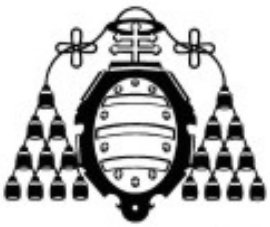
UNIVERSITY OF OVIEDO



SCHOOL OF COMPUTER SCIENCE
BACHELOR OF SOFTWARE ENGINEERING

FINAL DEGREE PROJECT

TELA: A FRAMEWORK FOR SOCIAL NETWORK ANALYSIS

DIRECTOR: FRANCISCO ORTIN SOLER

AUTHOR: ÁLVARO GONZÁLEZ RENESES

Abstract

The purpose of this final degree project is to develop an open source framework for social network analysis named “Tela”, which applications can communicate to via a REST API.

Tela aims to help developers to create applications that interact with one or more social networks, simplifying and unifying the authentication process and action execution. Some other helpful features offered by Tela are scheduling capabilities, request caching and data warehouse functionality.

Tela is designed to be highly extensible by the use of functionality modules that developers can implement. Furthermore, Tela already has two built-in modules: an Instagram one, which completely covers the read functionality of the Instagram API; and a Twitter module, with basic analysis functionality.

In addition to Tela, three more applications have been developed: “Tela CLI”, “Tela Who?” and “Tela Hawk”. These systems, which are based on Tela, offer different types of social network analysis for distinct target users; at the same time than they act as proof of concept for applications powered by Tela.

Keywords

Social Networks, Social Network Analysis, Instagram, Twitter, Framework, REST API.

Table of Contents

Index of Figures

1 Project Report

1.1 Introduction

The purpose of this document is to offer an overview of the motivation and utility of the project, as well as offering technical details about its design and implementation.

First, we will describe information that will be useful for the understanding of the project. Then, we will describe the proposed solution, as well as elaborate a planning and estimation. After that, we will describe each of the systems, including their analysis, design, test strategy, implementation, and manuals. Finally, we will comment the conclusions of the project.

1.2 Theoretical Aspects

Within this section, we will introduce some of the terms that will be most used across the project so that the non-specialized reader is able to grasp the contents of it.

1.2.1 API

An Application Program Interface (API) is a set of routines, protocols, and tools for building software applications, specifying how software components should interact (Beal, 2016).

APIs help developers to use technologies in building applications, abstracting the underlying implementation and providing an interface -or "contract"- between pieces of software (Beach, 2016).

APIs can have different forms and uses: provide an interface between the application and the operating system, using libraries of third parties, or manipulating remote resources.

1.2.1.1 REST API

Web APIs are a special type of web services that offer functionality to applications using the HTTP protocol, which is the same than web pages use. REST APIs are a type of Web APIs that follow certain constraints (Deering, 2012), like:

- There is a client handling the front end, and a server proving functionality. These two components are independent and can be easily replaced.
- It is stateless, which means that each request to the server contains all the information needed to answer to it.

In addition to that, REST APIs use HTTP methods, like GET, POST or DELETE; and answer using the JSON, XML or YAML format.

1.2.2 Databases

A database is a collection of information that is organized so that it can easily be accessed, managed, and updated (Rouse, 2006). Databases allow data persistence so that applications can store data among executions.

1.2.2.1 Graph Databases

Graph databases are a special type of databases, where information is stored with a graph structure composed of nodes/vertices, that have connections among them called edges. They offer some advantages over traditional databases; for example, is simpler and faster to travel from one node to another, and it is easier to model some problems using graphs.

1.2.3 Framework

A (software) framework is a platform for developing software applications, which provides a foundation that software developers can take advantage of to build applications. Frameworks can also contain libraries, tools, or other programs, easing the developing process by avoiding repetitive or complex processes.

A framework is similar to an API; in fact, it technically includes an API. While frameworks serve as a foundation for programming, APIs provides access to the elements supported by the framework (Christensson, 2013).

1.2.4 Social Network

When referred to the digital technology, a social network is an online community of people who interact and communicate with each other using a website –or any other piece of software-, sharing information as media, news, etc. (Dictionary.com, 2016).

1.2.4.1 Instagram

Instagram is a media-sharing application for mobile devices. In their own words (Instagram, 2016):

> *Instagram is a fun and quirky way to share your life with friends through a series of pictures. Snap a photo with your mobile phone, then choose a filter to transform the image into a memory to keep around forever. We're building Instagram to allow you to experience moments in your friends' lives through pictures as they happen. We imagine a world more connected through photos.*

What makes it different to other social networks is its simplicity and the fact that is completely focused on media. Some of the most common actions users can do with Instagram are:

- Upload a picture/video, edit it and publish it.
- See the pictures a user has published.
- Follow other users, so that you see their new posts in your timeline.
- Comment in the media of other users.
- Specify the location where a picture/video was taken.
- Tag other users in a picture.

1.2.4.2 Twitter

Twitter is a microblogging application that allows people to share tweets: posts up to 140 characters of media, characters or link. Twitter defines itself as (Twitter Help Center, 2016):

> *Twitter is a service for friends, family, and coworkers to communicate and stay connected through the exchange of quick, frequent messages. People post Tweets, which may contain photos, videos, links and up to 140 characters of text. These messages are posted to your profile, sent to your followers, and are searchable on Twitter search.*

1.2.4.3 Social Network Analysis

Social Network analysis is the study of social relations among a set of actors, which may have different types: kinship (e.g. brother of), affective (e.g. likes), cognitive (e.g. knows), actions (e.g. comments), etc. (Analytictech, 2016).

These relationships can range from quite simple, like User A follows User B, up to fairly complex, as User A is related to User B through X actors.

1.3 Motivation

Social network analysis has become a huge market with an incredible potential. There are hundreds of applications, even for the smallest functionalities as showing the friends that have unfollowed the user.

However, each social network has their own API, with different authentication, endpoints, data types, and so on. For a developer who wants to start a new application without a background in those APIs, it can be very hard; especially if the only purpose is to use part of the functionality as showing the followers.

Tela aims to ease the development of applications that perform social network analysis so that even non-experienced programmers can focus on what they want: developing applications, not fighting against APIs.

1.4 Current Situation

Right now, developers that want to interact with several social networks have to adapt their software to each of their APIs, which include studying the documentation, adapting their applications to the authentication flows, creating models and data classes, and adding a lot of dependencies to their code.

Some social networks have libraries that ease this interaction, but they are just offered in a few programming languages, and developers still have to study the documentation of each of them.

1.5 Alternatives

Only two alternatives have been considered: either an internal solution, which would be a library abstracting the functionality of social networks, or an external system taking care of the interaction with the APIs.

1.5.1 Abstraction Library

The most straightforward alternative would be to develop a library which serves as an abstraction layer for the rest of libraries. However, this solution has three main drawbacks:

- This solution is platform dependent. We should choose a programming language we would develop the library on. We would improve the current situation, but just for a minority of developers.

- Not all the social networks offer libraries in every programming language. Therefore, we should choose between finding a programming language that already has official support by every social network, or using unofficial libraries whose development might be discontinued at any time.
- Social networks offer different functionality. Twitter, for example, is focused on tweets (short messages), while others like Instagram focuses on media publications. It is arguable if it would even be possible to abstract common parts without sacrificing the core functionality of each API.

1.5.2 External System

The other alternative is to create an external system that applications can connect to in a platform-independent way. This would still create a dependency on an external service, which might not be desirable depending on the project, but offers high flexibility and scalability.

In some way, is similar to the microservices architecture that is becoming so popular nowadays, where similar functionality is grouped as a service, and communication among them is usually performed via synchronous protocols like HTTP, or asynchronous ones such as AMQP (Richardson, 2014).

1.6 Chosen Solution

After analyzing both solutions, we have decided to implement the later one: an external system. The ultimate purpose of this project is to provide an abstraction to developers, helping them as much as possible while keeping the full functionality intact.

We will implement a REST API developers will be able to interact with. This system, which will be called "Tela", will take care of communicating with the APIs of the social networks, abstracting all the particularities (e.g. the different authentication flows) and offering a unified API.

In addition to this system, we will also develop three more applications:

- **Tela CLI**. A tool that interacts with a Tela instance from the command line, targeting developers, and automation tasks.
- **Tela Who?** Tela Who? is a web application for companies, brands, and regular users, that allows them to obtain a better understanding about how a user is connected with them, so that they can improve their relationship and target similar users.
- **Tela Hawk**. Tela Hawk is a tool that monitors the variation over time of the number of media posted by the user, followers and people the user is following; displaying this information to the user.

1.7 Scope

The desired outcome of this project is to produce a stable version of Tela, completely functional and production ready. In addition to that, three more applications will be developed.

Regarding the scope of each one, we will focus on developing Tela, as it is the main system of the project. We will create design documents, extensive tests suites, and detailed manuals.

On the other hand, the other system will not be as prioritized. We will create design documents and test strategies, but they will not be complete nor exhaustive. Particularly, the expected outcome for the Tela Who? and Tela Hawk applications are mere functional prototypes, which might not be recommended to use with real users.

1.8 Definitions and abbreviations

- **Actions**: functions that Tela realizes. In the official APIs, they would be the endpoints. For example: retrieving the followers of a user.
- **Clients / Client Applications**: applications that are powered by Tela.
- **Followers**: a group of people that follows a certain user in a social network.
- **Friends**: users that are followers of a certain user, and that are being followed back by him.
- **Modules**: a group of actions, usually from a specific social network: For example, the Instagram module.
- **Tela Framework / Tela Server**: same as Tela, main system of this project.

2 Planning

2.1 Temporal Estimation

The estimated development time of each system can be seen in the following Gantt chart.

Note that the weeks are composed by five (working) days.

Iteration Outcome	Days
Tela	
Analysis & Design	7
Project ready (e.g. folders, tools…)	1
Basic architecture	3
Databases configured	2
Users authorization	7
HTTP Server	5
Instagram module manages users	10
Scheduling functionality	10
Instagram module manages media	7
Twitter module	5
Final application	5
Tela CLI	
Analysis & Design	2
Project ready (e.g. folders, tools…)	1
Basic architecture	2
Console commands are parsed	1
User authentication in Tela	2
Execute requests to the server	2
Login into Instagram	2
Interact with Instagram user Tela API	3
Interact with scheduling	3
Interact with the Instagram media API	2
Login into Twitter	2
Interact with the Twitter module API	2
Final application	2
Tela Who?	
Analysis & Design	2
Project ready (e.g. folders, tools…)	2
Basic architecture	2
Visual interface	3
Complete Functionality	5
Final application	2
Tela Hawk	
Analysis & Design	1
Project ready (e.g. folders, tools…)	1
Basic architecture	1
Visual interface	2
Complete Functionality	3
Final application	2

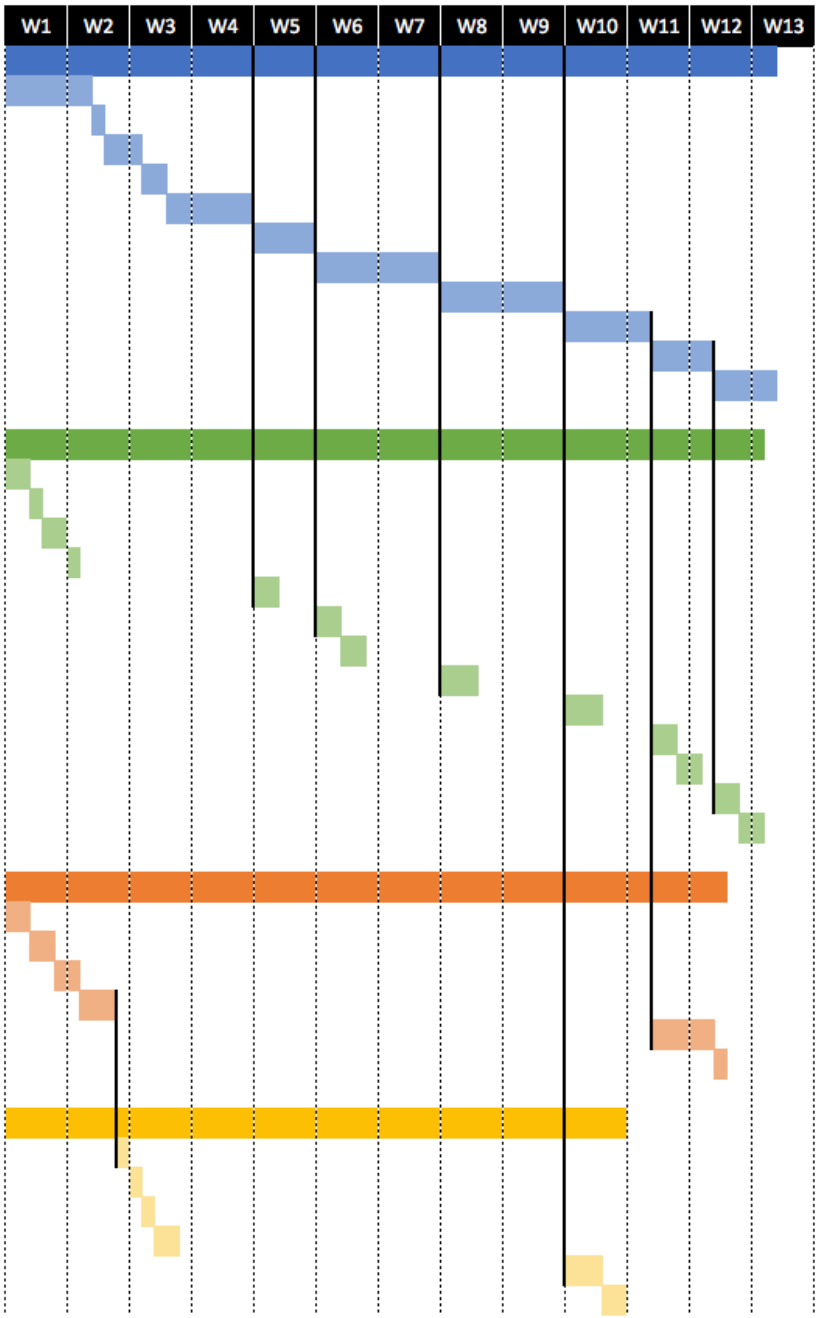


Figure 2.1 Gantt Chart

In order to calculate the estimated time depending on the number of developers (considering that they are full stack developers), we will aggregate each system in a unidimensional line, and then combine the lines:

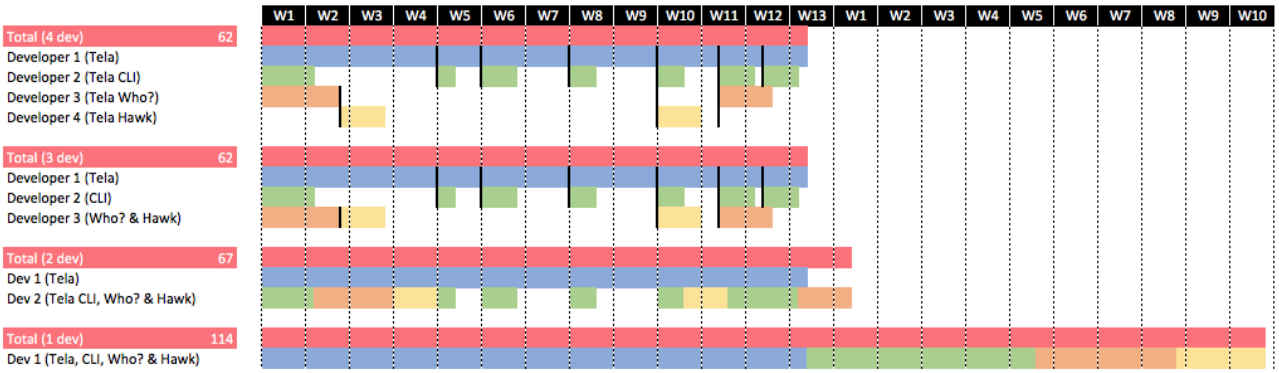


Figure 2.2 Gantt Chart by Developers

Conclusions:

- Both three and four developers would employ 62 days.
- Two developers would employ 67 days.
- One developer would employ 114 days.

2.2 Cost Estimate

2.2.1 Cost of Developers

We will use a salary of 2000 euros per month per developer for our calculations (Cuanto Gana, 2015), which divided by 20 working days per month is 100 euros per working day.

With this information, we will elaborate a comparative table:

Developers	Days	Total Days to Pay	Total Cost (€)
1	114	114 x 1 = 114	114 x 100 = 11,400
2	67	67 x 2 = 134	134 x 100 = 13,400
3	62	62 x 3 = 186	186 x 100 = 18,600
4	62	62 x 4 = 248	248 x 100 = 24,800

Analyzing this information, we could conclude that the best option is to develop the application between two developers. If there is only one, the cost is slightly smaller (€ 2.000 less). However, it would take 47 more days, whose opportunity cost, plus the monthly variable costs, would end up being more expensive.

2.2.2 Other Costs

2.2.2.1 Deployment Architecture

During the development, we could use Heroku[1] as a PaaS (Platform as a Service) cloud provider. Its free plan is permissive enough, however, we could always join the hobby plan[2] which is $ 7 per application, therefore $ 21 (€ 19) per month (as Tela CLI does not need a deployment platform).

2.2.2.2 Licenses

The only licenses needed are those corresponding to the IDE the developers would like to use. This is hard to estimate as it depends on personal preferences. In order to make this calculation, we would consider the subscription for the "All Products Pack" commercial license of JetBrains[3], which is € 64.90 per user per month.

2.2.2.3 Hardware for Developers

For short projects as this one, the best option would be to rent the equipment. Flex IT Rent offers Apple iMacs[4] of 21.5" for € 142.00 per unit per month, which for two developers would be a monthly cost of € 284.00.

2.2.3 Final Cost Estimate

Description	Quantity	Price (€)	Total Ex. Tax (€)
Developing Days	134	100.00	13,400.00
Deployment	2	19.00	38.00
JetBrains Licenses	2	64.90	129.80
Hardware	2	284.00	568.00
		Total Excl. Tax:	**14,135.80**
		Tax (21%):	**2,968.518**
		Total Incl. Tax:	**17,104.32**

[1] https://www.heroku.com/

[2] https://www.heroku.com/pricing

[3] https://www.jetbrains.com/idea/buy/#edition=commercial

[4] https://www.flexitrent.es/es/alquilar-un-apple

3 System I: Tela Framework

3.1 Introduction

3.1.1 Description of the System

Tela is an open-source framework for social network analysis, which applications can connect to via a REST JSON API. Some of the features of Tela are extensibility via functionality modules, action execution, and scheduling, request caching and authentication management.

3.1.2 Scope

Within this project, this is the main and most important system. Therefore, most part of the effort has been dedicated on this. The outcome of the project will be a fully functional production-ready system, with detailed manuals and documentation, and an in-deep suite of automated tests.

3.1.3 Development Process

The requirements of the system are not well defined: there are several things that cannot be planned in advance until functional prototypes are produced and the viability of each decision is tested. For example, Tela should be extensible but… How extensible should it be? Is our model extensible enough? Is it viable taking into account the APIs that the social networks provide? Also, every component -and therefore, requirement- of the system depends on the viability and design of the rest.

For this reasons, considering a non-iterative process as the waterfall model might not be very adequate. Instead, we will use an agile approach based on fast iterations seeking for functional prototypes as soon as possible. At each iteration, we will apply a Test Driven Development strategy putting special effort on unit tests, as regression testing is an essential part of agile development (Padmanaban, 2014) (Winkels, 2010) (Rajkumar, 2016).

Regarding the production of documentation, at each iteration design documents (e.g. class diagrams) will be used, but they will not be very detailed nor done with proper software, so that we do not employ too much time when each design might be discarded at the next iteration. At the end of the development, complete diagrams will be produced in order to help the understanding and maintainability of the system.

On the other hand, in-class documentation (using Javadoc) will be produced during each phase.

3.2 Analysis of the System

3.2.1 Identification and Description of Subsystems

The system can be decomposed into three main subsystems: the core, the functionality modules, and the assembler:

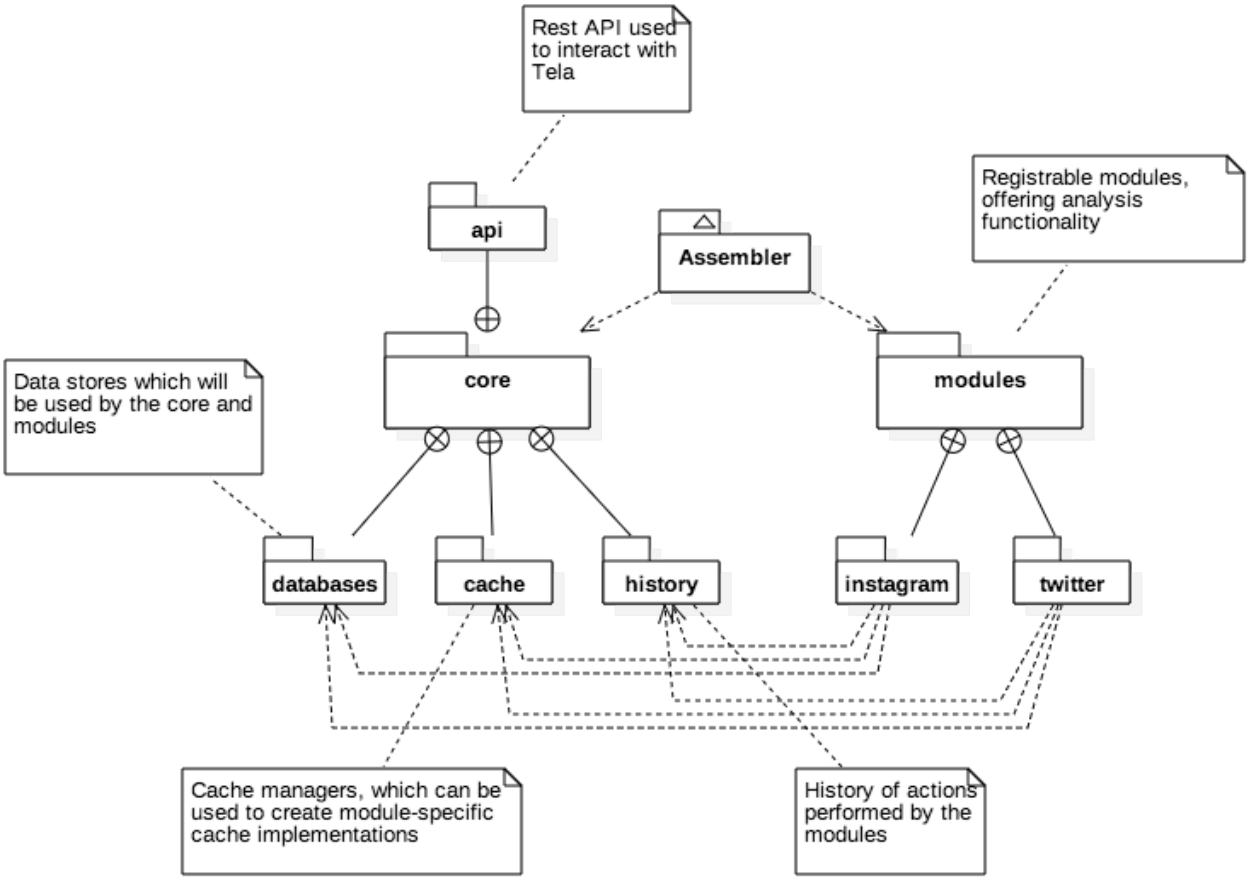


Figure 3.1 Package Diagram Overview of the Subsystem of the Framework

3.2.1.1 Core

The 'core' is the main subsystem of the framework, which in turn is composed of several components that aim to be as independent as possible to each other.

3.2.1.1.1 API

This component exposes a REST API, which is the interface the client applications will use to communicate and interact with the framework.

3.2.1.1.2 Dispatcher

The Dispatcher is the component responsible for scanning the modules and finding and storing the actions contained in them. Then, when an action execution request is received, the dispatcher will retrieve and invoke the action with the supplied parameters.

3.2.1.1.3 Scheduler

The Scheduler is the component responsible for providing scheduling functionality so that actions can be programmed to be executed periodically.

3.2.1.1.4 Sessions

This component is responsible for the authentication and authorization of the framework; this is creating, validating, updating and deleting sessions. It also stores the security tokens of the modules (e.g., the access token for a social network), associating them to existing sessions.

3.2.1.1.5 History

The History component offers a record of the latest executed actions. This information is mainly used for caching strategies, although it could also be used for debugging, statistical or profiling purposes. Each module is responsible for interact with the History.

3.2.1.1.6 Cache

The Cache component offers data caching functionality so that modules can easily implement concrete cache implementations without worrying about the underlying implementation (e.g., a cache storing IDs with their associated usernames).

3.2.1.1.7 Databases

This component includes all the functionality regarding the databases which other core components and the modules will use.

3.2.1.1.8 Configuration

The configuration component is responsible for reading the configuration specified by the user, as well as providing a programmatically way of configuring the system. It is important to denote that it does not configure any component, but reads and stores its configuration values.

3.2.1.1.9 Util

This component includes functionality independent from the framework, but that are used by it in order to speed or ease certain tasks (e.g., a customized HTTP client).

3.2.1.2 Modules

The modules are registrable components that add functionality to the framework. Although it is not mandatory, modules use some components from the core in order to make its development simpler.

Apart from that, modules are quite independent from the core, so that they can be easily developed or modified, extending the capabilities of the framework. For example, each Social Network should have (at least) one module. In this way, developers may decide whether or not include them in their custom bundles.

Within this project there two modules will also be developed Instagram and Twitter.

3.2.1.3 Assembler

The Assembler is the component responsible of:

1. Reading the configuration from the Configuration component, and applying it to the corresponding components.
2. Instantiating the components and injecting them into the Server.
3. Instantiating and returning an instance of the Server.

The usage of the Assembler is not mandatory, as the configuration and injection could be done programmatically (e.g., in the Main method). However, it is highly recommended to do it.

3.2.2 Analysis of Use Cases

There are two kinds of actors that interact with the Framework:

- On one hand, there exist client applications, which interact with the framework through the API. These actors realize general tasks as testing the connection, authorize themselves through Sessions, and execute and schedule actions.
- On the other hand, there are developers who want to build their own distribution of the framework.

Regarding the latter case, developers might want to build it if:

- They want to configure Tela programmatically.
- They aim to integrate Tela within their own software.
- They want to extend the functionality of the framework by developing their own modules.
- They want to exclude the components/modules that they do not use.
- They want to use another database or component implementation.

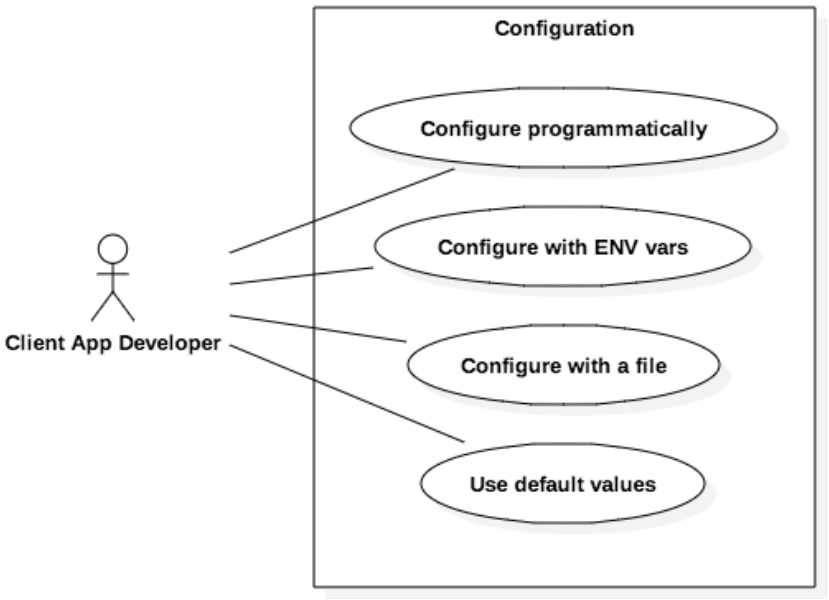


Figure 3.2 Client App Developer Use Case Diagram

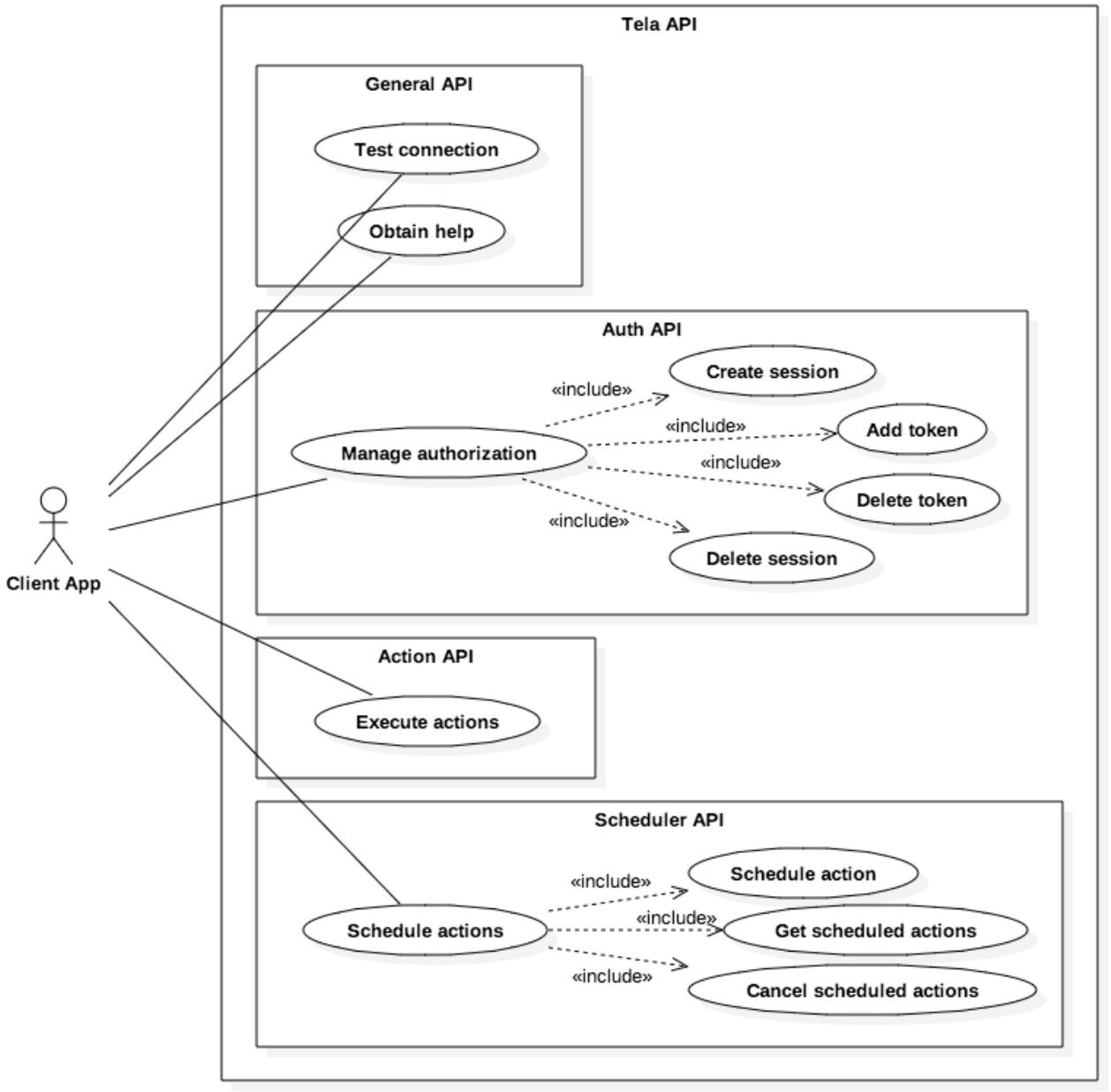


Figure 3.3 Client App Use Case Diagram

3.2.3 Specification of System Requirements

From the use cases identified in the previous section, we will elaborate a list of functional and non-functional requirements the system should meet.

3.2.3.1 Functional Requirements

ID	Name	Description
FR1	Authentication Management	
FR1.1	Create session	Crate a session with a unique access token
FR1.2	Add token	Add a module token to an existing session
FR1.3	Delete token	Delete a token from an existing session
FR1.4	Delete session	Delete a session, as well as all the module tokens it has
FR2	Action execution	
FR2.1	Execute action	Execute an action and return its results
FR3	Scheduling management	
FR3.1	Schedule	Schedule an action to be executed periodically with a given delay
FR3.2	Get all scheduled	Retrieve information about all the actions that a user has scheduled
FR3.3	Cancel scheduled	Cancel a scheduled execution of a certain action
FR3.4	Cancel all scheduled	Cancel the scheduled execution of all the actions scheduled by a user
FR4	General	
FR4.1	Test connection	Return a simple message
FR4.2	Help	Return information about all the modules installed in the Tela Server, as well as about their actions.
FR4.3	Module Help	Return information about all the actions from a given module.

3.2.3.2 Non-Functional Requirements

ID	Description
NFR1	Installation
NFR1.1	The tool will be multiplatform.
NFR2	Documentation
NFR2.1	The tool will have several manuals detailing its installation and usage.
NFR2.2	All the APIs offered by the system will be documented in detail.
NFR2.3	A manual for developers will be produced.
NFR3	Execution
NFR3.1	The tool will be executed from the console.
NFR4	Scalability
NFR4.1	The architecture of the system should be able to scale.
NFR4.2	The system implementation should be efficient and scalable.
NFR5	Implementation
NFR5.1	The tool will be easily extensible, so that new modules can be added to it.
NFR6	Security
NFR6.1	Authentication and authorization will be critical part of the system, specially regarding to their access tokens and scheduled actions.
NFR7	Logging
NFR7.1	The system should log main events and request, to both the console and a file.

3.2.4 Design of Use Case Scenarios

3.2.4.1 Authentication Management (FR1)

Create session (FR1.1)	
Description	Create and return a session with a unique access token.
Primary Actors	Client Application & Tela.
Preconditions	-
Post conditions	A new session has been created.
Trigger	The API receives a create session request
Flow	1. A session with unique session ID and access token is created 2. The access token of the session is sent back to the client

Add token (FR1.2)	
Description	Add a module token to an existing session.
Primary Actors	Client Application & Tela.
Preconditions	-
Post conditions	The module token is stored into the session.
Trigger	The API receives an add module token request
Flow	1. An access token, module name, and token are received. 2. The session corresponding to the access token received is loaded. 2.1 If the session does not exist, an error message is returned. 3. The module token is stored into the session, overwriting previous values 3.1 If the module does not exist, an error message is returned 3.1.1 If the module token already existed on other session, it is deleted from it 4. The system returns a success message.

Delete token (FR1.3)	
Description	Delete a module token from an existing session.
Primary Actors	Client Application & Tela.
Preconditions	-
Post conditions	The module token is deleted from the session.
Trigger	The API receives a delete module token request
Flow	1. An access token and module name are received. 2. The session corresponding to the access token received is loaded. 2.1 If the session does not exist, an error message is returned. 3. The module token is deleted from the session 3.1 If the module does not exist, an error message is returned 3.1.1 If it does, but there was no module token stored, an error message is returned 4. The system returns a success message.

Delete session (FR1.2)	
Description	Delete a session and all its tokens
Primary Actor	Client Application & Tela
Preconditions	-
Post conditions	The session and its tokens are deleted.
Trigger	The API receives a delete session request.
Flow	1. An access token is received. 2. The session corresponding to the access token received is loaded. 2.1 If the session does not exist, an error message is returned. 3. All the modules tokens of the session are deleted. 4. The session is deleted 5. The system returns a success message.

3.2.4.2 Action Execution (FR2)

Execute action (FR2.1)	
Description	Execute an action.
Primary Actors	Client Application & Tela
Preconditions	-
Post conditions	-
Trigger	The API receives an execution request.
Flow	1. An access token, module, action and parameters are received. 2. The session corresponding to the access token received is loaded. 2.1 If the session does not exist, an error message is returned. 3. The action is loaded 3.1 If the module does not exist, an error message is returned 3.1.1 If it does, but the action does not exist, an error message is returned 3.1.1.1 If it does, but the number or type of its parameters do not match with the received ones, an error message is returned. 4. The action is dispatched (executed) with the received parameters. 4.1 If there is any problem, an error message is returned 5. The system returns the result

3.2.4.3 Scheduling Management (FR3)

Schedule (FR3.1)	
Description	Schedule an action to be executed periodically
Primary Actors	Client Application & Tela
Preconditions	-
Post conditions	The action has been scheduled
Trigger	The API receives a schedule request.
Flow	1. An access token, delay, module, action and parameters are received. 2. The session corresponding to the access token received is loaded. 2.1 If the session does not exist, an error message is returned. 3. The action is dispatched (see FR2.1) and the result temporally stored. 4. An scheduled action is created and stored. 4.1 If the delay was not supplied, or it is invalid, the default of the action is used. 4.1 If the action is not schedulable, an error message is returned. 5. The information about the scheduled action and its result are returned.

Get scheduled (FR3.2)	
Description	Get all the actions that a user has scheduled.
Primary Actors	Client Application & Tela
Preconditions	-
Post conditions	-
Trigger	The API receives a get scheduled request.
Flow	1. An access token is received. 2. The session corresponding to the access token received is loaded. 2.1 If the session does not exist, an error message is returned. 3. The scheduled actions by the retrieved session are loaded 4. The system returns the scheduled actions

Cancel scheduled (FR3.3)	
Description	Cancel the scheduled execution of an action.
Primary Actors	Client Application & Tela
Preconditions	-
Post conditions	The scheduling of the action has been canceled.
Trigger	The API receives a cancel scheduled request.
Flow	1. An access token and scheduled action ID are received. 2. The session corresponding to the access token received is loaded. 2.1 If the session does not exist, an error message is returned. 3. The scheduled action corresponding to the ID received is loaded 3.1 If the action does not exist, an error message is returned 3.1.1 If it does, but it was not scheduled by the retrieved session, an error message is returned 4. The scheduled action is deleted. 5. The system returns a success message.

Cancel all scheduled (FR3.4)	
Description	Cancel all the scheduled execution by the authorized user.
Primary Actors	Client Application & Tela
Preconditions	A connection has been established with a running Tela Server.
Post conditions	All the scheduled tasks of the user have been canceled.
Trigger	The API receives a cancel all scheduled request.
Flow	1. An access token is received. 2. The session corresponding to the access token received is loaded. 2.1 If the session does not exist, an error message is returned. 3. The scheduled actions created by the retrieved session are deleted. 4. The system returns a success message.

3.2.4.4 Obtain Help (FR4)

Test connection (FR4.1)	
Description	Return a simple OK message.
Primary Actors	Client Application & Tela
Preconditions	-
Post conditions	-
Trigger	The API receives a test request.
Flow	1. The system returns a “OK” message.

Help (FR4.2)	
Description	Return information about all the modules and their actions.
Primary Actors	Client Application & Tela
Preconditions	-
Post conditions	-
Trigger	The API receives a help request.
Flow	1. All the modules are retrieved. 2. For each module, the help of its actions is generated. 3. The system returns the help.

Help (FR4.3)	
Description	Obtain information about a modules and its actions.
Primary Actors	Client Application & Tela
Preconditions	A connection has been established with a running Tela Server.
Post conditions	-
Trigger	The API receives a module help request.
Flow	1. The system received a module name. 2. The corresponding module is retrieved. 2.1 If the module does not exist, an error message is returned. 3. The help of the actions of the module is generated. 4. The system returns the help.

3.2.5 Analysis of Classes and Packages

Each component will have its own package, which will try to be as isolated as possible from the others. The most dependent functionality is the configuration, which is also isolated on its own package.

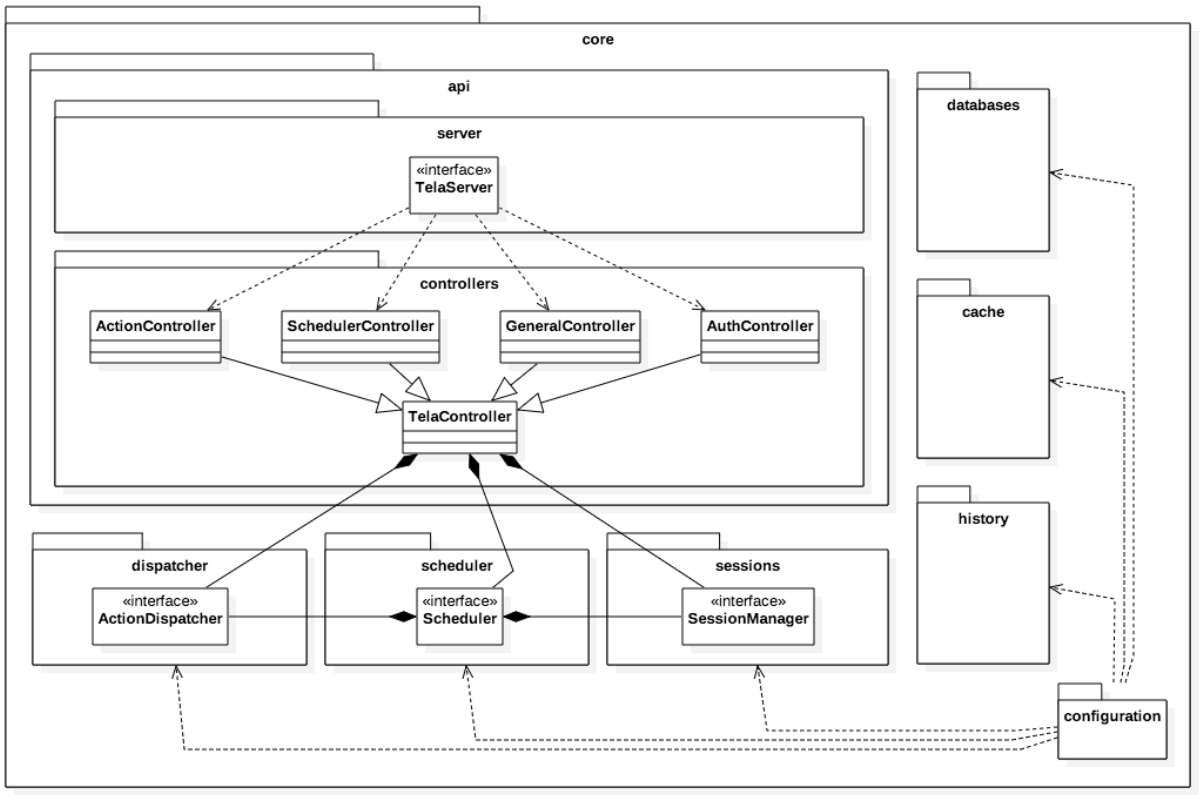


Figure 3.4 Tela Package/Class Overview Diagram

With respect to the modules, its package contains the `TelaModule` abstract class, the 'root' class of each module will inherit. Inside the package, its module will have its own package.

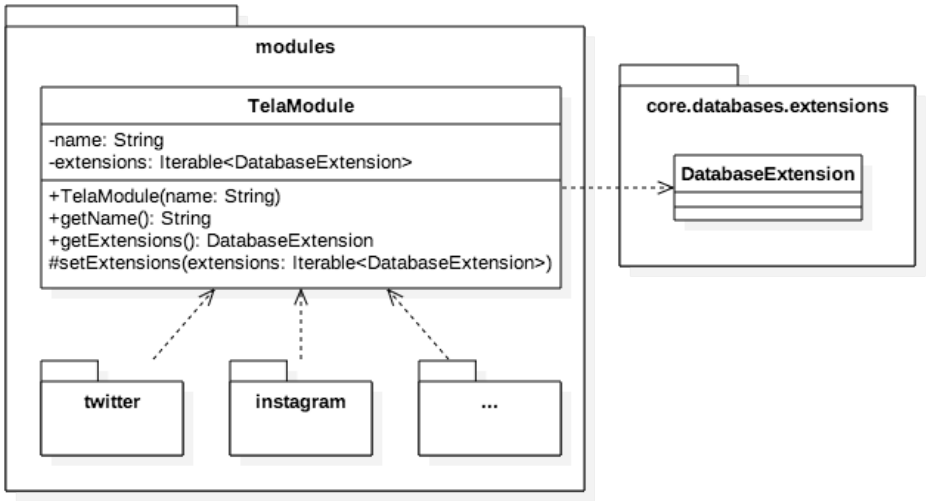


Figure 3.5 Tela Modules Package/Class Diagram

3.3 Design

3.3.1 Core

3.3.1.1 Action Dispatcher

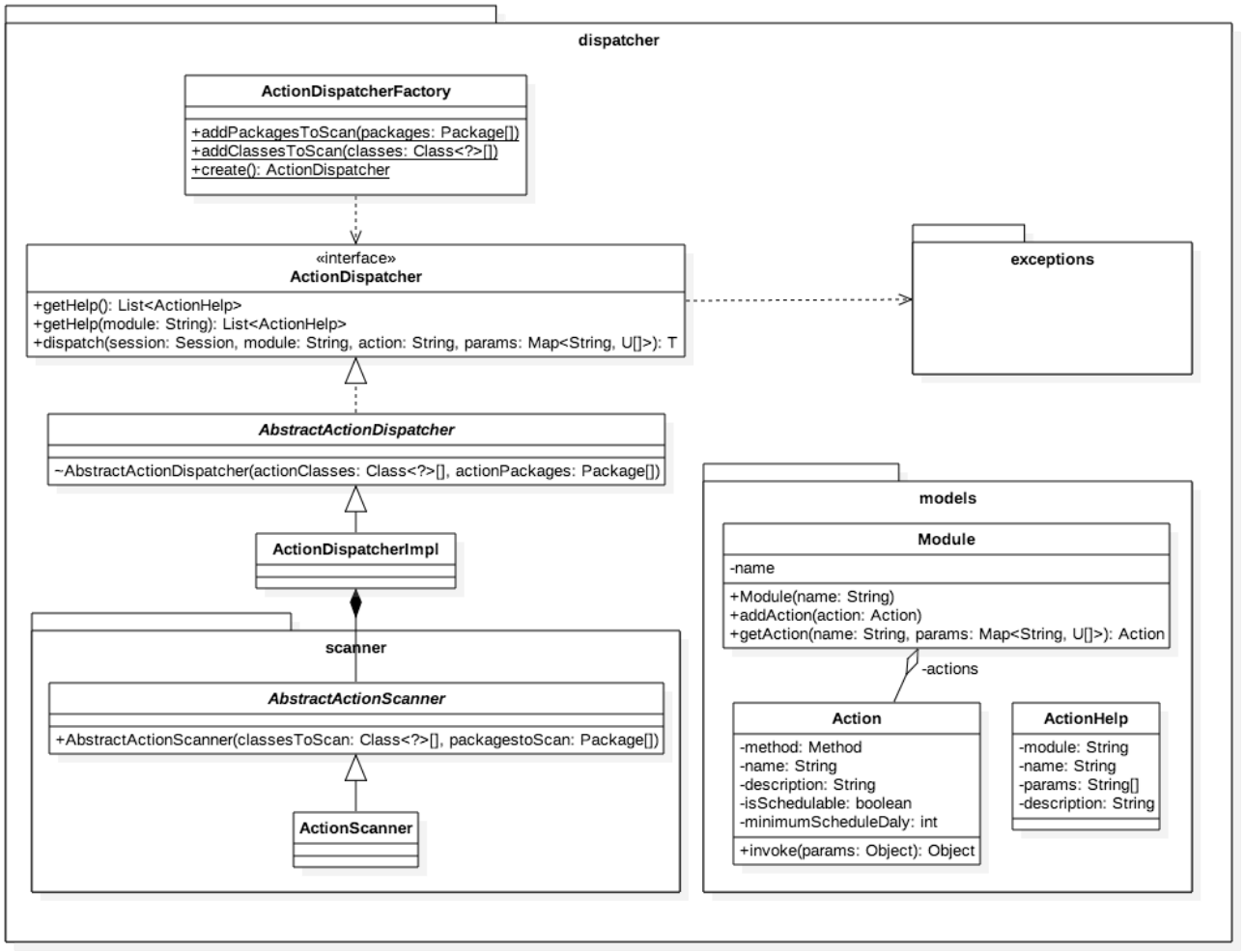


Figure 3.6 Tela Action Dispatcher Class Diagram

The `ActionDispatcher` interface is the most important entity of this package. An implementation of it will be created by `ActionDispatcherFactory`.

