engagements to build effective frontend applications to meet various business requirements. This is typically facilitated through a “one-stop shop” approach, where Appian manages a frontend, business processes, and its own hosted database. Businesses using Appian can choose to leverage these services or use their own managed solutions. While the monolith approach is a great model for standalone applications, we determined that in a system with an expansive tech stack using many different technologies, an alternative approach could be more effective.



In many business models, there are multiple different applications that need to share data with Appian. While Appian can technically serve this data to other applications using APIs, this adds unnecessary complexity to supporting APIs, as there needs to be additional communication with the teams developing Appian features. As the stack becomes larger, it makes more sense to have the APIs that control these interactions exist on a dedicated platform. In scenarios where multiple applications need data, an ESB is very valuable to read, write, and update data. API endpoints written in MuleSoft serve various requests from different front-end applications. These endpoints then have a number of actions associated with the flow, perform the necessary data transformations and relay that data back to the original requester.

Proof of Concept

We developed a proof of concept to demonstrate this design pattern. The application is a simple Customer record where customers can have a number of documents associated with them. The goal of this system was to decouple the database and document storage from Appian so that other applications could potentially leverage those services via a shared MuleSoft ESB. The system contained the following elements:

- An Appian frontend to display Customer data
- A MuleSoft ESB with generic APIs for Appian and other potential services to hit
- A database hosted outside of Appian to hold information about Customer and their documents
- An Amazon S3 Bucket for file storage, such as Customer profile pictures and documents

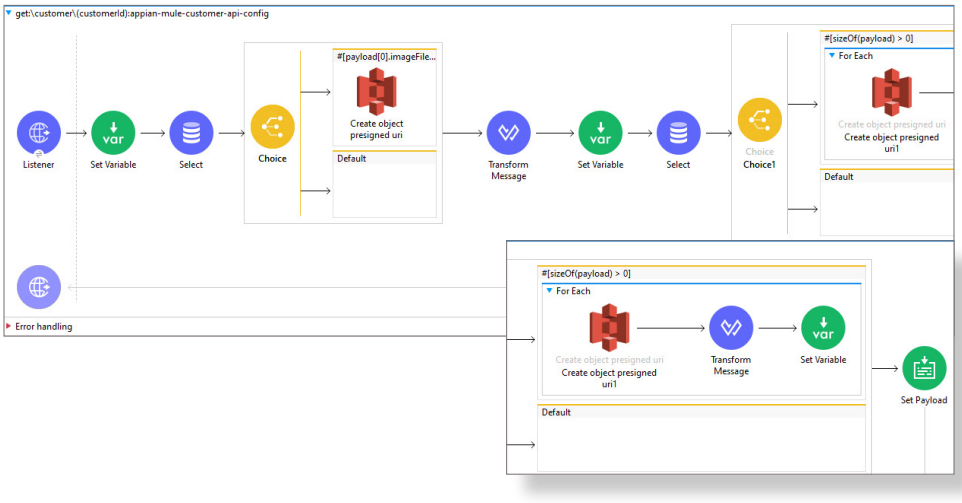
When displaying a Customer's profile, the flow is:

1. Appian calls an Integration rule designed to hit that MuleSoft Endpoint
2. The Integration makes a GET request to the MuleSoft ESB on the /customer/{ID} endpoint
3. The MuleSoft ESB receives the request and routes the call to the flow for that endpoint
4. The flow is called and performs the following actions
    - Makes a SELECT statement on the databases to retrieve information for that customer
    - Makes a call to Amazon S3 to retrieve signed URLs for the documents in the bucket associated with that customer
    - Formats all this data into a JSON format to be consumed by Appian
    - Sends the data back to the original requester
5. Appian receives the payload, parses the response body into a Record that can be read by Appian, and displays the data in the frontend application. Any image urls included in the body of the response are uploaded into Appian Documents via a process

When updating or deleting a customer, a similar flow is followed. Appian will make an integration call to exposed endpoints on MuleSoft to trigger dedicated flows for these actions. MuleSoft will perform checks using SELECT statements to ensure that the data that is trying to be updated/deleted exists, and if it does, similar nodes to connect to the database and S3 bucket are used. MuleSoft will then generate a success message to send back to the client.

