

Dive into the Spring Security Architecture



Sohail Shah · [Follow](#)

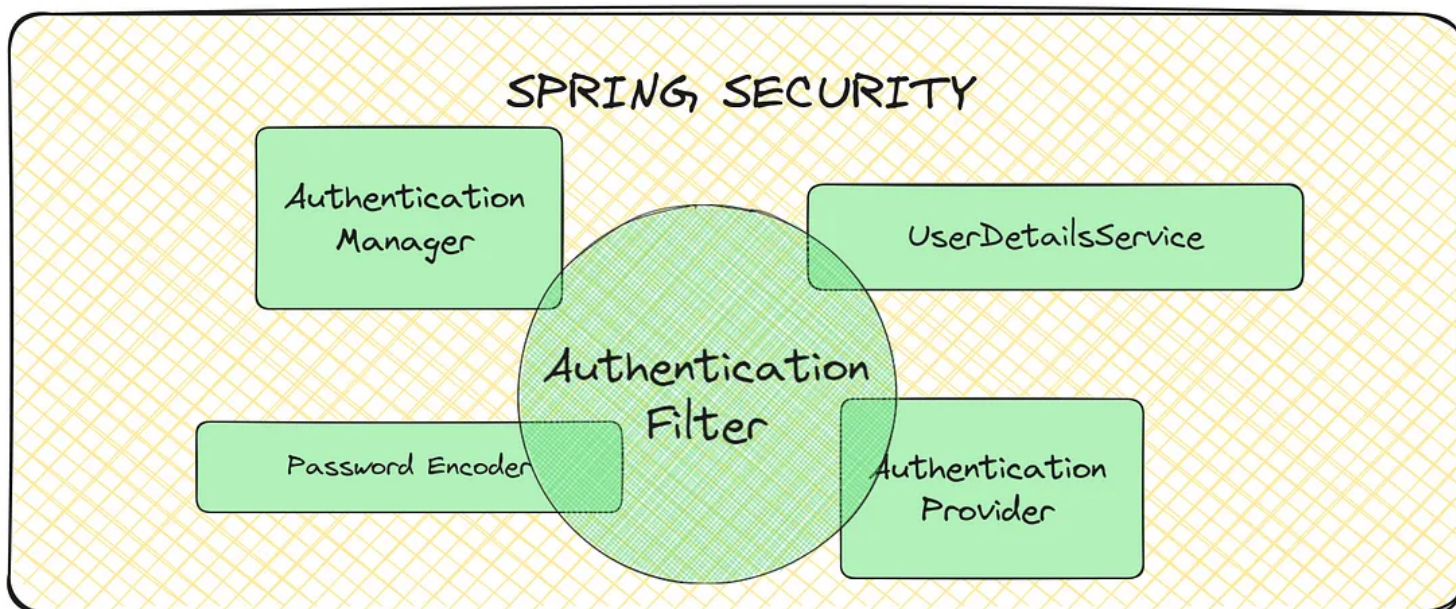
4 min read · Jul 2



Listen



Share



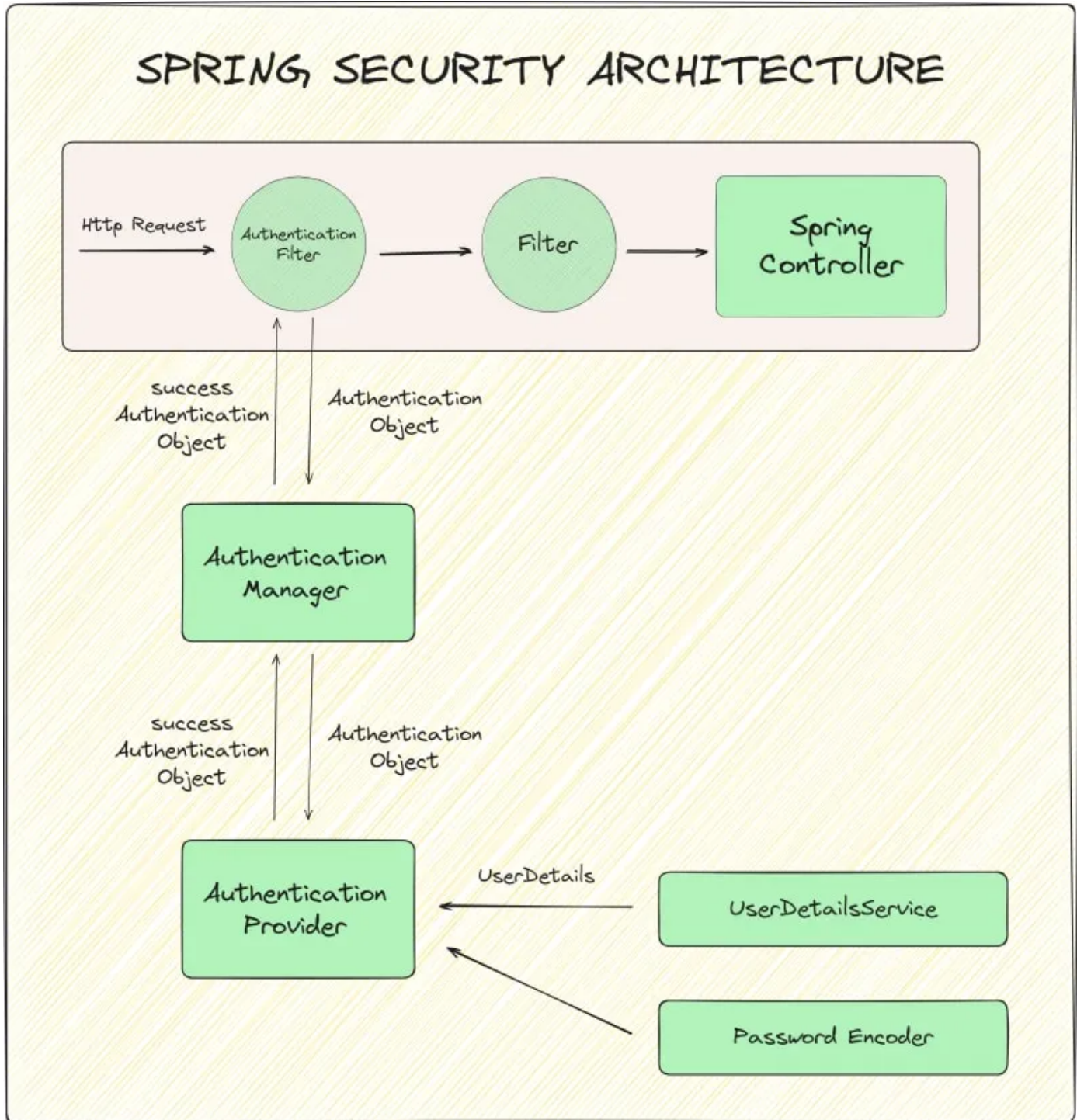
In this post, we'll be looking at the components that form spring security and understand how the spring security architecture works. By getting to know the components of spring security and their working, it becomes easy to configure and implement our own security mechanisms.

Components of Spring Security

The following are the basic components that form the spring security architecture.

