

Database Essentials Home Assignment Documentation:

This document covers the criteria of screenshots as outlined by the Appendices section and more.

The following criteria are answered in this document.

Appendices

Appendix A: Submission Checklist

- ♦ Public GIT (Github/Bitbucket/Gitlab) url:
- ♦ Public API (Vercel or otherwise) url:
- ♦ Task 1 Screenshot of your development environment setup:
- ♦ Task 2: Screenshot/s of your Mongo Database on MongoAtlas:
- ♦ Task 3A: Screenshot of the API running on localhost in your browser:
- ♦ Task 3B: Screenshot of the hosted API running on a public url such as Vercel:
- ♦ Task 4A: Screenshot showing credential setup.
- ♦ Task 4B: Screenshot showing IP whitelisting.
- ♦ Task 4C: Screenshot showing how you are preventing SQL injection.

SQL Injection Notice:

Kindly note that given no SQL is ever parsed by the Spring boot server, there poses no security concern in this regard. In light of this, this document will not contain images or documentation related to any measures taken to prevent it. However, some form of data sanitisation has been performed. Please see the end of this document for such evidence. See the “Data Sanitisation” section for this documentation.


Task 1:

Git URL: https://github.com/Null-Nil-None/database_essentials

Hosted public URL: <https://database-essentials.onrender.com>

Screenshot of the development environment:

Maven was used as the package manager and build tool, whilst VS Code was used as the integrated development environment (IDE) for writing the source.



.vscode	01/04/2025 21:49	File folder	
_resources	01/04/2025 21:49	File folder	
documentation	04/04/2025 13:02	File folder	
src	01/04/2025 21:49	File folder	
target	02/04/2025 23:38	File folder	
.render.yaml	02/04/2025 22:36	Yaml Source File	1 KB
Dockerfile	01/04/2025 22:10	File	1 KB
pom.xml	01/04/2025 15:31	XML Source File	3 KB
README.md	01/04/2025 16:08	Markdown Source...	2 KB

Figure 1: Shows all root-level files and folders, predominantly, `the pom.xml`

Screenshots of the MongoDB Collections:

The following are screenshots of the MongoDB’s collections, or rather tables.