On top of these flows that interact with the database, customer images, and customer documents, simpler versions that only interact with one of these services also exist, for if a client is created without any documents or just deleting one document associated with a customer.

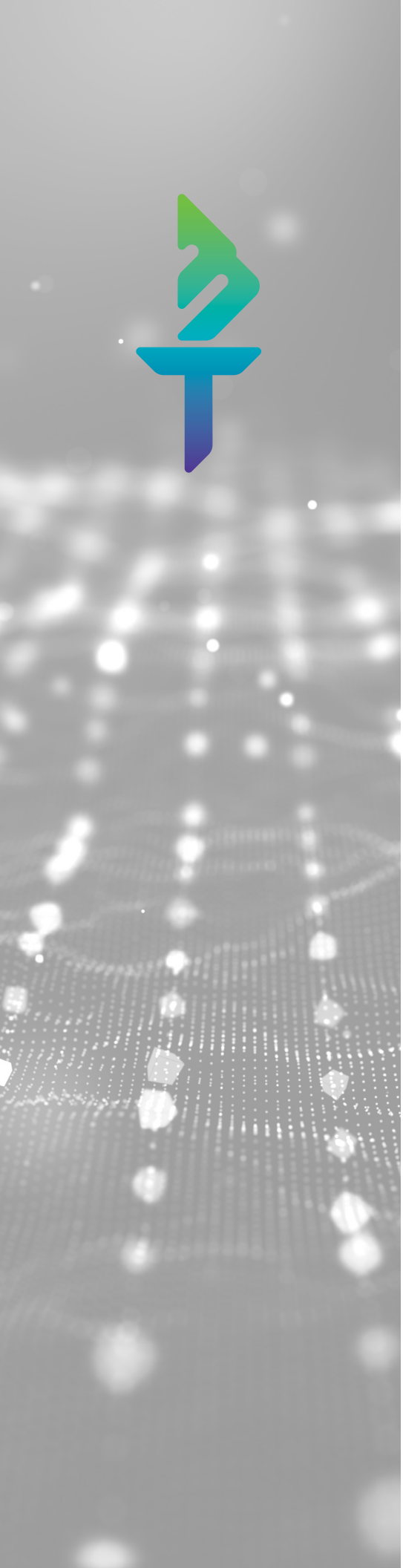
This very successful proof of concept demonstrates how Appian can become more flexible using MuleSoft. In scenarios where a customer has existing document storage, databases, and other possible backend applications, Appian can become an effective standalone frontend that can be plugged into MuleSoft with relative ease.



Limitations

Appian has a limit to the size of payloads that can be received, which places some restrictions on how to handle incoming documents and images. In the proof of concept, we got around this by supplying signed Amazon S3 URLs to Appian for the system to then interpret. However, a new issue then arises from this. Most document functionality within Appian can only be performed on Appian Document objects that have been created within Appian, not on URLs. Therefore steps need to be taken to load the documents from the URL into Appian before they can be displayed or downloaded by users. This adds some additional overhead and results in document duplication, as they are now being stored within both Amazon and Appian.

Since one of the goals of this approach is to limit the amount of data stored within Appian, an automatic process that removes unused documents on a set schedule would be recommended to keep things clean. It is important to hold that while documents are stored in Appian, the Amazon S3 document storage should be seen as the main source of truth since other applications may also be downloading and using these documents.



Best Practices

When writing out the specifications for the API, it is highly recommended that the files are written using OAS 2.0 or 3.0 (also previously referred to as Swagger). This API format is very useful in a Mule+Appian project as both systems natively support this format. This means that once the API specification is written up, it can be imported into MuleSoft to generate all the flows in the middleware and then imported into Appian to generate a Connected System object that has all the flows built in.

When creating a new integration using this Connected System, all of the endpoints/methods will be listed in a drop-down for easy access, and some important fields will be pre-populated once the Integration is created. Using this approach will ensure the systems are kept in sync and will reduce the overall development time of flows across both systems.

