Real-Time Demand Capacity Tool

Design Report

Olivier Clermont (6445938)
Jonathan Ermel (6408238)
Mathieu Fortin-Boulay (6571541)
Philippe Legault (6376254)
Nicolas Ménard (6357275)

Presented to Dr. Liam Peyton
for the Software Engineering Capstone Project II
SEG4911

University of Ottawa
October 19, 2015

Table of Contents

Table of Figures

Architecture

Our application uses a client-server model. The client applications we are developing will be available on Android and iOS devices. For the server, we have two server components. The first is the Java server that acts as our application server, interacting with a PostgreSQL relational database. The communication of data between the clients and that component is done in a RESTful manner. The second component is an AsteriskNow server, which acts as a proxy to enable the communications over text, voice and video between users of the application, using SIP (Session Initialization Protocol).

In order to reuse as much code as possible throughout the different components of the application, we created a Core package that is compiled and shared between the Android and the iOS apps and the Java server. This package contains classes for the object model, JSON classes, controllers, exceptions classes, and interfaces for the views. With the use of **J2ObjC**, a command-line tool to translate Java code to Objective-C, we are able to use these classes with no modification in the iOS app. That way, the majority of the code that is not shared between the apps relates to the UI or platform-specific implementation. The next figure illustrates the relationship between the four modules:

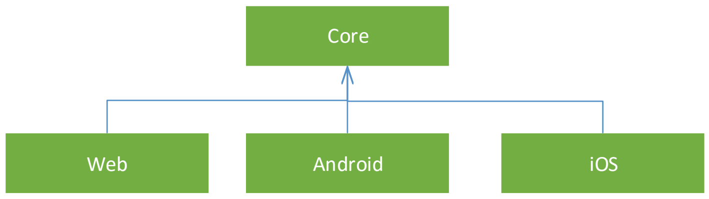


Figure 1: Relationship between modules

android

The majority of the android package consists of the user interface implementation found in the **presenter** sub-package. It also contains some platform-specific implementation of certain classes in the package **impl.** The **res** package includes the necessary graphics and XML files. It also implements the views' interfaces. There is also a **voip** package, where the code to communicate using SIP is contained. The description of the packages under **core** will be described later.

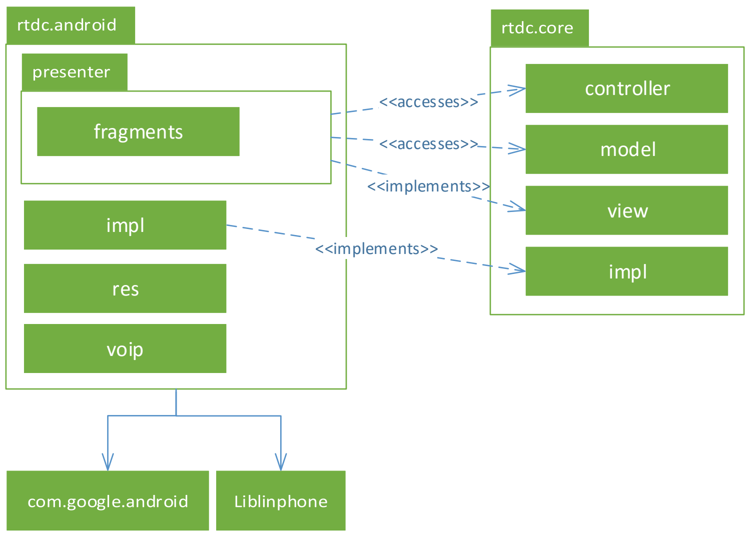


Figure 2: Android package diagram

ios

This package contains the code for the iOS app. The business logic code is inherited from **core**, by a translation of the Java code with J2ObjC, in a manner very similar to the Android app. The interface and the platform-specific code is implemented in Swift. One notable difference with Android is the **presenter** package, which contains the view-controllers for iOS, as opposed to Activities.

