# Chapter 2.

# analyzis and REQUIREMENTS SPECIFICATION.

## 2.1. Overview of system structure.

Full software system name: “Center of Ukrainian Aviation Training”.

Purpose of system: evaluation of pilots, maintaining pilot skills and provide information about flight simulation results.

Field of usage: flight simulators training centers with orientation on training semi-professional and professional pilots.

The objectives of application: make intuitive information resource that would help pilots perform their training and get evaluation, for instructors – to evaluate pilots.

The main idea of development this system is to create comfort, effective and secure system with intuitive interface and powerful back-end. Business logic is commonly easy. The functionality is to manage users and provide correct managing of data. One of structural ideas is to cut database rights of user portal for security reasons. That is why the connection user portal will be read-only. For database itself will be created two types for connection: for main application and user portal. The simulator will be working only with main application.

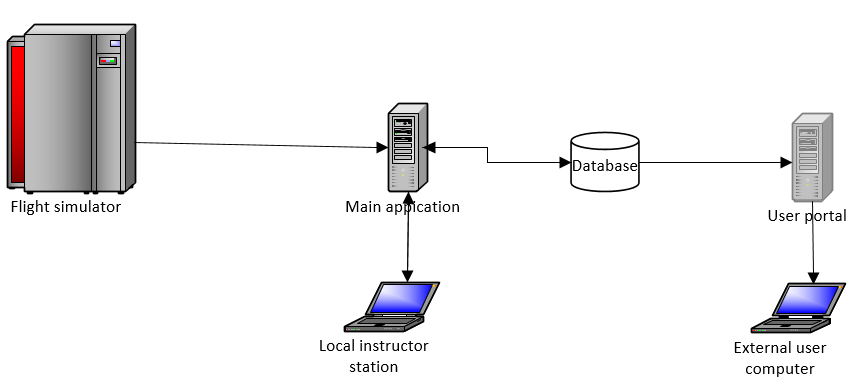


Fig.2.1. Structure architecture diagram

The whole system will be structured of four sub-systems that are described in Table 2.1.

*Table 2.1.*

Sub-systems description and purposes.

|  |  |
| --- | --- |
| Sub-system | Purpose |
| Flight simulator | Flight simulation is used to perform gathering data of pilot training process. |
| Main web-application | Web application stored locally and perform gathering data, calculation and user interface for comfort usage of application in web browser. Perform writing data and instructor evaluation to database. |
| User portal | Web site or portal for users to get to know the information about information center and evaluation of pilots. |
| Database | Storing the data for both main application and user portal. |

As seen from table all subsystems will be interconnected and working with database but in different ways. The structure of system is displayed on Structure architecture diagram (Fig.2.2.).

### 2.1.2. Functional requirements.

Main application functional requirements:

1. Ability to receive data from flight simulator.
2. Ability to process data from flight simulator.
3. Ability to expand data with additional information (username, plain model, etc.).
4. Ability to access database with read-write rights.
5. Ability to manage user accounts.
6. Ability to manage plain models.
7. Ability to manage metrics.
8. Ability to manage flight simulation records.
9. Ability to evaluate flight simulation records.
10. Ability to connect to flight simulator with socket.
11. Ability of authentication with session.
12. Ability of authorization with session.
13. Ability to authorize with administrator rights.
14. Ability for administrator to set roles.
15. Ability to manage data in database.
16. Ability of session management.
17. Ability to receive data from database.
18. Ability to visualize data received from database.
19. Ability to connect to application from only from local computers.
20. Ability to deploy application on different operational systems.
21. Ability to use application functionality with browser.
22. Ability to start session of gathering data from simulation.
23. Ability to stop session of gathering data from simulation.
24. Ability to setup port with which simulator can send data to application.

User portal functional requirements:

1. Ability to access database with read-only rights.
2. Ability to receive data from database.
3. Ability to visualize data received from database.
4. Ability of authentication with session.
5. Ability of authorization with session.
6. Ability to authorize with administrator rights.
7. Ability to deploy application on different operational systems.
8. Ability to use portal/site with browser.
9. Ability to perform visualization of possible data of flight simulator center.
10. Ability to perform easy visualization of record data gathered from database.

## 2.2. System architecture overview

Both main application and user portal use client-server architecture.

