Introduction of Microservice Architecture

Fiona Shyne

August 26, 2022

Abstract

Microservice Architecture (MSA) is an increasingly popular choice for designing large, complex web applications. A MSA based application is composed of many small and independent services that communicate over frameworks such as REST or GraphQL. This provides many advantages including, robustness, scalability, and rapid development. However, it creates several challenges from the complexity of the application, that has implications in security and testing. This paper provides an overview of the research and tools available for microservices. It first introduces the design tenants and challenges of MSA applications. It then provides an implementation guide for creating an example service for simulating a bank account. Overall this is meant as a entry point into MSA for new developers.

Contents

1 Introduction

Microservice Architecture (MSA) has become increasingly popular for large, complex, and rapidly changing applications [5, 2, 3]. Extending from the Software Oriented Architecture (SOA) approach, MSA consists of small, independent, and loosely coupled services [10]. Each microservice consists of one and only one business capability and has access to its own database. Communication is done through lightweight frameworks like REST or GraphQL, which sends JSON files (or other formats) between services [10]. This contrasts traditional, monolithic architecture, where entire applications could exist within a single entity and communication is entirely internal. MSA has many advantages, but some key challenges to overcome as well.

There are several reasons why microservices are increasingly being chosen over monolithic approaches, when it comes to large applications. Microservices can make applications more robust, through the independence of each service. When one microservice breaks, others can remain unaffected [5]. The small size of each service increases modularity, which can make changes to applications faster to implement [3]. Microservices can be automatically scaled, which can minimize performance bottlenecks. Lastly, Microservices can be made with a polyglot approach where a variety of technology can be used within one application [2].

There are, however, several key challenges when it comes to MSA design that developers should be aware of. The large number of small, independent services greatly increases the overall complexity of the application [10]. This has a variety of consequences in terms of security, monitoring, and testing. The complexity of the system means that greater testing practices must be implemented, especially those that target the integration and interaction between services [6, 14]. Additionally, microservice design increase the attack area of a system, which means more security practices must be implemented [11, 17, 12]. The polyglot approach makes the creation of logs more complex, which effects both security and testing design [8].

While MSA is widely implemented, 75% of MSA developers received no professional training in microservices [14]. This paper is meant to introduce new developers to MSA, along with identifying key research areas in MSA and tools available for MSA applications. It was written to be friendly to beginners, with no expertise of MSA or software engineering. That being said, some basic knowledge of computer science fundamentals is expected. Section 2 discusses the design tenants of MSA design, in comparison to other architecture approaches. Section 3 discuses the major frameworks for developing microservices interfaces. Section 4 addresses the main challenges, along with potential solutions, in MSA design. Specifically topics of testing in section 4.2 and security in section 4.1 will be addressed. Lastly section 5 will discuss the implementation of a microservice in Java using Springboot. This includes the creation of an API in section 5.5, connection to a database in section 5.4, implementation of security with JWT in section 5.6, and testing strategies in section 5.7.

2 Design Tenants

Service-Oriented Architecture (SOA), and eventually Microservice Architecture (MSA), came out of the increasing problems with monolithic design (see figure 1). In a monolithic software, all components are placed into a single program and platform. For small scale applications, with long development cycles, this allows for easy development and deployment. This approach has also performances benefits for small applications. However, for large applications where business needs are rapidly evolving, monolithic software quickly becomes hazardous [2, 3]. Changes to any part of the software can have serious consequences to

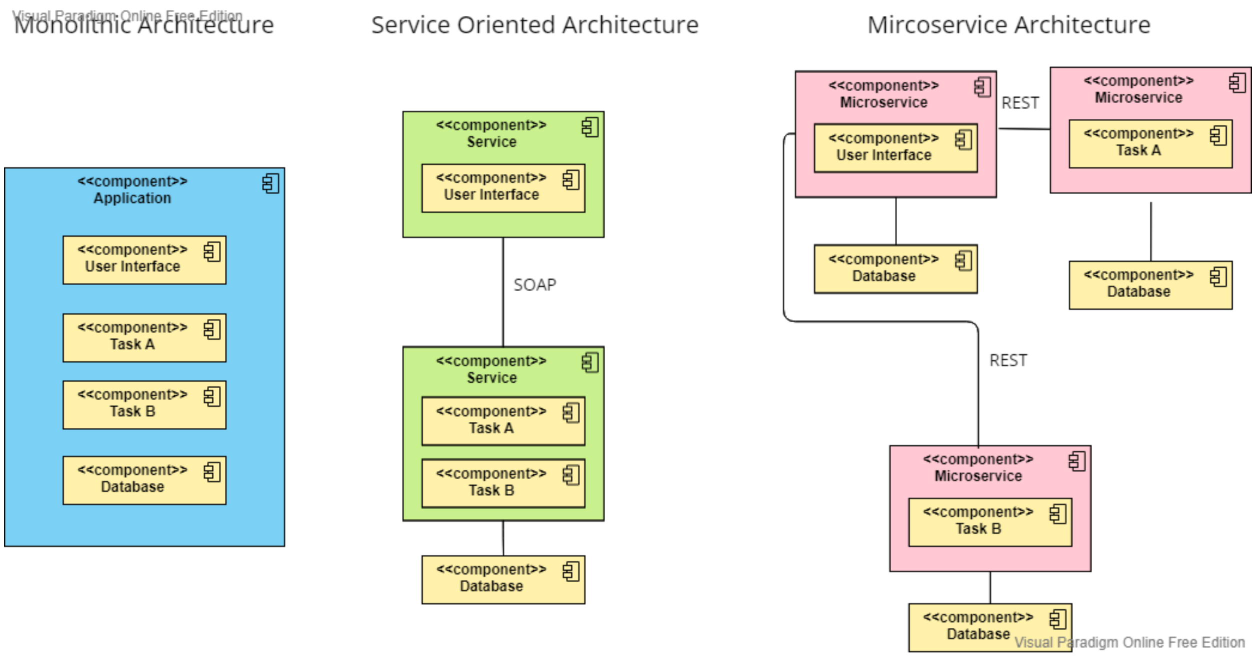

Visual Paradigm Online Free Edition
Monolithic Architecture
Service Oriented Architecture
Mircoservice Architecture
<<component>> Application
<<component>> User Interface
<<component>> Task A
<<component>> Task B
<<component>> Database
<<component>> Service
<<component>> User Interface
SOAP
<<component>> Service
<<component>> Task A
<<component>> Task B
<<component>> Database
<<component>> Microservice
<<component>> User Interface
REST
<<component>> Microservice
<<component>> Task A
<<component>> Database
<<component>> Database
REST
<<component>> Microservice
<<component>> Task B
<<component>> Database
Visual Paradigm Online Free Edition

Figure 1: A comparison of monolithic (left), service oriented (middle), and microservice (right) architecture

the rest of application, meaning any introduction of defects can shut down the entire application. Further, teams working on individual components cannot work completely independently as their modifications can have ramifications for the entire application [3]. There is also effect on the scalability of the application, as changes in capacity and functionality involve large changes in software implementation [5]. Additionally, deployments require the entire system to re-deployed, which often means that service shut downs for lengthy periods of time are required [3]. When business demands are constantly changing, and rapid deployment of new functionality is required, monolithic design becomes a large barrier.

To address these concerns, SOA was developed to break up large monolithic projects. Instead of one program, applications are broken up into services, which are loosely coupled, meaning as few dependencies and interactions exist between services [10]. Services are designed to be interoperable, so that communication between them is done seamlessly without input from an end user. While SOA breaks up applications into services, it doesn't define the scale of each service. This means that, as requirements grow, each service can become monolithic in size.

Microservice Architecture (MSA) initially came about as an extension of SOA, but distinguishes itself in several important ways. The following principles define how a MSA application should be designed.

- **Size**: Principally, as the name would imply, a microservice is the smallest possible unit of service and should consist of only one business capability [5].
- **Light weight communication**: The increased amount of communication necessary to facilitate having many small services, makes the use of light weight communication more important. MSA distinguishes itself from SOA by using light weight frameworks (see 3), which can pass JSON files between services.
- **Independence**: MSA takes after SOA in the way that each service should treated as an independent entity. In this way, calling a service becomes not unlike relying on some third party software.
- **Loose Coupling**: Following from independence, services should be loosely coupled. This means minimizing interdependency (the extent a service relies on other services), coordination (the extent to which services need to collaborate), and communication (the size and frequency of data exchange) between services.

- **Independent Database**: Each service, when possible, should be given it's own database [2]. This reinforces the principle of each service being independent from one another.
- **Containerization and Deployment**: Deployment in MSA is often done with platforms like Docker to create Linux containers. This process allows each service to be neatly packaged and deployed independent of other services [2, 3].

Despite the large amount of overhead required, companies choose to make the transition from Monolithic architecture to MSA for the following reasons:

- **Scalability**: Within traditional applications, scaling often required restructuring of the entire application. Within MSA, this is done on the service level. Bottle necks can easily be identified and modified to accommodate increasing demand. New services are easily added to add new functionality as the application grows. Additionally, auto-scaling practices can be implemented so that resources can be allocated dynamically based on need [5]. This can result in an increase in performance when many users are making requests, compared to monolithic [7] or SOA based applications [10].
- **Robustness**: Given that each service is independent, a disruption to one service should have minimal impact on other parts of an application. By breaking up functionality, applications become less prone to large disruptions [5].
- **Technology Flexibility**: With the breaking down of application into services, each service is no longer tied to one programming language or set of libraries. This means companies can implement polyglot architecture (the use of different programming languages), where technology is chosen on the needs of the team and the functionality desired [2].
- **Deployment**: Each service is deployed independently, which means creation, modification, and deletion of services can be done on a case-by-case basis [3]. A single service can be deployed without having an effect on unrelated functions.
- **Rapid Development**: MSA suits itself well to DevOps and Continuous Integration / Continuous deployment (CI/CD) development styles that emphases short development cycles for incremental improvements Chen [3]. Independent deployment aids this model, by allowing services to be deployed quickly with no down time. If many updates are deployed multiple times a month, any amount of down time can cause large disruption to service. Additionally the small scope of each service, means that teams can quickly create and update services, and unnecessary communication is kept to a minimal.
- **Small Team Size**: A single team can be develop a single service. Since each service is small, teams can consist of only 2-5 members Waseem et al. [14].
- **Reduced Feature Overlap**: The well defined function of each service reduces the potential for unnecessarily re-implementing features. Instead these are all reduced to one service. Bucchiarone et al. [2].
- **Enforced Modularity**: While modularity should be implemented in monolithic applications, the independent nature of microservices makes modularity required. During stressful period of development, it is tempting for developers to take short cuts and break modularity Chen [3], but with MSA this is not possible..

These factors have make MSA appealing to applications with complex, and rapidly changing needs. Transiting to MSA from monolithic applications can be a long and difficult process but has been worthwhile for many companies. Following a transition to MSA, future design modification become much faster, rapid development cycles can be implemented, and systems become more robust against defects.

3 API Frameworks

An Application Programming Interface (API) describes the way an user interacts with a program. Typically this involves a user sending a request to read or modify data which the API has access to. For a MSA application, the API of the service is vital for communicating the ways the service can be interacted with. Usually this involves defining methods of reading or modifying data within the microservice's database. To encourage uniform patterns of how requests should be made, a variety of frameworks can been implemented. This helps the user understand the ways in which a API (or service) can be communicated with. For MSA application, REST and GraphQL are two of the most popular frameworks.

3.1 REST

REpresentational State Transfer (REST) is an architectural style that is defined by several constraints [4].