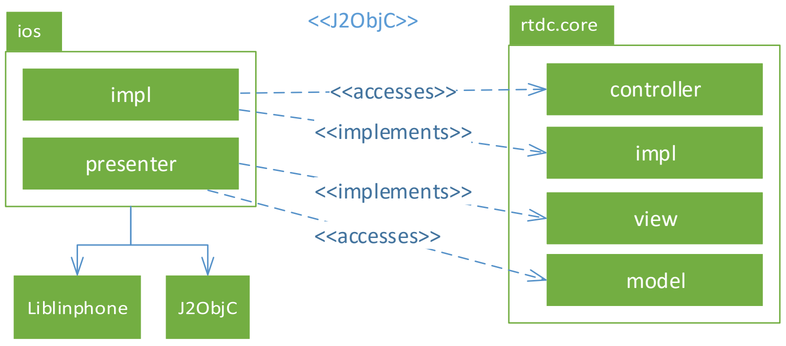


Figure 3: iOS package diagram

web

The **servlet** package contains the servlet definitions. Servlets provide a RESTful interface to access much of the functionality, for the apps. The **filter** package is used mainly for authentication and control of requests. The **config** package serves as a centralized way to define the behavior of Hibernate and Jersey, two of the library used server-side. The **model** package under **web** extends that of the **core**, and contains entities such as the authentication credentials, which only have meaning server-side. Finally, the **service** package contains classes to manager users, sessions, authentication, etc. It centralizes that logic, instead of having it spread over all servlets.

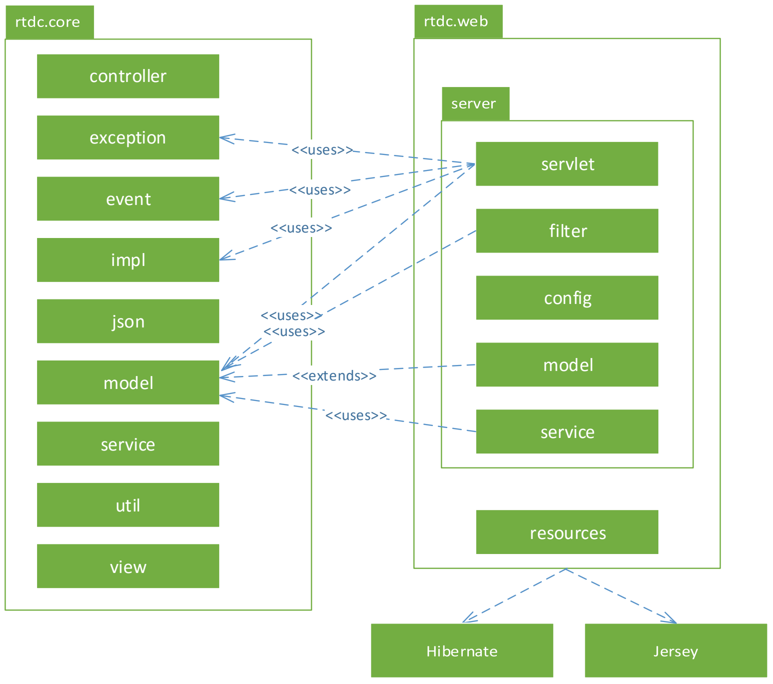


Figure 4: Web & Core package diagram

core

As seen in the previous diagram, the core contains all the logic that is shared between the apps. It is assumed, in the RTDC tool, that there is a view for each important part of the application, independent of the platform, and that each of those views has a controller (in the **controller** package.) Exceptions and events that may propagate through the network are implemented in the **exception** package and the **event** package, respectively.

impl contains interfaces that make uniform the access to platform-dependent features such as Storage, or network access.

The **json** package contains the logic used to serialize and deserialize the model, the events and the exceptions to JSON.

The **model** package contains basic objects needed both on the client and the server.

service defines how the core obtains information from the server, via network calls.

The **util** package contains utilities.

The **view** package contains interfaces that provide a uniform way to manipulate the UI from the controllers.

Unless a package is directly accessed from another module, then it is not part of the public API of that module. In a module, most packages can be accessed from anywhere, with the exception being the **json** package, which may only be accessed through a serializer/deserializer class.