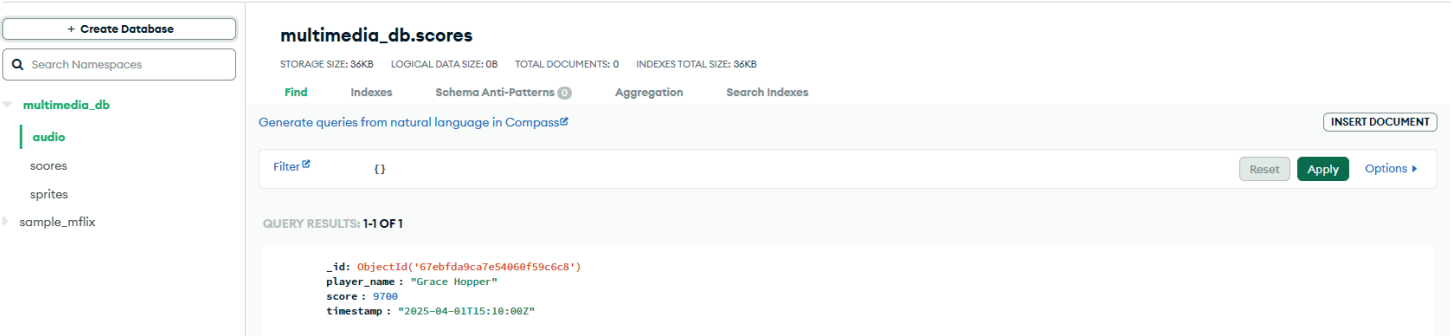


Figure 1: The audio collection within `multimedia_db` database

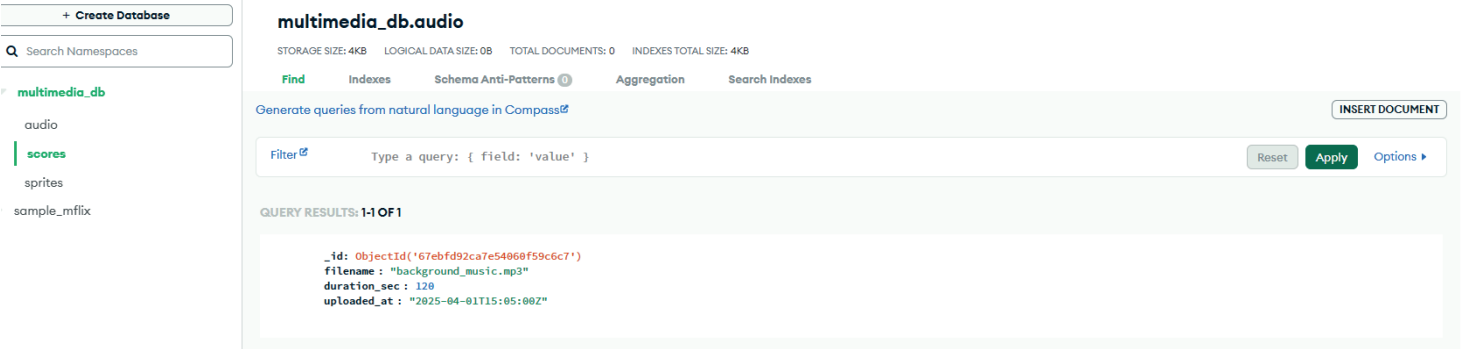


Figure 2: The scores collection within `multimedia_db` database

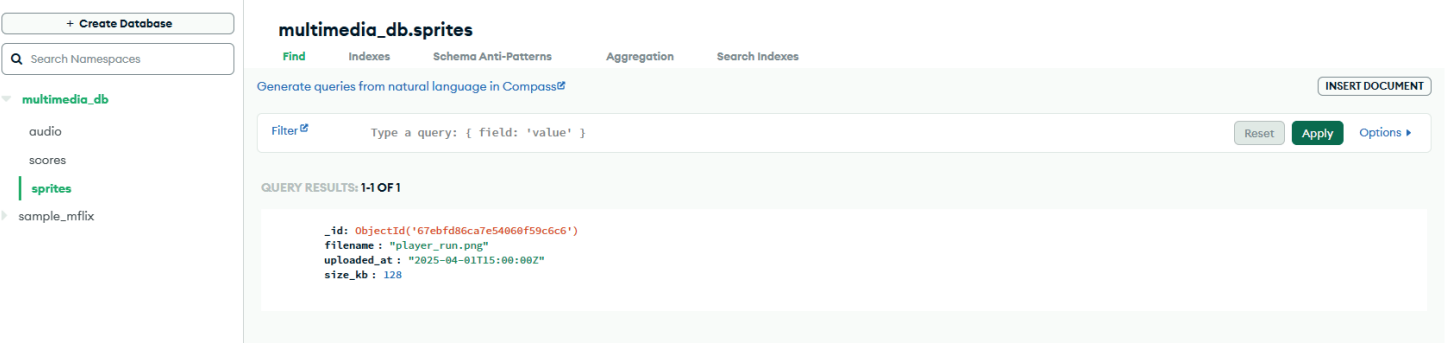


Figure 3: The sprites collection within `multimedia_db` database

Screenshots of Springboot running locally:

The below are screenshots of the Springboot server running locally at <http://localhost:8080/>

```
PS E:\IMP-MCAST\Level_6\Sem3\Database\IMP-home_assignment\database_essentials_github\database_essentials> mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.darrene:database-essentials >-----
[INFO] Building my-project 1.0-SNAPSHOT
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot:3.0.0:run (default-cli) > test-compile @ database-essentials >>>
[INFO]
[INFO] --- resources:3.3.0:resources (default-resources) @ database-essentials ---
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- compiler:3.10.1:compile (default-compile) @ database-essentials ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- resources:3.3.0:testResources (default-testResources) @ database-essentials ---
[INFO] skip non existing resourceDirectory E:\IMP-MCAST\Level_6\Sem3\Database\IMP-home_assignment\database_essentials_github\database_essentials\src\test\resources
[INFO]
[INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ database-essentials ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] <<< spring-boot:3.0.0:run (default-cli) < test-compile @ database-essentials <<<
[INFO]
[INFO] --- spring-boot:3.0.0:run (default-cli) @ database-essentials ---
[INFO] Attaching agents: []
```