- **Client-Server:** There are two parties in a REST request, the client and the server. The client makes the request, and the server fulfills it. This representation is used to separate the concerns of a user interface (the client) and of data storage (the server)
- **Uniform Interface:** The methods to interact with a RESTful API should be consistent and easy for clients to access. Individual resources are identified in the request (often within the URL of the request), and the returned response contains enough information for the client to access or modify this resource later.
- **Stateless:** A REST request should contain all of the information required to access a desired resource. Therefore REST APIs do not store session information, and each request is independent of one another.
- **Cacheable:** Each response should signal whether it is able to be cached by the client. If it is cached, the client will store the data to be retrieved faster later. This constraint balances needs for performance (having fast access to resources) vs. reliability (ensuring the information used is consistent with current conditions).
- **Layered System:** A RESTful application is composed of several layers, each the only have access to the layer above and below it. Like the Client-Server constraint this acts to separate the concerns of different entities.

In practice, a REST request is often made by providing a specific URL over HTTP. With this model four main request types, using HTTP verbs, can be created: a GET request reads from a database and returns an entity, a POST request creates a new entity to place in the database, a PUT or PATCH requests modified a resource, and a DELETE request removes a resource. Given that REST is stateless, each request must provide all necessary information for the request to be fulfilled. For example, request parameters are appended to the end of the URL using the question mark (?) symbol (ex: .../getAccountByNumber?accountNumber="1111"). While REST does not define the format of the data returned, a common format is JSON. JSON provides a way to represent objects or data that is programming language agnostic (any language can be used). JSON files map fields and values, and explicitly present the value of every datatype. An example of a JSON file is given below.

```
        {
    "getAccountByNumber": {
      "accountNumber": "1111",
      "value": 55,
      "incomingTransactions": [
        {
          "id": 30,
          "value": 24,
          "fromAccount": "2222",
          "toAccount": "1111"
        },
        {
          "id": 31,
          "value": 24,
          "fromAccount": "2222",
          "toAccount": "1111"
        }
      ],
      "outgoingTransactions": [
        {
          "id": 29,
          "value": 42,
          "fromAccount": "1111",
          "toAccount": "2222"
        },
        {
          "id": 43,
          "value": 100,
          "fromAccount": "1111",
          "toAccount": "2222"
        }
      ]
    }
  }
```

3.2 GraphQL

GraphQL is a query language that was created as an alternative to REST, developed by Facebook. GraphQL allows clients to not only define the request, but the desired fields of the response. For large entities that could contain hundreds of fields, a client can specific only the few fields that it will need for it's purpose. This can cut down on response sizes significantly. It also allows for multiple queries to be made in a single request, which can cut down on the amount of communication needed between a client and a server. In a GraphQL API entities and queries are exposed to the user through a schema, which provide a mapping between fields (or parameters) and data-types. For complicated queries, GraphQL is able to out-preform REST applications both in term of performance [9] and in speed of implementing requests by developers [1].

However, there is a danger that comes with implementing GraphQL infrastructure. This relates to the worse case response size of a schema, which defines how large a response entity can be in relation to a request in the worse case. If a schema includes a self-referencing term, the response can be exponentially larger than the request. For example, if an entity User contains a field called friends, which is a list of Users, a malicious or ignorant client could make query with a massively large return entity. This leaves the application vulnerable to denial of service attacks, where an application is effectively shut down due to the number of resources being requested and sent to a malicious user. The majority of both commercial and open-source GraphQL schema have a exponential worse case response size (meaning that self-referential schemas were used) [16]. A major protection against this threat is limiting the size of the response using slicing parameters in the query, but very few of studied schemas implement this practice universally [16].

4 Challenges

There are many challenges that uniquely manifest themselves in MSA applications [13, 15]. These issues largely arise due to an increase in system complexity. The large number of services, which can be in the thousands for a particularly large system, inevitably leads to an increase in the complexity of the system. Even with loose coupling principles, the number of connections and interactions in a MSA application can quickly become unmanageable. A 50% increase in services can cause an 84% increase in dependencies [10]. This makes the surface that needs to be secured and tested much larger [10].

4.1 Security

In a monolithic application, the backend was tied to the frontend. This meant that internal functions could be reasonably assured that when a request is sent the them, authentication had already occurred Rudrabhatla [12]. It could be confident of this because all calls to the backend were done within the context of the application. Additionally, monolithic applications were often state based, so backend functions could have access to user information through a web session.

However, this is not true for microservices. The independence and loose coupling of each service means that it is likely, if not expected, that requests to a service comes not just from internal processes but from outside sources as well. This means that every service, and every connection between services, is at risk for attack and needs to be secured Rudrabhatla [12]. Therefore, the attack area of a MSA application is much greater than that of a monolithic application and traditional techniques for security need to be adapted. In an analysis of 5 open-source MSA projects, 10.5% of the issues reported to GitHub were related to security concerns [15].

One of the major focuses for security for microservices is authentication and authorization. Yarygina and Bagge [17] conducted a literature review that looked at both academic and gray literature (the majority of which was published in professional communities and company sites) that related to security for microservices. From that review 61% of academic and 68% of gray literature addressed authentication or authorization. Authentication refers to a software's ability to confirm a user has the appropriate credentials to gain access to an application. Authorization refers to the process of validating that an authenticated user can access a particular task or function. Failure in authorization can lead to the risk of privilege escalation. There are two major forms of privilege escalation, vertical and horizontal. Vertical privilege escalation refers to a user having access to functionality reserved for a higher status user, such as an ordinary user having access to admin only functions. Horizontal escalation refers to a user accessing information or functions associated with a different user (where both parties are of equal status), such as a user gaining access to another users account information.

For a microservice system, authentication credentials are necessary for every service that provides non-public functions or resources. This is true even for internal services that, in theory, shouldn't't be accessed by any third parties. Unlike in monolithic applications, internal services in a MSA application always face a risk of being exposed unknown sources. Of 28 security practices, identified through open-source projects, authenticating internal services was one of the most recommended practice from microservice developers [11].

Authentications and authorization is most often done through a combination of API Gateways and tokens [12]. Tokens are a way to contain identity and access information, without the system having to store password information. Token also last for a short period of time, reducing the impact of them being exposed. JSON web tokens (JWT), for token format, and OAuth, for token issuing, have been identified as some of the most appropriate technologies for this purpose [12, 17]. When a user makes a request for a resource, housed on a protected microservice, the frontend will first check for a token. In the case that a token is invalid or does not exist, the user is sent to authentication, which involves them inputting login information or other identification measures. This information is sent to an authentication server via the API gateway. If the authentication server can validate the credentials, it will create a token which will be sent to and stored in the frontend. The frontend will then once again attempt to access the resource, this time by passing along the token it stored. The API gateway filters this request, by validating the token with the authentication server, and if successful the request will be sent to the appropriate microservice. For services that require more security, checkpoints can be added via additional API gateways added after this first filter point (see figure 2). Both academic and gray literature [17], along with microservice developers themselves [11], identify API gateways as important security features.

There are, however, some gaps in security research for microservices. One of these gaps are a lack of research relating to the monitoring and detection of attacks. Few of the papers, 38% of the academic and 9% of the gray literature, found in the literature review related to the detection of threats, rather than prevention of attacks [17]. The independence of microservices create a complicated environment in terms of monitoring and logs. The polyglot (using multiple languages) nature of many MSA systems makes uniform monitoring and logging frameworks difficult [12]. Additionally, the speed of which deployment occurs for microservices, can make the detection of "normal" behavior more challenging [8]. The potential lack of universal monitoring, along with the moving baselines of behavior, can make the detection of threats a struggle for developers. More sophisticated tools for aggregating logs and detecting abnormalities, may become required for MSA based applications. Other gaps in security research including finding methods to recover and react from attacks, security patterns, and focus on secure application as a whole [17].

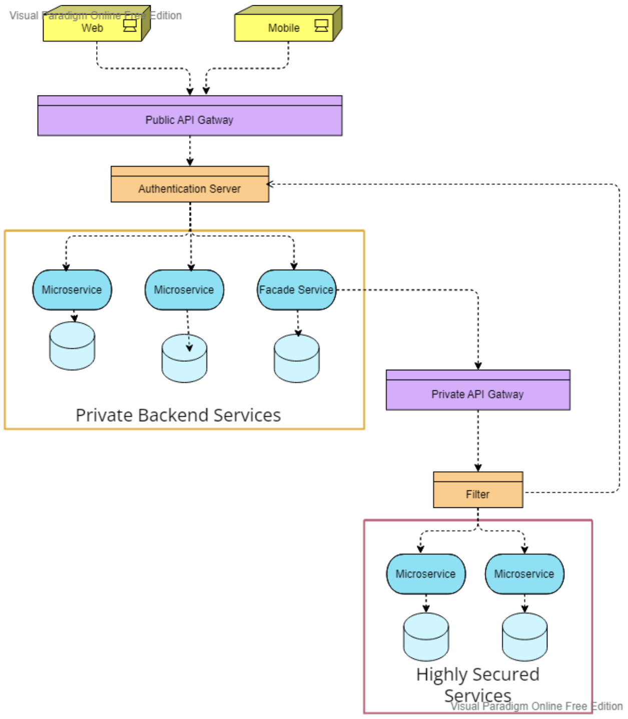


Figure 2: A high level overview of a public and private API gateway for securing microservices.

4.2 Testing

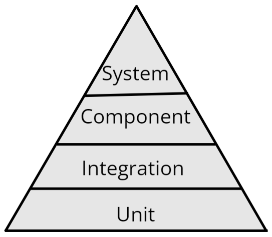


Figure 3: The testing pyramid. The most tests implemented should be the most isolated (unit tests) and the fewest number of tests should be the least isolated (system or E2E testing/

For any application, testing is vital for success deployment of new functionality. That being said, the added complexity of microservice design makes this more challenging in several ways. Similar to the increase of attack area resulting in security risks, this larger area of connections results in a greater amount of tests required. Not only does every microservice need to be tested in isolation, but their interactions with other services and third parties must be tested as well. This means that a developer needs a greater understanding of the system and its interactions. Additionally, a multitude of testing strategies are required to cover all potential failure points.

There are a multitude of testing strategies that can be implemented for MSA applications:

- **Unit testing**: Unit testing seeks to test the smallest possible software component, in isolation from the rest of the application. A unit test can test a component as small as a single method and should not exceed a class. Anything that this component relies on (for example if a components calls on a method in a separate class) should be replaced with a fake version that returns fabricated results.

- **Integration testing (IT)**: IT testing tests the interaction between two or more components. Often this would test how two microservices interact with each other but can also include a microservice interacting with a database or some third party.
- **Component testing**: A component test looks at the microservice as a whole entity. This is a black-box test, meaning that no internal information about how the system works should be used when designing tests. In this sense the test role-plays a end user attempting to use the service. Any services that the microservice depend on, are replaced with doubles so the service can be tested in isolation.
- **Contract Testing**: Contract testing, and its most common variety Consumer Driven Contract (CDC) testing, focuses on the connection between two microservices but ignores the business logic. In CDC testing, the consumer (the service making the request) creates a contract of what response it expects from the provider (the service receiving the request). The provider than confirms it can fulfill this request. This test ensures that what is being provided from a service is the same as what each of its consumers expects, despite changing functionality and requirements.
- **End to End Testing (E2E)**: E2E or system testing examines the system, and all the microservices involved, as a whole unit from user request to completed response. This is another black-box testing technique, where the test acts like a user without any internal information about how the system behaves.
- **A/B Testing**: A/B testing is used to determine which version of a piece of software is better suited for a specific business need. Here, users are randomly provided one of two possible versions of a particular software and metrics (as determine by the testing goal) are recorded and compared.