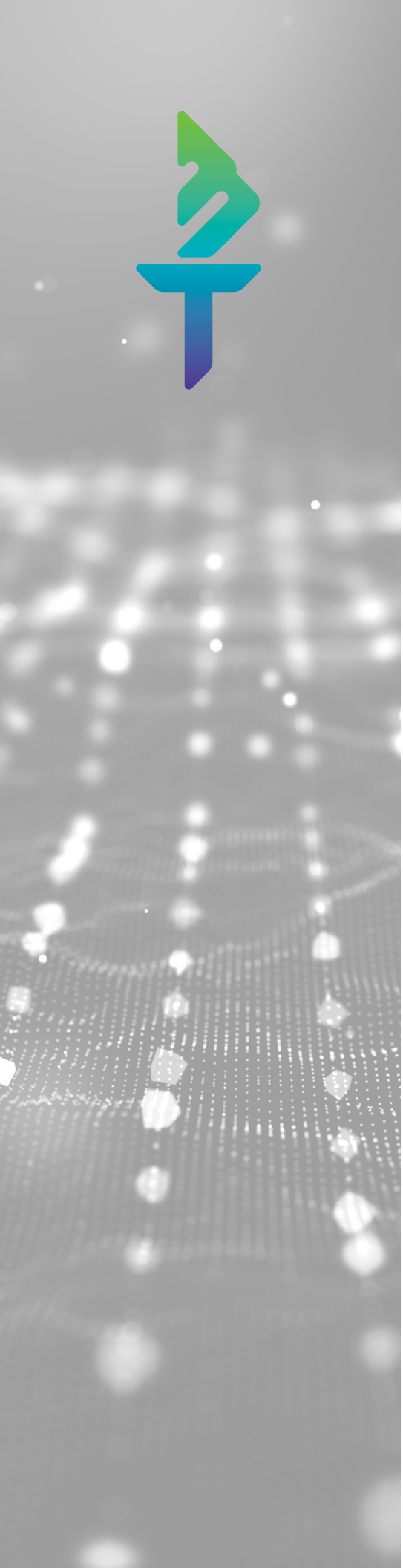
Appian has a feature to back its records through a web service API. Data syncing will load in data from the database through user-defined sync expressions, housing the data within Appian to make transactions quicker. This feature is particularly useful for large databases that can be expensive to query multiple times through GET requests. Any value that is updated within Appian will automatically be updated in the original database. However, data that is changed by other applications that access the data will not automatically be reflected within Appian, so it is important to include systems to inform Appian of a change to sync its data when needed.

We performed this by creating a Web API on Appian that listens for incoming connections and upon being reached, will perform a sync on a list of database ids in the body of the request. This Web API is hit by Mulesoft, with an HTTP Request node in any POST or DELETE flow that makes changes to the database. It is important to create an efficient syncing and querying model across MuleSoft and Appian that only updates the ids that are necessary, as an Appian timeout can result in unsynced data.

Case Study: Leading West-Coast Bank

In this section, we will discuss a client that Macedon has worked with who used an ESB, how the ESB was helpful, how they could have further improved their use of the ESB, and how Macedon hopes to use this knowledge in their own upcoming MuleSoft work. The primary benefit of using an ESB is that it centralizes many of a business's operations. This theory can be applied not only to the flows and applications being leveraged but also to the teams involved in building out those connections. An ESB, by design, links together many different applications and, therefore many different teams that control those applications. Complex flows without an ESB can be very difficult to coordinate due to the number of stakeholders involved.

The financial institution in question had an ESB and a team to support that ESB, however, we felt that the setup used didn't lean far enough into the strengths that an ESB provides. One of the major benefits of using a central-



