Netbanking Application

Authors: Monil Soni (201601049), Bhavik Mehta (201601223)

Table of contents

Abstract

The goal of this document is to provide a basic understanding of Spring while creating a sample application for Netbanking that covers the following use-cases: Authentication, Mini Statement and Money Transfer. The information presented in this document is not exhaustive and consequently this document is not meant to be treated as a course for Spring. However, it may very well serve as a starting point for understanding some aspects of Spring and how to use them to create a functional application.

More information about the application is mentioned in the design documents that follow. Following that are use-cases for selected features, then some theory of Spring in context to our application is provided. Finally sequence diagrams are shown to see the big picture of how it all works together.

Design Document

User Classes and Characteristics

After analysing the various requirements of the system, we listed down classes which are necessary for such a system. Following are the entities identified and their associated properties (as of Oct 6, 2019). The definition may have been modified from the actual words for system specific usages.

- User: A user is defined as someone who has an account in the concerned bank. A user can view the summary of transactions done as a bank statement, do fund transfer to accounts linked with one of his/her accounts or some other account in the same bank. A user can own multiple accounts in one bank and can access the details for all of them using the same user id.
- Account: A bank account is a financial account maintained by a bank for a customer. All the transactions that a user executes happen on the account.
- Bank
- Transaction: A transaction is an event where a sum of money is transferred from one account to another regardless of the owner being the same or different. The nature of the transaction can be Debit(Dr) or Credit(Cr) relative to the person sending or receiving the money respectively. Each transaction has an id, date, amount and a closing balance associated with it.
- Statement: A bank statement is a summary/collection of transactions from a starting date to an ending date as specified by the requestor. However, statements do not get database entries in this case as they are just query results for the dates provided by the user.

Following are the classes and their corresponding attributes[1]:

- User - id, name, mobile, email, address, username, password
- Account - id, name, opening_date, ifsc_code, customer_id, currency, balance
- Bank - id, name, headquarters
- Transaction - id, details, date, cr_dr, amount, closing_balance

Following are the classes and their corresponding methods:

- User - addUser(), deleteUser(), updateUser(), searchUser(), getAccounts()
- Account - addAccount(), deleteAccount(), updateAccount(), searchAccount(), generateStatement()
- Bank[2] -
- Transaction - addTransaction(), updateTransactionDetails(), searchTransaction()

[1] All the attributes will be prefixed by entity name in small letters plus an underscore (eg: id for User will be written as user_id)

[2] Since we have only one bank here, there will not be any CRUD operations associated with the class

