Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação
Mestrado Acadêmico em Sistemas e Computação

Comparison of Execution Time of Unit and Integration Tests for a Medical Appointments Management API

Lázaro Lawrence Pereira Damasceno

Natal-RN
May 20, 2025

Acronyms

API Application Programming Interface

BSON Binary JavaScript Object Notation

DBMS Database Management System

DTO Data Transfer Object

ERD Entity-Relationship Diagram

IANA Internet Assigned Numbers Authority

JVM Java Virtual Machine

JS JavaScript

JSON JavaScript Object Notation

MVC Model-View-Controller

NoSQL Not Only Structured Query Language

RDBMS Relational Database Management System

RFC Request for Comments

SDLC Software Development Life Cycle

STLC Software Testing Life Cycle

SIN Social Identification Number

SQL Structured Query Language

UI User Interface

UML Unified Modeling Language

URI Uniform Resource Identifier

HATEOAS Hypermedia As the Engine Of Application State

HTTP Hypertext Transfer Protocol

XML Extensible Markup Language

List of Tables

List of Figures

Contents

1 Introduction

2 Contextual Foundation

This chapter establishes the foundational knowledge required for a comprehensive exploration of automated software testing in the context of an API. The section is divided into four subsections:

1. Database: Examines the strengths and limitations of NoSQL databases, with a focus on MongoDB as a document store. Key topics include scalability, schema flexibility, and the advantages of JSON/BSON for integration testing.

2. HTTP Semantics: Explores the principles of HTTP as a stateless protocol, covering request-response models, status codes, methods, and their relevance to API testing.

3. Application Programming Interface: Introduces the design and architecture of the Medical Appointments Management API, including UML class diagrams, modular monolith structure, and domain functionalities.

4. Automated Software Testing: A critical component of software quality, consuming significant project time. It identifies issues, ensures system functionality, and meets user needs. Good tests cover diverse scenarios, prevent future problems, and are supported by tools for efficient execution. Types include manual and automated testing (e.g., unit and integration tests).

The discussion bridges theoretical concepts with practical applications, emphasizing how these fundamentals underpin the development and testing of the API.

2.1 Database

Databases are an organized collection of data that enables the handling of large-scale datasets by inputting, storing, retrieving, and managing them. A database management system is defined as software that provides the interface between users and a database [**?**].

The DBMS has the responsibility of maintaining the integrity and security of the stored data. Furthermore, databases can be categorized into different criteria: i.e., when a database relies on tables to organize data, it is labeled as a *relational*. Otherwise, when the database does not rely on tables, it is labeled as *non-relational*. Besides, it is worth highlighting that non-relational databases are also known as *NoSQL databases* [**?**, **?**].

For instance, DBMS relies on structured data organization using tables, schemata, columns, rows, primary keys, and foreign keys. Foreign keys are essential for establishing one-to-one, one-to-many, and many-to-many relationships between entities [**?**].

The rigid data structure of DBMS often encounters scalability challenges with increasing data volumes and transaction loads, as included in the Table **??**:

Table 2.1: Relational Database Scalability Challenges

Technical Challenges	**Structural Limitations**
Difficulty in predicting performance bottlenecks	Rigid table format complicates complex data modeling
Complex data integrity in shared environments	SQL joins become ineffective with growing datasets
Dual design needs (operational vs analytical)	

Source: Adapated from [?, ?].

These constraints underscore why NoSQL databases excel in handling large datasets where relational models falter, particularly for handling large datasets where a rigid relational model can impede performance [?].

Consequently, NoSQL databases are crucial for Big Data applications, enabling efficient storage and retrieval of massive datasets. Unlike DBMS, which complies with ACID (Atomicity, Consistency, Isolation, and Durability) properties, NoSQL databases typically prioritize BASE (Basically Available, Soft State, and Eventual Consistency) principles [?].

BASE derives from the CAP (Consistency, Availability, Partition-tolerance) theorem. Due to the partial use of the theorem by selecting two of the three principles from the theorem, many NoSQL databases have relaxed the requirements to match their own needs on the consistency principle in order to achieve better availability and partitioning, which resulted in the creation of BASE [?].

Scalability in distributed systems is closely tied to the CAP theorem and the BASE model. By prioritizing availability and partition tolerance over strict consistency, as facilitated by BASE principles, NoSQL databases enhance their ability to scale. This trade-off allows for efficient data distribution and workload management across multiple nodes, crucial for handling the demands of modern, large-scale applications [?].

Turning to scalability, it refers to an architecture's capacity to handle a growing number of components and interactions. This capacity is influenced by factors such as interaction frequency, load distribution (uniform vs. peak), delivery guarantees (assured vs. best-effort), request handling (synchronous vs. asynchronous), and environmental control (controlled vs. anarchic) [?].

A key advantage of NoSQL's scalability is its ability to scale both vertically and horizontally. As another advantage, NoSQL databases are more suitable to store data across a cloud of servers. [?, ?] states that vertical scalability is done by adding resources to a single server. Horizontal scalability is achieved by distributing data across multiple servers. The dual scalability overcomes the vertical scalability limitations of traditional SQL databases.

In contrast with NoSQL's scalability, traditional DBMS scalability often involves expensive vertical scaling. The traditional RDBMS scalability is done by adding memory and CPU to servers. The adoption of NoSQL databases was largely driven by the high costs associated with scaling DBMS for large-scale data processing [**?**, **?**].

As a consequence of the unaffordable costs associated with scaling DBMS for large-scale data processing, the cost-effectiveness and performance gains of NoSQL databases are primarily attributed to their distributed architecture, which utilizes inexpensive commodity servers [**?**].

Building on these principles, NoSQL databases are categorized into the following types, as mentioned in the Table **??**:

Table 2.2: NoSQL Database Types

Primary Categories	**Alternative Classifications**
Key-value stores	Key-value pairs
Document stores	Document store
Column-family stores	Wide-column store
Graph databases	
Column-oriented databases	Column-based databases
Object-oriented databases	
Grid and cloud databases	
XML databases	
Multidimensional databases	
Multi-value databases	
Multi-model databases	

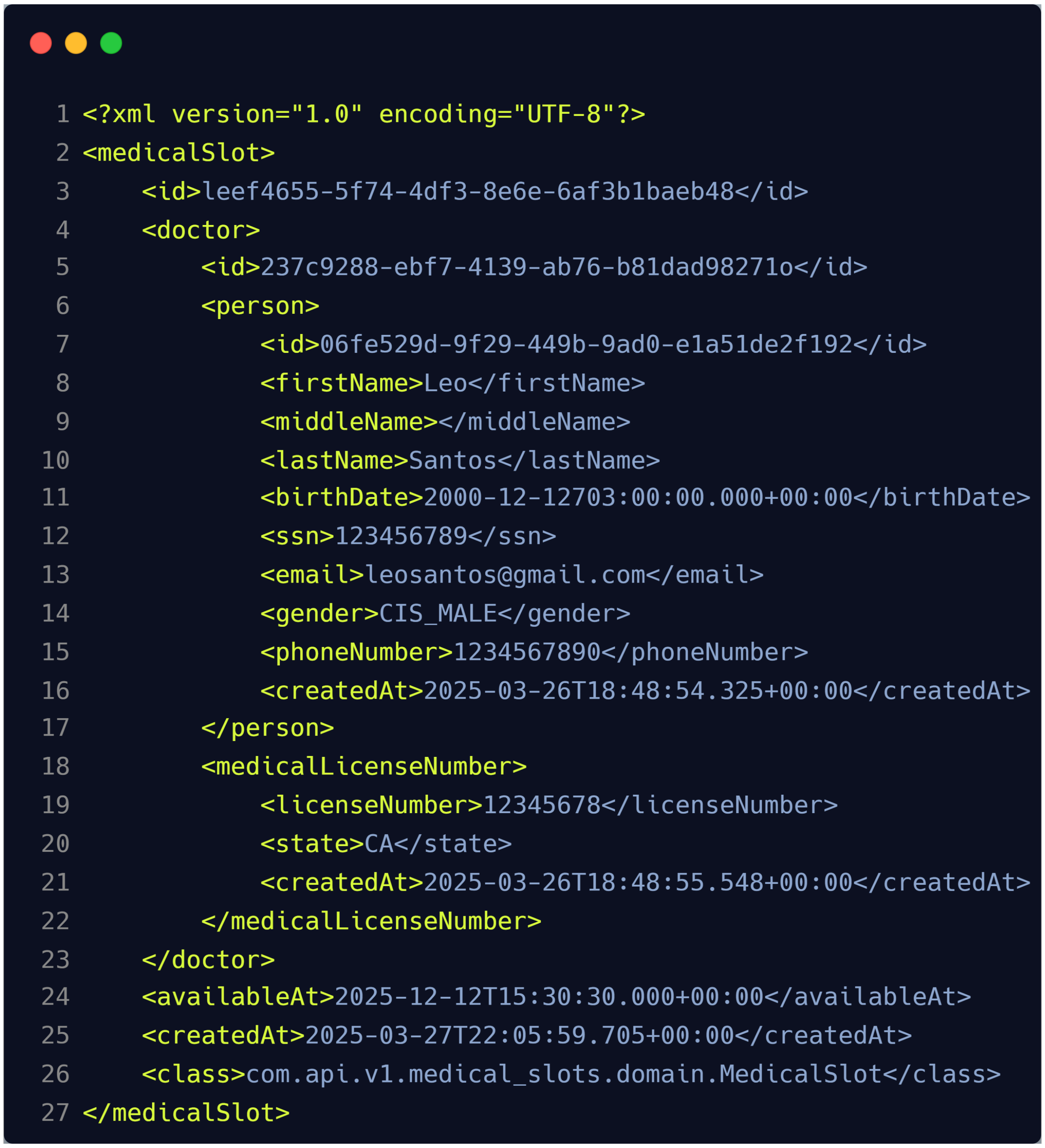
Source: Adapated from [**?**, **?**, **?**, **?**, **?**].

Among these categories, key-value stores—the simplest NoSQL type—universally support insertion, deletion, and lookup operations while enabling scalability through distributed key management. Notably, document stores extend this model by encoding values as semi-structured documents (e.g., JSON or BSON), offering schema flexibility that allows runtime modifications [**?**, **?**, **?**].

For instance, MongoDB is a prominent example of a document store NoSQL database. Document stores, like MongoDB, organize data in a hierarchical structure similar to XML documents [**?**, **?**].

The figures **??** and **??** contain, respectively, examples of an XML and a MongoDB document.

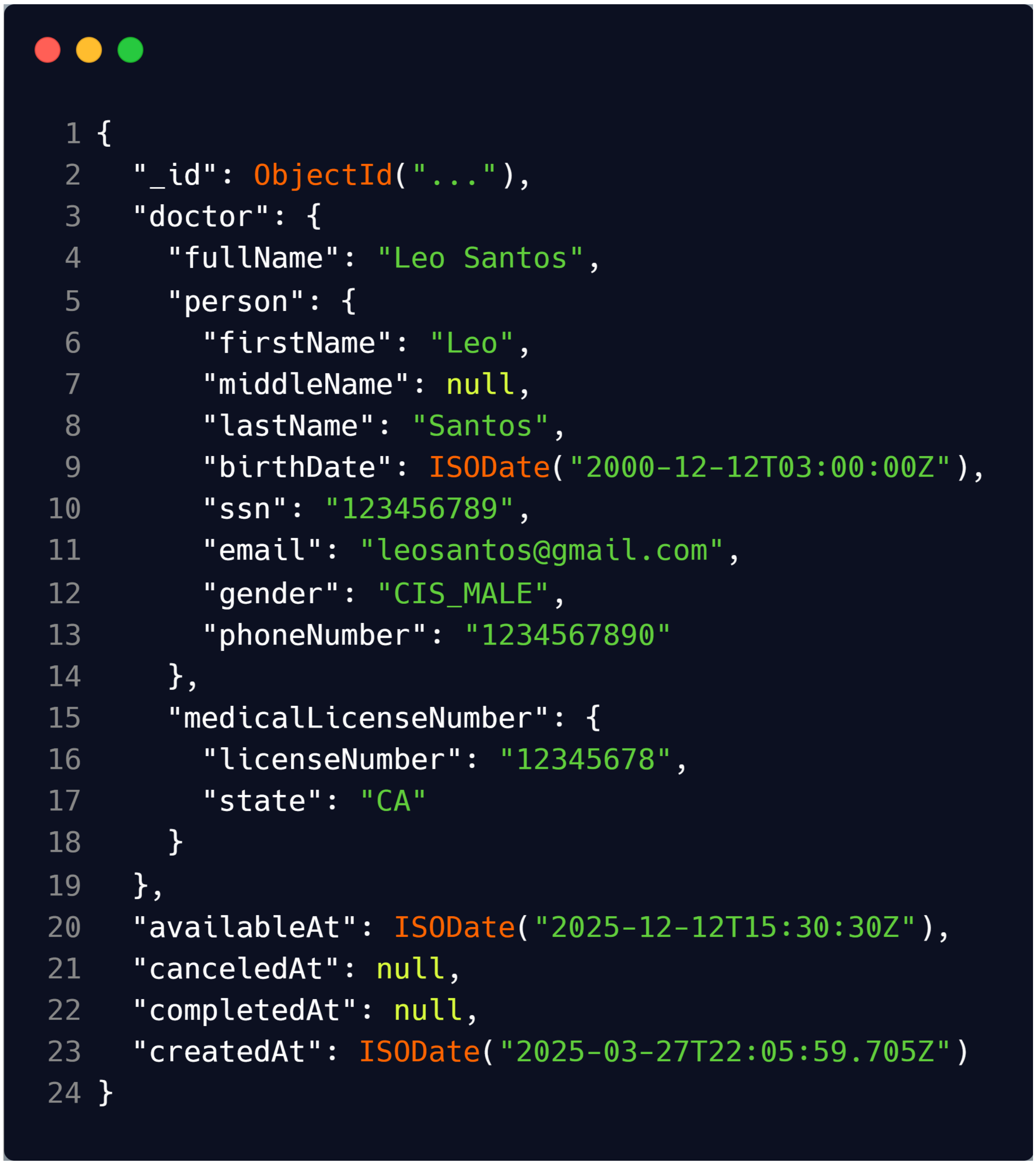
Figure 2.1: Example of XML Document



```xml
<?xml version="1.0" encoding="UTF-8"?>
<medicalSlot>
    <id>1eef4655-5f74-4df3-8e6e-6af3b1baeb48</id>
    <doctor>
        <id>237c9288-ebf7-4139-ab76-b81dad98271o</id>
        <person>
            <id>06fe529d-9f29-449b-9ad0-e1a51de2f192</id>
            <firstName>Leo</firstName>
            <middleName></middleName>
            <lastName>Santos</lastName>
            <birthDate>2000-12-12T03:00:00.000+00:00</birthDate>
            <ssn>123456789</ssn>
            <email>leosantos@gmail.com</email>
            <gender>CIS_MALE</gender>
            <phoneNumber>1234567890</phoneNumber>
            <createdAt>2025-03-26T18:48:54.325+00:00</createdAt>
        </person>
        <medicalLicenseNumber>
            <licenseNumber>12345678</licenseNumber>
            <state>CA</state>
            <createdAt>2025-03-26T18:48:55.548+00:00</createdAt>
        </medicalLicenseNumber>
    </doctor>
    <availableAt>2025-12-12T15:30:30.000+00:00</availableAt>
    <createdAt>2025-03-27T22:05:59.705+00:00</createdAt>
    <class>com.api.v1.medical_slots.domain.MedicalSlot</class>
</medicalSlot>
```

Source: Author's creation.