The package also contains the sub-package `scanner`, which is used by the implementation of the ActionDispatcher in order to scan packages and classes.

3.3.1.2 API

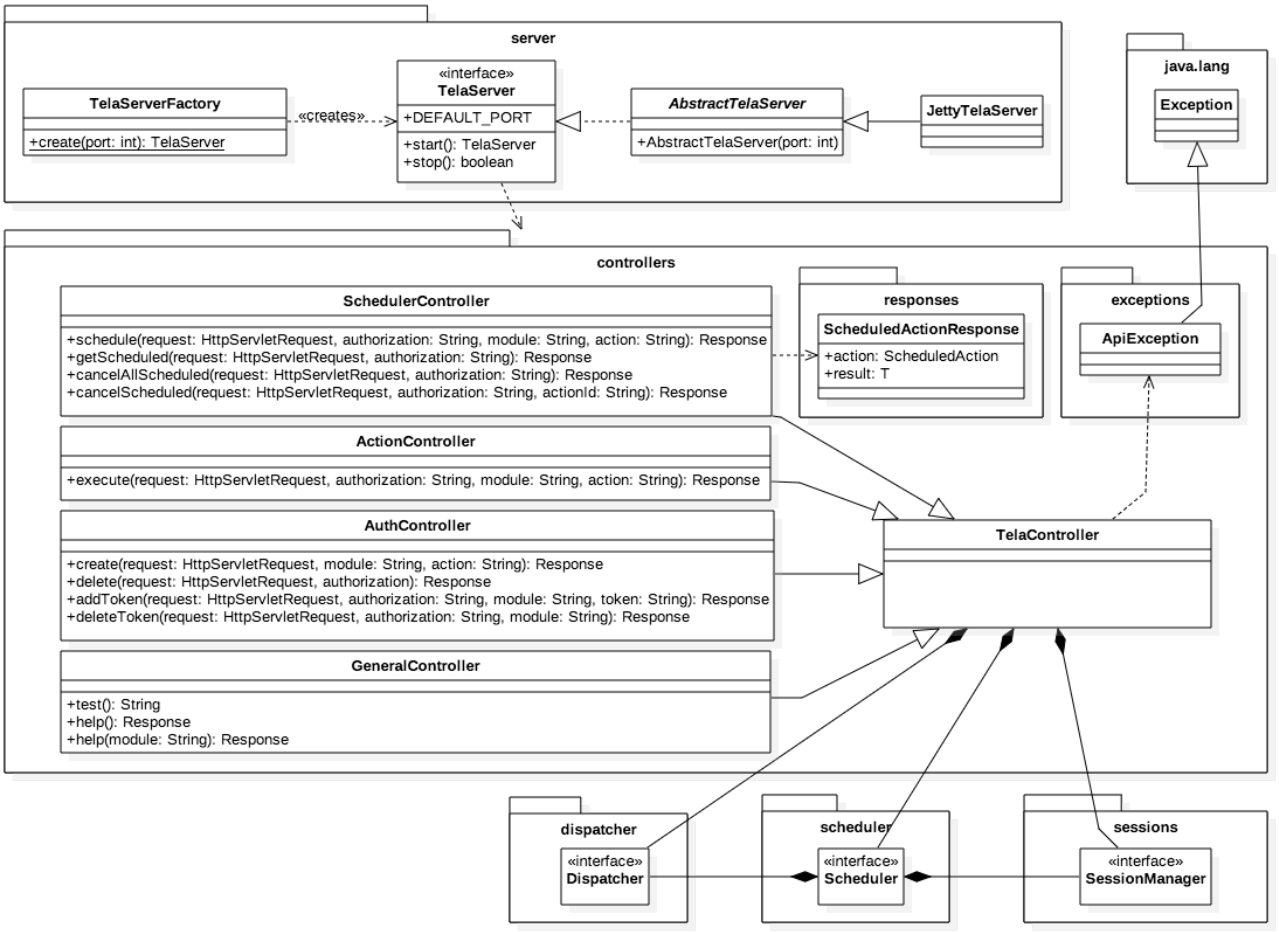


Figure 3.7 Tela API Class Diagram

The API is composed of the following packages:

- **controllers**: Endpoints of the application which will be registered in the server.
- **server**.
- **exceptions**: Exceptions related to the API.
- **responses**: Custom responses.

Regarding the controllers, it is important to observe that the four of them inherit from an abstract controller `TelaController`, which will be injected with the `Dispatcher`, `Scheduler`, and `SessionManager` components. These controllers will be registered into the implementation of the server, that will be created using the `TelaServerFactory`.

3.3.1.3 Scheduler

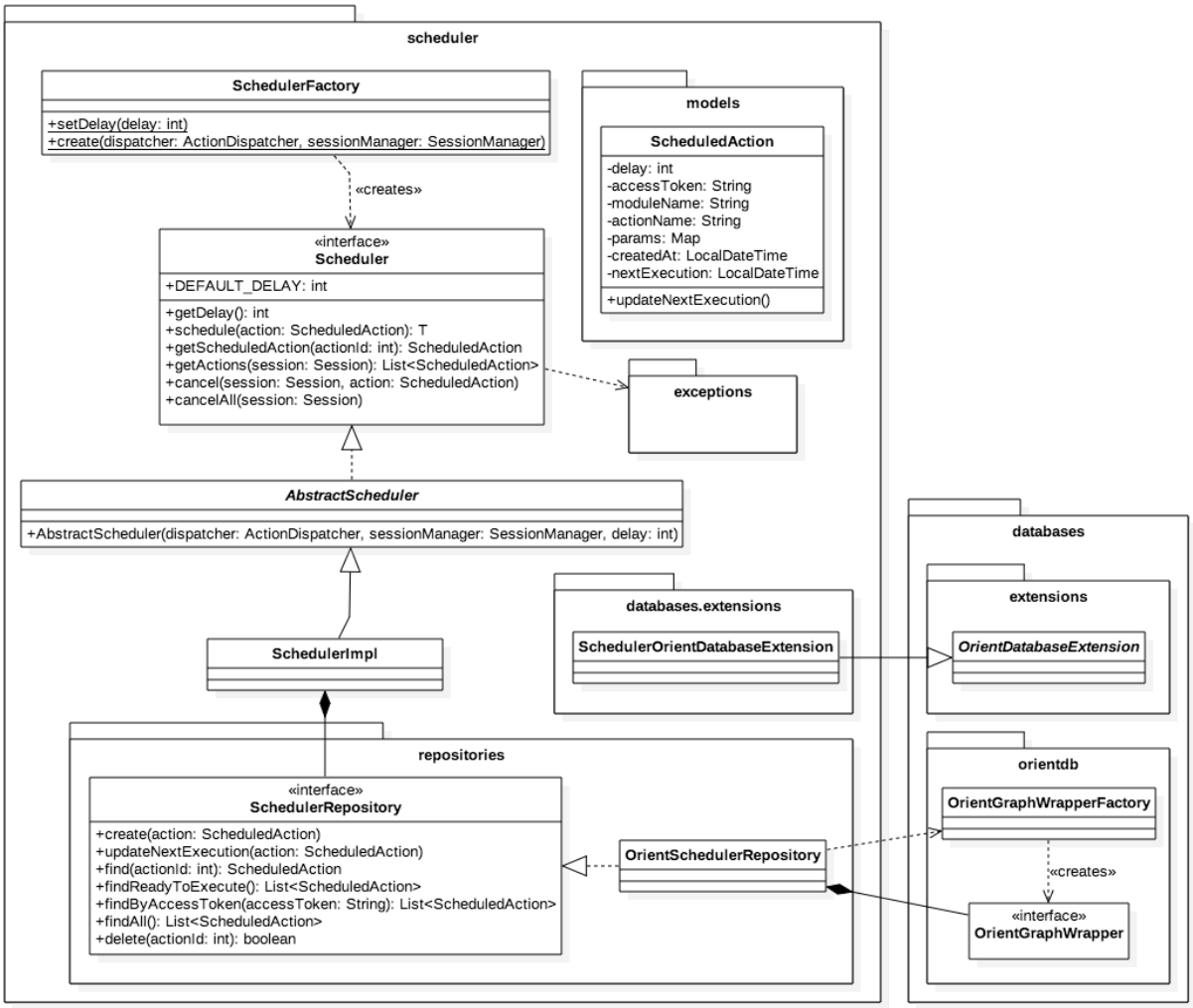


Figure 3.8 Tela Scheduler Class Diagram

The scheduler follows the design of the rest of the application: there is the main interface, `Scheduler`, and a factory `SchedulerFactory` that creates an implementation of the `Scheduler`. In addition to that, we found a `repositories` package in charge of interacting with the persistence layer, and a `SchedulerOrientDatabaseExtension` which will initiate the required schema for the database:

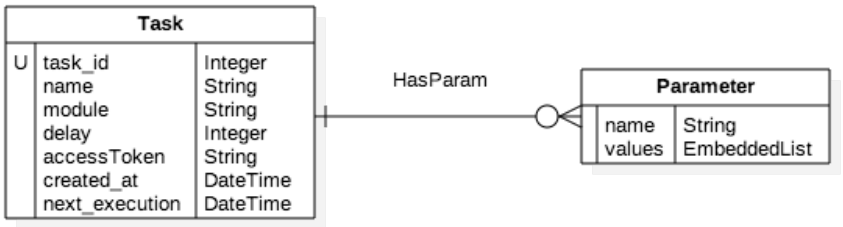


Figure 3.9 Tela Scheduler ER Diagram

3.3.1.4 Sessions

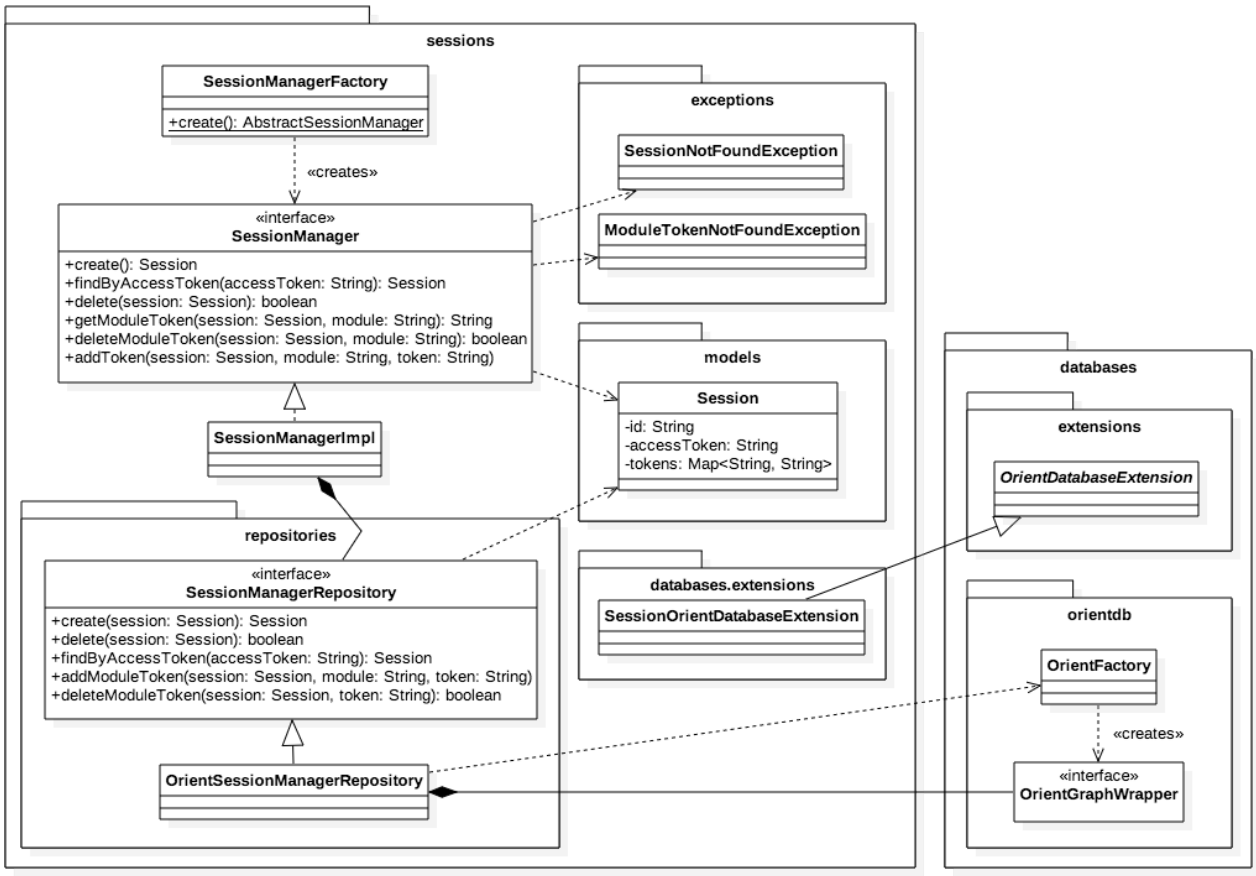


Figure 3.10 Tela Sessions Class Diagram

The Sessions package is quite similar to the Scheduler one. On one hand, we have the main interface `SessionManager`, whose implementations are created by the `SessionManagerFactory`. On the other one,the SessionManagerRepository is in charge of interacting with the database, and the `SessionOrientDatabaseExtension` contain the logic to create the required schema.

Figure 3.11 Tela Sessions ER Diagram

3.3.1.5 Cache

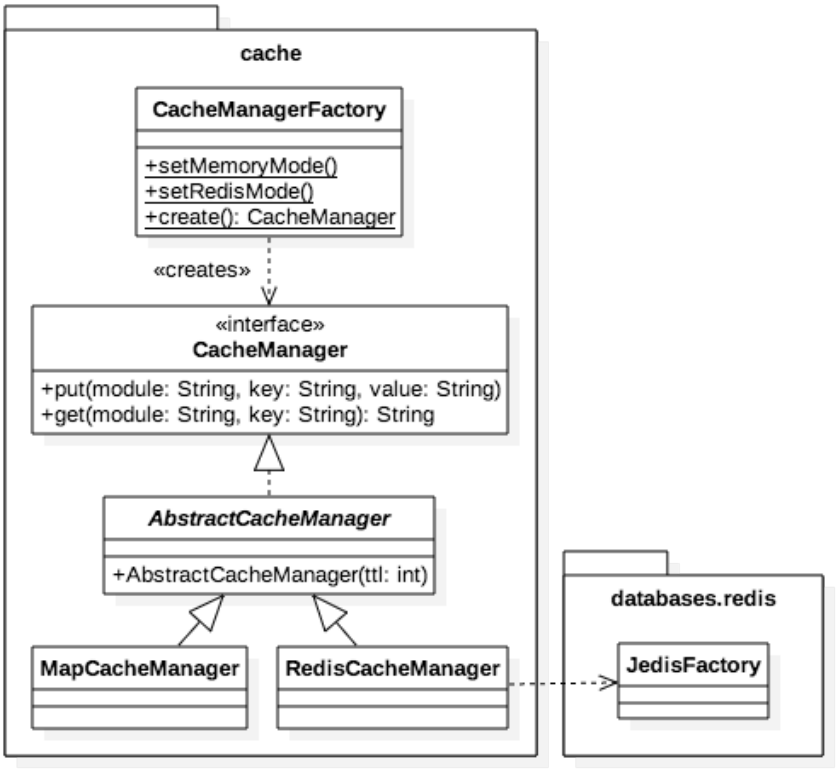


Figure 3.12 Tela Cache Class Diagram

Tela has two built in `CacheManager` implementations: an in-memory one, and another one that uses Redis (and therefore, uses the `JedisFactory`).

The `CacheManagerFactory` is in charge of instantiating the `CacheManager` implementation with the configured option, which can be set with the `setMemoryMode()` and `setRedisMode()` methods. It also sets the time to live the instances will have, with the `setTtl()` method.

3.3.1.6 History

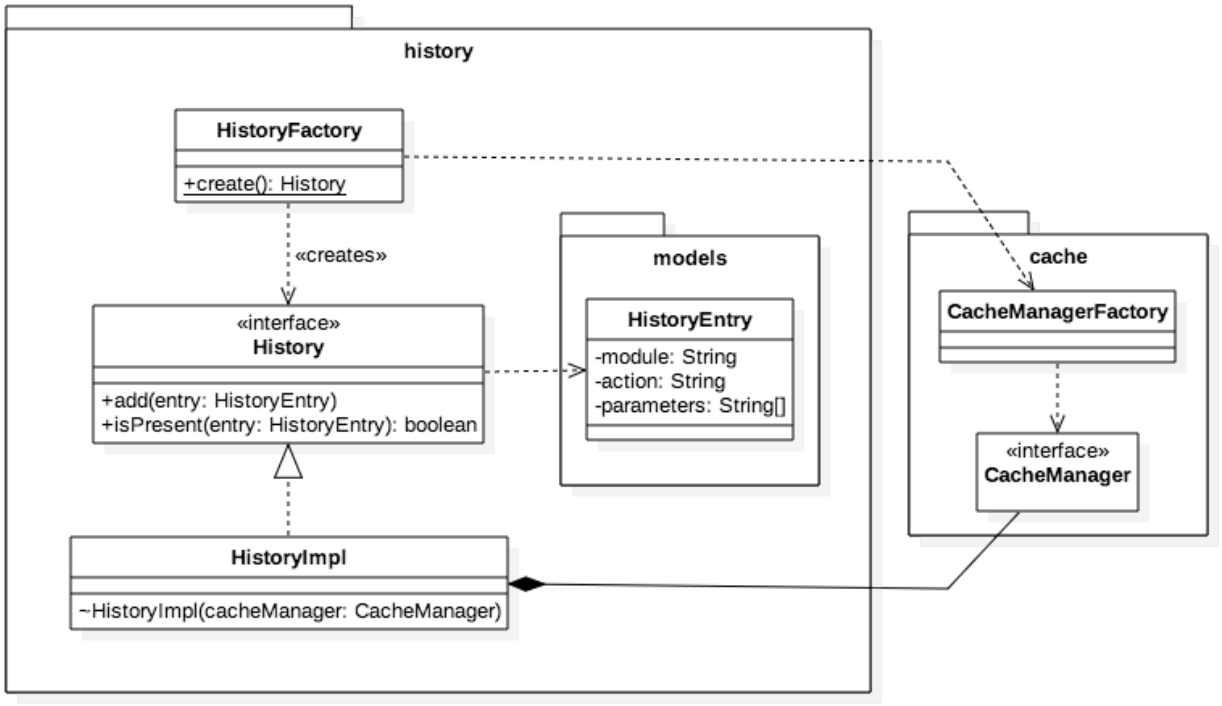


Figure 3.13 Tela History Class Diagram

The `History` component is a special case of cache, and therefore it uses the `CacheManagerFactory` described in the previous section to obtain a `CacheManager` instance.

3.3.1.7 Databases

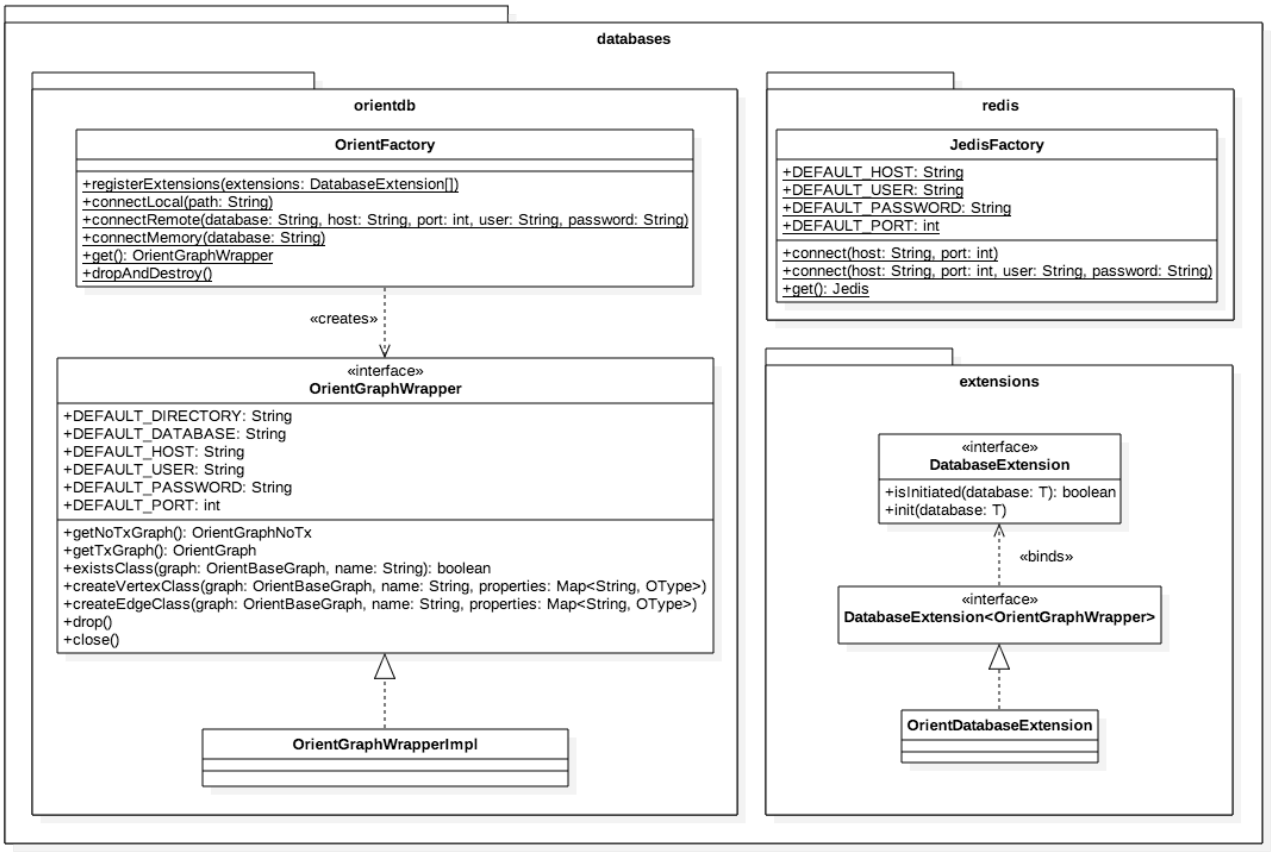


Figure 3.14 Tela Databases Class Diagram

This package contains all the classes which wrap and interact with the databases. These components will be further explained in the implementation section.

3.3.1.8 Configuration

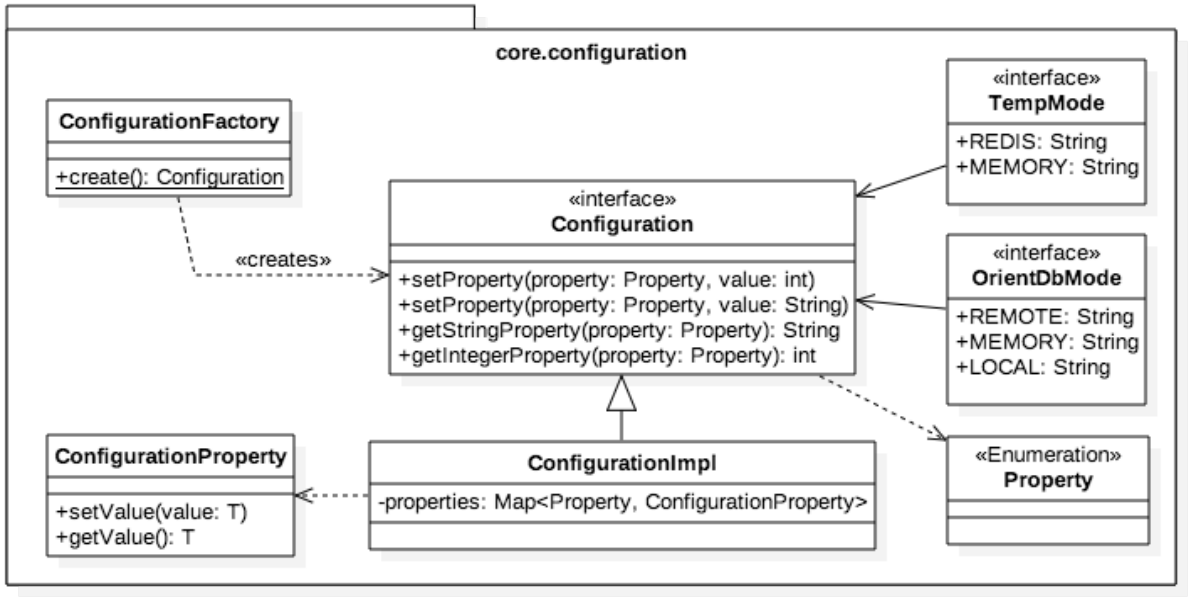


Figure 3.15 Tela Configuration Class Diagram

The `Configuration` interface is the main entity of the package, whose implementation is instantiated by the `ConfigurationFactory`. The component has a map of `ConfigurationProperty`'s. The `Property` enumeration is used as map keys.

3.3.2 Modules

The two built-in modules follow the package and file directory convention for module development, which will be covered in the Development Manual.

The package should have the same name than the module (e.g. “instagram"). Inside, the following sub-packages can be found:

- **actions**: Action classes.
- **api**: Classes that interact with external APIs (e.g. Instagram official API)
- **api.exceptions**: Exceptions related to the API.
- **api.models**: Models specific for the API, like pagination or response elements.
- **api.responses**: Data classes representing the responses from the external APIs.
- **cache**: Cache implementations.
- **databases**: Databases.
- **databases.extensions**: Database extensions.
- **models**: Models of the module (e.g. users and comments).
- **repositories**: Classes that interact with the persistence layer.

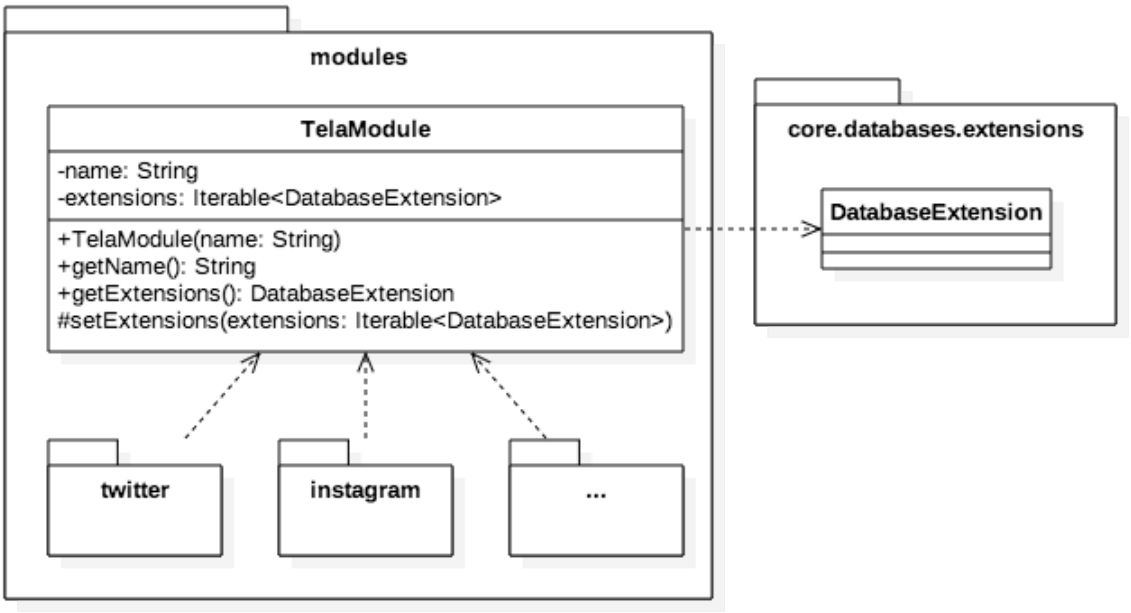


Figure 3.16 Tela Modules Package/Class Diagram (bis)

3.3.2.1 Twitter

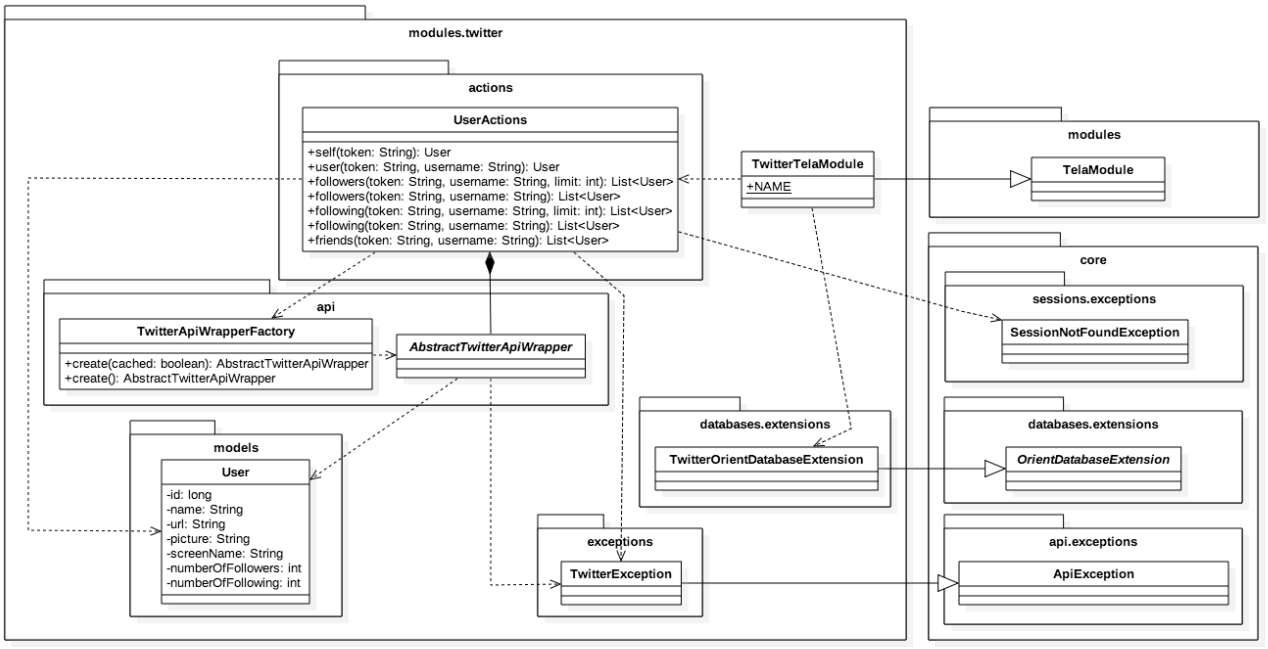


Figure 3.17 Tela Twitter Class Diagram (I)

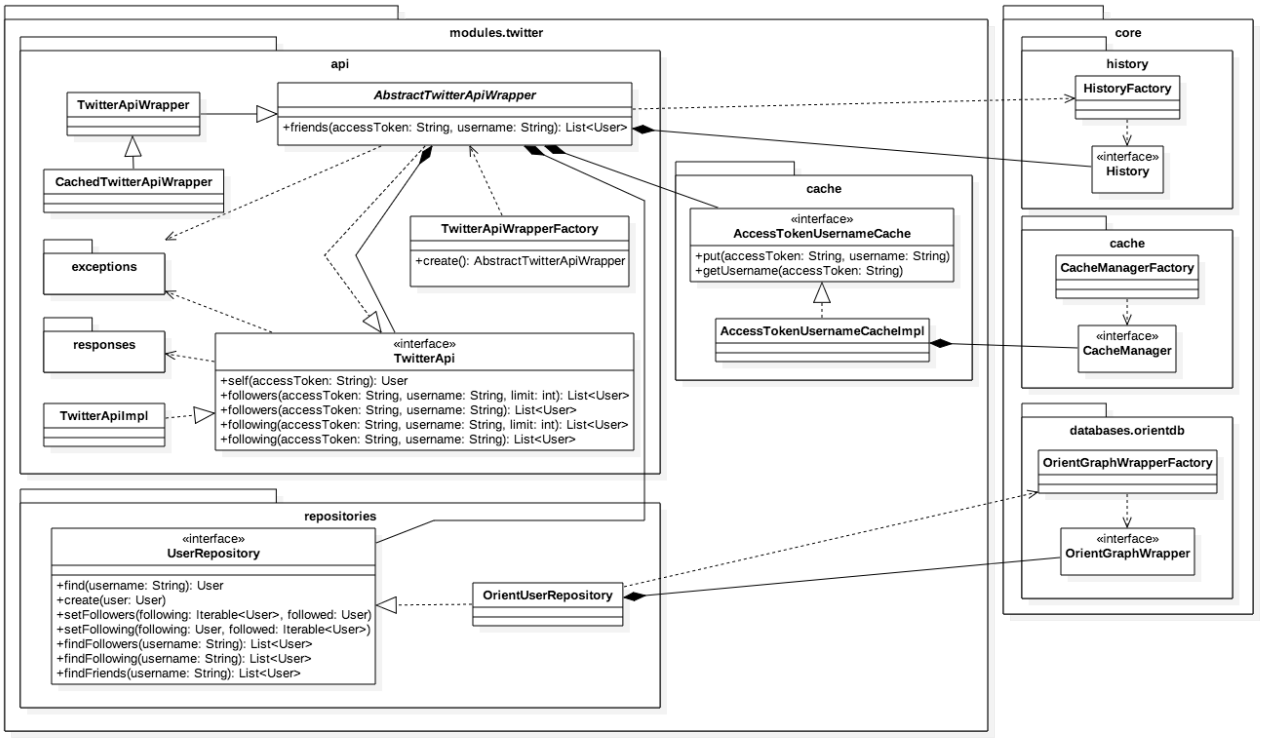


Figure 3.18 Tela Twitter Class Diagram (II)

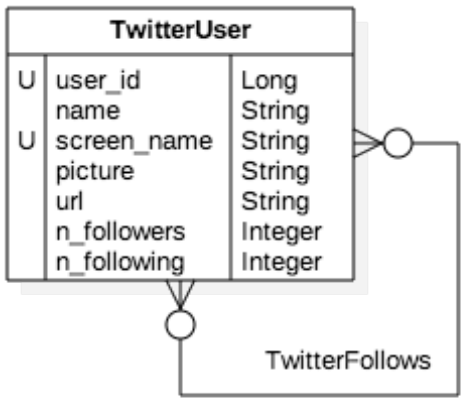


Figure 3.19 Tela Twitter ER Diagram

3.3.2.2 Instagram

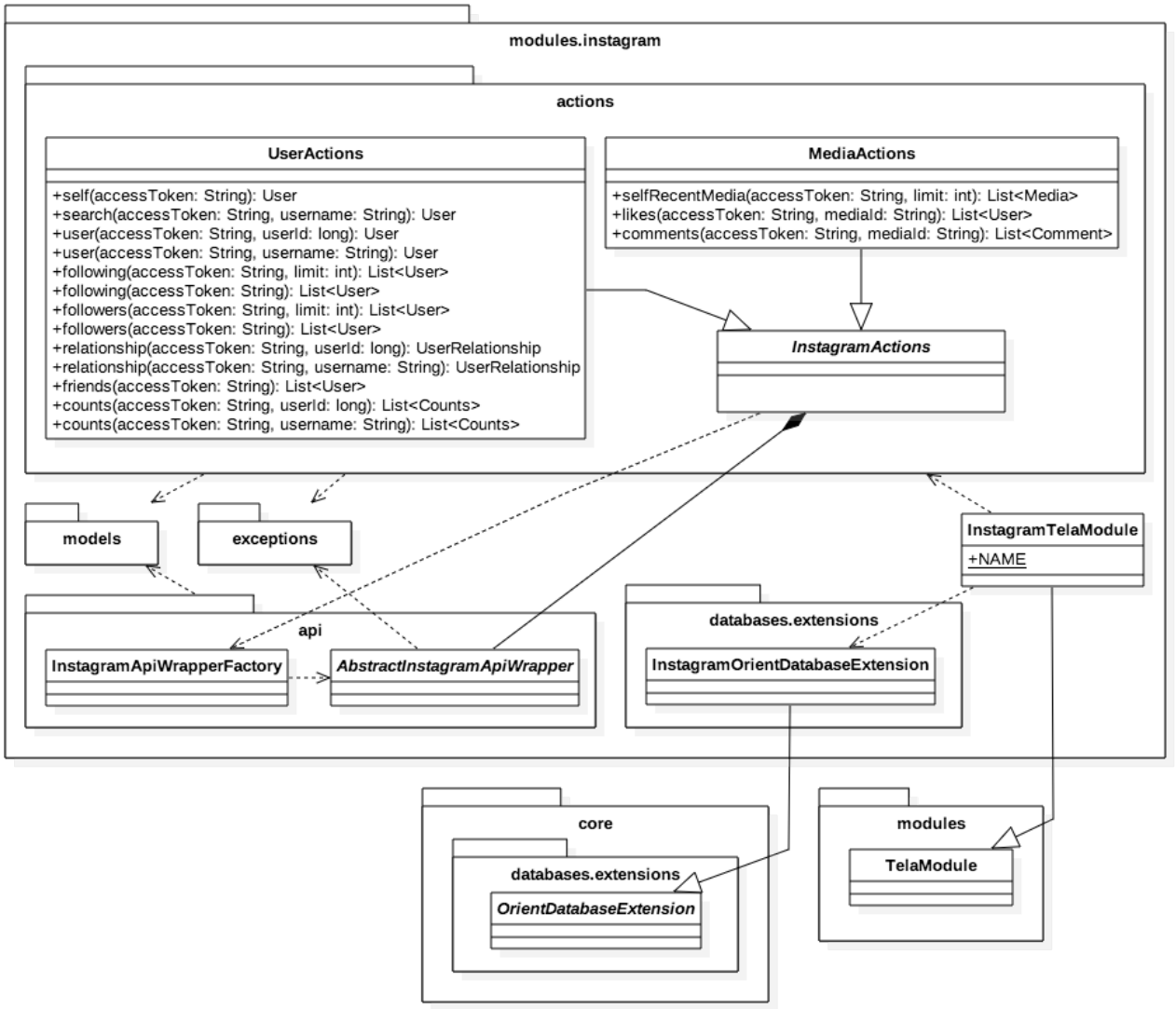


Figure 3.20 Tela Instagram Class Diagram (I)

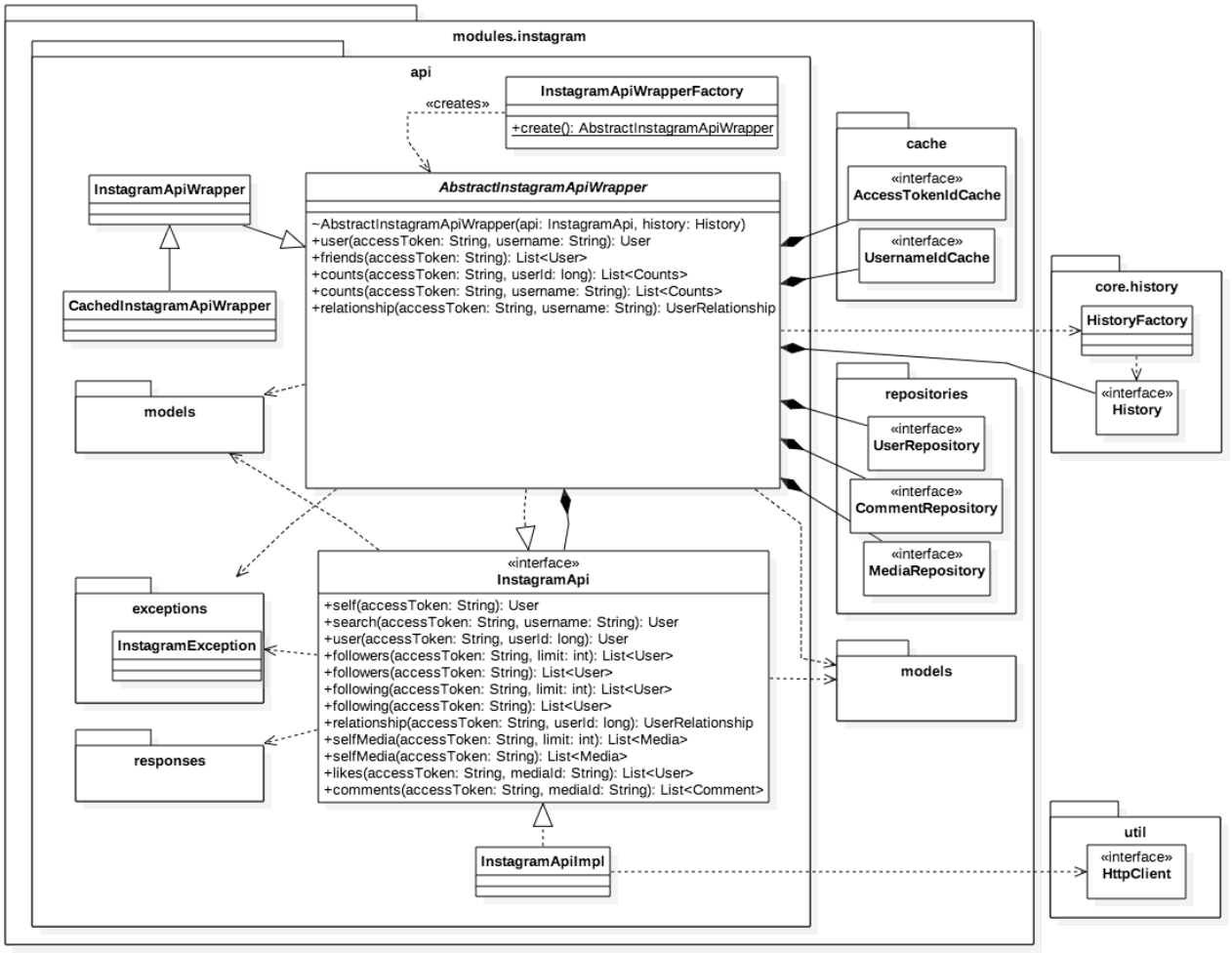


Figure 3.21 Tela Instagram Class Diagram (II)

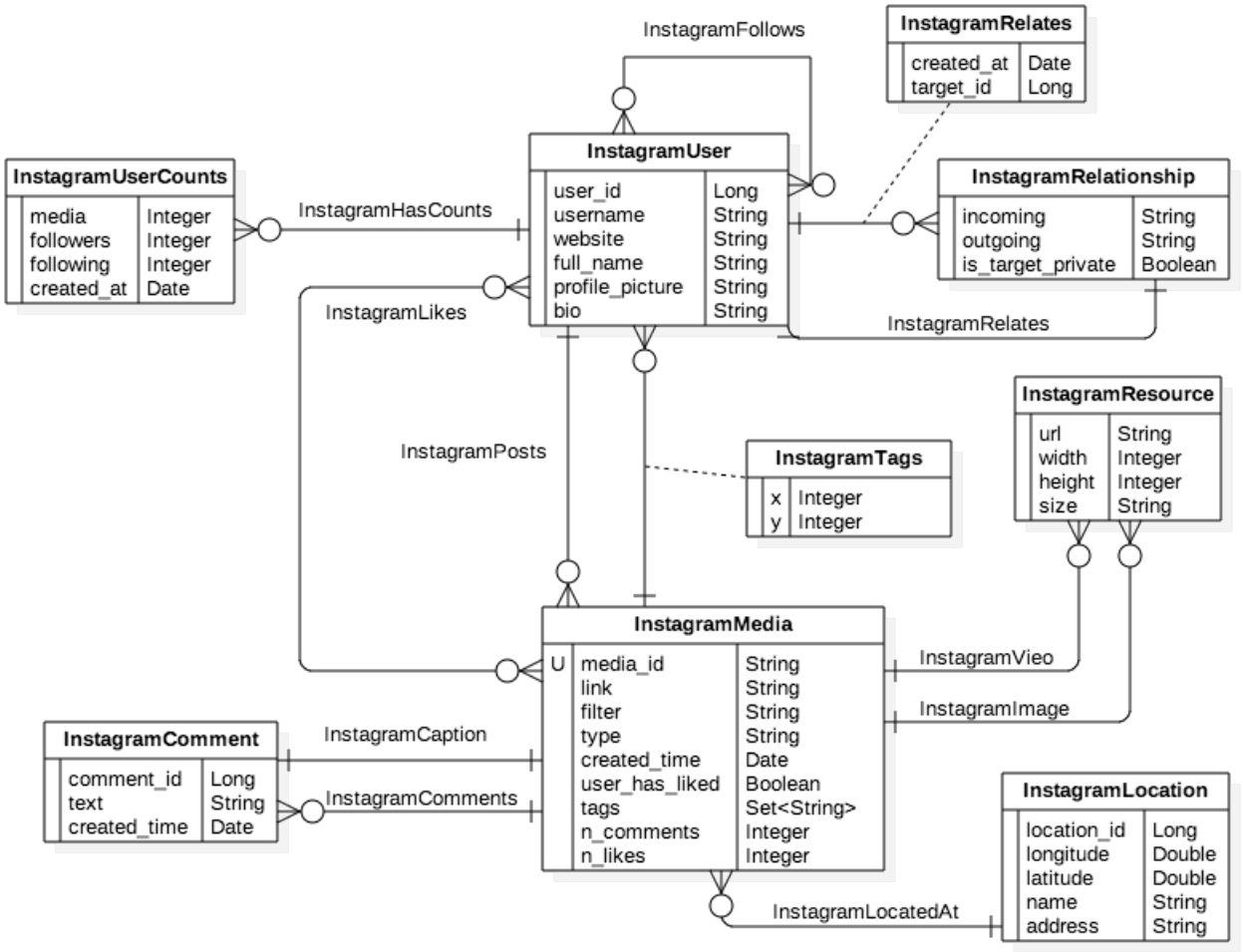


Figure 3.22 Tela Instagram ER Diagram

3.3.3 Assembler

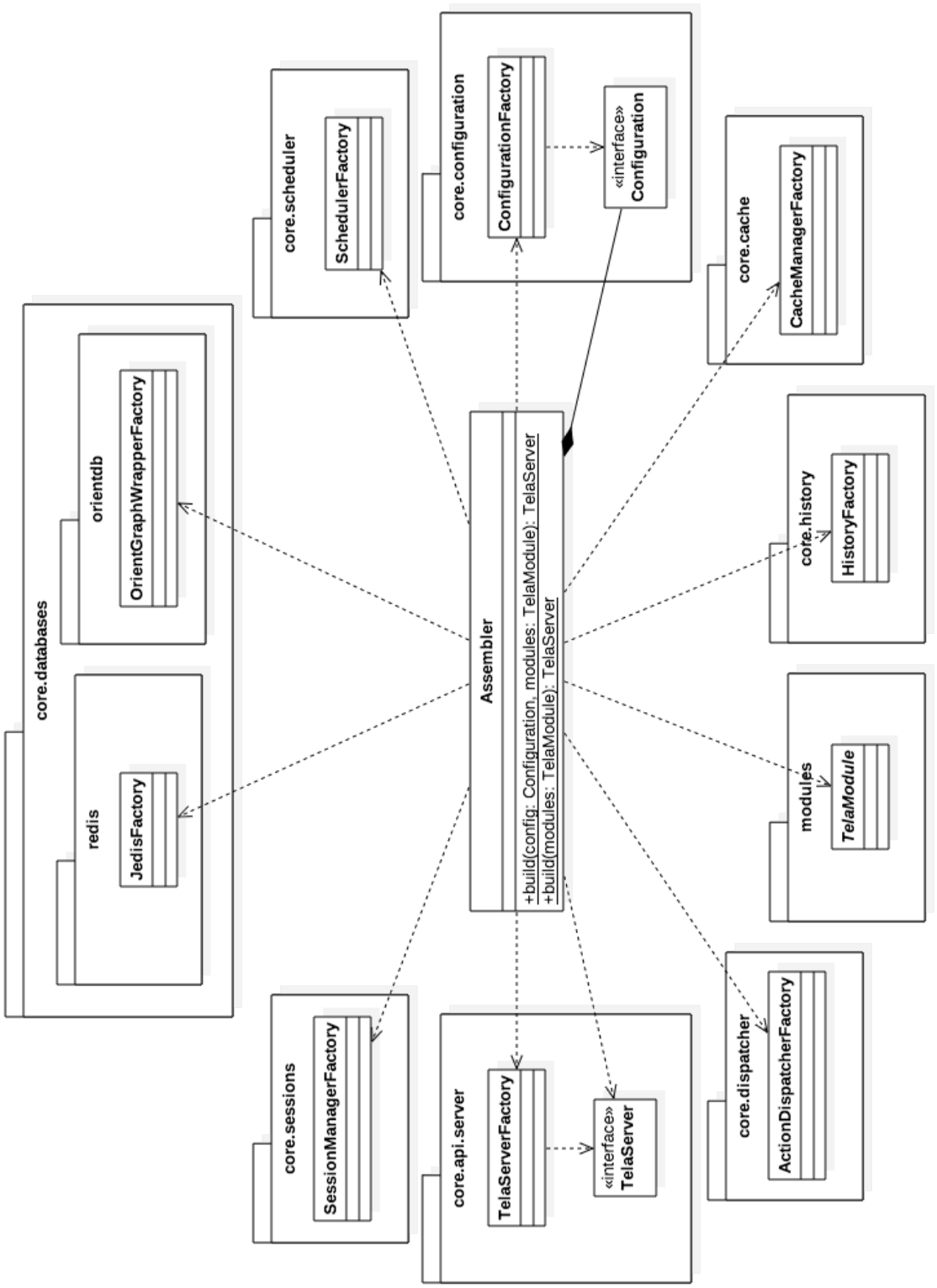


Figure 3.23 Tela Assembler Class Diagram

The `Assembler` reads the configuration and configures, builds and injects the main components of the core, along with the modules. For this reason, is tightly coupled to the rest of components.