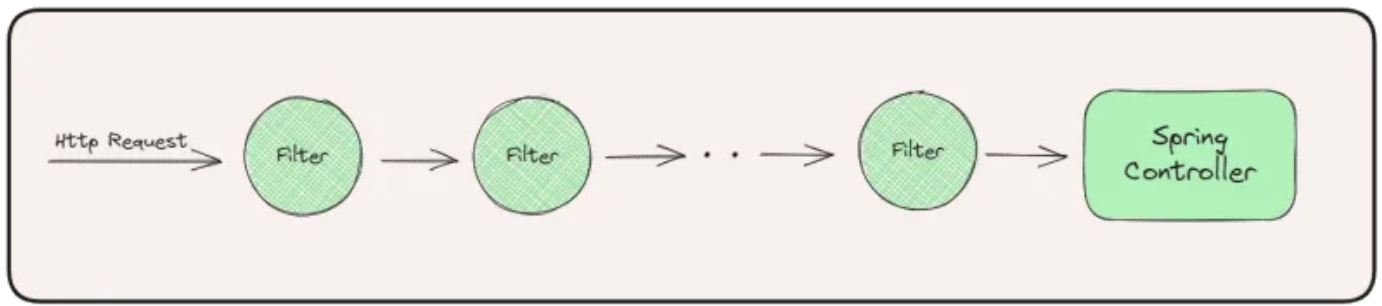
- Filters
- Authentication Manager

- Authentication Provider
- User Details Service
- Password Encoder



Let's talk about each one of them in detail

Filters



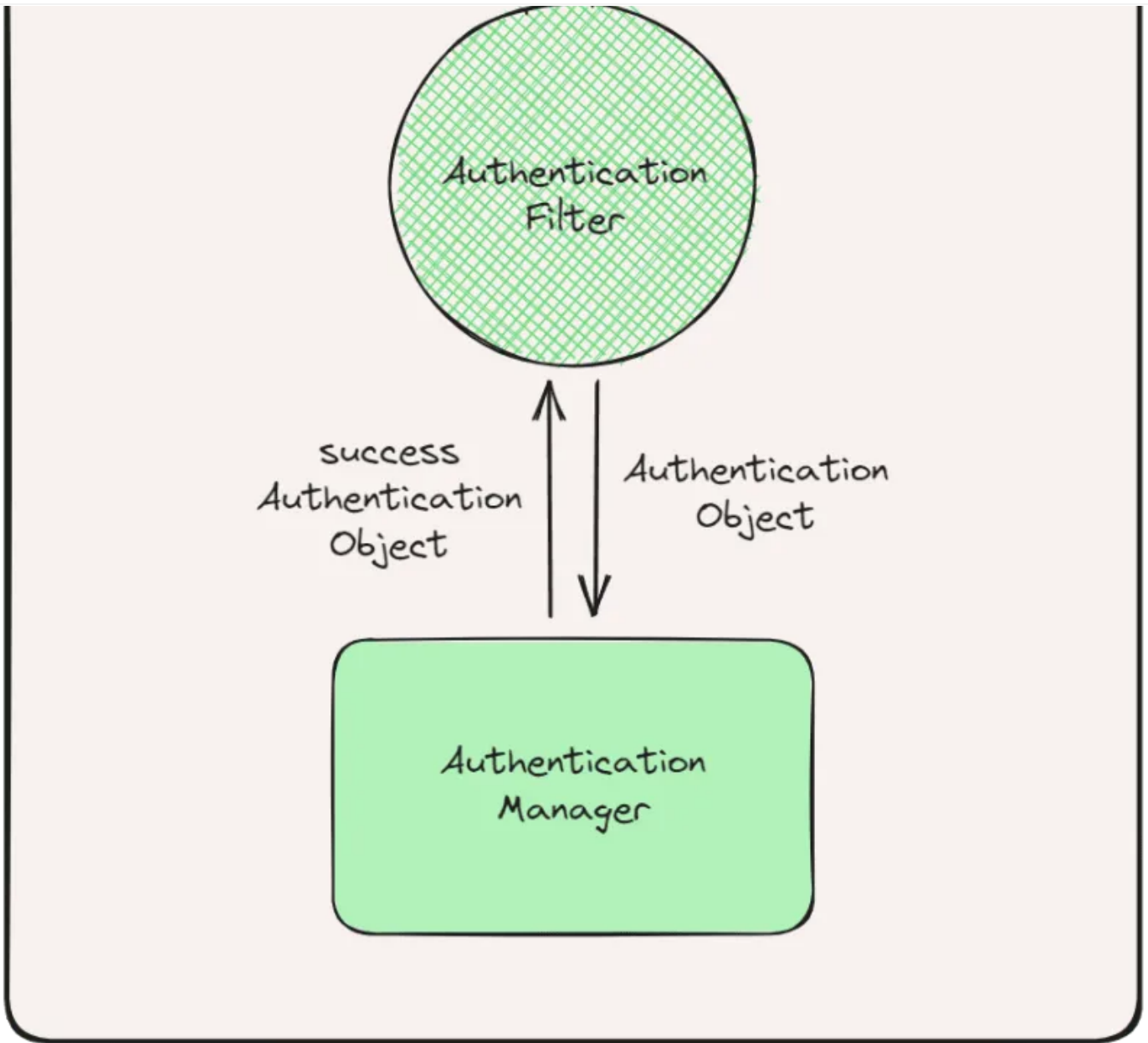
In spring applications, every request needs to pass through a chain of filters before the request ends up in the controller class. These filters are responsible for the authentication and authorization of users and their requests to access resources.

