PAYROLL MANAGEMENT SYSTEM

Introduction:

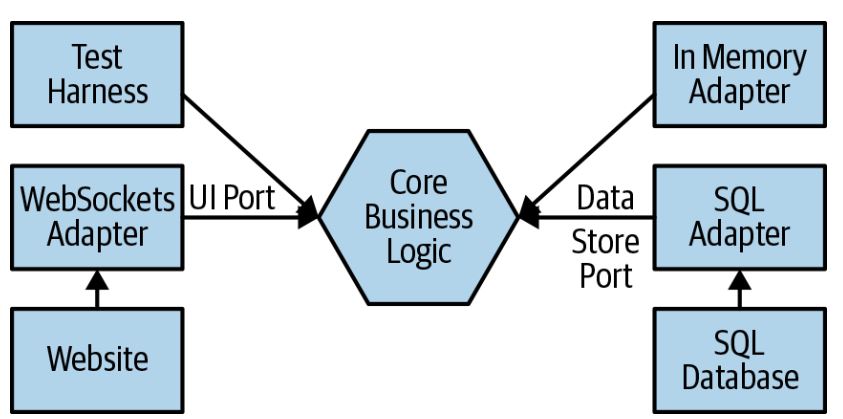
**SOFTWARE DESIGN**

This project is designed using **microservices** to manage different domain modules regarding the payroll management system.

**HEXAGONAL ARCHITECTURE SOLUTION WITH OUR MICROSERVICES**

We use the (ports and Adapters) or Hexagonal architecture introduced by Alister Cockburn to implement our **microservices** approach to solving this problem.

This will help separate the **business logic** from different implementation choices. **Events** from the outside world would arrive and depart from the business logic core through a port. An **adapter** is technology-specific implementation code that plugs into the port.



The ports of the system, would allow

1. Authentication services from external applications to log in users and into our system.
2. User Management, where various users can perform operations against our application, which is then persisted to our database using persistence frameworks.

**MANDATORY SERVICES FOR GOOD MICROSERVICES**

There are a few patterns/services that should be in place for implementing microservice-based design. This list consists of the following:

* Service discovery and registration
* Edge or proxy server
* Load balancing
* Circuit breaker
* Monitoring

And this conforms to the hexagonal architecture implementation to developing our microservices.

**SERVICE DISCOVERY AND REGISTRATION**

The **Netflix Eureka server** is used for service discovery and registration. We created the Eureka service in the last chapter. It not only allows you to register and discover services, but also provides load balancing using Ribbon.

**EDGE SERVERS**

An edge server provides a single point of access to allow the external world to interact with your system. All of your APIs and frontends are only accessible using this server. Therefore, these are also referred to as gateway or proxy servers. These are configured to route requests to different microservices or frontend applications. We'll use the **Netflix Zuul server** as an edge server in this application.

**LOAD BALANCING**

**Netflix Ribbon** is used for load balancing. It is integrated with the Zuul and Eureka services to provide load balancing for both internal and external calls.

**CIRCUIT BREAKERS**

A fault or break should not prevent your whole system from working. Also, the repeated failure of a service or an API should be handled properly. Circuit breakers provide these features**. Netflix Hystrix** is used as a circuit breaker and helps to keep the system up.

**MONITORING**

The **Hystrix dashboard** is used with **Netflix Turbine** for microservice monitoring. It provides a dashboard to check the health of running microservices.