3.4 Test Plan

3.4.1 Automated tests

This system is a framework, which means that developers will build applications on the top of it. Therefore, it has to be as secure and reliable as possible. In addition to that, developers should be able to extend the framework adding modules, even modifying components of the core.

For these reasons, automated tests of both unit and integration tests are a crucial part of the system.

In this section, we will specify some of the automated tests we will implement, along with its type:

- "+": positive test, whose purpose is to check the functionality works as expected.
- "-": negative test, whose objective is to break the functionality.

3.4.1.1 Core

Component: API			
TelaServer			
Name	**Variation**	**Type**	**Functionality Tested**
start		+	The server starts
error404		+	The server gives the customized 404 error message
TelaServerFactory			
Name	**Variation**	**Type**	**Functionality Tested**
create		+	The server is created
create	CorrectPort	+	The server runs at the desired port
TelaController			
Name	**Variation**	**Type**	**Functionality Tested**
extractSession		+	The session is extracted from the Authorization header
extractSession	InvalidMethod	-	Only Bearer Authorization is valid
extractSession	EmptyHeader	-	Empty header throws exception
extractSession	NotFound	-	Session not found throws exception
getParameter		+	The parameter is obtained
getParameter	firstOfArray	+	Retrieves the first entry from the array
getParameter	NotExists	-	Return default if the parameter is not set
getParameter	IsEmpty	-	Return default if the parameter is empty
getIntParameter		+	The parameter is obtained

getIntParameter	firstOfArray	+	Retrieves the first entry from the array
getIntParameter	NotExists	-	Return default if the parameter is not set
getIntParameter	IsEmpty	-	Return default if the parameter is empty
getIntParameter	NoInteger	-	Non-integer parameter throws exception
ActionController			
Name	**Variation**	**Type**	**Functionality Tested**
execute		+	Action is correctly executed
execute	WithParams	+	Action is correctly executed
execute	WithArrayParam	+	Action is correctly executed
execute	WithInvalidTypeParams	-	422 error returned
execute	WithMissingParams	-	404 error returned
execute	WithInvalidParams	-	404 error returned
execute	WithoutSession	-	403 error returned
execute	WithNotExistingModule	-	404 error returned
execute	WithNotExistingAction	-	404 error returned
SchedulerController			
Name	**Variation**	**Type**	**Functionality Tested**
schedule		+	Action is scheduled
schedule	WithoutDelay	+	Default delay is used
schedule	WithParams	+	Action is scheduled
schedule	WithoutRequiredParam	-	403 error returned
schedule	WithInvalidParamType	-	404 error returned
schedule	WithInvalidDelay	-	422 error returned
schedule	WithoutSession	-	403 error returned
schedule	WithInvalidSession	-	403 error returned
schedule	WithActionNotDefined	-	404 error returned
schedule	WithActionNotSchedulable	-	404 error returned
schedule	Repeated	-	409 error returned
getScheduled		+	Actions returned
getScheduled	Empty	+	Empty list returned
getScheduled	WithoutSession	-	403 error returned
getScheduled	WithInvalidSession	-	403 error returned
cancel		+	Action is cancelled
cancel	WithoutSession	-	403 error returned

cancel	WithInvalidSession	-	403 error returned
cancel	WithInvalidAction	-	404 error returned
cancel	Forbidden	-	403 error returned
cancelAll		+	Actions are cancelled
cancelAll	WithoutSession	-	403 error returned
cancelAll	WithInvalidSession	-	403 error returned

Component: Dispatcher			
ActionDispatcherImpl			
Name	**Variation**	**Type**	**Functionality Tested**
dispatch	WithoutParamters	+	Action is correctly dispatched
dispatch	WithStringParameter	+	Action is correctly dispatched
dispatch	WithIntegerParam	+	Action is correctly dispatched
dispatch	WithCastedIntegerParam	+	Action is correctly dispatched
dispatch	WithArrayParam	+	Action is correctly dispatched
dispatch	WithTwoIntegerParams	+	Action is correctly dispatched
dispatch	WithInvalidIntegerParam	-	Exception is thrown
dispatch	ThrowingException	+	The original exception is thrown again
ActionScanner			
Name	**Variation**	**Type**	**Functionality Tested**
scan	Class	+	The class is scanned
scan	Package	+	The package is scanned
scan	Null	-	Empty map is returned

Component: Scheduler			
SchedulerImpl			
Name	**Variation**	**Type**	**Functionality Tested**
schedule		+	Action is scheduled
schedule	WithParameters	+	Action is scheduled
schedule	DifferentTimes	+	Actions are scheduled
schedule	BelowMinimumDelay	-	Default delay is used
schedule	Null	-	Exception is thrown
schedule	NotSchedulable	-	Exception is thrown
schedule	WithBadSession	-	Exception is thrown

schedule	ActionNotDefined	-	Exception is thrown
schedule	WithoutRequiredParameters	-	Exception is thrown
schedule	WithInvalidParameterType	-	Exception is thrown
close		+	Scheduler is stopped
close	ActionsPersists	+	Actions are not deleted
cancel		+	Scheduling is cancelled
ScheduledAction			
Name	**Variation**	**Type**	**Functionality Tested**
serialization		+	Model is deserialized
constructor	WithoutDates	+	Dates are initiated
updateNextExecution		+	The next execution is correct
equals	WithoutParams	+	Equality comparison works
equals	WithParams	+	Equality comparison works
OrientSchedulerRepository			
Name	**Variation**	**Type**	**Functionality Tested**
create	WithoutParams	+	Action is stored
create	WithParams	+	Parameters are stored too
contains		+	Returns true if it exists
contains	NotExisting	+	Returns false if it does not exist
findReadyToExecute		+	The expected actions are returned
findAll		+	All the actions are returned
findAll	Empty	+	Empty list is returned
findBySession		+	Correct action is returned
findBySession	NotExisting	+	Empty list is returned
delete		+	Action is deleted
delete	ParamsAreDeleted	+	Parameters are deleted too
deleteByAccessToken		+	Action is deleted
deleteByAccessToken	NotExisting	+	No action is deleted

Component: Sessions			
SessionManagerImpl			
Name	**Variation**	**Type**	**Functionality Tested**
create		+	A session is created
create	WithModuleToken	+	A session with module is created

create	WithModuleTokenNullModule	-	Creating a session with null module throws exception
create	WithModuleTokenNullToken	-	Creating a session with null token throws exception
existsByAccessToken		+	Works properly
existsByAccessToken	WithNull	-	Null access token returns false
delete		+	The session is deleted
delete	WithToken	+	The tokens of the module are deleted
getModuleToken		+	The token is retrieved
getModuleToken	NotExistingSession	-	Exception is thrown
getModuleToken	NotExistingToken	-	Exception is thrown
getModuleToken	AfterDeleteSession	-	The session is deleted
deleteModuleToken		+	The token is deleted
deleteModuleToken	NotSessionFound	-	Not existing session does nothing
deleteModuleToken	NotExisting	-	Not existing token does nothing
deleteModuleToken	WithOtherTokens	+	Deleting a token does not affect the other module tokens of the session
findByModuleByToken		+	The session is returned
findByModuleByToken	NotExisting	-	Not found return null
deleteByModuleToken		+	The session is deleted
deleteByModuleToken	NotExistingToken	-	Not existing token does nothing
deleteByModuleToken	OtherTokens	+	The whole session is deleted, including other tokens
addToken		+	A token is added
addToken	SessionNotExisting	-	Cannot add token to non-existing session
addToken	AlreadyExisting	+	A token overwrites the previous
OrientSessionManagerRepository			
Name	**Variation**	**Type**	**Functionality Tested**
create	WithoutModules	+	The session is created
create	WithModules	+	The modules are created too
delete		+	The session is deleted
delete	WithTokens	+	The tokens are deleted too
delete	NotExisting	-	Nothing happens, returns false
findByAccessToken		+	The session is returned

findByAccessToken	WithTokens	+	The tokens are returned too
findByAccessToken	NotExisting	+	Null is returned
existsByAccessToken		+	Returns true if it does existing
existsByAccessToken	NotExisting	+	Returns false if it does not exist
addModuleToken		+	Token is added
addModuleToken	MultipleTokens	+	More than one token can be added
addModuleToken	NotExistingSession	-	Does nothing, returns false
findByModuleToken		+	Session is returned
findByModuleToken	NotExisting	+	Null is returned
deleteModuleToken		+	Token is returned
deleteModuleToken	NotExistingSession	-	Does nothing, returns false
deleteModuleToken	NotExistingToken	-	Does nothing, returns false

Component: History			
HistoryImpl			
Name	**Variation**	**Type**	**Functionality Tested**
add		+	Entry is added
add	Expires	+	TTL is respected
add	OverwriteTtl	+	Overwrite updates the TTL
add	Null	-	Add null throws exception
add	DifferentParameters	+	Entries with different parameters can be added
isPresent	Null	-	Null entry returns false

Component: Cache			
MapCacheManager & RedisCacheManager			
Name	**Variation**	**Type**	**Functionality Tested**
put		+	Entry is cached
put	Several	+	Several entries are cached
put	Null	-	Caching null has no effect
put	Overwrite	+	Entries with equal module and key are overwritten
put	NullOverwrites	+	Null value overwrites the previous entry
put	ModuleCollision	+	Entries with equal module, different key, can coexist
get	NotExisting	-	Get not existing return null
clear		+	Entries are deleted

CacheManagerFactory			
Name	**Variation**	**Type**	**Functionality Tested**
setMemoryMode		+	Memory mode is set
setRedisMode		+	Redis mode is set

Component: Databases			
OrientGrapWrapper			
Name	**Variation**	**Type**	**Functionality Tested**
init		+	Graph extension is initiated

Component: Configuration			
ConfigurationImpl			
Name	**Variation**	**Type**	**Functionality Tested**
getStringProperty	File	+	Property is read
getStringProperty	Env	+	Property is read before file
getStringProperty	Programmatically	+	Property is read before env
getIntegerProperty	File	+	Property is read
getIntegerProperty	Env	+	Property is read before file
getIntegerProperty	Programmatically	+	Property is read before env

3.4.1.2 Modules

3.4.1.2.1 Instagram

Component: API &Actions			
UserActionsTest			
Name	**Variation**	**Type**	**Functionality Tested**
self		+	The auth user is returned
search		+	The basic information of the user with the given username is returned
userById		+	The full information of the user with the given ID is returned
userByUsername		+	The full information of the user with the given username is returned
followers	WithoutLimit	+	All the followers of the auth user are returned
followers	WithLimit	+	The expected number of followers are returned

following	WithoutLimit	+	All the following of the auth user are returned
following	WithLimit	+	The expected number of following are returned
friends		+	The friends of the auth user are returned
counts	ById	+	The counts of the user with the given ID are returned
counts	ByUsername	+	The counts of the user with the given username are returned
relationship	ById	+	The relationship between the auth user and the user with the given ID is returned
relationship	ByUsername	+	The relationship between the auth user and the user with the given username is returned
MediaActionsTest			
Name	**Variation**	**Type**	**Functionality Tested**
selfMedia	WithoutLimit	+	All the media of the auth user is returned
selfMedia	WithLimit	+	The expected number of media is returned
likes		+	The likes of the given media are returned
comments		+	The comments of the given media are returned
CachedInstagramApiWrapper & InstagramApiWrapper			
Name	**Variation**	**Type**	**Functionality Tested**
self		+	The auth user is returned
search		+	The basic information of the user with the given username is returned
userById		+	The full information of the user with the given ID is returned
userByUsername		+	The full information of the user with the given username is returned
followers	WithoutLimit	+	All the followers of the auth user are returned
followers	WithLimit	+	The expected number of followers are returned
following	WithoutLimit	+	All the following of the auth user are returned
following	WithLimit	+	The expected number of following are returned
friends		+	The friends of the auth user are returned
counts	ById	+	The counts of the user with the given ID are returned
counts	ByUsername	+	The counts of the user with the given username are returned
relationship	ById	+	The relationship between the auth user and the user with the given ID is returned

relationship	ByUsername	+	The relationship between the auth user and the user with the given username is returned
selfMedia	WithoutLimit	+	All the media of the auth user is returned
selfMedia	WithLimit	+	The expected number of media is returned
likes		+	The likes of the given media are returned
comments		+	The comments of the given media are returned
CachedInstagramApiWrapper & InstagramApiWrapper			
Name	**Variation**	**Type**	**Functionality Tested**
self		+	The auth user is returned
search		+	The basic information of the user with the given username is returned
user		+	The full information of the user is returned
followers	WithoutLimit	+	All the followers of the auth user are returned
followers	WithLimit	+	The expected number of followers are returned
following	WithoutLimit	+	All the following of the auth user are returned
following	WithLimit	+	The expected number of following are returned
relationship		+	The relationship between the auth user and the user with the given ID is returned
selfMedia	WithoutLimit	+	All the media of the auth user is returned
selfMedia	WithLimit	+	The expected number of media is returned
likes		+	The likes of the given media are returned
comments		+	The comments of the given media are returned

Component: Models			
Media			
Name	**Variation**	**Type**	**Functionality Tested**
imageUnwrap		+	An image is properly deserialized
videoUnwrap		+	A video is properly deserialized

Component: Repositories			
OrientUserRepository			
Name	**Variation**	**Type**	**Functionality Tested**
create	Basic	+	A basic user is created
create	Full	+	A full user is created
Create	Updating	+	If the user already exists, it is updated

findFollowers		+	Followers are returned
findFollowing		+	Following are returned
findFriends		+	Friends are returned
following	WithoutLimit	+	All the following of the auth user are returned
following	WithLimit	+	The expected number of following are returned
friends		+	The friends of the auth user are returned
counts		+	The counts are returned, sorted by time
findLatestCount		+	The latest count is returned
findLatestRelationship		+	The latest relationship is returned
findRelationships		+	All the relationships are returned
OrientMediaRepository			
Name	**Variation**	**Type**	**Functionality Tested**
create	savesSimpleProperties	+	The media is created
create	savesWithoutUserHasLiked	+	The media is created, null userHasLiked
create	savesUpdatingExistingUser	+	The user is updated
create	withoutCaption	+	The media is created, null caption
create	savesCaption	+	The media and caption are created
create	withComments	+	Media and comments are created
create	withLikes	+	Media and likes are created
create	withTags	+	Media an tags are created
create	Location	+	Media and location are created
create	Images	+	Images of the media are created
create	Videos	+	Videos of the media are created
findAll		+	All the media of a user is returned
findLatest		+	The latest media is returned
OrientCommentRepository			
Name	**Variation**	**Type**	**Functionality Tested**
create		+	The comment is created
create	Several	+	Several comments can be created
update		+	The comment is updated
findAll		+	All the comments are returned

3.4.1.2.2 Twitter

Component: API &Actions			
UserActionsTest, CachedTwitterApiWrapper & TwitterApiWrapper			
Name	**Variation**	**Type**	**Functionality Tested**
self		+	The auth user is returned
user		+	The user with the given username is returned
followers	WithoutLimit	+	All the followers of the given user are returned
followers	WithLimit	+	The expected number of followers are returned
following	WithoutLimit	+	All the following of the given user are returned
following	WithLimit	+	The expected number of following are returned
friends		+	The friends of the given user are returned
TwitterApiImpl			
Name	**Variation**	**Type**	**Functionality Tested**
self		+	The auth user is returned
user		+	The user with the given username is returned
followers	WithoutLimit	+	All the followers of the given user are returned
followers	WithLimit	+	The expected number of followers are returned
following	WithoutLimit	+	All the following of the given user are returned
following	WithLimit	+	The expected number of following are returned

Component: Cache			
AccessTokenUsernameCacheImpl			
Name	**Variation**	**Type**	**Functionality Tested**
put		+	The username is cached
putSeveral		+	Several usernames can be cached

Component: Models			
User			
Name	**Variation**	**Type**	**Functionality Tested**
deserialize		+	The user is properly deserialized

Component: Repositories			
OrientUserRepository			
Name	**Variation**	**Type**	**Functionality Tested**
create		+	A user is created

setFollowers		+	The followers are stored
setFollowing		+	The following are stored
findFriends		+	Friends are returned

3.4.2 Manual tests

Although we will try to automate as many tests as possible, there are certain cases where this is not a good approach. For example, the interaction of the system with the real APIs of the social networks. APIs had a limit of requests per hour. Even worse, some (e.g. Instagram) force developers to use a Sandbox mode, where each user we want to retrieve information from has to grant permission.

For these APIs, we will mock the responses of the requests during the automated tests. We will not elaborate a list of manual tests we have to follow, as this will be done in the next system, Tela CLI. Tela CLI completely mirrors the functionality of Tela. During its manual tests, we will also be testing Tela, without having to manually interact with it using a standard REST client.

3.5 System Implementation

3.5.1 Programming Languages and Technologies

3.5.1.1 Java 8

The main programming language employed in the implementation of Tela is Java.

The Java Virtual Machine is fast, well-tested, well documented and under active development; qualities that make Java (or other languages that run on top of the JVM) pretty attractive, especially for big technological companies like Spotify (Johansson, 2016).

In addition to that, Java scales well, and it is very maintainable (especially compared against non-typed languages), two qualities indispensable for API development.

Another reason to take into account is popularity. Java is the most popular language in 2016, according to the rankings published by TIOBE (TIOBE, 2016) and PYPL (PYPL, 2016). This factor does not potentially affect the usage of Tela, as the REST API applications use to communicate with the framework is platform agnostic, but it does if we consider the possibility of other developers contributing to the open source project.

Regarding the version, Tela requires a(t least) Java 8, as features like lambdas and streams have been used.

- Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
- Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)

3.5.1.2 Graph Database: OrientDB (+ OrientDB SQL)

Social Networks are basically complex graphs, we nodes (e.g. users, pictures, places…) are connected to each other. Therefore, it seems like the best way to represent this information is by directly using a database that supports a Graph Model.

Traditional (relational) databases compute relationships expensively at query time, forcing developers to create indexes and optimization tweaks. On the contrary, graph databases stores connections as first class citizens (Neo4j, 2016) and therefore traversing from one vertex to another has a constant O(1) complexity.

The most popular Graph database by far is Neo4j (DB-Engines Ranking, 2016). However, the final choice was OrientDB, for several reasons (OrientDB, 2016):

- First, we were already familiar with the product.
- Neo4j uses an (A)GPL license, which might be a problem for developers who would like embed Tela within their close systems. On the contrary, OrientDB has a permissive Apache 2.0 license.
- Several benchmarks show that the performance of OrientDB is the same than Neo4j in the worst cases, while quite better in others.
- OrientDB can be embedded into Tela (if configured with the ORIENTDB_MODE property). This is not possible with (A)GPL licenses.
- Neo4j cypher querying language seemed overwhelming compared with OrientDB SQL.

OrientDB uses a SQL dialect quite similar to the standard SQL syntax. The reason they do this is (OrientDB Manual, 2016):

> *When it comes to query languages, SQL is the mostly widely recognized standard. The majority of developers have experience and are comfortable with SQL. For this reason Orient DB uses SQL as it's query language and adds some extensions to enable graph functionality. There are a few differences between the standard SQL syntax and that supported by OrientDB, but for the most part, it should feel very natural.*

For example, the friends of a user U_1 (people who U_1 follow, and who follow U_1 back) can be retrieved by simply executing:

```
SELECT EXPAND(INTERSECT(IN('Follows'), OUT('Follows'))) FROM User WHERE user_id = ?
```

3.5.1.3 Redis

Tela is shipped with an in-memory cache manager implementation (based on a `ConcurrentMap`). However, it is not completely efficient and scalable, and it should only be used for testing and development purposes. On the other hand, Redis is an open source in-memory data structure store (Redis.io, 2016). Using Redis as a cache engine is one of the most common and recommended uses cases (Sanfilippo, 2011) , offering both scalability and performance. For these reasons, Tela also has a built-in cache manager implementation based on Redis.

3.5.2 Coding Style and Standards Followed

A brief description of the standards and rules that we have used in our application when developing your code and if we have dealt with validating that those standards are met effectively.

3.5.2.1 Code Style: Google Java Style Guide

At the beginning of the implementation, this system followed the JetBrains (developers of IntelliJ IDEA) Java code style. However, in the later phases of the implementation, the style choice shifted to the Google's coding standards, as it is quite more extended and documented (Google, 2016), and it is possible to scan the source code finding problems with the code style using the Checkstyle tool[5], which also has a Maven plugin.

Although some of the problems Checkstyle reported were corrected, most of them have not due to time constraints, and the fact that they are just minor (but time-consuming) fixes, as JetBrains style is quite similar to the one of Google.

3.5.2.2 Code Quality: SonarQube

SonarQube[6] defines itself as "an open platform to manage code quality" which "cover the 7 axes of code quality: architecture & design, comments, coding rules, potential bugs, complexity, unit tests and duplications" (SonarQube, 2016). During and after the implementation, SonarQube and its Sonar Way Quality Profile for Java analysis have been used to analyze and improve the quality of the produced code.

Version used: 6.1

[5] http://checkstyle.sourceforge.net

[6] http://www.sonarqube.org

3.5.3 Tools and Programs Used for Development

3.5.3.1 Version Control System: Git

Git was used as a VCS during the development, in order to back up the code, allow branches with different features and manage different versions of the system, being able to seamlessly revert to previous snapshots of the application.

As a developing tool, Git is not required for building, publishing, installing or executing the system, and the only footprint it has on the project is the '.gitignore' files.

Version used: 2.8.4

3.5.3.2 Package Management & Building Automation: Maven

Maven 3.0.5 was used as package manager and build automation tool. Along with it, the following plugins were used:

3.5.3.2.1 maven-jar-plugin

Pack a jar file. Version: 3.0.0

Configuration:

- Output the jar to the target/jar folder
- Exclude the properties (as it will be copied in the same directory).
- Specify the main class

```
<configuration>
    <outputDirectory>${project.build.directory}/jar</outputDirectory>
    <excludes>
        <exclude>**/tela.properties</exclude>
    </excludes>
    <archive>
        <manifest>
            <mainClass>io.reneses.tela.App</mainClass>
        </manifest>
    </archive>
</configuration>
```

3.5.3.2.2 maven-dependency-plugin

The 'analyze-duplicate', 'tree' and 'analyze' goals were used during the development in order to determine unused, duplicated, conflictive or missing dependencies. Version: 2.10.

3.5.3.2.3 maven-resources-plugin

Copy the tela.properties file into the jar output directory, so that it can be edited directly. Version: 2.6.

Configuration of the `copy-resources` goal:

```
<configuration>
    <outputDirectory>${project.build.directory}/jar</outputDirectory>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/tela.properties</include>
            </includes>
        </resource>
    </resources>
</configuration>
```

3.5.3.2.4 maven-shade-plugin

Build a shaded jar with all the dependencies, so that Tela can be executed regardless of the installed packages and repositories. Version: 2.4.3.

Configuration:

- The jar is no minimized because it removes certain libraries that are actually needed, causing problems with Jetty and OrientDB.
- The dependency reduced pom is not generated.
- Two transformers are used. Note that the ServiceResourceTransformer is required by OrientDB (OrientDB Manual, 2016).

```
<configuration>
    <minimizeJar>false</minimizeJar>
    <createDependencyReducedPom>false</createDependencyReducedPom>
    <outputDirectory>${project.build.directory}/jar</outputDirectory>
    <transformers>
        <transformer implementation=
              "org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
        <transformer implementation=
              "org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
            <mainClass>io.reneses.tela.App</mainClass>
        </transformer>
    </transformers>
</configuration>
```

3.5.3.2.5 maven-compiler-plugin

Used to specify the source and target version of the Java compiler. Version: 2.3.2.

3.5.3.2.6 maven-javadoc-plugin

Render the Javadoc comments into html files. Version: 2.10.4.

It uses the pdfdoclet to generate a PDF document with the Javadoc documentation. Version: 1.0.3

Configuration:

- Generate documentation of public and package entities.
- Output the PDF to the documentation folder.
- Generate a printable PDF with links.

```
<configuration>
    <show>package</show>
    <doclet>com.tarsec.javadoc.pdfdoclet.PDFDoclet</doclet>
    <docletPath>${project.basedir}/docs/pdfdoclet.jar</docletPath>
    <additionalparam>
         -api.title.page true
         -allow.printing true
         -create.links true
         -pdf ${project.basedir}/docs/javadoc.pdf
    </additionalparam>
    <useStandardDocletOptions>false</useStandardDocletOptions>
</configuration>
```

3.5.3.2.7 maven-checkstyle-plugin

Execute Checkstyle obtaining a report of the adherence to the Google Java Style Guide. Version: 2.17

3.5.3.3 IDE: IntelliJ IDEA

The main IDE software used during the development of Tela was IntelliJ IDEA. However, files created by the IDE have been ignored in Git, so that developers are not biased and are able to choose whatever IDE or editor they consider.

- Version: IntelliJ IDEA 2016.2.4
- JRE: 1.8.0_112-release-b343 x86_64
- JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

3.5.3.4 Deployment

3.5.3.4.1 Cloud Deployment: Heroku

Heroku is a PaaS cloud provider with a solution that provides very simple deployments, along with a very convenient free plan. During the developing stage, Heroku was used to deploy each stable release of Tela, in order to use it with the client applications. As it will be covered in the deployment manual, Tela is already configured to make seamlessly Heroku deployments.

3.5.3.4.2 Containerization: Docker

Docker and Docker Compose have been used for:

- Starting Redis and OrientDB containers, so that Tela could be tried and tested interacting with them without having to install or configure anything, in a platform independent way.
- Running Tela locally, so that local client applications could interact with it.
- Deploying Tela in cloud providers accepting containers.

As with Heroku, Tela is shipped with Docker scripts and files that help the process of building and starting containers; which will be reviewed in the deployment manual.

3.5.4 Dependencies

3.5.4.1 Logging: Simple Logging Facade for Java (SLF4J) & Apache Log4j

> *The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time. (Slf4j, 2016)*

For logging purposes, the chosen solution was the combination of SLF4J + LOG4J, due to its simplicity and popularity, which make it become an industry-standard.

With respect to its configuration, it outputs all the logs to the console, while it writes them (excluding the DEBUG level) into the server.log file. Logging configuration can easily be configured modifying the log4j.properties file, located in `src/main/resources`.

Packages and versions:

- org.slf4j.slf4j-api (1.7.21)
- org.slf4j.slf4j-log4j12 (1.7.21)

3.5.4.2 REST API: Jersey & Jetty

For developing the REST API, the chosen solution was a combination of Jersey (implementation of the JAX-RS API) and Jetty, a web server and javax.servlet container.

Regarding its implementation, its configuration has been done programmatically, in order to avoid the old-fashioned XML configuration files. It can be explored at the `core/api/server/JettyTelaServer.java` file.

In order to provide JSON serialization and deserialization, Jackson is also used.

Packages and versions:

- org.eclipse.jetty.jetty-servlet (9.3.8.v20160314)
- org.eclipse.jetty.jetty-servlets (9.3.8.v20160314)
- org.glassfish.jersey.containers.jersey-container-servlet-core (2.19)
- org.glassfish.jersey.media.jersey-media-json-jackson (2.19)
- com.fasterxml.jackson.core.jackson-annotations (2.6.0)
- com.fasterxml.jackson.core.jackson-databind (2.6.3)
- org.apache.httpcomponents.httpcore (4.4.3)

3.5.4.3 Redis: Jedis & Embedded Redis

Jedis is a lightweight Redis java client, which was chosen for Tela for its popularity and easiness of use. For testing the components which interact with Redis using Jedis, the embedded-redis library is used to start a local Redis server.

Packages and versions:

- org.eclipse.jetty.jetty-client (9.3.7.v20160115)
- com.github.kstyrc.embedded-redis (0.6)

3.5.4.4 OrientDB

In order to interact with OrientDB and launch in-memory and local server directly from the application (if configured in this way), Tela uses the official Java libraries of the database. In addition to them, there are also dependencies to other packages, which are required by them.

Packages and versions:

- com.orientechnologies.orientdb-core (2.2.0)
- com.orientechnologies.orientdb-graphdb (2.2.0)
- com.orientechnologies.orientdb-client (2.2.0)
- com.tinkerpop.blueprints.blueprints-core (2.6.0)
- net.java.dev.jna.jna (4.2.1)
- net.java.dev.jna.jna-platform (4.2.1)
- com.googlecode.concurrentlinkedhashmap.concurrentlinkedhashmap-lru (1.4.2)

3.5.4.5 Testing: JUnit & Mockito

For unit testing, the industry standard JUnit was used, in combination with Mockito for mocking and stubbing components, allowing detailed and more independent testing.

Packages and versions:

- junit.junit (4.12)
- org.mockito.mockito-core (1.10.19)

3.5.5 Detailed Description of the Classes

Javadoc has been extensively used during the implementation of the system in order to generate a detailed description of each public and package entity of it. A PDF document has been included in the documentation folder `/tela-server/docs`, as well as an annex to this document.

3.5.6 Implementation Class Diagrams

In this section, we will elaborate detailed class diagrams representing the final implementation.