Figure 1: Maven performing a clean build and execution of the project, through ``mvn spring-boot@run``

```

  ____  __
 / ___/  /
/  /_  /  ___
/   /  /  /  ___
/___/  /___/

:: Spring Boot ::      (v3.0.0)

2025-04-04T22:39:36.640+02:00 INFO 22476 --- [main] com.darren.backend.Application : Starting Application using Java 23.0.1 with PID 22476 (E:\IMP-MCAST\Level_6\Sem3\Database\IMP-home_assignment\database_essentials_github\database_essentials\target\classes started by darre in E:\IMP-MCAST\Level_6\Sem3\Database\IMP-home_assignment\database_essentials_github\database_essentials)
2025-04-04T22:39:36.643+02:00 INFO 22476 --- [main] com.darren.backend.Application : No active profile set, falling back to 1 default profile: "default"
2025-04-04T22:39:37.042+02:00 INFO 22476 --- [main] s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data Reactive MongoDB repositories in DEFAULT mode.
2025-04-04T22:39:37.054+02:00 INFO 22476 --- [main] s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 8 ms. Found 0 Reactive MongoDB repository interface s.
2025-04-04T22:39:38.066+02:00 INFO 22476 --- [main] o.s.b.w.e.n.NettyWebServer : Netty started on port 8080
2025-04-04T22:39:38.075+02:00 INFO 22476 --- [main] com.darren.backend.Application : Started Application in 1.705 seconds (process running for 1.979)

```

Figure 2: Springboot running locally at `http://localhost:8080/`

Screenshots of the Springboot server running live on Render:

The presented are screenshots of the Springboot server running publically online at <https://dashboard.render.com/>

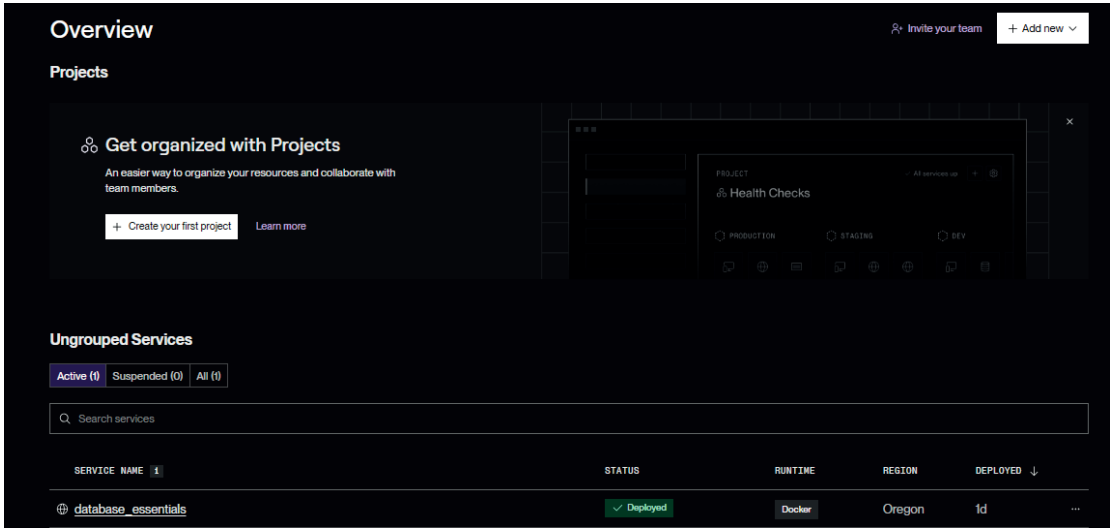


Figure 1: The Springboot server running online at `https://dashboard.render.com/`, through a Docker runtime.



Figure 2: A close up of Figure 1, displaying the Springboot server running publically available online, through a Docker runtime.

Screenshots of the credential setup:

The displayed images present the credential's configuration in the project.