Client-server architecture is one of the architectural templates software and is the dominant concept in the creation of distributed network applications and provides for cooperation and the exchange of data between them. It provides the following key components:

• a set of servers that provide information services or other programs that appeal to them;

• set of clients using services provided by servers;

• network that provides interaction between clients and servers.

Servers are independent of each other. Customers also operate in parallel and independently of each other. No strict binding clients to servers. More than a typical situation is when one server simultaneously handles requests from different clients; on the other hand, the client can then apply to a single server, then to another. Customers should know about available servers, but may not have any idea about the existence of other customers.

### 2.2.1. Interface of system.

Interface of main application provides following capabilities:

* 1. User should login with credentials he/she has been provided.
  2. User can use menu items to browse through application.
  3. On main page administrator can setup port and start session of gathering data.
  4. On main page administratro can stop session.
  5. User can look at all records that have been saved in system due to his access rights.
  6. User can get all plain models.
  7. User can get all metrics from selected plain model.
  8. User can log out from main application.
  9. Administrator can manage users’ accounts.
  10. Administrator can manage plain models.
  11. Administrator can manage metrics from selected plain models.
  12. Administrator can manage records from flight simulator.

Interface of user portal provides following capabilities:

1. User can get to know information about flight simulation center.
2. User can log in and look through his last flight simulation records.
3. User can browse thought site with menu.
4. User can get all plain models.

### 2.3. Instruments for development of software.

Both user portal and main application will be using same technologies and instruments for development. This makes development process less complicated and more maintained. The architecture is client-server and main design pattern is Mode-View-Controller (MVC).

### 2.3.1. Model–view–controller

Model–view–controller (MVC) is a software design pattern for implementing user interfaces on computers. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

### 2.3.2. Separation of concerns.

In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. A concern can be as general as the details of the hardware the code is being optimized for, or as specific as the name of a class to instantiate. A program that embodies SoC well is called a modular program. Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface. Encapsulation is a means of information hiding. Layered designs in information systems are another embodiment of separation of concerns (e.g., presentation layer, business logic layer, data access layer, persistence layer).

In software engineering, the terms front end and back end refers to the separation of concerns between the presentation layer (front end), and the data access layer (back end) of a piece of software, or the physical infrastructure or hardware.

In software design, the model-view-controller architecture provides front and back ends for the database, the user and the data processing components. The "model" and "controller" make up the back, while the "view" makes up the front.

In content management systems, the terms front end and back end may refer to the end-user facing views of the CMS and the administrative views, respectively.

In speech synthesis, the front end refers to the part of the synthesis system that converts the input text into a symbolic phonetic representation, and the back end converts the symbolic phonetic representation into actual sounds.

For major computer subsystems, a graphical file manager is a front end to the computer’s file system, and a shell interfaces with the operating system. The front end faces the user, and the back end launches the programs of the operating system in response.

In compilers, the front end translates a computer programming source code into an intermediate representation, and the back end works with the intermediate representation to produce code in a computer output language. The back end usually optimizes to produce code that runs faster. The front-end/back-end distinction can separate the parser section that deals with source code and the back end that generates code and optimizes. Some designs, such as GCC, offer choices between multiple front ends (parsing different source languages) or back ends (generating code for different target processors).

Using the command-line interface (CLI) requires the acquisition of special terminology and memorization of commands, so a graphical user interface (GUI) acts as a front end desktop environment instead.

### 2.3.3. Back-end technologies.

The language for development back-end or data-access level of both applications is Java and Java EE like widely used computing platform for enterprise software.Data management will consist two technologies: Hibernate and JPA.

### 2.3.4. Java Persistence API.

The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

Persistence in this context covers three areas:

* the API itself, defined in the javax.persistence package;
* the Java Persistence Query Language (JPQL);
* object/relational metadata.