The scope (how many components a test looks at) determines the cost (time or resources) a test requires to both be developed and ran. Unit tests, being highly isolated, are very simple to develop and run. At the other end, E2E testing require significantly more resources to complete. Therefore, developers should aim to find the majority of the defects from the most isolated tests, to minimise the time spent on more complicated tests. The testing pyramid (see figure 3) describes how the amount of highly isolated tests (unit tests) should outnumber the amount of highly integrated (E2E) tests. Following the testing pyramid can reduce the amount of overhead required for preforming resource intense tests [6].

While these testing strageties can be applied to all software development, some strategies are more utilized with MSA applications than others. Waseem et al. [14] found that certain techniques, in practice, are used more regularly than others by MSA developers. Following the testing pyramid, unit testing is the most frequent testing technique utilized [14]. Additionally, the majority of MSA developers use integration and E2E, often or very often [14]. This is likely due to the complexity of a MSA system that demands more in-depth system wide testing like E2E, and the number of interacting pieces making integration testing more important. Component testing was also used often by the majority of MSA developers [14], emphasizing the importance of validating a service as an independent entity. Interestingly, A/B and CDC were the least used strategies [14]. According to participants, A/B testing is time consuming, not supported by enough libraries, and required too many users [14]. For CDC testing, the communication issues for large teams and use of third parties, make it less appealing [14]. That being said CDC testing presents a unique opportunity to isolate connections, and potentially minimize the number of integration tests required [6]. Further investigation could be done to find way to make CDC testing a more feasible option for developers.

Most of these techniques are not unique to MSA, despite the increased difficulty of testing complex and interconnected microservices. The most common tools used by MSA developers, including JUnit, fall back on traditional techniques [14]. There is currently little inclusion of tools to automatically or systematically test MSA systems. This provides an opportunity to ease the burden to developers responsible for testing increasingly complex systems. Tests designed specifically for MSA applications, such as Chaos Monkey development by Netflix, should be further investigated. Chaos Monkey would randomly turn off services, and measure the outcomes to test the robustness of the system [5]. In the switch from monolithic to microservice based applications, a testing first policy becomes even more important [3]. Future work

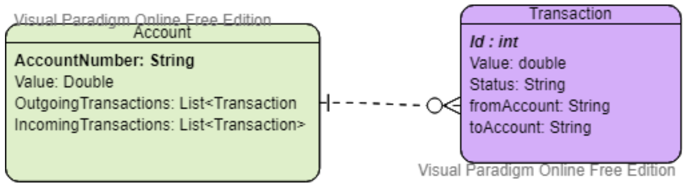

Visual Paradigm Online Free Edition
Account
AccountNumber: String
Value: Double
OutgoingTransactions: List<Transaction
IncomingTransactions: List<Transaction>
Transaction
Id : int
Value: double
Status: String
fromAccount: String
toAccount: String
Visual Paradigm Online Free Edition

Figure 4: The main models used in service

can look into how testing systems can be designed and adapted to better suit the needs of microservice infrastructure.

5 Example Microservice

To demonstrate how these principles are applied in practice, the following section will describe the process for developing a complete microservice using Spring Boot and Java. This service simulates how a system for managing banking accounts and financial transactions, could be done. Section 5.1 discuss the design of this system and the different models used. Section 5.2 explains what Spring Boot is, and the structure of a Spring Boot project. Section 5.5 addresses how to construct API requests both for RESTful API and a GraphQL API. Section 5.4 implements a repository that connects to a SQL database. Section **??** and 5.7 deal with security and testing accordingly.

The complete example code is available on Git Hub

5.1 Design Specifications

r

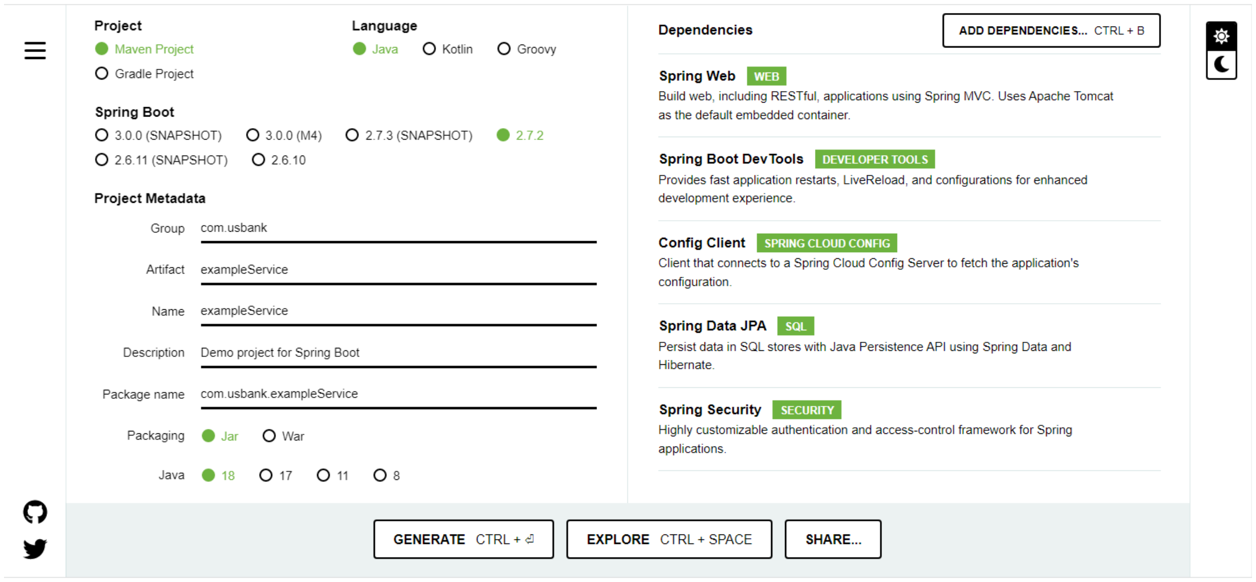
There are two main models (see figure 4) that are used to represent banking account features. The first is an account, that is accessible with an account number. An account contains a value (current funds of an outgoing), and incoming and outgoing transactions. The other model, therefore, is a transaction, that is accessible with an transaction id. An transaction represents a transfer of funds from one account to another. It has a from account (the account sending the funds), a to account (account receiving funds), a value (amount of funds being transferred), and a status (representation of current state of transaction).

The methods provided to the end user are determined by their authority. An admin user may preform the CRUD (create, read, update, delete) operations on all accounts and transactions. A user, on the other hand, only has access to their own account. They can read their account information, along with any transactions from which they are the sender of receiver. They may also create and process (moves money and updates status) transactions of which they are the sending parting. A non-admin user cannot, however create, delete, or update an account. They also cannot delete a transaction.

5.2 Building a Web Application with Spring Boot

Java Spring Boot is a tool for developers that uses the Spring Framework to manage dependencies within a Java application. Spring Boot allows for quick development, by automatically configuring and wiring dependencies within an application. Spring Boot uses the opinionated approach, where it will automatically make dependency decisions for developers. This free up development time in terms of specifications. Within a Spring Boot application, a minimal amount of specifications can lead to a great about of functionality.

Spring Boot provides a online tool, called the Sprint Initialzr, which builds Spring Boot projects with starting dependencies. The dependencies needed for a web application are Spring Web and Client Config. Other dependencies are used in this project, including Spring Security, JPA and GraphQL, but will be discussed in later sections. Spring Web packages many dependencies related to building web applications, including the needed tools to build RESTful applications. By default it runs an Appache Tomcat server on port 8080. Figure 5 show how the Initialzr can be used to set up starting dependencies. However, dependencies can be added at any point in the development process.


Project
[✓ Maven Project] Gradle Project

Language
[✓ Java] Kotlin Groovy

Spring Boot
3.0.0 (SNAPSHOT) 3.0.0 (M4) 2.7.3 (SNAPSHOT) [✓ 2.7.2] 2.6.11 (SNAPSHOT) 2.6.10

Project Metadata

Group	com.usbank
Artifact	exampleService
Name	exampleService
Description	Demo project for Spring Boot
Package name	com.usbank.exampleService
Packaging	[✓ Jar] War
Java	[✓ 18] 17 11 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Config Client SPRING CLOUD CONFIG
Client that connects to a Spring Cloud Config Server to fetch the application's configuration.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.

Figure 5: An example of how the Spring Initialzr can be used to set up starting dependencies

Once project settings are entered, the Generate button will create a project that can be imported into any IDE. Eclipse was used in this example, but the process is similar for other IDEs. There are several important files within a Spring Boot projects, that are created during the generations process.

1. **pom.xml**: The XML file located in the base directory, called pom.xml is where dependency configuration is done. For example, in the pom file you can see the Spring Web dependency listed as:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

   New dependencies are added in this format between the lines `<dependencies>` and `</dependencies>`.

2. **application.properties** The properties file, located in the "src/main/resources" folder, named application.properties define many important application variables. For example the default server port can be changed by adding `server.port=8081` to application.properties.

3. **Application.java** The Java class located in packaged named exampleService with the main Java folder, called ExampleServiceApplication.java, will be what starts the Tomcat server in this project. Later in this project it will also scan other files for annotations such as `@RestController` to deploy the API to the server.

Spring Boot relies on annotations to know how to build and run the application. Spring Boot uses several stereotypes, given as a annotation, which provides information about what kind of operation a class provides.

1. **@Component:** As the most general stereotype, a component is defined as a entity that is a target for auto-detection. That is if a class uses a `@Component` annotation, other classes can use the `@Autowired` annotation to automatically inject a instance of this component into this new class. All other stereotypes are a component, along with their specified function.

2. **@Repository:** A repository is interested in the retrieval, modification, storage of information. In this instance, a repository is responsible for interacting directly with the database storing data.

3. **@Service:** A service contains the business logic of the application. It communicates directly between a repository and a controller. This project does not implement a service component, and therefore the controller interacts directly with the repository. More complicated applications may require a service layer.

4. **@Controller:** A controller is the most public facing layer of a system. It is responsible for sending the user provided information into the appropriate service or repository. It then packages the output it receives into an appropriate format (often as a JSON) to send back to the user.

This project will create an account and transaction repository, which will interact with the account and transaction table of our database. It will then send the information they gather from the database to our controller, which presents it to the end user.

In order to make sure Spring Boot looks at all packages in the project, the `@ComponentScan` and `@EntityScan` annotations must be added in the application file. The following code block scans for Java files that contain the aforementioned annotations across our project, and allows them to be injected throughout our service.

```
@SpringBootApplication
@ComponentScan("com.usbank.*")
@EntityScan("com.usbank.*")
public class ExampleServiceApplication {
        public static void main(String[] args) {
                SpringApplication.run(ExampleServiceApplication.class, args);
        }
}
```

Dependencies can be installed using Maven install. Within Eclipse this is an option available by right clicking the top most directory and selecting run as Maven Install. Once the dependencies are installed, the server can be started by running ExampleService.Java as a Java application (right clicking on ExampleService.Java and selecting run as Java application). Once the server is started, it can be accessed on ”localhost:8080” from any browser. However, nothing will appear initially at this url, as the controllers are not yet implemented.

5.3 Models

The “Models” package contains the Account and Transaction models as specified in the design specifications. Appropriate constructors, getters, and setters are implemented.