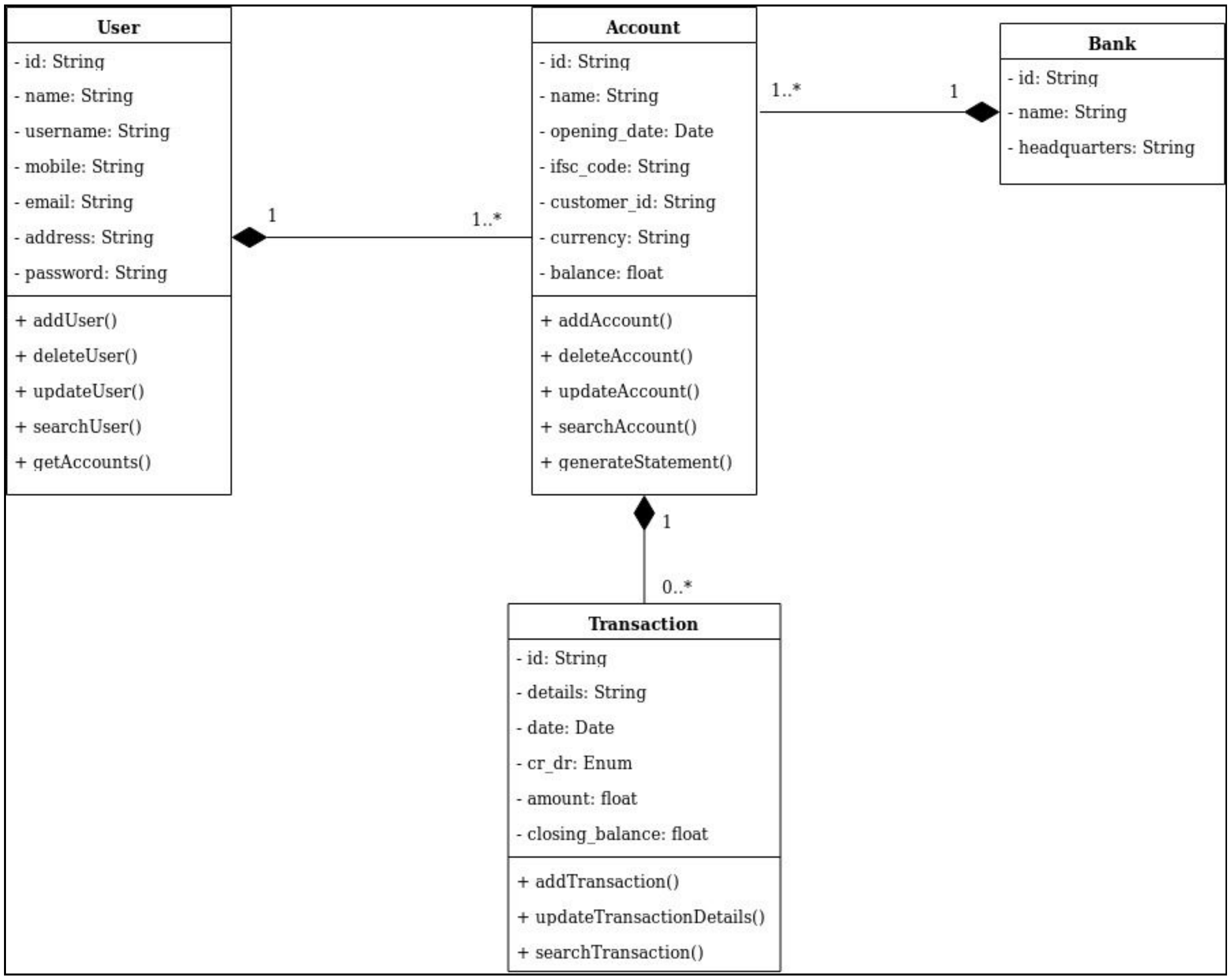


Figure 1: UML Class Diagram

System Features

1. Secure Login/Logout

   Description:
   This feature allows user to safely log into the system as well as unintended people to stay out of the system. Moreover, once you are logged in to the system, you will be able to logout from any page. This is a high priority.

   Response:
   When a user clicks the login button, a new window opens and he is asked to enter his user id and password. If the password is correct, he is redirected to his Dashboard, else he is shown a message saying “Username or password is incorrect.” When the user clicks the logout button, he will be redirected to the login page and will be shown a message saying “You have been logged out.”

   Functional Requirements:
   -

2. Personal Dashboard

   Description:
   Personal dashboard is a single page where the user can see the accounts associated with his username. These accounts are listed in the form of a table where one account occupies one row. At the end of the row two cells with links to “Account Transfer” and “Mini Statement” are present.
   A user is directed to his dashboard after a successful login. The bare bones of this feature is a high priority.

   Response:
   On clicking the links for the features mentioned, the user should be directed towards the respective features.

   Functional Requirements:
   -

3. Funds Transfer

   Description:
   User can transfer money to any account in the same bank. This is again a high priority. A user needs to enter the account number to which the money is being transferred, the amount to be transferred and a detail message.

   Response:

The balance gets updated in both the accounts and he will be able to see the transaction in the summary feature. When the user clicks the transfer button, he is redirected to his dashboard, or else an error message will be shown to the user.

Functional Requirements:
Transfer must be an atomic operation so to speak in order to avoid inconsistencies in the database.

4. Mini Statement

   Description:
   This feature allows user to get details of the last 10 transactions. This is a low priority.

   Response:
   When a user clicks the "Mini Statement" link on the dashboard, he is redirected to a page where the transactions are displayed.

   Functional Requirements:
   -

Use Case

Authentication

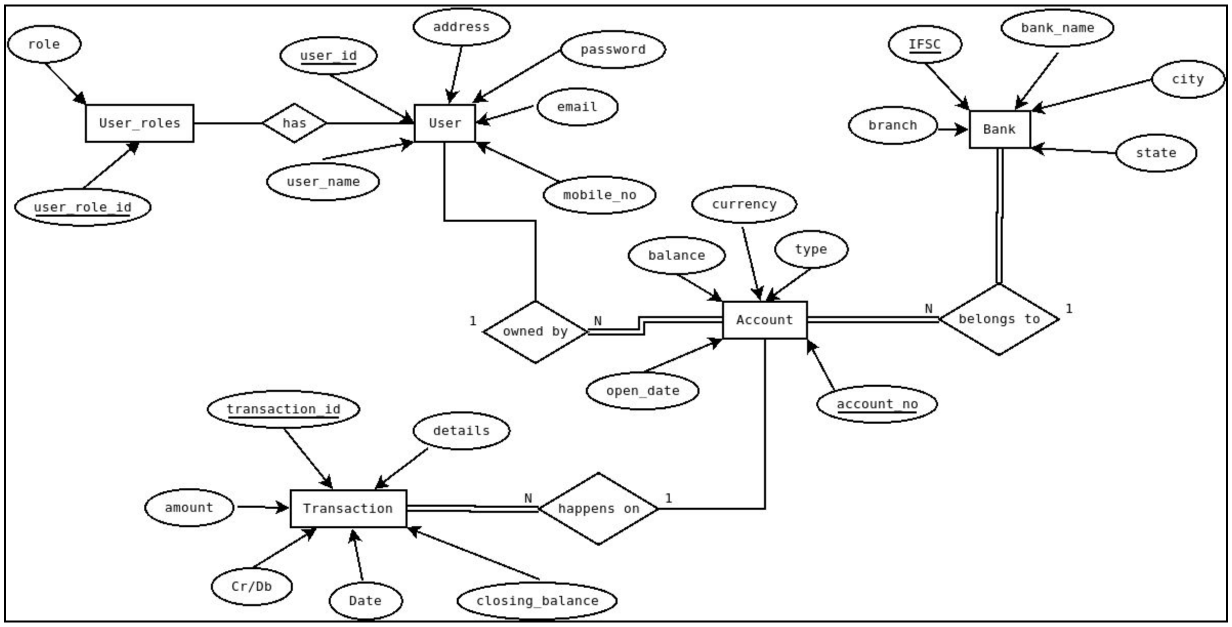
Use Case Name	Log In
Goal:	To log into the system.
Actor(s):	Netbanking user
Preconditions:	The actor is registered in the system
Main flow of events:	1. The system requests the user to enter his username and password. 2. The user enters his credentials. 3. The system validates the credentials and logs the user into the system.
Alternative flow of events:	Invalid Name/Password: If the user enters an invalid name and/or password, the system displays an error message. The user can choose to either return to the beginning of the main flow or cancel the login, at which point the use case ends.
Post conditions:	If the use case was successful, the user is now logged into the system, else the state remains the same.