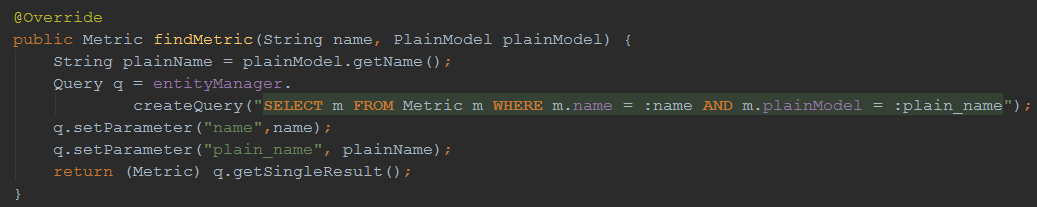
The Java Persistence Query Language (JPQL) makes queries against entities stored in a relational database. Queries resemble SQL queries in syntax, but operate against entity objects rather than directly with database tables. Example of method with usage of JPQL is visualized on Fig.2.2.

Fig.2.2.Example of usage JPQL in method findMetric.

**2.3.5. Hibernate ORM.**

Hibernate is an object-relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.

Hibernate is free software that is distributed under the GNU Lesser General Public License 2.1.

Hibernate's primary feature is mapping from Java classes to database tables, and mapping from Java data types to SQL data types. Hibernate also provides data query and retrieval facilities. It generates SQL calls and relieves the developer from the manual handling and object conversion of the result set.

The mapping of Java classes to database tables is implemented by the configuration of an XML file or by using Java Annotations. When using an XML file, Hibernate can generate skeleton source code for the persistence classes. This is auxiliary when annotations are used. Hibernate can use the XML file or the Java annotations to maintain the database schema.

There are provided facilities to arrange one-to-many and many-to-many relationships between classes. In addition to managing associations between objects, Hibernate can also manage reflexive associations wherein an object has a one-to-many relationship with other instances of the class type.

Hibernate supports the mapping of custom value types. This makes the following scenarios possible:

1. Overriding the default SQL type when mapping a column to a property.
2. Mapping Java Enums to columns as though they were regular properties.
3. Mapping a single property to multiple columns.

Definition: Objects in an object-oriented application follow OOP principles, while objects in the back-end follow database normalization principles, resulting in different representation requirements. This problem is called "object-relational impedance mismatch". Mapping is a way of resolving the object-relational impedance mismatch problem.

Mapping informs the ORM tool of what Java class object to store in which database table.

Hibernate provides transparent persistence for Plain Old Java Objects (POJOs). The only strict requirement for a persistent class is a no-argument constructor, not necessarily public. Proper behavior in some applications also requires special attention to the equals() and hashCode() methods.

Collections of data objects are typically stored in Java collection classes such as implementations of the Set and List interfaces. Java generics, introduced in Java 5, are supported. Hibernate can be configured to lazy load associated collections. Lazy loading is the default as of Hibernate 3.

Related objects can be configured to cascade operations from one to the other. For example, a parent Album object can be configured to cascade its save and/or delete operation to its child Track objects.

In Hibernate jargon, an entity is a stand-alone object in Hibernate's persistent mechanism which can be manipulated independently of other objects. In contrast, a component is subordinate to an entity and can be manipulated only with respect to that entity. For example, an Album object may represent an entity; but the Tracks object associated with the Album objects would represent a component of the Album entity, if it were assumed that Tracks could only be saved or retrieved from the database through the Album object. Unlike J2EE, Hibernate can switch databases.

### 2.3.6. Data access object (DAO).

In computer software, a data access object (DAO) is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, the DAO provides some specific data operations without exposing details of the database. This isolation supports the Single responsibility principle. It separates what data access the application needs, in terms of domain-specific objects and data types (the public interface of the DAO), from how these needs can be satisfied with a specific DBMS, database schema, etc. (the implementation of the DAO).

Although this design pattern is equally applicable to the following: (1- most programming languages; 2- most types of software with persistence needs; and 3- most types of databases) it is traditionally associated with Java EE applications and with relational databases.

The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that can but should not know anything of each other, and which can be expected to evolve frequently and independently. Changing business logic can rely on the same DAO interface, while changes to persistence logic do not affect DAO clients as long as the interface remains correctly implemented. All details of storage are hidden from the rest of the application (see information hiding). Thus, possible changes to the persistence mechanism can be implemented by just modifying one DAO implementation while the rest of the application isn't affected. DAOs act as an intermediary between the application and the database. They move data back and forth between objects and database records. Unit testing the code is facilitated by substituting the DAO with a test double in the test, thereby making the tests non-dependent on the persistence layer.