5.4 Connecting to a Database

Each microservice should be given it’s own database to pull data from. This section first explains how to mock a database using ArrayList for testing purpose (see section 5.4.1. After that, it explains how to create a mySQL database, which the service can access on a local network (see section 5.4.2. The following sections demonstrate how to connect the database to the Spring Boot application with either JBCDTemplate (see section 5.4.3) or JPA (see section 5.4.4).

5.4.1 Mock Repository

To get started, it can be useful to create a mock repository to test that controllers act as expected. In this mock, accounts and transactions are placed in an ArrayList within the repository itself. To make this class a repository the `@Repository` annotation must be placed above the class. This allows other classes to automatically inject this repository into their system to access it with the `@Autowire` annotation as shown below.

```
@Autowired
MockDataRepository repo;
```

Later in this class, the object `repo` will be automatically instantiated and the methods within the repository can be accessed.

Within the mock repository, we can create several useful methods for interacting with account and transaction entities. For example we can get an account with a given name, as shown below.

```
@Repository
public class MockDataRepository {
    List<Account> accounts;
    public Account getAccount(String account) {
                for(Account acc : accounts){
                        if (acc.accountNumber.equals(account)){
                                return acc;
                        }
                }

                return null;
        }
}
```

5.4.2 Creating a mySQL database

In addition to the Spring Boot project, a external database must be created and connected to. This project uses mySql to provide a database that is available locally on your computer. The first step is to create a connection and a user with access to the connection. This can be done in the initialization process, but can be completed after the fact as well (see mySql documentation). In this project, a root user was created with read and write access and has a username of "root" and a password of "IAmRoot." Additionally, the connection is accessible on port 3306. If this is different on your system, make the appropriate changes to the instructions below.

After establishing a connection, a database must be created. Our database, which is called AccountService, can be created and entered on the mySQL command line with the following lines.

```
CREATE DATABASE AccountService;
USE AccountService;
```

This next sequence will go over how to create tables and basics of SQL syntax. If you are implemented JPA, the creation of tables is done automatically (although it can also modify existing tables). For people only interested in using JPA, this next sequence may be skipped.

The accounts table can be created with the following SQL code in the mySQL command line.

```
CREATE TABLE Accounts {
    account_number varchar(255) NOT NULL
    value double,

    PRIMARY KEY (account_number)
};
```

There a couple things to notice from this SQL statement. One is that `account_number` is named using the snake case (`snake_case`) convention and not with camel case (`camelCase`) as is done in the Java syntax. This is important as tools, such as JPA, will assume that variables are referenced with snake case within the database and will automatically link camel case variables (within Java code) to snake case variables (within the database). The second important thing to note is the PRIMARY KEY command at the bottom. This does a couple of things. First it ensures that account numbers will be unique and secondly it will set account number as the primary means to reference an account. The last thing to note is that incoming and outgoing transactions are not represented here. These relationships are linked logically within our application, and not within the database itself. The `from_account` and `to_account` fields within a transaction are sufficient in recording the relationship between a transaction and an account.

A similar process can be done to create the transactions table.

```
CREATE TABLE Transactions {
    id int NOT NULL AUTO_INCREMENT,
    value double,
    from_account varchar(255) NOT NULL,
    to_account varchar(255) NOT NULL

    PRIMARY KEY (id)
};
```

And important thing to note here, is the `AUTO_INCREMENT` value added after id. This means that an id value does not have to explicitly added, and that mySql will instead automatically assign an transaction with a unique id.

Values can be added to a table with the insert command. For example we can create an account with account number ”1111” and starting value 1,000 with the following command.

```
INSERT INTO Accounts (account_number,  value) values ("1111",1000);
```

We can get values in our table using the select command.

```
SELECT * FROM Accounts;
```

This command provides all the columns (*) for all entities in the accounts table. We can also select entities with specific values, such as getting an account by account number.

```
SELECT * FROM Accounts WHERE account_number = "1111";
```

5.4.3 Using JdbcTemplate

The JdbcTemplate class provides a way we can directly make SQL calls to our database. JDBC can be added by adding the dependency to the pom.xml file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

It also comes with the JPA dependency that is added in the next section. We will also need the SQL dependency to access our database.

```
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
</dependency>
```

We then need to give Spring Boot the information needed to access the database, by adding it to the application proprieties file.

```
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver
spring.datasource.url =
 ↪  jdbc:mysql://localhost:3306/AccountService?autoreconnect = true
spring.datasource.username = root
spring.datasource.password = IAmRoot
spring.datasource.testOnBorrow = true
spring.datasource.testWhileIdle = true
spring.datasource.timeBetweenEvictionRunsMillis = 60000
spring.datasource.minEvictableIdleTimeMillis = 30000
spring.datasource.validationQuery = SELECT 1
spring.datasource.max-active = 15
spring.datasource.max-idle = 10
spring.datasource.max-wait = 8000
```

Once this information is added, the database can be accessed with the JdbcTemplate object, which can be injected with the autowired annotation.

```
@Autowired
JdbcTemplate jdbcTemplate;
```

As long as the autowired annotation is used, an instance of JdbcTemplate that has access to our database, will be created. This assumes that only one database is used, and further configuration would be required to set up multiple databases. This template can be used to make queries to the database.

```
BeanPropertyRowMapper<Account> mapper = new
 ↪  BeanPropertyRowMapper<Account>(Account.class);
List<Account> rows = jdbcTemplate.query("SELECT * FROM Accounts", mapper);
```

Here the BeanPropertyRowMapper is used to convert the JSON response from the query, into an account object. The mapper can recognize the fields within the Account class and map them to the corresponding fields in the SQL database. It also knows to convert between snake and camel case.

More complex queries can be implemented as well.

```
BeanPropertyRowMapper<Account> mapper = new
 ↪  BeanPropertyRowMapper<Account>(Account.class);
String query = String.format("Select * from accounts where account_number =
 ↪  '%s'", accountNumber);

List<Account> rows = jdbcTemplate.query(query, mapper);
```

However, this does lead to a risk of SQL injection (where the end user inputs their own SQL code as an input) and parameterized queries such as what is described below in section 5.4.4 is recommended.

5.4.4 Using JPA

The most common types of queries made to a database are likely to be fairly predictable. Therefore tools such as JPA automatically create queries for developers. The JPA dependency (which can replace the Jdbc dependency) can be added to the pom.xml.

```
 <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

The mySQL database can be linked as described in section 5.4.3.

JPA will look for database entities with the `@Entity` annotation. If a table with the class name does not exist, JPA will create one with the field specifications. We can also explicitly tell JPA what the table should be called with the name parameter in the entity annotation. Below shows how to modify the Account model to become a JPA entity.

```
@Entity(name="accounts")
public class Account implements Serializable {

        @Id
        public String accountNumber;
        double value;

        @OneToMany()
        @JoinColumn(name = "fromAccount")
        List<Transaction> outgoingTransactions;

        @OneToMany()
        @JoinColumn(name = "toAccount")
        List<Transaction> incomingTransactions;

        // Getters and Setters below
```

Here account number is set as the primary key with the `@Id` annotation. Also the outgoing and incoming transactions are linked with the `@OneToMany()` and `@JoinColumn` annotations.

Once entities are defined, a repository can be created with the `JpaRepository<Entity, Key>` interface. The account interface is shown below.

```
@Repository
public interface JpaAccountRepository extends JpaRepository<Account, String>
 ↪  {
}
```

Many basic operations, such as findAll() or save(Account account) will automatically implemented. Naming conventions, such as findBy... can also be used to automatically create new queries.

```
@Repository
public interface JpaAccountRepository extends JpaRepository<Account, String>
 ↪  {
    List<Account> findByAccountNumber(String AccountNumber);
}
```

However, if there are queries that cannot be implemented with these naming conventions, custom queries can also be created. A parameterized query is shown below.

```
@Repository
public interface JpaAccountRepository extends JpaRepository<Account, String>
 ↪  {
    @Modifying
    @Transactional
    @Query("update accounts a set a.value = :value where a.accountNumber =
     ↪  :accountNumber")
    void updateValue(@Param(value = "accountNumber") String accountNumber,
     ↪  @Param(value = "value") double value);
}
```

5.5 Creating an API

In this project, two API are implemented. The first uses REST verbs to make requests, and is accessible via the Swagger UI tool. The second implements a GraphlQL API that is accessible via the GraphiQL tool.

Within a REST API, each end point is associated with a specific URL. For example `localhost:8080/getAllAccounts/` is used to to make a request to read all account entities. When the request is full-filled, the entire entity (in this case list of accounts) is sent back as a JSON object.

Within a GraphQL API, each query is made from a single URL and follows a specified schema. Here end users can specify which columns are needed for the business function. The advantage of this is that the responses do not contained unused information, which is important for cutting down on response size. This is especially useful for mobile applications.

This section assumes that data is being handled by a repository, as laid out in section 5.4.

5.5.1 REST API with Swagger UI

Spring Web contains tools needed to build REST components. To make interaction with the API easier, Swagger UI is implemented in this project. However, tools such as Postman can also be used.

The following dependencies need to be added to the pom.xml file to add Swagger to you application.

```
 <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>3.0.0</version>
</dependency>
```

To specify where to Swagger should be located the following lines can be added to the application.properties file.

```
springfox.documentation.swagger.v2.path: /api-docs
springdoc.swagger-ui.path=/swagger-ui.html
```

We also need to create a configuration class with the annotation `@Configuration` to configure swagger for our application.

```
@Configuration
public class SpringFoxConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
          .select()
          .apis(RequestHandlerSelectors.any())
          .paths(PathSelectors.any())
          .build();
    }
}
```

Once this done and the server is started, Swagger UI can be found at "localhost:8080/swagger-ui/". It will be empty however, until a rest controller is added.

To create a rest controller, add a `@RestController` annotation at the top of a class. Within this class `@ReqestMapping` can be applied to a method to locate a request at a specified URL. Within a request mapping annotation, you can also specify what the request will produce and what HTTP verb it uses.

```
@RequestMapping(value = "/admin/getAllAccounts/", produces = {
 ↪  "application/json" }, method = RequestMethod.GET)
```

The method with this annotation will host the following method at localhost:8080/admin/getAllAccounts, produce a JSON file, and is a get request. Swagger UI allows additional information to be given to Swagger, with the `@ApiOperation` annotation.

```
@ApiOperation(value = "Get all accounts from database", nickname =
 ↪  "getAllAccounts", tags = {"SQLAccounts", })
```

Along with naming the operation, this places it within the "SQLAccount" tag in Swagger.

The complete request for getAllAccount is given below.

```
@ApiOperation(value = "Get all accounts from database", nickname =
 ↪  "getAllAccounts", tags = {"SQLAccounts", })
        @RequestMapping(value = "/admin/getAllAccounts/", produces = {
         ↪  "application/json" }, method = RequestMethod.GET)
        ResponseEntity<?> getAllAccounts(){

                List<Account> queryResult = accRepo.getAllAccounts();
                return new ResponseEntity<>(queryResult, HttpStatus.OK);

        }
```

Query results gathers data from the repository accRepo, which is fed into a response entity containing both the account list and an OK status.

REST requests can also take in parameters. This is done through the `@RequestParam` annotation, and information is given to SwaggerUi with the `@ApiParam`. An example for a parameterized request is given below.