Figure 2.2: Example of MongoDB Document



```
{
  "_id": ObjectId("..."),
  "doctor": {
    "fullName": "Leo Santos",
    "person": {
      "firstName": "Leo",
      "middleName": null,
      "lastName": "Santos",
      "birthDate": ISODate("2000-12-12T03:00:00Z"),
      "ssn": "123456789",
      "email": "leosantos@gmail.com",
      "gender": "CIS_MALE",
      "phoneNumber": "1234567890"
    },
    "medicalLicenseNumber": {
      "licenseNumber": "12345678",
      "state": "CA"
    }
  },
  "availableAt": ISODate("2025-12-12T15:30:30Z"),
  "canceledAt": null,
  "completedAt": null,
  "createdAt": ISODate("2025-03-27T22:05:59.705Z")
}
```

Source: Author's creation.

It is noticeable the key-value structure in the shape of tags in figure **??**. Such structure resembles the way the data is organized in figure **??**. Furthermore, there's a notable hierarchical structure in both figures. As illustrated in figure **??**, the embedded *doctor* object that contains the embedded *person* and *medicalLicenseNumber* objects within a single document demonstrates MongoDB's schema flexibility, contrasting with the relational model's reliance on foreign key setting relationships. Notably, MongoDB has some key strengths and weaknesses described in the Table **??**:

Table 2.3: MongoDB Strengths and Weaknesses

Strengths	**Weaknesses**
Sharding capability	No native join operations
High speed performance	Absence of transactions
Efficient storage and distribution of large binary files	Historically immature platform
Fault tolerance policy	No standardized query language
Complex query language	Complex query language may be difficult for users
Open-source nature	Consistency compromise may cause issues for sensitive transactions
Better performance and scalability when consistency is compromised	
No requirement for a predefined model or structure for data storage	

Source: Adapated from [**?**, **?**, **?**, **?**, **?**, **?**].

Additionally to what was mentioned in the Table **??**, document store databases offer schema flexibility, allowing documents within a collection to have varying local schemata. While this flexibility supports agile development, it requires careful data management to prevent inconsistencies due to the absence of enforced schema constraints [**?**]. JSON is a lightweight data-interchange format whose strengths are described in the Table **??**:

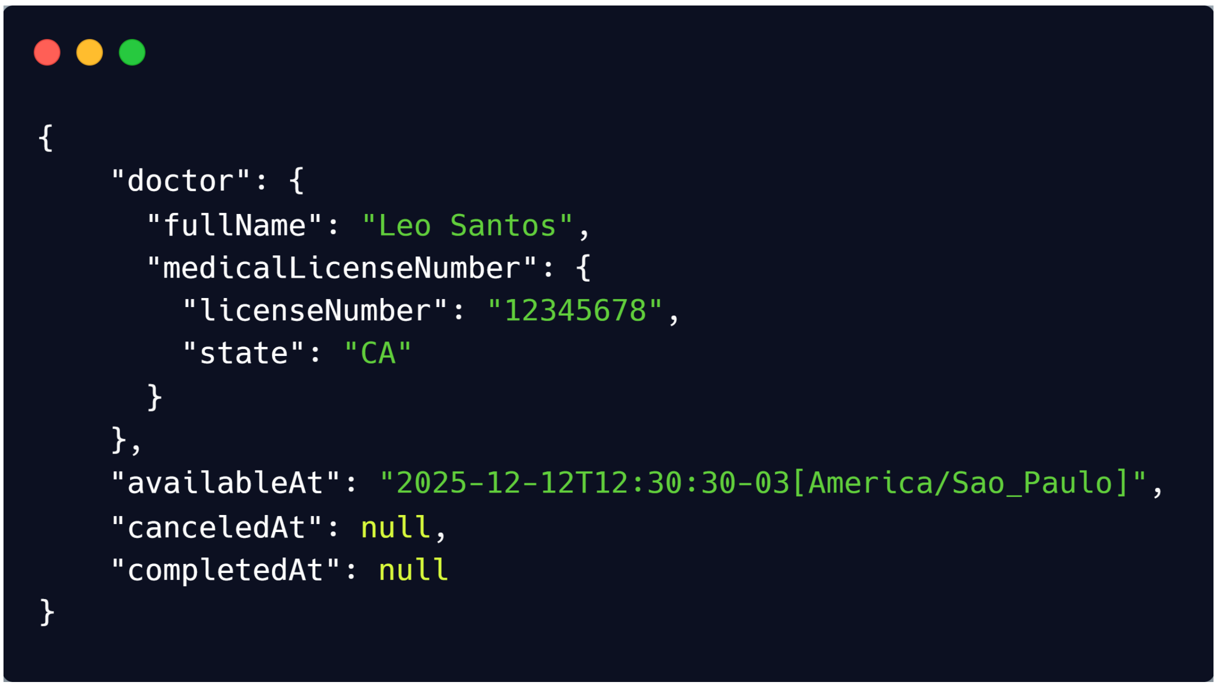
Table 2.4: JSON Characteristics

Advantages	**Features**
Straightforward readability for humans and machines	Uses familiar conventions of C-languages
Feasible to write and generate	Compatible with C-inspired languages (Java, JavaScript, etc.)
Text format independent of any programming language	Straightforward format of name-value pairs
Simple data interchange format	Supports ordered lists of values

Source: Adapated from [**?**].

The figure **??** provides an example of a JSON document that illustrates the straightforward readability, familiar C-language conventions, and straightforward name-value pair format.

Figure 2.3: Example of JSON Document



```
{
    "doctor": {
      "fullName": "Leo Santos",
      "medicalLicenseNumber": {
        "licenseNumber": "12345678",
        "state": "CA"
      }
    },
    "availableAt": "2025-12-12T12:30:30-03[America/Sao_Paulo]",
    "canceledAt": null,
    "completedAt": null
}
```

Source: Author's creation

As displayed in the figure **??**, the structure of the JSON is unambiguous. The use of curly and square brackets to delimit where the JSON commences and finishes is a clear manner of organization. [**?**] highlights that the format is used by MongoDB to store collections of JSON objects in lieu of relational tables, and JSON is used as the primary data format to interact with applications.

For instance, even though there is a resemblance between JSON and XML, and even their shared hierarchical semi-structured data model, JSON is indeed the replacement of XML due to JSON's simplicity, compactness, and the use of JS as the mapping language used to map data [**?**].

As a consequence of a good product, BSON was created to suit the need of traversability of MongoDB in its primary data representation. Similar to JSON, BSON is lightweight, efficient, a binary-encoded serialization of JSON-like documents that supports embedding documents and arrays. Due to MongoDB's adoption of an extended key-value store, BSON holds data in a binary format in which zero or more ordered key/value pairs are stored as a single document [**?**, **?**, **?**, **?**].

One example of the mentioned key-value structure is shown in the figure **??**.

Considering all the benefits of NoSQL and MongoDB, therefore in the context of automated software testing, MongoDB's flexible schema and document-oriented structure offer significant advantages for integration testing. The ability to easily insert and persist embedded documents in BSON format directly reflects the successful execution of integration tests.

2.2 HTTP Semantics

HTTP, a stateless application-layer protocol, uses a request-response model and self-descriptive messages for flexible hypermedia communication. It provides a uniform interface, abstracting resource access, and serves as the primary Web communication protocol for resource representation transfer [?, ?].

Within this framework, communication is structured around messages, which are categorized as either requests or responses. A client initiates interaction by constructing request messages that articulate its intentions and are directed to a designated server [?].

The server performs the following steps mentioned in the list below, as evidenced by [?]:

1. Listens for requests;
2. Parses each received message;
3. Interprets message semantics relative to the target resource;
4. Responds with one or more response messages.

Once these steps are completed, the client evaluates the response's status code to verify successful execution and determines subsequent actions, as defined by HTTP semantics [?].

Moreover, in order to identify resources across HTTP, URI references are used to target requests, indicate redirects, and define relationships. Despite any regulation issued by IANA[1], the HTTP server has two inherent schemes, as demonstrated by the Table ??:

Table 2.5: HTTP server schemes

Scheme	**Full Name**	**Example**
http	Hypertext Transfer Protocol	”http” ”://” authority path-abempty [”?” query]
https	Hypertext Transfer Protocol Secure	”https” ”://” authority path-abempty [”?” query]

Source: Adapated from [?].

In relation to the http schemes, the resource relies on the scheme to route its message to the appropriate server and port. The schemes allow the request to be listened to by the correct server and return the actual result to the client.

Turning to the status codes, as stated by [?], the status code of a response is a three-digit integer code within 100 to 599 that describes the result of the request and the semantics of the response, as well as the success or failure of the request and what content is enclosed. In order to facilitate the comprehension of the status codes, only their first digit categorizes the response's class [?].

Thus, the status codes are divided into 5 categories, as shown below in the Table ??:

[1] See about IANA.

Table 2.6: HTTP Status Codes

Status code	**Description**
1xx	The request was received, continuing process
2xx	The request was successfully received, understood, and accepted
3xx	Further action needs to be taken in order to complete the request
4xx	The request contains bad syntax or cannot be fulfilled
5xx	The server failed to fulfill an apparently valid request

Source: Adapated from [**?**].

A more granular detailing of all status codes is described in the Appendix **??**.

Furthermore, HTTP uses standardized request methods, independent of specific resources, for uniform network interactions.While the method's meaning remains consistent, resource implementation varies. The Table **??** explains what each HTTP method does.

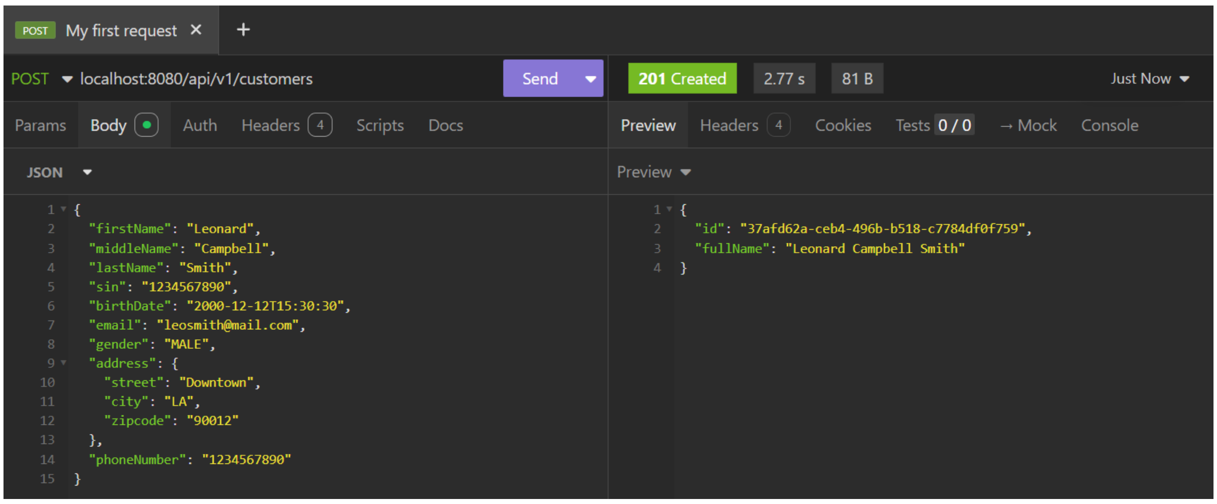
Table 2.7: HTTP Methods

Method Name	**Description**
GET	Transfer a current representation of the target resource.
HEAD	Same as GET, but do not transfer the response content.
POST	Perform resource-specific processing on the request content.
PUT	Replace all current representations of the target resource with the request content.
DELETE	Remove all current representations of the target resource.
CONNECT	Establish a tunnel to the server identified by the target resource.
OPTIONS	Describe the communication options for the target resource.
TRACE	Perform a message loop-back test along the path to the target resource.

Source: Adapated from [**?**].

In order to complement the table above and the function of the status codes (Table **??**) of displaying the response's status to the client, the HTTP methods indicate the action expected from the execution of the request made by the client. Thus, below is an example of a resource that requests the registration of a doctor in the figure **??** and the service class that supports it in the figure **??**.

