Integration Service: Building a Bridge with Code

An Honors Thesis (CS 498)

by

Andrew Thomas

Thesis Advisor

Dr. Huseyin Ergin

Ball State University
Muncie, Indiana

March 2021

Expected Date of Graduation

December 2021

Abstract

Lead Sigma is a startup company in Kansas City that is trying to build a sales pipeline so prospective customers are responded to quicker, resulting in more deals. Their mission is to “drive revenue growth and create incredible customer experiences by empowering businesses with powerful and easy-to-use tools (LeadSigma, 2021).” One of the hallmarks of a powerful tool is its ability to easily integrate with other existing tools. I create an independent mechanism that allows for data to flow from Lead Sigma’s systems to external services as configured by their customers. It forms a bridge between the services, over which data can easily pass. It is resilient to downtime and minimizes the flow of unnecessary information and events by using modern cloud architecture techniques coupled with a microservice architecture. This is significant because it allows for Lead Sigma to quickly scale their integrations with little management.

Acknowledgments

I would like to thank Grayson Kuhns, whom I could not have implemented this architecture without. He is a software engineer at Lead Sigma whose cloud knowledge helped form the basis for the starting architectural decisions. Grayson reviewed every pull request to make sure the quality of the code was consistently good.

I would like to thank my professor and mentor, Dr. Huseyin Ergin, who consistently worked with me to help me meet certain class requirements despite having such a unique project. It was especially meaningful to see the amount of time he has dedicated to ensuring that I, as well as the other capstone teams could work on useful projects for real companies.

I would like to thank my two teammates Alan Bauer and Noah Ziems who allowed me to work on this service independently while they worked on a separate concrete integration. They were excellent partners in the previous semester when I worked with them on the sales activity service, and their hard work and dedication helped motivate me to do the best work possible.

Lastly, I would like to thank my girlfriend Abbigail Wainscott who has remained supportive the entire year, despite most weekends and nights being dedicated towards this endeavor.

Table of Contents

Process Analysis Statement

In this process analysis statement, I describe how I built an integration service for Lead Sigma, a start-up based in Kansas City that is trying to help businesses respond to customers faster. The integration service acts as a bridge between Lead Sigma's software and other third-party software. As an event happens in Lead Sigma, it should show up in the other software. My system orchestrates that process by controlling how the data is connected so it always goes to the correct place. As it is a very technical service, I have included a Glossary to aid the reader.

Abstract Implementation Architecture

Before tool choices and implementation details were discussed, an abstract architecture and process was created. This helped shape decisions regarding which tools and services to use. From a high-level overview, an event should trigger a change or an update in one of Lead Sigma's services, most likely the Sales Activity Service. This service should then notify the integration service in a fashion that would allow for the moving of data to external services. Distinctions should be made based on the event type and which integrations an organization has enabled. If an integration only handles aggregate events, it should not be sent discrete events. If an organization does not have a certain integration enabled, data should not be sent to that integration.

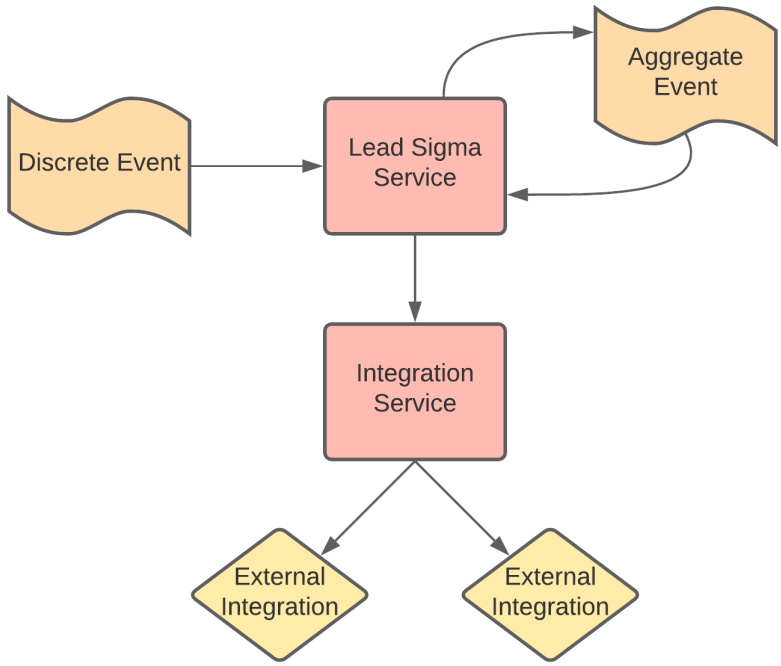

Figure 1: Abstract Architecture Diagram (LucidChart)

Choice of Tools, Software, and Languages

To implement the solution, I first worked with engineers at Lead Sigma to select the language and tools to be used. We decided on using Ruby on Rails for the microservice's language and framework. There were several reasons for this choice. Firstly, we knew we would be interacting with Amazon Web Services' APIs for the implementation. While this API is accessible via nearly any programming language, Amazon also builds and supports SDKs in a few widely used languages to make development of new software faster, and easier. At the time of writing, the SDKs were limited to these languages/frameworks: C++, Go, Java, JavaScript, .NET, Node.js, PHP, Python, and Ruby (Amazon Web Services, 2021). This quickly narrowed our language candidates. While I am more proficient in Java and PHP, much of Lead Sigma's existing codebase is written in Python and Ruby, so to limit the effort required for programmers to work on the different microservices within Lead Sigma, we chose to not introduce a new language to their ecosystem. Ruby was chosen over

Python due to the lead engineer's experience with it, as well the existence of its Rails framework, which allows for the quick and efficient building of APIs. This was weighed against the long-term costs of using a dynamically typed language regarding the extra development effort required in testing what a compiler would usually catch, as well as difficulty scaling the service after reaching a large codebase size. Preference was given to speed, to allow Lead Sigma to satisfy its customers more quickly, and to remain competitive as a startup. The code scalability was believed to be a non-issue as they are using the microservice architecture and this Integration Service should have few responsibilities, preventing it from growing to an unmaintainable size, if properly modularized.