Design

One of the most important concept in the application is the core, which relies on interfaces for the view, manipulated by the controllers. The next diagram illustrates the relationship between controllers and views, with the example of the login controller and view.

The views themselves, at least for the login, do nothing more than expose UiElement instances, which are used to update and get the text from textbox elements, in Android and iOS. These are our own abstraction of common UI elements. The task falls on the controller to detect that a login is required, to make the proper network call, store the user information, display any authentication error, and finally dispatch the user to the correct next view.

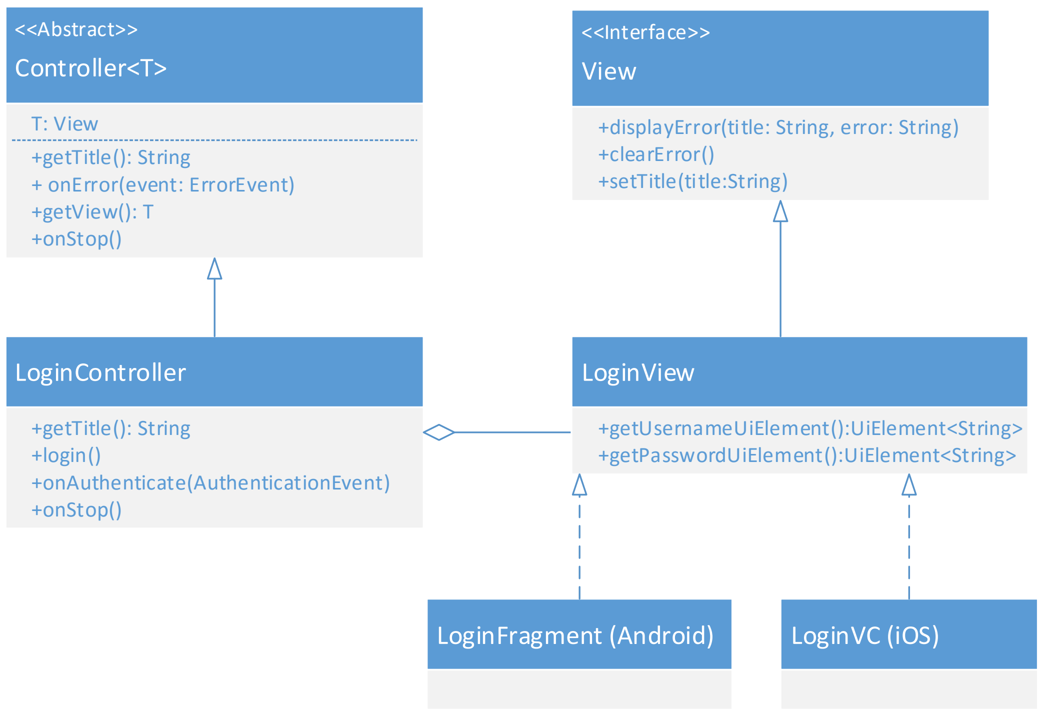


Figure 5: Relationship between the controller & the view for the login

This behavior is also made evident in the following interaction diagram, which illustrates the calls in the code to add a unit in an Android application. The call first propagates from the Android-specific code to the more generic core code (in the AddUnitController class). That class makes a network call to the web server, namely the UnitServlet, which in turn executes the correct sequence of action before returning a response. The control then goes back to the core, then to the Android app.