MongoDB:

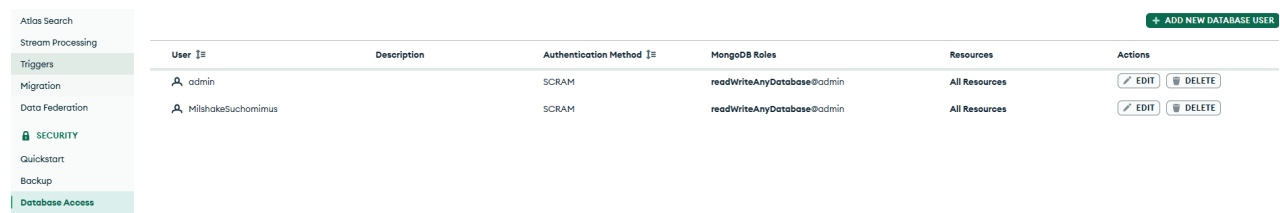


Figure 1: The 'Database Access' tab on Mongo Atlas, showing the users and their permissions.



Figure 2: A close-up of Figure 1, highlighting the user account used in the Maven project to access and manipulate database resources.

The Maven project:

The two images below display the contents of the `application.properties` file within the `resources` folder in the `main` directory. The `application.properties` file can clearly be seen containing both configuration details, and the sensitive username and password of the MongoDB account used for database access and manipulation, in the project.

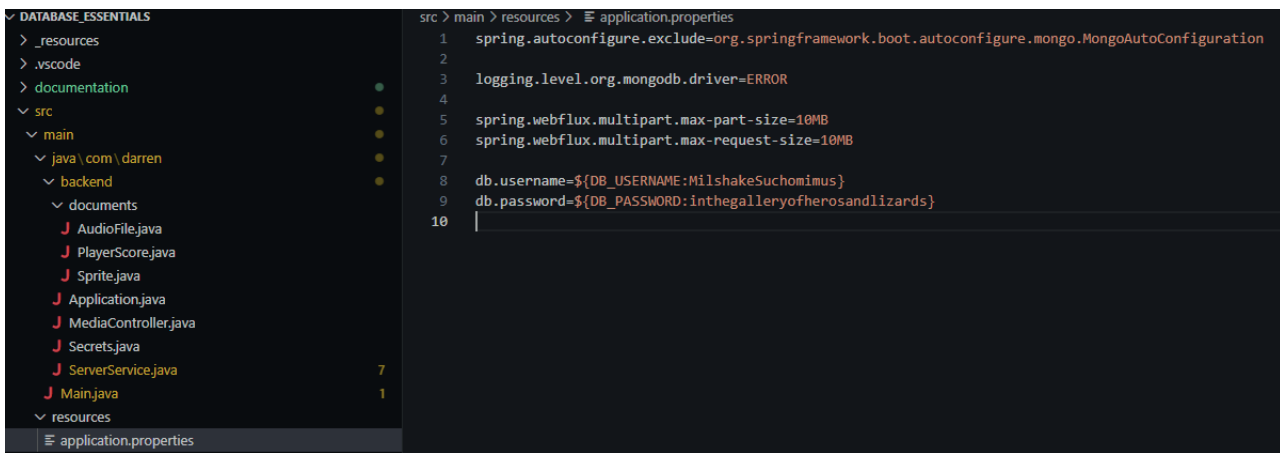


Figure 3: The `application.properties` file in the Maven project, used by Sprinboot to pull both configuration and sensitive information

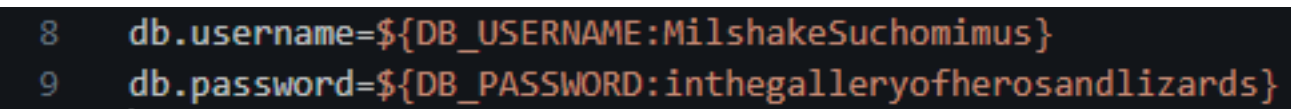


Figure 4: A close up Figure 3, showing the username and password of the MongoDB account used for this project.