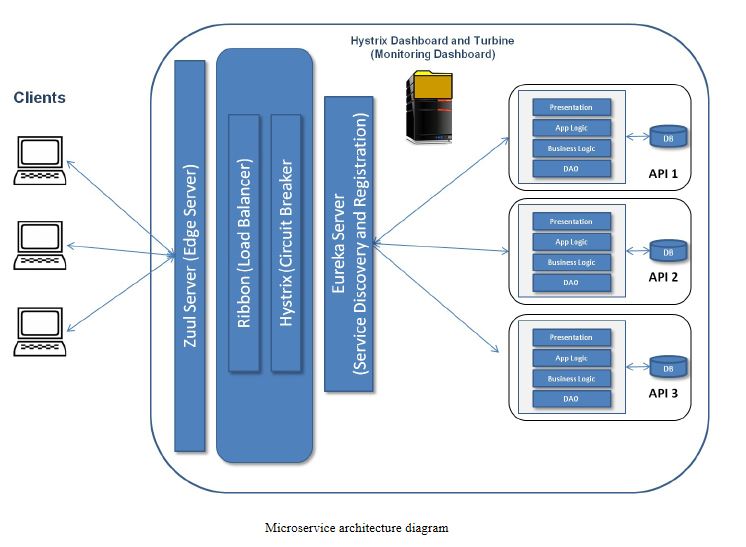
**AN OVERVIEW OF MICROSERVICES ARCHITECTURE USING Netflix OSS**

Netflix are pioneers in microservice architecture. They were the first to successfully implement microservice architecture on a large scale. They also helped increase its popularity and contributed immensely to microservices by open sourcing most of their microservice tools with Netflix **Open-Source Software Centre** (**OS**).

According to the Netflix blog, when Netflix was developing their platform, they used Apache Cassandra for data storage, which is an open-source tool from Apache. They started contributing to Cassandra with fixes and optimization extensions. This led to Netflix seeing the benefits of releasing Netflix projects with the name OSS.

Spring took the opportunity to integrate many Netflix OSS projects, such as Zuul, Ribbon, Hystrix, the Eureka server, and Turbine, into Spring Cloud. This is one of the reasons Spring Cloud provides a ready-made platform for developing production-ready microservices.

Now, let's take a look at a few important Netflix tools and how they fit into microservice architecture:



* **Edge server**: We use the Netflix Zuul server as an edge server.
* **Load balancing**: Netflix Ribbon is used for load balancing.
* **Circuit breaker**: Netflix Hystrix is used as a circuit breaker and helps to keep the system up.
* **Service discovery and registration**: The Netflix Eureka server is used for service discovery and registration.
* **Monitoring dashboard**: The Hystrix dashboard is used with Netflix Turbine for microservice monitoring. It provides a dashboard to check the health of running microservices.

**LOAD BALANCING**

Load balancing is required to service requests in a manner that maximizes speed and capacity utilization, and it makes sure that no server is overloaded with requests. The load balancer also redirects requests to the remaining host servers if a server goes down. In microservice architecture, a microservice can serve internal or external requests. Based on this, we can have two types of load balancing—client-side and server-side load balancing.

* We use server-side load balancing because of the following tradeoffs:

1. **Throughput**

Server side load balancers can handle a very high throughput than client side load balancers. Normally serverside LB vendors manufacture hardware load balancers as well. They are even more optimized to handle hundreds of thousand requests per second.

1. **Lite Weight Client Support**

For lite weight client applications (Single Page Applications, Mobile Application, IoT device) server side load balancing is the best option. Since lite weight devices have limited memory and processing power, it is difficult for them to tolerate the additional overhead of a client side load balancer.

1. **Client to Server Hight Network Latency**

Client side load balancers send heart beats to available service instances periodically (or contact discovery server to refresh available service instances). These connections get delayed or timeout might occurs if the latency is high. If the load balancing happens at the server side, then client does not need to send additional requests.

**CIRCUIT BREAKERS AND MONITORING**

In general terms, a circuit breaker is a*n automatic device for stopping the flow of current in an electric circuit as a safety measure.*

The same concept is used for microservice development, known as the **circuit breaker** design pattern. It tracks the availability of external services such as the Eureka server, API services such as restaurant-service, and so on, and prevents service consumers from performing any action on any service that is not available.

It is another important aspect of microservice architecture, a safety measure  
(failsafe mechanism) when the service does not respond to a call made by the service consumer, which is called a circuit breaker.

We'll use Netflix Hystrix as a circuit breaker. It calls the internal fallback method in the service consumer when failures occur (for example, due to a communication error or timeout). It executes embedded within its consumer of service. In the next section, you will find the code that implements this feature.