The filters check the validity of each request based on internal rules defined. You can create your custom filters with your own rules.

Let's say the request is in the authentication filter. The authentication filter is responsible for extracting user authentication details and tokens. These user details are packaged as authentication objects and passed to **Authentication Manager**.

The auth object is created using an implementation **Authentication** interface such as `UsernamePasswordAuthenticationToken`.

Authentication Manager



```
public interface AuthenticationManager {  
    Authentication authenticate(Authentication authentication) throws AuthenticationException;  
}
```

The **Authentication Manager** receives an authentication object from the



12

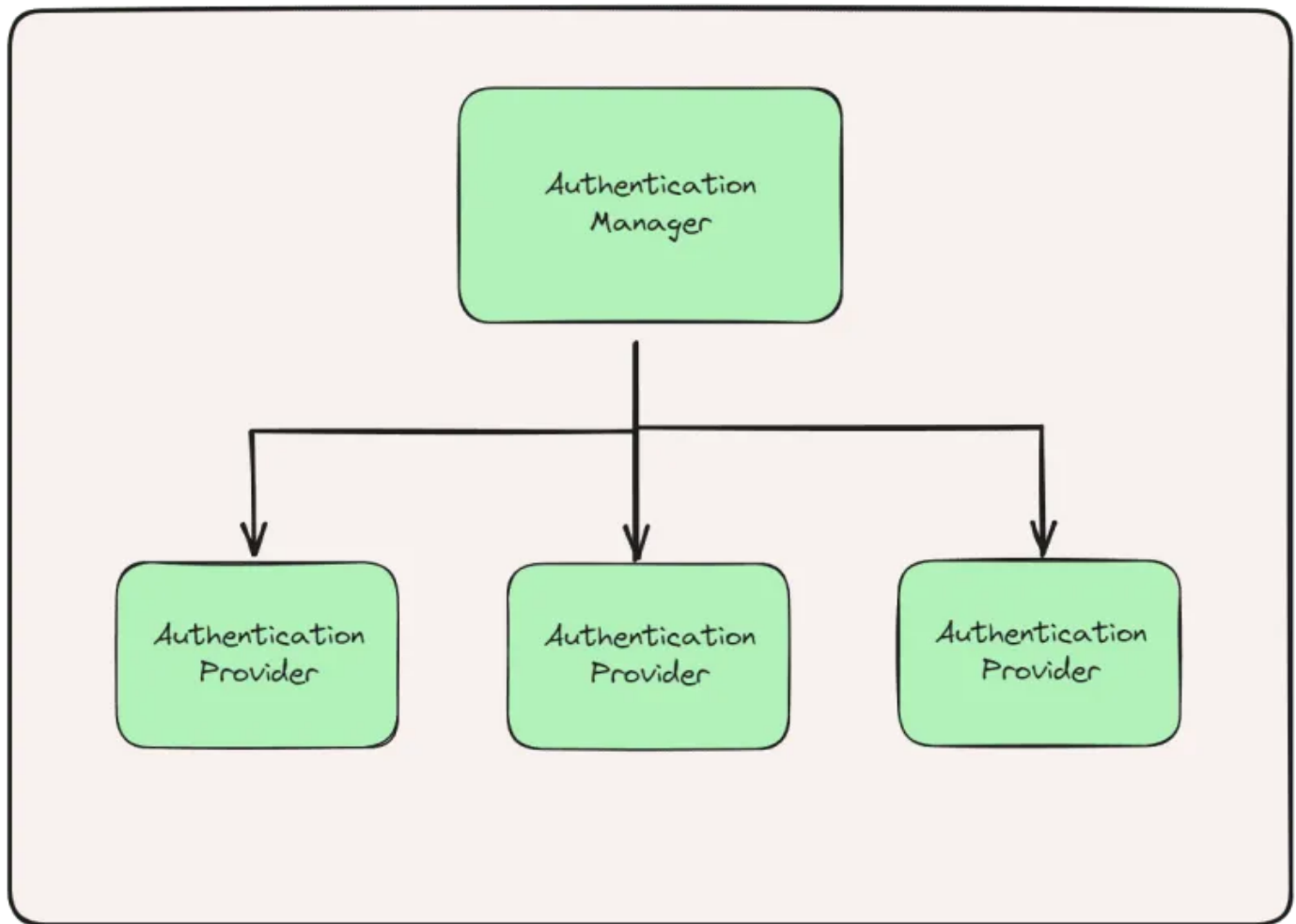


find a way to authenticate users. That is achieved by using **Authentication Provider**.