```
@ApiOperation(value = "Get an Account from number", nickname = "getAccount",
 ↪  tags = {"SQLAccounts", })
@RequestMapping(value = "/user/getSQLAccount/", produces = {
 ↪  "application/json" }, method = RequestMethod.GET)
ResponseEntity<?> getAccount(
                @ApiParam(value = "Account number") @RequestParam(value =
                 ↪  "accountNumber", required = true) String accountNumber){

        List<Account> queryResult = accRepo.getAccount(accountNumber);
        return new ResponseEntity<>(queryResult, HttpStatus.OK);

}
```

5.5.2 GraphQL API with GraphiQL

To implement a GraphQL API , the following dependencies need to be added to the pom.xml file.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-graphql</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.graphql</groupId>
  <artifactId>spring-graphql-test</artifactId>
  <scope>test</scope>
</dependency>
```

To access our GraphQL API through GraphiQL, the following lines should be added to application.properties.

```
spring.graphql.graphiql.enabled=true
spring.graphql.graphiql.path=/graphiql
```

Now our API can be accessed at "http://localhost:8080/graphiql".

To define queries and entities, a schema file needs to be created. Spring Boot will automatically look in the resources folder for any .graphqls files. An entity can be defined in the following format, which specifics what an account entity contains.

```
type Account {
    incomingTransactions: [Transaction]
    outgoingTransactions: [Transaction]
    value : Float
    accountNumber : String!
}
```

Queries that return an Account type, must return an object that has fields that match exactly what is specified in the schema. Any other fields in the object are ignored. In this project the Account class contains fields accountNumber, value, incomingTransactions, and outgoingTransactions. The brackets ([]) define an collection or list. In this schema the datatype of incomingTransactions and outgoingTransactions are set as a list of Transaction types, which is defined in a similar way as account and in the same schema. The exclamation points (!) at the end of accountNumber means that it is a required field.

A schema is also used to define queries. A new type called 'Query' is created to store all the functions we want to define.

```
type Query{
        getAllAccounts: [Account]
        getAccountByNumber (accountNumber: String): Account

        ....
}
```

Here getAllAccounts is a parameterless function that returns a list of accounts, and getAccountByNumber takes in a string called accountNumber and returns an account type.

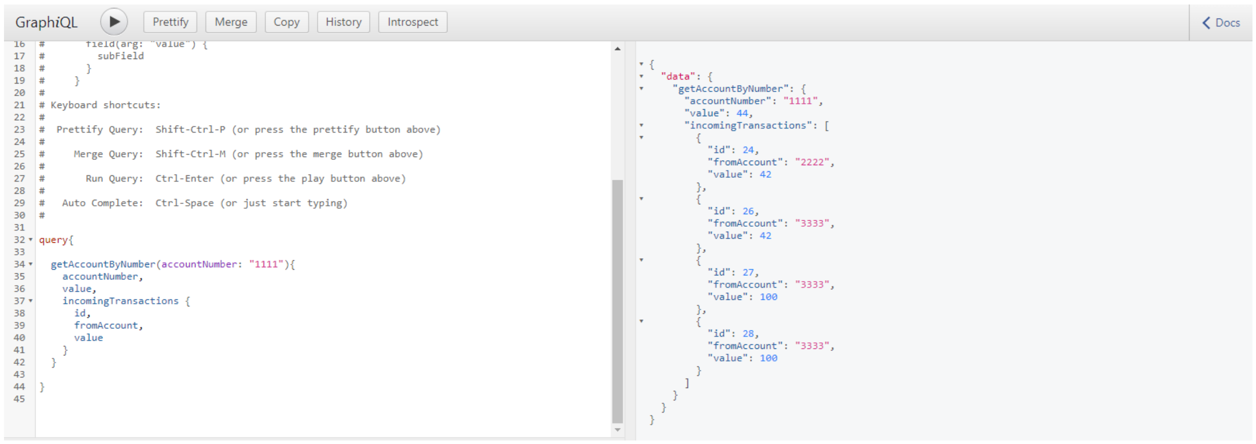
These queries can be implemented in a `@Controller` class, within a method that has a `@QueryMapping` annotation. The method name must exactly match the query name in the schema. Additionally parameters can be added with the `@Argument` annotation, and also must exactly match the parameter name in the schema. A query method can look like the following.

```
@QueryMapping
public Account getAccountByNumber (@Argument String accountNumber) {
        return  jpaAccountRepo.findByAccountNumber(accountNumber).get(0);
}
```

Additionally, GraphQL can map type relationships between entities with the `@SchemaMapping` annotation as seen in the example below. This can be used instead of mapping between entities that JPA does.

```
@SchemaMapping
        public List<Transaction> incomingTransactions (Account account){
                return jpaTransRepo.findByToAccount(account.accountNumber);
        }
```

Once the controller is complete and all queries are mapped, queries can be made at in Graphiql graphic interface. Figure 6 shows how a query can be implemented to make a request in GraphiQl. In this example, the query specifies which account to pull from the data from ("1111") along with what fields it wants in response. Notice that outgoing transactions is not requested, and therefore is not given in the output. Additionally, the incoming transactions, being another entity, requires the user to specify which field of transaction is wanted.



```
#       field(arg: "value") {
#         subField
#       }
#     }
#
# Keyboard shortcuts:
#
#  Prettify Query:  Shift-Ctrl-P (or press the prettify button above)
#
#     Merge Query:  Shift-Ctrl-M (or press the merge button above)
#
#       Run Query:  Ctrl-Enter (or press the play button above)
#
#   Auto Complete:  Ctrl-Space (or just start typing)
#

query{

  getAccountByNumber(accountNumber: "1111"){
    accountNumber,
    value,
    incomingTransactions {
      id,
      fromAccount,
      value
    }
  }

}
```

```
{
  "data": {
    "getAccountByNumber": {
      "accountNumber": "1111",
      "value": 44,
      "incomingTransactions": [
        {
          "id": 24,
          "fromAccount": "2222",
          "value": 42
        },
        {
          "id": 26,
          "fromAccount": "3333",
          "value": 42
        },
        {
          "id": 27,
          "fromAccount": "3333",
          "value": 100
        },
        {
          "id": 28,
          "fromAccount": "3333",
          "value": 100
        }
      ]
    }
  }
}
```

Figure 6: An example of using GraphiQL to create queries for getting an account

5.6 Security

Authentication and authorization is implemented in this project through the insertion of JSON Web Tokens (JWTs) into the header of the request being made. This makes for a stateless application, where user information is not stored in the session itself. In a real application the creation and storage of JWTs would likely be within a separate service, but for this example a mock service is added directly into same microservice.

5.6.1 Configuration

Spring Security is the main dependency needed to implement security on our system. Additionally, the JWT dependency will be used to create and interpret tokens.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.9.1</version>
</dependency>
```

Certain settings must be configured to implement JWT as a means of security. This can be done through a configuration class (using the `@Configuration`) that as a filterChain implemented.

```
@Autowired
private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

@Autowired
private JwtUserDetailsService jwtUserDetailsService;

@Autowired
private JwtRequestFilter jwtRequestFilter;

@Bean
public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws
 ↪  Exception {
        httpSecurity.cors().and().csrf().disable()
        .exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint).and()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
        .authorizeRequests().antMatchers("/admin/**").hasAuthority("admin")
        .antMatchers("/user/**").hasAuthority("user");

        httpSecurity.addFilterBefore(jwtRequestFilter,
         ↪  UsernamePasswordAuthenticationFilter.class);
        return httpSecurity.build();
}
```

This does several things:

1. Disable cross-site request forgery (csrf) protection: Since our application is stateless, this protection is not useful for us.

2. Set the entry point: This will set the entry point for any unauthenticated user to our custom object jwtAuthenticationEntryPoint. Since there is no login page (a JWT is created using the API itself instead), this object will just throw an error.

3. Set session to stateless: The use of JWT tokens means that our session can be stateless. No variables will be stored in the session of the application.

4. Set security points for URLs: The URLs starting with ”user” and ”admin” have been restricted to users with user or admin privilege.

5. Add a security filter: This will add the filter jwtRequestFilter before every requests, which sets authentication details before attempting the requests. Further details about the request filter are provided in section 5.6.3.

Other methods are also added to this class to configure the authentication manager.

```
public void configureGlobal(AuthenticationManagerBuilder auth) throws
↪  Exception {

        auth.userDetailsService(jwtUserDetailsService).passwordEncoder(passwordEncoder());
}

        @Bean
        public PasswordEncoder passwordEncoder() {
                return NoOpPasswordEncoder.getInstance();
        }


        @Bean
        public AuthenticationManager authenticationManager(
                AuthenticationConfiguration authConfig) throws Exception {
            return authConfig.getAuthenticationManager();
        }
```

The first method, configureGlobal sets the user service to the mock user service that will be described section 5.6.2, and adds the password encoder. For this project no encoder is used, but for production code this should be more secure. The last method authenticationManager reveals the authenticationManager so that other classes may access it.

To set up our entry point, the following class can be implemented. This will throw an error whenever an unauthorized access attempt is made.

```
@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint,
↪  Serializable {

        private static final long serialVersionUID = -7858869558953243875L;

        @Override
        public void commence(HttpServletRequest request, HttpServletResponse
        ↪  response,
                        AuthenticationException authException) throws
                        ↪  IOException {

                response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                ↪  "Unauthorized");
        }
}
```

5.6.2 Mock User service

To simulate a service containing user details, we will pre-define three users. First we will define an admin, who has access to all methods for all accounts. Next we will define two users, user1, and user2, where user1 has access to account “1111” and user2 has access to “2222”. A user only has access to requests related to their accounts. Specifically a user can:

- Get accounts with their account number
- Get transactions which their account number is either the from or to account
- Create a new transactions from which they are the sender
- Process transactions from which they are the sender

The passwords for each account are defined as ”IAmAdmin”, ”IAmUser1” and ”IAmUser2”.

The user class provided by org.springframework.security.core.userdetails.User, allows us to create these example users easily. Authorities are encoded as a string within an authority object, provided by org.springframework.security.core.authority.SimpleGrantedAuthority. The admin user is given the authorities of ”admin” and ”user”, while users are given the authority of ”user” and `"ACC_<accountNumber>"`. Using this information we can create a collection of users as shown below.

```
private Map<String, User> getUsers(){
        Map<String, User> users = new HashMap<String, User>();

        List<GrantedAuthority> user1Auth = new ArrayList<>();
        List<GrantedAuthority> user2Auth = new ArrayList<>();
        List<GrantedAuthority> adminAuth = new ArrayList<>();
        adminAuth.add(new SimpleGrantedAuthority (ADMIN_AUTH));
        adminAuth.add(new SimpleGrantedAuthority (USER_AUTH));

        user1Auth.add(new SimpleGrantedAuthority (USER_AUTH));
        user1Auth.add(new SimpleGrantedAuthority ("ACC_1111"));

        user2Auth.add(new SimpleGrantedAuthority (USER_AUTH));
        user2Auth.add(new SimpleGrantedAuthority ("ACC_2222"));


        User admin = new User("admin", "IAmAdmin", adminAuth);
        User user1 = new User("user1", "IAmUser1", user1Auth);
        User user2 = new User("user2", "IAmUser2", user2Auth);

        users.put("admin", admin);
        users.put("user1", user1);
        users.put("user2", user2);

        return users;
}
```

To create a user details service, the class must implement the UserDetailsService interface, which requires a loadUserByUsername method to be created.

```
@Override
public UserDetails loadUserByUsername(String username) throws
 ↪  UsernameNotFoundException {
        Map<String, User> users = getUsers();
        if(users.containsKey(username)) {
                return users.get(username);
        } else {
                throw new UsernameNotFoundException("User not found with
                 ↪  username: " + username);
        }
}
```

The configuration described in section 5.6.1 will inject this service into the AuthenticationManager, which will allow our controller to easily authenticate a user's given credentials.

5.6.3 Creating and validating JWT Tokens

In order to encrypt and decrypt a JWT token, a secret key must be created. We can add this to application properties.

