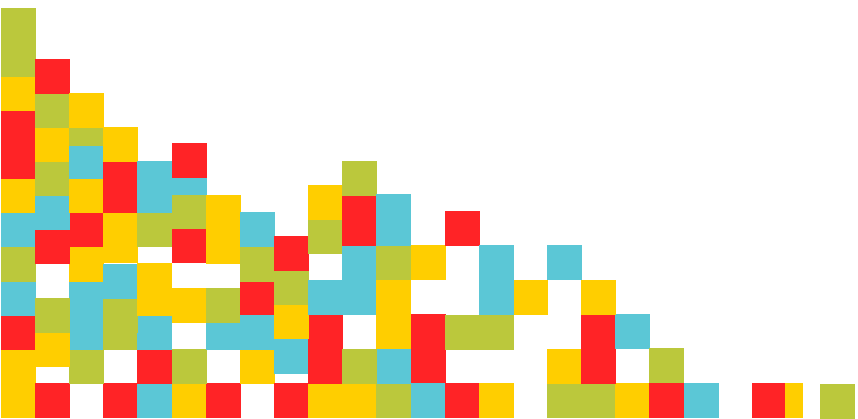
How AppNeta Moved to Microservices



AppNeta®
Performance in Production

Building a single app is hard enough. Building out multiple capabilities into that app is even harder. As part of our ongoing experience to build a full-stack monitoring tool, we recently refactored our logins out of the individual modules we've built and into a Microservice, built in Java. This article will cover AppNeta's decisions in the following:

- Building the business case to migrate to microservices
- Picking a default stack for all future microservices
- Tooling to help slice apart the monolithic application

What is a Microservice?

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery."

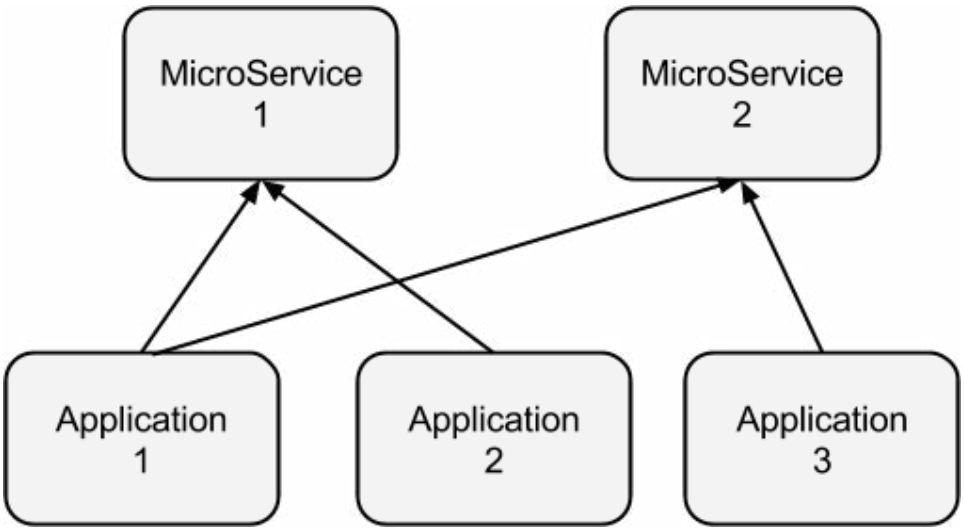
- Martin Fowler

The business case for us was the AppNeta Identity Manager (or AIM in short). The goal of AIM is to provide a shared global organizational information for all our products: PathView, TraceView, FlowView, and AppView.

Inspired by the recent series of "An Opinionated Guide to Modern Java", we thought it would be useful to share how we built our Microservice and the logic behind our tool choices.

Architecture of Microservice

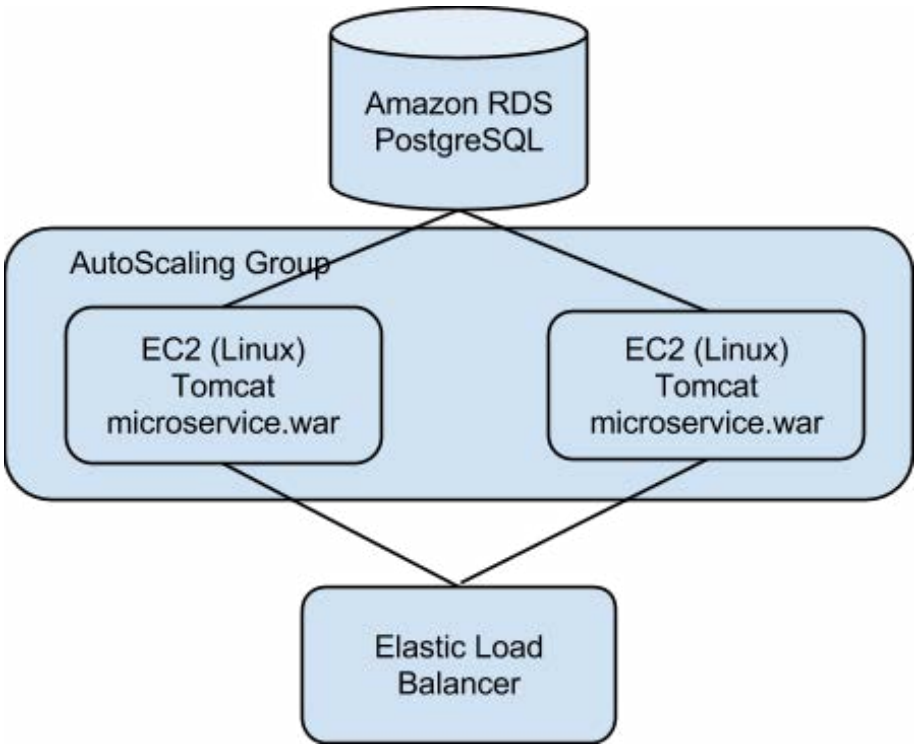
For companies that have multiple products in their portfolio, Microservices usually sit behind the products in terms of the infrastructure. For instance, if we have 2 microservices and 3 applications, some applications may consume both microservices, and others may only consume one or the other.