Potential disadvantages of using DAO include leaky abstraction, code duplication, and abstraction inversion. In particular, the abstraction of the DAO as a regular Java object can hide the high cost of each database access, and can also force developers to trigger multiple database queries to retrieve information that could otherwise be returned in a single operation with normal SQL set operations. If an application requires multiple DAOs, one might find oneself repeating the essentially the same create, read, update, and delete code for each DAO. This boiler-plate code may be avoided however, by implementing a generic DAO that handles these common operations. Time consumption is moderate.

### 2.3.7. Java API for RESTful Web Services.

JAX-RS: Java API for RESTful Web Services (JAX-RS) is a Java programming language API spec that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern. JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints.

From version 1.1 on, JAX-RS is an official part of Java EE 6. A notable feature of being an official part of Java EE is that no configuration is necessary to start using JAX-RS. For non-Java EE 6 environments a (small) entry in the web.xml deployment descriptor is required.

JAX-RS provides some annotations to aid in mapping a resource class (a POJO) as a web resource. The annotations include:

@Path specifies the relative path for a resource class or method.

@GET, @PUT, @POST, @DELETE and @HEAD specify the HTTP request type of a resource.

@Produces specifies the response Internet media types (used for content negotiation).

@Consumes specifies the accepted request Internet media types.

In addition, it provides further annotations to method parameters to pull information out of the request. All the @\*Param annotations take a key of some form which is used to look up the value required.

@PathParam binds the method parameter to a path segment.

@QueryParam binds the method parameter to the value of an HTTP query parameter.

@MatrixParam binds the method parameter to the value of an HTTP matrix parameter.

@HeaderParam binds the method parameter to an HTTP header value.

@CookieParam binds the method parameter to a cookie value.

@FormParam binds the method parameter to a form value.

@DefaultValue specifies a default value for the above bindings when the key is not found.

@Context returns the entire context of the object (for example @Context HttpServletRequest request).

### 2.3.8. Front-end technologies.

The basis for front-end part is common for nowadays – HTML5 for markup the page, CSS to stylize page markup and JavaScript for processing the basic calculations on side of user. The JavaScript framework for expanding the possibilities of development is AngularJS. Factory for REST interconnection with back-end is ngResource factory.

### 2.3.9. AngularJS framework.

AngularJS (commonly referred to as "Angular" or "Angular.js") is a complete JavaScript-based open-source front-end web application framework mainly maintained by Google and by a community of individuals and corporations to address many of the challenges encountered in developing single-page applications. The JavaScript components complement Apache Cordova, the framework used for developing cross-platform mobile apps. It aims to simplify both the development and the testing of such applications by providing a framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.

The AngularJS framework works by first reading the HTML page, which has embedded into it additional custom tag attributes. Angular interprets those attributes as directives to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code, or retrieved from static or dynamic JSON resources.

AngularJS is built on the belief that declarative programming should be used to create user interfaces and connect software components, while imperative programming is better suited to defining an application's business logic. The framework adapts and extends traditional HTML to present dynamic content through two-way data-binding that allows for the automatic synchronization of models and views. As a result, AngularJS de-emphasizes explicit DOM manipulation with the goal of improving testability and performance.

AngularJS's design goals include:

* + to decouple DOM manipulation from application logic. The difficulty of this is dramatically affected by the way the code is structured.
  + to decouple the client side of an application from the server side. This allows development work to progress in parallel, and allows for reuse of both sides.
  + to provide structure for the journey of building an application: from designing the UI, through writing the business logic, to testing.

Angular implements the MVC pattern to separate presentation, data, and logic components. Using dependency injection, Angular brings traditionally server-side services, such as view-dependent controllers, to client-side web applications. Consequently, much of the burden on the server can be reduced.

Angular uses the term "**scope**" in a manner akin to the fundamentals of computer science.

Scope in computer science describes when in the program a particular binding is valid. The ECMA-262 specification defines scope as: a lexical environment in which a Function object is executed in client-side web scripts; akin to how scope is defined in lambda calculus.

As a part of the "MVC" architecture, the scope forms the "Model", and all variables defined in the scope can be accessed by the "View" as well as the "Controller". The scope behaves as a glue and binds the "View" and the "Controller".