Hystrix opens the circuit and fail fast when the service fails to respond repeatedly, until the service is available again. When calls to a particular service reach a certain threshold (the default threshold is 20 failures in five seconds), the circuit opens and the call is not made. You must be wondering, if Hystrix opens the circuit, then how does it know that the service is available? It exceptionally allows some requests to call the service.

**MONITORING**

Hystrix provides a dashboard with a web UI that provides nice graphics of circuit breakers:

Netflix Turbine is a web application that connects to the instances of your Hystrix applications in a cluster and aggregates information, which it does in real time (updated every 0.5 seconds). Turbine provides information using a stream that is known as a Turbine stream.

If you combine Hystrix with Netflix Turbine, then you can get all of the information from the Eureka server on the Hystrix dashboard. This gives you a landscape view of all of the information about the circuit breakers.

To use Turbine with Hystrix, just type in the Turbine URL http://localhost:8989/turbine.stream (port 8989 is configured for the Turbine server in **application.properties**) in the first textbox shown in the preceding screenshot, and click on Monitor Stream.

Netflix Hystrix and Turbine use RabbitMQ, an open-source message queuing software. RabbitMQ works on **Advance Messaging Queue Protocol** (**AMQP**). It is a software in which queues can be defined and used by connected applications to exchange messages. A message can include any kind of information. A message can be stored in the RabbitMQ queue until a receiver application connects and consumes the message (taking the message off the queue).

Hystrix uses RabbitMQ to send metrics data feed to Turbine.

***Before we configure Hystrix and Turbine, please install the RabbitMQ application on your platform. Hystrix and Turbine use RabbitMQ to communicate between themselves.***

**CREATING TURBINE SERVICES**

Turbine aggregates all **/hystrix.stream** endpoints into a combined **/turbine.stream** for use in the Hystrix dashboard, which is more helpful as it allows to see the overall health of the system in a single dashboard rather than monitoring the individual services using **/hystrix.stream**. We'll create another service project in the IDE like the others. Then, we'll add Maven dependencies for Turbine in pom.xml.

**PASSWORDS AND SECURITY**

Because storing passwords as plain texts is a bad practice, as malicious users can log on to our system and pretend to be the users, we apply a **cryptographic hash function** to the password. This is a function that takes some arbitrarily sized **input string**  and converts it to **some output**, called a **digest.** Cryptographic hash functions are **deterministic** , so if we hash the same input again we get the same result. This is essential in order to be able to check the **hashed password later.** Another key property is that while it should be quick to go from the **input**  to the **digest**, the reverse function should take so long or use so much memory that it is impractical for an attacker to reverse the digest. We use an established Java library called **Bouncy Castle.** This is open source and has undergone heavy peer review. We use the **Scrypt** hashing function, which is a modern algorithm specifically designed for storing passwords.

**SOFTWARE and TOOLS USED**

1. Spring Tool Suite
2. Spring cloud APIS, which provides a cloud-ready solution that is available through SpringBoot.
3. Wamp Server
4. MySQL Workbench
5. Angular SPA framework (for UI)
6. Maven
7. To manage dependencies, and packaging in a specific format such as WAR or JAR.
8. To automate repetitive tasks including building, testing and deploying your application.

**MAVEN PROJECT STRUCTURE**

The great thing about Maven is that from the get-go it comes with structure to help maintenance. A Maven project starts with two main folders:

**/src/main/java**

This is where you will develop and find all the Java classes required for your project.

**src/test/java**

This where you will develop and find all the tests for your project.