Technology Stack

One of the advantages of architecting your application in this style is that Microservices aren't tied to a particular technology stack. This gave us the flexibility to choose technologies instead of defaulting to a technology that may or may not make sense. When we had our first meeting to discuss what technology stack we should use, we came up with a short list of preferred stacks: NodeJS/JS, Ruby/Sinatra, Python/Flask, Java/JAX-RS. Out of the four, we decided to use Java/JAX-RS due to our experience managing Java based applications (JVM has excellent tools around it) and the excellent tracing capabilities of our TraceView stack.

We also decided to stick with the same persistence technology as the rest of the applications: PostgreSQL on Amazon RDS (PostgreSQL). Here's a simplified version of our architecture.



The entry point to our microservice is the Amazon Elastic Load Balancer that can distribute the load to any of the EC2 instances within the Auto Scaling Group. Each microservice has it's own autoscale group, where Amazon is responsible for adding or removing EC2 instances depending on the load.

Libraries and Frameworks

Java has one of the richest ecosystems among other programming languages/platforms out there and when it comes to tools, libraries, and frameworks, there are endless options. We decided to use select few of the Spring libraries (and not the whole framework) to augment JAX-RS. The Spring libraries that we use are as follows: Spring Security, Spring Core, Spring-JAX-RS integration, and Spring-Data to augment our JPA/Hibernate ORM.

We made a couple of specific choices that made building this service much easier.

1. Spring-Data

By default, JPA's EntityManager can only handle simple row retrieval by object "id". Any other queries require writing JPQL.

Let's assume we have a User table schema as follows:

id	username	email	active
1	appneta	AppNeta@gmail.com	true

To fetch the user based on case insensitive email address, one must write the following JPQL query in JPA:

```
1  String queryInJpql = "SELECT u FROM Users u WHERE
2  lower(u.email) = ?1";
3  List<User> users = em.createQuery(queryInJpql)
       .setParameter(1, email).getResultList();
```

Spring-Data, on the other hand, includes accessor methods for these types of simple operations:

```
1  // Note that this method exist in a Java interface without
2  any implementation provided
3  // Spring will inject the actual implementation via
   dependency-injection
   Collection<User> findByEmailIgnoringCase(String emailAddress);
```

If the query is transaction or includes an update, there are further annotations to signal those properties to Spring-Data.

```
1  @Modifying
2  @Transactional
3  @Query("UPDATE User u SET u.active = true WHERE u.email like
4  ?1")
   void activateUserBasedOnEmail(String email);
```

Spring Data still needs a bit of JPQL help, but there is not as much Java code. All that is involved is annotation, method declaration (interface, no implementation), and a little bit of JQPL.

2. JAX-RS

JAX-RS is a well-designed RESTful library that has been part of the JavaEE standard since version 6. JAX-RS has gotten better with every release since its inception.

Let's say we have a simple endpoint for User that should support CRUD (Create-Retrieve-Update-Delete). Defining the relevant endpoints is straightforward and explicit:

```
01   @Path("/user")
02   public class UserResource{
03   @POST
04   @Produces(APPLICATION_JSON)
05   @Consumes(APPLICATION_JSON)
06   public User create(User newUser){ … }
07
08   @GET
09   @Path("/{id})
10   @Produces(APPLICATION_JSON)
11   public User retrieve(@PathParam("id") int id){
12       // will throw javax.persistence.NoResultException
13       // but we have implemented an exception mapper
14       // that maps NoResultException => 404
15       User user = userRepository.find(id);
16       return user;
17   }
18
19   @PUT
20   @Produces(APPLICATION_JSON)
21   @Consumes(APPLICATION_JSON)
22   public User update(User user){ … }
23
24   @DELETE
25   @Path("/{id})
26   public Response delete(@PathParam("id") int id){ … }
27   }
```

To create a user, perform an HTTP POST (the @POST annotation) to the following relative URL: "/user" and send the User object in JSON format (denotes by @Consumes annotation) as the Request body. If the operation is successful, the caller will receive the newly updated resource in JSON format as well (the @Produces annotation).