3.5.6.1 Core

3.5.6.1.1 Action Dispatcher

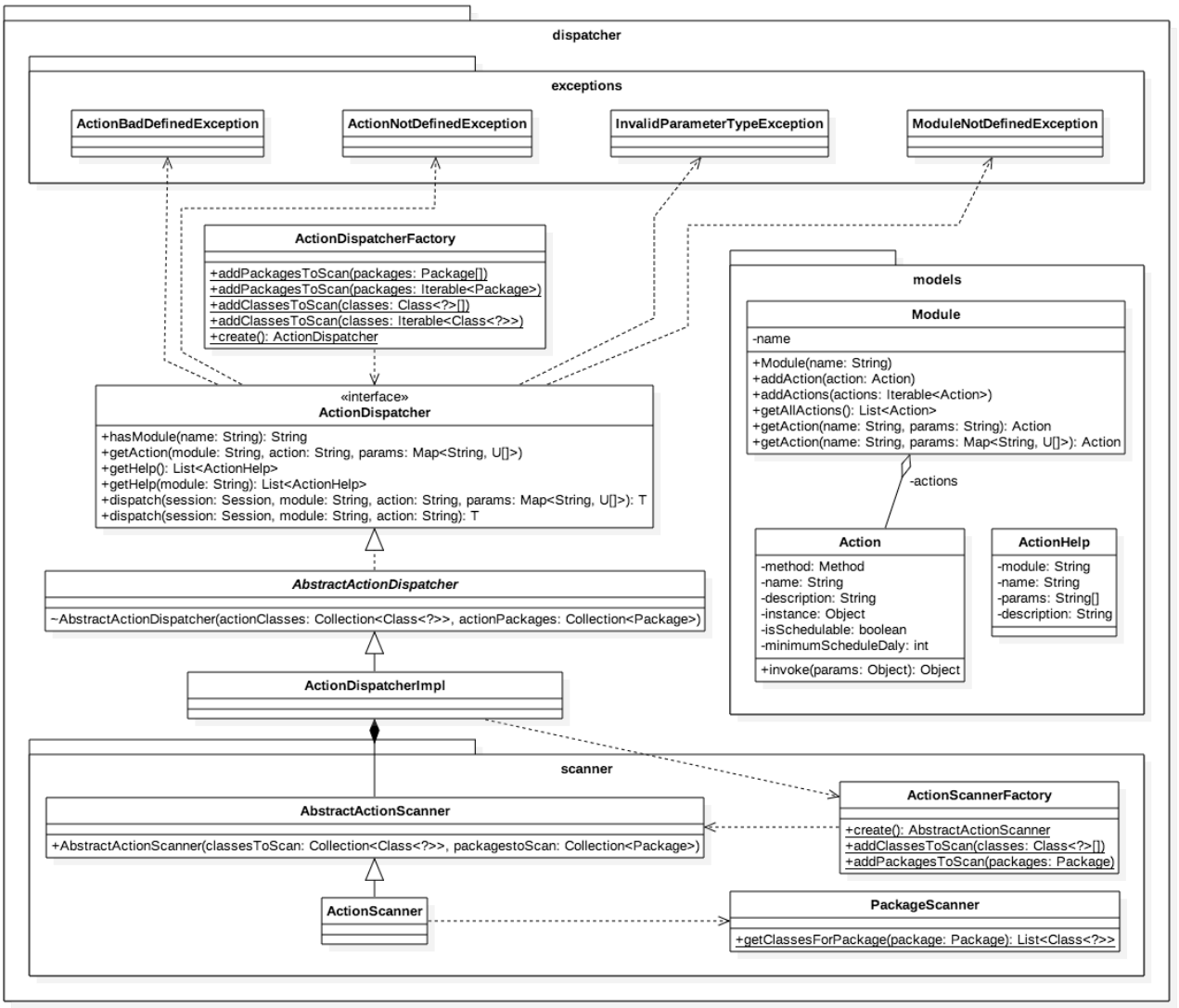


Figure 3.24 Tela Action Dispatcher Implementation Class Diagram

3.5.6.1.2 API

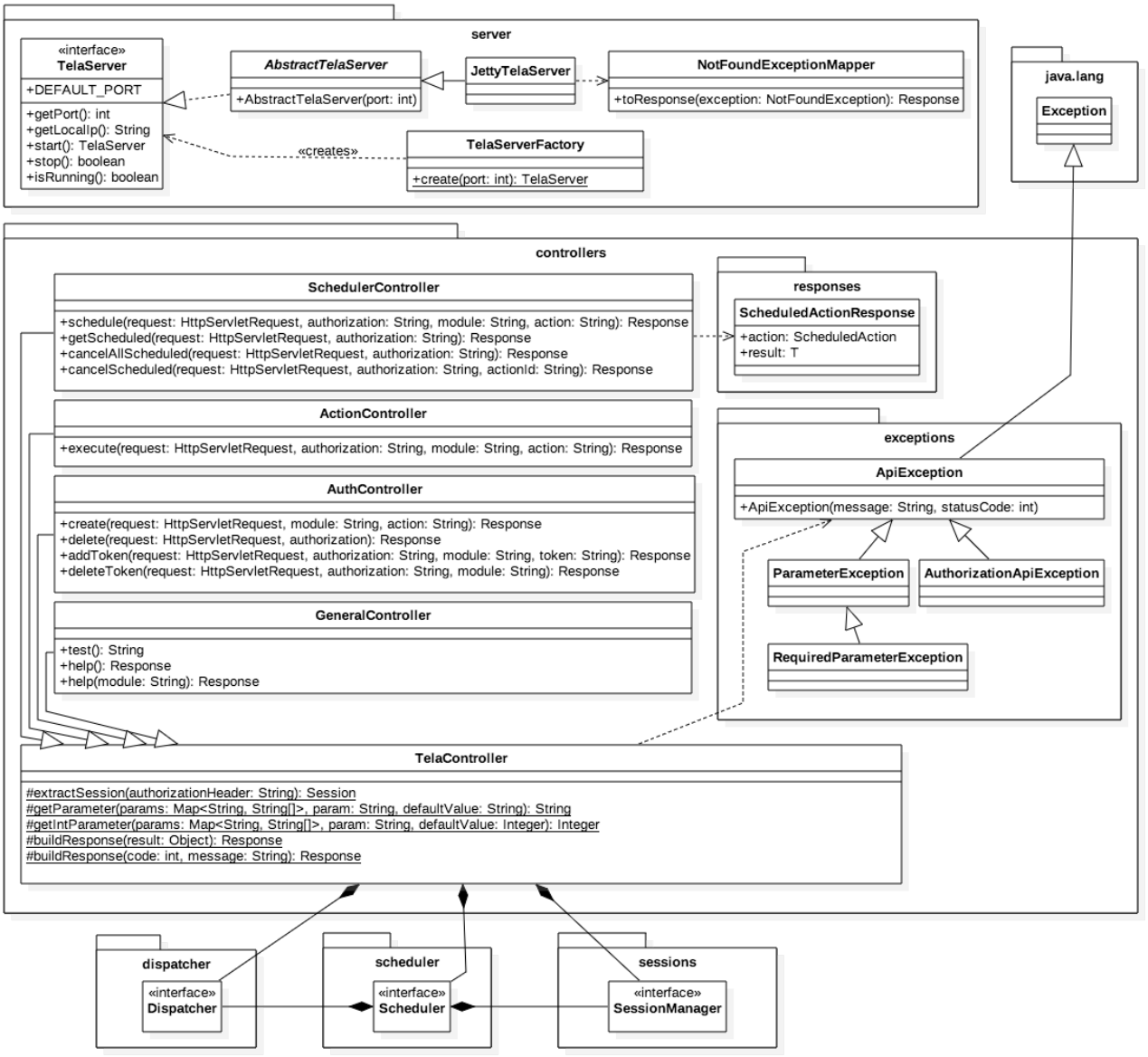


Figure 3.25 Tela API Implementation Class Diagram

3.5.6.1.3 Scheduler

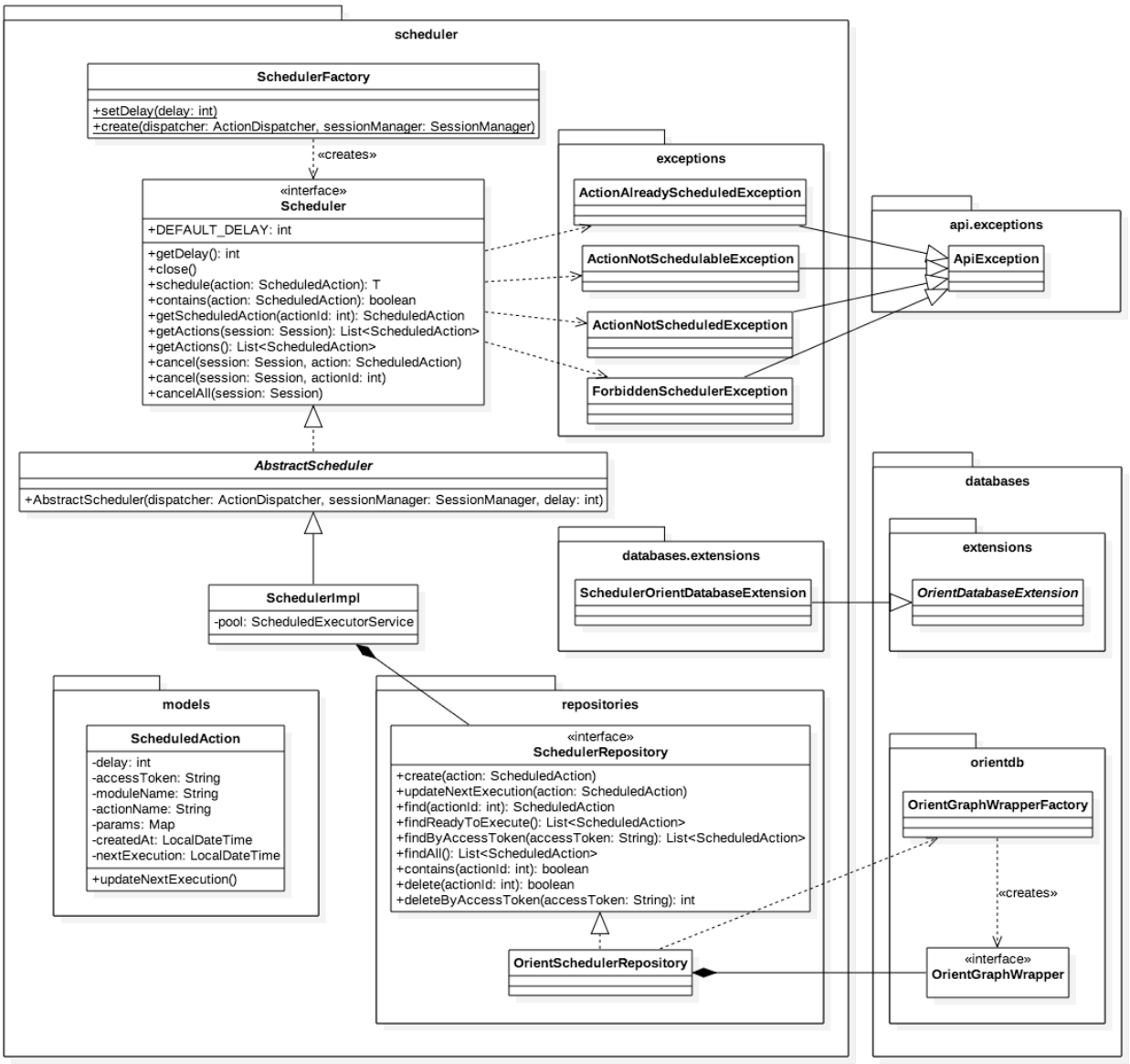


Figure 3.26 Tela Scheduler Implementation Class Diagram

3.5.6.1.4 Sessions

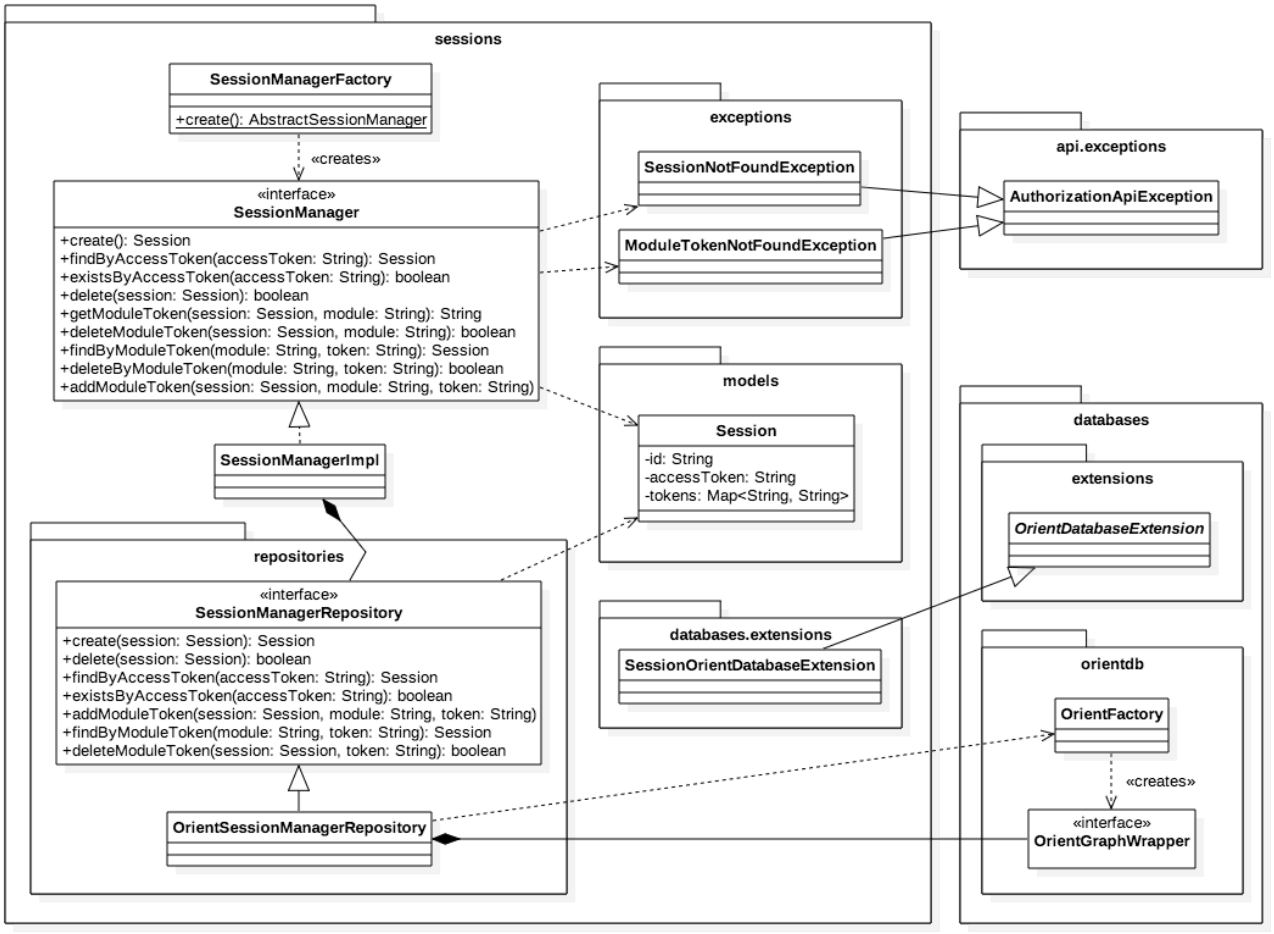


Figure 3.27 Tela Sessions Implementation Class Diagram

3.5.6.1.5 Cache

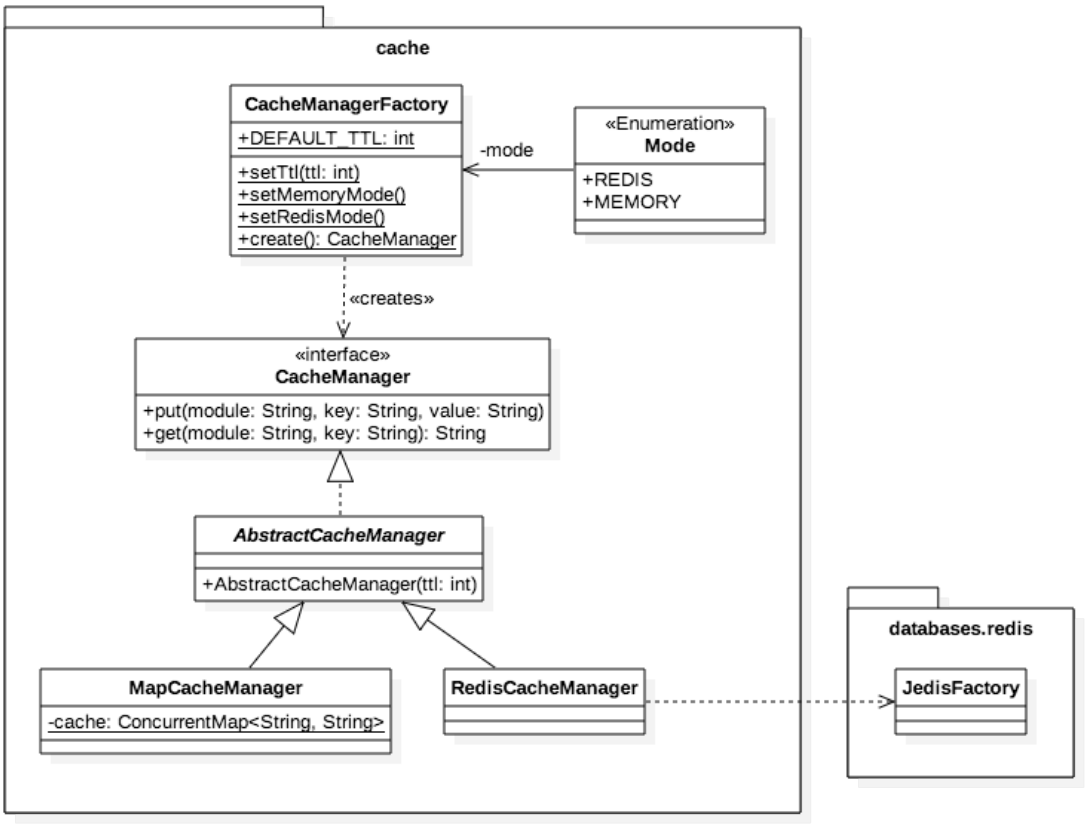


Figure 3.28 Tela Cache Implementation Class Diagram

3.5.6.1.6 History

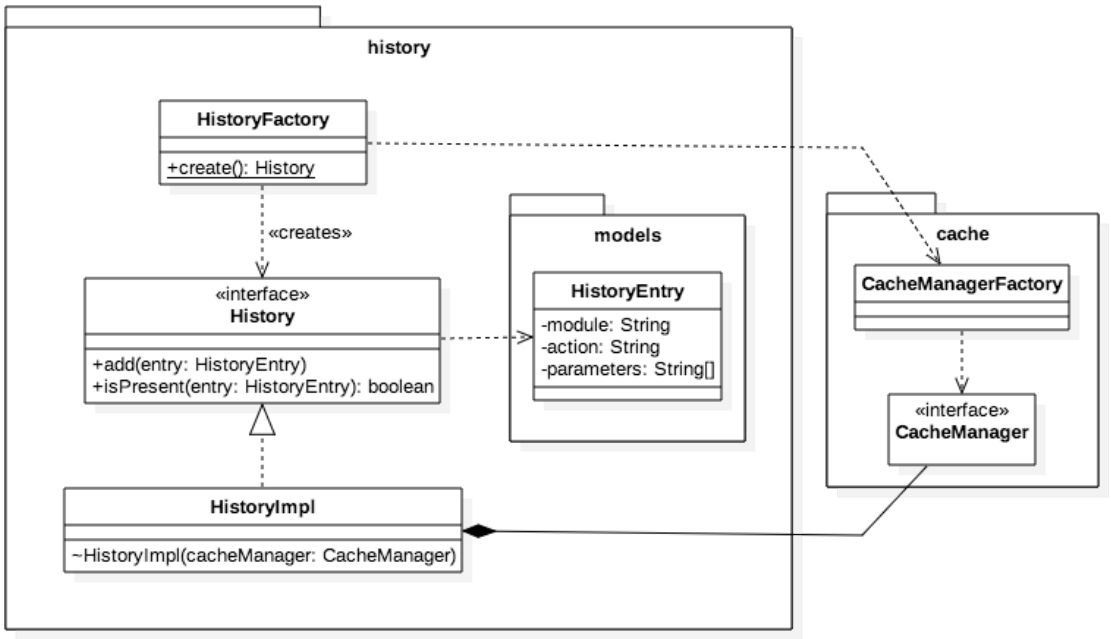


Figure 3.29 Tela History Implementation Class Diagram

3.5.6.1.7 Databases

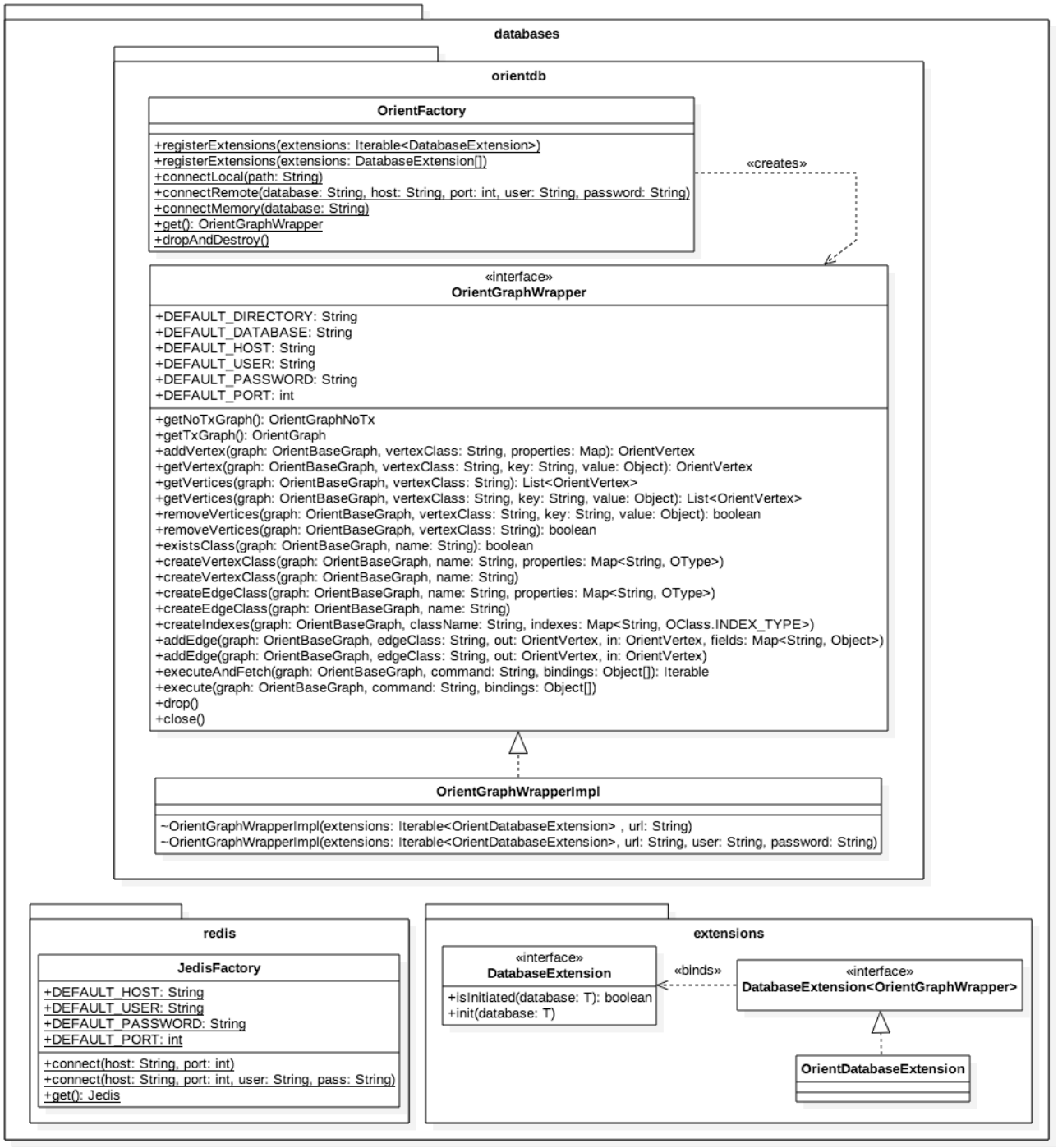


Figure 3.30 Tela Databases Implementation Class Diagram

3.5.6.1.8 Configuration

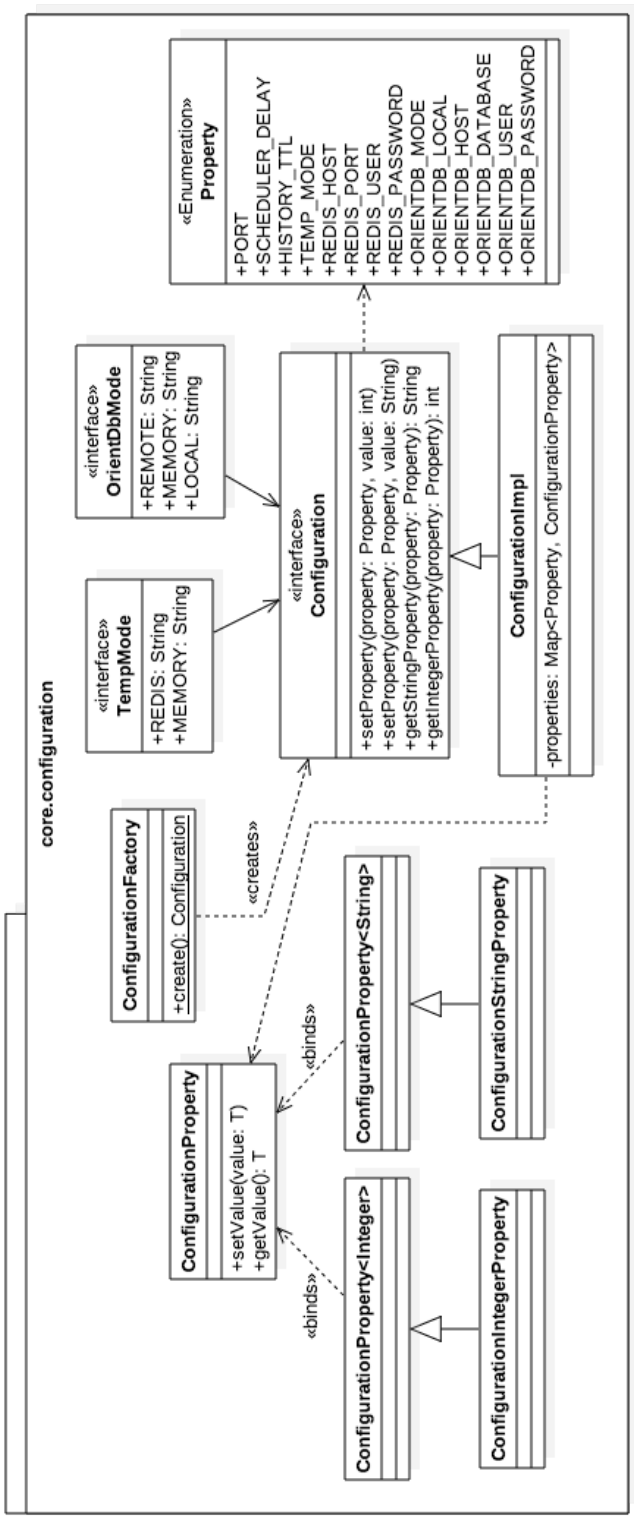


Figure 3.31 Tela Configuration Class Implementation Diagram

3.5.6.2 Modules

3.5.6.2.1 Twitter

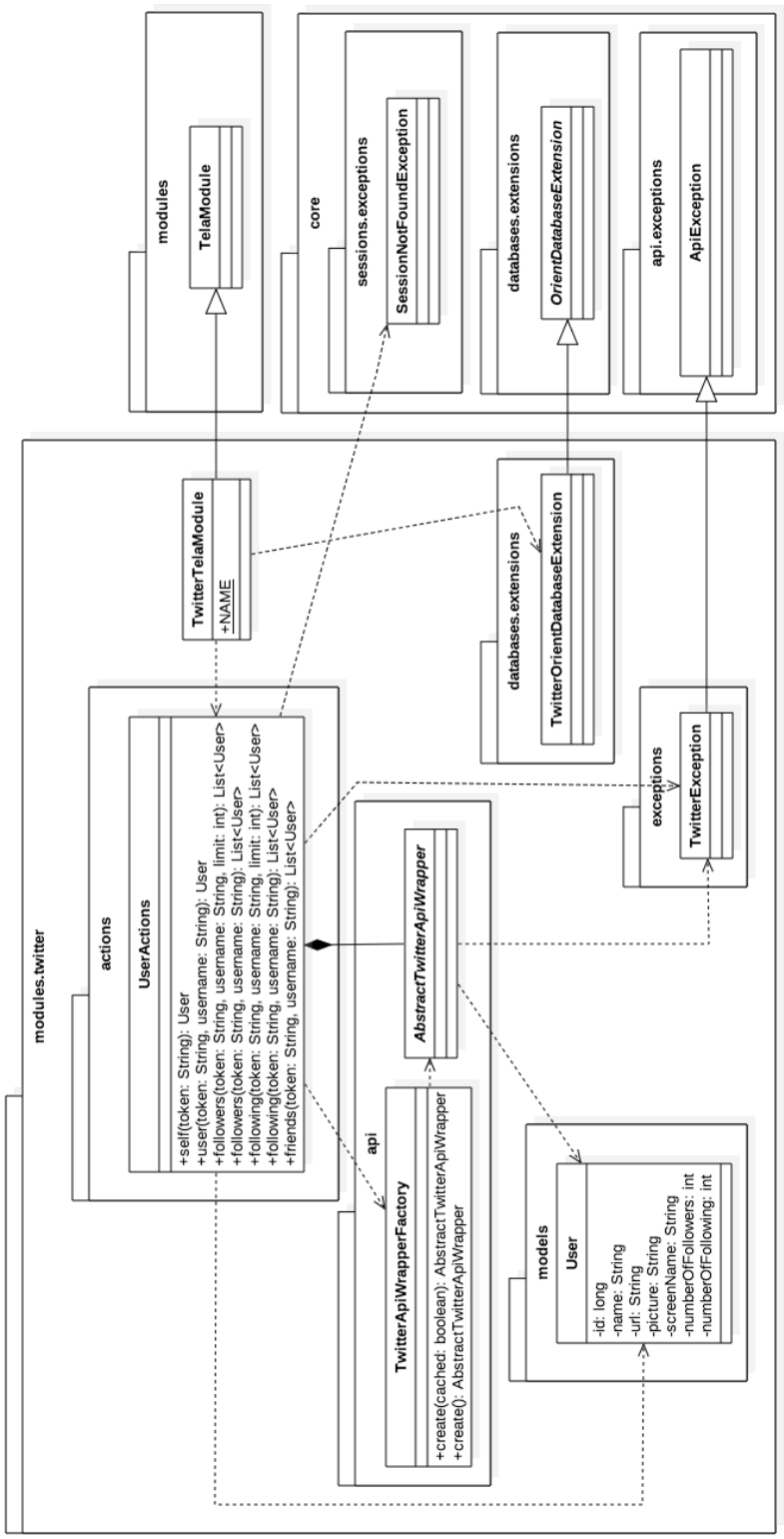


Figure 3.32 Tela Twitter Implementation Class Diagram (I)

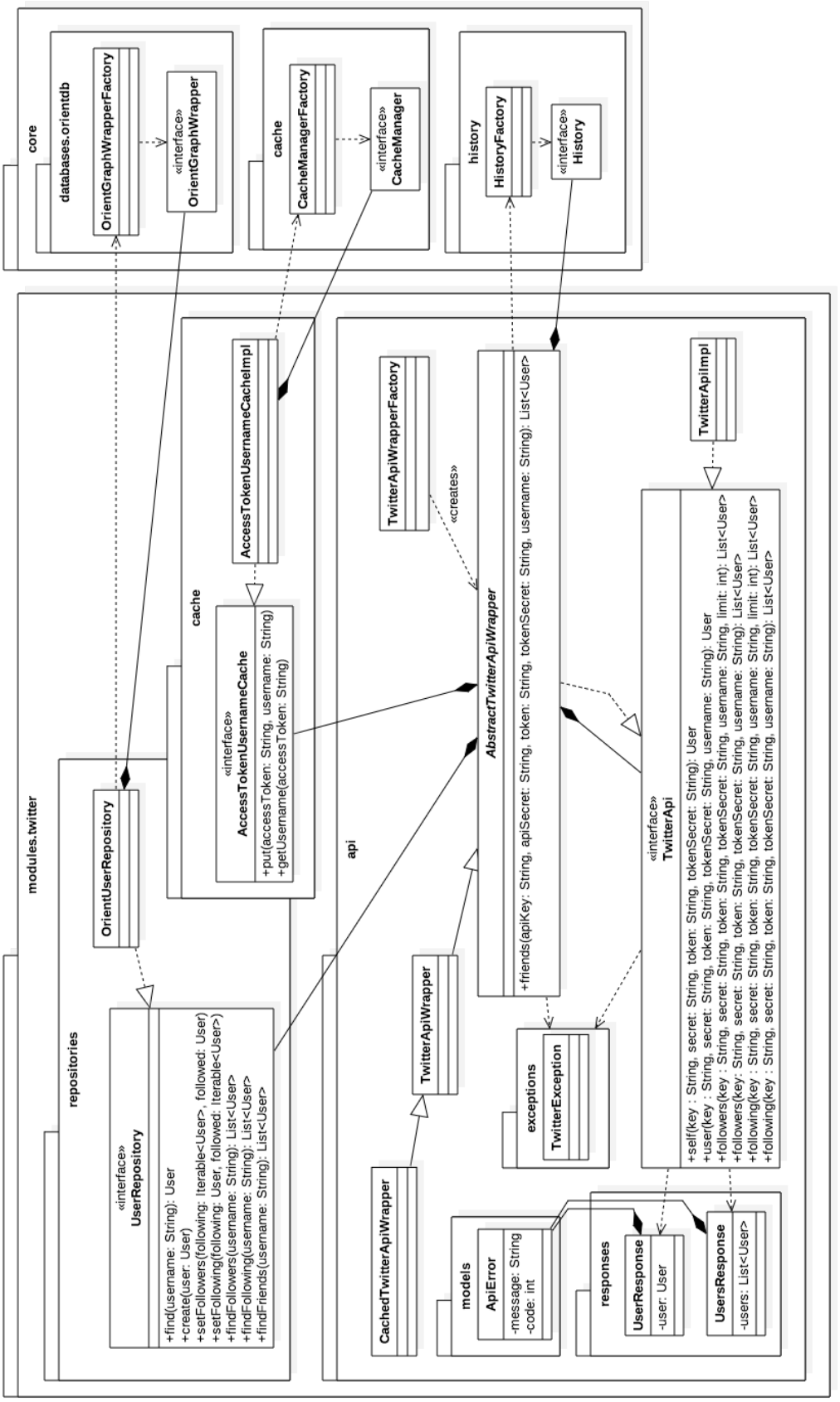


Figure 3.33 Tela Twitter Implementation Class Diagram (II)

3.5.6.2.2 Instagram

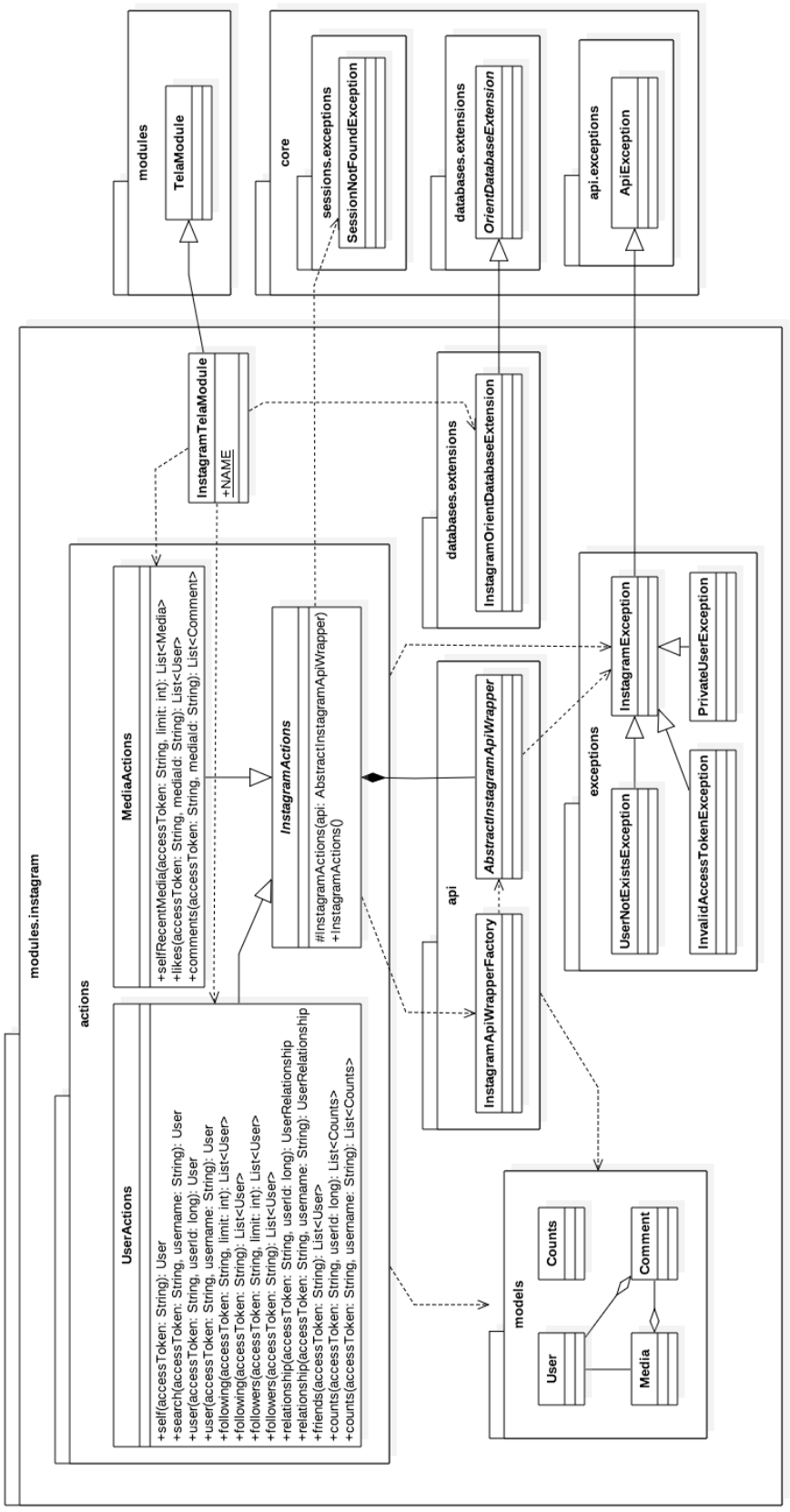


Figure 3.34 Tela Instagram Implementation Class Diagram (I)

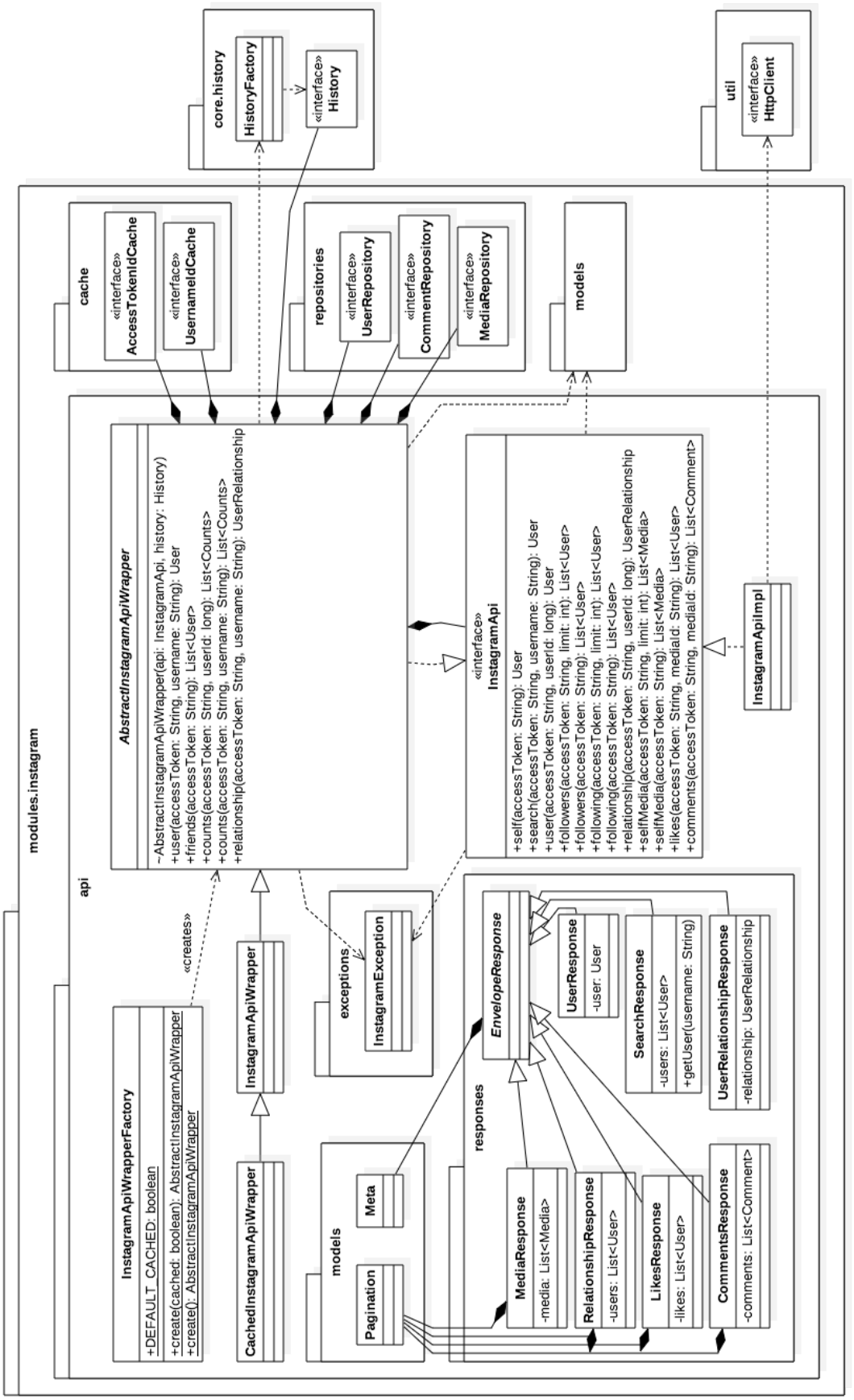


Figure 3.35 Tela Instagram Implementation Class Diagram (II)

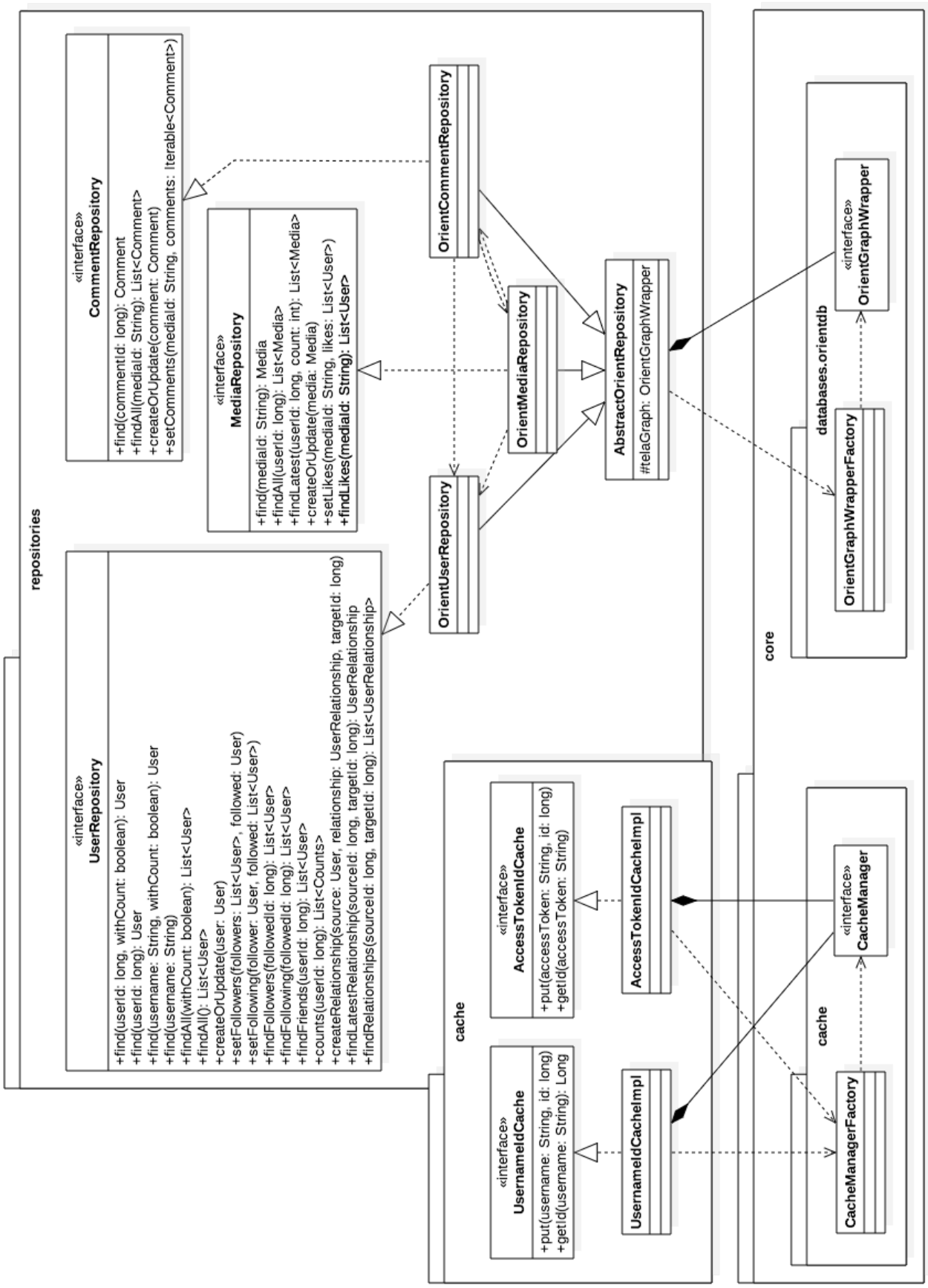


Figure 3.36 Tela Instagram Implementation Class Diagram (III)

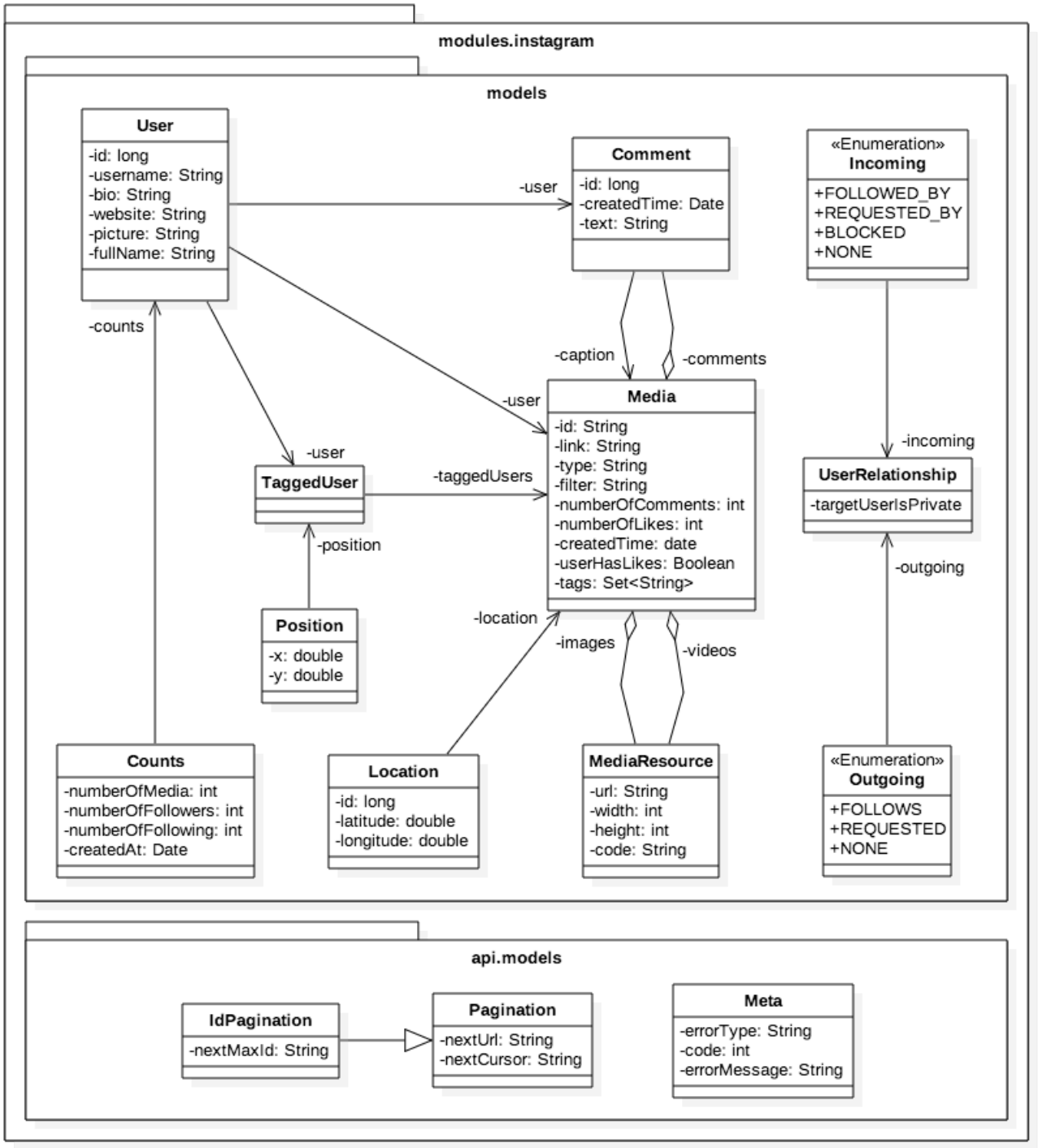


Figure 3.37 Tela Instagram Implementation Class Diagram (IV)

3.5.7 Implementation Details

In this section, we will comment specific implementation details of the components of Tela.

Some additional comments are:

- The design heavily relies on dependency injection, so that components can be easily mocked and tested.
- We will try to avoid the usage of libraries and frameworks as much as possible.

In order to ease the reading of the code, all the Javadocs, and inline comments have been deleted.

3.5.7.1 Action Dispatcher

3.5.7.1.1 Motivation and Overview

The action dispatcher is probably the most important component of the core, and the one whose design and implementation were more challenging.

One of the most important goals of Tela is to be easily extensible so that developers can add their own modules. However, once they have developed the functionality they want… How do they integrate their actions within the API and the scheduler? How can they receive security tokens? Even if a way of registering their own controllers / endpoints was offered, that would imply a lot of boilerplate code.

The way the action dispatcher works is as following:

1. Developers create modules, with the functionality they want to offer.
2. The annotation `@Module` is used to indicate that a class contains methods we want to offer via Tela.
3. Those methods are annotated with `@Action`.
4. When the modules are registered using the assembler, the `ActionDispatcher` scans the package of the module, saving references to all the methods annotated with `@Action` in `@Module` classes.
5. Later on, when an execution request is received, the action dispatcher retrieves the action that matches the name and parameters supplied, regardless of their order.
6. The parameters are cast if needed and supplied to the action.
7. If required and configured, the corresponding module token is supplied to the action.
8. Finally, the action is executed and its results are returned.

3.5.7.1.2 Annotations

These are the `@Module` and `@Action` annotations:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
public @interface Module {
    String value();
}
```

The value of the `@Module` annotation will be its name. For example: `@Module("Instagram")`

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Documented
public @interface Action {
    String name() default "";
    String description() default "";
    String[] parameters();
}
```

- The name is optional. If no name is defined, the name of the method will be used.
- The description is used to generate help about the actions, which the `ActionDispatcher` automatically does.
- The name of the parameters is mandatory. As the order of the parameters that we receive is not taken into account, the name is used to map each input with the actual parameter.

3.5.7.1.3 Action Class Example

The following example forms part of the module development tutorial, which is included in the Manuals Section.

```
@Module("tutorial")
public class TutorialActions {
    @Action(description = "Simple hello", parameters = {})
    public String hello() {
        return "Hello World!";
    }
    @Action(description = "Hello with name", parameters = {"name"})
    public String hello(String name) {
        return "Hello " + name + "!";
    }
    @Action(name = "hello", parameters = {"name", "age"})
    public String helloFriends(String name, int age) {
        return "Hello " + name + " of " + age + " years!";
    }
}
```

3.5.7.1.4 ActionDispatcher

The `ActionDispatcher` is the main component. Its interface is as following:

```
public interface ActionDispatcher {
    boolean hasModule(String name);
    <U> Action getAction(String moduleName, String actionName, Map<String, U[]> params)
              throws ActionNotDefinedException, ModuleNotDefinedException;
    List<ActionHelp> getHelp();
    List<ActionHelp> getHelp(String module) throws ModuleNotDefinedException;
    <T, U> T dispatch(Session session, String moduleName, String actionName,
                      Map<String, U[]> params) throws Exception;
    <T> T dispatch(Session session, String moduleName, String actionName)
              throws Exception;
}
```

Its implementation contains a map of modules, with their names as keys:

```
class ActionDispatcherImpl extends AbstractActionDispatcher {
    private Map<String, Module> modules = new HashMap<>();
    ...
}
```

This map is only generated once, at the moment of construction, as it would be computationally expensive to scan looking for actions at every execution request.

The module class looks as following:

```
public class Module {
    private String name;
    private Map<String, List<Action>> actions = new HashMap<>();
    public Module(String name) {
        this.name = normalizeName(name);
    }
    private String normalizeName(String name) {
        return name.toLowerCase().replace("[\\s/]", "-");
    }
    public List<Action> getActions(String name) { ... }
    public Action getAction(String name, String... params) { ... }
    ...
}
```

While, in turn, the action class looks like:

```
public class Action {
    private Method method;
    private String name, description;
    private Object instance;
    ...
    private LinkedHashMap<String, Class> parameters;
    public Action(Method method) { ... }
    public Object invoke(Object[] params) throws InvocationTargetException {
        try {
            return method.invoke(instance, params);
        } catch (IllegalAccessException e) {
            return null;
        }
    }
}
```

Note that the constructor accepts one method. The order of execution the constructor follows is as following:

1. Checks that the method is annotated as an action.
2. Saves the method.
3. Loads the properties of the action (name, description, and parameter names).
4. Parses the parameters of the method, saving their name and class.
5. Create an instance of the container class, checking that it is possible.
6. If it is schedulable, store it (we will cover this in the following section).

3.5.7.1.5 Specifying the Classes that Contain Actions

In order for the `ActionDispatcher` to map the actions, we have to specify the packages and/or packages that it will have to scan. The `ActionDispatcherFactory`, in addition to creating an implementation of the `ActionDispatcher` interface, offers two static methods to do it:

```
public static void addPackagesToScan(Package... packages) { ... }
public static void addClassesToScan(Class<?>... classes) { ... }
```

However, if we are using the `Assembler` to configure and inject all the components, there is no need to worry about manually registering the packages or classes to scan. When configuring the `ActionDispatcher`, it sets all the module packages to be scanned:

```
private static ActionDispatcher configureActionDispatcher(TelaModule... modules) {
    for (TelaModule module : modules) {
        ActionDispatcherFactory.addPackagesToScan(module.getClass().getPackage());
    }
    return ActionDispatcherFactory.create();
}
```

Note that registering the classes that will be scanned has to be done before creating the instance of the ActionDispatcher; as it has been previously commented, the scanning and mapping of the classes only happen at the construction of the object.

3.5.7.1.6 Action Scanner

The action dispatcher uses the `ActionScanner` to get the map of modules that it will use. The `ActionScanner` only exposes one method, `scan()`, which returns the `Map<String, Module>` that we are expecting.

```
public Map<String, Module> scan() {
    List<Class<?>> classesToScan = collectClassesToScan();
    Map<String, Module> modules = new HashMap<>();
    classesToScan
            .stream()
            .filter(clazz ->
clazz.isAnnotationPresent(io.reneses.tela.core.dispatcher.annotations.Module.class))
            .forEach(clazz -> {
                Module module = processModule(clazz);
                if (modules.containsKey(module.getName()))
                    modules.get(module.getName()).addActions(module.getAllActions());
                else
                    modules.put(module.getName(), module);
            });
    return modules;
}
```

- The `collectClassesToScan()` iterates through all the packages we set to scan, and aggregates all their classes with the classes that we specified, returning a list of all the entities that should be scanned.
- The `processModule()` method receives a class and instantiates a module with it:

```
private Module processModule(Class<?> c) {
    io.reneses.tela.core.dispatcher.annotations.Module annotation =
            c.getAnnotation(io.reneses.tela.core.dispatcher.annotations.Module.class);
    String name = annotation.value();
    Module module = new Module(name);
    for (Method m: c.getMethods()) {
        if (m.isAnnotationPresent(
                            io.reneses.tela.core.dispatcher.annotations.Action.class)) {
            try {
                Action action = new Action(m);
                module.addAction(action);
            } catch (Exception e) { ... }
        }
    }
    return module;
}
```

3.5.7.1.7 ActionDispatcher and ActionController

Another advantage of the `ActionDispatcher` is how simple the `ActionController` becomes. Without it, there should be a method for each action. There would also be several methods for different combinations of parameter types, number of parameters, and so forth.

However, by using the `ActionDispatcher` this is the only method that receives execution requests:

```
@GET
@Path("/{module}/{action}/")
@Produces(MediaType.APPLICATION_JSON)
public Response execute(
        @Context HttpServletRequest request,
        @HeaderParam("Authorization") String authorization,
        @PathParam("module") String module,
        @PathParam("action") String action) {
    Session session = null;
    try {
        session = extractSession(authorization);
        Map<String, String[]> params = request.getParameterMap();
        Object result = dispatcher.dispatch(session, module, action, params);
        return buildResponse(result);
    } catch (ApiException e) {
        return buildErrorResponse(e.getStatusCode(), e.getMessage());
    } catch (Exception e) {
        return buildErrorResponse(500, e.getMessage());
    }
}
```

3.5.7.1.8 ActionDispatcher and Module Tokens

Another convenient advantage that has been briefly commented before is the injection of the module token if required. If the action has a `token` parameter, the action dispatcher will automatically retrieve the token configured for the module of the action and bind it to the parameter.

For example, in the following action only the `username` parameter must be supplied:

```
@Action(name = "search", parameters = {"token", "username"})
public User search(String accessToken, String username) throws InstagramException {
    return api.search(accessToken, username);
}
```

This separates the modules from the auth, so that developers can focus on developing.

3.5.7.2 Scheduler

The scheduler is the component responsible for scheduling actions and executing them periodically.

The scheduler has a delay property, which is the number of seconds it has to wait after its execution, before running again:

```
public interface Scheduler {
    int DEFAULT_DELAY = 300;
    int getDelay();
    ...
}
```

This value is configured in the `SchedulerFactory`, which is also responsible for creating instances that implement the `Scheduler` interface:

```
public class SchedulerFactory {

    private static int delay = Scheduler.DEFAULT_DELAY;
    private SchedulerFactory(){}

    public static void setDelay(int schedulerDelay) {
        if (schedulerDelay < 1)
            throw new IllegalArgumentException(...);
        delay = schedulerDelay;
    }

    public static Scheduler create(ActionDispatcher dispatcher,
                                   SessionManager sessionManager) {
        return new SchedulerImpl(dispatcher, sessionManager, delay);
    }
}
```

Regarding its implementation, the `Scheduler` has an `executor` function which:

- Retrieves the scheduled actions that are ready to be executed from the `SchedulerRepository`.
- Executes each action.
- If an exception is thrown, the scheduling of the responsible action will be canceled.
- Updates their `nextExecution`.
- Uses the `SchedulerRepository` to store this update.

```
private void executor() {
    repository
        .findReadyToExecute()
        .parallelStream()
        .forEach(scheduled -> {
            try {
                executeScheduled(scheduled);
                scheduled.updateNextExecution();
                repository.updateNextExecution(scheduled);
            } catch ( ... ) { ... }
    });
}
```

This `executor()` function is scheduled in a `scheduledThreadPool` in the constructor:

```
class SchedulerImpl extends AbstractScheduler {

    private SchedulerRepository repository;
    private ScheduledExecutorService pool;

    SchedulerImpl(ActionDispatcher dispatcher,SessionManager sessionManager,int delay){
        super(dispatcher, sessionManager, delay);
        this.repository = new OrientSchedulerRepository();
        this.pool = Executors.newScheduledThreadPool(1);
        this.pool.scheduleAtFixedRate(this::executor,
                                      getDelay(), getDelay(),
                                      TimeUnit.SECONDS);
    }
    ...
}
```

The `ScheduledAction` class contains all the information that is needed to execute the action:

```
public class ScheduledAction {
    private int id, delay;
    private String accessToken, moduleName, actionName;
    private Map<String, String[]> params;
    private LocalDateTime createdAt, nextExecution;
    ...
}
```

And they are scheduled by the `Scheduler`, which has a `schedule` method,…

```
<T> T schedule(ScheduledAction action) throws Exception;
```

… methods to retrieve the `ScheduledAction`'s…

```
boolean contains(ScheduledAction action);
ScheduledAction getScheduledAction(int actionId) throws ActionNotScheduledException;
List<ScheduledAction> getScheduledActions(Session session);
List<ScheduledAction> getScheduledActions();
```

… and to cancel them…

```
void cancel(Session session, ScheduledAction action) throws
                              ActionNotScheduledException, ForbiddenSchedulerException;
void cancel(Session session, int actionId) throws
                              ActionNotScheduledException, ForbiddenSchedulerException;
void cancelAll(Session session);
```

3.5.7.3 API

3.5.7.3.1 Dependency Injection

As it has been previously stated, Tela heavily uses dependency injection. In the case of the API, the four controllers (`ActionController`, `AuthController`, `GeneralController` and `SchedulerController`) inherit from the abstract class `TelaController`, which has static instances of the three components they depend on: the `SessionManager`, the `ActionDispatcher`, and the `Scheduler`.