MySQL was chosen as the database management system. A ruby gem by the name of "mysql2" allows for easy connections to a MySQL database, and AWS's Relational Database Service has a MySQL engine which would make MySQL viable when deployed in a production environment, and compatible with local development (Mario, 2021). As a relational database, MySQL can keep referential integrity between resources, and provides domain constraints to ensure entities their defined structure (Oracle, 2021).

GitHub was selected to host the central version-controlled repository. Lead Sigma has been using GitHub for its other services, so switching to a different solution would unnecessarily increase complexity. GitHub allows for teams to securely store their repositories, and track changes via Git. It has powerful continuous integration features grouped under GitHub Actions, that allow for tests and linting to run when code is pushed to the repository. Furthermore, it provides an easy-to-use web interface for reviewing code changes when submitted via a pull request (GitHub, 2021). These features are sufficient for Lead Sigma's needs.

To keep track of project decisions and progress, Jira was selected for issue tracking. Jira is an industry standard that has many powerful features including integrations with GitHub. It was primarily selected to organize, prioritize, and monitor tasks for the integration service. This would allow for a history of progress to be seen. It also adds redundancy to the next steps covered in meetings, keeping everyone on the same page

and accountable. For a single person team, this was not necessarily need, but its benefits were ruled to outweigh the costs of spending time keeping it up to date (Atlassian, 2021).

To provide a more scalable, fault tolerant service, various AWS services were selected to help provide the integration service's functionality. To allow for quick, independent development of different integrations, it was decided that integrations would be deployed via AWS Lambda. This provides serverless functionality so Lead Sigma would only be charged by usage of the integrations, rather than the costs of keeping the integrations running on their own reserved servers. It would be financially unfeasible to host the 70+ different integrations Lead Sigma has planned on separate servers. In addition, some integrations may not be frequently used, so paying to keep the server running would be a waste of money for a large portion of the time. AWS Simple Notification Service (SNS) Topics, and Simple Queue Service (SQS) Queues were selected to form the bridge between the Sales Activity Service and the integration lambdas. Every organization would be given a topic for each event type in the system, i.e. an organization would get a discrete topic and an aggregate topic. An integration's lambda would poll from a single attached SQS queue to retrieve events. When an organization enabled an integration, the associated SQS queue would subscribe to the organization's SNS topics relevant to the integration. This would allow for data to be pushed to an organizations topic, which would be fed into an SQS queue, which would then be consumed by an integration lambda which would ultimately handle putting the event into the third-party system. Using topics and queues keeps the system highly available because AWS scales them as needed to keep them running, without any user input or maintenance. It improves reliability as well. If a lambda function fails to handle an event, the event will stay in the queue for up to 14 days so no data will be lost if maintenance occurs, or a programming error results in a failure. This process will be further explained in the next section.

Concrete Implementation Architecture

With these tools in mind, we created a concrete plan for the architecture. When an event is triggered, either via an external change for a discrete event, or an internal process for an aggregate event, it would update the

Sales Activity Service, which would handle the event, save it in AWS RDS and then send it to the SNS topic associated with the event's organization and type. The integrations that an organization had enabled would be represented by a subscription between their SQS queue and the SNS topic. The event data from the SNS topic would be pushed to the subscribed SQS queues which would then feed the integration lambda (Amazon Simple Notification Service, 2021). This process keeps the integration service that I would be creating, separate from the flow of data, allowing for unintended downtime without loss of data, as the connections would already be established for existing integrations and subscriptions. The integration service would then handle the SNS topics for each organization, and the subscription of SQS queues to them depending on the organizations' preferences. This would allow for the quick scaling on integrations and organizations without increasing the code complexity of other services.