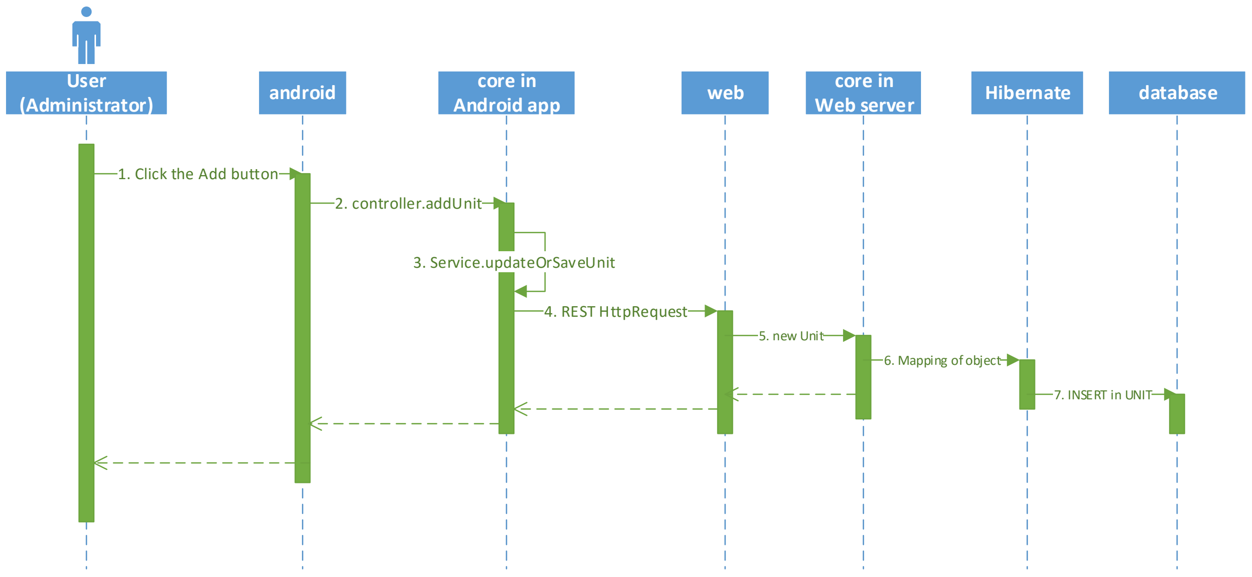
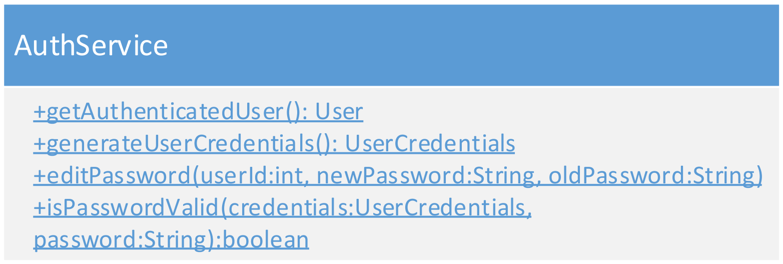


Figure 6: Add unit interaction diagram

Key mechanisms

There are several key mechanisms used in the RTDC tool that are worthy of lengthier explanations. These concepts are described here:

1. **Automatic propagation of events and exception over the network**: All of the classes that are meant to be accessed by both the server and the client are designed to be automatically serializable and deserializable to JSON. Therefore, errors that are thrown server-side are automatically returned to the client, re-instantiated and re-thrown. This allows for a simpler design, as information is shared easily between different parts of the application
2. **Java ME JSON library**: Classes from Java Mobile Edition for the manipulation of JSON are in use in the app. The reason for the use of this specific library is because of its absence of dependencies, which allows it to be easily transpiled to Objective C. The use of a single JSON library across the app ensures consistent serialization.
3. **Encapsulation:** To free the servlets of logic meant to manage some well-defined part of the application, like users or authentication, we introduce the concept of *service*, which is a uniform way to make common actions such as updating a user's information. Furthermore, it also has the benefit of centralizing all the logic pertaining to one aspect of a system in a single class with a unique point of access. The situation is illustrated in the next figure, pertaining to the authentication of users.