In Angular, "scope" is a certain kind of object that itself can be in scope or out of scope in any given part of the program, following the usual rules of variable scope in JavaScript like any other object. When the term "scope" is used below, it refers to the Angular scope object and not the scope of a name binding. Angular is an easy way to bind data in HTML DOM.

### 2.3.10. Bootstrap

The tasks performed by the AngularJS bootstrapper occur in three phases after the DOM has been loaded:

* Creation of a new Injector
* Compilation of the directives that decorate the DOM
* Linking of all directives to scope

AngularJS directives allow the developer to specify custom and reusable HTML-like elements and attributes that define data bindings and the behavior of presentation components. Some of the most commonly used directives listed in AngularJS directives (Table 2.2).

*Table 2.2.*

AngularJS directives

|  |  |
| --- | --- |
| Directives | Description |
| ng-app | Declares the root element of an AngularJS application, under which directives can be used to declare bindings and define behavior. |
| ng-bind | Sets the text of a DOM element to the value of an expression. For example, <span ng-bind="name"></span> displays the value of ‘name’ inside the span element. Any change to the variable ‘name’ in the application's scope reflect instantly in the DOM. |
| ng-model | Similar to ng-bind, but establishes a two-way data binding between the view and the scope. |
| ng-model-options | Provides tuning for how model updates are done. |
| ng-class | Lets class attributes be dynamically loaded. |
| ng-controller | Specifies a JavaScript controller class that evaluates HTML expressions. |
| ng-repeat | Instantiate an element once per item from a collection. |
| ng-show & ng-hide | Conditionally show or hide an element, depending on the value of a boolean expression. Show and hide is achieved by setting the CSS display style. |

*Table 2.2.*

AngularJS directives

|  |  |
| --- | --- |
| Directives | Description |
| ng-switch | Conditionally instantiate one template from a set of choices, depending on the value of a selection expression. |
| ng-view | The base directive responsible for handling routes that resolve JSON before rendering templates driven by specified controllers. |
| ng-if | Basic if statement directive that allow to show the following element if the conditions are true. When the condition is false, the element is removed from the DOM. When true, a clone of the compiled element is re-inserted. |
| ng-aria | A module for accessibility support of common ARIA attributes. |
| ng-animate | A module provides support for JavaScript, CSS3 transition and CSS3 keyframe animation hooks within existing core and custom directives. |
| ng-table | Simple table with sorting and filtering on AngularJS |

Since ng-\* attributes are not valid in HTML specifications, data-ng-\* can also be used as a prefix. For example, both ng-app and data-ng-app are valid in AngularJS

### 2.3.11. ngResorce.

A factory which creates a resource object that lets you interact with RESTful server-side data sources.

The returned resource object has action methods which provide high-level behaviors without the need to interact with the low level $http service.

Requires the ngResource module to be installed.

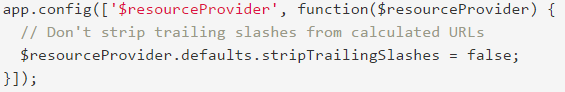
By default, trailing slashes will be stripped from the calculated URLs, which can pose problems with server backends that do not expect that behavior. This can be disabled by configuring the $resourceProvider (Fig.2.3.).

Fig.2.3. $resourceProvider configuration

### 2.3.12. Application server.

An application server is a software framework that provides both facilities to create web applications and a server environment to run them.

Application Server Frameworks contain a comprehensive service layer model. An application server acts as a set of components accessible to the software developer through an API defined by the platform itself. For Web applications, these components are usually performed in the same running environment as their web server(s), and their main job is to support the construction of dynamic pages. However, many application servers target much more than just Web page generation: they implement services like clustering, fail-over, and load-balancing, so developers can focus on implementing the business logic.

In the case of Java application servers, the server behaves like an extended virtual machine for running applications, transparently handling connections to the database on one side, and, often, connections to the Web client on the other.

The role of application server is WildFly.

WildFly, formerly known as JBoss AS, or simply JBoss, is an application server authored by JBoss, now developed by Red Hat. WildFly is written in Java, and implements the Java Platform, Enterprise Edition (Java EE) specification. It runs on multiple platforms.

WildFly is free and open-source software, subject to the requirements of the GNU Lesser General Public License (LGPL), version 2.1.

