# Assumption:

For this project, I had performed basic research to understand if suburbs and postcodes exhibit a 1:1 relation. It turns out that multiple suburbs could map to one postcode, and a suburb could have multiple postcodes. An example of the former is postcode 3168 which is shared by Clayton and Nottinghill. An example of the latter is the Parramatta Suburb with 2124 and 2150 postcodes. As a result, each record in the DB would be uniquely identified by the combination of postcodes and suburbs (composite primary key).

# API endpoints and Response Design

## Get Endpoints:

* An API that allows clients to retrieve the suburb information by postcode
  + http://localhost:8080/api/v1/area/suburb/{postcode}
  + Example: <http://localhost:8080/api/v1/area/suburb/3168>
* An API that allows clients to retrieve a postcode given a suburb name
  + http://localhost:8080/api/v1/area/postcode/{suburb}
  + Example: http://localhost:8080/api/v1/area/postcode/Mulgrave

## Post Enpoint:

* API to add new suburb and postcode combinations
  + http://localhost:8080/api/v1/area
  + Example: <http://localhost:8080/api/v1/area/>
  + Note: Request body must contain suburb and postcode combination and the header must contain cookie ID and CSRF token. This has been detailed in Security and Demo section.

The GET endpoints return an array of objects with either suburb or postcode as the only property in the objects as shown in the figure below. Although, I could have designed it to return a list of Suburbs/postcodes, I decided to not use that approach as the array of objects approach would provide better extensibility should we need to send additional properties (such as population, area) unique to each suburb/postcode in the future.



# Set up

Before being to access/store data, a Database needs to be spun up using Docker. All the steps regarding how to get the application up and running have been detailed in the read.md file. Upon running the application, some basic test data that has been set up in the application will be stored in the DB. Upon stopping the application, the table and its data will be dropped.

# Security

I was able to build a secure API by using basic authentication and CSRF tokens. The user needs to first send a get request and include a basic authentication in the request. If the user doesn’t include the basic authentication header, they’d receive a 401 unauthorised error message. But once the user is authenticated, they needn’t have to include it again in the headers. Note: For the Post requests in particular, the CSRF token and the cookie ID from the get response needs to be included in the header every time a POST request is made. This is to add a layer of security because the data could be changed if there is no protection via the POST request.

Diagram

Description automatically generated

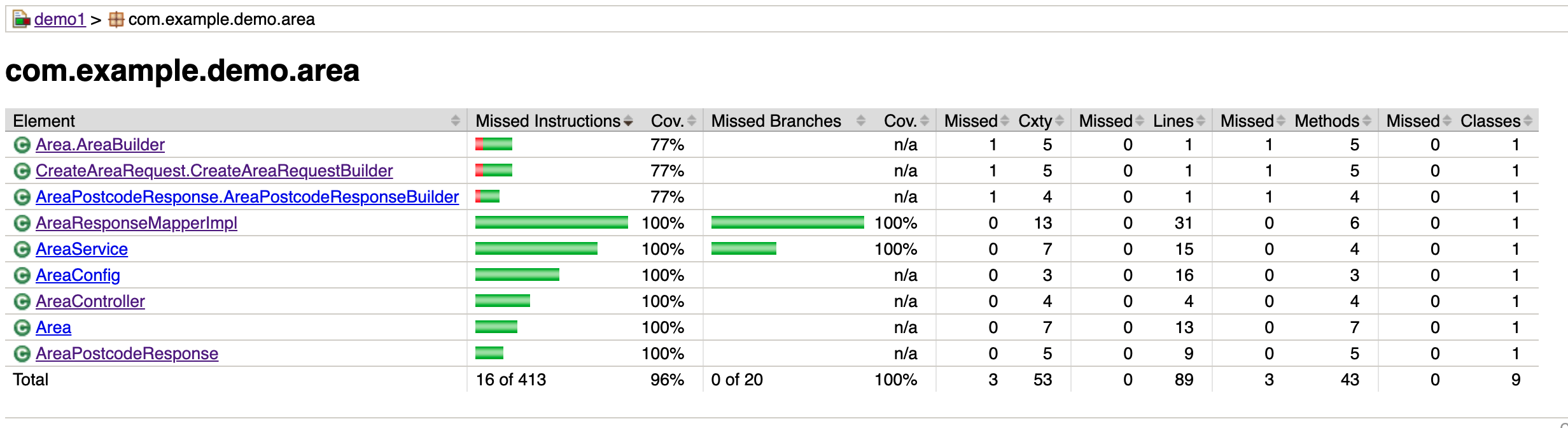
The credentials for the basic Authorization can be found below:

Username: demoUserName

Password: demoUserPassword

# Testing

During the testing process, I endeavoured to achieve as much coverage as possible. Unit Testing was performed on the Repository, Services and Mapper Classes. The tests for the Repository utilised the in-memory H2 DB. Additionally, integration test was performed on the Controller Test. The project uses a Jacoco plugin, and the Jacoco Coverage Test Report could be found at target/site/jacaco.html. A screenshot of the coverage report for the tests, however, can be seen below.



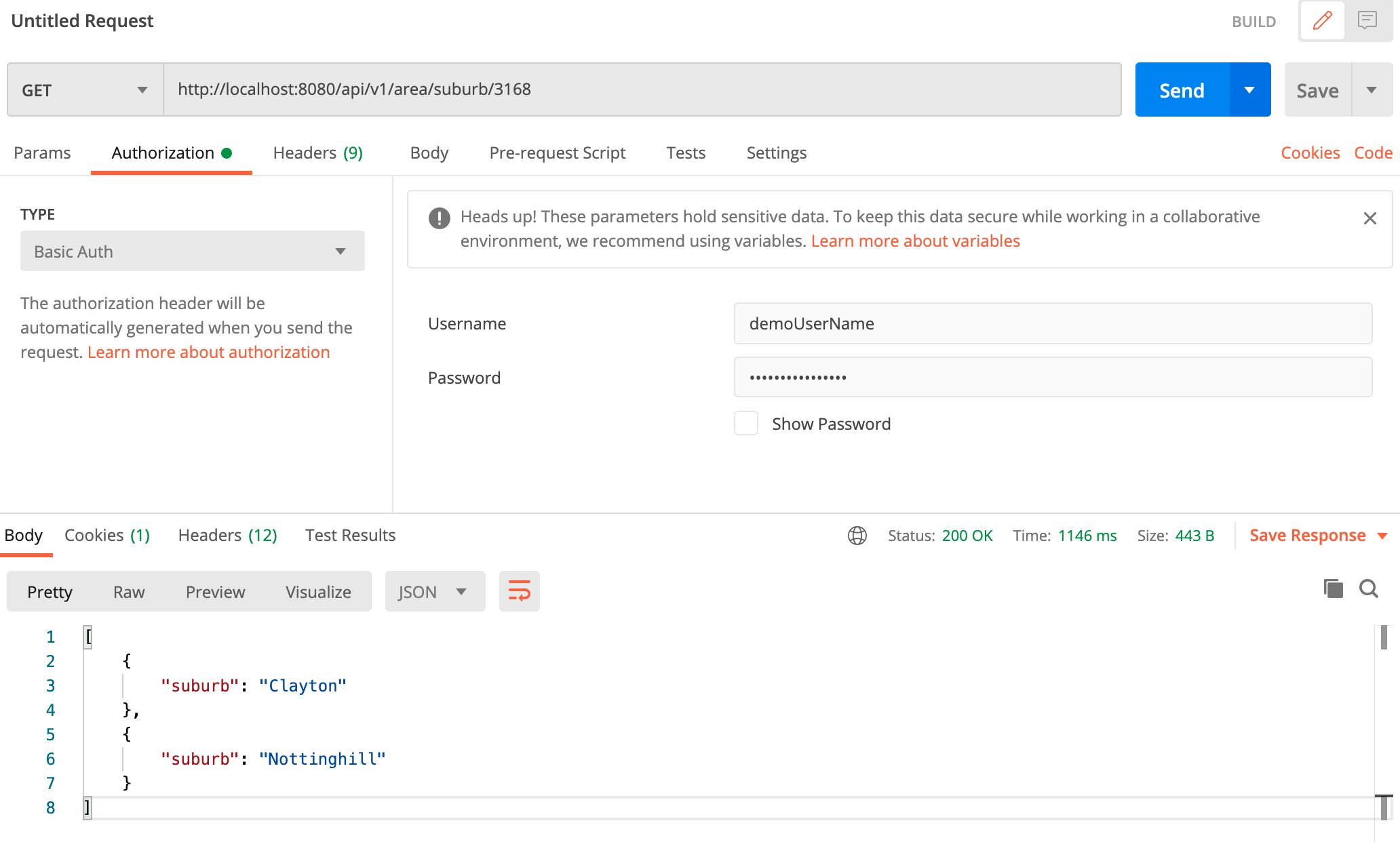
# Folder Structure

Since this is a small project, I have arranged the folder structure to have a package named area which has the controller, repository, service, and mapper classes in it. This helps in easily accessing the files. Should the scope of this project extend to include other controllers, services, and repositories, it might be worthwhile to refactor each class into their separate controller, services, repository folders.