There are two additional folders that are useful but not required:

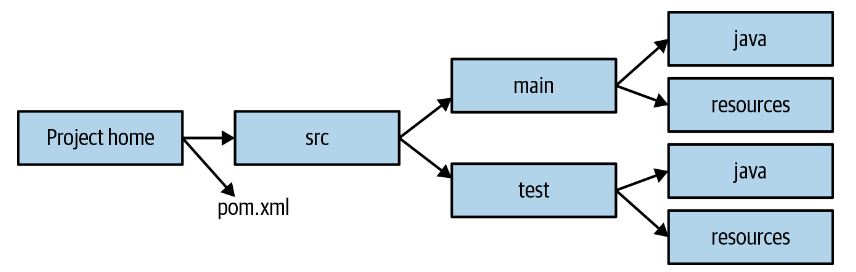
**src/main/resources**

This is where you can include extra resources such as text files needed by your application.

**src/test/resources**

This is where you can include extra resources used by your tests.

Having this common directory layout allows anyone familiar with Maven to be immediately able to locate important files. To specify the build process, you will need to create a *pom.xml* file where you specify various XML declarations to document the steps required to build your application. The figure summarizes the common Maven project layout.



**EXAMPLE BUILD FILE**

The next step is to create the *pom.xml* that will dictate the build process. The code snippet in [Example 3-26](file:///C:\Users\STHIGWGFWB\AppData\Local\Temp\pmtoq3eq.pqt\OEBPS\ch03.xhtml#maven_pom_simple) shows a basic example that you can use for building the Bank Statements Analyzer project. You will see several elements in this file:

**project**

This is the top-level element in all *pom.xml* files.

**groupId**

This element indicates the unique identifier of the organization that created the project.

**artifactId**

This element specifies a unique base name for the artifact generated by the build process.

**packaging**

This element indicates the package type to be used by this artifact (e.g., JAR, WAR, EAR, etc.). The default is JAR if the XML element packaging is omitted.

**version**

The version of the artifact generated by the project.

**build**

This element specifies various configurations to guide the build process such as plug-ins and resources.

**dependencies**

This element specifies a dependency list for the project.

**DEVELOPING AND IMPLEMENTING OUR MICROSERVICE**

We use a domain-driven implementation and approach to implement the microservices using **Spring Cloud.**

1. Entities

These are categories of objects that are identifiable and remain the same throughout the states of the product/services. These objects are *not* defined by their attributes, but by their identities and threads of continuity. Entities have traits such as identity, a thread of continuity, and attributes that do not define their identity.

1. Value Objects

just have the attributes and no conceptual identity. A best practice is to keep VOs as immutable objects. In the Spring Framework, entities are pure POJOs; therefore, we'll also use them as VOs.

1. Service Objects

These are common in technical frameworks. These are also used in the domain layer in domain-driven design. A service object does not have an internal state; the only purpose of it is to provide the behavior to the domain. Service objects provide behaviors that cannot be related with specific entities or VOs. Service objects may provide one or more related behaviors to one or more entities or VOs. It is best practice to define the services explicitly in the domain model.

1. Repository objects

A repository object is a part of the domain model that interacts with storage, such as databases, external sources, and so on, to retrieve the persisted objects. When a request is received by the repository for an object reference, it returns the existing object reference. If the requested object does not exist in the repository, then it retrieves the object from storage.

**REGISTRATION AND DISCOVERY SERVICE**

We need a place where all microservices get registered and can be referenced—a service discovery and registration application. **Spring Cloud** provides the state-of-the-art service registry and discovery application **Netflix Eureka**

