Y3887821

2.1.1

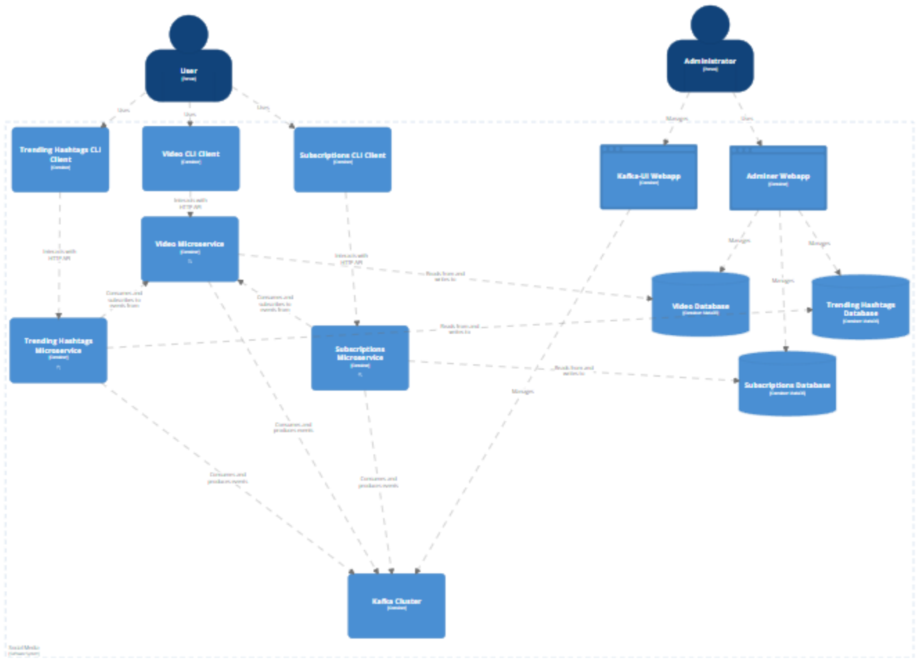


Figure 1 C4 Container Diagram

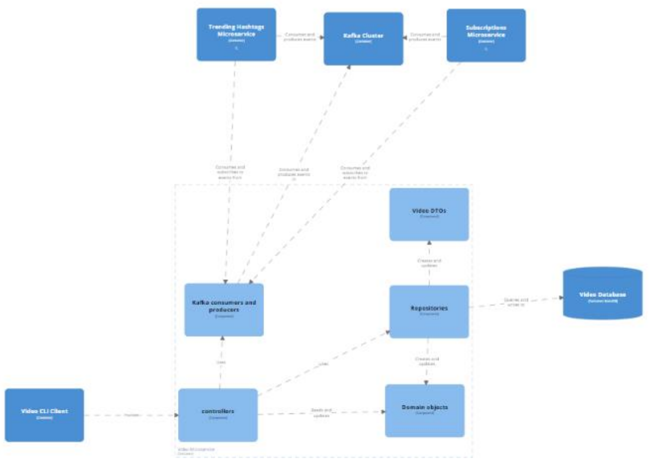
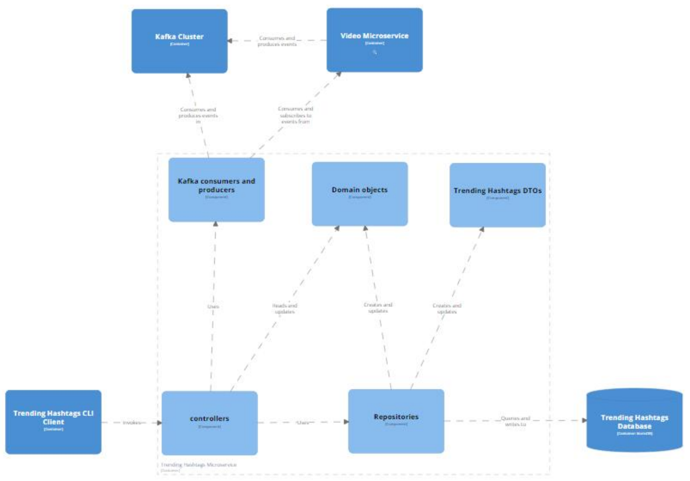