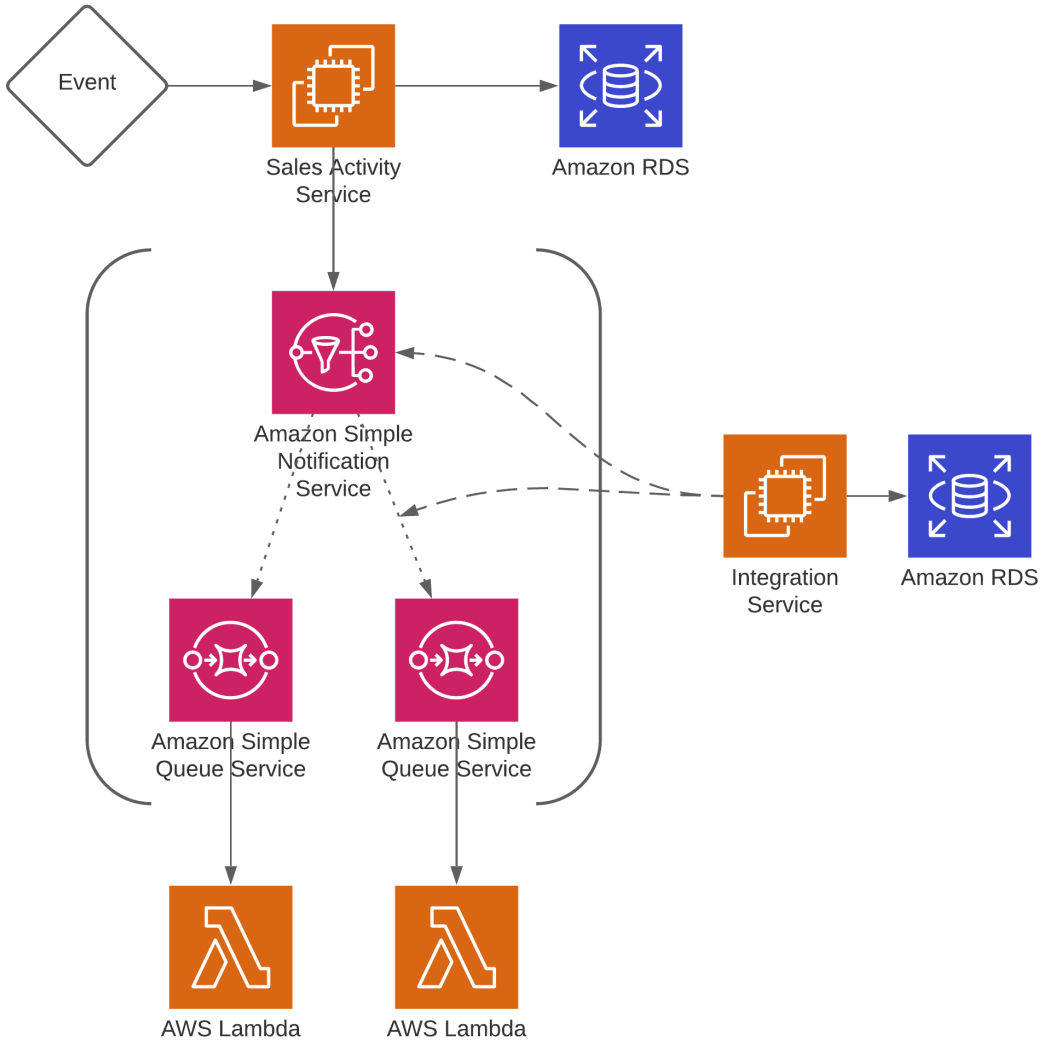

Figure 2: Concrete Implementation Diagram (LucidChart)

Implementation Details

API Structure

After the concrete architecture was decided upon, I was left to implement it as I saw fit. The first thing I did was decide upon an API structure. This would determine what things the service could handle and document the necessary requests and expected responses. I used Postman and Swagger documentation to give other

developers quick access to the API, but I am omitting them here for brevity. They may be found within the documentation repository referenced in digital supplements.

Firstly, I defined the key resources and entities. These would help me shape the API and understand what was required. **Integrations** are the SQS queues that Lead Sigma has set up to enable the flow of data to third party systems. **Integrations_Organizations** define the integrations that each organization has enabled. This follows the typical naming convention of a many-to-many relationship where an integration can belong to multiple organization and organizations can have multiple integrations. The entities are named in their plural form in alphabetical order (Chin, 2014). **Events_Integrations** define the event types that the specific integration accepts. **Topics** define the SNS topics that an organization has. These formed the backbone of the API. I followed standard REST API practices and defined routes to view an index of the resources, view an individual resource, create a resource, update a resource, and delete a resource (Au-Yeung, 2020).

POST requests will return a 201 [Created] status code if the resources could be successfully created with the response body detailing information about the resource. If there is a validation error it will return a 422 [Unprocessable Entity] status code, and an error message denoting what went wrong. GET requests will return a 200 [Success] status code with the response as an array of objects if it is an index route, and as a single object if it is a *show* route for a specific resource. If the resource cannot be found for GET, UPDATE, or DELETE requests, the API will return a 404 [Not Found] status code. An UPDATE route will return a 200 [Success] status code if the resource was updated, with a response body containing the updated object. Like the CREATE request, the API will return a 422 [Unprocessable Entity] status code with a detailed error message if something wrong was detected with the update. A DELETE request will return a 204 [No Content] status code upon successful deletion of a resource. These status codes follow best practices for REST APIs and allow them to be quickly understood by other developers and for successes and failures to be more easily handled in different systems (Mozilla, 2021). In addition to the status codes, to follow standard practices, the

API consumes and emits JSON to encode information about the objects and errors. Index routes can be filtered with the use of query parameters, i.e. key-value pairs that appear after a question mark in the URL.

The routes are as follows, with the endpoint being prefixed by https://[integration-service-domain]

Integrations

Request Type	Endpoint
POST	/integrations
Creates an integration in the system. A name, SQS queue ARN and event types should be passed.	
GET	/integrations
Lists the integrations known to the system.	
GET	/integrations/{integration-id}
Returns the integration with the integration-id passed as a route parameter.	
PUT/PATCH	/integrations/{integration-id}
Modifies the integration with the new data passed in the request body.	
DELETE	/integrations/{integration-id}
Deletes the specified integration.	

Integrations Organizations

Request Type	Endpoint
POST	/integrations/{integration-id}/organizations
Connects on organization to the specified integration in the system. The organization ID should be passed in the request. This will subscribe the integration's SQS queue to the topics of the organization that correspond to the event types that the integration accepts.	
GET	/integrations/{integration-id}/organizations
Lists the organizations that have the specified integration enabled.	
GET	/integrations/{integration-id}/organizations/{organization-id}
Returns the integration organization specified.	
PUT/PATCH	/integrations/{integration-id}/organizations/{organization-id}
Modifies the integration organization with the new data passed in the request body. If an enabled flag is set to false, it will unsubscribe the SQS queue from the organization's SNS topics, and if reenabled, it will subscribe them again.	
DELETE	/integrations/{integration-id}/organizations/{organization-id}
Deletes the specified integration organization. This will unsubscribe the integration's SQS queue from the organization's SNS topics.	

Events Integrations

Request Type	Endpoint
POST	/integrations/{integration-id}/event_types
Adds a new event type for an integration to accept. Can be discrete, aggregate, or a new event type added in the future. When this is created, it will subscribe the associated integration's SQS queue to the topics associated with the new event of the subscribed organizations.	
GET	/integrations/{integration-id}/event_types
Lists the event types an integration has enabled.	
GET	/integrations/{integration-id}/event_types/{event-type}
Returns the event type of that integration.	
PUT/PATCH	/integrations/{integration-id}/event_types/{event-type}
While this route exists, there is nothing that can currently be modified, but in the future if a required attribute is added, or similar, this route is set up to be useful.	
DELETE	/integrations/{integration-id}/event_types/{event-type}
Deletes the specified event integration. This will unsubscribe the integration's SQS queue from all topics of that event type.	

Topics

Request Type	Endpoint
POST	/topics
Creates a topic entity in the database. An organization ID, SNS Topic ARN, and event type must be passed.	
GET	/topics
Lists the topics known to the system.	
GET	/topics/{topic-id}
Returns the topic with the topic-id passed as a route parameter.	
PUT/PATCH	/topics/{topic-id}
Modifies the topic with the new data passed in the request body.	
DELETE	/topics/{topic-id}
Deletes the specified topic.	