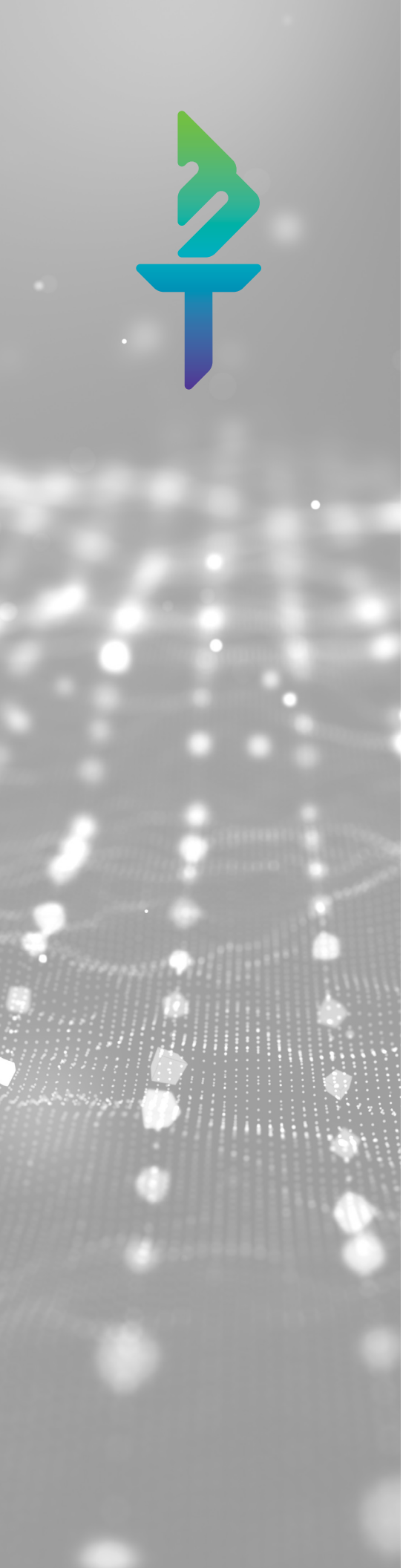
ized service layer is a reduction in the amount of custom work that needs to be done within the outer layer systems. While some services like database connectivity were moved to the ESB, others, such as accessing queues and certain file transfer protocols, were not, resulting in a messy structure where some custom work needed to be done both in Appian and presumably other frontend and backend systems that connect with the ESB. Centralizing this work into the ESB would have greatly reduced the amount of custom work needed in the outer layer applications.

MuleSoft has a massive catalog of modules, connectors, and APIs that other developers can leverage. Because of the fact that not all services at this institution were not centralized into the ESB, there were some applications that needed to be connected to by services like Appian, such as accessing a file-store and a message queue. Because of this, custom work needed to be done in Appian to create plugins to access these services. This resulted in many hours sunk into a custom service that only served to connect one system to another. MuleSoft has a deep well of plugins for many major technologies that can easily be dropped into a project, which could have made this feature far easier to implement. Additionally, even if the module didn't exist for one of the services, the development time put in to create a new module in MuleSoft would be better spent, as this module would serve to connect any service that integrates with MuleSoft to then talk to the technology in question. Mule's position as an integration layer, combined with the depth of available plugins, makes it a perfect candidate for this scenario.

While this financial institution did have an ESB of its own, it is important to note that just having an ESB is not enough to make communication smoother. Because of the state of this project, any new features needed the consultation of approval of many different stakeholders who controlled various different applications or pieces of data. The goal of an ESB is not just to centralize the routing of data but also to centralize and reduce the need to get so many individuals involved in changes. MuleSoft strongly encourages separating out its API structure into multiple layers. While the ESB is one interconnected web of systems with countless different applications and stakeholders, the flows within ESBs should also be designed in a way to be independent of each other and simple to allow for continuous development of the platform without the need to get so many people involved, which hinders development.

MuleSoft's dynamic scaling options would have proven very useful, as there was an issue with this client where spikes of high data usage were resulting in requests to the API being lost periodically. Because of this, a system where requests were automatically retried on failure had to be implemented across several applications. Mule's ability to scale up either vertically or horizontally when the demand requires it would have allowed developers to focus resources on other development, giving peace of mind that the system could handle any load that was required of it.

Another benefit to using an ESB, as observed by our team members, is that an ESB creates a single point of entry/exit for the institution's external flows. Clients like this one have incredibly high security expectations, which means



adding additional external endpoints could require significant lead time, which can cause blockers. Having the ESB be the single, trusted point of external communication makes adding new connections easier and safer.