Figure 3 Video C4 Component Diagram



Kafka Cluster
Consumes and produces events
Video Microservice
Consumes and subscribes to events from
Kafka consumers and producers
Domain objects
Trending Hashtags DTOs
Uses
Reads and updates
Creates and updates
Trending Hashtags CLI Client
Invokes
controllers
Repositories
Queries and writes to
Trending Hashtags Database
Trending Hashtags Microservice

Figure 2 Trending Hashtags C4 Component Diagram

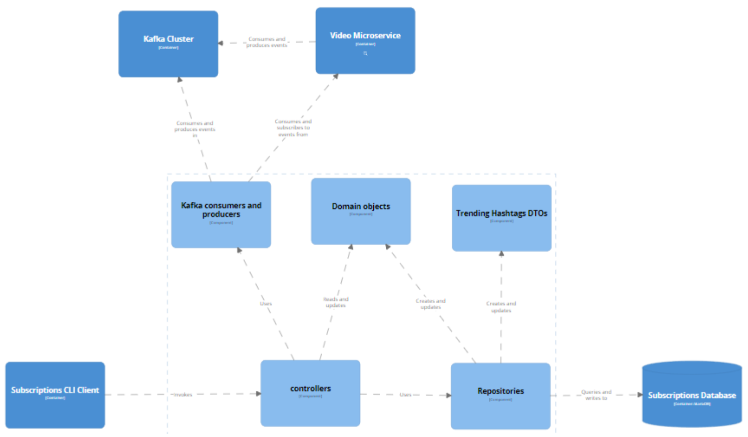


Figure 4 Subscriptions C4 Component Diagram

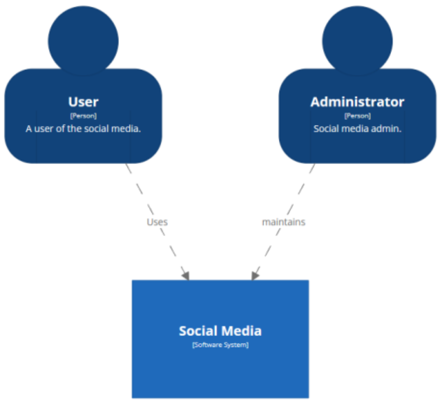


Figure 5 C4 Context Diagram

The architecture defined in the diagrams above can be scaled horizontally to account for an increasing user demand. One method we can use to horizontally scale this implementation without changing the architecture is through replication. By deploying multiple instances of each microservice and using load balancers to distribute incoming requests across these instances you can allow for an increased amount of user traffic. The load balancers ensure that no singular instance of each microservice is overwhelmed by too much traffic coming in at once and instead utilises the resources of each microservice optimally. Load balancing can also be useful to reroute traffic to another instance if one of the instances fail which becomes more likely with a higher number of users. This reduces downtime and improves the user experience as they can continue to use the service without being aware of any issues that may be occurring in the background. They can also be used to reduce load times as a user can be loaded onto the fastest instance for them. There are multiple strategies for load balancing but the two most applicable for this implementation would be round robin or least connections. Round robin as the name suggests forwards requests to the next instance in a rotation. And least connections will forward the requests to the instance that currently has the least amount of user traffic. Overall, by scaling the implementation in this way we can provide for an increasing number of users without having to change the architecture at all. Another way this architecture can be scaled horizontally is by using sharding to divide the databases into smaller more manageable databases as the number of users increases. This will improve the read write performance to the database for a large number of users and allows for the database to be efficiently distributed across multiple servers if necessary. This can be achieved without having to massively modify the architecture. We can also horizontally scale the kafka implementation by adding more brokers to distribute the data of the kafka topic partitions.

Due to each microservice being hosted within its own container adding a new recommendation microservice would require creating a new microservice and creating consumers allowing it to subscribe to the relevant events within the existing microservices. The current events produced by each microservice could also be extended to send messages containing relevant information for the recommendation service such as the users most recently viewed hashtags and what the user has liked or subscribed to. These events can be updated without causing much disruption to the overall service due to the architecture.

2.1.2

The data intensive system includes three microservices:

a video microservice containing a video database and a user database that allows the user to upload videos, get a list of videos including a certain hashtag, watch videos and like and dislike videos. You can do these things through the CLI using the following commands: add-video, get-videos, view-video, like-video and dislike-video.