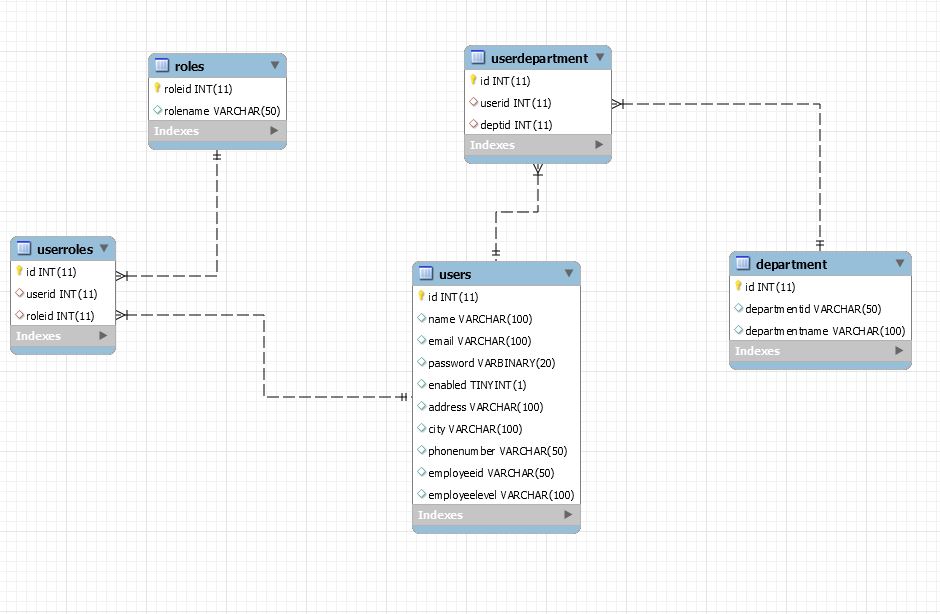
Once you have configured the Eureka service as described in this section, it will be available for all incoming requests to list it on the Eureka service. The Eureka service registers/lists all microservices that have been configured by the Eureka client. Once you start your service, it pings the Eureka service configured in your **application.properties** and once a connection is established, the Eureka service registers the service.

It also enables the discovery of microservices through a uniform way to connect to other microservices. You don't need any IP, hostname, or port to find the service, you just need to provide the service ID to it. Service IDs are configured in the **application.properties** of the respective microservices.

**EUREKA CLIENT**

Similar to the **Eureka server**, each service should also contain the **Eureka client** configuration, so that a connection between the Eureka server and the client can be established. Without this, the registration and discovery of services is not possible.

DATABASE SCHEMA



DATABASE CODES

create database payrollusermanagement;

use payrollusermanagement;

########################################

#create table users

########################################

create table users (

id int primary key auto\_increment,

name varchar (100),

email varchar (100) unique,

password varbinary (20),

enabled boolean,

address varchar (100),

city varchar(100),

phonenumber varchar (50) unique,

employeeid varchar (50) unique,

employeelevel varchar (100)

) engine =InnoDb;

########################################

#create table roles

########################################

create table roles (

roleid int primary key auto\_increment,

rolename varchar(50) unique

) engine =InnoDB;

########################################

#create table userroles

########################################

create table userroles (

id int primary key auto\_increment,

userid int,

roleid int,

foreign key(userid) references users(id) on delete set null,

foreign key(roleid) references roles(roleid) on delete set null

) engine = InnoDB;

########################################

#create table department

########################################

create table department (

id int primary key auto\_increment,

departmentid varchar (50) unique,

departmentname varchar (100)

) engine = InnoDB;

########################################

#create table userdepartment

########################################

create table userdepartment (

id int primary key auto\_increment,

userid int,

deptid int,

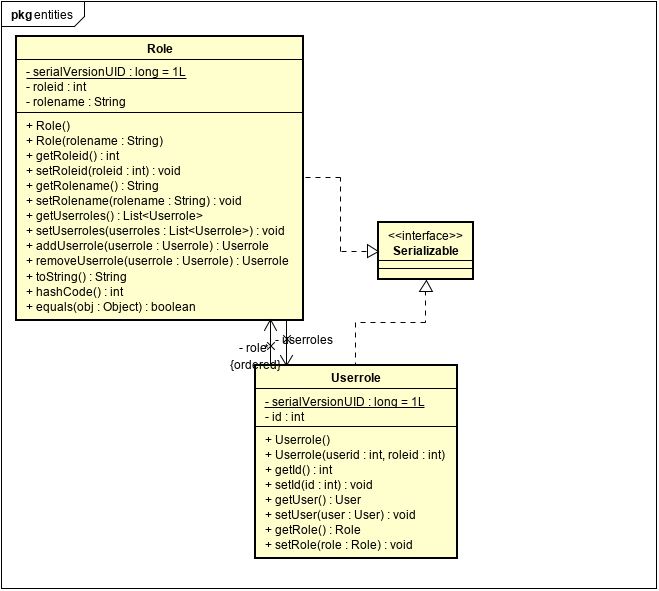
foreign key(userid) references users(id) on delete set null,