Use Case Name	Log Out
Goal:	To log out of the system.
Actor(s):	Netbanking user
Preconditions:	The actor is logged in the system
Main flow of events:	1. The user clicks the logout button. 2. The system logs the user out.
Alternative flow of events:	None
Post conditions:	If the use case was successful, the user is now logged out of the system, else the state remains the same.

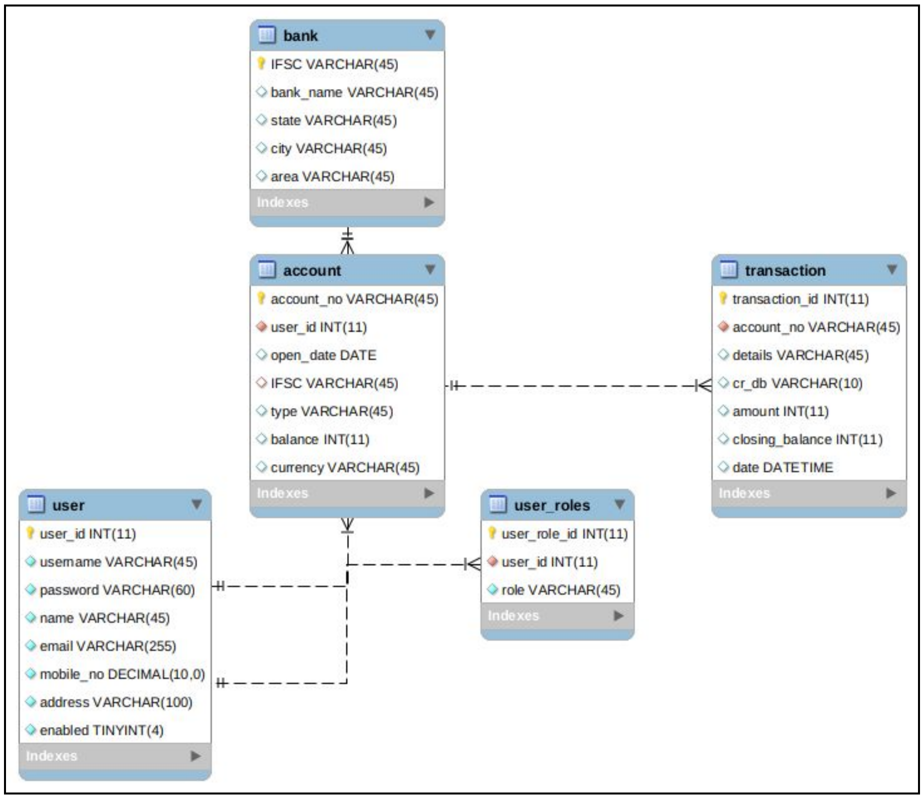
Money Transfer

Use Case Name	**Money transfer**
Goal:	To transfer money into an account
Actor(s):	Netbanking user
Preconditions:	The actor is registered in the system.
Main flow of events:	1. The system requests the user to enter account no. of to whom money is being transferred, amount to be transferred and message. 2. The user enters this details. 3. The system validates this details and transfers the amount.
Alternative flow of events:	<u>Invalid Account No.:</u> If the user enters an invalid account the system clears the account no. and displays an error message that account no. is valid, at this point the use case ends. <u>Invalid Amount:</u> If the user enters the amount more than the current balance, the system displays clears the amount textbox and displays error that user does not have sufficient balance, at this point the use case ends.
Post conditions:	If the use case was successful, the user will be on the dashboard/home page.