```
jwt.secret = secret
```

JWT provides a way to construct a token given a map of claims and a subject (username). This also sets the expiration time for the token (which is set to 5 hours after creation in this example)

```
public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;

@Value("${jwt.secret}")
private String secret;

private String doGenerateToken(Map<String, Object> claims, String subject) {

        return Jwts.builder()
            .setClaims(claims)
            .setSubject(subject)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() +
             ↪  JWT_TOKEN_VALIDITY * 1000))
            .signWith(SignatureAlgorithm.HS512, secret).compact();
}
```

For this example we don't need to specify any claims, apart from the username. The user service will pull authorities granted given a username. Therefore we will send in a empty hashmap as our claim object.

```
public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        return doGenerateToken(claims, userDetails.getUsername());
}
```

This generation token assumes that authentication has already occurred, so it does not check for credential information. Authentication will occur in the controller classes for this project.

We can then decrypt a token, given the secret.

```
public <T> T getClaimFromToken(String token, Function<Claims, T>
 ↪  claimsResolver) {
        final Claims claims = getAllClaimsFromToken(token);
        return claimsResolver.apply(claims);
}

private Claims getAllClaimsFromToken(String token) {
        return
         ↪  Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();
}
```

These claim are then used to extract user name and expiration details.

```
public Date getExpirationDateFromToken(String token) {
        return getClaimFromToken(token, Claims::getExpiration);
}

public String getUsernameFromToken(String token) {
        return getClaimFromToken(token, Claims::getSubject);
}
```

With these utilities, we can then set the security context with the filter set in the configuration class (see section 5.6.1). This filter will run before any requests are made, and add the authorities of the user, encoded within a token, to the security context. Any method ran after this point can use the authorities of this user to constrain access to the method.

```
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

@Autowired
private JwtUserDetailsService jwtUserDetailsService;

@Autowired
private JwtUtil jwtTokenUtil;

@Override
protected void doFilterInternal(HttpServletRequest request,
 ↪ HttpServletResponse response, FilterChain chain)
                throws ServletException, IOException {

        final String requestTokenHeader = request.getHeader("Authorization");

        String username = null;
        String jwtToken = null;
        // JWT Token is in the form "Bearer token". Remove Bearer word and
         ↪ get
        // only the Token
        if (requestTokenHeader != null &&
         ↪ requestTokenHeader.startsWith("Bearer ")) {
                jwtToken = requestTokenHeader.substring(7);
                try {
                        username =
                         ↪ jwtTokenUtil.getUsernameFromToken(jwtToken);
                } catch (IllegalArgumentException e) {
                        System.out.println("Unable to get JWT Token");
                } catch (ExpiredJwtException e) {
                        System.out.println("JWT Token has expired");
                }
        } else {
                logger.warn("JWT Token does not begin with Bearer String");
        }

        if (username != null &&
         ↪ SecurityContextHolder.getContext().getAuthentication() == null) {

                UserDetails userDetails =
                 ↪ this.jwtUserDetailsService.loadUserByUsername(username);
                if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {

                        UsernamePasswordAuthenticationToken
                         ↪  usernamePasswordAuthenticationToken = new
                         ↪  UsernamePasswordAuthenticationToken(
                                        userDetails, null,
                                         ↪  userDetails.getAuthorities());

                        usernamePasswordAuthenticationToken
                        .setDetails(new WebAuthenticationDetailsSource()
                        .buildDetails(request));

            SecurityContextHolder.getContext()
            .setAuthentication(usernamePasswordAuthenticationToken);
                }
        }
        chain.doFilter(request, response);
}

}
```

This method looks for a JWT token in the authorization header that starts with the string “Bearer”. If the token is not expired, it will extract the username from the token. The user service is then used to find user details and set the security context, before continuing on with the request. This filter will occur before any request made, and therefore a JWT token needs to be passed in for any non-public requests.

5.6.4 Securing REST API

We need a way for an end user to create a JWT token and insert it into the headers of the request being made. We can first update our Swagger configuration class to accept a JWT token in the header.

```
@Configuration
public class SpringFoxConfig {

        private ApiKey apiKey() {
            return new ApiKey("JWT", "Authorization", "header");
        }

    private SecurityContext securityContext(){
        return SecurityContext.builder()
        .securityReferences(defaultAuth())
        .build();
    }

    private List<SecurityReference> defaultAuth(){
        AuthorizationScope authorizationScope = new
         ↪  AuthorizationScope("global", "accessEverything");
        AuthorizationScope[] authorizationScopes = new AuthorizationScope[1];
        authorizationScopes[0] = authorizationScope;
        return Arrays.asList(new SecurityReference("JWT",
         ↪  authorizationScopes));
    }
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
          .securityContexts(Arrays.asList(securityContext()))
          .securitySchemes(Arrays.asList(apiKey()))
          .select()
          .apis(RequestHandlerSelectors.any())
          .paths(PathSelectors.any())
          .build();
    }
}
```

Next we can create a controller to take in a username and password and produce a JWT token.

```
@RestController
@CrossOrigin
public class RestAuthController {

        @Autowired
        private AuthenticationManager authenticationManager; //cofigured in
         ↪  web security config

        @Autowired
        private JwtUtil jwtTokenUtil;

        @Autowired
        private JwtUserDetailsService userDetailsService;

        @ApiOperation(value = "Create a new Auth token", nickname =
         ↪  "createAuthenticationToken", tags = {"Auth", })
        @RequestMapping(value = "/authenticate", method = RequestMethod.POST)
        public ResponseEntity<?> createAuthenticationToken(
                        @ApiParam(value = "Username") @RequestParam(value =
                         ↪  "username", required = true) String username,
                        @ApiParam(value = "Password") @RequestParam(value =
                         ↪  "password", required = true) String password
                        ) throws Exception {

                try {
                        authenticate(username, password);


                } catch (Exception e) {
                        return new ResponseEntity<>("Authentication Failed",
                         ↪  HttpStatus.UNAUTHORIZED);

                }

                final UserDetails userDetails = userDetailsService
                                .loadUserByUsername(username);

                final String token = jwtTokenUtil.generateToken(userDetails);
                return ResponseEntity.ok(new JwtResponse(token));


        }


        private void authenticate(String username, String password) throws
         ↪  Exception {
                try {
                        authenticationManager.authenticate(new
                         ↪  UsernamePasswordAuthenticationToken(username,
                         ↪  password));
                } catch (DisabledException e) {
                        throw new Exception("USER_DISABLED", e);
                } catch (BadCredentialsException e) {
                        throw new Exception("INVALID_CREDENTIALS", e);
                }
        }
}
```

Since the authentication URL ("localhost:8080/authenticate") doesn't start with user or admin, unauthenticated users can access it. The token generated from this request can be entered into the Authorize button (see figure 7 and 8) that now appears at the upper left of the Swagger screen. Our system expects JWT tokens to start with the string "Bearer". Therefore the string "Bearer" should be added before the token. Once this token is entered, all request made from Swagger UI will contain it in the header of the request.

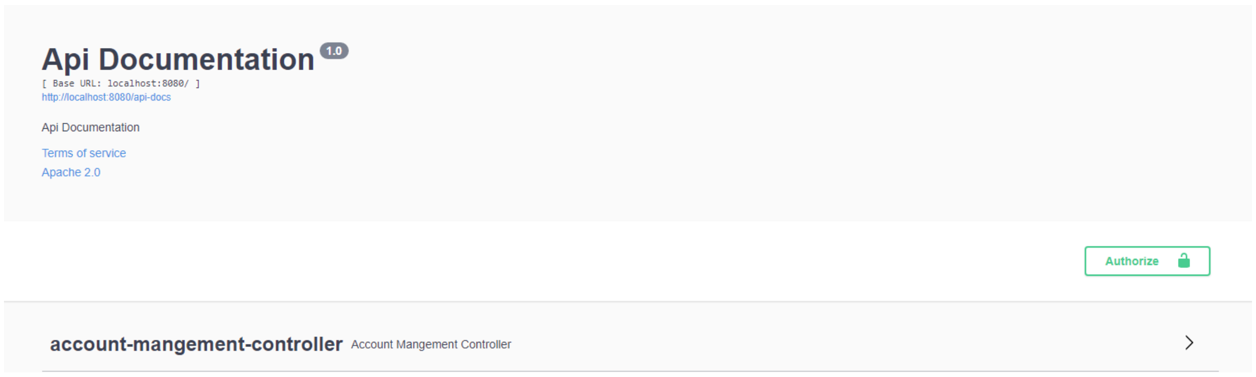
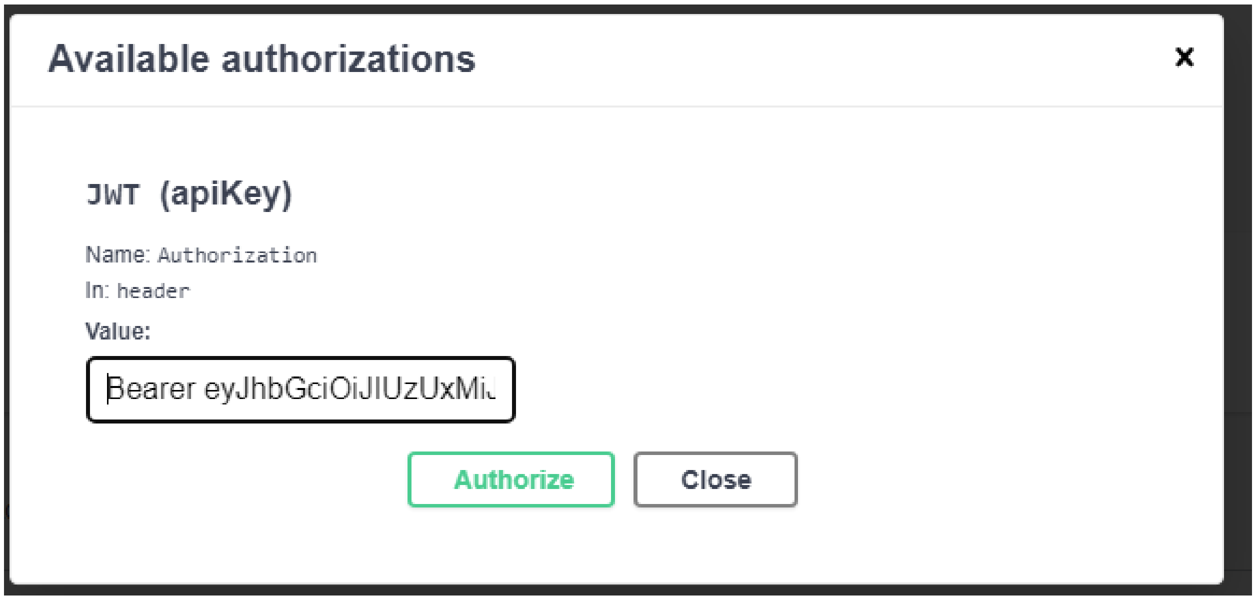

Api Documentation 1.0
[Base URL: localhost:8080/]
http://localhost:8080/api-docs
Api Documentation
Terms of service
Apache 2.0
Authorize
account-mangement-controller Account Mangement Controller

Figure 7: Swagger UI with the authorize button visual

REST requests are somewhat simpler to secure than GraphQL, as each request exists on an independent URL. It is possible to restrict access to every individual URL, but this can become tedious. To make security simpler to implement, all admin level requests have the prefix "admin" in their URL and all user requests have the prefix "user." Our security configuration already restricts non-admins from URLs with the prefix "admin" and non-user from URLs starting with "user". However, even within user requests, we may need to filter based on parameters. For example, all users should be able to access the getAccount request, but only for their own account. That is, in addition to the "user" authority they will require a `"ACC_<accounNumber>"` for the account they are requesting. This is done through the `@PreAuthorize` annotation as demonstrated below.