Organizations

Breaking from typical REST principles for the sake of convenience, it was decided to add organizations routes that do not tie to specific resources within the service, but instead handle the automated set up or tear down for new organizations.

Request Type	Endpoint
POST	/organizations
This will set up the organization's resources in AWS and store them in the database. It will create the SNS topics for each event type currently in the system and return them in the response body.	
DELETE	/organizations/{organization_id}
This will delete the resources associated with the organization. It will unsubscribe the SQS queues from the SNS topics and then will delete the topics from AWS.	

Database Schema

The database schema needed to be created to ensure the necessary information was stored and could be accessed as required. Before I discuss the schema itself, I will define some of the database terms used to help the reader understand the decisions made. A table holds the attribute definitions for an entity type. These attributes have different types. Tables can be related to each other via foreign keys which MySQL uses to establish referential integrity. Varchar is a type that allows the schema to define the maximum number of characters that attribute can hold. Tinyint is a type that allows for storing of integers up to one byte in length. Tinyints are signed by default so they can store values from -128 to 127. Datetime is how MySQL stores a date and time in a field (Elmasri & Navathe, 2016). A universally unique identifier (UUID) is a 128-bit number that for practical purposes can be considered unique to the world, as there are 2^{128} different UUIDs possible.

Topics

Topics use an ID as their primary key. This is stored as a varchar(36) for use as a UUID in hexadecimal form. An organization_id is stored, which is a UUID for an organization in Lead Sigma's system. An event_type is set to a tinyint, which is translated into an event type defined in an Enum at the application level. A sns_topic_arn is a varchar(400) which stores the identifier AWS gives to SNS topics. Created_at and updated_at are datetimes that store information about when the topic was created or last updated. A unique index is placed on the organization_id and event_type tuple so each organization can only have one topic associated with each event type. A unique index is similarly placed on the sns_topic_arn to prevent the same AWS SNS topic from being used more than once.

Integrations

Like topics, integrations use an ID as their primary key, stored as a varchar(36). They store a name for the integration with the varchar (255) type. An sqs_queue_arn is stored with varchar (400). This uniquely identifies a SQS queue within AWS. Created_at and updated_at are stored as datetimes. A unique key is placed on the sqs_queue_arn to prevent the same SQS queue from being associated with multiple resources.

Events_integrations

Events_integrations use the integration_id and event_type as a composite primary key. This ensures that each integration only has a specific event associated with it once and removes the need for another ID to be used as a key. The integration_id is a varchar(36) which is used as a foreign key to relate to the integrations table. The event_type is a tinyint. Events_integrations store the created_at and updated_at fields for historical purposes as datetimes.

Integrations_organizations

Integrations_organizations use the integration_id and organization_id as a composite primary key. The integration_id is a varchar(36) and is used as a foreign key to relate to the integrations table. It stores an enabled flag as a tinyint to allow an organization to disable an integration without fully deleting it from the system. Like the other tables, it stores created_at and updated_at to give a sortable timeline.

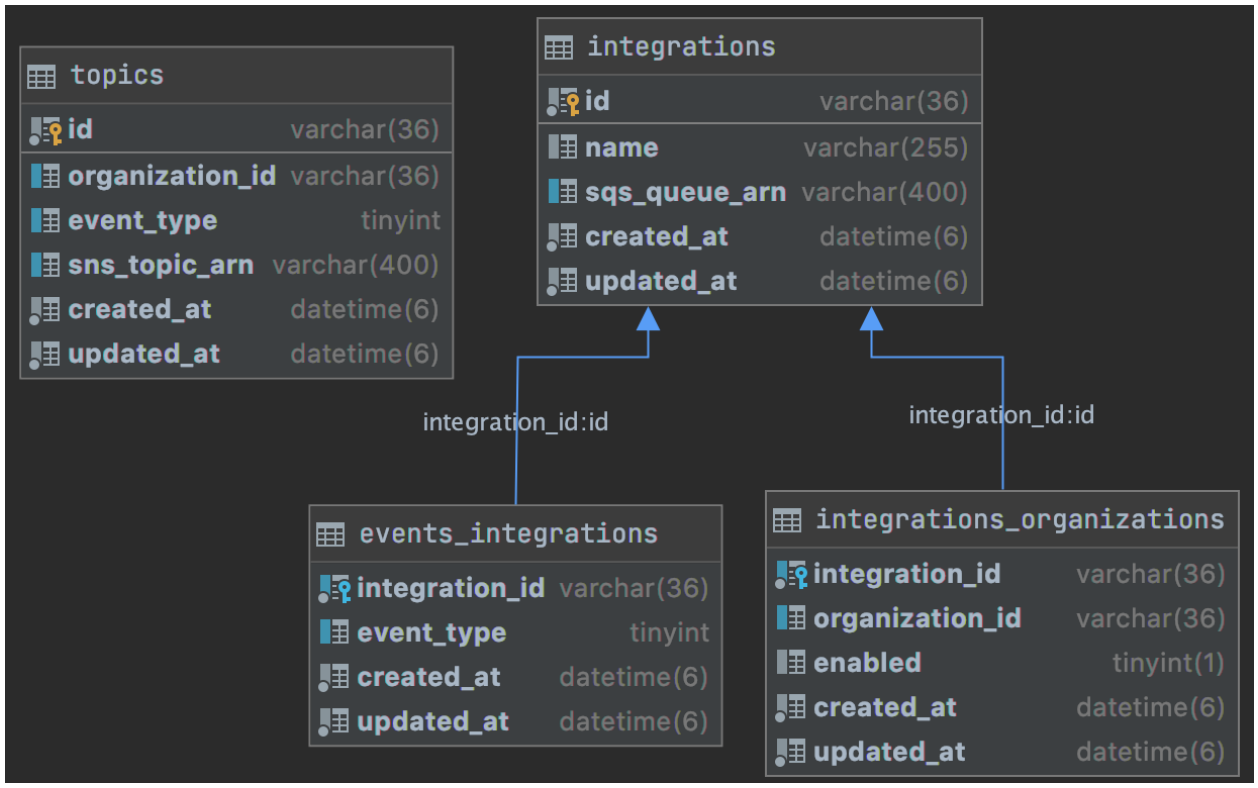

Figure 3: Database Diagram (DataGrip)