Database Diagrams

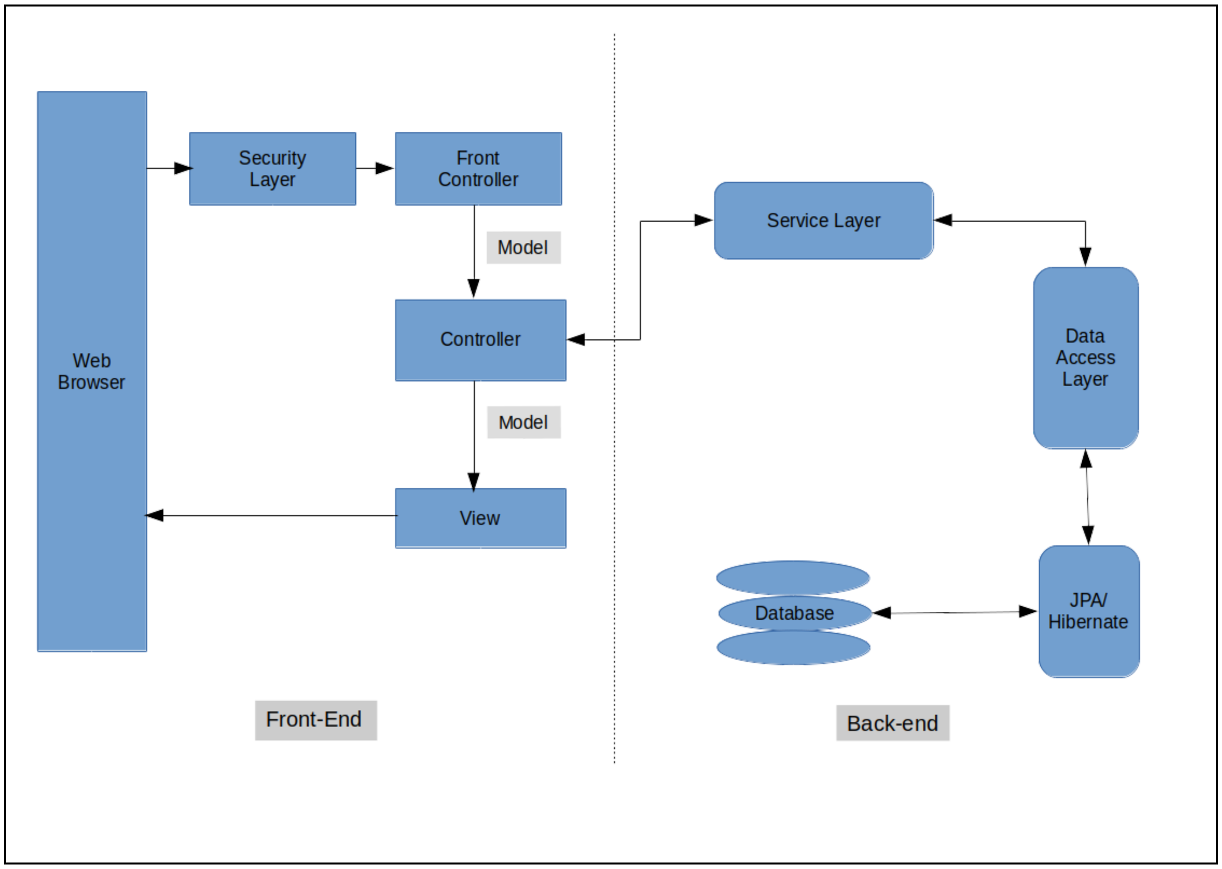


[Figure 2: ERD]



[Figure 3: Relational Diagram]

Overview of Application's Working



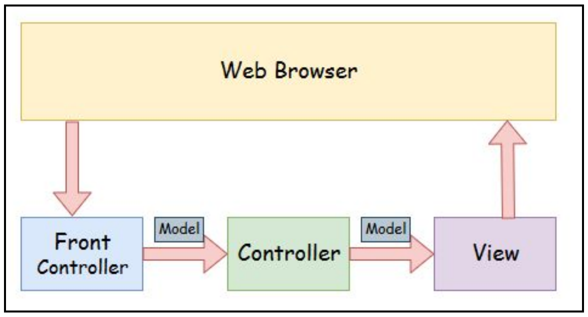
[Figure 4: Overview of application's working]

The application is divided into three parts: Spring MVC, Spring Security and JPA. This document will be explaining each of them as individual topics in the following pages.

Spring MVC

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

Spring MVC flow



[Figure 5: Spring MVC Flow]

1. **Front Controller :** It handles the HTTP requests and response coming from web browser. In Spring MVC, *dispatcher-servlet.xml* acts as a Front Controller and it is already implemented for the developers.
2. **Model :** It handles the passing of data between front-end and Controller. This data can be a single object or collection of an object.
3. **Controller :** It handles the actual implementation of requests and responses that are handed over by the Front Controller. Controller is the java class where all the business logic resides.
4. **View :** It handles the UI of the application. View is typically a JSP+JSTL page although Spring provides other frameworks too like thymeleaf, velocity etc.

So typically, the web browser sends a request in the form of URL, front controller identifies this request and sends to Controller via an object of Model. The Controller has the request mapping in the form that there is a method implemented for each expected request. So, the method which is mapped to this URL will be executed. Now, these methods will always have return type of String. This string is the name of JSP page(View) that you want to display in the response of that request. Now, if one wants to transfer some information residing in Controller to the View, it can be done using Model class. This is the typical flow of web application implemented in Spring MVC.