Figure 2.4: Example of a HTTP request and response

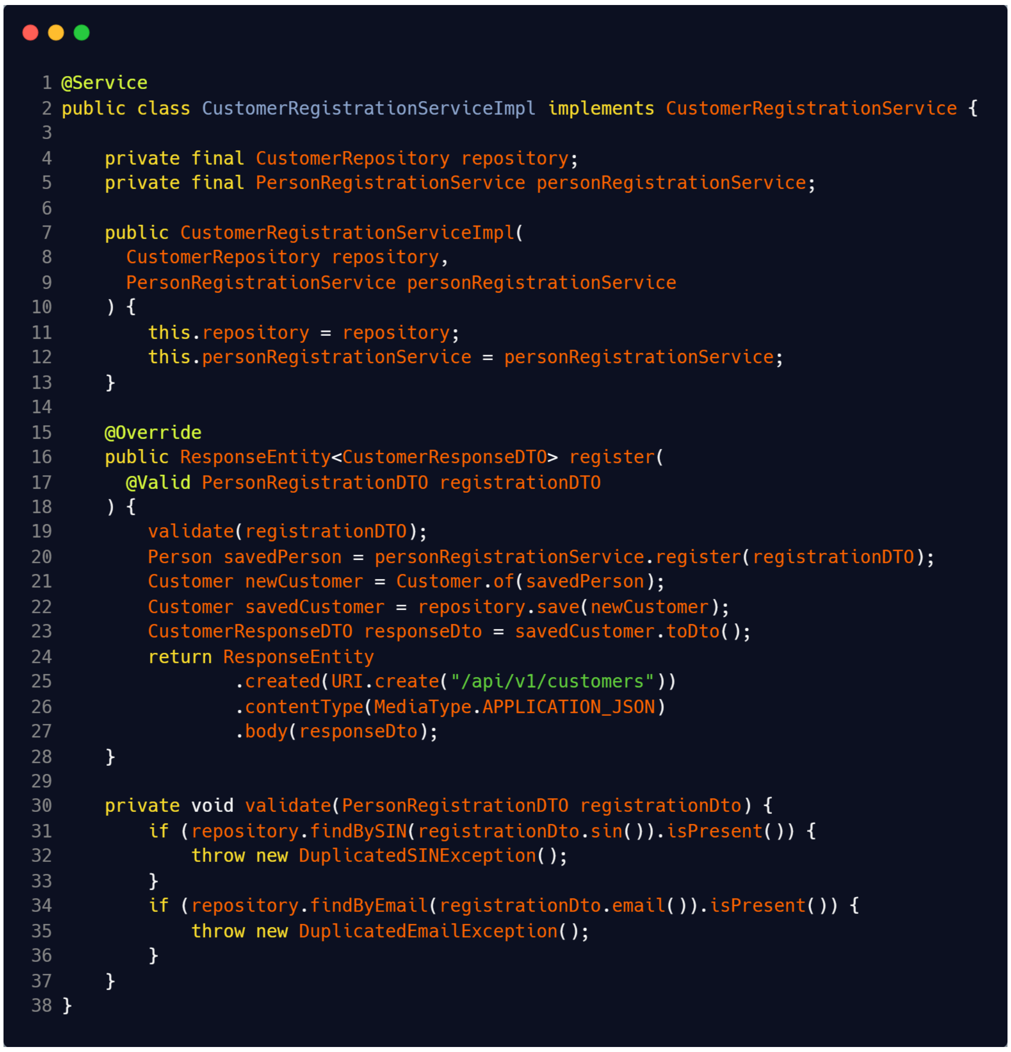

POST My first request
POST localhost:8080/api/v1/customers
Send
201 Created
2.77 s
81 B
Just Now
Params Body Auth Headers 4 Scripts Docs
Preview Headers 4 Cookies Tests 0 / 0 → Mock Console
JSON
Preview

```
{
  "firstName": "Leonard",
  "middleName": "Campbell",
  "lastName": "Smith",
  "sin": "1234567890",
  "birthDate": "2000-12-12T15:30:30",
  "email": "leosmith@mail.com",
  "gender": "MALE",
  "address": {
    "street": "Downtown",
    "city": "LA",
    "zipcode": "90012"
  },
  "phoneNumber": "1234567890"
}
```

```
{
  "id": "37afd62a-ceb4-496b-b518-c7784df0f759",
  "fullName": "Leonard Campbell Smith"
}
```

Source: Author’s creation

Figure 2.5: Example of a Registration Service



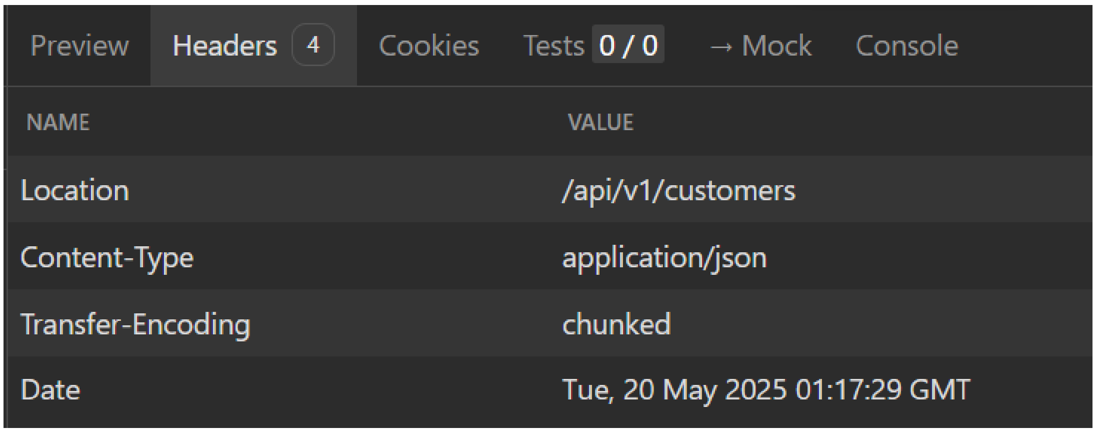
```java
@Service
public class CustomerRegistrationServiceImpl implements CustomerRegistrationService {

    private final CustomerRepository repository;
    private final PersonRegistrationService personRegistrationService;

    public CustomerRegistrationServiceImpl(
      CustomerRepository repository,
      PersonRegistrationService personRegistrationService
    ) {
        this.repository = repository;
        this.personRegistrationService = personRegistrationService;
    }

    @Override
    public ResponseEntity<CustomerResponseDTO> register(
      @Valid PersonRegistrationDTO registrationDTO
    ) {
        validate(registrationDTO);
        Person savedPerson = personRegistrationService.register(registrationDTO);
        Customer newCustomer = Customer.of(savedPerson);
        Customer savedCustomer = repository.save(newCustomer);
        CustomerResponseDTO responseDto = savedCustomer.toDto();
        return ResponseEntity
                .created(URI.create("/api/v1/customers"))
                .contentType(MediaType.APPLICATION_JSON)
                .body(responseDto);
    }

    private void validate(PersonRegistrationDTO registrationDto) {
        if (repository.findBySIN(registrationDto.sin()).isPresent()) {
            throw new DuplicatedSINException();
        }
        if (repository.findByEmail(registrationDto.email()).isPresent()) {
            throw new DuplicatedEmailException();
        }
    }
}
```

Source: Author’s creation.

As exposed by the figure **??**, the POST method was used alongside the URI that was mentioned in the given figure. It follows the structure presented by the Table **??**, especially the **http**'s scheme. The service's code shown by the figure **??** demonstrates the resource returns the status code 201.

The figure **??** illustrates the header cited in the subsection 9.3.3 of the RFC 9110.

Figure 2.6: Example of a Header



NAME	VALUE
Location	/api/v1/customers
Content-Type	application/json
Transfer-Encoding	chunked
Date	Tue, 20 May 2025 01:17:29 GMT

Source: Author's creation.

Moreover, the content in the figure **??**, there is a header location that can be used as an identifier for a specific resource corresponding to the representation in this message's content, as determined by [**?**]. Therefore, this decision is aligned with RFC 9110, especially in the subsection 9.3.3, as determined by [**?**], which emphasizes that the POST method is supposed to work with the status code 201 when the operation is successful.

Finally, in relation to automated software testing, the integration tests are going to behave as stated in the Table **??**:

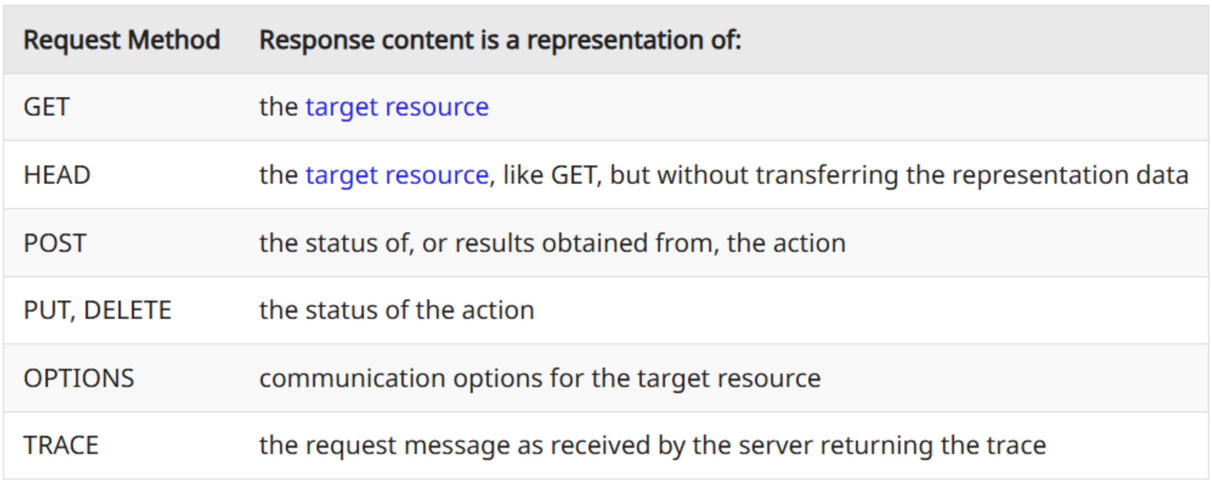
Table 2.8: HTTP Method Status Code Outcomes

HTTP Method	**HTTP Status Codes When Successful**	**HTTP Status Codes When Failure**
GET	200	400 or 404
POST	201	400 or 409
PUT	200 or 204	400 or 404
PATCH	200 or 204	400 or 404
DELETE	204	404

Source: Adapated from [**?**].

In the case of an integration test whose methods are PUT or PATCH, preference shall be given to the status code 204; unless HATEOAS is inserted into the resource's response, the chosen status code is 200. Therefore, the figure **??** displays the intended meaning of the content.

Figure 2.7: HTTP Methods's Intended Content Meaning



Request Method	Response content is a representation of:
GET	the target resource
HEAD	the target resource, like GET, but without transferring the representation data
POST	the status of, or results obtained from, the action
PUT, DELETE	the status of the action
OPTIONS	communication options for the target resource
TRACE	the request message as received by the server returning the trace

Souce: [**?**].

As illustrated in figure **??**, the GET and POST methods are the only HTTP methods whose response content directly represents the target resource. In contrast, methods like PUT and DELETE primarily focus on conveying the status of the action rather than resource content. Meanwhile, OPTIONS and TRACE do not concern themselves with either the target resource or the status code in the same way. The HEAD method is similar to GET in that its metadata describes the target resource, but it intentionally omits the actual content transfer.

Given these distinctions, GET and POST are the most suitable choices for HATEOAS implementations, as they align with the principles outlined in RFC 9110. While developers technically have the flexibility to use any HTTP method in HATEOAS, adhering to the standard ensures consistency, interoperability, and compliance with RESTful design best practices.

Additionally, proper use of HTTP status codes is critical for clear API semantics, as displayed by the Table **??**:

Table 2.9: HTTP Status Code Semantics for Error Handling

Status Code	**Usage Context**
409	Request conflicts with server state (e.g., duplicate data in PUT requests: SSN, email, or medical license number)
404	Requested resource does not exist
400	Fallback for client errors not covered by 404/409 (e.g., malformed syntax or invalid parameters)

Source: Adapated from [**?**].

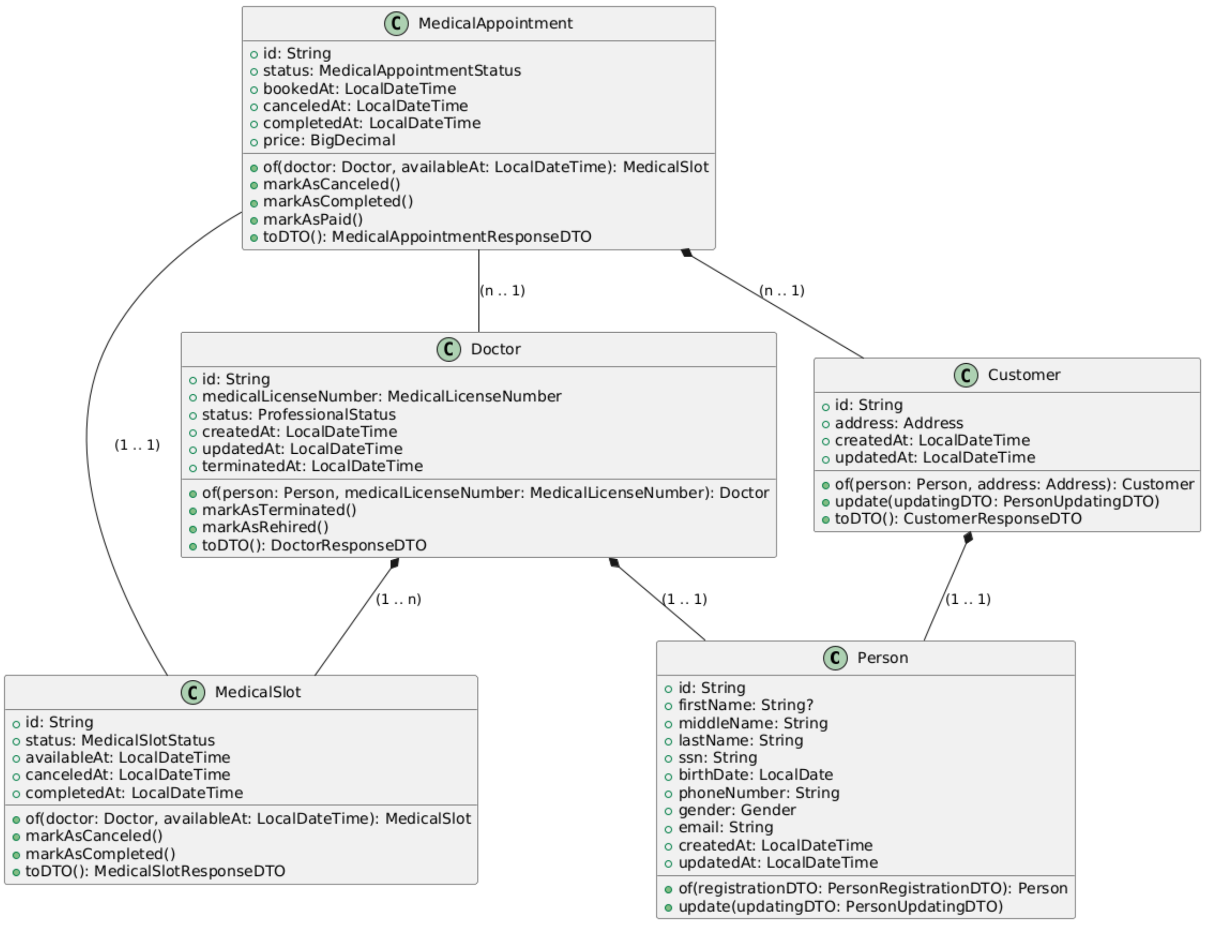
It is evident that the status code 400 is the default behavior when failure occurs. Hence, the importance of using specific status codes for specific situations highlights the impactful use of the status 404 and 409.