Creation Process

Creating the CRUD API

After the API structure and Database Schema were created, the Ruby on Rails implementation could begin. To allow for easy development set up, Docker was utilized to run the ruby interpreter and MySQL 8.0 database. This required a `docker-compose.yml` file and `Dockerfile` to be created. Once this was done, it could be ensured that the project could be set up on any environment and still run the same (Docker, 2021). Next, the database was configured, and the connection was enabled with the mysql2 gem. Following that, rails scaffolding was used to set up the basic code structure. Rails generates model, controller, route, fixture, tests, and migration files when a scaffold command is ran. For example, `rails g:scaffold integration name:string sqs_queue_arn:string` would set up the boilerplate code for the integrations entity. This allowed for more time to be spent on the implementation details rather than the

trivial set up. The generated migrations needed to be overwritten to allow for UUIDs instead of the default auto-incrementing IDs. The size of strings could be adjusted as well. A migration could be run with rails db:migrate, which would update the database according to the latest migrations (Ruby on Rails, 2020). This allowed for early testing to make sure the code was generating the database in the same way the schema defined.

Following the migrations, the models and controllers were updated to provide the basic create, read, update, and delete functionality. Validations were added to the models, so the code could catch errors before sending faulty data to the database. These were carefully tested with model unit tests. Tests were written to ensure that the standard entity was recognized as valid, and that breaking one of the intended validation rules would set an error to be returned. For example, it was specified that event types within a topic should be unique to an organization. A test was written that added a discrete topic to an organization, then created another and asserted the model highlighted that the newly attempted creation would fail. To ensure that it was correctly scoped another test was written with two different discrete topics, but each were associated with different organizations. The test asserted that both were valid. The controllers for events integrations and integrations organizations needed to be modified alongside the routes to ensure the code would accept the nested routes specified in the API structure. These were associated with a JBuilder view which allowed for the response json to be specified. Controllers were tested via integration tests, that would start at the http request level and assert details about the response from the API as well as that the resource counts were updated correctly. Tests were written for each endpoint to ensure they succeeded when expected, as well as handled error appropriately. For POST requests, tests were written to ensure that a new resource was added to the associated table when successful. DELETE requests had tests that asserted the number of resources in the database decreased by one.

Continuous Integration

To ensure code quality was maintained, I set up continuous integration with GitHub Actions. This would perform a variety of tests against my code whenever I pushed it to the remote repository on GitHub (GitHub, 2021). First it would perform linting of my code with Rubocop. This would check that my code style matched the standards set by the Rails community and would reject my changes if it did not. This included checking things such as method or file line count, the use of symbols in hashes, and naming conventions (Batsov, 2020). After this, the action would run the rails tests. If these failed, then my new code broke something, and thus would be rejected. Finally, it would use the simplecov gem to determine code coverage. This is the percentage of lines in the codebase that were touched when running the tests. If a line is not touched, that means the tests never triggered code to reach that section. If the code coverage was low, the merge would be rejected, to ensure that an adequate number of tests were written (SimpleCov, 2021).

Building Integrations

With the basic set up in place, the process of integrating the different AWS services could begin. After reading the documentation, and getting IAM user permissions, I used the AWS web client to set up sample SNS Topics and SQS queues to feed the API (Amazon Simple Notification Service, 2021). With sample topics, integrations, and event integrations set up, I began to work on the integrations organizations to manage the subscriptions between queues and topics. I installed the AWS Rails gem and the AWS SNS gem to give me access to AWS's SDK (Amazon Web Services, 2021). It became apparent that I would need to provide credentials to manipulate resources within AWS from within the API. To securely do this, I opted to use Rails encrypted credentials. This allowed me to put the encrypted credentials into version control, but they could only be decrypted with a secure master key (Ruby on Rails, 2020). After this was done, I wrote code to utilize a SNS client, that had a method to get a SNS topic from AWS. The returned topic was a rails object that had a method to subscribe an SQS queue with its ARN. After setting some default configuration values, the code was written to subscribe an integrations SQS queue to organization's SNS topics. The rails hook system, allowed for consistency between the database, and the external resources. I utilized the '`before_create`'

hook on the integrations_organizations model to bind the subscription of queues to topics within AWS to the creation of new rows in the API's database. Similar methods were written to unsubscribe queues from topics on delete, or upon update given the enabled flag was changed from true to false. These integration methods were also written for the organization routes, and the event types to match the API structure defined above.

Properly unit testing the integration between the integration-service and AWS proved difficult. The SDK allowed for the injection of a client into the calls to provide typical mocking functionality. Rails does not have a simple mechanism for typical dependency injection, so I devised a hacky solution that made use of the environment variables the system had set, and rails autoloaders, but after reviewing this with a software engineer at Lead Sigma, we decided it was too brittle, and too far removed from the core code and would likely result in bugs and errors in the future. Instead, I discovered that a global config value for each AWS resource could be overwritten at runtime during tests to set the API to return defined stubbed responses (Amazon Web Services, 2021). By doing this, the code could be tested to ensure it was working correctly, without making potentially expensive calls to update actual AWS resources.

After finishing the AWS integrations, it was decided that it would be useful for the integration-service to integrate with another Lead Sigma microservice known as the secret service. This service would handle the OAuth flow and management of organizations' credentials for the different integrations and would only be accessible from within Lead Sigma's virtual private cloud for obvious security reasons. The details of this service are not pertinent to the creation of the integration service, so I will omit them. Instead, I will highlight the few functionalities of the secret service that the integration service would work with. Firstly, when an organization enabled an integration, it would be given an authorization code, this was added to the expected request body of the integrations organizations POST route and would then be sent to the secret service to allow for the generation of access and refresh tokens. As this authorization code is not tied to a model within the database, the code for sending this was instead kept in the controller, an exception to the general rails standard of keeping controllers slim. Riding the wave of progress of getting this route integrated, I wrote

code to delete an organization's access tokens and refresh tokens from the secret service if they deleted an integration-organization. This created a notable distinction between updating the enabled field to false, as reenabling would not require another authorization code, but readding after deleting would. From the user's point of view, they would have the option to stop an integration from sending data for a period, to perhaps fix an issue, and then easily enable it when they finished. However, they would still be given the opportunity to completely remove access and stop the flow of data if that was desired as well.