```
@PreAuthorize("hasAnyAuthority('admin', 'ACC_' + #accountNumber)")
@RequestMapping(value = "/user/getSQLAccount/", produces = {
 ↪  "application/json" }, method = RequestMethod.GET)
```


Available authorizations

JWT (apiKey)

Name: Authorization

In: header

Value:

Bearer eyJhbGciOiJIUzUxMi

Figure 8: How to enter a Jwt token into Swagger UI

```
ResponseEntity<?> getAccount(
                @ApiParam(value = "Account number") @RequestParam(value =
                 ↪  "accountNumber", required = true) String accountNumber){

        List<Account> queryResult = accRepo.getAccount(accountNumber);
        return new ResponseEntity<>(queryResult, HttpStatus.OK);

}
```

In this annotation `#accountNumber` references the parameter accountNumber of the method. From this only admins or people with `ACC_<accounNumber>` that matches the accountNumber parameter, can access this method.

However, sometimes information within a return object is needed to determine security level, such as in getTransactionById. This can be done with the `@PostAuthorize` annotation, as in the example below.

```
@PostAuthorize("hasAnyAuthority('admin', 'ACC_' +
 ↪  returnObject.getBody().get(0).fromAccount, 'ACC_' +
 ↪  returnObject.getBody().get(0).toAccount)")
@RequestMapping(value = "/user/getTransactionById/", produces = {
 ↪  "application/json" }, method = RequestMethod.GET)
ResponseEntity<?> getTransactionById(
                @ApiParam(value = "Id number") @RequestParam(value = "ID",
                 ↪  required = true) int id){

        List<Transaction> queryResult = transRepo.getTransactionFromId(id);

        return new ResponseEntity<>(queryResult, HttpStatus.OK);

}
```

In this example returnObject references whatever is returned from the method. Here returnObject will be the transaction the user requested. We want to restrict access to this object to only users that are either the sender or receiver of the transaction. Therefore we require that the user has the authority for either the fromAccount or toAccount of the transaction.

The `@PostAuthorize` annotation should be used with cation as it filters after the method is ran. In this case it will stop the object from being returned, but not from the query being made to the repository. For a get method this is fine, but this should not be used for any post or update methods.

5.6.5 Securing GraphQL API

Securing GraphQLs is similar is many ways. First we need to add a controller to generate JWT tokens, by updating the schema.graphqls file.

```
type Query{

        ....

        getAuthToken(username : String, password: String): String

        ....
}
```

Along with the controller Java Class.

```
@Controller
public class GraphQLController {
....
    @QueryMapping
        String getAuthToken (@Argument String username, @Argument String
         ↪  password) throws Exception {
                try {
                        authenticate(username, password);


                } catch (Exception e) {
                        throw new Exception(e);

                }

                final UserDetails userDetails = userDetailsService
                                .loadUserByUsername(username);

                final String token = jwtTokenUtil.generateToken(userDetails);

                return token;
        }

            private void authenticate(String username, String password)
             ↪  throws Exception {
            try {

                    authenticationManager.authenticate(new
                     ↪  UsernamePasswordAuthenticationToken(username,
                     ↪  password));
            } catch (DisabledException e) {
                    throw new Exception("USER_DISABLED", e);
            } catch (BadCredentialsException e) {
                  throw new Exception("INVALID_CREDENTIALS", e);
            }
    }

 ....

 }
```

Within the GrahiQL interface we can enter a JWT token in the header at the bottom of the page. This header needs to take the form of `{Authorize : Bearer <JwtToken>}`. After this, all queries will contain the JWT Token in the header of the request.

Unlike REST, GraphQL uses a single URL, which makes filtering URLs by privileges not possible. Instead every method in our controller needs to contain a `@PreAuthorize` or a `@PostAuthorize` annotation to be secure. An example is given below.

```
@QueryMapping
@PreAuthorize("hasAnyAuthority('admin', 'ACC_' + #accountNumber)")
public Account getAccountByNumber (@Argument String accountNumber) {
        return  jpaAccountRepo.findByAccountNumber(accountNumber).get(0);

}
```

5.7 Testing

The tests for this service fall into three categories. The most isolated tests, are the unit tests. Here only one class (the controller classes) are tested and all other components (repositories) are mocked. The second most isolated is the integration testing, which tests the connection between the repository and the database. The highest level test, is the component test which make requests to the API as if it was and end-user or another service.

All tests are found in the `src\test` folder in our project, and run when a Maven Install is ran. They can also be ran using JUnit. This is done in Eclipse through run configurations (which can be found in the tab ”run”).

The following dependencies are going to be used for testing.

```
<!-- Junit dependacies -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
```

```
</dependency>

<!-- Handle JSon files -->
<dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.6</version>
</dependency>
```

5.7.1 Unit Testing

To get our unit tests to run the server and run along with the base test case, we add the following annotations to the test class.

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = ExampleServiceApplication.class)
```

This allows us to use the `@Autowired` function. The downside to this method is that it runs the entire server, which is more than what is necessary for the unit test. If faster tests are desired, other methods for running tests should be considered.

We also want to be able have security access to all the methods we want to test. We can do this with the WithMockUser annotation. For now we will provide our mock user with admin access, but this can be used to test security methods as well.

```
@WithMockUser(username = "admin", authorities = { "admin", "user" })
```

To have access to the controller class we want to test, we can get it with the `@Autowired` annotation.

```
@Autowired
GraphQLController controller;
```

However, we only want to test the controller, and not the repositories it relies on. Therefore we will mock the repositories with the Mock Bean annotation.

```
@MockBean
JpaAccountRepository accountRepo;

@MockBean
JpaTransactionRepository transRepo;
```

The `@MockBean` annotation will inject a fake version of the repositories into our controller. By default it will return null for all calls made to them, but we can tell it to send back specific values also. To set up the mock repositories, we can add these specifications to a method with a `@Before` annotation, which will run before the test methods are called. For example we can mock the findAll method for the account repository, using the method shown below.

```
@Before
public void accountSetup() {
        acc1 = new Account("1111", 1111);
        acc2 = new Account("2222", 2222);

        ArrayList<Account> accounts = new ArrayList<Account>();
        accounts.add(acc1);
        accounts.add(acc2);

        Mockito.when(accountRepo.findAll()).thenReturn(accounts);
        }
```

After accountSetup is ran, whenever our controller calls accountRepo.findAll(), it will return this sample of accounts instead of any real accounts in our database. We can also specify what parameters should be given to the repository to send a specific response.

```
ArrayList<Account> account1 = new ArrayList<Account>();
account1.add(acc1);

Mockito.when(accountRepo.findByAccountNumber("1111")).thenReturn(account1);
```

Sometimes we don't care what the parameter is so we can use the ”any” method provided in mockito.ArgumentMatchers.* to allow any input of a given type. We can also use the returnFirstArugment method provided, in the AdditionalAnswers class, to return the argument object instead of a pre-defined object. The combination of these two methods allows us to make implement a save method that simply return the account object it was given.

```
Mockito.when(accountRepo.save(any(Account.class)))
.then(AdditionalAnswers.returnsFirstArg());
```

Sometimes we need slightly more complicated mock functions, which can be done with the doAnswer method and an AnswerObject. For example, we can have the updateValue method, change the value of our fake accounts.

```
Mockito.doAnswer(new Answer<Object>() {
    @Override
    public Object answer(InvocationOnMock a) throws Throwable {
            if(a.getArgument(0).equals("1111")) {
                    acc1.setValue(a.getArgument(1));
            }else if (a.getArgument(0).equals("2222")) {
                    acc2.setValue(a.getArgument(1));
            }
            return null;
    }
  }).when(accountRepo).updateValue(anyString(), anyDouble());
```

Each test should have a `@Test` annotation to tell JUnit to run it. By default a test method passes when no errors are thrown, and fails otherwise. To make a test fail when unexpected values are returned, we can use the assertTrue function provided in org.junit.jupiter.api.Assertions.

```
@Test
public void getAccount() {
        Account account = controller.getAccountByNumber("1111");

        assertTrue(account.getValue() == 1111);
}
```

This method will fail whenever the account returned from the controller does not have the value 1111. This can be done also with the assertEquals method.

```
@Test
public void getAccount() {
        Account account = controller.getAccountByNumber("1111");

        assertEquals(1111, account.getValue());
}
```

Multiple assert functions can be added to one unit test as well.

```
@Test
public void createTransaction(){
        Transaction transaction = controller.createTransaction("1111",
         ↪  "2222", 20);

        assertEquals(transaction.getFromAccount(), "1111");
        assertEquals(transaction.getToAccount(), "2222");
        assertEquals(transaction.getValue(), 20);
}
```

To test security at the method level (those that use `@PreAuthorize` or `@PostAuthorize`), the `@WithMockUser` annotation can be use to test specific authorities. For example we can test that an admin user can access the method getAllAccounts using the following test.

```
@Test
@WithMockUser(username = "admin", authorities = { "admin", "user" })
public void getAllAccountAdmin() {
    controller.getAllAccounts();
}
```

However, we do not want a regular user to access this method. To created a test that is expected to fail (like when a user attempted to gain access to an admin method), we can add a expected parameter to the test annotation. This takes in an exception class. The exception for unauthorized requests is an AccessDeniedException exception.

```
@Test(expected = AccessDeniedException.class)
@WithMockUser(username = "user1", authorities = {"user", "ACC_1111" })
public void getAllAccountUser() {
        controller.getAllAccounts();
}
```

It should be noted that `@WithMockedUser` cannot be used to test method that are restricted based on URL (as is done here with the REST API). To test the security for these methods a MockMvc object can be used to access the request.

```
@Autowired
private MockMvc mvc;

private String getAdminToken() {
        UserDetails admin =  userService.loadUserByUsername("admin");
        return "Bearer "  + tokenUtil.generateToken(admin);
}

@Test
public void getAllAccountAdmin() {
        try {
                mvc.perform(get("/admin/getAllAccounts/").header("Authorization",
                 ↪  getAdminToken())).andExpect(status().isOk());
        } catch (Exception e) {
                assertTrue(false);
        }
}
```

Here we make a request given an admin token and attempt to get all accounts. We except that this request returns on ok status, and the test will fail otherwise. We can add the same mocked repositories here to avoid any changes to the real database being made. More details about MockMvc are given in section 5.7.3.

5.7.2 Integration Testing

Since only a single service is implemented in this example, the only integration tests that are implemented test the connection between the repository and the database. It should be noted that since these tests actual connections with the database, there is a risk of altering the database itself. Since this is not a real application, this is not a huge concern. However, for real applications a separate test and production database should be implemented.