The following images show the use of the `Secrets` class being used to get the username and password from the `application.properties` file.

```
public ServerService(Secrets secrets) {  
    String uri = String.format(  
        format: "mongodb+srv://%s:%s@cluster0.znflm.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0",  
        secrets.getDbUsername(), secrets.getDbPassword()  
    );  
  
    client = MongoClient.create(  
        MongoClientSettings.builder()  
        .applyConnectionString(new ConnectionString(uri))  
        .build()  
    );  
  
    db = client.getDatabase("multimedia_db");  
}
```

Figure 5: A new instance of `Secrets`, inserted by Spring boot, to get the username and password set within the `application.properties` file.

```
String uri = String.format(  
    format: "mongodb+srv://%s:%s@cluster0.znflm.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0",  
    secrets.getDbUsername(), secrets.getDbPassword()  
);
```

Figure 6: A close up of Figure 5, showing the code of most interest relating to how the username and password of the environment variables are derived from an instance of `Secrets`.

Screenshot of IP whitelisting on Mongo Atlas:

The below screenshot shows the IP of the live Render server hosting the Spring boot application, whitelisted on Mongo Atlas.

SECURITY	44.227.217.144/32	Active	EDIT	DELETE
Quickstart	212.56.139.2/32	Active	EDIT	DELETE
Backup				
Database Access				
Network Access				

Figure 1: The `Network Access` tab, showing a list of whitelisted IPs, among is the Render server's public IP, that is hosting the Spring boot application.

How the IP was retrieved:

Figure 2 presents the snippet of code that functioned as the endpoint to retrieve the IP of the live Render server, whilst Figure 3 shows the result of a query sent via a REST GET request, to said endpoint .

```
@GetMapping("/my-ip")  
public Mono<String> getServerIp() {  
    return Mono.fromCallable(() -> {  
        HttpClient client = HttpClient.newHttpClient();  
  
        HttpRequest request = HttpRequest.newBuilder()  
            .uri(URI.create("https://api.ipify.org"))  
            .GET()  
            .build();  
  
        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());  
        return response.body();  
    });  
}
```

Figure 2: The Java method within the `MediaController` class, that functions as a GET endpoint to retrieve the IP of the server.

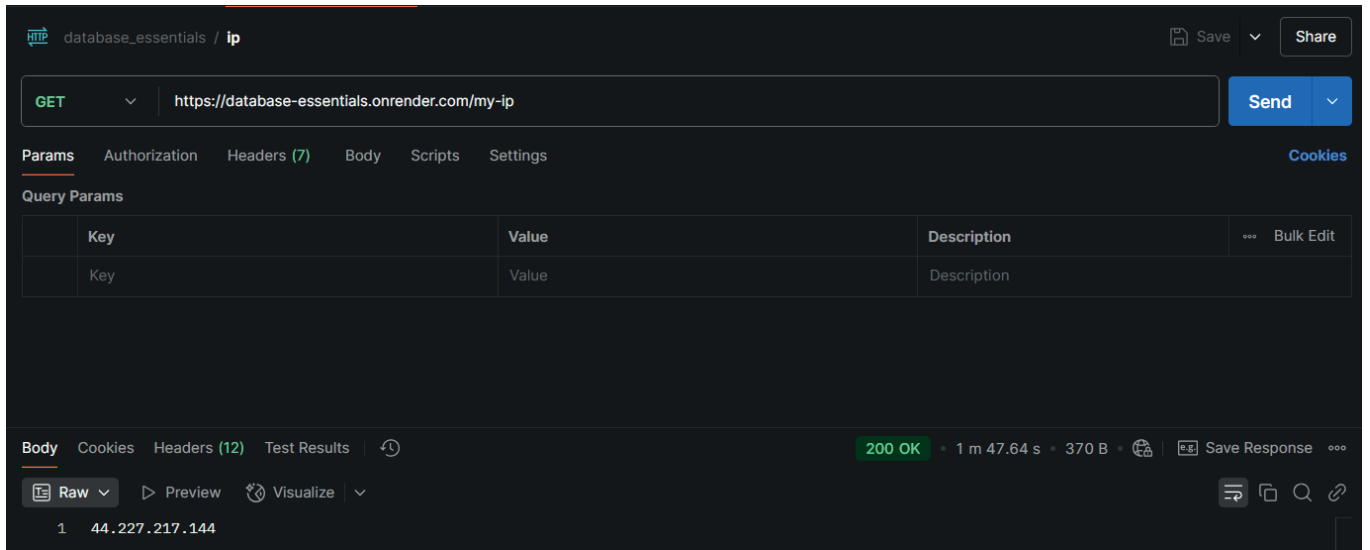


Figure 3: The result of GET request sent to `my-ip`, clearly showing the whitelisted returned IP of the live Render server.