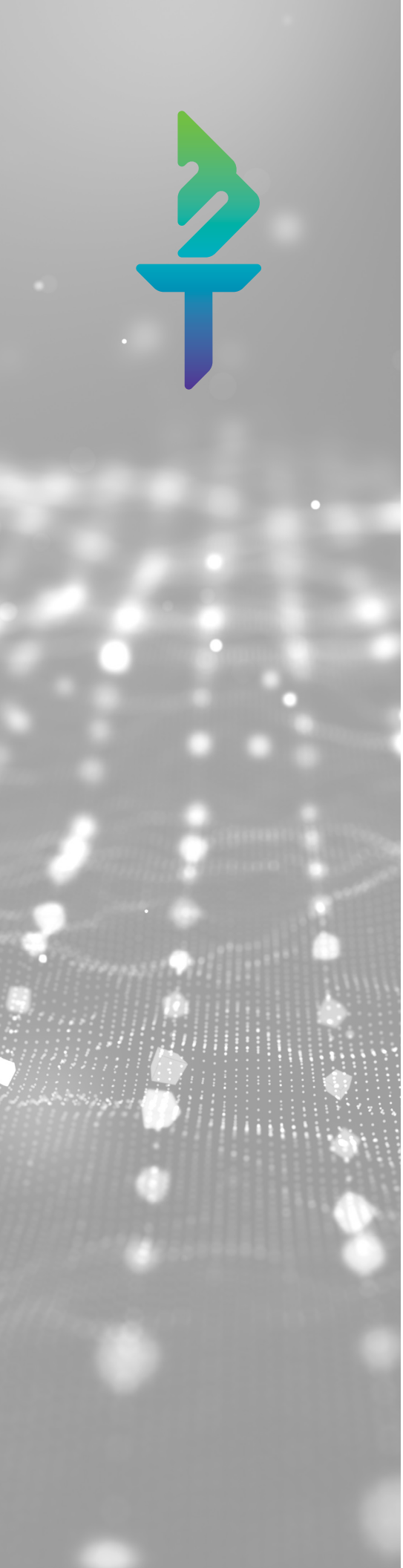
Conclusion

MuleSoft provides a new dimension to Macedon's Appian offerings, allowing Appian to integrate easily with countless different services that a customer may use. Using an ESB as a single point in a system with many interlocking endpoints can improve the efficiency of development and communication across teams.

For systems that include a varied tech stack, an ESB that allows each application to operate independently makes practical sense. This setup makes adding new services seamlessly, improves the scalability of the entire system, and allows for easier upgrades/replacements of existing services, all while reducing the overhead of coordinating between various teams for adding functionality. MuleSoft's wide pool of existing connectors to other services, with the addition of the ability to create and download custom connectors from the greater MuleSoft community, further enforces MuleSoft's emphasis on reuse and streamlined development.

MuleSoft being paired with Appian allows developers to leverage Appian's core frontend design functionality while also delivering a more flexible backend and database structure that gives greater opportunities for other applications to leverage that data. Developers with previous Appian experience should be able to pick up MuleSoft with relative ease, as many of Appian's core drag-and-drop principles from process development are also applicable to MuleSoft's API flow design.

Macedon Technologies will continue to use its existing Appian expertise and growing MuleSoft knowledge base to create new possibilities in the ESB space and develop further best practices for combining the technologies. Macedon fully believes in the potential presented by ESBs, and we believe that MuleSoft best suits our goals to create first-class software solutions that meet the needs and goals of clients of all kinds.



About the Author

Brandon Sultana is a New York-based Senior Consultant at Macedon Technologies. He initially focused on Appian development for internal tools but has since focused on researching new technologies as part of the Macedon Labs team. He has led many of the discoveries in MuleSoft and has since become a MuleSoft Certified Developer.

[1] Autoscaling in CloudHub:
https://docs.mulesoft.com/cloudhub-1/autoscaling-in-cloudhub

[2] Modern ESBs: Pros and Cons
https://arc.cdata.com/blog/20200114-modern-esb

[3] Rate Limiting: SLA-based policy
https://docs.mulesoft.com/policies/policies-included-rate-limiting-sla

About Macedon

Macedon is a recognized leader in intelligent automation and cloud data solutions. We have deep expertise with industry-leading technologies that we leverage to solve our clients' unique challenges. Our hybrid roles achieve better solutions faster than traditional development teams.

Contact: (571) 526-4281
info@macedontechnologies.com