In a typical spring security configuration, there is only one **Authentication Manager** which delegates the authentication request to the correct **Authentication Provider**.

The **Authentication Manager** interface has only one method called `authenticate` that takes in an authentication object passed down from the authentication filter. Upon successful authentication of the user, the **Authentication Manager** returns an authenticated **Authentication** object.

Authentication Provider



```
public interface AuthenticationProvider {  
    Authentication authenticate(Authentication authentication) throws AuthenticationException;  
    boolean supports(Class<?> authentication);  
}
```

There can be multiple Authentication Providers in a spring security context. Every **Authentication Provider** is responsible for handling different authentication mechanisms. For example, a spring application can have both **username and password** authentication as well as **HttpBasic** authentication. For both of these authentication mechanisms, there will be an **Authentication Provider** implementation.

The Authentication Provider interface has a **authenticate method** like the Authentication Manager and a **supports method**.

The supports method checks if the current provider supports the authentication for the given type of credentials, if not the credentials are passed to the next provider/filter.

The authenticate method actually validates the credentials or tokens of the user. This method gets the user details using the **UserDetailsService** interface which is responsible for retrieving user details from a user store-like database. Upon successful authentication, the method returns an authenticated Authentication object back to the Authentication Manager. At last, this authenticated object is stored by the **Authentication Manager** in the spring security context, which holds the users' authentication information. This information can be accessed throughout the application.

UserDetailsService

The **UserDetailsService** interface has only one method **loadUserByUsername** which takes the username from the authentication object passed down from the filter.

The `loadUserByUsername` method returns a **UserDetails** object which contains user details like username, password, authorities, and other details.

```
public interface UserDetailsService {  
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;  
}
```

```
public interface UserDetails extends Serializable {  
    Collection<? extends GrantedAuthority> getAuthorities();  
    String getPassword();  
    String getUsername();  
    boolean isAccountNonExpired();  
    boolean isAccountNonLocked();  
    boolean isCredentialsNonExpired();  
    boolean isEnabled();  
}
```

Password Encoder

Since we are talking about security, it is very important that the users' passwords are secured for the entirety of the authentication process. Spring Security comes with some de facto password encoders out of the box. This makes it easier for us developers to manage users while authenticating and storing them in the database.

Some popular implementations of the **PasswordEncoder** interface are **BCryptPasswordEncoder**, **SCryptPasswordEncoder**, and **AbstractPasswordEncoder**.

To work with the password encoder, define a bean that returns a type **PasswordEncoder**

```
@Bean  
public PasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}
```

Now you can inject this bean in the user service to encode the password before storing it in the database and in the authentication provider to decrypt the password for validation.

Knowing and understanding these components of spring security helps to implement security better for our applications.

If made any mistakes please point it out in the comments.