Screenshots of the credential setup:

The following images present the process of uploading a sprite using the `./upload_sprite` endpoint, defined within the `MediaController` Java class.

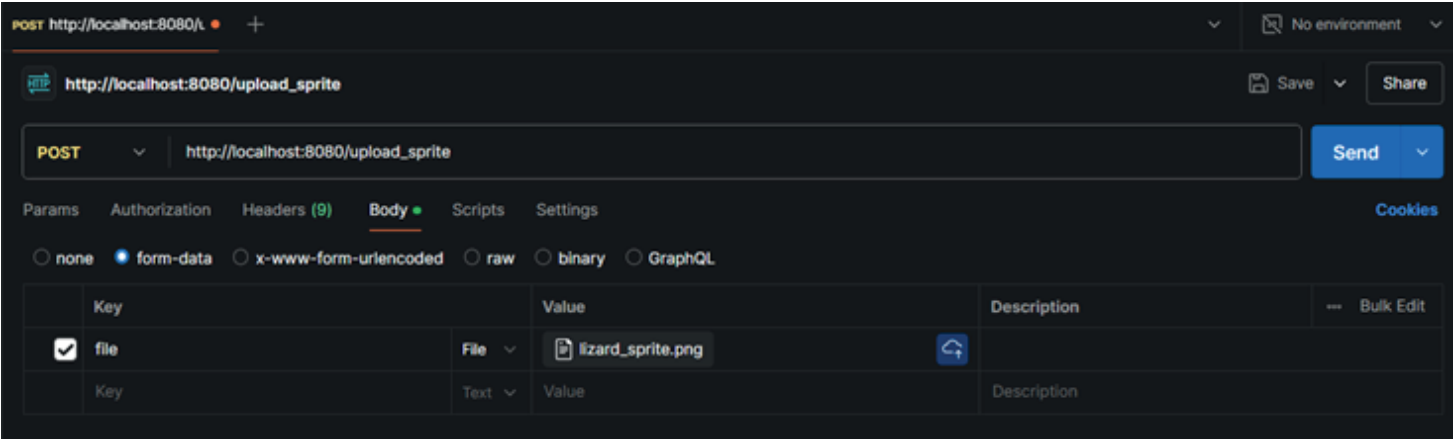


Figure 1: The settings used on Postman, to upload the sprite titled `lizard_sprite.png`, to the live Render-hosted Spring boot server.

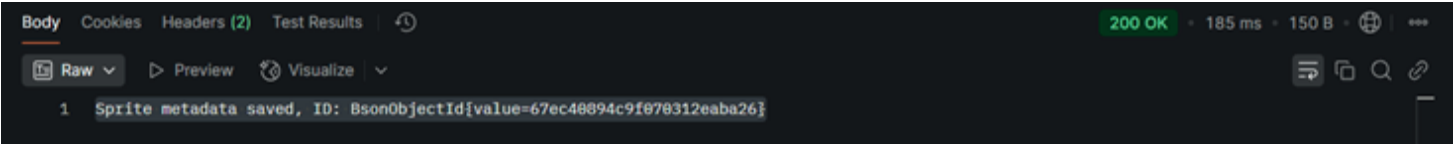


Figure 2: The response result, clearly indicating that the operation was a success.



Figure 3: The lizard sprite stored on the MongoDB database. The `content` of the image can evidently be seen as stored as a base 64 String.

Testing the retrieved data:

The following images display the testing of sprite storage, by first using `/sprites` endpoint to get all the sprites in the database, then using a third-party web application to convert that base-64-encoded, back to an image.