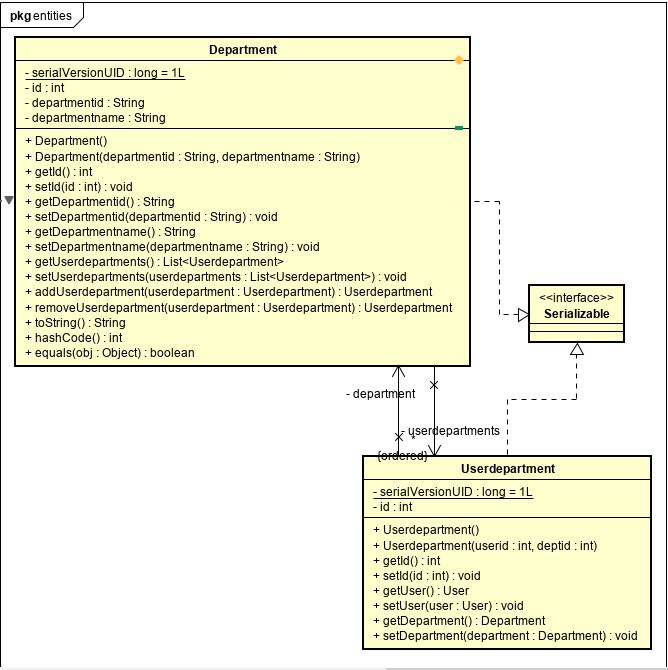
foreign key(deptid) references department(id) on delete set null

) engine = InnoDB;

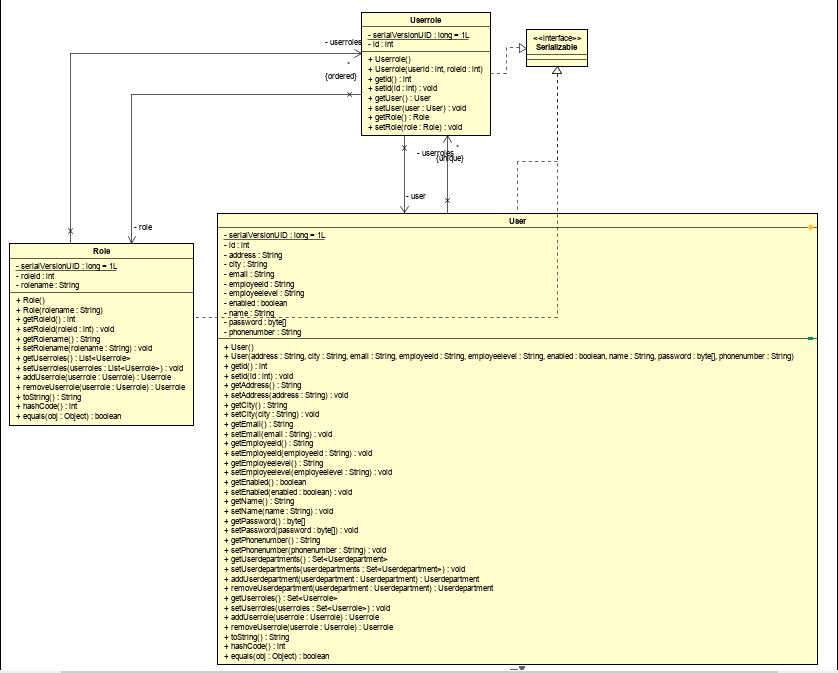
ENTITIES CLASS DIAGRAM

ROLE



DEPARTMENT#

USERROLE



ERRO HANDLING MESSAGES

ON RESTFUL ENDPOINTS

Using IETF specification known as Problem Details for HTTP APIs,

<https://tools.ietf.org/html/draft-nottingham-http-problem-06>

the error response chosen for the application includes:

{  
"title" : "",  
"status" : "",  
"detail" : ",  
"timestamp" : "",  
"developerMessage: "",  
"errors": {}  
}

title—The title field provides a brief title for the error condition. For example,  
errors resulting as a result of input validation will have the title “Validation Failure”.  
Similarly, an “Internal Server Error” will be used for internal server errors.