The best part about JAX-RS is that you can specify multiple request/response format without affecting your business logic code (assuming your object can be serialize/deserialize):

```
01   @Produces({APPLICATION_JSON, APPLICATION_XML})
02   @Consumes({APPLICATION_JSON, APPLICATION_XML})
```

This allows you to consume multiple message formats; in this case, both XML and JSON.

Let's compare JAX-RS resource class with Ruby on Rails ActionController:

```
01  class UsersController < ApplicationController
02    def index
        @users = User.all
        respond_to do |format|
          format.html # index.html.erb
03        format.xml  { render xml: @users}
04        format.json { render json: @users}
05    end
06   end
07  end
08
09
10
```

In Rails, you'd have to write the rendering code in each of the controller methods and it is part of your method logic.

Here's another example in Sinatra:

```
01  @Provider
02  public class ResourceNotFoundMapper implements
03  ExceptionMapper<NoResultException> {
04    @Override
05    public Response toResponse(NoResultException e) {
06      return Response.status(Status.NOT_FOUND)
07          .entity(new
08  ErrorResponse(Status.NOT_FOUND.getStatusCode(), e))
09          .type(APPLICATION_JSON).build();
      }
    }
```

Once we mapped NoResultException to 404, anytime JAX-RS resource class encountered that particular exception before sending the response, it will translate it to a proper WebApplicationException with status code 404. This makes your Java logic code straightforward without specific logic that ties to the REST/HTTP paradigm.

In short, JAX-RS makes dealing with REST extremely easy!

3. Database Migration

Because our new service has an entirely separate codebase and deployment path, we needed to change the way we handled our data model. We would need to change the schema over time, but we would also have to initially migrate data into the new service from the monolithic application. To do this, we chose the DB migration library Flyway.

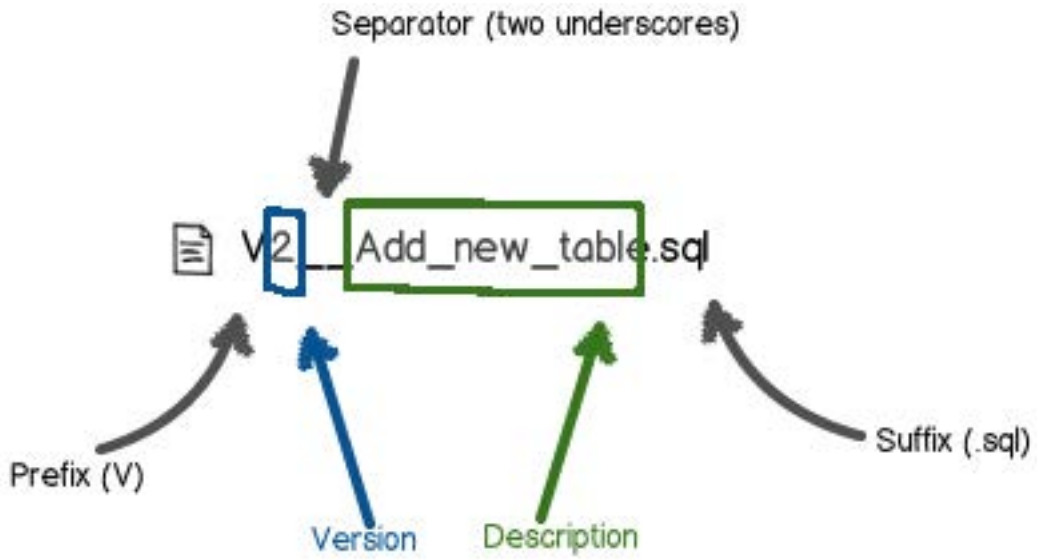
Flyway is an interesting database migration library for 2 reasons:

- It does not perform rollback (by design, as there are a number of operations that databases cannot roll back in any situation)
- It allows you to write the migration using either Java or SQL

Flyway can be executed in two ways: outside or as a part of the software launch. Outside the running application means that someone or a script will be executed before the software is launched.

An example of Flyway being executed while the application starts up is when you deploy a web application to Tomcat; during the start-up of the service/web application, Flyway will run first. If the migration does not work, the deployment can be marked as failure hence the service/web-application will not be deployed/work.

Flyway relies on convention over configuration for the migration script/code.



(The above image is taken from the Flyway website)

Java-based migration example:

```
01  /**
02   * Example of a Spring Jdbc migration.
03   */
04  public class V1_2__Another_user implements SpringJdbcMigration
05  {
06      public void migrate(JdbcTemplate jdbcTemplate) throws
07  Exception {
08          jdbcTemplate.execute("
    INSERT INTO user (id, username, email, active)
    VALUES (1, 'appneta', 'AppNeta@appneta.com', true)"
          );
      }
09  }
10
11
```

Conclusion

With AIM launched and in production, we've found that this architectural style is both faster to develop initially and easier to maintain operationally. It allowed us not only to refactor much of our common authentication code into one place, but also to separate the operational concerns of authentication vs. data processing and storage. With these libraries, we'll be looking to build out further Microservices in other parts of the stack.