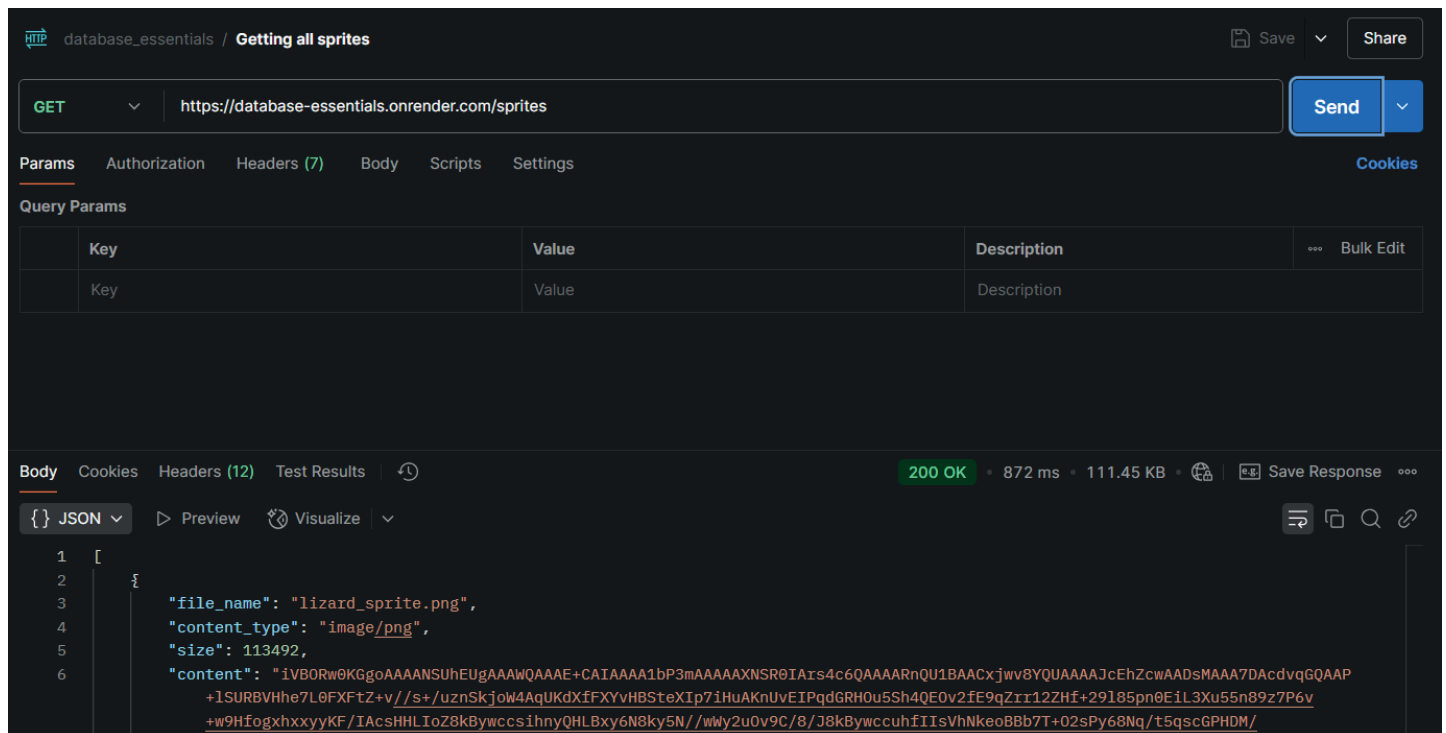


Figure 1: The `/sprites` endpoint used in Postman, to get the sprites from the database. In this case, only one sprite entry is returned.

The below image show the base-64-encoded string read from the database above, being re-parsed into an image. This proves that images are being stored into the database by the server accordingly.

Base64 to Image

Convert Base64 to image online using a free decoding tool which allows you to decode Base64 as image and preview it directly in the browser. In addition, you will receive some basic information about this image (resolution, MIME type, extension, size). And, of course, you will have a special link to download the image to your device. If you are looking for the reverse process, check [Image to Base64](#).