Spring

Spring Boot

Spring Security

Java



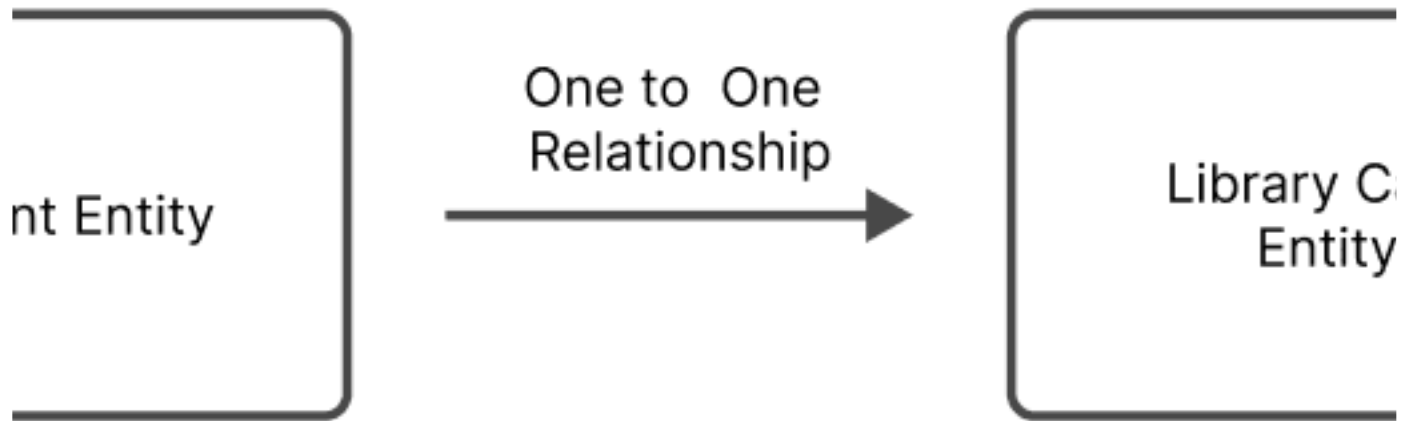
Follow

Written by Sohail Shah

8 Followers

Java/Spring Boot Developer <https://linktr.ee/sohailshah20>

More from Sohail Shah



Sohail Shah

JPA : One-To-One Relationship, Fetch type, Relationship Direction And Query Optimization

In the previous posts, we had only a single entity called the student. Let's create another entity named the library card. The library...

8 min read · Jul 11



2



Simplifying Database Operations with JPA and Hibernate: Creating and Persisting Objects



Sohail Shah · Jun 15



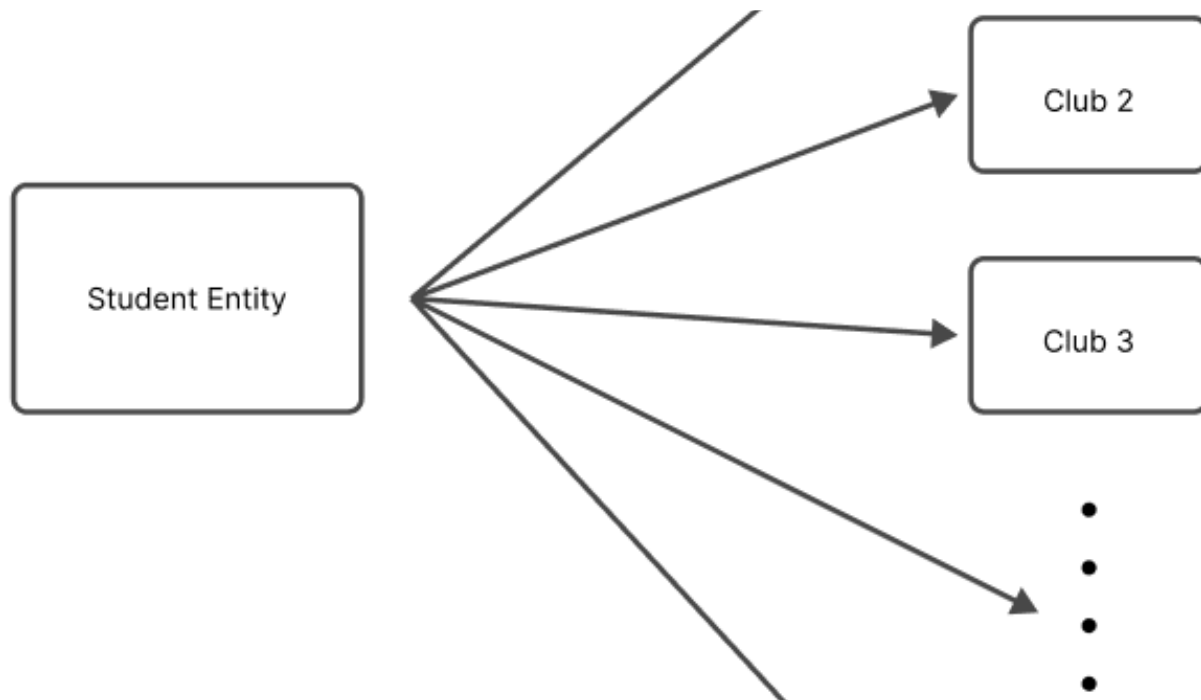
Sohail Shah

Simplifying Database Operations with JPA and Hibernate: Creating and Persisting Objects

In the previous post, we set up a database and JPA configurations, Let's create a Java class Student, and see how to make it work with JPA.