Like the problems encountered with unit testing the AWS integrations, there was not a simple way to inject a dependency into the controller that would handle calls to the secret service. Instead, other ruby developers created the webmock gem that listened for outbound HTTP requests and returned stubbed data instead of making the actual API call during tests (Blimke, 2021). This allowed for quick additions to the testing suite to stub the responses for the secret service and test that the code to handle the requests acted as expected with the new integration in place.

Authentication & Authorization

At this point, the main functionality of the integration service was complete. To finalize this "minimum viable product" version, i.e., the version with just the necessary base features, authentication and authorization was added to provide security for each organization. Continuing to follow the microservice architecture principles, authentication was delegated to another service, AWS Cognito. Cognito handles account credentials, and once authenticated with a username and password, or a third-party account, it would grant a JSON Web Token (JWT). This JWT could be decoded to gain information about the user who sent the request, as well as determine its validity (Amazon Cognito, 2021). To ensure only authenticated requests reached the integration service, it was placed behind AWS API Gateway, which would validate the JWT from Cognito before passing the request on. This provided many benefits. Firstly, code for authenticating a JWT would not need to be duplicated across the various microservices. Secondly, because the JWT could be trusted, it removed the need to send potentially slow requests to AWS to retrieve JSON Web Key Sets for the valid

decoding of the JWT. Instead, it could be decoded without a public key, and trusted that it had been validated before it was received. This provided the final benefit of being able to use fake and modified JWTs for testing purposes.

With a user successfully authenticated, thought could be moved to the authorization process. Information about users and organizations was stored in the sales activity service, so rather than try to reimplement the same set up, resulting in duplicated data, I decided to integrate with the sales activity service instead. This was a sizeable endeavor, as it required going through the client credentials OAuth flow through Cognito to gain access to the sales activity service. To retrieve an access token for use with the sales activity service, the integration service would first need to exchange its private client credentials for a token via AWS Cognito. This access token could then be passed to the sales activity service through an authorization bearer token header (Recordon, 2012). The sales activity service would then need to recognize the integration service as a valid client before returning any data.

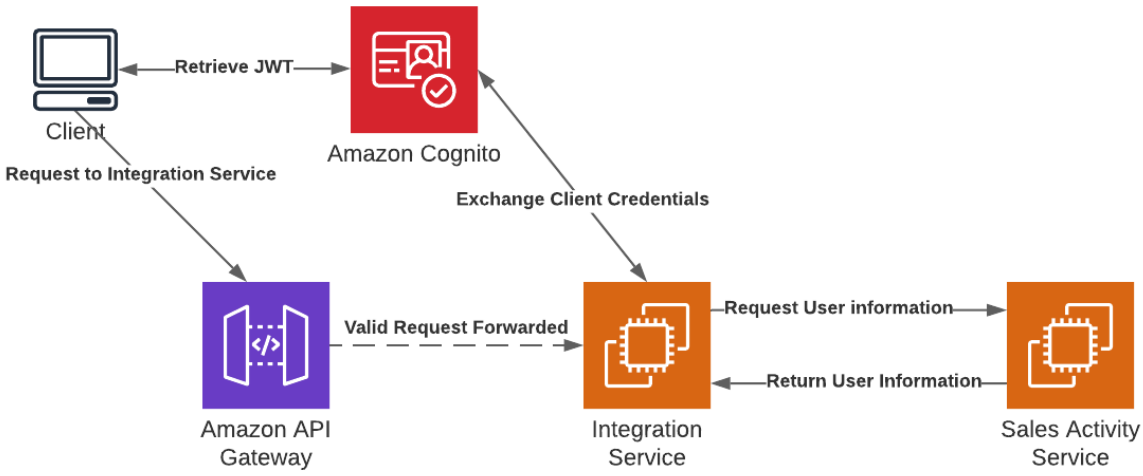

Figure 4: Integration Service OAuth Flow (LucidChart)

After finishing the OAuth flow code, I began working on getting the actual user information from the sales activity service. The JWT decoded from the user, contained an identifier for the user by the name of ‘sub’.

This was the Cognito ID (Amazon Cognito, 2021). I used this ID to send a request to the sales activity service, filtering by the Cognito in the query params, i.e., https://sales-activity-service-url/users?cognito_id={sub}. Filtering on the user's route had not been built into the sales activity service, so I opened a pull request with the added functionality, which Lead Sigma promptly merged. This route would return an array of the single user that had the same Cognito ID as found in the sub field of the JWT. Within the user response contained fields denoting the user's roles, including whether they had admin privileges. As the response contained the organization ID of the user, any requests made by a non admin user would automatically set the organization ID to their actual organization ID rather than what they passed in the request body. Admins could override this behavior. With the sales activity service integrated, I added additional code to publish all an organization's events to an integration's SQS queue upon subscription. This would ensure that all the data in Lead Sigma would get sent to the third-party, even if it was enabled well after the initial usage of Lead Sigma's services.

This allowed me to begin the actual authorization process. I opted to use the Pundit gem, which allows for the separation of concerns from authorization and the controller (Varvet, 2020). All code defining what a user was authorized to use, was placed in policy classes, while the controllers remained largely unchanged. The following resources were set to admin use only: topics, organizations, integrations, and events_integrations. This would prevent normal users from messing with Lead Sigma's resources and other organizations' data. Integrations_organizations had more granular restrictions. Users were given access to all the routes, but they would receive an unauthorized exception if they tried to access a resource that was not owned by their organization. The index route was scoped for users so it would only return the resources associated with their organization.