Figure 2: The base 64 string converted back into an image, using `Base 64 Guru` at <https://base64.guru/converter/decode/image>

Data sanitisation:

This part of the documentation centres around collecting evidence of data validation performed at each method defined within the `MediaController` class.

File name validation:

This section is dedicated to presenting the `isValidFileName` method within `MediaController`. This method is used for the uploading of sprites and audio files, including getting individual ones by name. This helper method checks if the passed value is not `null` and conforms to a regular expression that defines a valid file name.

```
/**
 * Utility method to validate filenames.
 * Accepts alphanumeric characters, dots, underscores, and hyphens only.
 *
 * @param filename The filename string to check.
 * @return True if valid, false otherwise.
 */
private boolean isValidFilename(String filename) {
    return filename != null && filename.matches(regex:"^[a-zA-Z0-9._-]{1,100}$");
}
```

Figure 1: The definition of `isValidFileName`, a private method dedicated to validating a singular String and returning a boolean primitive to indicate its validity as a file name.

The below method represents the endpoint used to get an audio file based on a name following the endpoint, such as `http://localhost:8080/audio/nameOfFile.mp3`. The passed String is passed to `isValidFileName`, to ensure it's a valid file name.

```
/**
 * Retrieves an audio file by filename after validation.
 *
 * @param filename The name of the audio file.
 * @return The matching audio file or a not found response.
 */
@GetMapping("/audio/{filename}")
public Mono<ResponseEntity<AudioFile>> getAudioByFilename(@PathVariable String filename) {
    // Check filename validity before querying
    if (!isValidFilename(filename)) {
        return Mono.just(ResponseEntity.badRequest().build());
    }

    return serverService.getAudioByFilename(filename)
        .map(ResponseEntity::ok)
        .defaultIfEmpty(ResponseEntity.notFound().build());
}
```

Figure 2: The use of `isValidFileName` to check the validity of the passed audio file name, in the `getAudioByFilename` method.

Identically to the endpoint formerly show that returns an audio file given a file name, the following endpoint returns an sprite matching the passed String. The String is also checked to conform to a valid file name.

```
/**
 * Retrieves a sprite by filename after validation.
 *
 * @param filename The name of the sprite file.
 * @return The matching sprite or a not found response.
 */
@GetMapping("/sprite/{filename}")
public Mono<ResponseEntity<Sprite>> getSpriteByFilename(@PathVariable String filename) {
    // Check filename validity before querying
    if (!isValidFilename(filename)) {
        return Mono.just(ResponseEntity.badRequest().build());
    }

    return serverService.getSpriteByFilename(filename)
        .map(ResponseEntity::ok)
        .defaultIfEmpty(ResponseEntity.notFound().build());
}
```

Figure 3: The definition of the `/sprite/{filename}`, endpoint in the `getSpriteByFileName` method.

Figure 4 shows the use of `isValidFileName`, being used for the uploading of sprites and audio files respectively. Since the only argument is of type `FilePart`, its `fileName()` method will be used to pass the implicitly-set file name, to the `isValidFileName` method.

```
/**
 * Uploads sprite metadata and stores it in the database.
 *
 * @param file The uploaded sprite file.
 * @return A response with confirmation or error.
 */
@PostMapping("/upload_sprite")
public Mono<ResponseEntity<String>> uploadSprite(@RequestPart("file") FilePart file) {
    // Validate filename format
    if (!isValidFilename(file.fileName())) {
        return Mono.just(ResponseEntity.badRequest().body("Invalid filename"));
    }

    return serverService.saveSpriteMetadata(file)
        .map(id -> ResponseEntity.ok("Sprite metadata saved, ID: " + id));
}

/**
 * Uploads audio file metadata and stores it in the database.
 *
 * @param file The uploaded audio file.
 * @return A response with confirmation or error.
 */
@PostMapping("/upload_audio")
public Mono<ResponseEntity<String>> uploadAudio(@RequestPart("file") FilePart file) {
    // Validate filename format
    if (!isValidFilename(file.fileName())) {
        return Mono.just(ResponseEntity.badRequest().body("Invalid filename"));
    }

    return serverService.saveAudioMetadata(file)
        .map(id -> ResponseEntity.ok("Audio metadata saved, ID: " + id));
}
```