Before any controller is instantiated (i.e. before starting the server) the three components have to be injected by using the static method `configureComponents`:

```
protected static SessionManager sessionManager;
protected static ActionDispatcher dispatcher;
protected static Scheduler scheduler;

public static void configureComponents(SessionManager telaSessionManager,
                                       ActionDispatcher telaDispatcher,
                                       Scheduler telaScheduler) {
    sessionManager = telaSessionManager;
    dispatcher = telaDispatcher;
    scheduler = telaScheduler;

}
```

If these components are injected, they system might not be stable. Therefore, in the moment of instantiation the constructor checks if any of them is null:

```
public TelaController() {

    if (sessionManager == null) {
        throw new IllegalStateException("The Session Manager has not been configured");

    }
    if (dispatcher == null) {
        throw new IllegalStateException("The Dispatcher has not been configured");

    }
    if (scheduler == null) {
        throw new IllegalStateException("The Scheduler has not been configured");

    }

    ...

}
```

3.5.7.3.2 Lightweight Controllers

The controllers are as lightweight as possible, delegating all the business logic to the components of the core. This, in addition to the usage of JAX-RS annotation, makes the code quite clean and simple. For example, this is the code of the general controller:

```
@GET
@Path("/test")
@Produces(MediaType.TEXT_PLAIN)
public String test() {
    return "OK";
}

@GET
@Path("/help")
@Produces(MediaType.APPLICATION_JSON)
public Response help() {
    return buildResponse(dispatcher.getHelp());
}

@GET
@Path("/help/{module}")
@Produces(MediaType.APPLICATION_JSON)
public Response help(@PathParam("module") String module) {
    try {
        return buildResponse(dispatcher.getHelp(module));
    }
    catch (ModuleNotDefinedException e) {
        return buildErrorResponse(404,
                                  "The module '" + module + "' is not configured");
    }
}
```

3.5.7.3.3 Jetty Server

Regarding the server, the design is as following:

- **TelaServer**: Interface declaring the functionality of the server (start, stop and so on).
- **AbstractTelaServer**: Abstract class which implements functionality common to any implementation, as obtaining the IP of the device in the local network.
- **JettyTelaServer**: Embedded Jetty implementation of the server.
- **TelaServerFactory**: factory class with the responsibility of creating and returning an implementation of the `TelaServer` interface.

In order to reduce the number of configuration files, and make the server easily embeddable, the Jetty server is configured programmatically rather than using XML files.

With respect to its initialization, the process is quite standard. The context is created, and resources are registered, being especially important:

```
resourceConfig.register(JacksonFeature.class);
resourceConfig.register(NotFoundExceptionMapper.class);
```

- The `JacksonFeature.class` allows automatic serialization of the server responses.
- On the other hand, the `NotFoundExceptionMapper.class` is a custom error mapper used to return custom 404 errors:

```
@Provider
class NotFoundExceptionMapper implements ExceptionMapper<NotFoundException> {
    public Response toResponse(NotFoundException exception) {
        return Response
                .status(404)
                .entity("Not Found")
                .build();
    }
}
```

Other important thing about the process of creating the server is to register a CORS filter so that requests can be performed even if they come from other:

```
FilterHolder holder = new FilterHolder(CrossOriginFilter.class);
holder.setInitParameter(CrossOriginFilter.ALLOWED_ORIGINS_PARAM, "*");
holder.setInitParameter(CrossOriginFilter.ACCESS_CONTROL_ALLOW_ORIGIN_HEADER, "*");
holder.setInitParameter(CrossOriginFilter.ALLOWED_METHODS_PARAM,
                        "GET,POST,HEAD,OPTIONS,DELETE");
holder.setInitParameter(CrossOriginFilter.ALLOWED_HEADERS_PARAM,
                        "X-Requested-With,Content-Type,Accept,Origin,Authorization");
holder.setName("cross-origin");
```

3.5.7.4 Cache

The cache manager is another important component of the system. The History component is based on a cache manager, and module developers can use it to implement their own caches taking full advantage of the Tela architecture, without having to launch dedicated databases.

The cache manager has the following interface:

```
public interface CacheManager {
    int DEFAULT_TTL = 3600;
    void put(String module, String key, String value);
    String get(String module, String key);
    void clear();
}
```

Tela has two built-in cache implementations: an in-memory one -based on a static `ConcurrentMap`-, and another based on Redis. However, developers can easily implement they preferred approach and configure it globally for the whole framework.

Choosing the desired implementation of the Cache Manager, as well as actually instantiate it, is done via the `CacheManagerFactory` and its static methods `setMemoryMode()`, `setRedisMode()` and `create()`. In addition to that, the factory also has a `setTtl(int ttl)` method so that the Time To Live of the cache entries can be configured regardless of the underlying implementation, and even before actually creating it. If no TTL has been configured, the `CacheManagerFactory` will use the default value specified in `CacheManager.DEFAULT_TTL`.

3.5.7.5 History

The history component is a special type of cache which stores the latest execution calls. Then, modules can decide whether or not send requests to the official APIs of the social networks, or retrieve the latest result from the database.

At the beginning of the implementation phase the history component was completely independent from the cache, even having different configuration properties. However, the history is actually an especial type of cache, and due to the similarity of both components we decided to use a cache manager.

The functionality exposed by the history is very simple:

- Store a history entry (i.e. an action that has been executed), which will expire after certain time.
- Check if an entry exists.

```
public interface History {
    void add(HistoryEntry entry);
    boolean isPresent(HistoryEntry entry);
}
```

Thanks to the cache component, the implementation of the history is quite simple:

```
public class HistoryImpl implements History {

    private static final String MODULE = "-history";
    private CacheManager cacheManager;

    public HistoryImpl(CacheManager cacheManager) {
        this.cacheManager = cacheManager;
    }
    private String getKey(HistoryEntry entry) { ... }
    public void add(HistoryEntry entry) {
        if (entry == null)
            throw new IllegalArgumentException(...);
        cacheManager.put(MODULE, getKey(entry), "");
    }
    public boolean isPresent(HistoryEntry entry) {
        return entry != null && cacheManager.get(MODULE, getKey(entry)) != null;
    }
}
```

- The `CacheManager` is injected via the constructor.
- In order to ensure that there is no colison with other modules, the History uses a dash prefix for its module key (as dashes are not allowed in module names).
- When adding an entry, the entry is used as key, and an empty string is used as the value. This is similar to implementing a set using a map of boolean values.

Finally, following the strategy of other components of the core, the HistoryFactory is the entity in charge of retrieving a CacheManager instance and pass it to the History implementation, at the moment of its instantiation:

```
public static History create() {
    return new HistoryImpl(CacheManagerFactory.create());
}
```

3.5.7.6 Databases

3.5.7.6.1 Jedis Factory

As it has been previously commented, Tela uses the Jedis library in order to interact with Redis. The JedisFactory class wraps a Jedis pool, offering a simple of obtaining Jedis resources. Although the class also offers some common functionality, the two main methods `connect` and `get` are almost identical to their official documentation[7]:

```
private static JedisPool pool;
public static void connect(String host, int port, String user, String password) {
    JedisPoolConfig poolConfig = new JedisPoolConfig();
    poolConfig.setMaxIdle(5);
    poolConfig.setMinIdle(1);
    poolConfig.setMaxTotal(10);
    poolConfig.setTestOnBorrow(true);
    poolConfig.setTestOnReturn(true);
    poolConfig.setTestWhileIdle(true);
    pool = new JedisPool(poolConfig, generateRedisUrl(host, port, user, password));
}
public static Jedis get() {
    if (pool == null) {
        throw new IllegalStateException(...);
    }
    return pool.getResource();
}
```

[7] https://github.com/xetorthio/jedis/wiki

3.5.7.6.2 OrientDB

OrientDB is the main database of the application. Some implementation strategies were considered in order to reduce the dependency of the framework to it, as using implementation-independent solutions as TinkerPop[8]. However, at the end was a trade-off between abstraction and performance, and we chose the latter one.

The final solution was to create an `OrientGraphWrapper` which abstracts some functionality of OrientDB graphs, at the same time than offers helping methods that will make development easier. For example:

- Adding a vertex with properties.
- Retrieving a vertex (o several ones) with a specific type and property.
- Check if a class exists.
- Create a class.
- Execute a query.

As a general rule, the code will try to always use the official OrientDB library, except for those cases where the performance can be significantly increased by the usage of queries. For example, obtaining all the vertices of a certain class (library usage example):

```
public List<OrientVertex> getVertices(OrientBaseGraph graph, String vertexClass) {
    List<OrientVertex> vertices = new ArrayList<>();
    for (Vertex v : graph.getVerticesOfClass(vertexClass))
        vertices.add((OrientVertex) v);
    return vertices;
}
```

And obtaining the vertices of a certain class, with a specific property (query usage example):

```
@Override
public List<OrientVertex> getVertices(OrientBaseGraph graph, String vertexClass,
                                      String key, Object value) {
    String query = String.format("SELECT FROM %s WHERE %s = ?", vertexClass, key);
    List<OrientVertex> vertices = new ArrayList<>();
    for (Object v : executeAndFetch(graph, query, value))
        vertices.add((OrientVertex) v);
    return vertices;
}
```

Conversely, the `OrientGraphFactory` class contains methods to connect to an OrientDB database…

```
public static void connectLocal(String path) { ... }
public static void connectRemote(String database, String host, int port,
                                 String user, String password) { ... }
public static void connectMemory(String database) { ... }
```

[8] https://tinkerpop.apache.org)

… and to obtain a singleton instance of the connection, with the `get()` method. In addition to that, it is also the responsible of registering the database extensions (which will be covered in the following section):

```
public static void registerExtensions(DatabaseExtension... extensions) {
```

Which will be initiated in when the `OrientGraphWrapper` is instantiated:

```
private void init(Iterable<OrientDatabaseExtension> extensions) {
    if (extensions == null || !extensions.iterator().hasNext()) {
        return;
    }
    for (OrientDatabaseExtension extension : extensions) {
        if (!extension.isInitiated(this)) {
            extension.init(this);
        }
    }
}
```

3.5.7.6.3 Database Extensions

The creation of the database schemas required by the modules was one of the trickiest aspects of the design and implementation:

- The modules should be completely independent, without having any knowledge about the rest of modules. The decorator design pattern was studied, but it introduced some degree of interrelation between modules.
- Modules can be added between executions of the server, and the database should detect this changes and initiate the required schemas.
- Once a module has been initiated, its schema should not be generated again.
- As commented before, we will try to avoid other frameworks, so persistence ones (e.g. Hibernate) are out of the question.
- The Migrations approach of the Laravel Framework[9] is attractive, however, its implementation would require a lot of time and it would still need manual triggering.

The solution chosen was to use 'Database Extensions', which implement the following interface where T is an instance of a database:

```
public interface DatabaseExtension<T> {
    boolean isInitiated(T database);
    void init(T database);
}
```

For example, database extensions which use OrientDB inherit from the following class:

[9] https://laravel.com/docs/5.3/migrations

```
public abstract class OrientDatabaseExtension implements
                                            DatabaseExtension<OrientGraphWrapper> { }
```

Then, module classes (inheriting from `TelaModule`) store a collection of extensions, which will be registered in their corresponding database by the assembler. For example:

```
for (TelaModule module : modules) {
    OrientGraphWrapperFactory.registerExtensions(module.getExtensions());
}
```

3.5.7.7 Sessions

Authentication and authorization in Tela is done via 'Sessions':

```
public class Session {

    private String id, accessToken;
    private Map<String, String> tokens;

    ...
}
```

When a user requests to create one session, two unique alphabetical tokens of 32 characters long are created for the session ID and the access token:

```
private static SecureRandom random = new SecureRandom();

private static String generateToken() {
    return new BigInteger(130, random).toString(32);
}

public Session() {
    this(generateToken(), generateToken());
}
```

Note that the creation of the `SecureRandom` is an expensive operation, and therefore is only done once.

The component in charge of handling sessions is the Session Manager, which offers the following methods: `create`, `createWithModuleToken`, `findByAccessToken`, `existsByAccessToken`, `delete`, `getModuleToken`, `deleteModuleToken`, `addModuleToken`.

Regarding its implementation, it stores the session in the main database (OrientDB). Therefore, the component has its own database extension `SessionOrientDatabaseExtension`, child of `OrientDatabaseExtension`. In order to communicate between the business layer and the data storage, a repository layer is used:

```
class SessionManagerImpl implements SessionManager {
    private AbstractSessionManagerRepository repository;
    SessionManagerImpl() {
        super();
        repository = new OrientSessionManagerRepository();
    }
    ...
}
```

The repository supports basic CRUD operations:

```
public interface AbstractSessionManagerRepository {
    void    create(Session session);
    boolean delete(Session session);
    Session findByAccessToken(String accessToken);
    boolean existsByAccessToken(String accessToken);
    boolean addModuleToken(Session session, String module, String token);
    Session findByModuleToken(String module, String token);
    boolean deleteModuleToken(Session session, String token);
}
```

The only implementation of this repository is the `OrientSessionManagerRepository`, which contains a reference to an OrientGraphWrapper instance:

```
public class OrientSessionManagerRepository implements AbstractSessionManagerRepository
{
    private OrientGraphWrapper telaGraph;
    public OrientSessionManagerRepository() {
        telaGraph = OrientGraphWrapperFactory.get();
    }
    ...
}
```

Finally, as with the rest of the components of the core, a Factory is in charge of creating and returning an implementation of the `SessionManager` interface:

```
public class SessionManagerFactory {
    private SessionManagerFactory(){}
    public static SessionManager create() {
        return new SessionManagerImpl();
    }
}
```

3.5.7.8 Twitter Module

As every module, the Twitter one has `TelaModule` child class, which initializes its name and `DatabaseExtension`'s:

```
public class TwitterTelaModule extends TelaModule {
    public static final String NAME = "twitter";
    public TwitterTelaModule() {
        super(NAME);
        setExtensions(Arrays.asList(new TwitterOrientDatabaseExtension()));
    }
}
```

3.5.7.8.1 User Model

The Twitter module has a basic implementation of all the possible actions that would be possible to do using Twitter. Therefore, there is only one model class, `User`, with only has its essential attributes:

```
public class User {
    private long id;
    private String name, screenName, url, picture;
    private int numberOfFollowers, numberOfFollowing;
}
```

3.5.7.8.2 Twitter API

`TwitterApi` implementations are responsible for communicating with the official Twitter API, mapping its responses to `User` models. It offers the following methods: `self()`, `user()`, `followers()` and `following()`.

These methods have at least for parameters which are required in order to authenticate and authorize the request: `apiKey`, `apiSecret`, `token`, and `tokenSecret`. For example:

```
User user(String apiKey, String apiSecret, String token, String tokenSecret,
          String username) throws TwitterException { ... }
```

Its implementation uses an OAuth library in order to execute the requests:

```
public User user(String apiKey, String apiSecret, String token, String tokenSecret,
                 String username) throws TwitterException {
    try {
        String endpoint = "https://api.twitter.com/1.1/users/show.json";
        String stringResponse = oAuthGetRequest(apiKey, apiSecret, token, tokenSecret,
                                    endpoint,
                                    "screen_name", username,
                                    "include_entities", "false");
        UserResponse response = new ObjectMapper().readValue(stringResponse,
                                                             UserResponse.class);
        if (response.hasErrors()) {
            throw new TwitterException(response.getErrors().toString(), 400);
        }
        return response.getUser();
    } catch (...) { ... }
}
```

The `oAuthGetRequest` method has the following implementation:

```
String oAuthGetRequest(String apiKey, String apiSecret, String token,
                       String tokenSecret, String endpoint, String... params)
                       throws IOException {
    OAuth10aService service = new ServiceBuilder()
            .apiKey(apiKey)
            .apiSecret(apiSecret)
            .build(com.github.scribejava.apis.TwitterApi.instance());
    OAuth1AccessToken accessToken = new OAuth1AccessToken(token, tokenSecret);
    OAuthRequest request = new OAuthRequest(Verb.GET, endpoint, service);
    if (params.length > 0) {
        for (int i = 0; i <= params.length / 2; i += 2)
            request.addParameter(params[i], params[i + 1]);
    }
    service.signRequest(accessToken, request);
    return request.send().getBody();
}
```

3.5.7.8.3 Twitter API Wrapper

The `TwitterApi` is wrapped by an `AbstractTwitterApiWrapper`, which has an instance of an implementation of the `TwitterApi` (composition over inheritance) at the same time that implements the `TwitterApi` interface.

The functionality of this wrapper is:

- Declare and implement methods that are not directly offered by the Twitter API, but that are derived from them. As an example, the `friends()` method returns the intersection of `following()` and `followers()`.
- Store all the returned values in the graph database.
- Add the executions to the `History`.

In addition to declaring the extra methods, the constructor of the `AbstractTwitterApiWrapper` is injected with a `TwitterApi` and a `History`, and initializes a `UserRepository` and an `AccessTokenUsernameCache` so that its children will have access to them.

```
public abstract class AbstractTwitterApiWrapper implements TwitterApi {
    TwitterApi api;
    UserRepository repository;
    History history;
    AccessTokenUsernameCache accessTokenUsernameCache;

    AbstractTwitterApiWrapper(TwitterApi api, History history) {
        his.api = api;
        this.history = history;
        this.repository = new OrientUserRepository();
        this.accessTokenUsernameCache = new AccessTokenUsernameCacheImpl();
    }

    public abstract List<User> friends(String apiKey, String apiSecret,
            String token, String tokenSecret, String username) throws TwitterException;
}
```

Its direct implementation`TwitterApiWrapper` stores the results into the database and the executions to the history. For example, the `self()` method has the following implementation:

```
class TwitterApiWrapper extends AbstractTwitterApiWrapper {

    @Override
    public User self(String apiKey, String apiSecret,
                     String token, String tokenSecret) throws TwitterException {

        User user = api.self(apiKey, apiSecret, token, tokenSecret);
        repository.create(user);
        history.add(new HistoryEntry(TwitterTelaModule.NAME, "self", token));
        accessTokenUsernameCache.put(token, user.getScreenName());
        return user;
    }
    ...
}
```

The other implementation, `CachedTwitterApiWrapper`, uses the `History` to check if there has been an execution of the same method within the TTL configured. If it does, it retrieves the result from the UserRepository rather than executing another API request. For example, the `user()` method:

```
class CachedTwitterApiWrapper extends TwitterApiWrapper {

    @Override
    public User user(String apiKey, String apiSecret,
                     String token, String tokenSecret,
                     String username) throws TwitterException {

        if (history.isPresent(
                        new HistoryEntry(TwitterTelaModule.NAME, "user",  username))) {
            return repository.find(username);
        }
        return super.user(apiKey, apiSecret, token, tokenSecret, username);
    }
    ...
}
```

3.5.7.8.4 Actions

The Twitter module only contains one class of actions, `UserActions`. This class contains a `AbstractTwitterApiWrapper`:

```
@Module("twitter")
public class UserActions {

    private AbstractTwitterApiWrapper api;
    public UserActions(AbstractTwitterApiWrapper api) { this.api = api; }
    public UserActions() { this(TwitterApiWrapperFactory.create()); }
    ...
}
```

As it has already been commented, the Twitter API required four auth parameters: the API key, the API secret, the access token and the access token secret. However, the action dispatcher only has the capability of injecting one module token.

The solution implemented was a composed module token with the following structure, inspired by the Basic authorization HTTP method (Franks, 1999) and its usage of the colon as separator:

```
<api_key>:<api_secret>:<access_token>:<access_token_secret>
```

Then, we will need a method to extract this credentials and check that the stored token is valid:

```
private String[] extractCredentials(String token) throws TwitterException {
    String[] twitterCredentials = token.split(":");
    if (twitterCredentials.length != 4) {
        throw new TwitterException("Invalid Twitter module token", 401);
    }
    return twitterCredentials;
}
```

Then, its usage is as simple as:

```
@Action(description = "Get the logged user", parameters = {"token", "username"})
@Schedulable(minimumDelay = 3600)
public User user(String token, String username) throws TwitterException {
    String[] credentials = extractCredentials(token);
    return api.user(credentials[0], credentials[1], credentials[2], credentials[3],
                    username);
}
```

3.5.7.8.5 Cache

The Twitter module has one cache implementation, the `AccessTokenUsernameCache`, used to store the username associated to the access token of the session, so that it is not needed to execute `self()` to obtain it.

```
public interface AccessTokenUsernameCache {
    void put(String accessToken, String username);
    String getUsername(String accessToken);
    void clear();
}
```

As expected, its implementation uses a `CacheManager` from the `CacheManagerFactory`:

```
public class AccessTokenUsernameCacheImpl implements AccessTokenUsernameCache {
    private static final String PREFIX = "token-username";
    private CacheManager cacheManager = CacheManagerFactory.create();
    ...
}
```

3.5.7.9 Instagram Module

The Instagram follows the same structure and design than the Twitter one, so we will just overview its components and implementation decisions.

As with the Twitter module, the first component we see in the package is the `InstagramTelaModule`:

```
public class InstagramTelaModule extends TelaModule {

    public static final String NAME = "instagram";

    public InstagramTelaModule() {
        super(NAME);
        setExtensions(Arrays.asList(new InstagramOrientDatabaseExtension()));
    }
}
```

3.5.7.9.1 Models

The Instagram module offers all the functionality of the Instagram API, and therefore contains many more models, with several properties which can be seen in the class diagrams: `Comment`, `Counts`, `Location`, `Media`, `MediaResource`, `Position`, `TaggedUser`, `User` and `UserRelationship`.

3.5.7.9.2 Instagram API

As the `TwitterApi`, the `InstagramApi` implementations are responsible of communicating with the official Instagram API:

```
interface InstagramApi {
    User self(String accessToken)                        throws InstagramException;
    User search(String accessToken, String username)     throws InstagramException;
    User user(String accessToken, long userId)           throws InstagramException;
    List<User> followers(String accessToken, int limit)  throws InstagramException;
    List<User> followers(String accessToken)             throws InstagramException;
    List<User> following(String accessToken, int limit)  throws InstagramException;
    List<User> following(String accessToken)             throws InstagramException;
    UserRelationship relationship(String accessToken, long userId)
                                                         throws InstagramException;
    List<Media> selfMedia(String accessToken, int limit) throws InstagramException;
    List<Media> selfMedia(String accessToken)            throws InstagramException;
    List<User> likes(String accessToken, String mediaId) throws InstagramException;
    List<Comment> comments(String accessToken, String mediaId)
                                                         throws InstagramException;
}
```

The first parameter of all these methods is an `accessToken`, which is the Bearer token used to authorize the requests. Therefore, any HTTP client can be used without any special complexity, as was the case of the Twitter module and the OAuth requests.

3.5.7.9.3 Instagrapm API Wrapper

As in the Twitter module, The `InstagramApi` is wrapped by an `AbstractInstagramApiWrapper`, which has an instance of an implementation of the `InstagramApi` (composition over inheritance) at the same time that implements the `InstagramApi` interface.

The `AbstractInstagramApiWrapper` declares these extra methods:

```
public abstract User user(String accessToken, String username)
                                                    throws InstagramException;
public abstract List<User> friends(String accessToken) throws InstagramException;
public abstract List<Counts> counts(String accessToken, long userId)
                                                    throws InstagramException
public abstract List<Counts> counts(String accessToken, String username)
                                                    throwsInstagram Exception;
public abstract UserRelationship relationship(String accessToken, String username)
                                                    throws InstagramException;
```

And its constructor is:

```
public abstract class AbstractInstagramApiWrapper implements InstagramApi {

    protected InstagramApi api;
    protected History history;
    protected UserRepository userRepository;
    protected MediaRepository mediaRepository;
    protected CommentRepository commentRepository;
    protected AccessTokenIdCache accessTokenIdCache;
    protected UsernameIdCache usernameIdCache;

    AbstractInstagramApiWrapper(InstagramApi api, History history) {
        this.api = api;
        this.history = history;
        this.userRepository = new OrientUserRepository();
        this.mediaRepository = new OrientMediaRepository();
        this.commentRepository = new OrientCommentRepository();
        this.accessTokenIdCache = new AccessTokenIdCacheImpl();
        this.usernameIdCache = new UsernameIdCacheImpl();
    }
    ...
}
```

Its direct implementation `InstagramApiWrapper` adds the results to the database and the executions to the history. On the other hand, the `CachedInstagramApiWrapper` –which extends the `InstagramApiWrapper`- uses the `History` to check if there has been an execution of the same method within the TTL configured. If it does, it retrieves the result from the corresponding repository rather than executing another API request.

3.5.7.9.4 Actions

The Instagram module contains two classes with actions: `UserActions` (containing all those actions related to the users) and `MediaActions` (containing the ones related with media posts). Both of them inherit from the abstract class `InstagramActions` which initializes (or receives by injection) an instance of `AbstractInstagramApiWrapper`:

```
abstract class InstagramActions {

    protected AbstractInstagramApiWrapper api;

    InstagramActions(AbstractInstagramApiWrapper api) {
        this.api = api;
    }

    public InstagramActions() {
        this(InstagramApiWrapperFactory.create());
    }
}
```

Unlike the Twitter module, the module token injected by the `ActionDispatcher` is the only access token required to execute requests to the API. Therefore, action methods are as simple as:

```
@Action(name = "search",
        description = "Get the basic information about a user",
        parameters = {"token", "username"})
 public User search(String accessToken, String username) throws InstagramException {
    return api.search(accessToken, username);
}
```

3.5.7.9.5 Cache

The Instagram module has two cache implementations:

On one hand, the `UsernameIdCache` caches the ID associated to each username:

```
public interface UsernameIdCache {
    void put(String username, long userId);
    Long getId(String username);
}
```

This is important due to the fact that Instagram, unlike Twitter, does not accept usernames as a query, only IDs. Without this cache we would have to execute `search()` all the time. As it is expected, it uses a `CacheManager`:

```
public class UsernameIdCacheImpl implements UsernameIdCache {
    private static final String PREFIX = "username-id";
    private CacheManager cache = CacheManagerFactory.create();
    ...
}
```

On the other hand, the `AccessTokenIdCache` caches the ID corresponding to each access token (i.e. the equivalent to the `AccessTokenUsernameCache` from the Twitter module):

```
public interface AccessTokenIdCache {
    void put(String accessToken, long userId);
    Long getId(String accessToken);
}

public class AccessTokenIdCacheImpl implements AccessTokenIdCache {
    private static final String PREFIX = "token-id";
    private CacheManager cache = CacheManagerFactory.create();
    ...
}
```

3.5.8 Problems Found

3.5.8.1 OrientDB Remote Engine not working from a shaded JAR file

While the application perfectly working if a run it from the IDE, it was not possible to execute it and make a remote connection if the code was packaged as a shaded JAR.

At the beginning, I thought that maybe there was a configuration problem or collision among packages. I tried to simplify the error case as much as possible, but it continued happening:

```
import com.tinkerpop.blueprints.impls.orient.OrientGraphFactory;

public class App {

public static void main(String[] args) throws Exception {
    OrientGraphFactory factory = new OrientGraphFactory("REMOTE:localhost/test",
                                                        "root", "root", true);
    factory.getNoTx();
  }
}
```

After spending one day debugging the decompiled source code of OrientDB instruction by instruction, I decided to open an issue in Github[10]. A member of the developing team identified the error as a documentation issue:

> *OrientDB uses Java services (META-INF/services) for creating components like Engines. You should add this in your pom.xml in transformers sections in order to append services in case of collision when creating a single shaded jar.*
>
> *I will mark this issue as documentation, i will add a section on how to build a shaded jar with OrientDB*

```
<transformer implementation=
                "org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
```

3.5.8.2 Jetty Blocking Cross Origin Requests

While the system was correctly working if both the client and the server were running in the same machine, once I deployed Tela online for the first time, they could not interact any longer.

It turned to be that by default Jetty was blocking requests from devices outside the same domain. In order to solve it, I had to register a `CrossOriginFilter`, which was not an easy task neither, as all the available documentation only specifies how to do it via XML files.

[10] https://github.com/orientechnologies/orientdb/issues/6299

Finally, after observing how the configuration of other files was done programmatically, along with their XML equivalent, and analysing the decompiled source code, I was able to figure out how to overcome these problems:

```
// CORS API Filter
FilterHolder holder = new FilterHolder(CrossOriginFilter.class);
holder.setInitParameter(CrossOriginFilter.ALLOWED_ORIGINS_PARAM, "*");
holder.setInitParameter(CrossOriginFilter.ACCESS_CONTROL_ALLOW_ORIGIN_HEADER, "*");
holder.setInitParameter(CrossOriginFilter.ALLOWED_METHODS_PARAM,
"GET,POST,HEAD,OPTIONS,DELETE");
holder.setInitParameter(CrossOriginFilter.ALLOWED_HEADERS_PARAM, "X-Requested-
With,Content-Type,Accept,Origin,Authorization");
holder.setName("cross-origin");
...
context.addFilter(holder, "*", EnumSet.of(DispatcherType.INCLUDE,
                                          DispatcherType.REQUEST));
```

3.6 Test Results

3.6.1 Automated Testing

At the beginning of each development iteration, right after setting the goals and choosing the appropriate design, unit tests were developed following a TDD methodology. Once the solution was implemented, if necessary, more automated tests were developed, including integration ones when possible.

Finally, before considering the iteration completed, the tests of the iteration were fired. In addition to that, all the previously tests were also executed as a way of regression testing.

If any test failed, the origin of the bug was delimited, found and fixed, and all the tests (including the regression ones) were executed again. This process was repeated until all the tests passed.

Once the system was considered finished, all the tests were executed, obtaining the following result.

```
Results :

Tests run: 308, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 04:00 min
[INFO] Finished at: 2016-11-04T17:55:15+01:00
[INFO] Final Memory: 13M/223M
[INFO] ------------------------------------------------------------------------
```

Figure 3.38 Tela Automated Tests Final Results

3.7 System Manuals

3.7.1 Tela Deployment: Building, Configuring, Packaging and Running Tela

This manual can be found (in Markdown format) at `/tela-server/docs/DEPLOYMENT.md`.

3.7.1.1 Troubleshooting

If you receive an `Exception in thread "main" java.lang.OutOfMemoryError: Direct buffer memory error`, increase the memory of the JVM or configure OrientDB as remote.

3.7.1.2 Building your own Tela Server

The Tela Framework contains a server already configured with two modules (Instagram & Twitter), which can be started by running the `main` method at `io.reneses.tela.App`.

In case we would like to register different modules, or integrate Tela within an existing app, the process is quite straightforward. First, we have to import the `TelaServer` and the `Assembler`:

```
import io.reneses.tela.Assembler;
import io.reneses.tela.core.api.server.TelaServer;
```

The `Assembler` is the component responsible of reading the configuration, configuring and injecting the rest of components and instantiating the server. Although its usage is not required as this process can be done manually, it is quite advisable to use it.

Then, we import the modules we will register:

```
import io.reneses.tela.modules.instagram.InstagramTelaModule;
import io.reneses.tela.modules.twitter.TwitterTelaModule;
```

Finally, we build the server, passing the modules as parameters to the `build` method...

```
TelaServer tela = Assembler.build(
    new InstagramTelaModule(),
    new TwitterTelaModule()
 );
```

... and call `start()` on the retrieved instance:

```
tela.start();
```

3.7.1.3 Configuring Tela

Tela can be configured in the following ways (from higher to lower priority):

1. Programmatically.
2. Setting the properties as system environmental variables.
3. Setting the properties in the `tela.properties` file.
4. If it is not present neither, it uses the default option.
5. If any configuration is present, Tela will use the default value.

3.7.1.3.1 Programmatically Configuration

The `Assembler.build()` method also admits a `Configuration` instance, which can be used to set properties programmatically. For example:

```
import io.reneses.tela.Assembler;
import io.reneses.tela.core.api.server.TelaServer;
import io.reneses.tela.core.configuration.Configuration;
import io.reneses.tela.core.configuration.ConfigurationFactory;
import io.reneses.tela.modules.twitter.TwitterTelaModule;

public class App {

    public static void main(String[] args) {

        Configuration config = ConfigurationFactory.create();
        config.setProperty(Configuration.Property.CACHE_TTL, 1000);
        config.setProperty(Configuration.Property.PORT, 80);

        TelaServer tela = Assembler.build(config, new TwitterTelaModule());
        tela.start();
    }
}
```

3.7.1.3.2 Using Environmental Variables

The process of setting environmental variables depends on the Operating System. A tutorial for OS X, Windows and Mac can be found at this link[11] (Schrodinger.com).

3.7.1.3.3 Using File Configuration

The `tela.properties` file is a standard Properties java text file[12], which should be placed within the same folder than the Jar distribution. Its syntax is: `[KEY]=[VALUE]`, for example:

[11] https://www.schrodinger.com/kb/1842

[12] https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html

```
# General
port=8080
scheduler.delay=300

# Cache
cache.mode=memory
cache.ttl=3600
```

3.7.1.3.4 Configuration Options

The available options, as well as their environmental properties, keys within the properties file, and default values can be found in the following table:

Option	Default value	Env property	File property	Description
Tela port	80	PORT	port	Port Tela will be binded to
Scheduler delay	300	SCHEDULER_DELAY	scheduler.delay	Delay for the scheduler
Cache TTL	3600	CACHE_TTL	cache.ttl	TTL of the cache &history entries
Cache mode	memory	CACHE_MODE	cache.mode	Cache mode (memory or redis)
Redis host	localhost	REDIS_HOST	redis.host	Redis host
Redis port	6379	REDIS_PORT	redis.port	Redis port
Redis user	null	REDIS_USER	redis.user	Redis user
Redis password	null	REDIS_PASSWORD	redis.password	Redis password
OrientDB mode	local	ORIENTDB_MODE	orientdb.mode	OrientDB storage[13]
OrientDB local	./data	ORIENTDB_LOCAL	orientdb.local	DB folder (local mode)

[13] http://orientdb.com/docs/master/Storages.html

Option	Default value	Env property	File property	Description
OrientDB host	localhost	ORIENTDB_HOST	orientdb.host	DB host (remote mode)
OrientDB port	2424	ORIENTDB_PORT	orientdb.port	DB port (remote mode)
OrientDB database	Tela	ORIENTDB_DATABASE	orientdb.database	DB database (memory/remote mode)
OrientDB user	root	ORIENTDB_USER	orientdb.user	OrientDB user
OrientDB password	root	ORIENTDB_PASSWORD	orientdb.password	OrientDB user

3.7.1.4 Compiling, Packaging & Installing Tela

Tela uses Maven[14] as build automation tool. Therefore, it can be compiled and packaged executing the `package` goal (or `install` if we also want to install Tela at our local Maven repository). If we want a faster build, tests can be skipped with the `-DskipTests=true` option:

```
# Execute tests and package Tela
mvn clean package

# Package Tela skipping tests
mvn clean package -DskipTests=true

# Package & install Tela
mvn clean install -DskipTests=true
```

3.7.1.5 Running Tela

At this point, you should already have a `tela.jar` file (and, optionally a `tela.properties` file). By default they are generated at the folder `target/jar`.

3.7.1.5.1 Docker

The easiest way to run Tela is using Docker and Docker Compose[15]. Both the `Dockerfile` and `docker-compose.yml` are included in the project, so running it is as easy as:

[14] https://maven.apache.org/

[15] https://www.docker.com

```
cmd /path/to/Tela
docker-compose up
```

The provided `Dockerfile` is already configured to use Redis and a remote OrientDB connection:

```
# Application port
ENV PORT 80
EXPOSE 80

# Configure redis
ENV CACHE_MODE redis
ENV REDIS_HOST redis
ENV REDIS_PORT 6379

# Configure OrientDB
ENV ORIENTDB_MODE remote
ENV ORIENTDB_HOST orientdb
ENV ORIENTDB_PORT 2480
ENV ORIENTDB_USER root
ENV ORIENTDB_PASSWORD tela
EXPOSE 2480
```

If other configuration is needed, the environmental variables can be changed by using the Dockerfile syntax[16].

3.7.1.5.2 Heroku

A Heroku's `Procfile` is provided with Tela, and the framework uses the `PORT` environmental variable as Heroku requires[17]. Therefore, the deployment on Heroku is straightforward. For example, if Git deployment is configured[18], it can be done executing:

```
git push heroku
```

For Heroku deployment, the easiest way to configure Tela is using environmental variables. A `.env` file is provided too, so Heroku Local can easily be configured by modificating its contents, and executed locally with:

```
heroku local
```

[16] https://docs.docker.com/engine/reference/builder/

[17] https://devcenter.heroku.com/articles/runtime-principles#web-servers

[18] https://devcenter.heroku.com/articles/git

3.7.1.5.3 Console execution

If neither Docker or Heroku are installed, or you want to execute Tela directly, it can be done by executing:

```
java [JVM configuration] -jar path/to/tela.jar
```

3.7.1.6 Production Deployment

Tela can be configured to run all its components on memory:

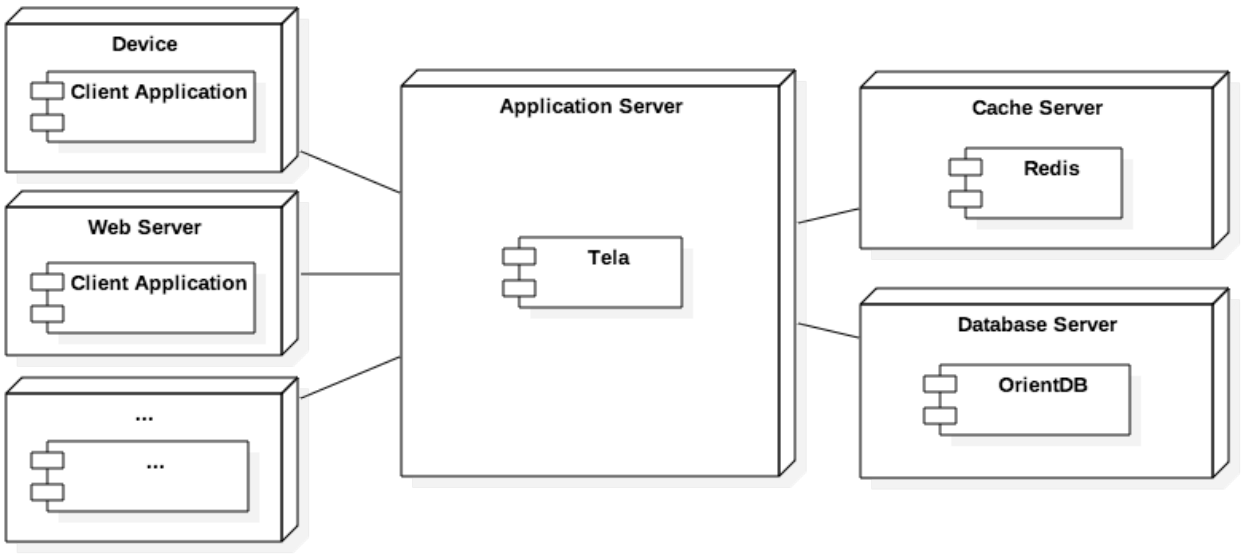
```
CACHE_MODE=memory
ORIENTDB_MODE=memory
```

This is the simplest and most straightforward configuration. However, it implies performance and scalability drawbacks.

For production environments, the suggested deployment architecture is:

```
CACHE_MODE=redis
# ... Redis configuration
ORIENTDB_MODE=remote
# ... OrientDB configuration
```

As it is shown in the following diagram:



3.7.1.6.1 Redis

> *"Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker" (redis.io[19])*

There are plenty[20] of tutorials[21] for installing and configuring Redis. Running Redis with Docker is extremely straightforward, and can be done executing the script `scripts/docker-redis.sh`:

```
#!/usr/bin/env bash
docker run -p 6379:6379 redis
```

3.7.1.6.2 OrientDB

OrientDB[22] is the main database of Tela. A OrientDB Docker container can be started with the script `scripts/docker-orientdb.sh`:

```
#!/usr/bin/env bash
docker run -p 2424:2424 -p 2480:2480 -e ORIENTDB_ROOT_PASSWORD=root orientdb
```

If we are seeking maximum performance, a OrientDB server is advised. Its deployment can be easily with any cloud provider. For example, if we are using Amazon EC2, the script `scripts/install-orientdb-server.sh` will prepare everything for us (it uses `yum`, the commands for `apt-get` should be similar):

```
#!/bin/bash

# This prepared to run on an Amazon Linux distribution, which already include Docker in
their repositories
# If this is not the case, please, add it first to yum

sudo yum update -y
sudo yum install -y docker
sudo service docker start
sudo usermod -a -G docker ec2-user
docker run --restart=always -d \
    -p 2424:2424 \
    -p 2480:2480 \
    -e ORIENTDB_ROOT_PASSWORD=root \
    -v /opt/orientdb/databases:/orientdb/databases \
    -v /opt/orientdb/backup:/orientdb/backup \
    orientdb:latest
```

Important: keep in mind that you should replace `root` by a secure password.

[19] http://redis.io

[20] http://redis.io/topics/quickstart

[21] https://www.digitalocean.com/community/tutorials/how-to-install-and-use-redis

[22] http://orientdb.com

3.7.2 Tela API

This manual documents all the endpoints of the API of the core of Tela, and can be found (in Markdown format) at `/tela-server/docs/API.md`.

3.7.2.1 Authentication

Some of the endpoints we will describe in this document require users to authorize themselves at the moment of the request. This is done using the `Authorization` header, along with a `Bearer` token:

3.7.2.1.1 Request Headers

Header	Content	Required
Authorization	Authorization: Bearer <access_token>	Yes

The process of obtaining an `access_token` will be covered in the following section.

3.7.2.2 Auth API

3.7.2.2.1 Create a Session

```
POST /auth
```

- **Description**: Create a session and send its access token back to the user. If the module and token parameters are supplied, it will directly store the given module token within the created session (shortcut for create + add token).
- **Requires authentication**: No.
- **Output**: Tela access token.

3.7.2.2.1.1 Parameters

Name	Type	Description	Required
module	string	Module name of the token we are supplying	If token
token	string	Access token for the module	If module

3.7.2.2.1.2 Examples

Create a session:

```
POST /auth
"123456789"
```

Create a session with a module token:

```
POST /auth?module=instagram&token=12345
"987654321"
```

3.7.2.2.2 Delete a session

```
DELETE /auth
```

- **Description**: Delete a session, as well as all the module tokens stores with it.
- **Requires authentication**: Yes.
- **Output**: Success message.

3.7.2.2.2.1 Example

```
DELETE /auth
"Success"
```

3.7.2.2.3 Add a Module Token

```
POST /auth/{module}
```

- **Description**: Add a module token to the current session.
- **Requires authentication**: Yes.
- **Output**: Success message.

3.7.2.2.3.1 Parameters

Name	Type	Description	Required
module	string	Module name of the token we are supplying	Yes
token	string	Access token for the module	Yes

3.7.2.2.3.2 Example

```
POST /auth/instagram?token=12345
"Success"
```

3.7.2.2.4 Delete a Session Module

```
DELETE /auth/{module}
```

- **Description**: Delete a specific module token.
- **Requires authentication**: Yes.
- **Output**: Success message.

3.7.2.2.4.1 Parameters

Name	Type	Description	Required
module	string	Module name of the token we are supplying	Yes

3.7.2.2.4.2 Example

```
DELETE /auth/{module}
"Success"
```

3.7.2.3 General API

3.7.2.3.1 Test the Server status

```
GET /test
```

- **Description**: Simple test endpoint.
- **Requires authentication**: No.
- **Output**: OK message (in plain text).

3.7.2.3.1.1 Example Response

```
GET /test
OK
```

3.7.2.3.2 Help

```
GET /help/{module}
```

- **Description**: Get the list of the available actions, as well as its information.
- **Requires authentication**: No.
- **Output**: List of information about the actions of the provided module, or of all if none given.

3.7.2.3.2.1 Parameters

Name	Type	Description	Required
module	string	Module name	Yes

3.7.2.3.2.2 Examples

Help of all modules:

```
GET /help
[
  {
    "module": "twitter",
    "name": "self",
    "params": [
      "token: String"
    ],
    "description": "Get the information about the logged user"
  },
  {
    "module": "instagram",
    "name": "user",
    "params": [
      "token: String",
      "userId: long"
    ],
    "description": "Get the information about a user"
  }
]
```

Help of a given module:

```
GET /help/instagram
[
  {
    "module": "instagram",
    "name": "self",
    "params": [
      "token: String"
    ],
    "description": "Get the information about the logged user"
  }
]
```

3.7.2.4 Actions API

3.7.2.4.1 Execute an Action

```
GET /action/{module}/{action}
```

- **Description**: Execute an action.
- **Requires authentication**: Yes.
- **Output**: Result of the action.

3.7.2.4.1.1 Parameters

This endpoint accepts a variable number of parameters, corresponding to the parameters of the action to be executed:

```
GET /action/{module}/{action}?param1=value1&param2=value2&...
```

Name	Type	Description	Required
module	string	Module name	Yes
action	string	Action name	Yes
{param1}	*type1*	Action param	No
{param2}	*type2*	Action param	No
...	...	...	No

3.7.2.4.1.2 Examples

Instagram self:

```
GET /action/instagram/user?username=snoopdogg
{
  "username": "snoopdogg",
  "profile_picture": "snoop.jpg",
  "id": 1574083,
  "full_name": "Snoop Dogg"
}
```

Instagram followers:

```
GET /action/instagram/followers?username=themedizine&limit=1
```

```
{
  "username": "snoopdogg",
  "profile_picture": "snoop.jpg",
  "id": 1574083,
  "full_name": "Snoop Dogg"
}
```

3.7.2.5 Scheduler API

3.7.2.5.1 Schedule

```
GET /schedule/{module}/{action}
```

- **Description**: Schedule an action.
- **Requires authentication**: Yes.
- **Output**: Scheduled action and result of the action.