These changes required sizeable changes to the tests, as well as new tests to ensure users were authorized correctly. The controller tests, which up to this point had not included an authorization token, all failed when I first ran them, as expected due to the new authorization system. To remedy this, I grabbed

authentic JSON from the sales activity service for the users route, and modified it to have an admin user, and a regular user, and saved these responses to separate files within the test fixtures. I then encoded fake JWTs for a user and admin. Using webmock, I stubbed responses from Cognito, and returned the different user JSON from the sales activity service depending on which JWT was used (Blimke, 2021). This allowed me to add the JWTs to the authorization header of the controller tests, and the requests would succeed after passing authorization. A few integration tests were written for the controllers to make sure 403 [Forbidden] status codes were returned if the user did not have access. Per a resource in the Pundit documentation, I also wrote unit tests for each Pundit policy for each request, testing that admins, users, and users of a different organization were only authorized when expected (Bougie, 2014). I spent effort to make these tests as simple to write as possible, so they could be easily read and used. An example of this can be seen with the integrations_organizations policy test. I have hidden some of the actual implementation as it is irrelevant to the general idea.

```
test name 'admin policy' do
  assert_permit @admin, record :integrations_organization, action :index
  assert_permit @admin, @integrations_organization, action :show
  assert_permit @admin, @integrations_organization, action :create
  assert_permit @admin, @integrations_organization, action :update
  assert_permit @admin, @integrations_organization, action :destroy
end

test name 'user policy' do
  assert_permit @user, record :integrations_organization, action :index
  refute_permit @user, @integrations_organization, action :show
  assert_permit @user, @integrations_organization, action :create
  refute_permit @user, @integrations_organization, action :update
  refute_permit @user, @integrations_organization, action :destroy
end

test name 'user owner policy' do
  assert_permit @user_owner, record :integrations_organization, action :index
  assert_permit @user_owner, @integrations_organization, action :show
  assert_permit @user_owner, @integrations_organization, action :create
  assert_permit @user_owner, @integrations_organization, action :update
  assert_permit @user_owner, @integrations_organization, action :destroy
end
```

Figure 5: Policy Test Code Screenshot

This concluded the MVP portion of the service, but before I could be satisfied, I refactored my code, to ensure I would be delivering high quality software. I created classes for the different services I integrated with the integration service and put them into AWS and Lead Sigma directories to help divide them. This included a class for Cognito, SNS Topics, the sales activity service, the sales activity user, and the secret service. This moved some duplicated logic out of the models and implemented the single responsibility principle of quality object-oriented programming. Next, I moved logic about the permitted attributes for each request from the controller to the pundit policy. Not only did this help slim the controller methods, but it provided ways to easily differentiate which attributes were permitted for each user type, as well as each request type, on an individual resource basis. While editing the policies, I noticed there was a lot of duplicated logic to limit requests to admin users. To remove this duplication, I created an "AdminOnlyPolicy" class, and extended it for each of the admin only policies, which allowed me to remove most of their code, while keeping the functionality (Martin, 2008).

Impacts

John Moses, the lead software engineer at Lead Sigma claims the integration service will "reduce development time for future integrations and reduce time spent supporting [them]," and thus reduces costs. Furthermore, as they did not have to pay for this high level of engineering, they could afford to hire content designers, marketers, and user experience experts to further grow their business. It is difficult to assign an amount to the value this service adds, but its architecture allows it to easily scale and support their needs for years to come. With its simplistic design, it will be easy to add a notification system atop the SNS topic and SQS queue structure to add features past the integration of third-party systems. Furthermore, with 96% code coverage, with 14 unused and therefore untested lines, the code will be easy to modify for future use, while ensuring that it continue to work properly.

It can serve as a model for other engineers as well. It shows that it's possible to build scalable architecture that can be supported at the start-up level, despite potentially massive numbers of integrations.

This helps create a better-connected web that will continue to benefit end users as they will not have to be tied to the single system where their data lives. By easily scaling the number of integrations that can be supported, users will be empowered to use the tools that work best for them, not just what a certain service claims are best.

Glossary

API

An Application Programming Interface (API) is the interface that defines what calls can be made to a resource, and how it affects that resource. A web API often defines four types of requests. POST is a request that should create a new resource in the system. GET is a request that retrieves information about the resource from the system. PUT/PATCH is a request that modifies data in the system. DELETE is a request that removes a resource in the system. These types are used alongside routes to define what resources are available for controlled, external access.

Architecture

Architecture describes the systems on the software and how they work together. Architectural decisions are made within a singular codebase in terms of modularity, code structure, and file hierarchy. Architecture at the system level describe how different microservices interact with each other, as well as how different third-party systems interact. Cloud Architecture is how different components of a cloud service are used in tandem to improve security, increase reliability, and ultimately deliver the intended service to a consumer.

Authentication & Authorization

Authentication is the process in which the system identifies a user. Authorization is the process in which the system determines whether to give the identified user access to the resources requested. These are the sine qua non of system security.

AWS

Amazon Web Services (AWS) is Amazon's cloud platform which allows for organizations to use Amazon's servers, services, and tools, to build and deploy their solutions for world-wide use. Lead Sigma hosts their services on AWS and utilizes many of the different tools to create a reliable, secure, and scalable architecture.

Continuous Integration

Continuous Integration is the modern software development practice of integrating a developer's code into a larger codebase. Continuous integration allows for the frequent running of tests to make sure the code is still functioning properly. It typically uses tools to ensure code quality, and code styles are conformed to, so the application can stay readable and maintainable.

Database

A Database stores the application's data. This project makes use of the relational database MySQL. For this type of database, a schema must be defined that states both how things are stored and how they relate to each other. Every row in the database represents a distinct entity. Columns describe the entity attributes, the properties that the entity should have and their types.

Deployment

Deployment is the way in which software is set up on a server to be able to be used. Most modern software uses cloud technologies to deploy their solutions. This helps improve scalability and reduces up-front costs.

Event

Events for organizations occur in Lead Sigma's services. These include inbound and outbound texts, emails, and phone calls. Form submissions, lead creations, task creations, note creations, etc., are included as well. These events are recorded and designed to be sent to different integrations to update third-party systems.

Event Types

There are multiple types of events for Lead Sigma. The initial design accounted for discrete and aggregate types, but more, such as notification, could be added in the future. Discrete events are any event triggered by a singular change in a Lead Sigma resource, such as a phone call or email. Aggregate events are events that are triggered by a larger job that aggregates statistics and other metrics to be sent out. Distinguishing between event types allows for better control of the events being passed to the integrations.