2.3 Application Programming Interface

UML class diagrams, as emphasized by [?, ?], provide useful pictures of the static structure, including the structure and components of each class and the relationships between classes, allowing for declarative modeling of an application's static structure in terms of concepts and their interrelations.

As an example of the previous idea, the figure **??** illustrates the relationship among the models **Person**, **Doctor**, **Customer**, **MedicalSlot**, **MedicalAppointment**, **Card** and **Payment**.

Figure 2.8: Class Diagram



MedicalAppointment
id: String
status: MedicalAppointmentStatus
bookedAt: LocalDateTime
canceledAt: LocalDateTime
completedAt: LocalDateTime
price: BigDecimal
of(doctor: Doctor, availableAt: LocalDateTime): MedicalSlot
markAsCanceled()
markAsCompleted()
markAsPaid()
toDTO(): MedicalAppointmentResponseDTO
(n .. 1)
(n .. 1)
(1 .. 1)
Doctor
id: String
medicalLicenseNumber: MedicalLicenseNumber
status: ProfessionalStatus
createdAt: LocalDateTime
updatedAt: LocalDateTime
terminatedAt: LocalDateTime
of(person: Person, medicalLicenseNumber: MedicalLicenseNumber): Doctor
markAsTerminated()
markAsRehired()
toDTO(): DoctorResponseDTO
Customer
id: String
address: Address
createdAt: LocalDateTime
updatedAt: LocalDateTime
of(person: Person, address: Address): Customer
update(updatingDTO: PersonUpdatingDTO)
toDTO(): CustomerResponseDTO
(1 .. n)
(1 .. 1)
(1 .. 1)
MedicalSlot
id: String
status: MedicalSlotStatus
availableAt: LocalDateTime
canceledAt: LocalDateTime
completedAt: LocalDateTime
of(doctor: Doctor, availableAt: LocalDateTime): MedicalSlot
markAsCanceled()
markAsCompleted()
toDTO(): MedicalSlotResponseDTO
Person
id: String
firstName: String?
middleName: String
lastName: String
ssn: String
birthDate: LocalDate
phoneNumber: String
gender: Gender
email: String
createdAt: LocalDateTime
updatedAt: LocalDateTime
of(registrationDTO: PersonRegistrationDTO): Person
update(updatingDTO: PersonUpdatingDTO)

Source: Author's creation

From the **??**, it is notable that both **Customer** and **Doctor** have a one-to-one relationship with **Person**. One **Doctor** can be associated with multiple **MedicalSlot** and **MedicalAppointment**, while **Customer** can be associated with multiple **MedicalSlot**. One **MedicalSlot** can associate with only one **MedicalAppontment**. One **Card** can be associated with multiple **Payment**. It can be associated with only one **MedicalAppointment**.

Similar to the class diagram, the figure **??** displays the ERD. [?] agrees that the ERD provides an overall grasp of the data requirements, modeling, and database structures of the information system before the implementation phase. Therefore, as exposed by the figure **??**, every model, except **Person** and **Card**, has at least one embedded model inside it, as listed below in the Table **??**:

Table 2.10: Entity Composition Relationships

Parent Entity	**Contained Entities**
Customer	Person
Doctor	Person
Payment	Card, MedicalAppointment
MedicalSlot	Doctor, MedicalAppointment
MedicalAppointment	Customer, Doctor

Source: Author's creation.

The first major visual difference between the figure **??**, that embodies a class diagram, and the figure **??**, that embodies an ERD, is that the ERD can contain embedded objects within its models. While class diagrams focus on static objects like models and the relationships among them, ERD focus on portraying the architecture of the system.

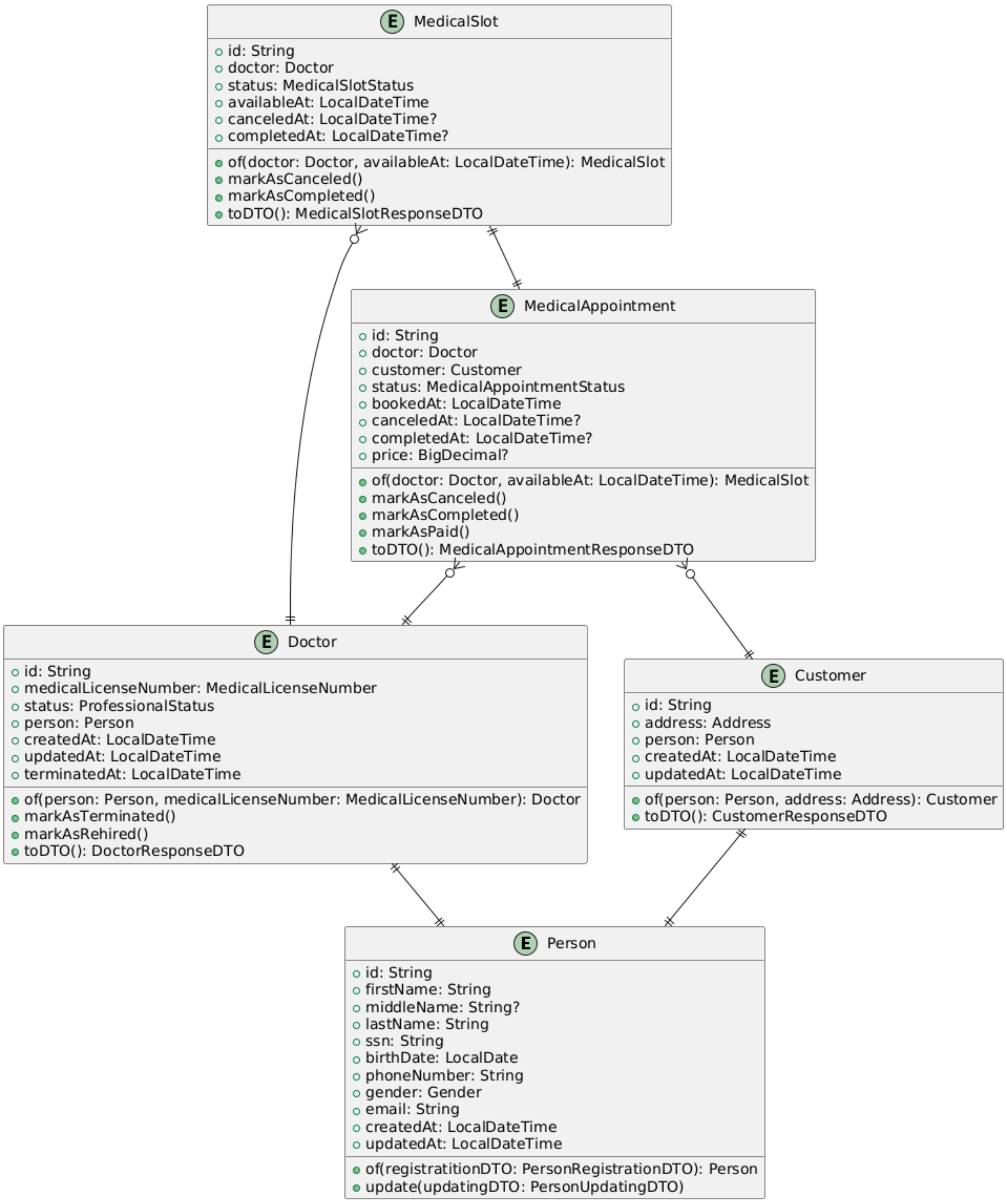
Because ERDs can include embedded objects within their models, they serve as valuable managerial analysis tools. This capability allows the diagram to illustrate a more comprehensive view of the domain layer's models and their embedded objects within the API's architecture. By visualizing how each embedded object is integrated and interacts with its parent model and the overall system architecture, ERDs facilitate a deeper understanding of the system's design.

While the ERD's capacity to represent embedded objects offers a significant advantage in visualizing the system's architecture and the intricate relationships within its data structures, it is crucial to recognize that it complements, rather than replaces, the class diagram. Class diagrams excel at defining the static structure of the system, clearly outlining the classes, their attributes, methods, and the various types of relationships between them, such as inheritance, association, and aggregation.

This provides a fundamental blueprint of the system's conceptual model. The ERD, on the other hand, builds upon this foundation by focusing on the data perspective, illustrating how entities are organized, their attributes, and the relationships crucial for data management and persistence. By showing how embedded objects are contained within these entities, the ERD offers a more granular view of the data's composition and flow within the system's architecture. Therefore, a comprehensive understanding necessitates the use of both diagrams.

The class diagram provides the abstract, object-oriented view of the system's building blocks, while the ERD offers the concrete, data-centric perspective, detailing how these building blocks are instantiated and interconnected at the data level. This dual perspective allows for a more thorough analysis, ensuring that both the structural integrity of the system's objects and the efficient organization of its data are well-understood and effectively designed.

Figure 2.9: Entity-Relationship Diagram



MedicalSlot
id: String
doctor: Doctor
status: MedicalSlotStatus
availableAt: LocalDateTime
canceledAt: LocalDateTime?
completedAt: LocalDateTime?
of(doctor: Doctor, availableAt: LocalDateTime): MedicalSlot
markAsCanceled()
markAsCompleted()
toDTO(): MedicalSlotResponseDTO
MedicalAppointment
id: String
doctor: Doctor
customer: Customer
status: MedicalAppointmentStatus
bookedAt: LocalDateTime
canceledAt: LocalDateTime?
completedAt: LocalDateTime?
price: BigDecimal?
of(doctor: Doctor, availableAt: LocalDateTime): MedicalSlot
markAsCanceled()
markAsCompleted()
markAsPaid()
toDTO(): MedicalAppointmentResponseDTO
Doctor
id: String
medicalLicenseNumber: MedicalLicenseNumber
status: ProfessionalStatus
person: Person
createdAt: LocalDateTime
updatedAt: LocalDateTime
terminatedAt: LocalDateTime
of(person: Person, medicalLicenseNumber: MedicalLicenseNumber): Doctor
markAsTerminated()
markAsRehired()
toDTO(): DoctorResponseDTO
Customer
id: String
address: Address
person: Person
createdAt: LocalDateTime
updatedAt: LocalDateTime
of(person: Person, address: Address): Customer
toDTO(): CustomerResponseDTO
Person
id: String
firstName: String
middleName: String?
lastName: String
ssn: String
birthDate: LocalDate
phoneNumber: String
gender: Gender
email: String
createdAt: LocalDateTime
updatedAt: LocalDateTime
of(registratitionDTO: PersonRegistrationDTO): Person
update(updatingDTO: PersonUpdatingDTO)

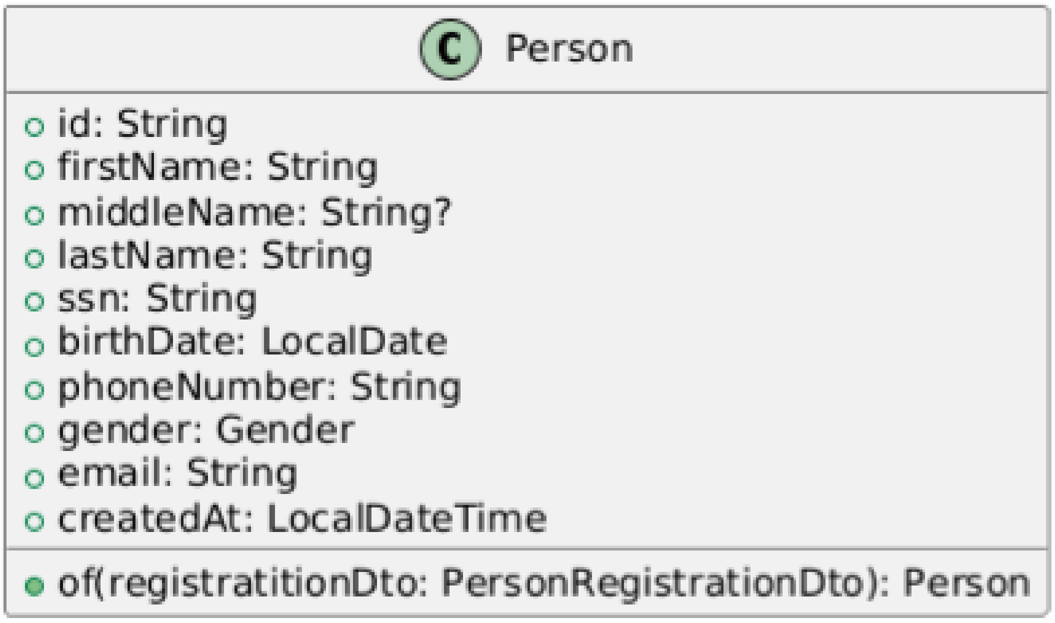
Source: Author's creation

Complementary to the aforementioned paragraph, it is notable the presence of embedded objects in the classes in the figure **??**, which indicates that the adopted database is likely a NoSQL database, due to the presence of objects instead of any foreign keys, which the schema flexibility permits foreign keys to be used, as discussed in the subsection MongoDB.

Additionally, [**?**] cites that the class diagrams are the design from which other models are derived. For this reason, APIs's models are modeled according to the class diagram. The use of class diagrams comes with the actual reflection of real-world information, as evaluated by [?].

Conceptually, a class of UML class diagram is, according to [?, ?], a set of objects with common features whose attributes represent real-world data. In order to exemplify it, the figure **??** is the UML class diagram's class that represents statically the model **Person**.