// Write about Model class if time remains

Annotations

Java Annotation is the meta-data about the program. It can be parsed by the annotation parsing tool or by the compiler.

1. **`@Controller`**
   Indicates that an annotated class is a "Controller" (e.g. a web controller). This annotation serves as a specialization of `@Component`, allowing for implementation classes to be autodetected through classpath scanning. Moreover, once a class is declared as a controller, one can declared annotated handler methods by the use of `@RequestMapping` annotation.

2. **`@Service`**
   The business logic resides in the classes annotated by `@Service`. This annotation, like `@Controller`, is a specialization of `@Component` and hence classes annotated with it are also autodetected through classpath scanning.

3. **`@RequestMapping`**
   Annotation for mapping web requests onto methods in request-handling classes with flexible method signatures. Moreover, it can also be declared at a controller level, hence providing a greater organization of requests and corresponding controllers. For example in a school application, controllers with methods for students and controllers with methods for faculties can be separated and annotated with `@RequestMapping(“/students”)` and `@RequestMapping(“/faculties”)` respectively so that all the requests directed to /students/something can be handled by the students controller and similarly for faculties.

4. **`@RequestParam`**
   This annotation is used to read the form data and bind it automatically to the parameter present in the provided method.

5. **`@Autowired`**
   `@Autowired` is a new style of using Dependency Injections. This annotation allows Spring to resolve and inject beans into your bean. Once enabled, this style of injection can be used on fields, setters and constructors.

6. **`@Repository`**
   Indicates that an annotated class is a "Repository", originally defined by Domain-Driven Design (Evans, 2003) as "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects". This annotation is also applicable to DAO classes as is the case with the Netbanking application. As of Spring 2.5, this annotation also serves as a specialization of `@Component`, allowing for implementation classes to be autodetected through classpath scanning.

7. **`@Transactional`**
   The transactional annotation itself defines the scope of a single database transaction. At the class level, this annotation applies as a default to all methods of the declaring class and its subclasses.

References

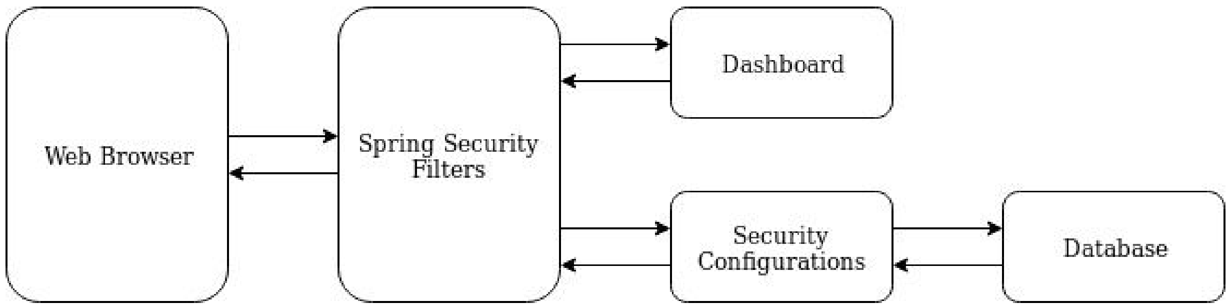
[1] https://docs.spring.io/spring/docs/5.2.1.RELEASE/spring-framework-reference/web.html#mvc
[2] https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm

Spring Security

Spring Security is a Java EE framework that provides authentication, authorization and other security features for enterprise applications. Authentication is the process of identifying and verifying the identity of the user trying to access the application. Authorization is the process of allowing authorities to perform certain restricted actions; for example something which has a clearance level of administrative authorities should not be visible to the normal users. Spring Security is an attractive option because of its seamless integration with an existing Spring MVC application. Few select features of Spring Security are:

- Single sign-on for accessing multiple applications.
- Remember-me feature to avoid login again from the same machine until the user logout by using HTTP cookies.
- Web-form authentication where the application collects and authorizes credentials from the web browser.

It also allows authentication by accessing user credentials from the database using JDBC, Cross Site Request Forgery (CSRF) protection and is completely Java configurable. Following figure shows how Spring Security operates at a higher level:



[Figure 6: Spring Security in action[3]]

Theory

Goal: Our intent is to add authentication to an existing Spring MVC application. We will be using Web-form authentication and verify the credentials from database using JDBC based authentication.

In its most vanilla form of Spring Security requires username, password and roles for authentication. However, we also have additional data for each user like name, email, phone number, address etc. Hence

[3] Created using draw.io

we need to understand how Spring Security works under the hood to provide authentication. We also need to setup default login page, default login processing url and adding the logout functionality. [4]

Following steps depict how authentication process takes place in Spring Security.

1. The username and password are obtained and combined into an instance of `UsernamePasswordAuthenticationToken`.
2. The token is passed to an instance of `AuthenticationManager` for validation.
3. The `AuthenticationManager` returns a fully populated Authentication instance on successful authentication.
4. The security context is established by calling `SecurityContextHolder.getContext().setAuthentication(...)`, passing in the returned authentication object.