AuthService

+getAuthenticatedUser(): User
+generateUserCredentials(): UserCredentials
+editPassword(userId:int, newPassword:String, oldPassword:String)
+isPasswordValid(credentials:UserCredentials, password:String):boolean

Figure 7: Example of common methods contained in a service class

Database

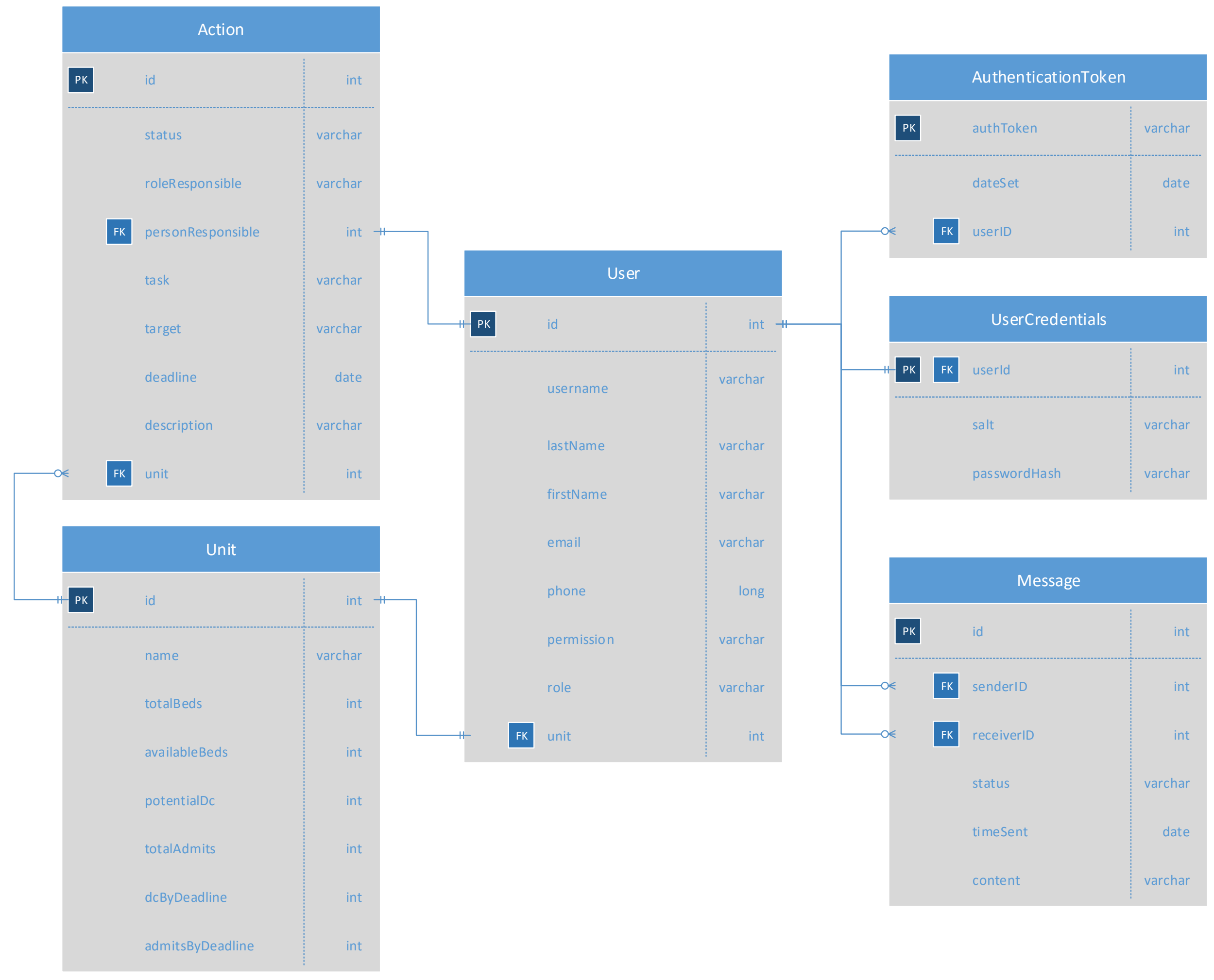


Figure 8: Entity-relationship diagram

As seen in the diagram, our database schema is very simple: a user entity models a real user of the system, be it a nurse, unit manager or administrator. Units depict sections to which a user may belong, as well as physical characteristics such as the name and the total number of beds. The action entity represents steps in the RTDC methodology that must be applied within a unit by a user. At last, user credentials and authentication token are linked to user session management and security, while the message entity allows users to communicate via text messages.