The add-video command takes the title of the video, a string of hashtags where each hashtag is separated by a comma and the username of the uploader of the video. It then adds the video to the videos database and publishes a video uploaded event which passes the title of the video and the hashtags to be consumed by the other microservices.

The view-video command takes a username and the title of the video they want to view as parameters. It then checks if the video exists returning not found if it doesn't. It then checks if the user is in the user database and if they are not, it adds them. It then adds a view to the video and adds the username into an arraylist of viewers for the video. Once it has updated the video element in the database it publishes a video viewed event that passes the title of the video and the user that viewed the video.

The like-video and dislike-video commands work similarly where they take the name of the video being liked/disliked as a parameter and adds 1 to the total number of likes and dislikes for the video with that title. It then publishes the event liked/disliked video that passes the video id and the hashtags for the video.

A trending hashtags microservice that allows the user to find out which hashtags are currently trending based on how many likes they have received within the last hour. This is done by implementing a rolling window using kafka streams. You can use the CLI get a list of all the current hashtags as well as the current top 10 using the following commands: get-hashtags and get-top-hashtags. The hashtags database is updated based on information consumed from the add video event. The microservice is subscribed to the add video topic and the like/dislike topics and updates the hashtag database according to the information received from these topics such as when a video with a new hashtag is uploaded the hashtag database is updated to include the new hashtag.

The get-hashtags command gets all the hashtags currently in the hashtags database and the get top 10 hashtags command gets the 10 hashtags that have received the most likes in the last hour.

And finally, a subscription microservice containing a video and subscriptions database that allows the user to subscribe to hashtags and get a recommendation of what videos to watch next based on a given hashtag. The CLI commands for these functions are get-next-video, subscribe and unsubscribe. The subscription microservice is subscribed to the add video and view video events which it uses to get information to update its own video database.

The get-next-video command takes a username and hashtag and will check for videos that have a given hashtag and if the user has not watched the video will add it to the recommendations list. It uses information passed from both the add video event and the view video event from the video microservice to get the videos title hashtags and viewers.

The subscribe/unsubscribe commands take a username and hashtag as a parameter and if the user subscribes to a hashtag that hashtag will be added to a list of subscriptions for that user and if they unsubscribe that hashtag is removed from the list. It will then publish an event when the user subscribes/unsubscribes.

The video microservice produces to four kafka topics “video-uploaded”, “video-liked”, “video-disliked” & “video-watched”. It sends different information about the video within each topic and the other microservices use the information produced in these events to add hashtags videos and users to their own databases without having to access the video microservice database. For example, when a video is uploaded the video-uploaded event is triggered from the video microservice which the trending hashtags microservice and subscription microservice subscribes to. They take the information from the event (the video title and its hashtags) and add that information to their own databases to be used later within their own functions.

2.1.3

As mentioned earlier in the report we can horizontally scale the solution in multiple ways to account for an increasing number of users. We can deploy multiple instances of each microservice to distribute the user load across instances. We can then use load balancers to aid in distributing the users across these instances using a round robin strategy to ensure an even distribution of users across each instance. We can also use the load balancers to reroute traffic should one of the instances fail. We can also increase the number of partitions for each topic or add more kafka brokers to the cluster to handle a higher volume of events as the number of users increases. We can also split the databases into smaller more manageable chunks using sharding which will improve the read write performance as the user load increases.

We are using docker compose to deploy the containers which can help automate processes such as scaling and recovering the containers. We can use things like docker health checks to determine if the container is in a healthy condition. Doing this will allow us to catch if a container begins to deteriorate so we can act appropriately by either fixing the issue or replacing the failing instance.

To run the docker containers you should cd into each microservice directory and run ./gradlew dockerbuild this should produce a docker image for each of the microservices. Then you should run the following command

```
docker compose up kafka-0 kafka-1 kafka-2
```

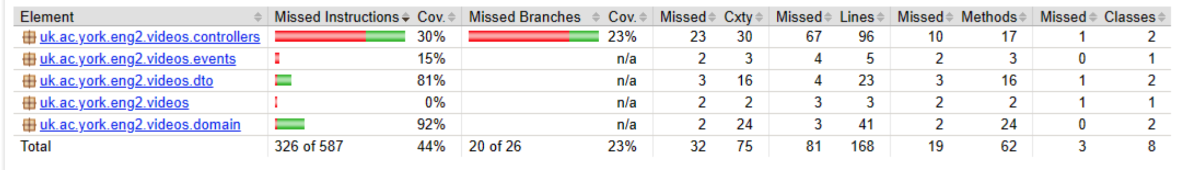
to compose all the kafka docker containers. You should then run the commands within the create-kafka-topics txt file in bash. This will create all the kafka topics needed for the microservices. Once you have done that cd into the microservices directory and run

```
docker compose -f compose-prod.yml up -d
```

now all of your microservices should be running and you will be able to use the command line clients for each microservice to interact with the social media system.