In this example, integration tests work nearly identical to unit tests. The only exception is the exclusion of the `@MockBean` objects.

```
@Autowired
JdbcAccountManagementRepository accountRepo;

@Test
 public void getAllAccount() {
        List<Account> accounts = accountRepo.getAllAccounts();
        assertNotEquals(accounts.size(), 0);
}
```

5.7.3 Component Testing

In component testing we role play an end user, who knows nothing about our application but the exposed interfaces. A useful tool for component testing is the MockMvc class, which allows us to make requests as if we were a user. MockMvc can be injected with the `@Autowired` annotation.

```
@Autowired
private MockMvc mvc;
```

We can call a request with mvc.preform method and add parameters with .param. We can then return a MvcResults object with using .thenReturn. The example below shows how to do this for our authentication request.

```
MvcResult tokenResult = mvc.perform(post("/authenticate/")
        .param("username", "user1").param("password",
         ↪  "IAmUser1")).andReturn();
```

This MvcResult object contains the result which we can access as a string version of the JSON object.

```
String response = tokenResult.getResponse().getContentAsString();
```

The Gson class provided by com.google.gson.Gson provides useful methods for dealing with JSON files. One of these allows us to turn a string JSON into another type of object. Using a TypeToken we can turn our JSON string into a map where we can access the token string.

```
Map<String, Object> responseMap = gson.fromJson(response, new
 ↪  TypeToken<Map<String, Object>>() {}.getType());

String token = (String) responseMap.get("token");
```

Once a token is generated, it can be placed in the header of the next requests.

```
MvcResult accountResult = mvc.perform(get("/user/getAccountByNumber/")
        .param("accountNumber", "1111").header("Authorization", "Bearer " +
         ↪  token))
        .andReturn();


String accountResponse = accountResult.getResponse().getContentAsString();
```

Gson can also turn the JSON string into one of our custom model objects.

```
List<Account> account  = gson.fromJson(accountResponse, new
 ↪  TypeToken<List<Account>>() {}.getType());

assertEquals(account.get(0).getAccountNumber(), "1111");
```

We can also make sure that a user1 does not have access to admin methods, by using the same token for an admin request.

```
mvc.perform(get("/admin/getAllAccounts/")
        .header("Authorization", "Bearer " + token))
        .andExpect(status().isForbidden());
```

In this example, mvc is told to expect a forbidden status for the request. If the request isn't forbidden (the user is able to access this request), an error will be thrown and the test will fail. A similar method can be used to test horizontal privilege elevation attempts.

```
mvc.perform(get("/user/getAccountByNumber/")
        .param("accountNumber", "2222").header("Authorization", "Bearer " +
         ↪  token))
        .andExpect(status().isForbidden());
```

Testing the GraphQl API is more complicated with MockMvc, given that all requests are made on the same URL. To do this we will build a request using a string query and a JSONObject containing variables. We first need to define the query we want to preform with variables.

```
query GetUser($user : String ,  $pass : String) {}
        getAuthToken(username: $user, password: $pass)
}
```

Which can be represented as a string within our test class.

```
String authQuery = "query GetUser($user : String ,  $pass : String){\n" +
        "  \n" +
        "       getAuthToken(username: $user, password: $pass)\n" +
        "}";
```

We can then create a variables JSON object and insert our variables.

```
JSONObject variables = new JSONObject();
variables.put("user", "user1");
variables.put("pass", "IAmUser1");
```

A helper function can join our string query and variables into one request as a string.

```
private String generateRequest(String query, JSONObject variables) throws
 ↪  JSONException {
            JSONObject jsonObject = new JSONObject();
            jsonObject.put("query", query);
            if (variables != null) {
              jsonObject.put("variables", variables);
            }
            return jsonObject.toString();
          }
```

To get the response back we need to preform both a post request and an asynchronous dispatch.

```
MvcResult authResult = mvc.perform(post("/graphql")
        .content(generateRequest(authQuery, variables))
        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andReturn();

String authResponse = mvc.perform(asyncDispatch(authResult))
        .andExpect(status().isOk())
        .andReturn().getResponse().getContentAsString();
```

Another helper-function can be used to extract just the data section from this response.

```
private JSONObject parseResponse(String response) throws JSONException {
          JSONObject obj = new JSONObject(response);

         return  obj.getJSONObject("data");
  }
```

Which can be used to get the access token.

```
JSONObject data = parseResponse(authResponse);
String token = data.getString("getAuthToken");
```

Which can be inserted into the header in a similar way to the REST API.

```
String getAccountQuery = "query \n" +
        "  GetAccount($accountNumber: String ){\n" +
        "    getAccountByNumber(accountNumber: $accountNumber){\n" +
        "    accountNumber, \n" +
        "    value \n" +
        "          }\n" +
        "   }";
variables = new JSONObject();
variables.put("accountNumber", "1111");
```

```
MvcResult accResult = mvc.perform(post("/graphql")
        .header("Authorization", "Bearer " + token)
    .content(generateRequest(getAccountQuery, variables))
    .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andReturn();

String accResponse = mvc.perform(asyncDispatch(accResult))
    .andExpect(status().isOk())
    .andReturn().getResponse().getContentAsString();

JSONObject accData =
 ↪  parseResponse(accResponse).getJSONObject("getAccountByNumber");

assertEquals("1111", accData.getString("accountNumber"));
```

Unlike the REST API, GraphQL requests doesnt return a forbidden status when unauthorized access attempts occurred. Instead the returned JSON object will have a errors variable, which will contain an error with a forbidden message in it. A helper function can be written to check for this message.

```
private boolean isForbidden(String response) throws JSONException {
    JSONObject obj = new JSONObject(response);
    Gson gson = new Gson();

    try {
        JSONArray jsons =   (JSONArray) obj.get("errors");
        for(int i = 0; i < jsons.length(); i ++) {
         JSONObject json = jsons.getJSONObject(i);
         if (json.getString("message").equals("Forbidden")) {
                 return true;
         }
        }
        return false;

    } catch (Exception e) {
        return false;
    }
  }
```

Which can be used to ensure unauthorized requests are stopped.

```
MvcResult adminResult = mvc.perform(post("/graphql")
    .header("Authorization", "Bearer " + token)
    .content(generateRequest(adminQuery, variables))
    .contentType(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andReturn();

String adminResponse = mvc.perform(asyncDispatch(adminResult))
    .andExpect(status().isOk())
    .andReturn().getResponse().getContentAsString();

assertTrue(isForbidden( adminResponse ));
```

6 Conclusion

The popularity of Microservice Architecture (MSA) is not likely going diminish in the next couple years. Its design model of small, independent services, aligns well with the current demand on modern web applications. It allows for scalable, robust, and modifiable applications. However there are several key challenges and opportunities that arise from this design model. The added complexity of many interacting components means special consideration for security and testing must be made. This provides an opportunity for innovation to provide new tools to help developers to combat these problems.

As an example for what a microservice could look like, we demonstrated how a microservice could be implemented using Java with Spring Boot and mySql. This service was able to access, create, modify, and delete entries in a database through either REST or GraphQL APIs. This application was secured with JWT tokens that are passed with the headers of each request and grant user individual authorities for a set period of time. JUnit was the main unit of testing implemented that made use of tools such as Mockito and MockMVC to preform several levels of testing.

This paper was meant as a introduction to developers who are new to microservice design philosophy. It first gave a high level overview of the design tenants and challenges that occur is MSA based applications. It then provided an example of how a microservice can be implemented using Java and Spring Boot. The goal of this paper was to provide a broad enough overview of the topic that new developers can gain confidence in implementing MSA based applications. However, this paper should also be used as a jumping off point for new developers to explore more about the underlying technologies they are using.

Appendices

A Term Table

Acronym	Meaning	Definition
API	Application Program Interface	A method to describe how a user can interact with a software
CI/CD	Continuous Integration / Continuous Deployment	A method implemented by development teams to frequently deploy new changes
GraphQL	Graph Query Language	An alternative to REST where the user can define the return type using pre-defined queries
HTTP	Hypertext Transfer Protocol	A set of protocols for sending information over the internet
MSA	MicroService Architecture	A design philosophy emphasizing small independent services
JDBC	Java DataBase Connectivity	A API to interact with databases in Java
JPA	Java Persistence API	Defines a way to interact with data the presists after a program is terminated, such as a database
JSON	JavaScript Object Notation	A human readable way to format entities that can be accessed across technologies and langauges
JUnit	Java Unit	A tool for creating unit tests in java
JWT	Javascript Web Token	An encrypted and short lived token that stores user information and can be used for authentication
REST	REpresentational State Transfer	An interface style emphasizes statelessness and uniform interface
SOA	Service Oriented Architecture	A software architecture were applications are broken up into individual components
SOAP	Simple Object Access Protocol	A communication framework that passes XML files between two entities
SQL	Structured Query Language	A language for dealing with databases and making queries
URL	Uniform Resource Locators	A way to uniquely specify location on the web
XML	eXtensible Markup Language	A markup-language that can be used to format web communication, especially in SOAP interfaces

References

[1] Gleison Brito and Marco Tulio Valente. “REST vs GraphQL: A controlled experiment”. In: *2020 IEEE international conference on software architecture (ICSA)*. IEEE. 2020, pp. 81–91.

[2] Antonio Bucchiarone et al. “From Monolithic to Microservices: An Experience Report from the Banking Domain”. In: *IEEE Software* 35.3 (2018), pp. 50–55. DOI: 10.1109/MS.2018.2141026.

[3] Lianping Chen. “Microservices: architecting for continuous delivery and DevOps”. In: *2018 IEEE International conference on software architecture (ICSA)*. IEEE. 2018, pp. 39–397.

[4] Bruno Costa et al. “Evaluating a Representational State Transfer (REST) architecture: What is the impact of REST in my architecture?” In: *2014 IEEE/IFIP Conference on Software Architecture*. IEEE. 2014, pp. 105–114.

[5] Nicola Dragoni et al. *Microservices: How To Make Your Application Scale*. 2017. DOI: 10.48550/ARXIV.1702.07149. URL: https://arxiv.org/abs/1702.07149.

[6] Hartmut Fischer. “Testing in microservice systems: A repository mining study on open-source systems using contract testing”. In: (2021).

[7] Konrad Gos and Wojciech Zabierowski. “The comparison of microservice and monolithic architecture”. In: *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. IEEE. 2020, pp. 150–153.

[8] Robert Heinrich et al. “Performance engineering for microservices: research challenges and directions”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 2017, pp. 223–226.

[9] Mafalda Isabel Landeiro and Isabel Azevedo. “Analyzing GraphQL performance: a case study”. In: *Software Engineering for Agile Application Development*. IGI Global, 2020, pp. 109–140.

[10] Kathryn B Laskey and Kenneth Laskey. “Service oriented architecture”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 1.1 (2009), pp. 101–105.

[11] Ali Rezaei Nasab et al. “An Empirical Study of Security Practices for Microservices Systems”. In: *arXiv preprint arXiv:2112.14927* (2021).

[12] Chaitanya K Rudrabhatla. “Security Design Patterns in Distributed Microservice Architecture”. In: *arXiv preprint arXiv:2008.03395* (2020).

[13] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. “A Systematic Mapping Study on Microservices Architecture in DevOps”. In: *Journal of Systems and Software* 170 (2020), p. 110798. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110798. URL: https://www.sciencedirect.com/science/article/pii/S0164121220302053.

[14] Muhammad Waseem et al. “Design, monitoring, and testing of microservices systems: The practitioners’ perspective”. In: *Journal of Systems and Software* 182 (2021), p. 111061.

[15] Muhammad Waseem et al. “On the nature of issues in five open source microservices systems: An empirical study”. In: *Evaluation and Assessment in Software Engineering*. 2021, pp. 201–210.

[16] Erik Wittern et al. “An empirical study of GraphQL schemas”. In: *International Conference on Service-Oriented Computing*. Springer. 2019, pp. 3–19.

[17] Tetiana Yarygina and Anya Helene Bagge. “Overcoming security challenges in microservice architectures”. In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE. 2018, pp. 11–20.