• status—The status field contains the HTTP status code for the current request.  
Even though it is redundant to include status code in the response body, it allows API  
clients to look for all the information that it needs to troubleshoot in one place.

• detail—The detail field contains a short description of the error. The information  
in this field is typically human-readable and can be presented to an end user.

• timestamp—The time in milliseconds when the error occurred.

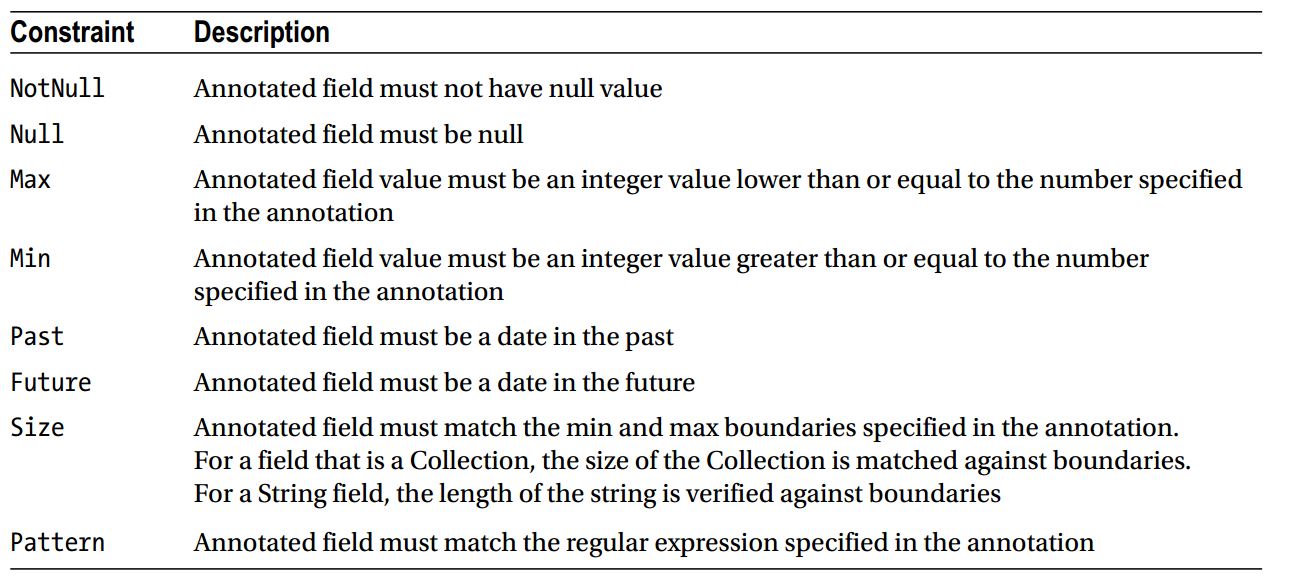
• developerMessage—The developerMessage contains information such as exception  
class name or stack trace that is relevant to developers.

• errors—The errors field is used to report field validation errors.

ON ENTITY CREATION

We use the JSR 303 validation, an API intended to simplify field validation in any layer of the application.  
The JSR 303 and JSR 349 define specifications for the Bean Validation API (version 1.0 and 1.1, respectively).  
They provide a metadata model for JavaBean validation via a set of standardized validation constraints. Using  
this API, you annotate domain object properties with validation constraints such as @NotNull and @Email.  
Implementing frameworks enforce these constraints at runtime. We use Hibernate  
Validator, a popular JSR 303/349 implementation framework. some of the out-of-the-box

validation constraints available with Bean Validation API are:



Additionally, it is possible to define your own  
custom constraints.