2.1.4

To test the microservices I started by writing video controller unit tests to test methods within the controller. The tests included uploading a video where I set each parameter needed for the video and mimicked how the code from within the CLI command added the video to test that everything is working correctly, getting a video where I try to get a video based on its title, trying to get a video that doesn’t exist to ensure it would send the correct response and deleting a video to ensure the video is removed from the database correctly. I also intend to add tests for viewing a video to ensure the behaviour is correct including if the view is added correctly and the viewers are updated correctly and liking/disliking a video. The tests all appear to pass except for the addVideo test which times out but they cause an issue within the database where SQL can no longer correctly extract the result set. It is likely that some of the parameters within the tests are not being set correctly but I am unable to identify where that may be. It is likely that the tests aren’t actually passing but are merely appearing to pass due to being skipped from earlier failures.

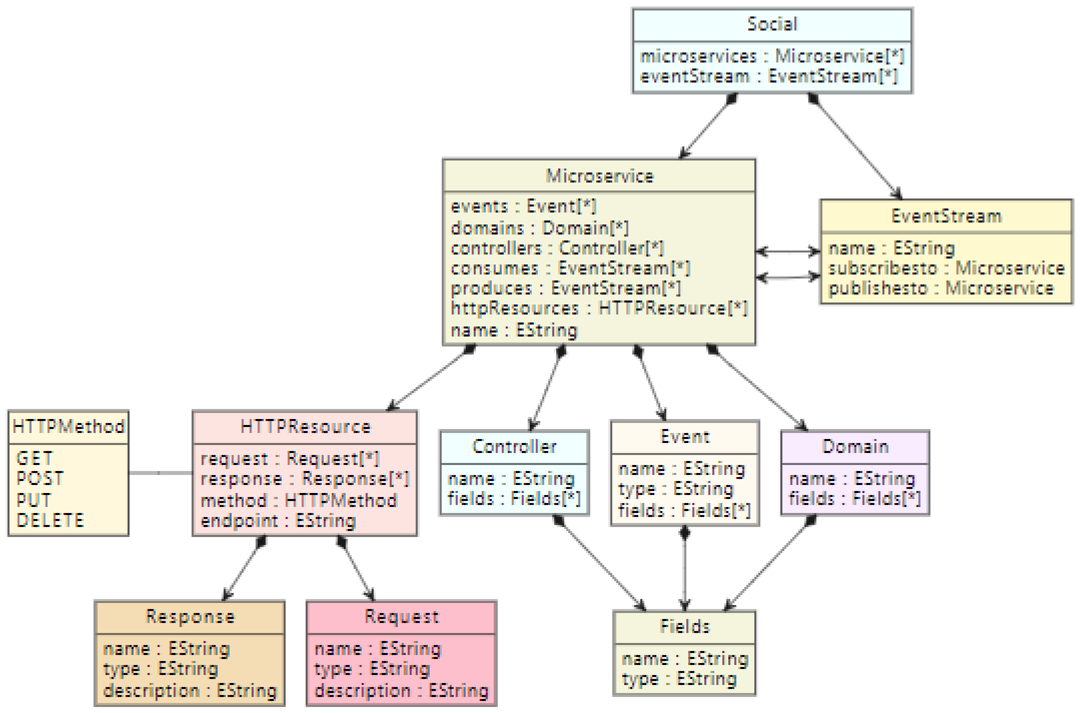


Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
uk.ac.york.eng2.videos.controllers	🟥🟩	30%	🟥🟩	23%	23	30	67	96	10	17	1	2
uk.ac.york.eng2.videos.events	🟥	15%		n/a	2	3	4	5	2	3	0	1
uk.ac.york.eng2.videos.dto	🟥🟩	81%		n/a	3	16	4	23	3	16	1	2
uk.ac.york.eng2.videos	🟥	0%		n/a	2	2	3	3	2	2	1	1
uk.ac.york.eng2.videos.domain	🟥🟩	92%		n/a	2	24	3	41	2	24	0	2
Total	326 of 587	44%	20 of 26	23%	32	75	81	168	19	62	3	8

Each microservice also has a basic test to check that it is running correctly.