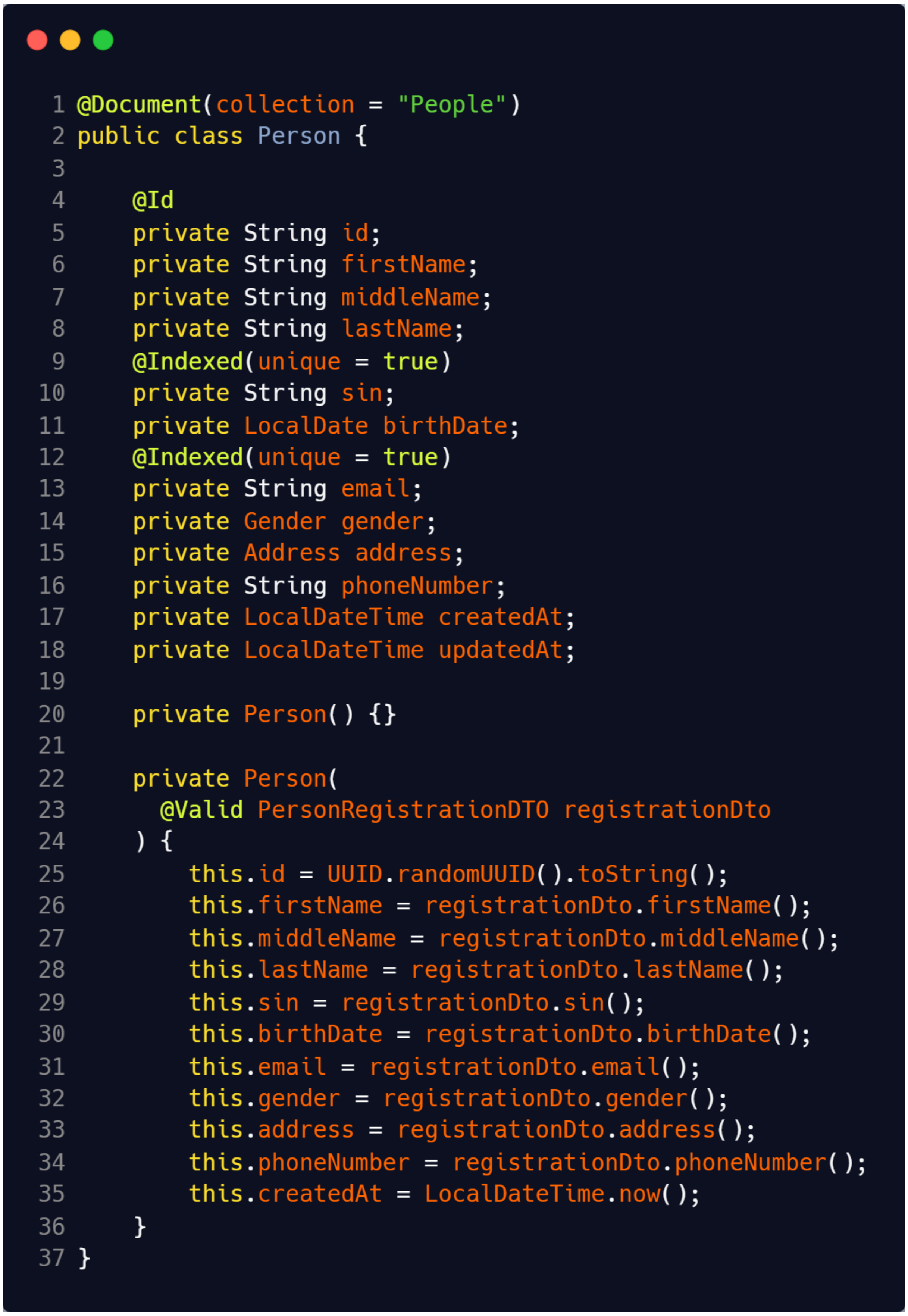
Figure 2.10: Example of a UML Class Diagram's Class



Source: Author's creation

As demonstrated by the figure **??**, the class was used to create the Java model presented in the figure **??**, that will be persisted into a MongoDB document.

Figure 2.11: Java Model of Person



```java
@Document(collection = "People")
public class Person {

    @Id
    private String id;
    private String firstName;
    private String middleName;
    private String lastName;
    @Indexed(unique = true)
    private String sin;
    private LocalDate birthDate;
    @Indexed(unique = true)
    private String email;
    private Gender gender;
    private Address address;
    private String phoneNumber;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;

    private Person() {}

    private Person(
      @Valid PersonRegistrationDTO registrationDto
    ) {
        this.id = UUID.randomUUID().toString();
        this.firstName = registrationDto.firstName();
        this.middleName = registrationDto.middleName();
        this.lastName = registrationDto.lastName();
        this.sin = registrationDto.sin();
        this.birthDate = registrationDto.birthDate();
        this.email = registrationDto.email();
        this.gender = registrationDto.gender();
        this.address = registrationDto.address();
        this.phoneNumber = registrationDto.phoneNumber();
        this.createdAt = LocalDateTime.now();
    }
}
```

Source: Author’s creation.

The shift from the class diagram to a model was possible due to the precision of the modeling that the UML class diagram offers.

Moreover, the selected API for this work is the present in the clickable link, hereafter referred to as *the API* which was developed by the author of this dissertation. It was developed using Spring Boot, Java and WebFlux, and adopts reactive and non-blocking programming. Its domain models are defined in figure **??**, with their relationships illustrated in figure **??**.

Formally, an API is a standardized set of tools, protocols, and definitions that enable interoperability—the ability of disparate systems to communicate seamlessly despite differences in language, platform, or interface. APIs provide both an abstraction layer for system interactions and concrete implementations (e.g., packaged classes or interface types). As controlled data-sharing mechanisms, they facilitate resource reuse and multi-source integration, addressing critical needs in application development and research [**?**, **?**, **?**, **?**, **?**, **?**, **?**].

Additionally to the aforementioned paragraph, the API adopts the modular monolith instead of microservices or monolith architectures. [**?**] defines modular monolith as an architecture that blends the simplicity of a monolith with the benefits of microservices. The author too informs that it structures the system into loosely coupled modules with clear boundaries and explicit dependencies. Each module remains independent and isolated, enabling separate development while being deployed as a single unit.

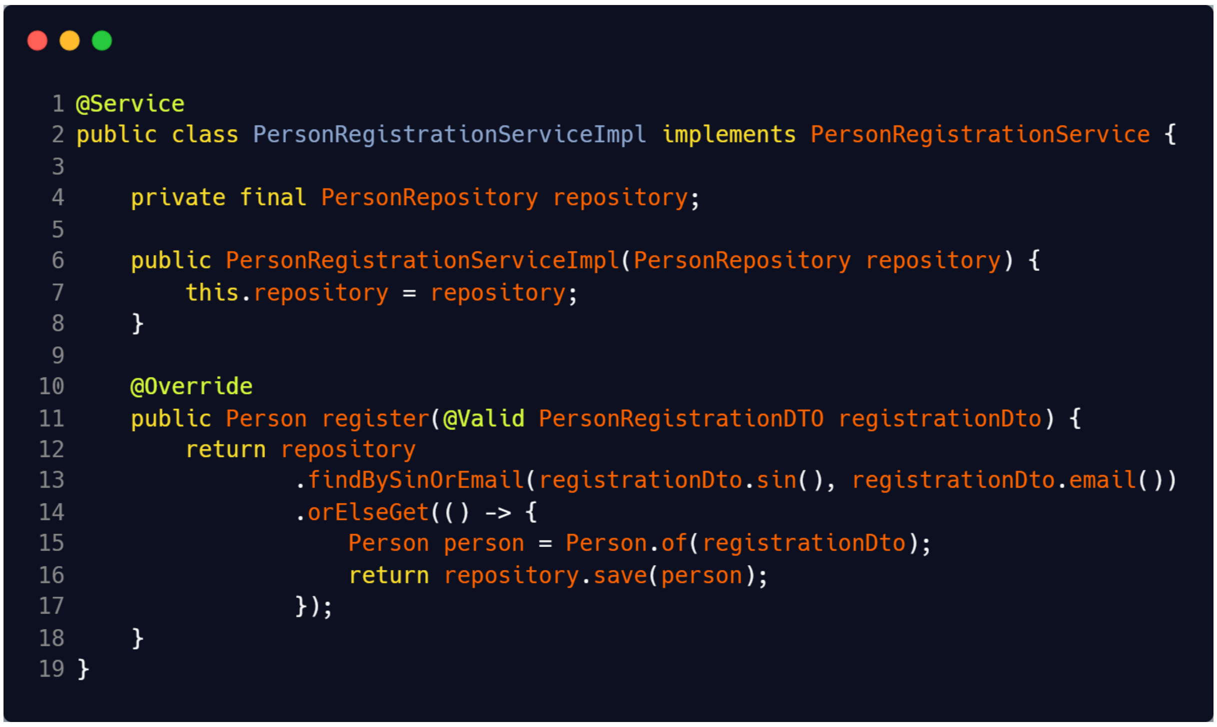
This approach strikes a middle ground between monoliths and microservices, emphasizing interchangeability, reusability, and well-defined interfaces. It simplifies development, testing, and deployment by encapsulating functionality within modules [**?**].

An example is Spring Modulith, an experimental Spring project that enforces modularity in Spring Boot applications. It provides conventions for declaring and validating modules, ensuring proper encapsulation and controlled exposure of Spring Beans. Modules can be organized as sub-packages, with restricted visibility between them to maintain clean boundaries [**?**].

Considering the explanation of Spring Modulith, the API uses Spring Modulith to organize its files. The project permits hiding components such as controllers, implementation classes of the services, repositories, and permits the exposure of components such as exceptions, utility classes, extension functions, and interfaces of the services.

One example is: the module *people* has as sub-module *services*, that has as files the interface *PersonRegistrationService*, which is an interface, as illustrated in the figure **??**. The class *PersonRegistrationServiceImpl*, which is the implementation of an interface, is illustrated in the figure **??**. Hereafter, both figures shall be referred to, respectively, as *the interface* and *the implementation* for brevity.

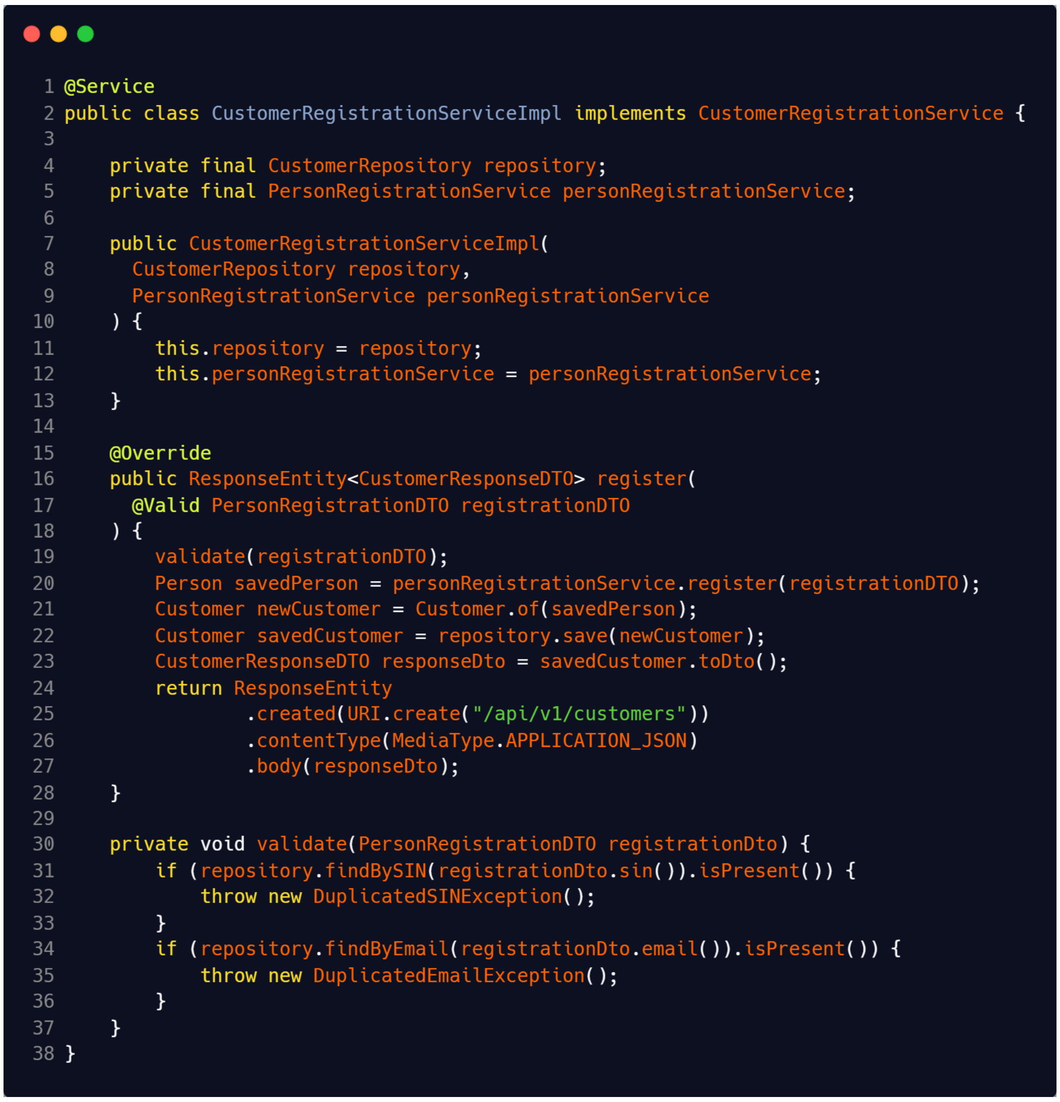
Figure 2.12: Person Registration Service



```java
@Service
public class PersonRegistrationServiceImpl implements PersonRegistrationService {

    private final PersonRepository repository;

    public PersonRegistrationServiceImpl(PersonRepository repository) {
        this.repository = repository;
    }

    @Override
    public Person register(@Valid PersonRegistrationDTO registrationDto) {
        return repository
                .findBySinOrEmail(registrationDto.sin(), registrationDto.email())
                .orElseGet(() -> {
                    Person person = Person.of(registrationDto);
                    return repository.save(person);
                });
    }
}
```

Source: Author's creation

Figure 2.13: Person Registration Service Implementation



```java
@Service
public class CustomerRegistrationServiceImpl implements CustomerRegistrationService {

    private final CustomerRepository repository;
    private final PersonRegistrationService personRegistrationService;

    public CustomerRegistrationServiceImpl(
      CustomerRepository repository,
      PersonRegistrationService personRegistrationService
    ) {
        this.repository = repository;
        this.personRegistrationService = personRegistrationService;
    }

    @Override
    public ResponseEntity<CustomerResponseDTO> register(
      @Valid PersonRegistrationDTO registrationDTO
    ) {
        validate(registrationDTO);
        Person savedPerson = personRegistrationService.register(registrationDTO);
        Customer newCustomer = Customer.of(savedPerson);
        Customer savedCustomer = repository.save(newCustomer);
        CustomerResponseDTO responseDto = savedCustomer.toDto();
        return ResponseEntity
                .created(URI.create("/api/v1/customers"))
                .contentType(MediaType.APPLICATION_JSON)
                .body(responseDto);
    }

    private void validate(PersonRegistrationDTO registrationDto) {
        if (repository.findBySIN(registrationDto.sin()).isPresent()) {
            throw new DuplicatedSINException();
        }
        if (repository.findByEmail(registrationDto.email()).isPresent()) {
            throw new DuplicatedEmailException();
        }
    }
}
```

Source: Author's creation

The implementation includes a repository (*personRepository*), shown in figure **??**. As the repository acts as an abstraction between layers—specifically within the data access layer—it manages database operations according to [**?**]. Since repositories handle sensitive data, they must be restricted from unrelated components to prevent unauthorized or unintended access. This is achieved through the interface depicted in figure **??**, which abstracts the implementation and enforces access control.