3 min read · Jun 15





 Sohail Shah


JPA : One-To-Many, Many-To-One And Many-To-Many Relationships

In the previous post, we looked at one-to-one relationship. In this post, we will look at one-to-many, many-to-one, and many-to-many...

7 min read · 3 days ago

 1 



 Sohail Shah

Persisting non-primitive types in JPA

In this post, we'll look at how we can persist non-primitive data types in JPA.

4 min read · Jun 25



See all from Sohail Shah

Recommended from Medium



 Syed Habib Ullah


9 Essential Functionalities in Spring Boot: A Comprehensive Guide(2023)

Welcome to this comprehensive guide on the essential functionalities of Spring Boot. As a powerful Java-based framework, Spring Boot offers...

9 min read · Apr 2

 34 



 Abdulkadar

Lombok in Spring Boot

Introduction

4 min read · Jul 18

 1 



Lists



It's never too late or early to start something

13 stories · 37 saves



General Coding Knowledge

20 stories · 99 saves



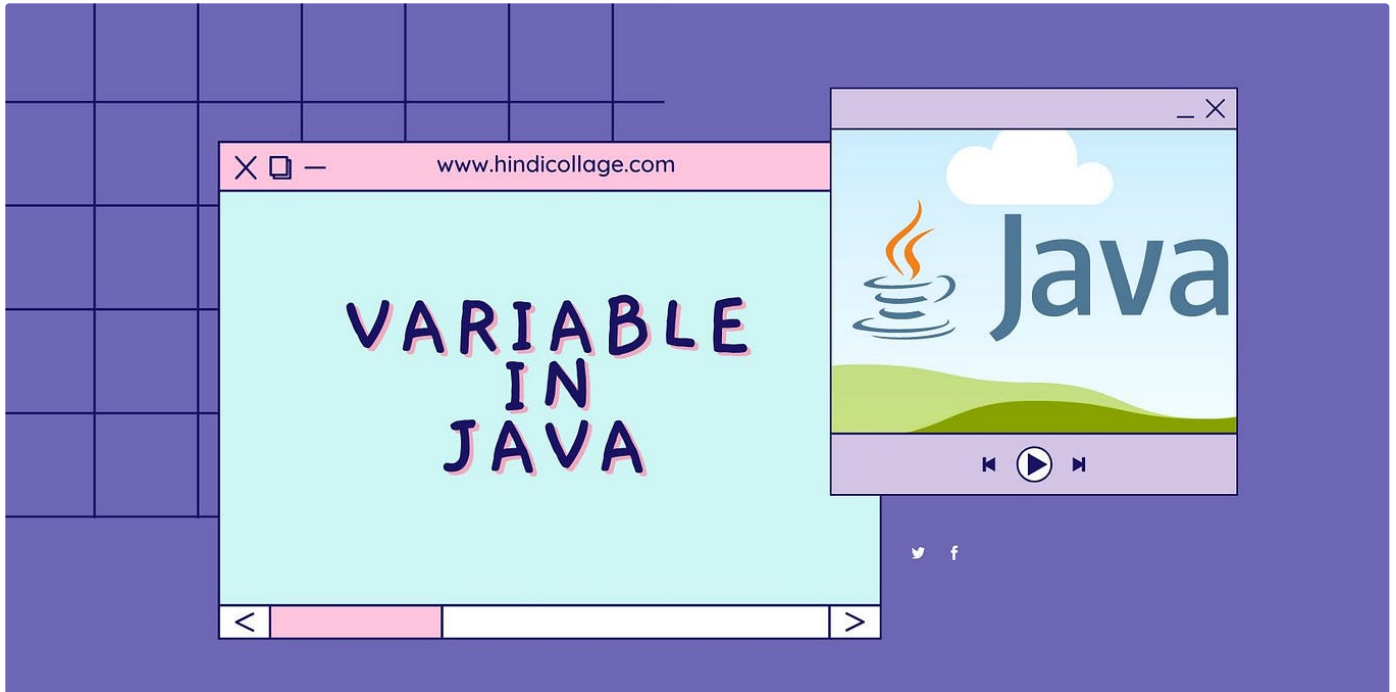