Fault Tolerance

Fault Tolerance is the ability of a system to continue to function despite a failure of one of its components. When designing the architecture of the system, fault tolerance is kept in mind to improve the user experience when a failure inevitably occurs.

Integration

An integration connects one service to another. These are necessary in the microservices architecture to allow for communication between them. Integrations are also made between internal services and third-party services to share data between them. The need for this project arose because of the large number of Customer Relationship Management systems (CRMs) that Lead Sigma wanted to integrate with. A concrete example of a successful third-party integration would be from a form to be filled on and noticed by Lead Sigma's services which would then send that data to Salesforce for it to be accessible there. Salesforce is one of several CRMs that Lead Sigma wants to integrate with to provide a better service to its customers.

Integration service

This is the name we gave the microservice that this process analysis statement outlines the creation of.

Microservices

Microservices are an architectural decision to split up an application into independent services. This decreases the size of each service as they will only have one responsibility. These services will then interact with each other to provide full functionality to and end user. As an example, Twitter might have one service

that handles monetary transactions for their ads, and another service that handles account information and user preferences.

OAuth

OAuth 2.0 is the standard in which applications and websites give access to their underlying resources via a third-party application. It is insecure for a user to share their password for one application with a separate application. OAuth solves this problem with authorization codes, access tokens, scopes, and refresh tokens, to make sure the outside system is only given access to the resources the user approved (Recordon, 2012).

Organizations

Organizations are companies that buy Lead Sigma's services. Organizations can have multiple users associated with them. This allows for access to different users' resources to be given to other users if they belong to the same organization. It also allows for company-wide policies to be enforced, such as which integrations to enable.

Sales Activity Service

This was a microservice my team and I created in the Fall. It controls the customers' lead, tasks, and event data. It is the primary data holder in Lead Sigma's architecture, and powers most of the user interface. This is the service that will be saving and triggering most of the events.

Scalability

Scalability describes how easy a service or platform can grow. It also describes the ability of the codebase to evolve to handle increasing responsibilities and features. When code is written too quickly, or without much thought, it can become difficult to add anything new to it without rewriting much of what already exists. Thus, it is important to spend ample time planning, as well as time refactoring to keep the code clean and organized. It is important for companies to have scalable services, so as their customer base hopefully grows, their services can grow with them.

SDK

A Software Development Kit (SDK) provides tools to help work with a service or a platform. This project makes heavy use of AWS's SDKs to manipulate cloud services. These SDKs put a layer of code over AWS's API allowing for even easier access, and faster development. Instead of figuring out the URLs, endpoints, and request body format, the SDK allows you to call a method on an object that will handle the API request for you.

Bibliography

Amazon Cognito. (2021). *Amazon Cognito*. Retrieved March 2021, from Amazon Web Services Web Site: https://aws.amazon.com/cognito/

Amazon Simple Notification Service. (2021). *AWS SNS Developer Guide*. Retrieved March 2021, from AWS Documentation: https://docs.aws.amazon.com/sns/latest/dg/sns-create-subscribe-endpoint-to-topic.html

Amazon Web Services. (2021). *AWS Ruby SDK*. Retrieved March 2021, from AWS Documentation: https://docs.aws.amazon.com/sdk-for-ruby/v3/api/index.html

Amazon Web Services. (2021). *SDKs and Programming Toolkits*. Retrieved March 2021, from AWS Web Site: https://aws.amazon.com/tools/

Atlassian. (2021). *Jira | Issue Tracking*. Retrieved March 2021, from Atlassian Web Site: https://www.atlassian.com/software/jira

Au-Yeung, J. (2020, March 2). *Best practices for REST API design*. Retrieved March 2021, from The Overflow Blog: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/

Batsov, B. (2020). *Rubocop*. Retrieved March 2021, from Rubocop Web site: https://rubocop.org/

Blimke, B. (2021, March 5). *WebMock*. Retrieved March 2021, from WebMock GitHub Repository: https://github.com/bblimke/webmock

Bougie, P. (2014, October 22). *Minittest Issue 204*. Retrieved March 2021, from Pundit GitHub Respository: https://github.com/varvet/pundit/issues/204#issuecomment-60166450

Chin, A. (2014, May 8). *Alex's Rails Cheat Seet*. Retrieved March 2021, from Ruby Rails Naming Conventions GitHub Gist: https://gist.github.com/alexpchin/f5d2be2ef3735889d315

Docker. (2021). *Docker Documentation*. Retrieved March 2021, from Docker Docs: https://docs.docker.com/

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems.* Arlignton, Texas, United States of America: Pearson.

GitHub. (2021). *GitHub Documentation*. Retrieved March 2021, from GitHub Docs Web site: https://docs.github.com/en

LeadSigma. (2021, 03 15). *Home Page*. Retrieved from Lead Sigma: https://leadsigma.com/

Mario, B. (2021). *mysql2*. Retrieved March 2021, from GitHub: https://github.com/brianmario/mysql2

Martin, B. (2008). *Clean Code: A Handbook of Agile Software.* Upper Saddle River, New Jersey: Pearson.

Mozilla. (2021). *HTTP response status codes*. Retrieved March 2021, from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Oracle. (2021). *MySQL Documentation*. Retrieved March 2021, from MySQL Dev Web site: https://dev.mysql.com/doc/

Recordon, D. (2012, October). *The OAuth 2.0 Authorization Framework*. Retrieved March 2021, from Internet Engineering Task Force Web site: https://tools.ietf.org/html/rfc6749

Ruby on Rails. (2020, December). *Ruby on Rails Guides*. Retrieved March 2021, from Rails Guides: https://guides.rubyonrails.org/

SimpleCov. (2021, February 5). *SimpleCov Repository*. Retrieved March 2021, from GitHub Web site: https://github.com/simplecov-ruby/simplecov

Varvet. (2020, October 27). *Varvet Pundit*. Retrieved March 2021, from Pundit GitHub Repository: https://github.com/varvet/pundit

Digital Supplements

The following repositories will require permission from Lead Sigma to access via a GitHub account.

Integration Service Repository:

https://github.com/LeadSigma/integration_service

Documentation Repository:

https://github.com/Ziems/lead-sigma