In summary, the API is responsible for the tasks listed in the Table **??**:

Table 2.11: API Functionalities by Domain

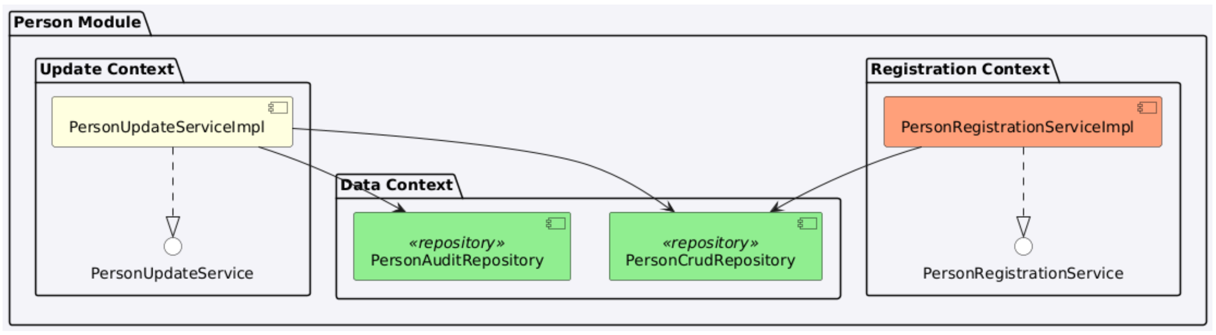
Domain	Functionalities
Person Management	Customer registration, Doctor registration/termination/rehiring
Medical Slots	Registration, cancellation, completion
Medical Appointments	Registration, cancellation, completion
Data Operations	Retrieval
Payment Simulation	Card creation, appointment payment

Source: Author's creation.

The functionalities listed in the Table **??** represent the core tasks performed by the API's service layer. These operations are visualized in the component diagram, which–as noted by [**?**]—portrays the API's component organization. [**?**, **?**] cite that a UML component diagram shows the physical structure of a system, illustrating software components (e.g., libraries, executables) and their dependencies. It focuses on the system's modular building blocks, not functionality, and can be viewed at different levels of detail. Multiple component diagrams are needed for complex systems, providing insights into implementation, dependencies, and modularity, aiding system construction.

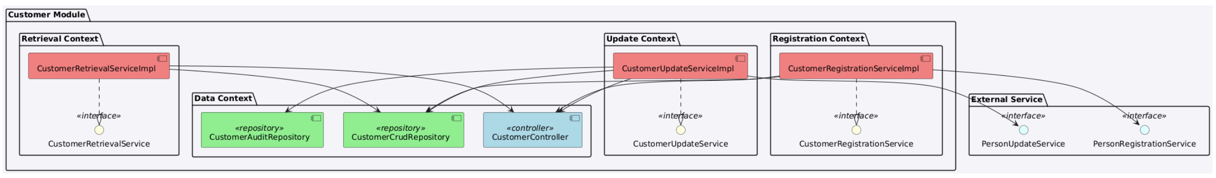
The component diagram was split into the figures **??**, **??**, **??**, **??** and **??**.

Figure 2.14: Component Diagram: Person's Piece



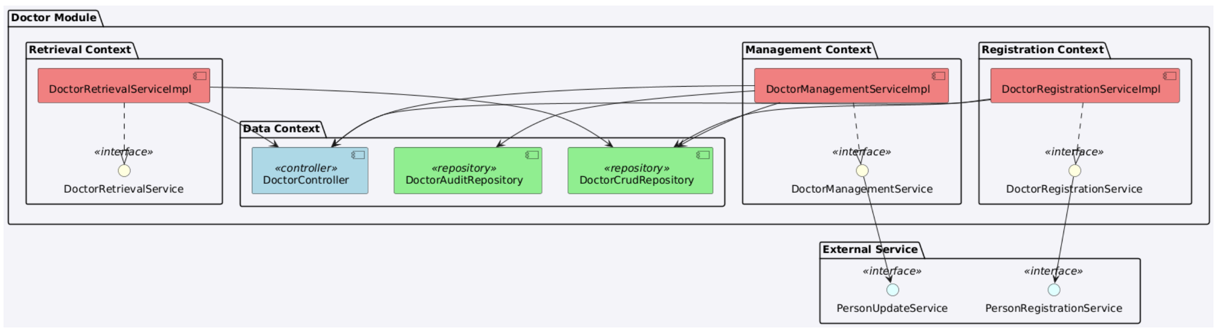
Source: Author's creation.

Figure 2.15: Component Diagram: Customer's Piece



Source: Author's creation.

Figure 2.16: Component Diagram: Doctor's Piece



Source: Author's creation.

Figure 2.17: Component Diagram: Medical Slot's Piece

`figures/api/component_diagram_medical_slot.png`

Source: Author's creation.

Figure 2.18: Component Diagram: Medical Appointment's Piece

figures/api/component_diagram_medical_appontment.png

Source: Author's creation.

The organization of the API is divided into 7 different modules and 18 services, although not all services were selected to undergo the planned software tests. The selected services are shown in the Table **??**:

Table 2.12: API's Selected Services

Modules	**Services**
payments	MedicalAppointmentPaymentService
doctors	DoctorRegistrationService, DoctorUpdatingService, DoctorManagementService
customers	CustomerRegistrationService, CustomerUpdatingService
medicalAppointments	MedicalAppointmentManagementService, MedicalAppointmentRegistrationService
medicalSlots	MedicalSlotManagementService, MedicalSlotRegistrationService

Source: Author's creation.

From the API's services, 10 out of 18 were selected. The test coverage is 55,56%. Additionally, the excluded services are:

1. PersonRegistrationService.
2. CustomerRetrievalService.
3. DoctorRetrievalService.
4. CardRetrievalService.
5. CardRegistrationService.
6. CardDeletionService.
7. MedicalAppointmentRetrievalService.
8. MedicalAppointmentUpdatingService.
9. MedicalSlotRetrievalService.
10. MedicalSlotUpdatingService.

These services were not selected primarily because their responses generally do not require a validation process to verify expected wrongful conditions and the subsequent throwing of exceptions. Furthermore, even in the cases where these services might throw an exception, it is not considered sufficiently important to include them in the unit and integration tests.

The selected services are the most essential and important of the API. They represent the most used functions. They are the core of the application.

Finally, the endpoints of the API are shown in the figures **??**, **??**, and **??** displaying the user interface that Swagger UI generated for the API's endpoints.

Figure 2.19: Swagger UI - Page 1

figures/swagger_ui_1.png

Source: Author's creation.

Figure 2.20: Swagger UI - Page 2

figures/swagger_ui_2.png

Source: Author's creation.

Figure 2.21: Swagger UI - Page 3

figures/swagger_ui_3.png

Source: Author's creation.

As discussed by the subsection HTTP Semantics, the endpoints present in the figures **??**, **??**, and **??** are compliant with RFC 9110 in both methods's guidelines and URI format, due to the presence of the method and the URI's relative path. Swagger UI was used to create the content shown in the figures **??**, **??**, and **??**, due to its ability, as stated by [**?**], to expose graphically API resources, which are automatically generated from an OpenAPI Specification.[2]

2.4 Automated Software Testing

Testing, one of the most crucial tasks along the SDLC, can easily exceed half of a project's total effort. A successful testing approach can save significant effort and increase product quality, thereby increasing customer satisfaction and lowering maintenance costs [**?**, **?**]

The potentially vast scope of testing can be visualized as sticking small pins into a doll, representing limited coverage across a large domain. This analogy underscores the necessity of maintaining a realistic expectation of what testing can achieve. Consequently, while capable of

[2]See the official site of Swagger UI.

identifying various errors and bugs, the inherent limitations mean that not all defects may be discovered, and thus the importance of testing should not be overemphasized. Recognizing that different testing techniques are complementary, a comprehensive approach utilizing multiple methods offers the greatest likelihood of detecting the majority of errors [**?**, **?**, **?**, **?**].

In contrast with the previous paragraph, testing is an indispensable approach that improves the quality of the system as well as increases the chances of a successful effort. Additionally, it is required to remove the discrepancy in the development process of the software product development [**?**].

Furthermore, testing is defined by [**?**] as the standard process used to validate that software conforms to the formal requirements. [**?**] is the process of writing a program in any programming or scripting language that duplicates the manual test case steps with the help of an external automation helper tool.

Good Testing implies more than just executing a program with desired inputs and expected output. Testing provides the chance to review requirements for important quality attributes. And it requires the planning of test cases, design of test cases, execution of test cases, and test review, which is given as feedback to the planning team. The dangerousness of the absence of tests in software tends to lead to more reckless use of resources and money, which is comparable to driving a car without any brakes or gears [**?**].

As a consequence, before implementing any software product, tests must be made. They must follow a set of various phases. Testing activities ask questions and resolve issues earlier. There are many activities that are performed during the testing. According to [**?**, **?**], the testing process has the following activities: planning, design and implementation, execution, evaluation.

Software testing includes all activities aimed at identifying the differences between specifications of the software and the actual behavior. The objectives of testing are the ones mentioned in the table **??** [**?**, **?**, **?**]:

Table 2.13: Key Aspects of Software Testing

Aspect	**Description**
Evaluation	Assesses works under uncommon conditions and demonstrates that components are prepared for integration.
Revealing	Finds defects, errors, and deficiencies. Evaluates system capabilities and limitations, quality of components, work artifacts, and the system.
Anticipation	Provides information to prevent or reduce errors, clarify system specifications and execution. Identifies approaches to avoid risks and issues in the future.

Continued on next page

Table 2.13 – continued from previous page

Aspect	**Description**
Improving quality	Effective testing minimizes errors and consequently enhances software quality.
Finding and Preventing Defects	The primary task is to identify defects and report them for rectification. Testing reveals the presence of defects.
Satisfying Requirements	Verifies compliance with Software Requirement Specification and Business Requirement Specification to ensure user needs are met.
Writing Quality Test Cases	Well-designed test cases validate requirements and can uncover problems in requirements or application design.
Reliability Estimation	Estimates the probability of failure-free operation and identifies the main failure causes.
Time Management	Requires effective scheduling to achieve testing objectives while ensuring timely delivery through early testing in each SDLC phase.
Quality Orientation	Focuses on user perspective and continuous quality improvement through proper quality attributes.
Tester Traits	Includes creativity, innovative thinking, a positive attitude, intellectual curiosity, and quick learning ability.

Resource: Adapated from [?, ?, ?]

Thereby, [?] argues that there are two types of software tests: manual and automated. The automated test is a type of software testing that uses software tools to execute test cases automatically. It involves the use of specialized testing software to perform the testing tasks instead of manual effort, while the manual test is a type of software testing that is carried out manually by a human tester. It involves the tester manually executing test cases without the use of automation testing tools.

Among the software testing, the automated testing, as cited by [?, ?], automated testing tools relieve the tester from the burden of the execution of the test cases, consist of a series of processes, activities, and tools that execute the software under test and record the results of the tests.

In addition, as conveys, a potential testing tool should be capable of exercising as many program statements as possible on execution when given a program statement with a set of input parameters. As understandability, portability, and maintainability of executable test descriptions are necessary qualities, repeatability and accuracy are sought by testing users. This is one reason organizations are both to uncover bugs and ensure that they meet performance standards [?].

Conversely, software test automation improves software quality and reduces software development costs in the long term by increasing the Return on Investment and further maturity of automatic test generation and debugging based on experience. Automation is not a quick solution for projects that are running behind schedule or over budget. There is a significant investment required to implement automated testing, but it is cost-effective when testing is performed on a regular basis. Software automation testing tools provide greater efficiency and consistency to the testing process, and have proven beneficial in terms of increasing quality while minimizing project schedules and effort [?, ?].

The types of tests are unit, integration, functional, system, acceptance, beta, and regression, as evidenced by [?]. The author describes in the table **??** via a visual approach what the types of tests relate to their specifications, scopes, opacity, and responsible developers.

Table 2.14: General Description of the Types of Test

Testing Type	**Specification**	**Scope**	**Opacity**	**Performed By**
Unit	Low-level code design	Classes	White box	Programmer
Integration	Multi-level design	Multiple classes	White/Black box	Programmer
Functional	High-level design	Whole product	Black box	Testers
System	Requirements	Product + environment	Black box	Testers
Acceptance	Requirements	Product + environment	Black box	Customer
Regression	Code changes	Product + environment	White/Black box	Testers/Programmer

Source: Adapted from [?].