`Authentication` is an object used to represent the principal using the application in a Spring Security specific manner. The principal is just an `Object`. Now, in order to build `Authentication` we require another object called `UserDetails`. It provides the necessary information from your application's DAOs or other source of security data. `UserDetails` is a core interface in Spring Security. The official Spring Security Reference gives a very interesting example: "Think of `UserDetails` as the adapter between your own user database and what Spring Security needs inside the `SecurityContextHolder`[5]". Hence, we will use UserDetails by casting it to our custom User object so that we can call our methods of `getName()`, `getEmail()` etc.

In order to give this `UserDetails` object to Spring Security, we make use of an interface called `UserDetailsService`. As the reference says, the only method on this interface is `loadUserByUsername(String username)` and returns a `UserDetails` object. This service is purely a DAO for user data and performs no other function than to supply that data to other components within the framework.

Now we look at `AuthenticationManager` because it seems to be the one that is responsible for authenticating the users. `AuthenticationManager` is an interface whose default implementation (called `ProviderManager`) delegates the task of handling the authentication request to a list of configured `AuthenticationProviders`. The most common approach to verifying an authentication request is to load the corresponding `UserDetails` and check the loaded password against the one that has been entered by the user. As discussed above, on successful authentication, this `UserDetails` object is used to build a fully populated `Authentication` object and is stored in `SecurityContext`. Hence we can configure the `UserDetailsService`, whose only method is supposed to return a `UserDetails` object, as an `AuthenticationProvider`.

Flow till now: A user who wants to access the site enters his credentials. `AuthenticationManager` calls the `loadUserByUsername` method implemented in `CustomUserDetailsService` and passed the entered username as a parameter. This method interacts with the DAO that we have already created to

[4] Note that this is not an exhaustive details of elements of Spring Security architecture. Details which have not been used have been eliminated. However, the mentioned content is a sufficient introduction to making a Netbanking Application or something of a similar kind.
[5] SecurityContextHolder is where we store details of the present security context of the application, which includes details of the principal currently using the application.

fetch the user with the given username. On finding the username, we create our `CustomUserDetails` object which is an extension of our custom `User` class and an implementation of `UserDetails` interface. This object, when returned to `AuthenticationManager`, is used to verify the entered credentials. On successful authentication, our `CustomUserDetails` object is used to populate the `Authentication` object so that we can get the user anywhere in the application. Note that this object is an extension of the custom `User` and hence we can call business-specific methods on it.

Implementation

Hence, so far we have identified that we need to implement `UserDetails`, `UserDetailsService` interfaces in the project. We also need to setup a user_roles table as Spring Security requires the user roles in order to facilitate authorization[6]. It takes these roles or "authorities" and uses it to populate the `Authentication` object.

Once we have our required tables and classes set up, we set up the configuration files. We have two files to make changes in - web.xml and security.xml (this file name can be changed).
We add the following snippet to the web.xml. We do this in order to setup Spring Security servlet filter as shown in figure 2.1 as explained in the comment with the code:

```
<!-- DelegatingFilterProxy looks for a Spring bean by the name of filter
(springSecurityFilterChain) and delegates all work to that Bean. This is how
the Servlet Container can get a Spring Bean to act as a Servlet Filter. -->
<filter>
      <filter-name>springSecurityFilterChain</filter-name>
      <filter-class>org.springframework.web.filter.DelegatingFilterProxy</fil
ter-class>
</filter>
<filter-mapping>
      <filter-name>springSecurityFilterChain</filter-name>
      <url-pattern>/*</url-pattern>
</filter-mapping>
```

The following snippet sets up our `CustomUserDetailsService` which is declared with `@Service("customUserDetailsService")` annotation as our `AuthenticationProvider`.

```
<authentication-manager>
       <authentication-provider user-service-ref="customUserDetailsService">
       </authentication-provider>
</authentication-manager>
```

The `<http>` element attributes control some of the properties on the core filters. Hence, we use that to setup login URL, login processing URL and logout URL. `<intercept-url>` is shown as a demonstration of restricting URLs with authorities and if the clearance is not given, the user is redirected to the URL specified in `<access-denied-handler>` element as shown in the snippet below:

[6] The database scripts can be found in the associated GitHub repository, the diagrams of which have already been shown above.

```
<http use-expressions="true">
       <intercept-url pattern="/" access="hasRole('USER')"/>
       <form-login login-page="/showMyLoginPage"
             login-processing-url="/authenticateTheUser" />
       <logout/>
       <access-denied-handler error-page="/access-denied"/>
</http>
```

Rest of the implementation details are pretty straight forward and can be understood by viewing the code. Following are the links:

- UserDetails interface implementation
- UserDetailsService interface implementation
- user and user_roles tables creation script
- DAO implementation for user table
- DAO implementation for user_roles table

Once these files are set up, login page also needs to be changed. Instead of `<form>` tag, `<form>` tag present in the Spring's form tag library is to be used. This will automatically send the CSRF token with it. If one wants to use the html `<form>` tag, then he has to supply the CSRF token as a hidden input with an attribute named "`_csrf`". Note that the token must also be provided along with the logout button and hence the Spring's form tag library can also be used there.