On 20 November 2014, JBoss Application Server was renamed WildFly. The JBoss Community and other Red Hat JBoss products like JBoss Enterprise Application Platform were not renamed.

## 2.4. Database model.

Entity-relationship model can be used as a basis for unification of different representations of data through the network model, relational model and set of entities.

In the entity-relationship model uses a more natural representation, according to which the real world consists of entities and relationships. This model is based on some important semantic information about the real world (description of other results related to semantics databases. Entity-relationship model can be used as a basis for a unified data presentation.

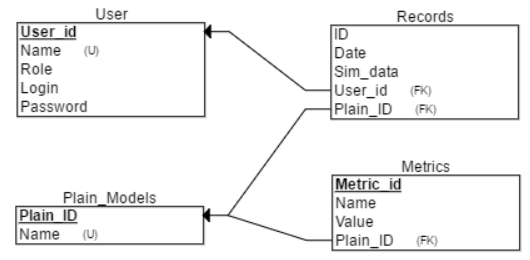
Conceptual ER-model (Fig. 2.2) provides more information on the composition of the database reveals the content database entities and specific fields for each database table, with a clear indication that the field will be the primary key (PK), which is a foreign key (FK) and which fields are required for entry.

Fig.2.4. ER model of database.

*Table 2.3.*

Entity Users

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column name | Type | Length | Limitations | Limitations integrity columns |
| User\_ID | INT |  | Unique ID | Primary Key |
| Name | Varchar | 100 | Unique user name | Not null, |
| Role | Varchar | 15 |  | Not null |
| Login | Varchar | 100 |  | Not null |
| Password | Varchar | 100 |  | Not null |

*Table 2.4.*

Entity Records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column name | Type | Length | Limitations | Limitations integrity columns |
| Record\_ID | INT |  | Unique ID | Primary Key |
| User\_ID | INT |  |  | Not null, Foreign Key |
| Date | Datetime |  |  | Not null |
| Plain\_ID | INT |  |  | Not null, Foreign Key |
| Sim\_data | Text |  | Simulator big text data type |  |

*Table 2.5.*

Entity Plain\_Models

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column name | Type | Length | Limitations | Limitations integrity columns |
| Plain\_ID | INT |  | Unique ID | Primary Key |
| Name | Varchar | 30 | Unique name |  |

*Table 2.6.*

Entity Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Column name | Type | Length | Limitations | Limitations integrity columns |
| Metric\_ID | INT |  | Unique ID | Primary Key |
| Name | Varchar | 20 |  | Not null |
| Value | Double |  |  | Not null |
| Plain\_ID | INT |  |  | Not null, Foreign Key |

## 2.5. Use-case diagrams representation in main application.

A use case diagram at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved. A use case diagram can identify the different types of users of a system and the different use cases and will often be accompanied by other types of diagrams as well.

### 2.5.1. Instructor use-case diagram.



Fig.2.5.Instructor use-case diagram.