New_Reading_List

174 stories · 32 saves



Staff Picks

408 stories · 170 saves



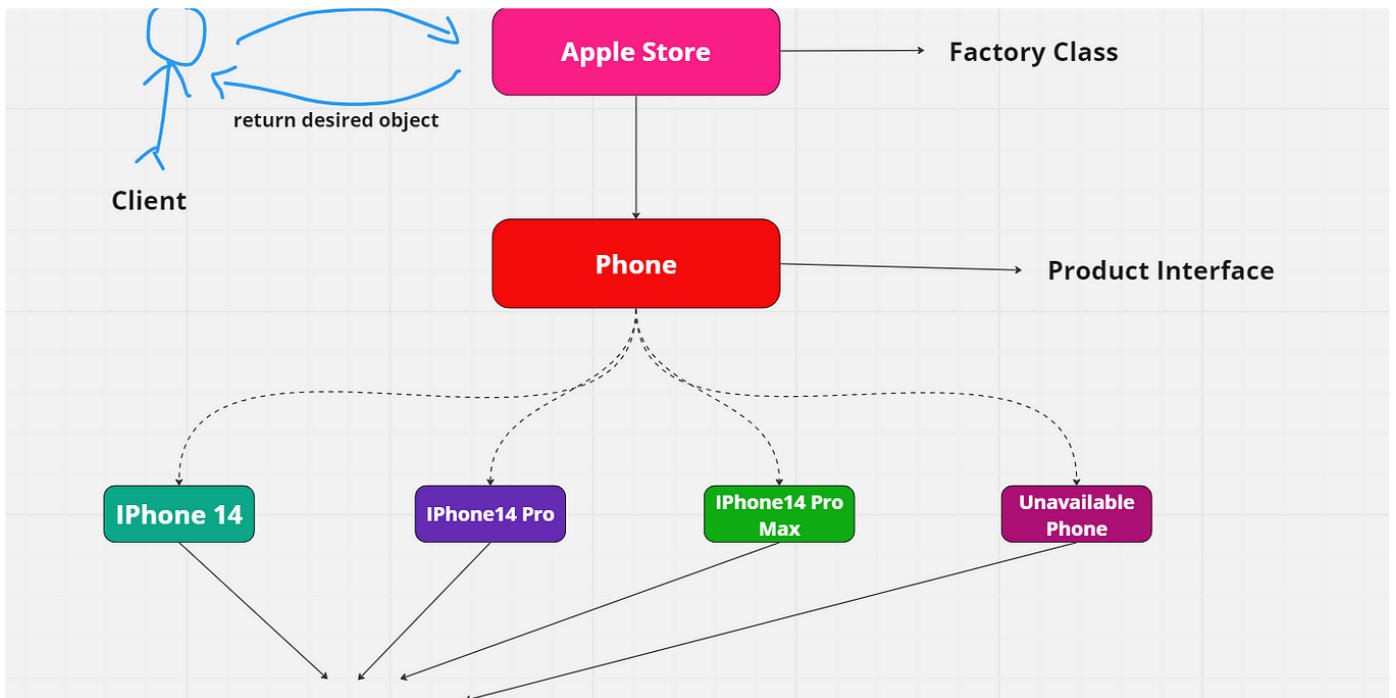
 Ramjee Chaurasiya


Variable in java—what is variable in java

What is variable in java -

4 min read · Jun 15





 Neha Gupta

Factory Design Pattern in Java

This is a complete blog on Factory Design Pattern

4 min read · Jun 2



57



Database Integration in Spring Boot: A Guide to Working with JDBC and JPA

In modern web application development, integrating a database is a crucial aspect. Spring Boot, a popular Java framework, provides...

5 min read · Jul 14



Simplifying Database Operations with JPA and Hibernate: Creating and Persisting Objects



Sohail Shah · Jun 15



Sohail Shah

Simplifying Database Operations with JPA and Hibernate: Creating and Persisting Objects

In the previous post, we set up a database and JPA configurations, Let's create a Java class Student, and see how to make it work with JPA.

3 min read · Jun 15



[See more recommendations](#)