References

https://docs.spring.io/spring-security/site/docs/5.2.2.BUILD-SNAPSHOT/reference/htmlsingle/
https://www.javatpoint.com/spring-security-introduction

Java Persistence API (JPA)

The Java Persistence API (JPA) is a specification of Java. It is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.

As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.

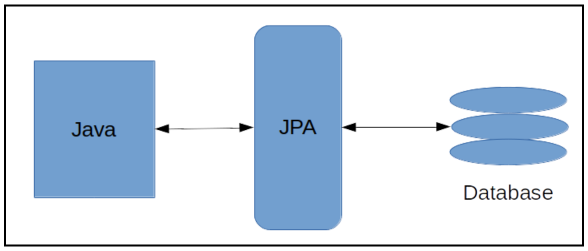


Figure 7: Architectural placement of JPA

It is important to distinguish between JDBC (Java Database Connector) and JPA. JDBC is a standard for connecting to a DB directly and running SQL against it. Data sets can be returned which you can handle in your app, and you can do all the usual things like INSERT, DELETE, run stored procedures, etc. One of the issues with traditional JDBC apps is that you can often have some crappy code where lots of mapping between data sets and objects occur, logic is mixed in with SQL, etc. While JPA is a technology which allows you to map between objects in code and database tables. This can "hide" the SQL from the developer so that all they deal with are Java classes, and the provider allows you to save them and load them magically. Although, under the hood, Hibernate and most other providers for JPA write SQL and use JDBC to read and write from and to the DB.

References

https://www.javatpoint.com/jpa-tutorial

Hibernate ORM

Hibernate ORM enables developers to more easily write applications whose data outlives the application process. As an Object/Relational Mapping (ORM) framework, Hibernate is concerned with data persistence as it applies to relational databases (via JDBC). Hibernate is also an implementation of the Java Persistence API (JPA) specification. As such, it can be easily used in any environment supporting JPA including Java SE applications, Java EE application servers, Enterprise OSGi containers, etc. Hibernate consistently offers superior performance over straight JDBC code, both in terms of developer productivity and runtime performance.

Advantages of Hibernate

- Open Source and Lightweight
- Fast Performance
- Database Independent Query
- Automatic Table Creation
- Simplifies Complex Join
- Provides Query Statistics and Database Status

Hibernate Architecture

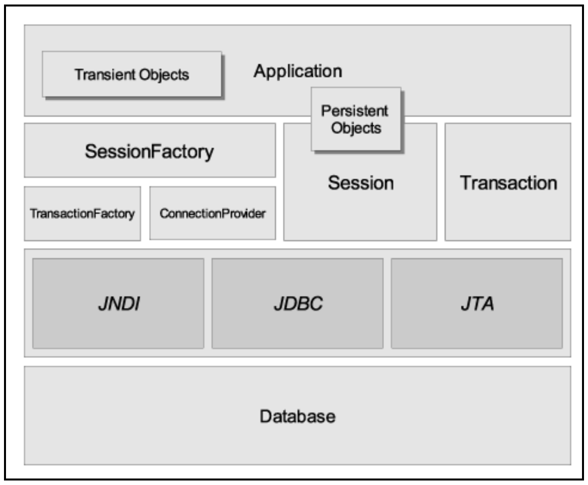


Figure 8: Hibernate Architecture

Hibernate framework uses many objects such as session factory, session, transaction etc. alongwith existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface). Each one of them is summarised below :

1. **ConnectionProvider :** It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource.

2. **Session Factory :** It is a factory of session and client of ConnectionProvider. The *org.hibernate.SessionFactory* interface provides factory method to get the object of Session.
3. **Session :** The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. The *org.hibernate.Session* interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.
4. **Transaction :** The transaction object specifies the atomic unit of work. The *org.hibernate.Transaction* interface provides methods for transaction management.

Hibernate LifeCycle

The Hibernate lifecycle contains the following states: -

- Transient state
- Persistent state
- Detached state

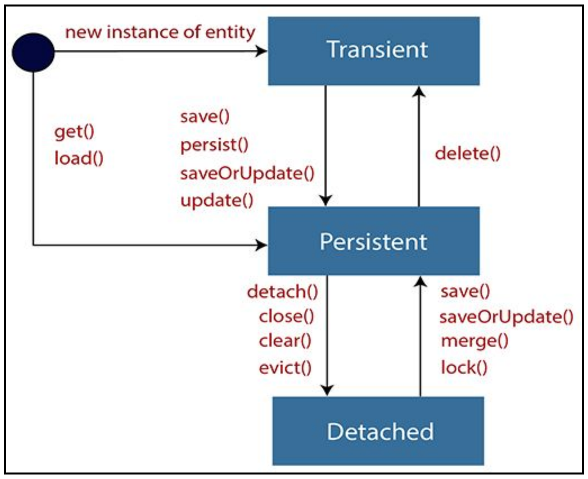


Figure 9: Hibernate Lifecycle

Transient state

The transient state is the initial state of an object. Once we create an instance of POJO class, then the object entered in the transient state.Here, an object is not associated with the Session. So, the transient state is not related to any database. Hence, modifications in the data don't affect any changes in the database. The transient objects exist in the heap memory. They are independent of Hibernate.

```
//Transient state
Transaction transaction = new Transaction(); //POJO class
```

```
transaction.setAccountNo(“1234567”);
```