3.7.2.5.1.1 Parameters

This endpoint accepts a variable number of parameters, corresponding to the parameters of the action to be executed:

```
GET /action/{module}/{action}?delay=5&param1=value1&param2=value2&...
```

Name	Type	Description	Required
module	string	Module name	Yes
action	string	Action name	Yes
delay	number	Execution delay in seconds	No
{param1}	*type1*	Action param	No
{param2}	*type2*	Action param	No
...	...	...	No

3.7.2.5.1.2 Example

```
GET /schedule/instagram/user?delay=4000&username=snoopdogg
{
  "scheduledAction": {
    "createdAt": 1477865603381,
    "nextExecution": 1477866603381,
    "delay": 4000,
    "module": "instagram",
    "action": "self",
    "params": {},
    "id": 523899944
  },
  "result": {
    "username": "snoopdogg",
    "profile_picture": "snoop.jpg",
    "id": 1574083,
    "full_name": "Snoop Dogg"
  }
}
```

3.7.2.5.2 Get Scheduled Actions

```
GET /schedule
```

- **Description**: Get all the scheduled actions by the authorized user.
- **Requires authentication**: Yes.
- **Output**: Scheduled actions.

3.7.2.5.2.1 Example

```
GET /schedule
[
  {
    "createdAt": 1477865603381,
    "nextExecution": 1477866603381,
    "delay": 4000,
    "module": "instagram",
    "action": "self",
    "params": {},
    "id": 523899944
  }
]
```

3.7.2.5.3 Cancel a Scheduled Action

```
DELETE /schedule/{scheduled}
```

- **Description**: Cancel a scheduled action.
- **Requires authentication**: Yes.
- **Output**: Success message.

3.7.2.5.3.1 Parameters

Name	Type	Description	Required
scheduled	number	Scheduled action ID	Yes

3.7.2.5.3.2 Example

```
DELETE /schedule/523899944
"Success"
```

3.7.2.5.4 Cancel All the Scheduled Actions

```
DELETE /schedule
```

- **Description**: Cancel all the scheduled actions.
- **Requires authentication**: Yes.
- **Output**: Success message.

3.7.2.5.4.1 Example

```
DELETE /schedule
"Success"
```

3.7.3 Instagram Tela Module API

This manual documents all the endpoints of the API of the Instagram Tela module, and can be found (in Markdown format) at `/tela-server/docs/API-INSTAGRAM.md`.

3.7.3.1 Response models

The Tela Instagram API results completely mirrors the structure of the answers provided by the Official Instagram API, which can be seen on their API Endpoints documentation[23]. Therefore, an existing application that uses the Instagram API can be directly switched to using Tela without major changes.

3.7.3.2 Authentication

The endpoints we will describe in this document require users to authorize themselves at the moment of the request with a Tela Access Token, with an Instagram Module Token associated to it. This is done using the `Authorization` header, along with a `Bearer` token:

3.7.3.2.1 Request Headers

Header	Examle	Required
Authorization	Authorization: Bearer <access_token>	Yes

3.7.3.2.2 Obtaining an Instagram Module Token

In order to retrieve data from the official Instagram API, an Instagram access token (an Instagram module token on Tela) is required.

Obtaining a token is responsibility of each client app, so that they are able to implement and integrate the authentication process however they want. You can read more about that in the official Instagram documentation[24], particularly in the client management[25] and auth flow[26] sections.

Important: by default, new applications are created in Sandbox Mode[27] until the review, and therefore the results of the API will be limited to Sandbox Users.

3.7.3.2.3 Creating a Tela Session with the Instagram Module Token

The process of creating a session and obtaining a bearer access token, as well as associating module tokens to it, is covered in the documentation of the API of the core.

[23] https://www.instagram.com/developer/endpoints/

[24] https://www.instagram.com/developer/

[25] https://www.instagram.com/developer/clients/manage/

[26] https://www.instagram.com/developer/authentication/

[27] https://www.instagram.com/developer/sandbox/

3.7.3.3 User Actions

Actions related with information retrieval of users.

3.7.3.3.1 Self

```
GET /instagram/self
```

- **Description**: Get the information about the authorized user.
- **Requires authentication**: Yes.
- **Output**: Full information about the logged user.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: No.

3.7.3.3.1.1 Examples

```
GET /instagram/self
{
  username: 'snoopdogg',
  bio: 'Snoop Snoop Snoop!',
  website: 'http://reneses.io',
  id: 1574083,
  profile_picture: 'https://scontent.cdninstagram.com/snoop.jpg',
  full_name: 'Snoop Dogg',
  counts: {
    media: 412,
    followed_by: 822,
    follows: 420,
    created_at: 1477910384251
  }
}
```

3.7.3.3.2 Search

```
GET /instagram/search
```

- **Description**: Search a user by its username.
- **Requires authentication**: Yes.
- **Output**: Basic information about the user with the given username.
- **Schedulable**: No.
- **Requires Instagram Scope**: `public_content`.

3.7.3.3.2.1 Parameters

Name	Type	Description	Required
username	string	Username to search token we are supplying	Yes

3.7.3.3.2.2 Examples

```
GET /instagram/search?username=snoopdogg
{
  username: 'snoopdogg',
  id: 1574083,
  profile_picture: 'https://scontent.cdninstagram.com/snoop.jpg',
  full_name: 'Snoop Dogg'
}
```

3.7.3.3.3 User

```
GET /instagram/user
```

- **Description**: Get the information about a user.
- **Requires authentication**: Yes.
- **Output**: Full information about a user.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: `public_content`.

3.7.3.3.3.1 Parameters

Name	Type	Description	Required
userId	long	ID of the user to retrieve	If not username
username	string	Username of the user to retrieve	If not userId

3.7.3.3.3.2 Examples

The two following requests are equivalent:

```
GET /instagram/user?userId=1574083
GET /instagram/user?username=snoopdogg
```

Producing:

```
{
  username: 'snoopdogg',
  bio: 'Snoop Snoop Snoop!',
  website: 'http://reneses.io',
  id: 1574083,
  profile_picture: 'https://scontent.cdninstagram.com/snoop.jpg',
  full_name: 'Snoop Dogg',
  counts: {
    media: 412,
    followed_by: 822,
    follows: 420,
    created_at: 1477910384251
  }
}
```

3.7.3.3.4 Counts

```
GET /instagram/counts
```

- **Description**: Get the retrieved counts of a user.
- **Requires authentication**: Yes.
- **Output**: Counts (number of media, followers and following) of a user.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: `public_content`.

3.7.3.3.4.1 Parameters

Name	Type	Description	Required
userId	long	ID of the user to retrieve	If not username
username	string	Username of the user to retrieve	If not userId

3.7.3.3.4.2 Examples

The two following requests are equivalent:

```
GET /instagram/counts?userId=1574083
GET /instagram/counts?username=snoopdogg
```

Producing:

```
[
  {
    media: 400,
    followed_by: 800,
    follows: 419,
    created_at: 1477400384256
  },
  {
    media: 412,
    followed_by: 822,
    follows: 420,
    created_at: 1477910384251
  }
]
```

3.7.3.3.5 Following

```
GET /instagram/following
```

- **Description**: Get the users the authorized user is following.
- **Requires authentication**: Yes.
- **Output**: Basic information of the users the authorized user is following.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: `follower_list`.

3.7.3.3.5.1 Parameters

Name	Type	Description	Required
limit	int	Number of users to retrieve	No

3.7.3.3.5.2 Examples

```
GET /instagram/following?limit=1
[
  {
    username: 'snoopdogg',
    id: 1574083,
    profile_picture: 'https://scontent.cdninstagram.com/snoop.jpg',
    full_name: 'Snoop Dogg',
  }
]
```

3.7.3.3.6 Followers

```
GET /instagram/followers
```

- **Description**: Get the followers of the authorized user.
- **Requires authentication**: Yes.
- **Output**: Basic information of the followers of the authorized user.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: `follower_list`.

3.7.3.3.6.1 Parameters

Name	Type	Description	Required
limit	int	Number of users to retrieve	No

3.7.3.3.6.2 Examples

```
GET /instagram/followers?limit=1
[
  {
    username: 'snoopdogg',
    id: 1574083,
    profile_picture: 'https://scontent.cdninstagram.com/snoop.jpg',
    full_name: 'Snoop Dogg',
  }
]
```

3.7.3.3.7 Friends

```
GET /instagram/friends
```

- **Description**: Get the friends (intersection of followers and following) of the authorized user.
- **Requires authentication**: Yes.
- **Output**: Basic information of the friends of the authorized user.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: `follower_list`.

3.7.3.3.7.1 Examples

```
GET /instagram/friends
[
  {
    username: 'snoopdogg',
    id: 1574083,
    profile_picture: 'https://scontent.cdninstagram.com/snoop.jpg',
    full_name: 'Snoop Dogg'
  }
]
```

3.7.3.3.8 Relationship

```
GET /instagram/relationship
```

- **Description**: Get the relationship between the authorized user and a user.
- **Requires authentication**: Yes.
- **Output**: Relationship.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: `follower_list`.

3.7.3.3.8.1 Parameters

Name	Type	Description	Required
userId	long	ID of the other user	If not username
username	string	Username of the other user	If not userId

3.7.3.3.8.2 Response values

- **Outgoing**: `follows`, `requested`, `none`.
- **Incoming**: `followed_by`, `requested_by`, `blocked_by_you` and `none`.

3.7.3.3.8.3 Examples

The two following requests are equivalent:

```
GET /instagram/relationship?userId=1574083
GET /instagram/relationship?username=snoopdogg
```

Producing:

```
{
  "outgoing_status": "follows",
  "incoming_status": "followed_by",
  "target_user_is_private": false
}
```

3.7.3.4 Media Actions

Actions related with information retrieval of media, comments and likes.

3.7.3.4.1 Self Media

```
GET /instagram/self-media
```

- **Description**: Get the latest media of the authorized user.
- **Requires authentication**: Yes.
- **Output**: Latest media of the authorized user.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: `public_content`.

3.7.3.4.1.1 Parameters

Name	Type	Description	Required
limit	int	Number of media to retrieve	No

3.7.3.4.1.2 Examples

```
GET /instagram/self-media?limit=1
[
  {
    "id": "1358943_477",
    "caption": {
      "id": 1786524624,
      "text": "What a night!",
      "created_time": 1477840385,
      "from": {
        "username": "snoopdogg",
        "id": 1574083,
        "profile_picture": "https://scontent.cdninstagram.com/snoop.jpg",
        "full_name": "Snoop Dogg",
      }
    },
    "link": "https://www.instagram.com/p/BMMRE9dDGvf/",
```

```
    "type": "image",
    "filter": "Rise",

    "user": {

      "username": "snoopdogg",
      "id": 1574083,
      "profile_picture": "https://scontent.cdninstagram.com/snoop.jpg",
      "full_name": "Snoop Dogg",
    },
    "tags": [
      "instapic"
    ],
    "location": null,
    "images": {
      "thumbnail": {
        "url": "https://scontent.cdninstagram.com/s150x150/1.jpg",
        "size": "thumbnail",
        "width": 150,
        "height": 150
      },
      "low_resolution": {
        "url": "https://scontent.cdninstagram.com/s320x320/1.jpg",
        "size": "low_resolution",
        "width": 320,
        "height": 320
      },
      "standard_resolution": {
        "url": "https://scontent.cdninstagram.com/s640x640/1.jpg",
        "size": "standard_resolution",
        "width": 640,
        "height": 640
      }
    },
    "videos": {},
    "comments": {
      "data": [],
      "count": 0
    },
    "likes": {
      "data": [],
      "count": 87
    },
    "created_time": 1477840385,
    "users_in_photo": [
      {
        "position": {
          "x": 0.2995169082125604,
          "y": 0.2526978142827535
        },
        "user": {
          "username": "Pedro",
          "id": 389893,
          "profile_picture": "https://pedro.jpg",
          "full_name": "Dj Pedro"
        }
```

```
      },
      {
        "position": {
          "x": 0.7141706678602431,
          "y": 0.2455035971223022
        },
        "user": {
          "username": "borja",
          "id": 48333393,
          "profile_picture": "https://borja.jpg",
          "full_name": "Borja"
        }
      }
    ],
    "user_has_liked": false,
    "is_video": false,
    "is_image": true
  }
]
```

3.7.3.4.2 Likes

```
GET /instagram/likes
```

- **Description**: Get the likes of a media.
- **Requires authentication**: Yes.
- **Output**: Users that have liked the media.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram Scope**: `public_content`.

3.7.3.4.2.1 Parameters

Name	Type	Description	Required
mediaId	String	ID of the media	Yes

3.7.3.4.2.2 Examples

```
GET /instagram/likes&mediaId=3948SD
[{
  "username": "snoopdogg",
  "id": 1574083,
  "profile_picture": "https://scontent.cdninstagram.com/snoop.jpg",
  "full_name": "Snoop Dogg",
  }, {
  "username": "Pedro",
   "id": 389893,
   "profile_picture": "https://pedro.jpg",
```

```
   "full_name": "Dj Pedro"
}]
```

3.7.3.4.3 Comments

```
GET /instagram/comments
```

- **Description**: Get the comments of a media.
- **Requires authentication**: Yes.
- **Output**: Comments of the media.
- **Schedulable**: Yes (minimum delay: 3600s).
- **Requires Instagram** Scope: `public_content`

3.7.3.4.3.1 Parameters

Name	Type	Description	Required
mediaId	String	ID of the media	Yes

3.7.3.4.3.2 Examples

```
GET /instagram/comments&mediaId=3948SD
[{
    "id": 1786344,
    "text": "Amazing!",
    "created_time": 1477840385,
    "from": {
      "username": "snoopdogg",
      "id": 1574083,
      "profile_picture": "https://scontent.cdninstagram.com/snoop.jpg",
      "full_name": "Snoop Dogg",
    }
  },{
    "id": 1786241,
    "text": "Amazing!",
    "created_time": 1477840385,
    "from": {
      "username": "Pedro",
      "id": 389893,
      "profile_picture": "https://pedro.jpg",
      "full_name": "Dj Pedro"
   }
}]
```

3.7.4 Tela Twitter API

This manual documents all the endpoints of the API of the Twitter Tela Module.

3.7.4.1 Response models

The Tela Twitter API results completely mirrors the structure of the answers provided by the Official Twitter API, which can be seen on their API Endpoints documentation[28]. Therefore, an existing application that uses the Twitter API can be directly switched to using Tela without major changes.

3.7.4.2 Authentication

The endpoints we will describe in this document require users to authorize themselves at the moment of the request with a Tela Access Token, with a Twitter Module Token associated with it. This is done using the `Authorization` header, along with a `Bearer` token:

3.7.4.2.1 Request Headers

Header	Examle	Required
Authorization	Authorization: Bearer <access_token>	Yes

3.7.4.2.2 Obtaining a Twitter Module Token

In order to retrieve data from the official Twitter API, a Twitter access token (a Twitter module token on Tela) is required.

Obtaining a token is responsibility of the each client app, so that they are able to implement and integrate the authentication process however they want. You can read more about that in the official Twitter documentation[29], particularly in the client management[30] and auth flow[31] sections.

3.7.4.2.3 Creating a Tela Session with the Twitter Module Token

The process of creating a session and obtaining a bearer access token, as well as associating module tokens to it, is covered in the documentation of the API of the core.

3.7.4.3 User Actions

Actions related with information retrieval of users.

[28] https://dev.twitter.com/rest/public

[29] https://dev.twitter.com/rest/public

[30] https://apps.twitter.com/app/new

[31] https://dev.twitter.com/oauth/application-only

3.7.4.3.1 Self

```
GET /twitter/self
```

- **Description**: Get the information about the authorized user.
- **Requires authentication**: Yes.
- **Output**: Full information about the logged user.
- **Schedulable**: Yes (minimum delay: 3600s).

3.7.4.3.1.1 Examples

```
GET /twitter/self
{
  screen_name: 'snoopdogg',
  url: 'http://twitter.com/snoopdogg',
  id: 1574083,
  profile_image_url: 'https://twitter.com/snoop.jpg',
  full_name: 'Snoop Dogg',
  followers_count: 822,
  friends_count: 420
}
```

3.7.4.3.2 Following

```
GET /twitter/following
```

- **Description**: Get the users the authorized user is following.
- **Requires authentication**: Yes.
- **Output**: Basic information of the users the authorized user is following.
- **Schedulable**: Yes (minimum delay: 3600s).

3.7.4.3.2.1 Parameters

Name	Type	Description	Required
limit	int	Number of users to retrieve	No

3.7.4.3.2.2 Examples

```
GET /twitter/following?limit=1
[
  {
    screen_name: 'snoopdogg',
    url: 'http://twitter.com/snoopdogg',
    id: 1574083,
    profile_image_url: 'https://twitter.com/snoop.jpg',
    full_name: 'Snoop Dogg',
    followers_count: 822,
    friends_count: 420
  }
]
```

3.7.4.3.3 Followers

```
GET /twitter/followers
```

- **Description**: Get the followers of the authorized user.
- **Requires authentication**: Yes.
- **Output**: Basic information of the followers of the authorized user.
- **Schedulable**: Yes (minimum delay: 3600s).

3.7.4.3.3.1 Parameters

Name	Type	Description	Required
limit	int	Number of users to retrieve	No

3.7.4.3.3.2 Examples

```
GET /twitter/followers?limit=1
[
  {
    screen_name: 'snoopdogg',
    url: 'http://twitter.com/snoopdogg',
    id: 1574083,
    profile_image_url: 'https://twitter.com/snoop.jpg',
    full_name: 'Snoop Dogg',
    followers_count: 822,
    friends_count: 420
  }
]
```

3.7.4.3.4 Friends

```
GET /twitter/friends
```

- **Description**: Get the friends (intersection of followers and following) of the authorized user.
- **Requires authentication**: Yes.
- **Output**: Basic information of the friends of the authorized user.
- **Schedulable**: Yes (minimum delay: 3600s).

3.7.4.3.4.1 Examples

```
GET /twitter/friends
[
  {
    screen_name: 'snoopdogg',
    url: 'http://twitter.com/snoopdogg',
    id: 1574083,
    profile_image_url: 'https://twitter.com/snoop.jpg',
    full_name: 'Snoop Dogg',
    followers_count: 822,
    friends_count: 420
  }
]
```

3.7.5 Developing a Module

This document covers the process of developing and building a module extending the functionality of Tela, and can be found (in Markdown format) at `/tela-server/docs/MODULE-DEVELOPMENT.md`.

3.7.5.1 Prerequisites

This tutorial assumes that Tela has already been installed via Maven, or that we are directly coding in the original source code.

In order to test the module we will use the tool Tela CLI, connecting to the local server we will be developing:

```
npm install -g tela-cli
tela-cli connect localhost 8080
```

3.7.5.2 Style Conventions

3.7.5.2.1 File structure

This is the recommended file structure:

```
.
└── [Module name]
    ├── [Module name]TelaModule.java
    ├── actions
    │   └── (Action classes)
    ├── api
    │   ├── (Classes interacting with external APIs)
    │   ├── exceptions
    │   │   └── (API exceptions)
    │   ├── models
    │   │   └── (API-specific models)
    │   └── responses
    │       └── (Response models from the external API)
    ├── cache
    │   └── (Cache implementations)
    ├── databases
    │   └── extensions
    │       └── (Database extensions)
    ├── models
    │   └── (Module models)
    └── repositories
        └── (Repositories)
```

3.7.5.3 Developing the Module

3.7.5.3.1 Creating a Basic Module

Let's create a new package called `tutorial` for our module. Inside, we will create a `TutorialTelaModule` class, extending `TelaModule`. It will simply invoke the parent constructor, passing its name.

```
package io.reneses.tela.modules.tutorial;
import io.reneses.tela.modules.TelaModule;

public class TutorialTelaModule extends TelaModule {
    public static final String NAME = "tutorial";
    public TutorialTelaModule() {
        super(NAME);
    }
}
```

Then, we will assemble the server registering the created module:

```
package io.reneses.tela;
import io.reneses.tela.modules.tutorial.TutorialTelaModule;

public class App {
    public static void main(String[] args) {
        Configuration config = ConfigurationFactory.create();
        config.setProperty(Configuration.Property.SCHEDULER_DELAY, 5);
        config.setProperty(Configuration.Property.ORIENTDB_MODE,
Configuration.OrientDbMode.MEMORY);
        Assembler.build(config, new TutorialTelaModule()).start();
    }
}
```

Note: in this code we have configure a short scheduler delay and the temporary mode for the database. This is not needed so far, but will be used in the future.

3.7.5.3.2 Adding our First 'Hello World' Action

Let's add a simple action to the module. First, create the file `/actions/Actions.java` and copy the following:

```
@Module("tutorial")
public class Actions {

    @Action(parameters = {})
    public String hello() {
        return "Hello World";
    }

}
```

If we execute the main method, the server will output at the console:

```
DEBUG   Action loaded: tutorial/hello []
```

And we will able to test it with Tela CLI:

```
> tela-cli execute tutorial hello
Hello World!
```

3.7.5.3.3 Adding a Parameter to the 'Hello World' Action

We will create another hello method admitting a parameter:

```
@Action(parameters = {"name"})
public String hello(String name) {
    return "Hello " + name + "!";
}
```

If we execute the main method, the server will output at the console:

```
18:12:34    DEBUG   Action loaded: tutorial/hello []
18:12:34    DEBUG   Action loaded: tutorial/hello [name]
```

And we will able to test it with Tela CLI:

```
> tela-cli execute tutorial hello name=Josh
Hello Josh!
```

3.7.5.3.4 Adding an Integer Parameter and Specifying an Action Name

Parameters are automatically casted, we just have to create a method with it and include it in the `@Action` annotation:

```
@Action(name = "hello", parameters = {"name", "age"})
public String helloWithAge(String name, int age) {
  return "Hello " + name + " of " + age + " years!";
}
```

By default, Tela uses the method name as action name. If we want to specify another value, we can do it with the `name` property of the `@Action` annotation.

Now we can restart the server and try the new method:

```
> tela-cli execute tutorial hello name=Josh age=10
Hello Josh of 10 years!
```

3.7.5.3.5 Adding an Array Parameter

Parameters can also be arrays. In order to assign a name to the corresponding parameters, single nouns are recommended, as the URL syntax for an array is `?key=value1&key=value2&key=value3`.

```
@Action(name = "hello", parameters = {"friend"})
public String helloFriends(String[] name) {
    return "Hello " + String.join(", ", (CharSequence[]) name) + "!";
}
```

Now we can restart the server and try the new method:

```
> tela-cli execute tutorial hello friend=Josh friend=Peter friend=Mark
Hello Josh, Peter, Mark!
```

3.7.5.3.6 Adding a Description and Generating Help

The `description` property of the `@Action` annotation is useful in order to describe what the action performs. Let's annotate our actions:

```
@Action(description = "Simple hello", parameters = {})
public String hello() {
    return "Hello World!";
}

@Action(description = "Hello with name", parameters = {"name"})
public String hello(String name) {
    return "Hello " + name + "!";
}

@Action(name = "hello", description = "Hello with name and age",
        parameters = {"name", "age"})
public String helloFriends(String name, int age) {
    return "Hello " + name + " of " + age + " years!";
}

@Action(name = "hello", description = "Hello to all your friends!",
        parameters = {"friend"})
public String helloFriends(String[] name) {
    return "Hello " + String.join(", ", (CharSequence[]) name) + "!";
}
```

Now, if we retrieve the help:

```
> tela-cli help
tutorial/hello
 - Description:  Hello with name
 - Parameters: name: String
tutorial/hello
 - Description:  Simple hello
tutorial/hello
 - Description:  Hello with name and age
 - Parameters: name: String, age: int
tutorial/hello
 - Description:  Hello to all your friends!
 - Parameters: friend: String[]
```

3.7.5.3.7 Adding Action Scheduling

Scheduling functionality can be added to an action using the `@Schedulable` annotation, along with the `minimumDelay` property (in seconds).

Let's create a schedulable method:

```
@Action(name = "hello-date", description = "Hello with date!", parameters = {})
@Schedulable(minimumDelay = 1)
public String helloDate() {
    return "Hello again! (At: " + new Date() + ")";
}
```

And execute it:

```
> execute tutorial hello-date
Hello again! (At: Wed Nov 02 13:44:42 CET 2016)
> tela-cli schedule 1 tutorial hello-date | jq
{
  "scheduledAction": {
    "id": 280354576,
    "delay": 1,
    "accessToken": "34grs87g65ebhj08v1in3sg39l",
    "params": {},
    "createdAt": 1478090696296,
    "nextExecution": 1478090697296,
    "module": "tutorial",
    "action": "hello-date"
  },
  "result": "Hello again! (At: Wed Nov 02 13:44:56 CET 2016)"
}
```

In the server logs we can see how the scheduled action was actually executed:

```
13:44:47    DEBUG   Scheduler executed
13:44:52    DEBUG   Scheduler executed
13:44:56    INFO    [Scheduler] Scheduled action: tutorial/hello-date (1s) [] with ID
280354576, next: 2016-11-02T13:44:57.296
13:44:56    INFO    127.0.0.1:tpic6pskdg5ptmklftqljb10ch:34grs87g65ebhj08v1in3sg39l
[Schedule] tutorial/hello-date each 1s
13:44:57    DEBUG   Scheduler executed
13:44:57    INFO    [Scheduler] Executing scheduled task: tutorial/hello-date (1s) []
with ID 280354576, next: 2016-11-02T13:44:57.296
13:45:02    DEBUG   Scheduler executed
13:45:02    INFO    [Scheduler] Executing scheduled task: tutorial/hello-date (1s) []
with ID 280354576, next: 2016-11-02T13:44:58.296
13:45:07    DEBUG   Scheduler executed
13:45:07    INFO    [Scheduler] Executing scheduled task: tutorial/hello-date (1s) []
with ID 280354576, next: 2016-11-02T13:45:03.296
13:45:09    INFO    127.0.0.1:tpic6pskdg5ptmklftqljb10ch:13:45:17   DEBUG   Scheduler
executed
```

3.8 System Outcome and Extensions

3.8.1 System Outcome

The outcome of this system is a fully functional 1.0 version, which could be deployed in controlled production environments. The application has 308 tests, and its source (excluding tests) is 11,000 lines long.

The following image is a screenshot of how Tela looks while running:

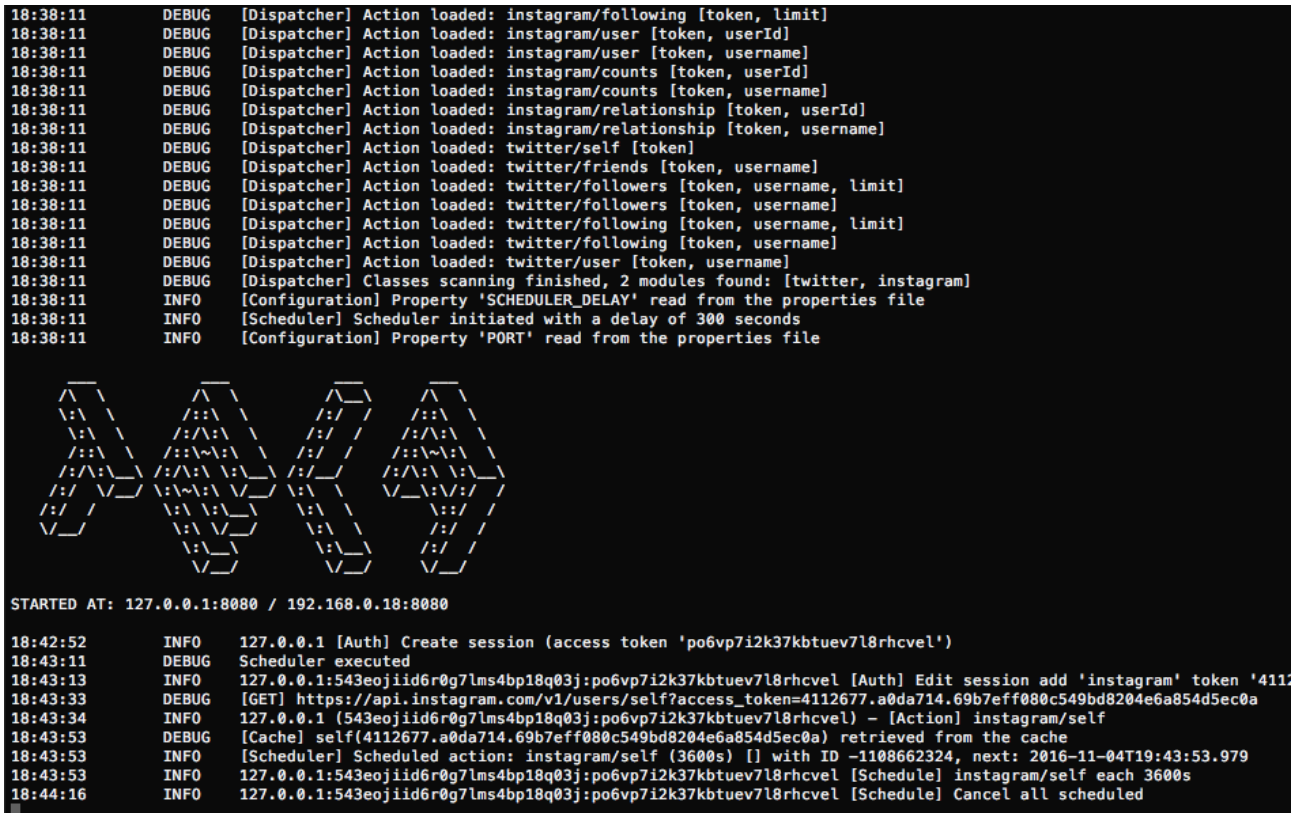


Figure 3.39 Tela Running Screenshot

3.8.2 Extensions

In case of continuing with the project, the next steps should be:

- Analyze the Checkstyle and Sonar reports, fixing the problems found and therefore improving the quality of the code.
- Implement non-blocking I/O via promises.
- Design performance tests, and use them in order to improve the number of requests per second Tela can handle.
- Complete the Twitter API.

3.9 Content delivered

We will include a packaged version of Tela, called `tela-server-1.0-jar.zip`. In addition to that, the source code of the project can be found at the tela-server folder. Within it, we find:

Folder: /tela-server	
Directories	
Folder	**Contents**
/docs	Documentation
/scripts	Scripts to help the deployment
/src	Java source code
/target	Build folder (once building & packaging has been completed)
Directory: /tela-server/src	
Folder	**Contents**
main/java	Source code of the application
main/resources	Resources of the application
test/java	Source code of the tests
test/resources	Resources of the tests
Directory: /tela-server/target/jar	
Folder	**Contents**
original-tela.jar	Tela Jar
tela.jar	Shaded Jar with all the dependencies
tela.properties	Configuration file
Files	
File	**Description**
.env	Environmental variables, to be used by Heroku Local.
.gitignore	Content ignored by Git.
docker-compose.yml	Docker Compose configuration
Dockerfile	Docker image build
pom.xml	Maven file
Procfile	Heroku deployment configuration
README.md	Index of documentation
server.log	Server log (once it has been executed)
sonar-project.properties	Sonar configuration

4 System II: Tela CLI

4.1 Introduction

4.1.1 Description of the System

Tela CLI is a multiplatform tool to interact with a Tela Server from the command line interface. Although its usage is more rudimentary than an application with a visual interface, it might be particularly interesting in the following cases:

- Users whose devices do not have a desktop environment (e.g. Unix servers).
- Scripting. As a regular command line interface tool, Tela CLI can be used within scripts. For example, a CRON-programmed task, or a script fired by another program.
- Tela module developers, who simply want to test the functionality they are working on.
- Users that want to execute simple tasks and manually review the returned JSON.
- Users that will process the returned JSON with another software.

4.1.2 Scope

Within this project, this system has two purposes:

- Help during the development of the System I (Tela Server), providing an easy way to interact with it.
- Act as a proof of concept of an application that interacts with a Tela Server.

For these reasons, some aspects of the system (e.g. automated tests and error handling) have been eliminated or simplified.

On the other hand, as this is the easiest way to communicate and test a Tela Server, we will develop a completely functional system, with a complete and detailed the usage manual.

4.1.3 Development Process

Tela CLI will be used during the development of Tela, to not only check the functionality of both systems, but also to test how smooth the interaction between an application and the server is.

For this reason, and as Tela will be developed in successive iterations, we will also use an iterative agile approach for this tool. However, as the requirements of this system are well defined, and it is not very complex, we will have longer iterations.

4.2 Analysis of the System

4.2.1 Analysis of Use Cases

The purpose of the Tela CLI system is being able to access the full functionality of Tela from the command line. Therefore, its use cases can be seen in the following diagram:

- Connect/disconnect to/from a Tela Server.
- Execute an action in the Tela Server.
- Scheduling actions, and canceling its scheduled execution.
- Obtain help regarding the installed modules in the Tela Server.
- Linking/unlinking modules (creating and storing module tokens).

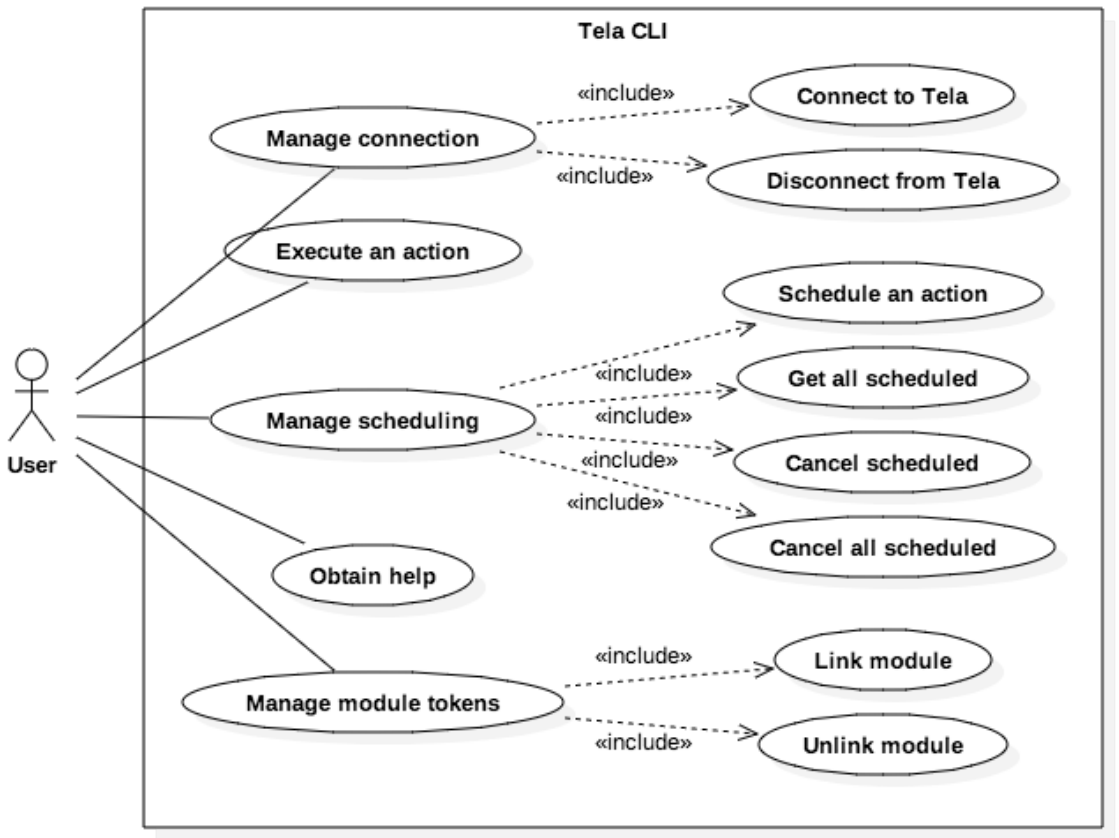


Figure 4.1 Tela CLI Use Cases Diagram

4.2.2 Specification of System Requirements

From the use cases identified in the previous section, we will elaborate a list of functional and non-functional requirements the system should meet. This will be the base for the design phase, and our focus during testing.

4.2.2.1 Functional Requirements

ID	Name	Description
FR1	Connection management	
FR1.1	Connection	Connect to a Tela Server, creating a session and storing the returned access token.
FR1.2	Disconnection	Disconnect from the Tela Server, deleting the session as well as all the associated tokens, and delete the stored access token.
FR2	Action execution	
FR2.1	Execute action	Execute an action in the Tela Server with the supplied parameters, and output its result.
FR3	Scheduling management	
FR3.1	Schedule	Schedule an action to be executed periodically with a given delay, outputting its information, as well as its result.
FR3.2	Get scheduled	Retrieve and output all the actions that the current user has scheduled.
FR3.3	Cancel scheduled	Cancel the scheduled execution of a certain action.
FR3.4	Cancel all scheduled	Cancel the scheduled execution of all the actions scheduled by the current user.
FR4	Obtain help	
FR4.1	Help	Obtain information about all the modules installed in the Tela Server, as well as about their actions.
FR4.2	Module help	Obtain information about all the actions from a given module.
FR5	Module token management	
FR5.1	Configure module	Configure module-specific properties.
FR5.2	Link module	Create a module token and store it into the current Tela session.
FR5.3	Unlink module	Delete a module token from the current Tela session.

4.2.2.2 Non-Functional Requirements

ID	Description
NFR1	Installation
NFR1.1	The tool will be easy to install, regardless of the system.
NFR1.2	The tool will be multiplatform.
NFR2	Documentation
NFR2.1	The tool will have a manual detailing about its installation and usage.
NFR3	Execution
NFR3.1	The tool will be executed from the console as a global command.
NFR3.2	The connection will persist, even if the system is restarted or the folder of the script is moved.
NFR3.3	The module configuration will persist, even if the system is restarted or the folder of the script is moved.
NFR3.4	The tool will properly handle the errors and incorrect inputs, and present understandable error messages.
NFR4	Implementation
NFR4.1	The tool will be easily extensible, so that new modules can be added to it, in the same way they can be added to the server.

4.2.3 Design of Use Case Scenarios

Due to the limited scope of the system and the common error handling strategy (just printing the error message), the following use case scenarios will omit the alternate flows.

4.2.3.1 Connection Management (FR1)

Connection (FR1.1)	
Description	Connect to a Tela Server so that it is possible to start executing authorized commands.
Primary Actors	User & Server
Preconditions	A Tela Server is running at a known host and port.
Post conditions	A session has been created in the Tela Server and its access token has been saved in the client.
Trigger	The user executes the connect command.
Flow	1. The user enters the host and port of the Tela Server. 2. The system send a request to the server to create a session. 3. The server replies with the access token of the created session 4. The system stores the retrieved access token.

	5. The system outputs a success message.

Disconnection (FR1.2)	
Description	Disconnect from the Tela Server, deleting the session as well as all the associated tokens, and delete the stored access token.
Primary Actor	User & Server
Preconditions	A connection has been established with a running Tela Server.
Post conditions	The session and its tokens are deleted from the server, as well as the access token from the client.
Trigger	The user executes the disconnect command.
Flow	1. The system sends a request to the server to delete the session and its tokens. 2. The server deletes the session and its tokens 3. The server sends a success message back 4. The system deletes the stored access token. 5. The system outputs a success message.

4.2.3.2 Action Execution (FR2)

Execute action (FR2.1)	
Description	Execute an action in the Tela Server and output its result.
Primary Actors	User & Server.
Preconditions	A connection has been established with a running Tela Server.
Post conditions	-
Trigger	The user executes the execute command.
Flow	1. The user enters the module, name and parameters of the action to execute. 2. The system sends a request to the server to execute the action. 3. The server executes it and returns its result. 4. The system outputs the action result.

4.2.3.3 Scheduling Management (FR3)

Schedule (FR3.1)	
Description	Schedule an action to be executed periodically and output its information and result.
Primary Actors	User & Server.
Preconditions	A connection has been established with a running Tela Server.
Post conditions	The action has been scheduled.
Trigger	The user executes the schedule command.
Flow	1. The user enters the delay, module, name and parameters of the action to schedule. 2. The system sends a request to the server to schedule the action. 3. The server executes it, schedules it, and returns its information and result. 4. The system outputs the information about the scheduled action and its result.

Get scheduled (FR3.2)	
Description	Retrieve and output all the actions that the current user has scheduled.
Primary Actors	User & Server.
Preconditions	A connection has been established with a running Tela Server.
Post conditions	-
Trigger	The user executes the scheduled command.
Flow	1. The system sends a request to the server to get the scheduled actions. 2. The server returns all the actions scheduled by the authorized user. 3. The system outputs the scheduled actions returned.

Cancel scheduled (FR3.3)	
Description	Cancel the scheduled execution of a scheduled action.
Primary Actors	User & Server.
Preconditions	A connection has been established with a running Tela Server, and has an action scheduled (with a known ID).
Post conditions	-
Trigger	The user executes the cancel command.
Flow	1. The user enters the ID of the scheduled action. 2. The system sends a request to the server to cancel the scheduled action with the given ID. 3. The server returns a success response. 4. The system outputs a success message.

Cancel all scheduled (FR3.4)	
Description	Cancel all the scheduled execution by the authorized user.
Primary Actors	User & Server.
Preconditions	A connection has been established with a running Tela Server.
Post conditions	-
Trigger	The user executes the cancel command.
Flow	1. The system sends a request to the server to cancel all the scheduled actions. 2. The server returns a success response. 3. The system outputs a success message.

4.2.3.4 Obtain Help (FR4)

Help (FR4.1)	
Description	Obtain information about all the modules and their actions installed in the Tela Server.
Primary Actors	User & Server.
Preconditions	A connection has been established with a running Tela Server.
Post conditions	-
Trigger	The user executes the help command.
Flow	1. The system sends a request to the server to obtain the help of all the modules. 2. The server returns the information about them. 3. The system outputs the help.

Help (FR4.2)	
Description	Obtain information about a modules and its actions.
Primary Actors	User & Server.
Preconditions	A connection has been established with a running Tela Server.
Post conditions	-
Trigger	The user executes the help command.
Flow	1. The user enters the name of the module he wants to obtain the help about. 2. The system sends a request to the server to obtain the help of the modules. 3. The server returns the information about it. 4. The system outputs the help.

4.2.3.5 Module Token Management (FR5)

Configure module (FR5.1)	
Description	Configure a module-specific property.
Primary Actors	User.
Preconditions	A connection has been established with a running Tela Server.
Post conditions	-
Trigger	The user executes the configure command.
Flow	1. The user enters the module, property and value he wants to set. 2. The system checks that it is a valid property. 3. The system stores the property and its value. 4. The system outputs a success message.

Link module (FR5.2)	
Description	Create a module token and store it into the current Tela session.
Primary Actors	User, Module API & Server.
Preconditions	A connection has been established with a running Tela Server, and a certain module is installed within the Tela CLI.
Post conditions	The created module token has been stored into the Tela session in the server.
Trigger	The user executes the link command.
Flow	1. The user enters the module he wants to link. 2. The system communicates with the corresponding external API (how this is done is responsibility of the modules installed in the system). 3. The external API returns a module token. 4. The system sends a request to the server with the module name and the token. 5. The server stores the module token into the session of the authorized user. 6. The server returns a success response. 7. The system outputs a success message.

Unlink module (FR5.3)	
Description	Delete a module token from the current Tela session.
Primary Actors	User & Server
Preconditions	A connection has been established with a running Tela Server, and a certain module token is stored into the authorized Tela session.
Post conditions	The token has been deleted from the session in the server

Trigger	The user executes the unlink command.
Flow	1. The user enters the module he wants to unlink. 2. The system sends a request to the server to delete the token of the module. 3. The server deletes the module token from the session of the authorized user. 4. The server returns a success response. 5. The system outputs a success message.

4.2.4 Analysis of Classes and Packages

This system will have three main packages:

- The `api`, containing the functionality to interact with the Tela server.
- The `modules`, which are pluggable extensions, similar to the modules in Tela. The main difference is that its purpose is obtaining module tokens (i.e. log the user into the associate social network retrieving an access token). It is not needed to create a module in the system for each module that exists in Tela, only for those that we will need to obtain a module token.
- The `connection`, managing the creation and storage of Tela sessions.

Then, the `TelaCli` will be the main class offering all the functionality of Tela. In order to be accessible from the command line interface, the `cli` script wraps `TelaCli` parsing and mapping the user inputs as method calls and parameters.