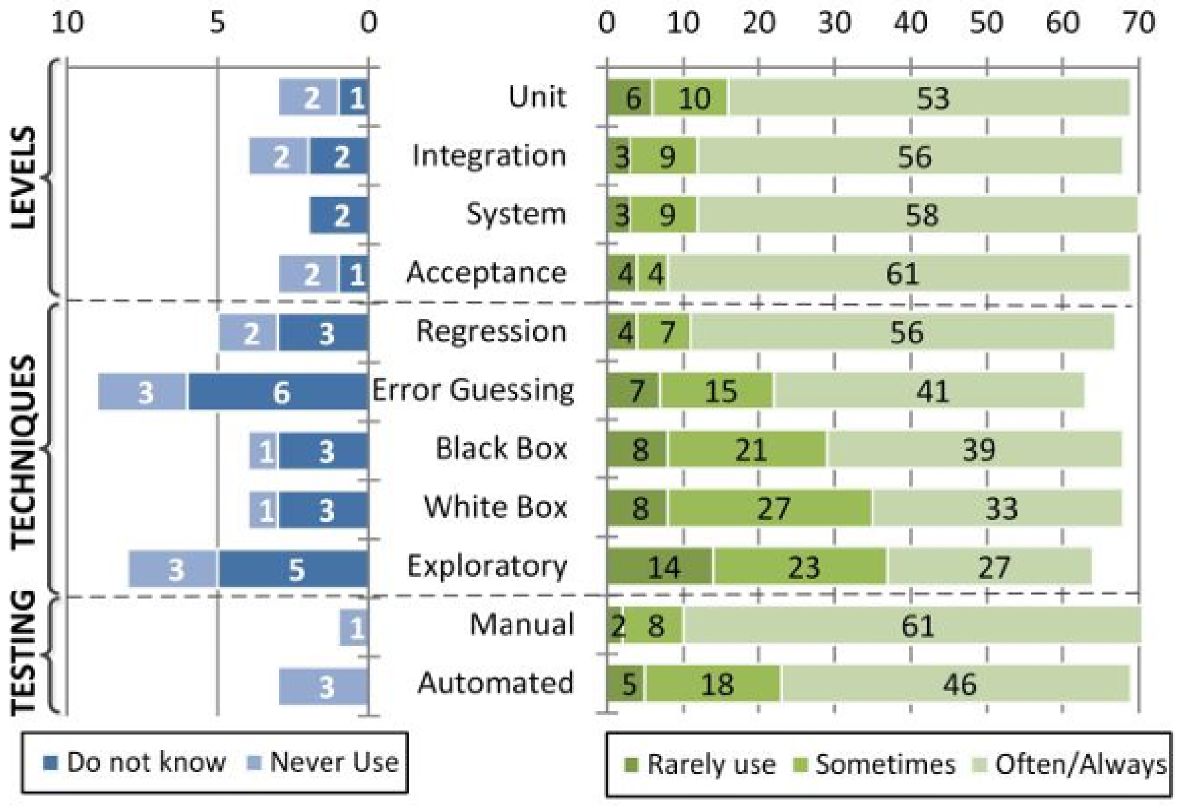
As the table **??** shows, there are two techniques: black box and white box. Thus, [?] argues that white-box testing analyzes the internal workings and structure of the software, focusing on how the system processes input to produce the required output. In contrast, black-box testing concentrates on the software's functional requirements. While an integral part of correctness testing, the principles of black-box testing extend beyond it. Black-box testing complements white-box testing techniques and is likely to identify a different set of errors.

Beyond that, black-box testing is employed throughout the SDLC and the STLC, including regression testing, acceptance testing, unit testing, integration testing, and system testing stages. The various types of testing within this technique are entirely centered on verifying the functionality of the software application [?].

Additionally, as a complement to the table **??**, Appendix **??** portrays all the definitions and key features of each type of test mentioned in the table **??**.

Given all the aforementioned definitions and key features, to the previous paragraph, a poll was made by [?] and they asked developers about the types of tests they know and use the most. The result is shown in the figure ??.

Figure 2.22: Frequency of Use: Levels, Techniques and Types of Testing

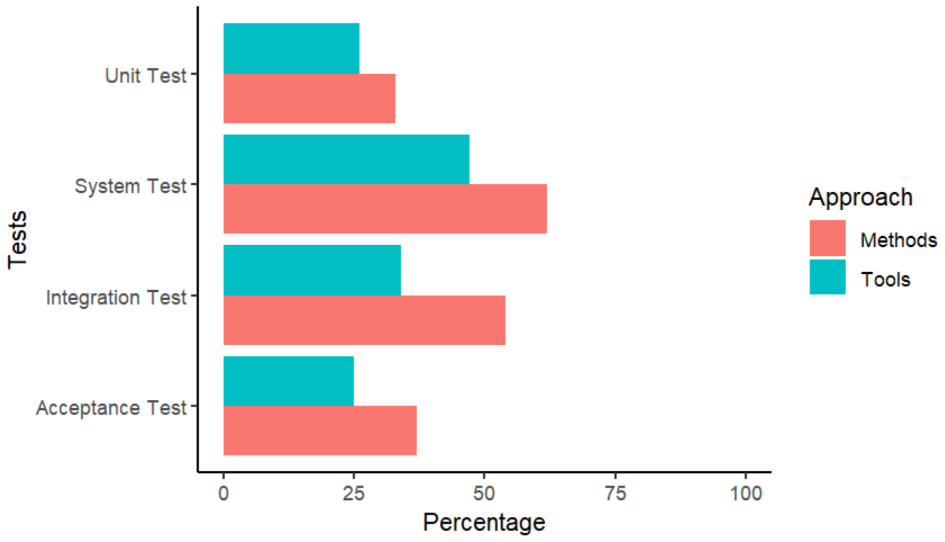


Source: [?]

The result shown in the figure ?? expresses clearly that integration and unit tests are among the most known and used types of tests. The majority of the interviewed people know about the existence and use of both the black and white box techniques. And all of them know about automated testing. The result highlighted the importance of the unit and integration tests. As illustrated by the figure ??, 53% and 56% of the developers use unit and integration tests. They, respectively, rank 3rd and 2nd, while acceptance test's rank is 1st.

Another survey conducted by [?] revealed that they are among the most used types of automated software tests, as exhibited in the figure ??.

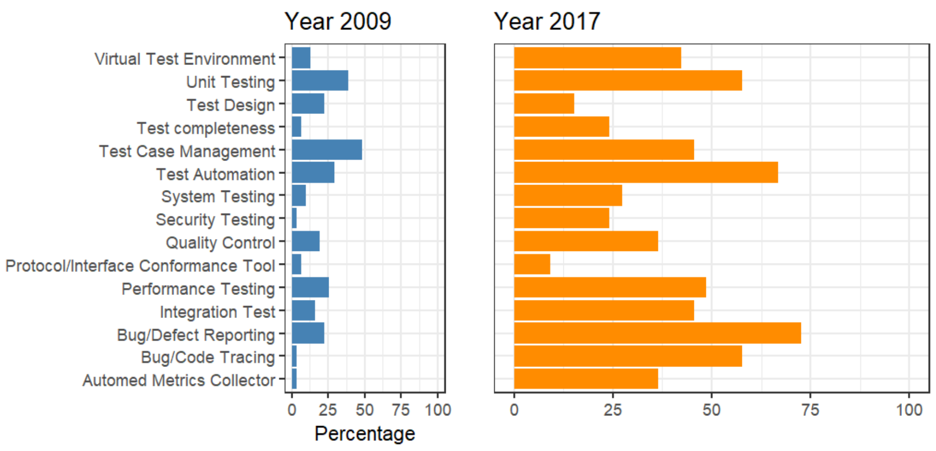
Figure 2.23: Usage of Software Testing Methods and Tools in Tests Processes



Source: Adapted from [?].

The result expressed in the figure ??, both unit and integration ranked 4th and 2nd, respectively, while system and acceptance tests ranked 1st and 3rd, respectively. As the last example of a survey, [?] illustrated in the figure ?? the number of uses of the most used types of tests.

Figure 2.24: Percentage of the Testing and QA Tools Utilized in the Industry



Source: Adapted from [?, ?].

It is notable the dominance of the unit and integration tests in the plot ??, ranking 1st and 3rd. Regardless of the ranks of unit and integration tests achieved in the surveys of [?, ?, ?, ?], they are the most used types of automated types of tests. Their importance can be noticed and measured effortlessly via the results of the surveys.

As a consequence, in order to create unit and integration tests executable, test cases are the correct approach. [?, ?] define a test case as a collection of input data used to execute the program under test, whose development is based on the requirements and specifications of the software and designed to cover all possible scenarios and edge cases.

Consequently, test cases require software testing automation tools to execute the planned tests. [?] defines the categories of software testing automation tools as follows: unit testing tools, functional testing tools, code coverage tools, test management tools, and performance testing tools.

Specially, the goal of functional testing of a software system is to find discrepancies between the actual behavior of the implemented system's functions and the desired behavior as described in the system's functional specification [?].

Considering all the paragraphs's information in this subsection, the unit and integration tests were the types of tests selected to be used in this work.

A unit test scrutinizes the behavior of an individual, self-contained unit of work. In Java applications, this *distinct unit of work* frequently, though not exclusively, corresponds to a single method. In contrast, integration and acceptance tests evaluate the interactions between different components. A unit of work is defined as a task that can be executed independently of other tasks. Consequently, unit tests often concentrate on verifying if a method adheres to the specifications outlined in its API contract [?].

For unit tests, among the types of automation tools cited by [?], JUnit was the chosen framework. The figure **??** displays the comparison between the advantages and disadvantages of JUnit.

Table 2.15: Comparison of Advantages and Disadvantages of JUnit

Advantages	**Disadvantages**
Facilitates changes and simplifies integration	Requires programming skills and consumes time to write and check.
Bugs can be found and resolved early without affecting other pieces of the code.	It can only run tests on one JVM, as such developers are unable to test applications that require multiple JVMs.

Source: Adapted from [?].

It is notable how the advantages overcome the disadvantages. In addition, for integration tests, *MockMvc* shall be the chosen framework. Complementing the paragraphs above, all integration tests shall use the following annotations present in the table **??**.

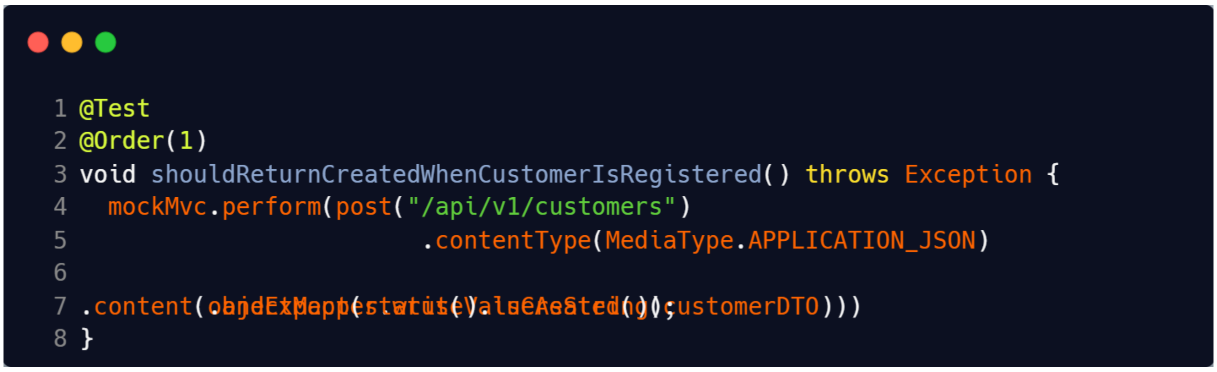
Table 2.16: Spring Boot Testing Annotations and Components

Name	Description
@SpringBootTest	Specifies that a test class runs Spring Boot based tests.
@TestMethodOrderer	A type-level annotation used to configure a MethodOrderer for the test methods of the annotated test class or test interface.
@MethodOrderer	Defines the API for ordering the test methods in a given test class.
@Test	Used to signal that the annotated method is a test method.
@Order	An annotation used to configure the order in which the annotated element should be evaluated or executed relative to other elements of the same category.

Source: Adapted from [?, ?, ?, ?, ?].

They are examples of methods of integration tests that use the annotations present in the table **??** the figures **??**, **??**, and **??**. The first figure illustrates an example of a success-oriented testing method that registers a medical slot.

Figure 2.25: Example of a Customer Successful Registration Testing Method

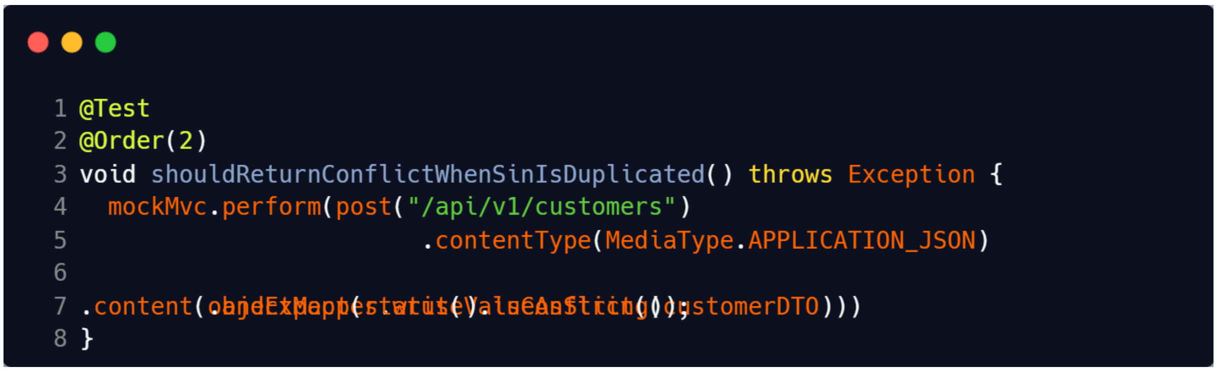


```
@Test
@Order(1)
void shouldReturnCreatedWhenCustomerIsRegistered() throws Exception {
  mockMvc.perform(post("/api/v1/customers")
                      .contentType(MediaType.APPLICATION_JSON)

.content(objectMapper.writeValueAsString(customerDTO)))
.andExpect(status().isCreated());
}
```

Source: Author's creation.

The method of the figure **??** returns the status code 201. In contrast with this method, there is the figure **??**.

Figure 2.26: Example of Customer Unsuccessful Registration Testing Method - Duplicated SIN



```
@Test
@Order(2)
void shouldReturnConflictWhenSinIsDuplicated() throws Exception {
  mockMvc.perform(post("/api/v1/customers")
                      .contentType(MediaType.APPLICATION_JSON)

.content(objectMapper.writeValueAsString(customerDTO)))
.andExpect(status().isConflict());
}
```

Source: Author's creation.

The figure **??** illustrates an unsuccessful-oriented testing method that blocks a medical slot due to a booking date and time that is in use in an active medical slot. The method of the figure **??** returns the status code 409.

Considering all the figures **??**, **??**, the methods illustrated inside them all have in common the annotations @*Order* and @*Test* and an instance of the framework *WebTestClient*. They are all using the annotations mentioned in the table **??**.