Persistent state

As soon as the object associated with the Session, it entered in the persistent state. Hence, we can say that an object is in the persistence state when we save or persist it. Here, each object represents the row of the database table. So, modifications in the data make changes in the database.

```
//Persistent state
session.save(transaction);
```

Detached state

Once we either close the session or clear its cache, then the object entered into the detached state. As an object is no more associated with the Session, modifications in the data don't affect any changes in the database. However, the detached object still has a representation in the database. If we want to persist the changes made to a detached object, it is required to reattach the application to a valid Hibernate session.

```
//Detached state
session.detach(transaction);
```

Hibernate Advanced Mapping

This is a widely used feature of Hibernate. This feature is used to cover the foreign , primary key, proportionality of relation concepts of the RDBMS. There are 4 types of mappings :

1. One to One Mapping
2. Many to One Mapping
3. One to Many Mapping
4. Many to Many Mapping

One to Many and Many to One are used interchangeably. Apart from this, there are two types of association in these mappings:

1. Unidirectional
2. Bi-directional

Annotations of Hibernate/JPA

We will cover only annotations that are used in our project. All these annotations are part of javax.persistence package (JPA).

1. **`@Entity` :** It marks the class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.
2. **`@Table` :** allows you to specify the details of the table that will be used to persist the entity in the database.
3. **`@Id` :** Each entity bean will have a primary key, which you annotate on the class with the `@Id` annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.
4. **`@GeneratedValue` :** The `@Id` annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the `@GeneratedValue` annotation, which takes two parameters strategy and generator.
5. **`@Column` :** It is used to specify the details of the column to which a field or property will be mapped.

Following code snippet covers the usage of all these annotations

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "account")
public class Account {

      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      @Column(name = "account_no")
      private String accountNo;

      @Column(name = "open_date")
      private Date openDate;

      @Column(name = "IFSC")
      private String IFSC;
```

```
    @Column(name = "type")
    private String type;

    @Column(name = "balance")
    private long balance;

    @Column(name = "currency")
    private String currency;

    @Column(name = "user_id")
    private long userId;
```

Hibernate Query Language (HQL)

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries, which in turn perform action on database.

Following is one of the Hibernate Query we used in our web-application

```
Query<Transaction> query = session.createQuery("from Transaction where
accountNo = :accountNo order by date desc", Transaction.class);

query.setParameter("accountNo", accountNo);
query.setMaxResults(10); //limit 10

List<Transaction> result = query.getResultList();
```

The SQL equivalent of above query is

```
Select * from transaction where accountNo = <accountNo> order by date desc
limit 10
```

References

https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/architecture.html
https://www.javatpoint.com/
https://www.tutorialspoint.com/hibernate/hibernate_annotations.htm

Log4j

Log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License. Log4j has been ported to the C, C++, C#, Perl, Python, Ruby, and Eiffel languages. Log4j is highly configurable through external configuration files at runtime. It views the logging process in terms of levels of priorities and offers mechanisms to direct logging information to a great variety of destinations, such as a database, file, console, UNIX Syslog, etc. Following are some of the features of Log4j:

- It is thread-safe.
- It is optimized for speed.
- It supports multiple output appenders per logger.
- Logging behavior can be set at runtime using a configuration file.
- It uses multiple levels, namely ALL, TRACE, DEBUG, INFO, WARN, ERROR and FATAL.
- The format of the log output can be easily changed by extending the Layout class.
- The target of the log output as well as the writing strategy can be altered by implementations of the Appender interface.
- It is fail-stop. However, although it certainly strives to ensure delivery, log4j does not guarantee that each log statement will be delivered to its destination.

Log4j has three main components:

- loggers: Responsible for capturing logging information.
- appenders: Responsible for publishing logging information to various preferred destinations.
- layouts: Responsible for formatting logging information in different styles.

There are 8 levels of logging in Log4j :-

1. **ALL :** All levels including custom levels.
2. **DEBUG :** Designates fine-grained informational events that are most useful to debug an application.
3. **INFO :** Designates informational messages that highlight the progress of the application at coarse-grained level.
4. **WARN :** Designates potentially harmful situations.
5. **ERROR :** Designates error events that might still allow the application to continue running.
6. **FATAL :** Designates very severe error events that will presumably lead the application to abort.
7. **OFF :** The highest possible rank and is intended to turn off logging.
8. **TRACE :** Designates finer-grained informational events than the DEBUG.

Setting up Log4j in the net banking system:

```
private Logger logger = Logger.getLogger(getClass().getName());

//Initialise console for Logger.
BasicConfigurator.configure();
```

```
logger.debug(<message>);
```

Sequence Diagrams