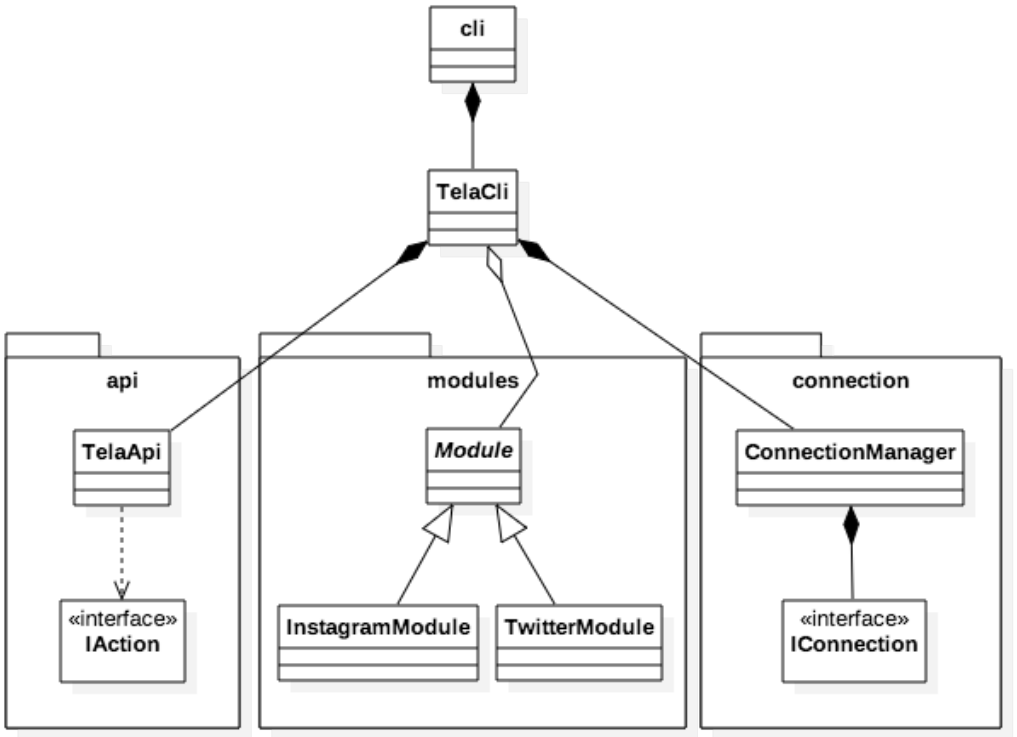


Figure 4.2 Tela CLI Class Diagram (Overview)

4.3 Design

This is an elaborate version of the class diagram shown in the previous section:

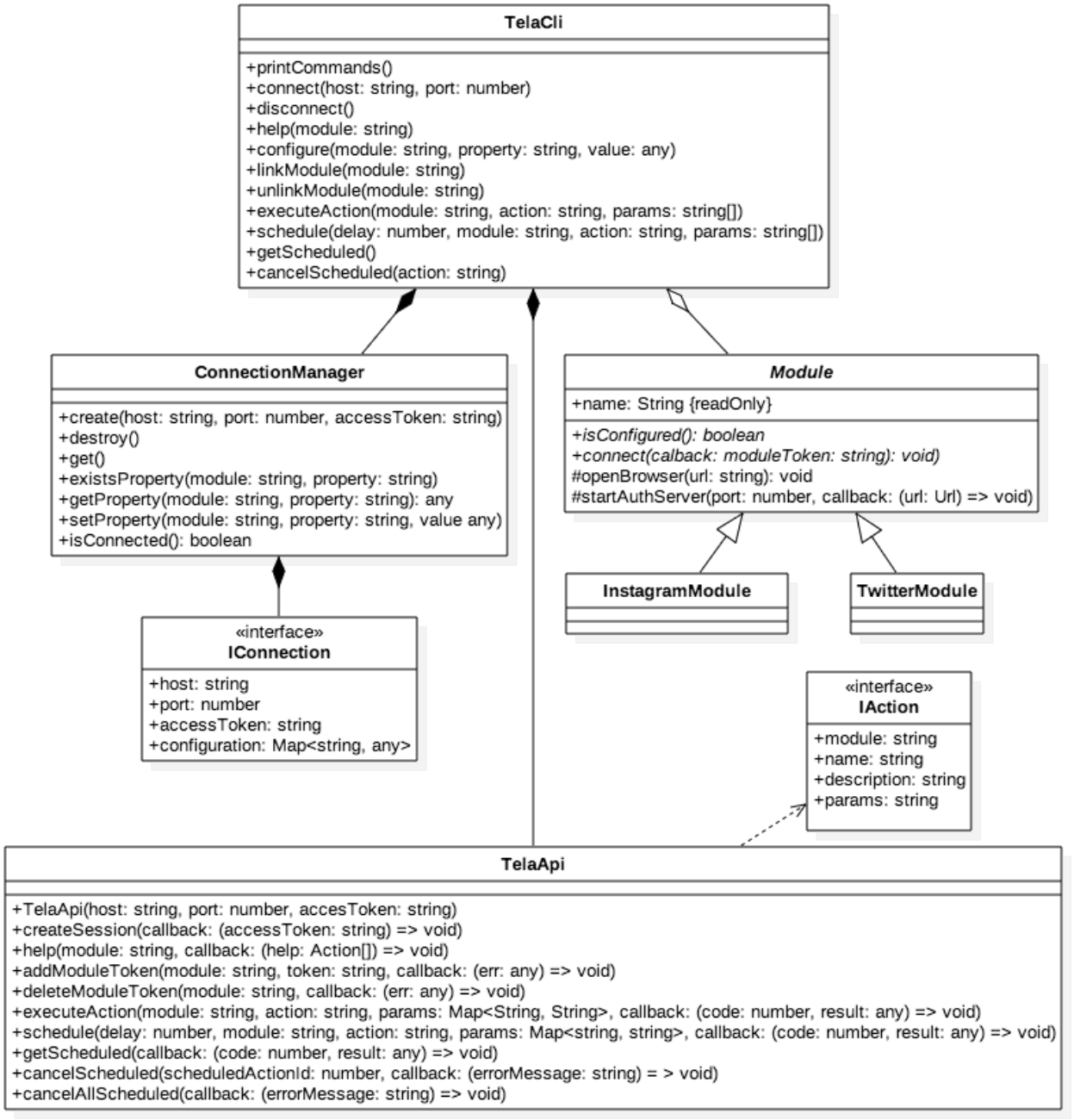


Figure 4.3 Tela CLI Class Diagram

4.4 Test Plan

4.4.1 Testing Strategy

4.4.1.1 Unit Tests

Due to the small size of the system, and the limited scoped of it as it acts as a proof of concept for a client application interacting with the Tela Server, unit tests have been omitted.

4.4.1.2 Integration Testing

Testing scripts will be designed outlining a sequence of tasks to perform in order to check the compliance with the functional and non-functional requirements.

4.4.1.3 System Tests

System Tests will be designed and checked in order to ensure that the tool can be successfully executed from the command line, regardless of the operating system.

4.4.2 Testing Outline for Integration and System Tests

The following tests will be manually executed:

Requirement FR1: Connect to a Tela Server	
Test case: TC1.1.1	
Description	Connect to a Tela Server which exists and it is running
Result expected	The system connects and creates a token in the Tela Server, which is stored within the system. If we execute the system again, the connection is kept.
Test case: TC1.1.2	
Description	Connect to a not existing Tela Server
Result expected	Error message is displayed
Test case: TC1.2.1	
Description	Disconnect from the Tela Server the system is connected to
Result expected	The session is deleted from the Server. If we execute the system again, it is not connected
Test case: TC1.2.2	
Description	Try to disconnect without being connected
Result expected	Error message is displayed.

Requirement FR2: Action Execution	
Test case: TC2.1.1	
Description	Execute an action that exists
Result expected	Results are returned and outputted to the console
Test case: TC2.1.2	
Description	Execute an action whose module does not exist
Result expected	Error message is displayed
Test case: TC2.1.3	
Description	Execute an action that does not exist
Result expected	Error message is displayed
Test case: TC2.1.4	
Description	Execute an action with no connection
Result expected	Error message is displayed

Requirement FR3: Scheduling management	
Test case: TC3.1.1	
Description	Schedule an action
Result expected	The action is correctly scheduled in the server
Test case: TC3.1.2	
Description	Schedule an action with no connection
Result expected	Error message is shown
Test case: TC3.2.1	
Description	Schedule an action, and request information about it by its ID
Result expected	The information received and displayed is the same than the action we had
Test case: TC3.2.2	
Description	Try to obtain information about a non-existing action
Result expected	Error message is displayed
Test case: TC3.2.3	
Description	Get information of an scheduled action with no connection
Result expected	Error message is shown
Test case: TC3.3.1	
Description	Schedule an action, and cancel it
Result expected	The action is canceled in the server

Test case: TC3.3.2	
Description	Cancel a scheduled action that does not exist
Result expected	Error message is displayed
Test case: TC3.3.3	
Description	Cancel an action with no connection
Result expected	Error message is shown
Test case: TC3.4.1	
Description	Schedule several actions and then cancel all
Result expected	All the actions are canceled in the server
Test case: TC3.4.2	
Description	Cancel all the scheduled actions, with no actions
Result expected	Error message is displayed
Test case: TC3.4.3	
Description	Cancel all the actions actions with no connection
Result expected	Error message is shown

Requirement FR4: Obtain help	
Test case: TC4.1.1	
Description	Obtain help about all the modules
Result expected	Results are returned and outputted to the console
Test case: TC4.1.2	
Description	Obtain help about all the modules with no connection
Result expected	Error message is displayed
Test case: TC4.2.1	
Description	Obtain help about one modules
Result expected	Results are returned and outputted to the console
Test case: TC4.2.2	
Description	Obtain help about a module that does not exist
Result expected	Error message is displayed
Test case: TC4.2.3	
Description	Obtain help about a module with no connection
Result expected	Error message is displayed

Requirement FR5: Module token management	
Test case: TC5.1.1	
Description	Configure a module
Result expected	The property is stored, and persist
Test case: TC5.2.1	
Description	Link a module
Result expected	The module is linked and the module token is stored in the server
Test case: TC5.2.3	
Description	Link a module with no connection
Result expected	Error message is displayed
Test case: TC5.3.1	
Description	Unlink a module that has been previously linked
Result expected	The module token is deleted in the server
Test case: TC5.3.2	
Description	Unlink a module that does not exist
Result expected	Error message is displayed
Test case: TC5.3.3	
Description	Unlink a module that with no connection
Result expected	Error message is displayed

Requirement NFR1: Installation	
Test case: TC6.1.1.1	
Description	Install the tool globally via npm
Result expected	The tool can be executed from the console
Test case: TC6.1.2.1	
Description	Install and execute the tool in Windows and Mac
Result expected	The system works in both operating systems
Test case: TC6.3.3.1	
Description	Connect and configure the system, then execute it from other folder
Result expected	The connection and configuration persist
Test case: TC6.3.3.2	
Description	Connect and configure the system, then restart the device
Result expected	The connection and configuration persist

4.5 System Implementation

4.5.1 Programming Language and Technologies

The implementation of this system is based on a combination of TypeScript (which compiles into JavaScript) and Node.js (JavaScript engine for server-side execution).

Node.js (JavaScript) was chosen for several reasons:

- The speed of development: As with most of the scripting languages, JavaScript development is very fast compared with languages likes Java or C#.
- JSON: JavaScript works perfectly with JSON, which is especially convenient when we are interacting with a JSON REST API. There is no need to create data classes, use libraries, and create boilerplate code.
- Installation and usage: with Node.js's package manager, npm the installation of global commands are as simple as `npm install -g <name>`.
- Performance: Node uses an event-driven, non-blocking I/O (Node.js, 2016), that makes it perform very fast, especially compared with other scripting languages (Zborowski, 2016) (Dotson, 2014).

On the other hand, TypeScript was mainly chosen due to the fact that although it is possible to develop JavaScript applications using the Object Oriented Programming paradigm, JavaScript has no true built-in "classic" Object-Oriented features, as abstraction (McDonnell, 2010). Moreover, as TypeScript is a typed language, it offers error checking at compile time and a better maintainability, without sacrificing the commodity of JavaScript. Finally, as it compiles into plain JavaScript, it is perfectly compatible with Node.js and npm.

- TypeScript version: 2.0.6.
- Node version: 7.0.0.

4.5.2 Coding Style and Standards Followed

For the implementation of this project, we will follow standard TypeScript conventions. In addition to the checks that are automatically performed by the TypeScript compiler, we will perform semantic and syntactic checks using the TSLint[32] TypeScript linter, developed by Palantir.

Along with it, we will use its 'tslint:recommended' set of rules, which are described as (npm, 2016):

> *tslint:recommended is a stable, somewhat opinionated set of rules which we encourage for general TypeScript programming. This configuration follows semver, so it will not have breaking changes across minor or patch releases.*

This set contains seventy-five rules, and is publicly accessible at their Github account[33]. The only modifications that have been applied to them are:

[32] https://palantir.github.io/tslint/

[33] https://github.com/palantir/tslint/blob/master/src/configs/recommended.ts

- "no-console"[34]: false – By default, output to the console is highlighted. However, this configuration is no thought to be applied to node modules. For example, the ESLint linter advises (ESLint, 2016):

  If you're using Node.js, however, console is used to output information to the user and so is not strictly used for debugging purposes. If you are developing for Node.js then you most likely do not want this rule enabled.

- "member-ordering"[35]: "fields-first" – As the "static-first" forces private methods to be the last, which might result confusing in short projects like this.

In order to ensure all the checks are passing before publishing the package, a custom pre-publishing task will be added to the building process.

4.5.3 Tools and Programs Used for Development

4.5.3.1 Version Control System: Git

Git was used as a VCS during the development, in order to back up the code, allow branches with different features and manage different versions of the system, being able to seamlessly revert to previous snapshots of the application.

As a developing tool, Git is not required for building, publishing, installing or executing the system, and the only footprint it has on the project is the '.gitignore' files.

- Version used: 2.8.4

4.5.3.2 Package Management & Building Automation: npm

Version: 3.6.0

4.5.3.2.1 Package manager

In order to install and manage the dependencies of the project, npm has been used. The choice has been straightforward, as npm is the industry *de facto* standard.

4.5.3.2.2 Building automation

In order to automate the build process, two custom tasks have been created:

```
"scripts": {
  "tslint": "tslint -c ./tslint.json ./**/*.ts",
  "prepublish": "tsc && npm run tslint"
}
```

[34] https://palantir.github.io/tslint/rules/no-console/

[35] https://palantir.github.io/tslint/rules/member-ordering/

- `tslint` will execute the TSLinter, reading the custom configuration and checking every TypeScript file.
- `prepublish` is a task which npm executes automatically right after `npm install` has been completed, and just before `npm publish` is executed. The custom task compiles all the TypeScript sources into JavaScript, and then checks them with `tslint`. If anything goes wrong, it will not continue publishing the module.

In addition to these tasks, since TypeScript 2, npm is also in charge of handling type declarations (which allow developers to use existing JavaScript libraries).

4.5.3.2.3 Publishing

npm also acts as a public (and free) repository. Tela CLI has been uploaded to it[36], so that it can be easily installed by any user executing:

```
npm install -g tela-cli
```

The project contains a `.npmignore` file, which determines the files that must be ignored in the publishing project. In this case, all the TypeScript related files are ignored (tslint.json, tsconfig.json, *.ts and *.js.map), so only the final compiled JavaScript files are uploaded.

4.5.3.3 IDE: WebStorm

The main IDE software used during the development of Tela CLI was WebStorm. However, the files created by the IDE have been ignored in Git, so that developers are not biased and are able to choose whatever IDE or editor they consider.

In the case of choosing WebStorm, we should keep in mind that only the latest versions (2016.2.X) work properly with the latest TypeScript 2 capabilities, as the 'readonly' property modifier or the npm type definitions handling.

- Version: WebStorm 2016.2.4 (Build #WS-162.2228.20, built on October 18, 2016)
- JRE: 1.8.0_112-release-287-b2 x86_64
- JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

4.5.4 Dependencies

In this section we will describe the list of dependencies (npm modules) that Tela CLI require.

4.5.4.1 Required Dependencies

These dependencies are required for the execution of the module.

[36] https://www.npmjs.com/package/tela-cli

4.5.4.1.1 Open

This module simplifies the process of opening a URL with a web browser. This is used in order to open the login page of the social networks.Version: 0.0.5

4.5.4.1.2 Request

This module is an HTTP client, which is used to communicate with the Tela Server, and with the API of the social networks to obtain access tokens. Version: 2.72.0

4.5.4.1.3 Node Twitter API

This is a Twitter library that helps in the process of issuing OAuth calls to the Twitter API. It is used to obtain a pair of access token and access token secret. Version: 1.8.0

4.5.4.2 Development Dependencies

These dependencies are not needed for the execution of the tool, but for its building and compilation.

4.5.4.2.1 TypeScript

TypeScript compiler. Version: 2.0.6.

4.5.4.2.2 TSLint

TSLint TypeScript linter. Version: 3.15.1.

4.5.4.2.3 Type definitions

- @types/node: version 6.0.46.
- @types/request: version 0.0.32.
- @types/open: version 0.0.29.

4.5.5 Description of Classes and Interfaces

- **TelaCli**: Main class of the module, exposing all the functionality of the system.
- **bin/cli**: Entry point for the console. It wraps the TelaCli class, reading the input of the console and passing it to the main class.
- **api/TelaApi**: Class in charge of interacting with the Tela Server, executing HTTP requests and returning the results or errors.
- **api/IAction:** Interface representing an action.
- **connection/ConnectionManager**: Class in charge of creating, updating and deleting connections.
- **connection/IConnection**: Interface representing a connection between the CLI tool and a Tela Server. It stores the host, port and access token of the server, and the module configuration properties.
- **modules/Module**: abstract class representing a module and containing common functionality.
- **modules/InstagramModule**: Class in charge of logging the user into Instagram and retrieving an access token.
- **modules/TwitterModule**: Class in charge of logging the user into Twitter and retrieving an access token.

4.5.6 Implementation Details

4.5.6.1 Usage of Interfaces over Classes

Interfaces have been preferred over data classes, as in TypeScript properties are allowed, and interfaces offer more flexibility than classes. For example:

```
export interface IAction {
    module: string;
    name: string;
    description: string;
    params: string[];
}
```

Then, we can directly declare an object as an `IAction`:

```
let responseString = ...
let action: IAction = JSON.parse(responseString);
```

Note that while this is quite convenient, it should only be used in cases when we are sure that the result we are parsing has the expected structure; otherwise, the parsed object might not comply with the interface.

4.5.6.2 Connection

4.5.6.2.1 Connection File

The connection is stored in a file called `.tela`, which will be located in the home directory.

- The file has a JSON format so that it can be directly mapped into a JavaScript object.
- It is saved in the home directory so that it is independent to the folder we execute the system at, and it persists to different installations.

An example of the file is:

```
{
  "host": "tela.herokuapp.com",
  "port": 80,
  "accessToken": "XXXX",
  "configuration": {
    "instagram": {
      "clientId": "XXXX",
      "clientSecret": "XXXX"
    }
  }
}
```

The `ConnectionManager` is in charge of managing this connection files. The filename of the connection file is initiated in the constructor:

```
private static readonly DEFAULT_FILENAME = ".tela";
private filename: string;

constructor(filename?: string) {
    this.filename = path.join(os.homedir(),
            path.basename(filename || ConnectionManager.DEFAULT_FILENAME));
}
```

It keeps a static instance of the current connection, so that the connection file does not have to be read and parsed every time we want to access to it. The process of saving a connection is quite straightforward…

```
private static connection: IConnection;

private save(connection: IConnection) {
    fs.writeFileSync(this.filename, JSON.stringify(connection));
    ConnectionManager.connection = connection;
};
```

… as well as destroying the session…

```
public destroy () {
    ConnectionManager.connection = null;
    if (fs.existsSync(this.filename)) {
        fs.unlinkSync(this.filename);
    }
}
```

… and retrieving it.

```
public get(): IConnection {
    if (!ConnectionManager.connection) {
        if (!fs.existsSync(this.filename)) {
            return {};
        }
        let fileContent = fs.readFileSync(this.filename).toString();
        let connection: IConnection = JSON.parse(fileContent);
        if (!connection.configuration) {
            connection.configuration = {};
        }
        ConnectionManager.connection = connection;
    }
    return ConnectionManager.connection;
}
```

The system is considered to be connected when the host, port and access token are specified:

```
public isConnected(): boolean {
    let connection = this.get();
    return !!connection.host && !!connection.port && !!connection.accessToken;
}
```

4.5.6.2.2 Configuring Modules

Setting and getting module configuration values is also done within the `ConnectionManager`, using the `getProperty()`…

```
public getProperty(module: string, property: string): any {
    let connection = this.get();
    return connection.configuration[module] ?
                connection.configuration[module][property] : null;
}
```

…`setProperty()`…

```
public setProperty(module: string, property: string, value: any) {
    let connection = this.get();
    if (!connection.configuration[module]) {
        connection.configuration[module] = {};
    }
    connection.configuration[module][property] = value;
    this.save(connection);
}
```

… and `existsProperty()`.

```
public existsProperty(module: string, property: string) {
        return !!this.getProperty(module, property);
}
```

4.5.6.2.3 Configuration is Kept Across Sessions

Sessions are created with the `create()` method:

```
public create(host: string, port: number, accessToken: string) {
    let connection = this.get();
    connection.host = host;
    connection.port = port;
    connection.accessToken = accessToken;
    this.save(connection);
}
```

It should be noted that the existing connection is not destroyed and replaced by a new one. Instead, if attributes are updated. This allows configuration properties to persist across different sessions.

4.5.6.3 Modules

4.5.6.3.1 The Module Abstract Class

The abstract class Module defines the two methods and the property every module should have:

```
public abstract readonly name: string;
public abstract isConfigured(): boolean;
public abstract connect(callback: (moduleToken: string) => void): void;
```

In addition to that, it offers two functions that subclasses might use:

```
protected startAuthServer(port: number, callback: (url: Url) => void) {
    let authServer = http.createServer((req: any, res: any) => {
        req.connection.ref();
        let html = "<html>" +
            "<head><script>close();</script></head>" +
            "<body>Login successful! You can now close this window</body>" +
            "</html>";
        res.end(html, () => {
            req.connection.unref();
            authServer.close();
        });
        callback(url.parse(req.url, true));
    });
    authServer.on("connection", (socket: any) => socket.unref()).listen(port);
}

protected openBrowser(url: string) {
    open(url);
}
```

- `startAuthServer()` initiates an HTTP server which will be listening to all the incoming requests. This is helpful so that the external APIs will use callbacks to our machine.
- `openBroser()` opens the supplied url in a browser window (e.g. to send the user to the login page).

4.5.6.3.2 Instagram and Twitter Modules

Both Instagram and Twitter modules inherit from the `Module` abstract class. The steps to obtain their respective access tokens are explained in their official documentation [37] [38], and its implementation can be checked in the source code of both `modules/InstagramModule.ts` and `modules/TwitterModule.ts` files.

[37] https://www.instagram.com/developer/authentication/

[38] https://dev.twitter.com/oauth

Basically, the algorithm in both cases is:

1. A local server starts at the port 8082.
2. A browser window is opened with the login page, with a callback URL to our local server.
3. When the user logs in, he is redirected to our localhost, with an access token.
4. Depending on the authentication process, the token we have just received might have to be sent to the API to exchange it for the real access token.
5. The access token is retrieved.
6. The local server is stopped.

4.5.6.4 Tela CLI Wrapper

The CLI wrapper first instantiates a `TelaCli`, and prepares the console arguments:

```
let telaCli = new TelaCli();
let args = process.argv.slice(2);
let command = args[0];
if (!command) {
    console.error("No argument supplied!");
    telaCli.printCommands();
} else {
    execute(telaCli, command, args.slice(1));
}
```

Then, it calls the `execute()` method, which is basically a switch mapping the command name to a `TelaCli` method:

```
switch (command.toLowerCase()) {
    case "--help":
    case "-h":
        telaCli.printCommands();
        break;
    case "connect":
        telaCli.connect(args[0], args[1]);
        break;
    ...
    default:
        console.error("Command not found");
        telaCli.printCommands();
}
```

4.5.7 Test Results

- After each iteration, the outcome of them was manually tested, ensuring that the developed functionality met the expectations and requirements.
- During this test, some problems were found, both in this system and in Tela. If they were in this system, the bugs were replicated and analyzing and fixed. If they were a problem of Tela, the bug was not only fixed but the associated test case, which should have detected it automatically.
- At the end of its development, the thirty-four test cases were manually performed, ensuring that the final version met the expectations and was completely ready to use.

4.5.8 Problems Found

4.5.8.1 Change of the Chosen Programming Language

At the beginning, this system was designed and started to be implemented using Java (and the Commons-CLI library), so that common libraries, classes, and responses could be shared with the Tela Server. However, this had some drawbacks:

- The CLI tool was too tied to the server. Either this tool was included within the same system, or a common library had to be extracted from the server.
- The installation of a Java tool as a global command is not as straightforward as it is with npm.
- A lot of boilerplate code and data classes were needed, especially for unwrapping the JSON responses. Although this might have its advantages, for our purpose (i.e. basic handling of the responses and outputting the information) was too time-consuming.

Therefore, the whole system was thrown away and started again using JavaScript.

4.5.8.2 Obtaining Auth Credentials from the Twitter API

Although the Instagram API documentation has some issues, its authentication is based on Bearer tokens, which are quite simple to obtain and use.

However, this is not the case with the Twitter API, where security and authentication is a major concern. The process is based on OAuth 1.0, which means that:

- The process of obtaining an access token is way more tedious.
- The request must be signed so that it cannot be tampered.
- It is not enough to send a Bearer access token to the server: there are four values needed, the API key, API secret, access token and access token secret.

In order to implement this process, we ended up using the unofficial Node Twitter API, and the modifications already explained in the previous system, as the composed module token with the structure:

```
<api_key>:<api_secret>:<token>:<token_secret>
```

4.6 System Manuals

The following manuals and can be found (in Markdown format) at `/tela-cli/README.md`.

4.6.1 Installation

4.6.1.1 npm

Tela CLI is a Node application and can be installed directly from npm[39]:

```
npm install -g tela-cli
```

Then, you should be able to execute:

```
tela-cli --help
```

4.6.1.2 Manual Installation

Assuming that we are inside the folder of the application, we just have to install the project with npm:

```
npm install
```

Then, we are ready to use the CLI tool:

```
node bin/cli --help
```

4.6.2 Connection to a Tela Server

4.6.2.1 Connection

In order to create a connection with our Tela Server, execute:

```
tela-cli connect <host> [<port>]
```

[39] https://www.npmjs.com/package/tela-cli

For example, using the public server:

```
tela-cli connect tela.herokuapp.com
```

This command will create a session in the server, and write it into a `.tela` file, which stores the details of the connection.

4.6.2.2 Disconnection

It is not necessary to explicitly disconnect from the Tela Server. However, if you want to, it can be done executing:

```
tela-cli disconnect
```

Or by manually removing the `.tela` file.

4.6.3 Executing Actions

Actions can be executed with the `execute` command:

```
tela-cli execute <module> <action> [<param1>=<value1> [<param2>=<value2> ...]]
```

The result is printed into the system output, encoded as JSON. This allows to use the Tela CLI along with console commands. For example, save the results into a file:

```
tela-cli execute <module> <action> > result.txt
```

Or format the JSON output:

```
tela-cli execute <module> <action> | jq
```

4.6.4 Scheduling

4.6.4.1 Schedule an Action

In order to schedule an action, execute the `schedule` command, followed by the delay of execution (in seconds) and the same arguments than the `execute` command:

```
tela-cli schedule <delay> <module> <action> [<param1>=<value1> [<param2>=<value2> ...]]
```

The information of the scheduled task, as well as the result of its execution, will be outputted to the console.

4.6.4.2 Get the Scheduled Actions

It is possible to retrieve all the actions that the authorized user has scheduled, by running:

```
tela-cli scheduled
```

4.6.4.3 Cancel Scheduled Action(s)

In order to stop the scheduled execution of an action using the `cancel` command:

```
tela-cli cancel (<scheduled_action_id> | all)
```

If an ID is provided, only that action will be cancelled. In case `all` is supplied, all the actions scheduled by the authorized user will be cancelled.

4.6.5 Obtaining Help

To obtain all the available modules and actions configured in the Tela Server we are connected to, execute:

```
tela-cli help
```

Or, just for a certain module:

```
tela-cli help <module>
```

Note: the `help` command returns all the modules and actions configured within the Tela Server. Some of them might not be supported in this CLI tool.

4.6.6 Working with Modules

4.6.6.1 Configuring Modules

Modules might have specific properties which can be configured, by executing:

```
tela-cli configure <module> <property> <value>
```

4.6.6.2 Linking & Unlinking Modules

Linking a module is the process of obtaining an access token for it, and storing it in our Tela session. This process is only needed if the module requires a token.

```
tela-cli link <module>
```

On the other hand, the `unlink` command deletes the module token from the session:

```
tela-cli unlink <module>
```

4.6.6.3 Supported Modules

4.6.6.3.1 Instagram

The Instagram module requires the configuration properties `clientId` and `clientSecret`, which can be obtained creating an Instagram Application[40] with the redirect URI `http://127.0.0.1:8082`.

4.6.6.3.1.1 Setting up

```
tela-cli connect <host> <port>
tela-cli configure instagram clientId <client_id>
tela-cli configure instagram clientSecret <client_secret>
tela-cli link instagram
```

4.6.6.3.1.2 Executing actions

Once the Instagram module has been set up, you can execute actions with:

```
tela-cli execute instagram <action> [<param1>=<value1> [<param2>=<value2> ...]]
```

[40] https://www.instagram.com/developer/clients/manage/

For example:

```
# Obtain info about the logged user
tela-cli execute instagram self

# Search a user
tela-cli execute instagram user username=themedizine
```

4.6.6.3.2 Twitter

The Instagram module requires the configuration properties `apiKey` and `apiSecret`, which can be obtained creating a Twitter Application[41], with the redirect URI `http://127.0.0.1:8082`.

4.6.6.3.2.1 Setting Up

```
tela-cli connect <host> <port>
tela-cli configure twitter apiKey <api_key>
tela-cli configure twitter apiSecret <api_secret>
tela-cli link twitter
```

4.6.6.3.2.2 Executing Actions

Once the Twitter module has been set up, you can execute actions with:

```
tela-cli execute twitter <action> [<param1>=<value1> [<param2>=<value2> ...]]
```

For example:

```
# Obtain info about the logged user
tela-cli execute twitter self
```

[41] https://apps.twitter.com/

4.7 System Outcome and Extensions

4.7.1 System Outcome

The outcome of this system is a complete functional system, which fulfils all our requirements. The final version of the tool is published in npm, and is ready to use.

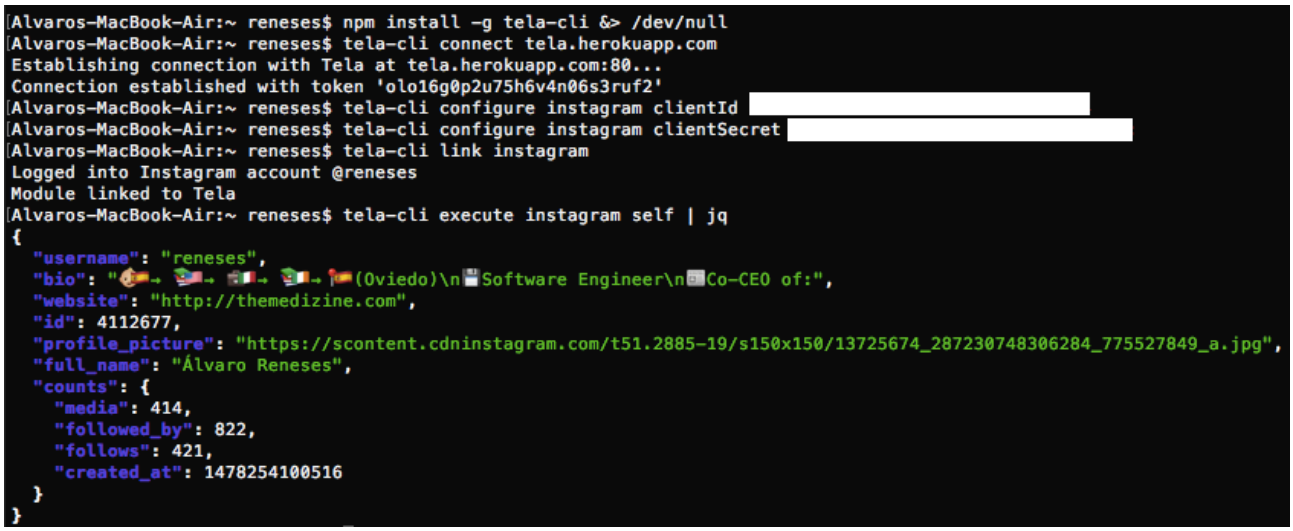


Figure 4.4 Tela CLI Running Screenshot

4.7.2 Extensions

Although the system is completely functional, it is not tested enough for enterprise applications with production environments. For it to reach this point, complete automated tests should be designed, and error handling of corner cases should be further implemented.

4.8 Content delivered

The source code of the project can be found at the `tela-cli` folder. Within it, we find:

Folder: /tela-cli	
Directories	
Folder	**Contents**
/api	Functionality regarding the interaction with the Tela Server
/bin	Entry point for the cli utility
/connection	Functionality regarding the management of connections with a Tela Server.
/modules	Functionality modules
/node_modules	Node modules (dependencies) installed by npm. Note: this folder only exists after 'npm install' has been executed
Files	
File	**Description**
.gitignore	Content ignored by Git
.npmignore	Content ignored by npm (when publishing the module)
.tela	Application data. Note: this file only exists when a connection has been created.
package.json	npm main file
README.md	Instructions for installation and execution
TelaCli.ts	Main script
tsconfig.json	Typescript compiler configuration
tslint.json	TSLint configuration

5 System III: Tela Who?

5.1 Introduction

5.1.1 Description of the System

Tela Who? is a web application for companies, brands, and regular users, that allows them to obtain a better understanding about how a user is connected with them, so that they can improve their relationship and target similar users.

5.1.2 Scope

The main purpose of this system is to act as a proof of concept of a web application that interacts with a Tela Server, at the same time that offers a solution addressing the needs of real end users

5.1.3 Development Process

As we have chosen to follow an iterative agile approach based on sprints, we will develop Tela Who? in the same way. However, as it is fairly simple application with well-defined requirements, we will try to complete it in a small number of sprints.

5.2 Analysis of the System

5.2.1 Analysis of Use Cases

Users will interact with Tela Who? to obtain a better understanding about how they are connected to users, which includes:

- Log in and log out using an Instagram account
- List the followers, people the user is following, and friends, seeing its basic user info.
- Search a user by its username, showing all its complete information.
- Show the relationship of a user with the logged user.
- Figure out how a user has interacted with the latest media.

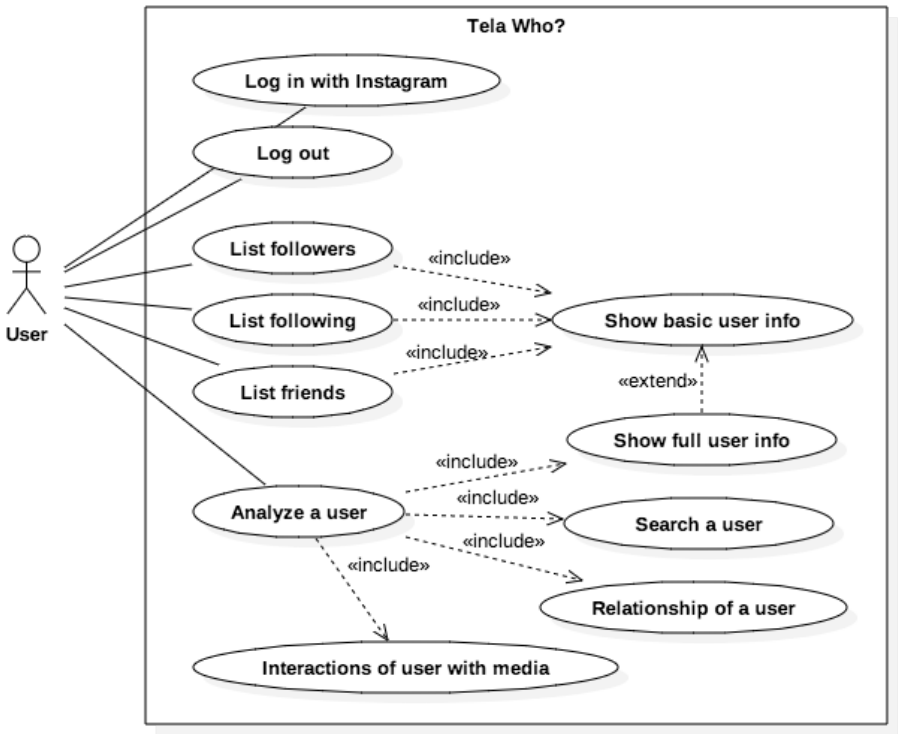


Figure 5.1 Tela Who? Use Cases Diagram

5.2.2 Specification of System Requirements

From the use cases identified in the previous section, we will elaborate a list of functional and non-functional requirements the system should meet. This will be the base for the design phase, and our focus during testing.

5.2.2.1 Functional Requirements

ID	Name	Description
FR1	Authentication	
FR1.1	Log in	The user can log in with his Instagram account, and his user and name are displayed.
FR1.2	Log out	The user can log out the system.
FR2	List users	
FR2.1	List followers	Show the username, name and picture of the users following the logged user.
FR2.2	List following	Show the username, name and picture of the users the logged user is following.
FR2.3	List friends	Show the username, name and picture of the users that are both following the logged user, and being followed by him.
FR3	User analysis	
FR3.1	Analyze a user	Show the information of a user, along with how it is connected to the logged user and how he has interacted with the latest media of the logged user.

5.2.2.2 Non-Functional Requirements

ID	Description
NFR1.1	The website deployment will be easy to perform.
NFR1.2	The website should not depend on the operating system of the web server.
NFR1.3	The application will be based on Angular, all the logic and operations are done in the client side.

5.2.3 Design of Use Case Scenarios

Due to the limited scope of the following use case scenarios will omit the alternate flows.

5.2.3.1 Authentication (FR1)

Log in (FR1.1)	
Description	Log in with his Instagram account.
Primary Actors	User, Instagram & Tela.
Preconditions	The user has an Instagram account.
Post conditions	The system stores the access token of the Tela Session.
Trigger	The user clicks the log in button.
Flow	1. The user is redirected to the Instagram login page. 2. The user logs into Instagram. 3. Instagram redirects the user to the system, along with an Instagram access token. 4. The system sends a request to Tela to create a session with the Instagram access token. 5. Tela sends the access token of a new session back to the system. 6. The system stores the access token and redirects the user to the dashboard.

Log out (FR1.2)	
Description	Log out from the system.
Primary Actor	User & Server.
Preconditions	The user is logged in.
Post conditions	The connection is destroyed, in both the server and the local machine.
Trigger	The user clicks the log out button.
Flow	1. The system sends a request to the server to delete the session. 2. The server deletes the session and answers with a success message. 3. The system deletes the connection. 4. The user is redirected to the log in page.

5.2.3.2 List Users (FR2)

List followers (FR2.1)	
Description	Show the followers of the logged user.
Primary Actors	User & Server.
Preconditions	The user is logged in.
Post conditions	-
Trigger	The user clicks the 'followers' button.
Flow	1. The system sends a request to Tela to obtain the followers. 2. Tela sends the followers back. 3. The followers are rendered.

List following (FR2.2)	
Description	Show the users the logged user is following.
Primary Actors	User & Server.
Preconditions	The user is logged in.
Post conditions	-
Trigger	The user clicks the 'following' button.
Flow	1. The system sends a request to Tela to obtain the following. 2. Tela sends the following back. 3. The following are rendered.

List friends (FR2.3)	
Description	Show the friends of the logged user.
Primary Actors	User & Server.
Preconditions	The user is logged in.
Post conditions	-
Trigger	The user clicks the 'friends button.
Flow	1. The system sends a request to Tela to obtain the friends. 2. Tela sends the friends back. 3. The friends are rendered.

5.2.3.3 User Analysis (FR3)

Analyze a user (FR3.1)	
Description	Show the information of a user, how it is connected to the logged user and how he has interacted with his latest media.
Primary Actors	User & Server.
Preconditions	The user is logged in.
Post conditions	-
Trigger	The user inputs a username and clicks search.
Flow	1. The system sends a request to Tela to obtain the user with the given username. 2. Tela sends the information of the user back. 3. The information is rendered. 4. The system sends a request to Tela to obtain the relationship with the logged user. 5. Tela sends the information of the relationship back. 6. The information is rendered. 7. The system sends a request to Tela to obtain the latest media of the logged user, along with the likes of those. 8. Tela sends the information back. 9. The media is filtered to those that the searched user has liked. 10. The media is shown.

5.2.4 Analysis of Components

Although it will be further explained in the implementation section, this system will be based in Angular. Angular applications are composed of special classes called components (Angular, 2016), which will be the base of our analysis and design.

This system will have two main components:

- The `login` component, which will be responsible to log the user into Instagram and create a Tela connection.
- The `who` component, which is the main one. It will be composed by a `toolbar` component, and a `dashboard` component.

5.3 Design

5.3.1 Design of Wireframes of the Visual Interface

First, this will be the login view:

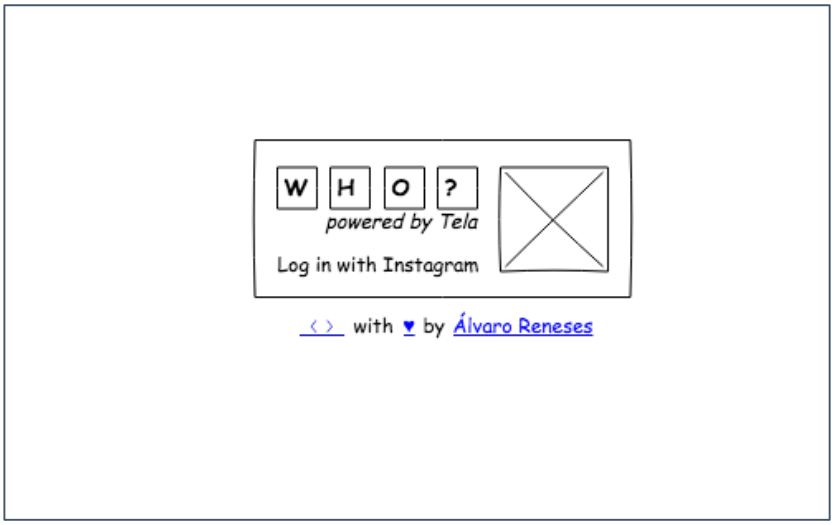


Figure 5.2 Tela Who? Login Wireframe

Once the user clicks on Log in, he will be redirected to Instagram. Then, after authentication, the user will be redirected back to the dashboard:

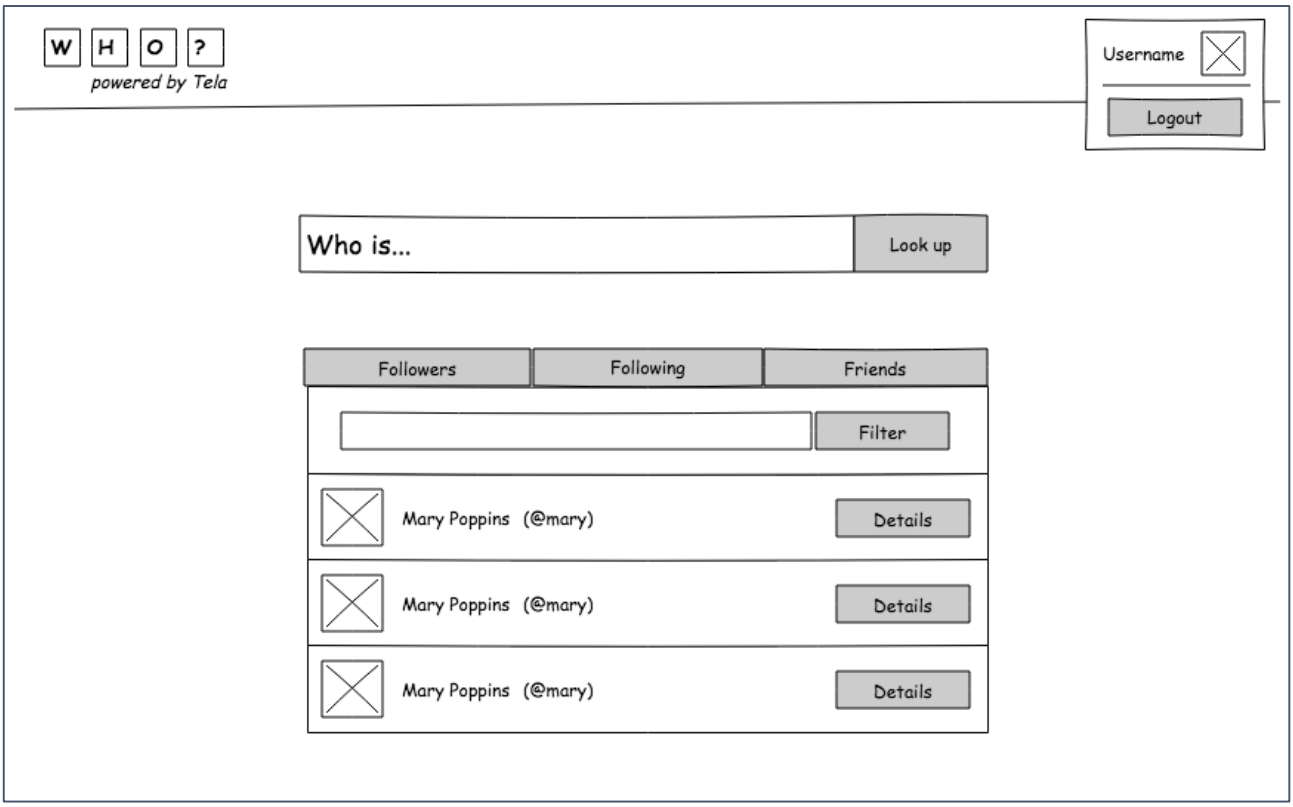


Figure 5.3 Tela Who? Dashboard Wireframe

- If the user hovers or clicks on the username (in the toolbar), a logout button will appear.
- The tab of buttons “Followers”, “Following” and “Friends” will refresh the list of users displayed.
- The results of these buttons can be filtered.

Then, if we write a username and click “Loop up”, the user will be searched and analyzed:

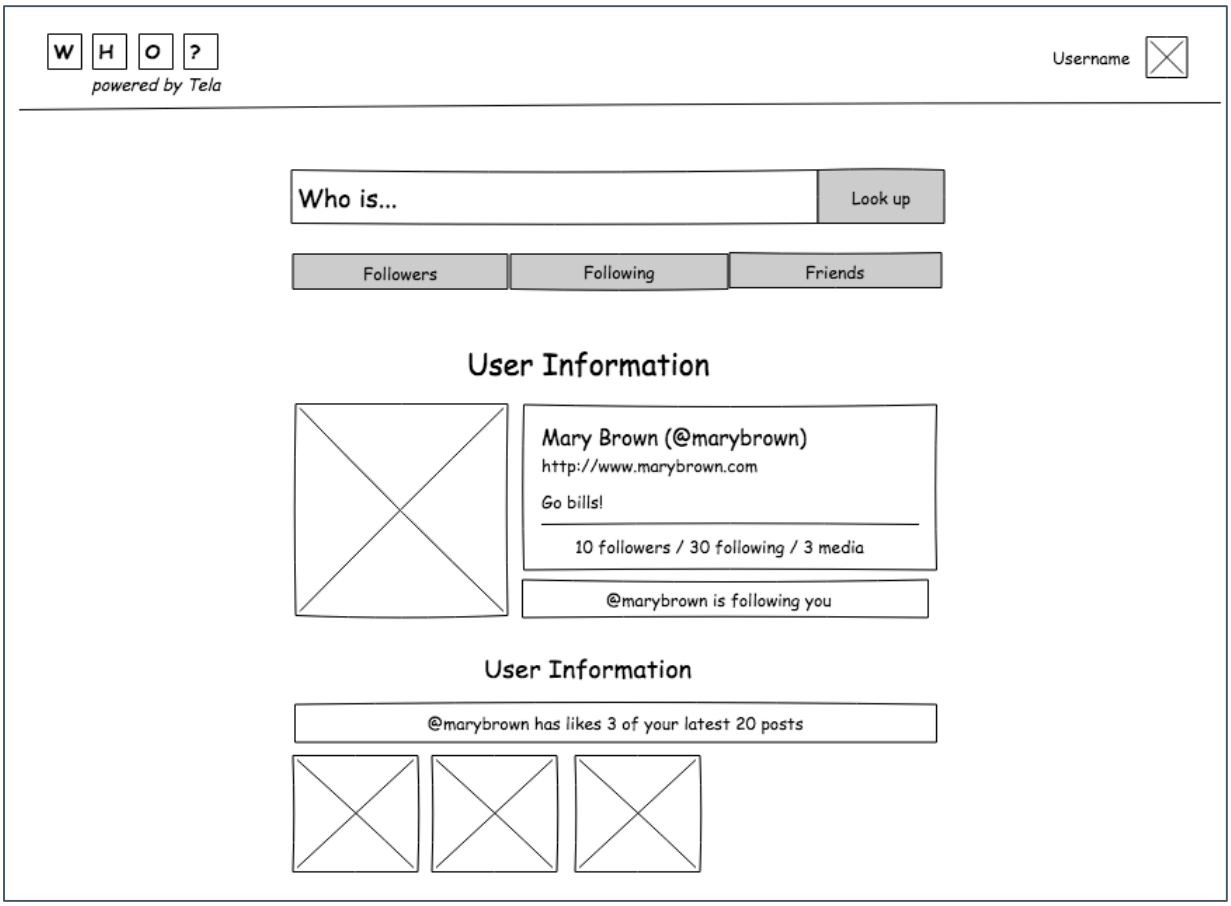


Figure 5.4 Tela Who? Analysis Wireframe

In this windows, we observe:

- The picture and information of the user is displayed.
- A panel shows the relationship of the user with us.
- The systems shows the posts the user has liked among our latest media.