# Sample Demo

Please note that at the start of every session, the user needs to send a GET request to any one of the 2 GET endpoints to authenticate and obtain the cookies and the CSRF token for the session. The user can thereafter skip having to send the authorization token in the subsequent requests.   
  
Additionally, please NOTE that while using Postman, I found that the GET request needs to be sent twice to authenticate the user before using the cookie and CSRF token from the response to make a POST request.

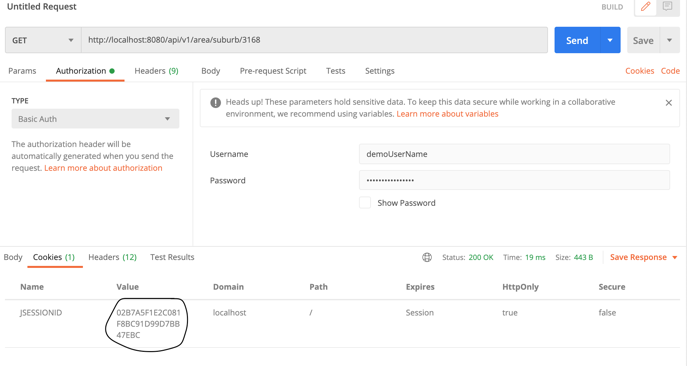
While Using Postman, the user can select Basic Auth and provide the credentials in the name and password fields as shown below.



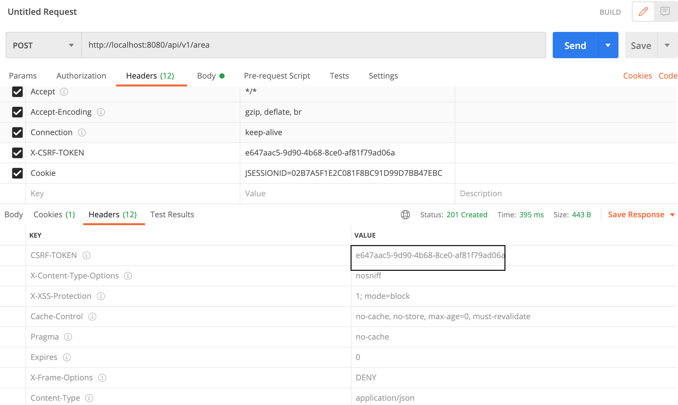
Upon receiving the response, make another GET request in Postman. (Can resend the first request). The reason for this has been detailed above.

Then extract the cookie ID and the CSRF tokens from the GET request. These tokens need to be used in every POST request.

Obtain Cookie value

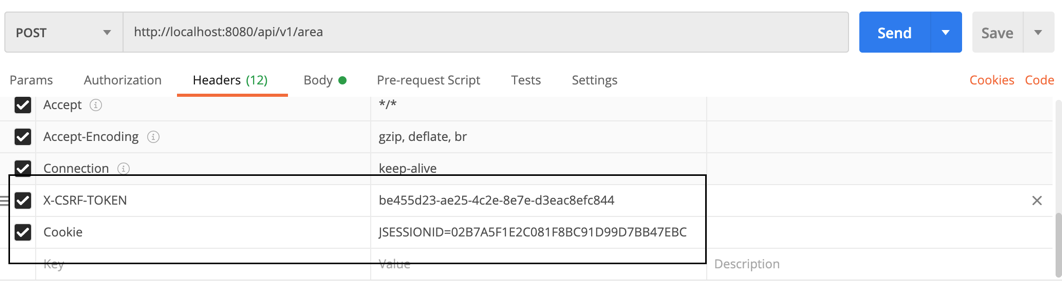


Obtain CSRF Token



Add these tokens in the POST Request’s header. Note that the name of the CSRF token is

X-CSRF-TOKEN.



Send the POST request.