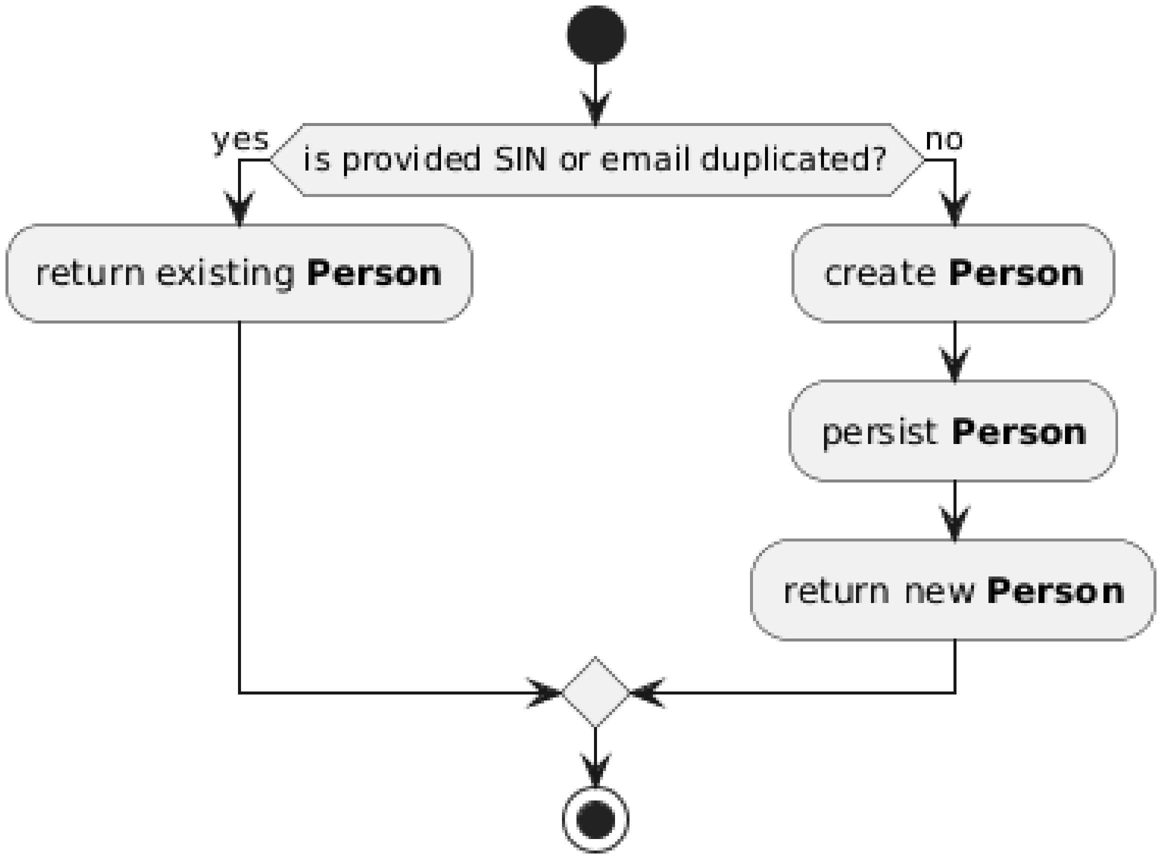
Finally, the use of the mentioned annotations created the perfect and functional environment for integration tests. As an example, the annotation @*SpringBootTest* can select a random port to avoid the use of the standard port (e.g., *8080:8080*). This prevents the API from having unexpected behaviors that interrupt the application ungracefully. Similar to the previous explanation, the annotation @*Order* derives from the annotation @*TestMethodOrder* that enables its ordering ability of setting the execution order of the test methods. One benefit of setting the execution order is barring occurrences when the successful-oriented method is executed after one of the unsuccessful-oriented ones, which blurs the test's result, if the tester is not aware of such an error.

3 Unit Tests Applied To Service Layer

4 Core Person Registration Process

The figure **??** is an activity diagram that shows visually the flow of the registration of a **Person**. As evidenced by [**?**], a UML activity diagram is a semi-formal semantic specification that is intuitive and flexible. It is used to describe a system's behaviors and the internal logic of complex operations. Therefore, it is widely utilized as a front-end tool for the system-level design of software and hardware systems.

Figure 4.1: Flow of the Registration of a Person


yes
is provided SIN or email duplicated?
no
return existing **Person**
create **Person**
persist **Person**
return new **Person**

Source: Author's creation.

If the verification discovers that the provided SSN or email is already associated with a persisted **Person**, it is retrieved. Otherwise, a new instance of **Person** is created and persisted.

5 Integration Tests Applied To The Endpoints

This section handles the integration tests applied to the endpoints. The endpoints tested are the ones selected in the subsection Application Programming Interface. Furthermore, this section covers the integration tests of the services mentioned in the table **??**.

6 Evaluation

7 Conclusions

A Appendix: Glossary

ACID stands for atomicity, consistency, isolation, and durability. ”Atomicity” guarantees that any transaction completes, and if not, prevents it. ”Consistency” ensures the stability of the database before and after any transaction. ”Isolation” guarantees that parallel transactions occur independently. Finally, ”Durability” maintains the state of the result of the transaction immutable [**?**].

BASE stands for basically available, soft state, and eventual consistency. ”Basically Available” ensures high availability of a database, even during system failures, through sharding or partitioning, which distributes and potentially replicates data across multiple servers. ”Soft State” facilitates transaction processing despite system failures or disturbances, allowing for temporary inconsistencies. While eventual consistency is guaranteed, consistency control is shifted to the application or business logic layer, due to the absence of ACID. Finally, ”Eventually consistent” weakens the constraints of consistency at the end of every transaction and guarantees that the consistent state will be granted to data stores at a later stage [**?**].

CAP (Not explicitly defined in the text)

DBMS is a software application that enables users to define, manipulate, and manage databases. It provides a structured environment for efficiently storing, organizing, retrieving, and managing large volumes of data. This includes tasks from the initial database creation and schema definition to ongoing operations such as data entry, modification, deletion, querying, and report generation, ensuring data integrity, security, and accessibility [**?**].

DTO encapsulates a set of values, enabling remote clients to retrieve the entire set in a single request. DTOs are stateless, possessing no business logic or data retrieval capabilities, and are independent of business objects, facilitating their reuse across various contexts and contract calls [**?**, **?**, **?**].

IANA is responsible for coordinating some of the key elements that keep the Internet running smoothly. Whilst the Internet is renowned for being a worldwide network free from central coordination, there is a technical need for some key parts of the Internet to be globally coordinated, and this coordination role is undertaken by us. Specifically, IANA allocates and maintains unique codes and numbering systems that are used in the technical standards (“protocols”) that drive the Internet. [**?**].

JVM is a crucial component of the Java platform, providing hardware and OS independence, compact compiled code, and protection against malicious programs. It is rogramming languages. The JVM is multithreaded, performs garbage collection, and generates events useful for profiling [**?**, **?**].

JS is a flexible and widely adopted scripting language crucial for developing interactive Web 2.0 applications. It enables offloading core functionality to the client-side browser and dynamically manipulating the Document Object Model create seamless state transitions. While its flexibility offers significant advantages, it also presents challenges in writing and maintaining robust code. As an object-based scripting language, JavaScript is natively interpreted by most common web browsers [**?**, **?**].

MVC is a software design pattern that separates an application into three interconnected parts: the Model (data and business logic), the View (user interface), and the Controller (handles user input and updates the Model 1 and View). This separation enhances code organization, reusability, flexibility, and simplifies development and maintenance of web applications by providing a clear understanding of each module. The primary goal of MVC is to isolate the business logic and application data from its presentation to the user [**?**, **?**, **?**, **?**].

RFC describes the Internet's technical foundations, such as addressing, routing, and transport technologies. RFCs also specify protocols like TLS 1.3, QUIC, and WebRTC that are used to deliver services used by billions of people every day, such as real-time collaboration, email, and the domain name system [**?**].

SQL is the standard language for retrieving and manipulating relational data. It is a high-level non-procedural data language which has received wide recognition in relational databases [**?**, **?**].

UI serves as the crucial communication bridge between the system and the user. Effective user interface design encompasses all user-visible aspects of the system. Consequently, its design is deeply intertwined with the overall architecture of the interactive system and must be integrated from the outset of the development process, rather than as an afterthought. A well-designed user interface yields significant improvements in training efficiency, task performance speed, reduced error rates, enhanced user satisfaction, and improved long-term retention of operational knowledge. Critically, the user interface should be designed using the users's own terminology and understanding of their tasks, rather than adhering to the technical jargon and conceptual framework of the programmers [**?**].

UML serves as a standard visual language for specifying, constructing, and documenting the artifacts of systems. UML employs various diagrams to represent system models graphically. This graphical notation lacks a direct textual equivalent; instead, the Object Constraint Language is utilized to express model constraints in a textual format. As a general-purpose modeling language, UML is compatible with all major object and component methods and can be applied across diverse application domains and implementation platforms [**?**, **?**].

URI is a compact sequence of characters that identifies an abstract or physical resource. As consequence, it is used throughout HTTP as the means for identifying resources [**?**, **?**].

Spring Boot is used as an alternative to deploy software products in application servers. It is considered the *de facto* standard for microservice development. The framework has a built-in server by which the process of implementing a REST application is significantly simplified [**?**].

B Appendix: Http Status Codes Summary

Table B.1: HTTP Status Codes Summary

HTTP Status Code	Name	Description
100	CONTINUE	Initial part of request received, not yet rejected by server.
101	SWITCHING PROTOCOLS	Server agrees to change application protocol.
200	OK	Request has succeeded.
201	CREATED	Request fulfilled, new resources created.
202	ACCEPTED	Request accepted but processing not completed.
203	NO AUTHORITATIVE INFORMATION	Content modified by transforming proxy.
204	NO CONTENT	Request succeeded, no additional content.
205	RESET CONTENT	Request fulfilled, user agent should reset view.
206	PARTIAL CONTENT	Server fulfilling range request.
300	MULTIPLE CHOICES	Multiple representations available for resource.
301	MOVED PERMANENTLY	Resource assigned new permanent URI.
302	FOUND	Resource temporarily under different URI.
303	SEE OTHER	Redirect to different resource for indirect response.
304	NOT MODIFIED	Conditional request condition evaluated false.
305	USE PROXY	No longer in use.
306	(UNUSED)	No longer in use.
307	TEMPORARY REDIRECT	Resource temporarily under different URI.
308	PERMANENT REDIRECT	Resource assigned new permanent URI.
400	BAD REQUEST	Server cannot process due to client error.
401	UNAUTHORIZED	Lacks valid authentication credentials.
402	PAYMENT REQUIRED	No current use.
403	FORBIDDEN	Server understood but refuses to fulfill.
404	NOT FOUND	No current representation for target resource.
405	METHOD NOT ALLOWED	Method known but not supported.
406	NOT ACCEPTABLE	No acceptable representation available.
407	PROXY AUTHENTICATION REQUIRED	Client needs to authenticate with proxy.

Continued on next page

Table B.1 – Continued from previous page

HTTP Status Code	Name	Description
408	REQUEST TIMEOUT	Server did not receive complete request.
409	CONFLICT	Conflict with current state of resource.
410	GONE	Access no longer available (likely permanent).
411	LENGTH REQUIRED	Server refuses without Content-Length.
412	PRECONDITION FAILED	Request header conditions evaluated false.
413	CONTENT TOO LARGE	Request content too large.
414	URI TOO LONG	URI too long.
415	UNSUPPORTED MEDIA TYPE	Content format not supported.
416	RANGE NOT SATISFIABLE	Requested ranges not satisfiable.
417	EXPECTATION FAILED	Expect header field could not be met.
418	(UNUSED)	No longer in use.
421	MISDIRECTED REQUEST	Server unable to produce authoritative response.
422	NON-PROCESSABLE CONTENT	Unable to process contained instructions.
426	UPGRADE REQUIRED	Server refuses with current protocol.
500	INTERNAL SERVER ERROR	Unexpected condition prevented fulfillment.
501	NOT IMPLEMENTED	Server lacks required functionality.
502	BAD GATEWAY	Invalid response from upstream server.
503	SERVICE UNAVAILABLE	Temporary overload or maintenance.
504	GATEWAY TIMEOUT	No timely response from upstream server.
505	HTTP VERSION NOT SUPPORTED	HTTP version not supported.

Source: [?].

C Appendix: Definitions of Automated Software Tests

Table C.1: Tests and Their Definitions

Testing Type	Description	Key Aspects
Unit Testing	Tests the smallest testable parts (units/modules) of an application independently.	• Focuses on individual modules • Also called module testing • Typically done by developers • Example: Testing individual functions like Addition, Subtraction in a Calculator • Known as white box testing (tests internal functionality) • Aims to eliminate interface errors early
Integration Testing	Tests the interaction and interfaces between integrated units/modules.	• Exposes faults in the interaction between units • Performed after unit testing • Approaches: Big-bang, Top-down, Bottom-up • Focuses on ensuring units work together correctly
System Testing	Tests the complete, integrated system to verify it meets specified requirements.	• Testing of the entire software and potentially hardware system • Last test by developers before acceptance testing • Includes types like usability, stress, and regression testing • Requires a System Test Plan • Typically black box testing (no knowledge of internal design) • Detects defects within inter-assemblages and the system as a whole
Acceptance Testing	Tests the system for acceptability by end users or clients.	• High-level testing on a fully integrated application • Types: Alpha, Beta, User Acceptance Testing (UAT), Business Acceptance Testing • Done by end users • Determines whether to accept or reject the product based on acceptance criteria

Continued on next page

Table C.1 – Continued from previous page

Testing Type	Description	Key Aspects
Regression Testing	Re-tests previously tested parts of the software after modifications (bug fixes, changes).	• Checks for new bugs introduced by changes • Verifies that previously reported bugs are fixed • often involves re-running failed test cases • Can be expensive but necessary for ensuring the correctness and reliability of modified software • Aims to increase confidence in the modified program • Targets potential errors like data corruption, inappropriate control sequencing, resource contention, and performance deficiencies
Installation Testing	Verifies if the software has been installed with all necessary components.	• Checks if the application can be downloaded and installed • Verifies registration processes (e.g., with a new mobile number) • Confirms receipt of necessary verification codes

Source: [?, ?]