### 2.5.2. Administrator use-case diagram.



Fig.2.5. Administrator use-case diagram

## 2.6. Class diagrams for main application.

Class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

### 2.6.1. Class diagram for “entity” package.

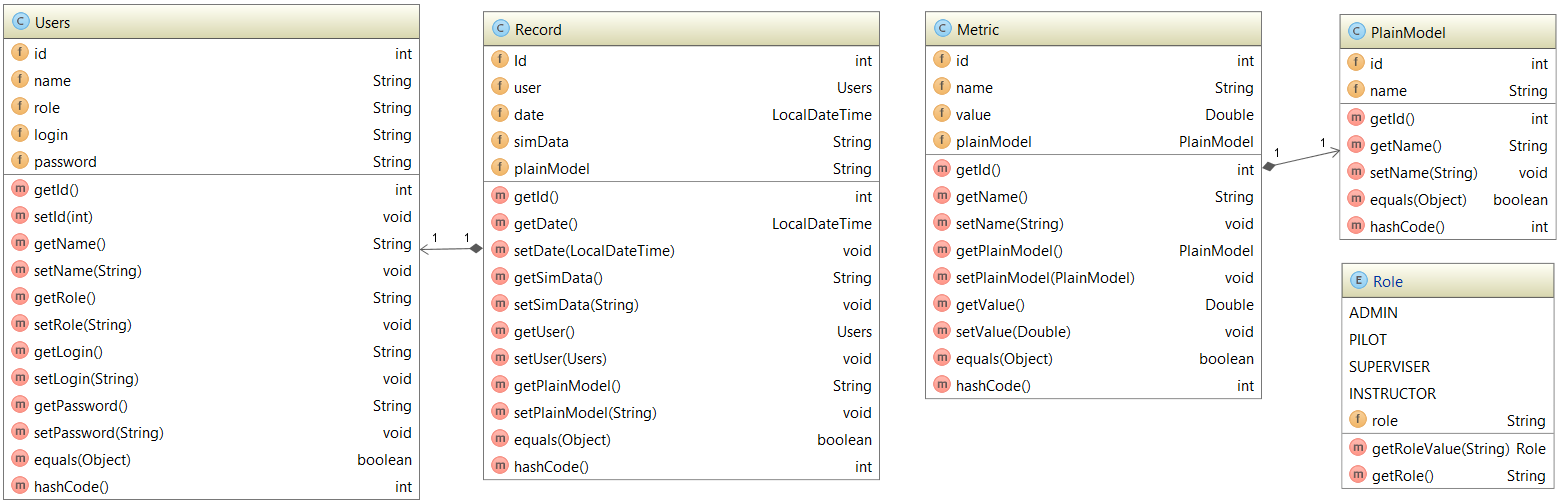
For representing user roles was chosen Enum because of uselessness of class usage. It is enough to use String values from enum and assign them to user.

Fig.2.6. Class diagram for entity package.

### 2.6.2. Class diagram for “DAOImplementation” package.

Fig.2.7. Class diagram for implementation classes of DAOs.

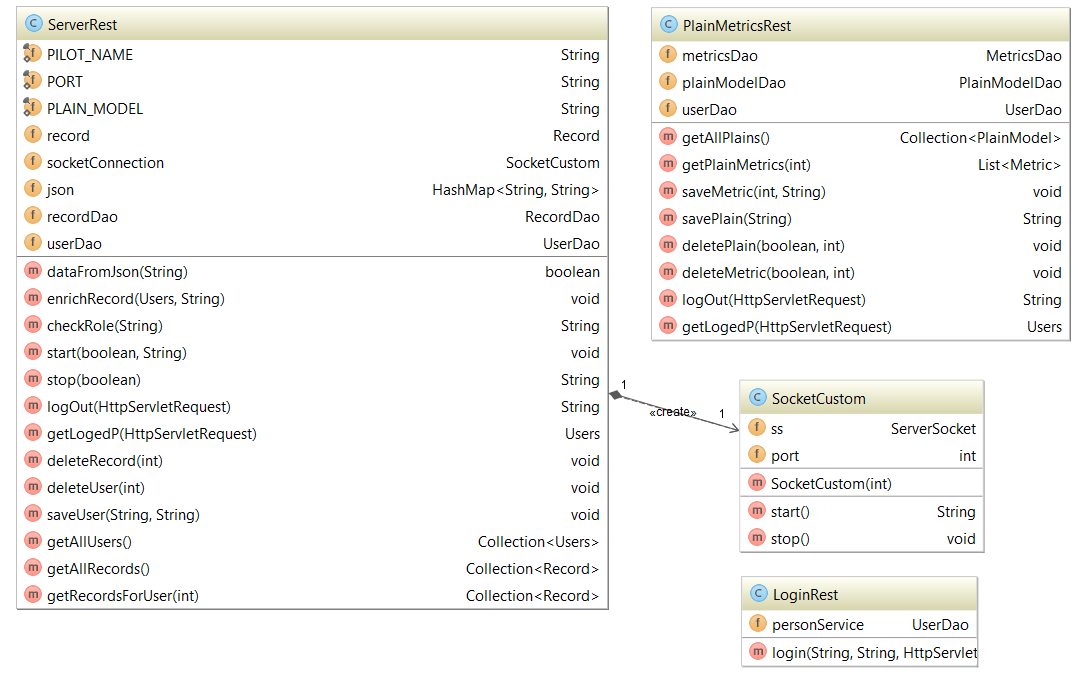
2.6.3. Class diagram for “rest” package with socketListener.

Fig.2.8. Class diagram for rest package with socket listener.