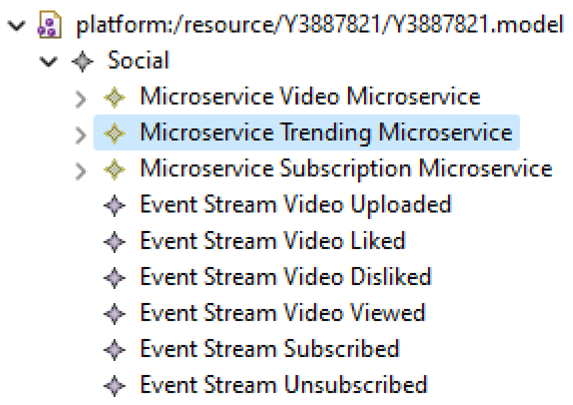
I inspected all of the docker images for any security vulnerabilities and found that there were a high number of vulnerabilities in each of the images with the mariadb image containing the most with 52 which included 2 critical vulnerabilities. After some analysis it was clear that most of the vulnerabilities could be solved by simply updating the packages used by the docker containers. Some of the packages used are unable to be updated however as the packages are either no longer maintained or a fix has not yet been released so these vulnerabilities were unable to be resolved.

2.2.1



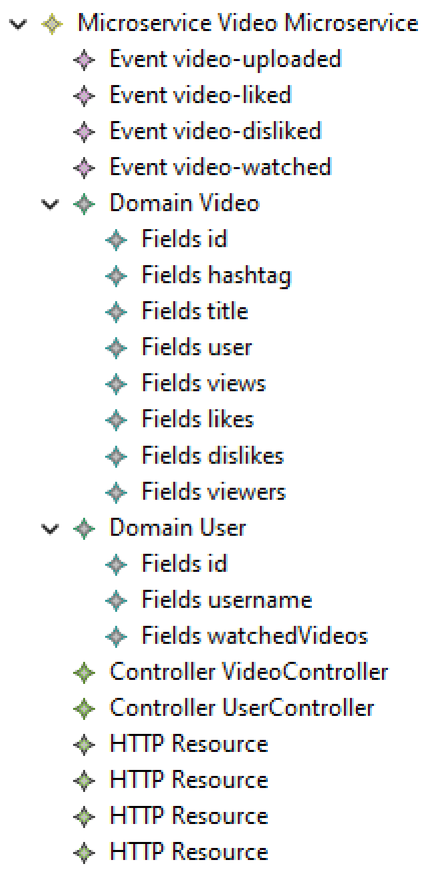
The metamodel has an overall social media class that contains the microservices and the event streams (kafka topics). The microservice class has a name, events (consumers/producers), which event stream it either consumes or produces to and the http resources it uses. I then defined a HTTP resource class which determines whether it's responding to or requesting something as well as the various HTTP methods it can use. I also included the domains and controllers from the microservices with a separate class in the emf file for their fields. The metamodel defined here is an abstraction of what the whole system should be, so I do not include a class for the CLI as it is assumed here that the CLI is part of the microservice.

The model looks a bit like this:



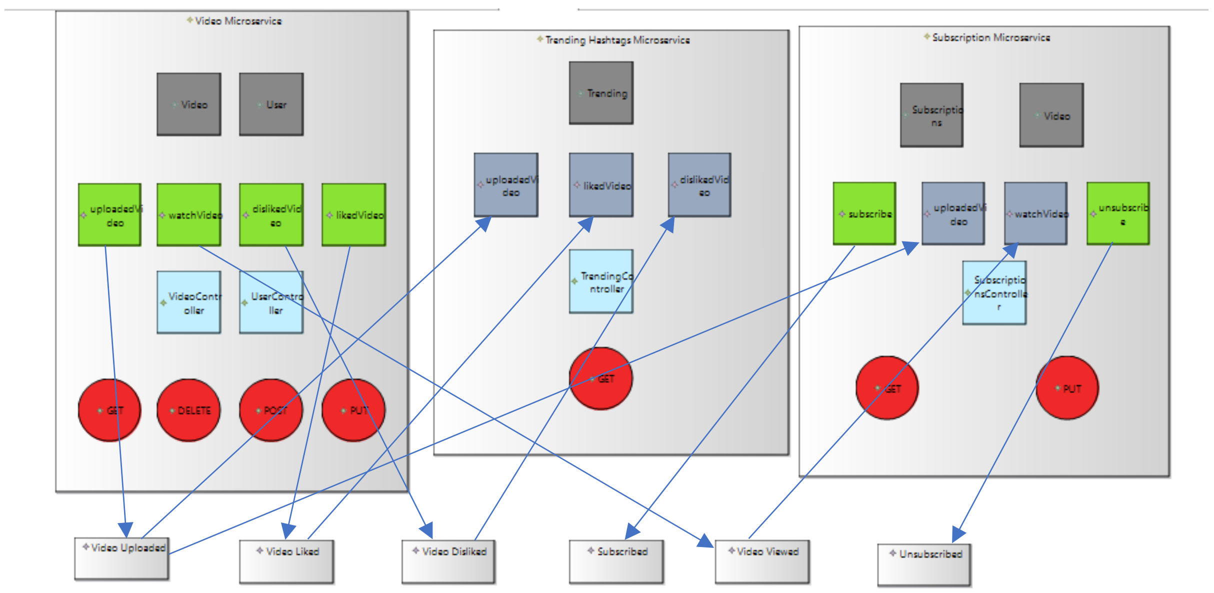
platform:/resource/Y3887821/Y3887821.model
- Social
  - Microservice Video Microservice
  - Microservice Trending Microservice
  - Microservice Subscription Microservice
  - Event Stream Video Uploaded
  - Event Stream Video Liked
  - Event Stream Video Disliked
  - Event Stream Video Viewed
  - Event Stream Subscribed
  - Event Stream Unsubscribed

Where each microservice is defined using the microservice class and the event streams defined using the event stream class. Then within each microservice there will be the various events, domains and controllers e.g for the video microservice:



```
Microservice Video Microservice
    Event video-uploaded
    Event video-liked
    Event video-disliked
    Event video-watched
    Domain Video
        Fields id
        Fields hashtag
        Fields title
        Fields user
        Fields views
        Fields likes
        Fields dislikes
        Fields viewers
    Domain User
        Fields id
        Fields username
        Fields watchedVideos
    Controller VideoController
    Controller UserController
    HTTP Resource
    HTTP Resource
    HTTP Resource
    HTTP Resource
```

2.2.2



The model here shows each microservice as well as the events it produces or consumes. The producers are in green and consumers in blue. It also shows all the Kafka topics as well as points to which event they are either consuming or producing to (this was implemented manually for demonstration purposes as I was unable to implement the transitions within Sirius). The HTTP methods each microservice uses are represented by red dots. And the controllers and domains are displayed using light blue and dark grey respectively. The graphical syntax is straightforward to understand due to the abstractness of it compared to the full system. However, it fails to show a lot of the inner workings of the microservices and is almost too abstract. It is also not entirely clear what each colour means when looking at the diagram. It would be useful to implement some kind of label for either the producers or the consumers so it’s clearer which is which from just looking at the diagram.

2.2.3

The first constraint I implemented checks that there is at least one microservice within the social media system. This is because the system needs microservices to function and without them there nothing will work.

The second constraint is that I checked that each microservice name has at least one character.

The third constraint was to check that each event stream had at least one publisher/subscriber to ensure that each event stream was being utilised correctly and had a purpose for being in the system.

The final constraint was to ensure that each microservice had at least one http resource to ensure that the microservices were performing correctly.

2.2.4

I have attempted to use model to text transformations to generate a very basic scaffold of the microservices to assist with developing them. It is organised so that there is an EGL file to generate the domains and controllers for the microservices. It takes basic things like fields within each class (such as a string for the title or hashtag) to save time over manually coding it but a limit is that it does not generate complex methods such as the subscribe method within the subscription controller. It generates each class into a similar directory structure to the handwritten microservice code with a folder for things like the domains and controllers. It also does not differentiate between microservices, so it currently generates every field and domain into the file rather than just the specific domains needed. The files are generated into the current directory allowing you to copy them into the necessary directories for your microservices where you can then edit them to fit your needs.