5.3.2 Design of Components

Based on the previous wireframes, we will identify all the components we could decompose the application in:

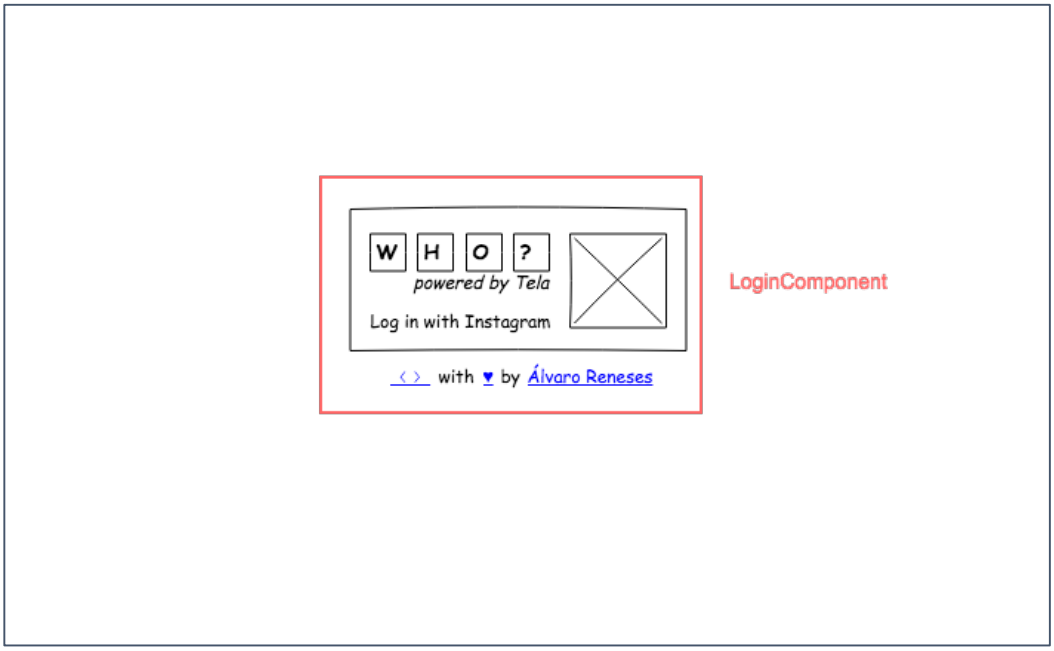


Figure 5.5 Tela Who? Login Components

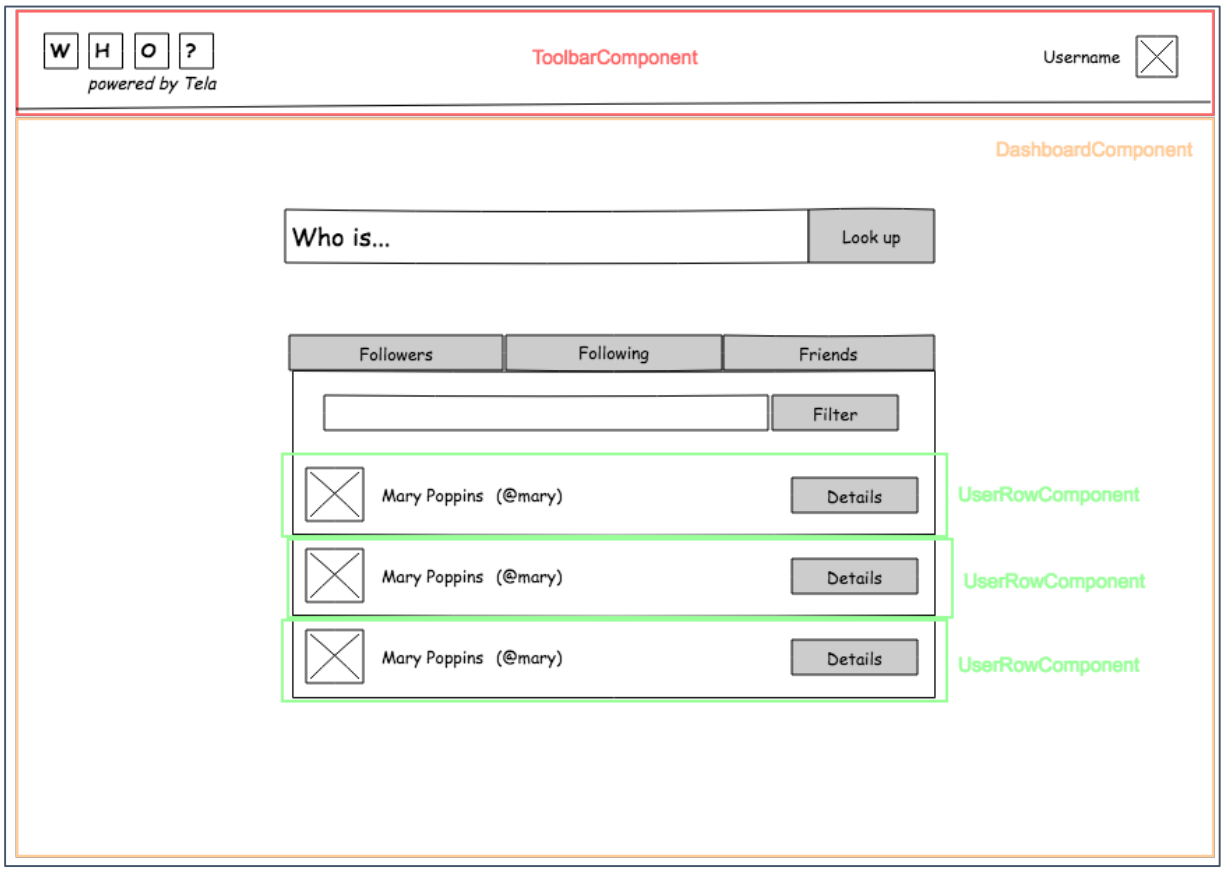


Figure 5.6 Tela Who? Dashboard Components

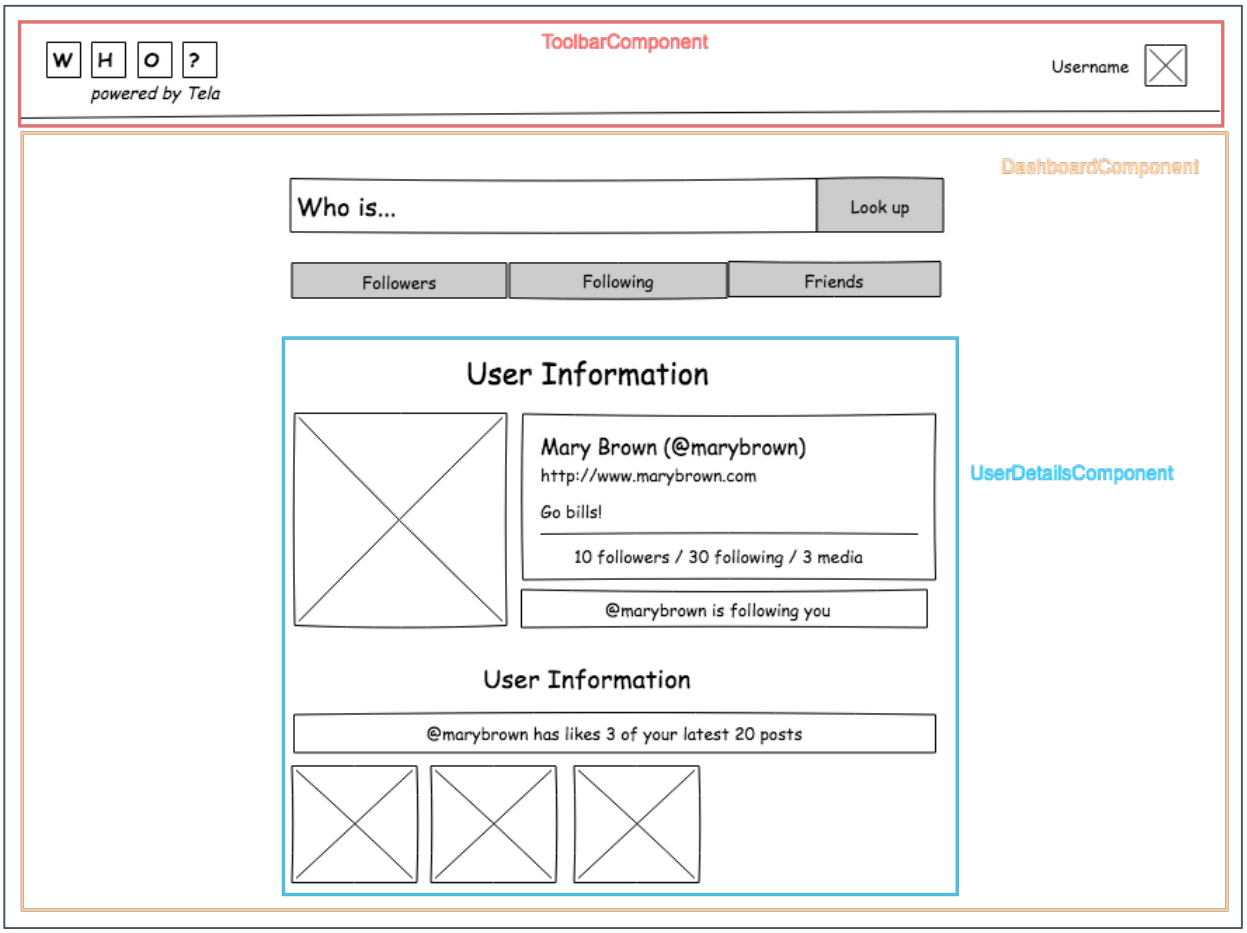


Figure 5.7 Tela Who? Analysis Components

These components could be represented as a component tree, so that we obtain a better view of how each component depends on the rest:

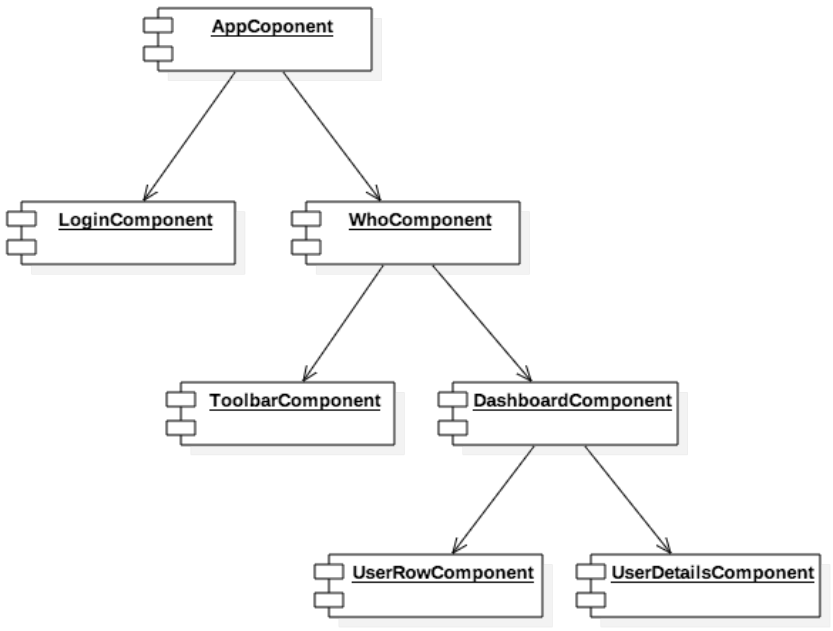


Figure 5.8 Tela Who? Component Diagram

5.4 Test Plan

5.4.1 Unit Tests

Due to the small size of the system, and the limited scoped of it as it acts as a proof of concept for a client application interacting with the Tela Server, unit tests have been omitted.

5.4.2 Integration Testing

Testing scripts will be designed outlining a sequence of tasks to perform in order to check the compliance with the functional and non-functional requirements:

Requirement FR1: Authentication	
Test case: TC1.1.1	
Description	The user logs in with his Instagram Account
Result expected	The user is redirected to the dashboard, and his username and image are displayed in the toolbar
Test case: TC1.1.2	
Description	Once logged, the user logs out
Result expected	The session is destroyed and the user is redirected to the login page
Requirement FR2: List users	
Test case: TC2.1.1	
Description	The user clicks on ‘followers’
Result expected	The followers of the user are displayed
Test case: TC2.1.2	
Description	The user clicks on ‘following’
Result expected	The following of the user are displayed
Test case: TC2.1.3	
Description	The user clicks on ‘friends’
Result expected	The friends of the user are displayed
Test case: TC2.1.4	
Description	The user clicks on ‘followers’, then on ‘following’, then on ‘friends’
Result expected	The results are displayed as displayed
Test case: TC2.1.5	
Description	The user clicks on ‘followers’, and then clicks on the ‘details’ button of any user
Result expected	The user is redirected to the analysis window

Requirement FR3: Analysis	
Test case: TC3.1.1	
Description	The user clicks on 'followers', and then clicks on the 'details' button of any user
Result expected	The user is redirected to the analysis window with the user clicked
Test case: TC3.1.2	
Description	The user searches a username
Result expected	The user is redirected to the analysis window with the user searched
Test case: TC3.2.1	
Description	The user searches a private user
Result expected	Message altering that the user is private is shown
Test case: TC3.2.2	
Description	The user analyzes a user
Result expected	The relationship message is true
Test case: TC3.2.3	
Description	The user analyzes a user
Result expected	The latest images liked by the analyzed are the expected ones

5.4.3 Usability Tests

Usability tests are an essential part of software development, as they allow developers to receive feedback about the final solution and observe if the system actually meets the needs of the users. These tests often include surveys, questionnaires, and interviews with real users in order to gather the results.

As the outcome of this system will be a functional prototype, we should realize prototype evaluations, which are based on (UsabilityNet, 2006):

- Selecting the most important parts of our system.
- Selecting 3 to 5 users.
- One by one, ask each user to complete the tasks.
- Observe how they do it, without giving hints or instructions.
- If a user is not capable of performing the task, or he is confused, we find a usability problem.
- Once each task is completed, ask the user about the problems he had during the process.

The tasks that we would choose are:

- Log into the system with Instagram, and the log out.
- Check the users you are following, and the ones that are following you.
- Analyze how a user is connected with you.
- See the friends of the user, and then the details of any of them.

5.5 System Implementation

5.5.1 Programming Language and Technologies

The implementation of this system is based on JavaScript. Tela Who? is a client-side application based on Angular 2[42] and TypeScript. Furthermore, the application is pure client-side: any server-side programming or database have been used.

Regarding the server, any solution capable of sending static files is adequate (e.g., Node, Flask, NGINX, Apache, etc.). An embedded Node server has been included, but developers are not forced to use it.

- Angular version: 2.0.0
- TypeScript version: 2.0.6.
- Node version: 7.0.0.

5.5.2 Coding Style and Standards Followed

This application follows the Angular style guide, which suggests and explains the preferred conventions for file structure, coding conventions, naming, structure and so on[43].

5.5.3 Tools and Programs Used for Development

5.5.3.1 Version Control System: Git

Git was used as a VCS during the development, in order to back up the code, allow branches with different features and manage different versions of the system, being able to seamlessly revert to previous snapshots of the application.

As a developing tool, Git is not required for building, publishing, installing or executing the system, and the only footprint it has on the project is the '.gitignore' files.

- Version used: 2.8.4

5.5.3.2 Package Management & Building Automation: npm

Version: 3.6.0

5.5.3.2.1 Package manager

In order to install and manage the dependencies of the project, npm has been used. The choice has been straightforward, as npm is the industry *de facto* standard.

[42] https://angular.io/

[43] https://angular.io/styleguide

5.5.3.2.2 Building automation

In order to automate the build and execution process, some custom tasks have been created:

```
"scripts": {
  "start": "node server",
  "dev": "webpack-dev-server --colors --inline --progress --port 3000",
  "build": "rimraf dist && webpack --config config/webpack.prod.js --progress --profile
--bail",
  "postinstall": "npm run build"
}
```

- `start` will start the built-in Node server (`server.js` file)
- `dev` will start the `webpack-dev-server` at the port 3000, watching for file changes and re-compiling the application.
- `build` will use webpack to build a production version of the application at the `dist/` folder.
- `postinstall` will make sure that after installing the packages, the `build` task is executed.

In addition to these tasks, since TypeScript 2, npm is also in charge of handling type declarations (which allow developers to use existing JavaScript libraries).

5.5.3.3 Webpack

Webpack is a module bundler, which takes modules with dependencies and generates static assets (Webpack, 2016); or what is the same: a web compiler, which takes the source files and merges them into a smaller number of files, performing different tasks as their optimization. Version: 1.13.0.

5.5.3.4 Semantic

Semantic defines itself as (Semantic UI, 2016):

> *Semantic is a development framework that helps create beautiful, responsive layouts using human-friendly HTML.*

Semantic has been used as HTML and CSS framework. Version: 2.2.4.

5.5.3.5 IDE: WebStorm

The main IDE software used during the development of Tela CLI was WebStorm. However, the files created by the IDE have been ignored in Git, so that developers are not biased and are able to choose whatever IDE or editor they consider.

In the case of choosing WebStorm, we should keep in mind that only the latest versions (2016.2.X) work properly with the latest TypeScript 2 capabilities, as the 'readonly' property modifier or the npm type definitions handling.

- Version: WebStorm 2016.2.4 (Build #WS-162.2228.20, built on October 18, 2016)
- JRE: 1.8.0_112-release-287-b2 x86_64
- JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

5.5.4 Dependencies

The only external dependency of Tela Who? is dotenv. Other than that, the remaining dependencies included in the `package.json` of `npm` are those from Angular 2, which can be consulted at their official documentation[44].

5.5.4.1 dotenv

This module read a `.env` file, parses it and assigns the variables found as environmental variables accessible from `process.env`. Version: 2.2.0.

5.5.5 Visual Interface

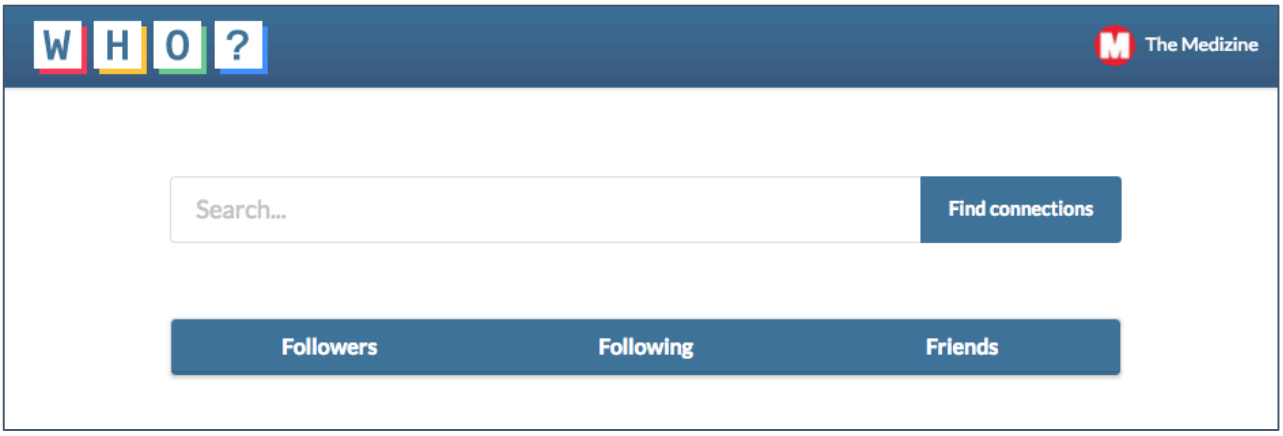
In this section we will include screenshots of how the final visual interface looks.

First, the login page:

Figure 5.9 Tela Who? Login Page

[44] https://angular.io/docs/ts/latest/guide/npm-packages.html

Once we are connected, we are redirected to the dashboard:



If we hover our username, the logout menu is displayed:

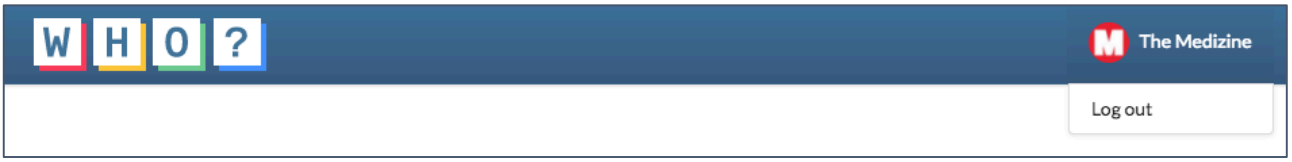


Figure 5.10 Tela Who? Dashboard Page – Logout

Clicking in any of the relationship buttons will show the corresponding users:

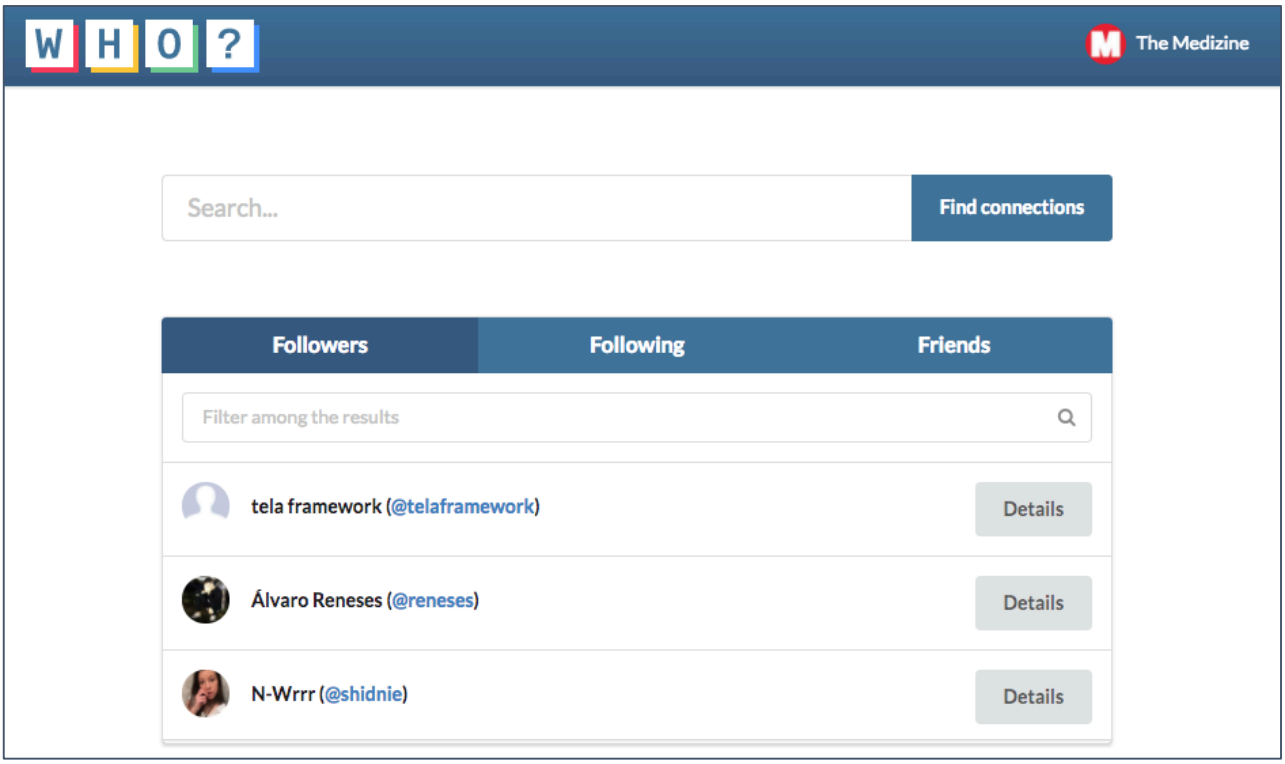


Figure 5.11 Tela Who? Dashboard Page – Followers

The results can be filtered by typing in the filter input:

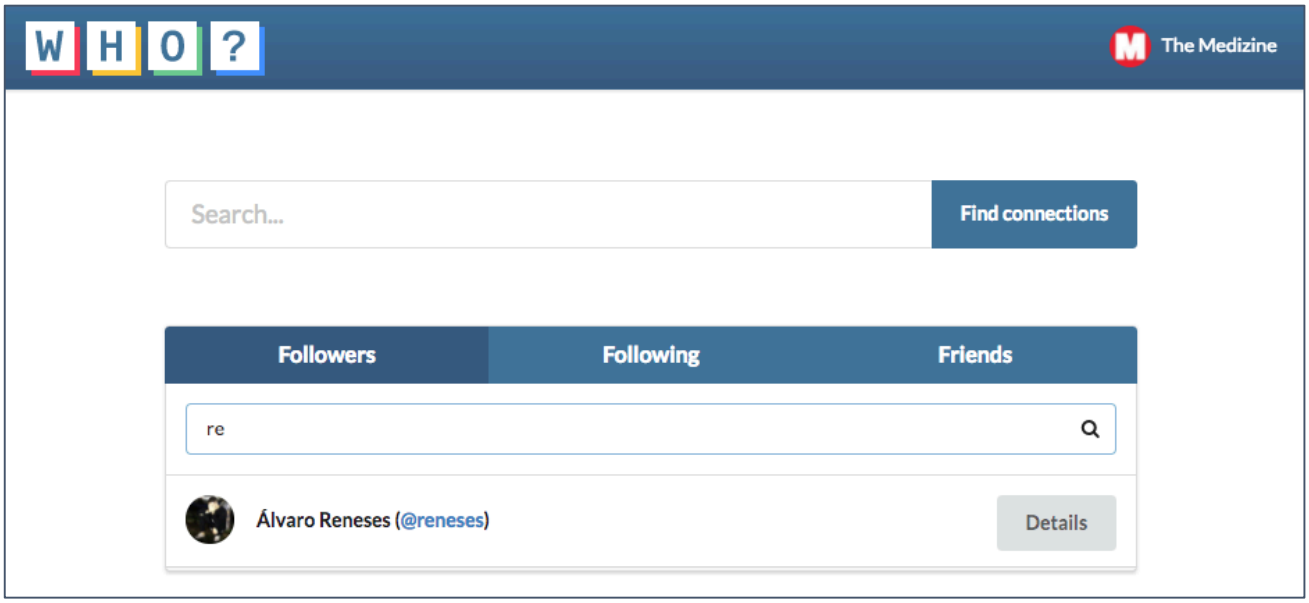


Figure 5.12 Tela Who? Dashboard Page

The page is completely response, in order to offer an enhanced user experienced in mobile devices:

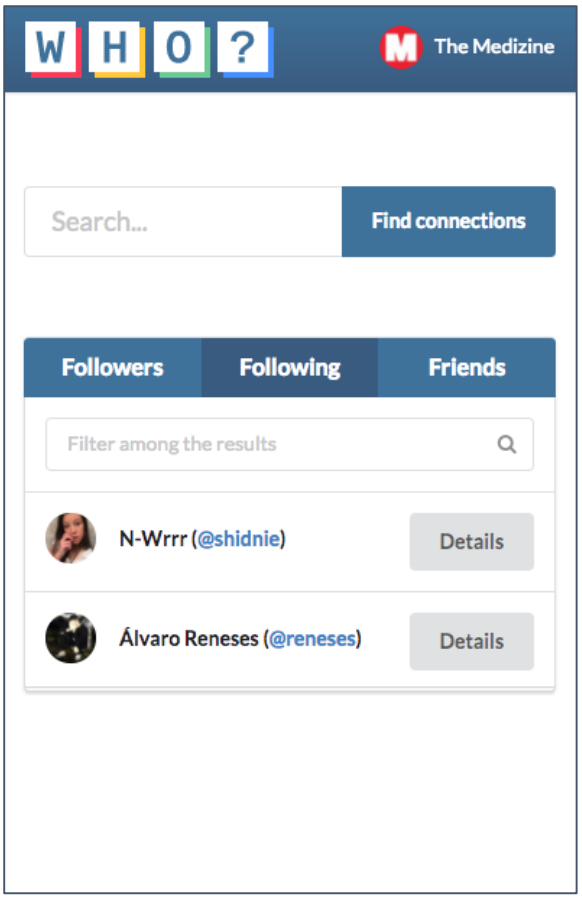


Figure 5.13 Tela Who? Dashboard in Mobile Devices

On the other hand, the analysis page look as following:

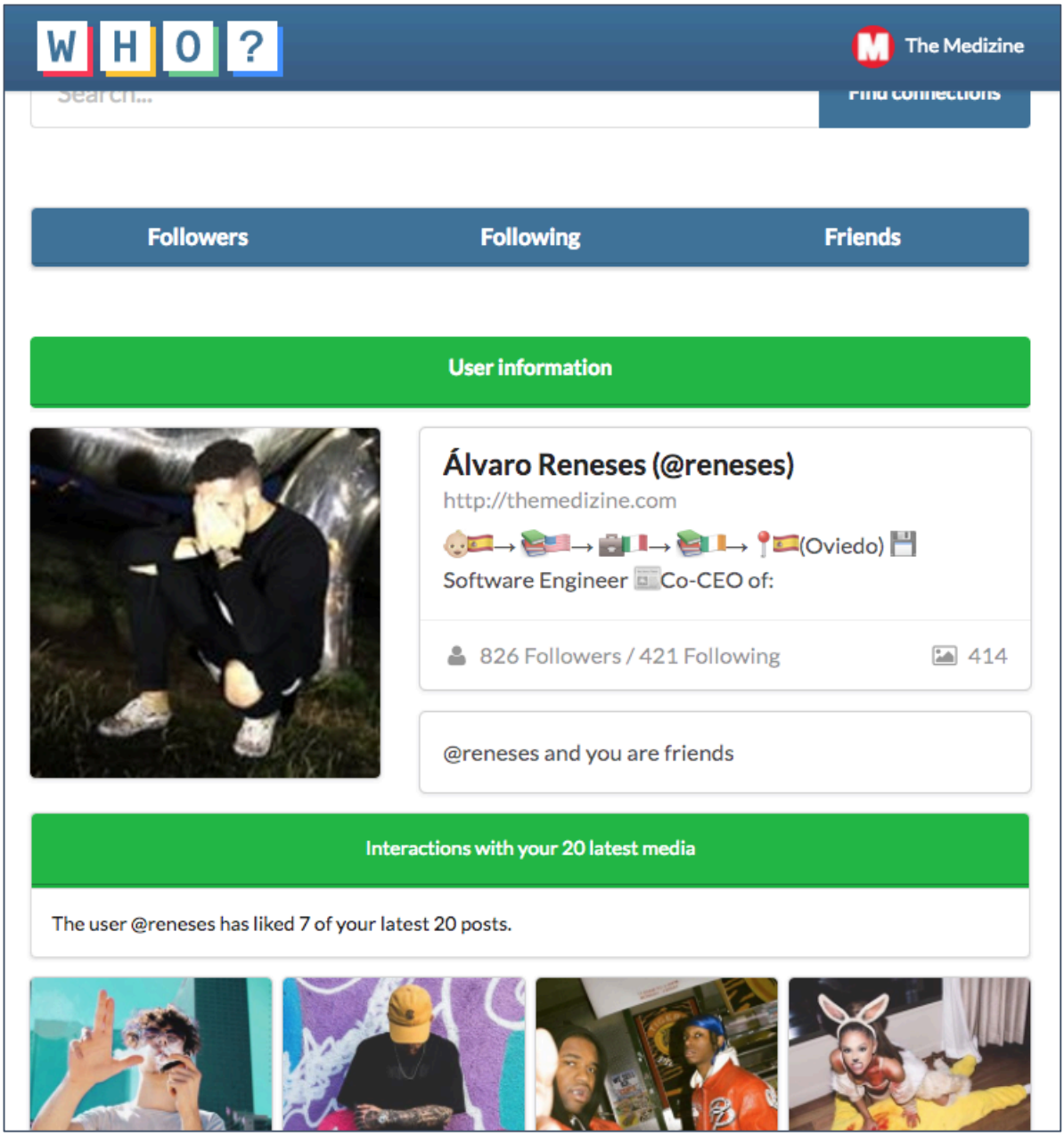


Figure 5.14 Tela Who? Analysis Page

If we try to analyse a private user, the following message is shown:

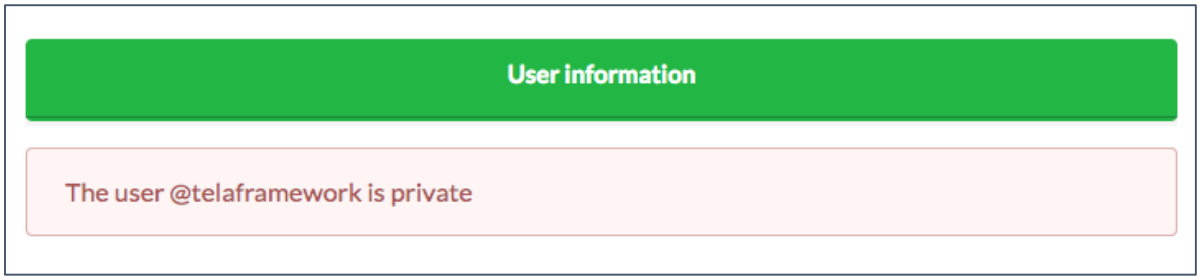


Figure 5.15 Tela Who? Analysis Page - Private User

5.5.6 Description of Components

- **AppComponent**: Main component, contains the router outlet which will select the corresponding view.
- **login/LoginComponent**: Login view.
- **login/AuthComponent:** Receives the callback from Instagram, extracts the access token, creates the session and redirects the user to the dashboard.
- **who/WhoComponent**: Dashboard wrapper, containing the `ToolbarComponent` and the `DashboardComponent`.
- **who/toolbar/ToolbarComponent**: Toolbar in the top of the application.
- **who/dashboard/DashboardComponent**: Dashboard with the main functionality.
- **who/dashboard/UserDetailsComponent**: User analysis.
- **who/dashboard/UserRowComponent**: A user row in the interface.
- **services/AuthService**: Service in charge of authentication management.
- **services/TelaService**: Service in charge of interacting with the Tela server.

5.5.7 Implementation Details

5.5.7.1 Node Server

As it has already been commented, the system includes a basic Node server whose only responsibility is to send the files of the application, which will be executed on the client side.

```
var express = require('express');
var path = require('path');
var app = express();

var publicPath = path.resolve(__dirname, 'dist');
app.use(express.static(publicPath));
app.all('*', function (req, res) {
    res.status(200).sendFile(path.join(publicPath, 'index.html'));
});

var port = process.env.PORT || 3000;
app.listen(port, function () {
    console.log('Server running on port ' + port);
});
```

It read the port from the `PORT` environmental variable, and redirects all the requests to the `dist/` folder where Webpack compiles the application.

The rest of calls are redirected to the `index.html` file, which is quite simple:

```
<!DOCTYPE html>
<html>
<head>
    <base href="/">
    <title>Tela Who?</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="https://fonts.googleapis.com/css?family=Cousine" rel="stylesheet">
</head>
<body>
<app>
    <div class="full">
        <div class="ui active inverted dimmer">
            <div class="ui big text loader"></div>
        </div>
    </div>
</app>
</body>
</html>
```

- The links to the compiled JavaScript are added by Webpack.
- The view only shows a loading image, which then will be replaced by the application when Angular is ready (the `<app></app>` tags).

5.5.7.2 Main Components

The main component `AppComponent`, which will be bind to the app tag, only contains the `router-outlet`:

```
@Component({
    selector: 'app',
    template: '<router-outlet></router-outlet>'
})
export class AppComponent { }
```

The `router-outlet` will include the corresponding routes, as configured in the `app.routing.ts` file:

```
const appRoutes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full'},
  { path: 'dashboard', children: [
    { path: 'user', children: [
      { path: ':username', component: WhoComponent},
      { path: '', redirectTo: '/dashboard', pathMatch: 'full'},
    ]},
      { path: '', component: WhoComponent},
  ]},
  { path: 'login', component: LoginComponent},
  { path: 'auth', component: AuthComponent}
];
```

On the other hand, the `app.module.ts` file prepares the application module registering all the imports, declarations and providers, and bootstrapping the `AppComponent` as the main component.

```
@NgModule({
    imports: [
        BrowserModule,
        HttpModule,
        routing
    ],
    declarations: [
        AppComponent,
        WhoComponent,
        ToolbarComponent,
        DashboardComponent,
        UserRowComponent,
        UserDetailsComponent,
        LoginComponent,
        AuthComponent,
    ],
    providers: [
        appRoutingProviders,
        AuthService,
        TelaService,
        Config
    ],
    bootstrap: [AppComponent]
})
export class AppModule {}
```

5.5.7.3 Configuration

The configuration service, `Config`, reads the values that have been configured as environment variables:

```
Injectable()
export class Config {
    ...
    constructor() {
        this.tela = {
            server: process.env.TELA_SERVER
        };
        this.instagram = {
            redirectHost: process.env.INSTAGRAM_REDIRECT_HOST,
            clientId: process.env.INSTAGRAM_CLIENT_ID
        };
    }
}
```

However, how can those environment variables be injected into the client application? The answer is via dotenv, which reads the configuration from the `.env` file, and Webpack, which defines it:

```
require('dotenv').config({silent: true});
const ENV = process.env.NODE_ENV = process.env.ENV = 'dev';
...
module.exports = webpackMerge(commonConfig, {
     ...
     plugins: [
        ...
        new webpack.DefinePlugin({
            'process.env': {
                'ENV': JSON.stringify(ENV)
                ...
            }
        })
    ],
    ...
});
```

5.5.7.4 TelaService

The TelaService is responsible for communicating with the Tela server, executing requests and parsing the responses.

It is important to note that Angular 2 uses RxJS[45] to implement the asynchronous observable pattern (Angular, 2016), which means that the service will not be based on promises but in observables.

When the service is instantiated, it retrieves the URL from the configuration:

```
@Injectable()
export class TelaService {
    private url: string;
    constructor(private http: Http, private config: Config) {
        this.url = config.tela.server;
    }
}
```

The list of Instagram actions the `TelaService` offers are:

```
create(token: string): Observable<String>;
self(token: string): Observable<User>;
user(token: string, username: string): Observable<User>;
following(token: string): Observable<User[]>;
followers(token: string): Observable<User[]>;
friends(token: string): Observable<User[]>;
relationship(token: string, username: string): Observable<UserRelationship>;
selfMedia(token: string, limit: number): Observable<Media[]>;
likes(token: string, mediaId: string): Observable<User[]>;
```

[45] https://github.com/ReactiveX/RxJS

The `TelaService` has auxiliary methods, like the ones to prepare the headers:

```
private static getHeaders(): Headers {
    return new Headers({'Content-Type': 'application/x-www-form-urlencoded'});
}

private static getAuthorizedHeaders(token: string): Headers {
    let headers = TelaService.getHeaders();
    headers.set("Authorization", 'Bearer ' + token);
    return headers;
}
```

For example, the function that retrieves the relationship between a user and the logged user:

```
relationship(token: string, username: string): Observable<UserRelationship> {
    let url = Location.joinWithSlash(this.url,
                                'action/instagram/relationship?username=' + username);
    return this.http
        .get(url, new RequestOptions(
            {headers: TelaService.getAuthorizedHeaders(token)}
        ))
        .map(res => res.json())
        .map(data =>
            new UserRelationship(data['incoming_status'],
                                 data['outgoing_status'],
                                 data['target_user_is_private'])
            )
            .catch(this.handleError);
}
```

5.5.7.5 AuthService

The `AuthService` is in charge of storing the logged user and Tela access token in the local storage, so they persist. Its implementation is quite straightforward and does not need any special explication:

```
@Injectable()
export class AuthService {

    static readonly USER_KEY: string = 'user';
    static readonly TOKEN_KEY: string = 'token';

    private setUser(user: User): void {
        localStorage.setItem(AuthService.USER_KEY, JSON.stringify(user));
    }

    getUser(): User {
        return JSON.parse(localStorage.getItem(AuthService.USER_KEY));
    }

    private setToken(token: string): void {
        localStorage.setItem(AuthService.TOKEN_KEY, JSON.stringify(token));
```

```
    }

    getToken(): string {
        return JSON.parse(localStorage.getItem(AuthService.TOKEN_KEY));
    }

    logIn(user: User, token: string): void {
        this.setUser(user);
        this.setToken(token);
    }

    logOut(): void {
        this.setToken(null);
        this.setUser(null);
    }

    isLogged(): boolean {
        return this.getToken() != null && this.getUser() != null;
    }

}
```

5.5.7.6 Login

The `LoginComponent` only renders the login page, only binding the Instagram URL the user will have to access to log in. When this occurs, the user is redirected to the `AuthComponent`, which shows a loading window while it creates the connection in Tela.

`AuthComponent` implements `OnInit`, which means that once the component is ready the `ngOnInit()` method is called:

```
ngOnInit():void {
    let token = AuthComponent.retrieveInstagramTokenFromUrl();
    if (token == null) {
        this.router.navigate(['login']);
    }
    this.tela
        .create(token)
        .subscribe((token:string) => {
            this.tela
                .self(token)
                .subscribe((user:User) => {
                    this.auth.logIn(user, token);
                    this.router.navigate(['']);
                });
        });
}
```

5.5.7.7 Dashboard

The `WhoComponent` file only contains the template of the window, and ensures that the user is logged:

```
@Component({
    selector: 'who',
    template: `<toolbar></toolbar><dashboard></dashboard>`
})
export class WhoComponent {
    constructor(private auth:AuthService, private router:Router) {
        if (!this.auth.isLogged()) {
            this.router.navigate(['login']);
        }
    }
}
```

The `ToolbarComponent` does not require especial attention. On the other hand, the `DashboardComponent` contains the main functionality.

It has the `showFollowers()`, `showFollowing()`, and `showFriends()` methods, which updates the properties that Angular will render.

The three methods will cache the results. Otherwise, the application will be executing requests to the server each time the user would change between "followers", "following" and "friends".

As an example, this is the implementation of `showFollowers()`:

```
showFollowers(): void {
    this.hideAll();
    if (!this.followers) {
        this.loading = true;
        this.tela
            .followers(this.token)
            .finally(() => this.loading = false)
            .subscribe(
                followers => {
                    this.followers = followers;
                    this.filteredResultUsers = this.resultUsers = followers;
                    this.showingFollowers = true;
                }
            );
    }
    else {
        this.filteredResultUsers = this.resultUsers = this.followers;
        this.showingFollowers = true;
    }
}
```

The `DashboardComponent` also has a `username` property which is an input for the UserDetailsComponent:

```
@Component({
    selector: 'user-details',
    inputs: ['username'],
    styleUrls: ['./user-details.component.css'],
    templateUrl: './user-details.component.html'
})
export class UserDetailsComponent implements OnChanges {

    private token:string;

    readonly numberOfMedia = 20;
    private numberOfProcessedMedia:number;

    isUserLoading:boolean;
    user: User;
    error:string;

    relationshipMessage: string;
    media: Media[];
    isMediaLoading:boolean;

    constructor(private auth:AuthService, private tela:TelaService) {
        this.token = this.auth.getToken();
        this.media = [];
        this.isUserLoading = this.isMediaLoading = false;
    }
    ...
}
```

When `username` is updated in the `DashboardComponent` (e.g. the user searches a username), it fires the `ngOnChange()` method in the `UserDetailsComponent`, which will retrieve the searched user from Tela:

```
ngOnChanges(changes:SimpleChanges):void {
    if (changes['username']) {
        let username = changes['username']['currentValue'];
        this.getUser(username);
    }
}
```

The `getUser()` method, in turn, has the following implementation:

```
getUser(username:string):void {
    this.user = this.error = null;
    this.isUserLoading = true;
    this.tela
        .user(this.token, username)
        .finally(() => this.isUserLoading = false)
        .subscribe(
            (user:User) => {
                this.user = user;
                this.relationship(username);
                this.getSelfMedia();
            },
            err => {
                if (err.status == 403)
                    this.error = `The user @${username} is private`;
                else
                    this.error = 'The user could not be displayed!';
            }
        );
}
```

- The `relationship()` method updates the message to show regarding how the user is connected to the logged user.
- The `getSelfMedia()` retrieves the latest media of the logged user and filters only those that have been liked by the analyzed user.

5.5.8 Test Results

- After each iteration, the outcome of them was manually tested, ensuring that the developed functionality met the expectations and requirements.
- At the end of its development, the twelve tests were manually performed, ensuring that the final version met the expectations and was completely ready to use.
- Regarding the prototype evaluation usability tests, the system was tested with two subject users. Both of them completed the tasks we ask them to do without any problem, so we might conclude that this early prototype is finished and ready for its further development –which is out of the scope of this project.

5.6 System Manuals

The following manual and can be found (in Markdown format) at `/tela-who/README.md`.

Tela Who? is a web application for companies, brands, and regular users, that allows them to obtain a better understanding about how a user is connected with them, so that they can improve their relationship and target similar users.

5.6.1 Building

Tela Who? building requires **npm** and **Webpack**. In order to generate the production files, execute:

```
npm install
```

The command will automatically execute Webpack, and the final files will be placed at the `dist/` folder.

5.6.2 Running the Built-In Server

Tela Who? is shipped with a Node server, so that you can start the application right after building it.

In order to do it, just execute:

```
npm install  # If not executed before
npm start
```

5.6.3 Deploying Tela Who? in a Server

You can deploy Tela Who? in any type of server, as long as it redirects all the not found requests to the `index.html` file.

In the built-in Node server, this is done by including the following lines:

```
var publicPath = path.resolve(__dirname, 'dist');
app.use(express.static(publicPath));
app.all('*', function (req, res) {
    res.status(200).sendFile(path.join(publicPath, 'index.html'));
});
```

5.7 System Outcome and Extensions

5.7.1 System Outcome

The outcome of this system is a functional prototype of our web application, which fulfils all our requirements, limited by the system scope.

However, it should be remarked that this is a prototype and not a final product. Therefore, this should not be used for production purposes nor real users; unless it is for testing purposes.

This application is deployed at: https://tela-who.herokuapp.com.

However, as commented in the report, only sandbox users can log in due to the restrictions of the Instagram Sandbox Mode.

5.7.2 Extensions

This system and its functionality is quite basic. If we would like to further develop it and obtain a final system, we should:

- Develop automated tests.
- Improve the error handling, checking corner cases.
- In addition to checking what media the user has liked in the analysis page, comments of the media should be checked too.
- Perform usability tests with real users over the final version of the application.

5.8 Content delivered

The source code of the project can be found at the `tela-who` folder. Within it, we find:

Folder: /tela-who	
Directories	
Folder	**Contents**
/config	Configuration files to build the application
/dist	Production files (if built)
/node_modules	Development dependencies (npm has install the project)
/public	Static files, like images and CSS files.
/src	Source code
Files	
File	**Description**
.env.sample	Sample of .env file
.gitignore	Content ignored by Git
package.json	npm main file
README.md	Instructions for building and deployment
server.js	Built-in Node server
tsconfig.json	Typescript compiler configuration
webpack.config.js	Webpack configuration

6 System IV: Tela Hawk

6.1 Introduction

6.1.1 Description of the System

Tela Hawk is a tool that monitors the variation over time of the number of media posted by the user, followers and people the user is following; displaying this information to the user.

At the same time, this project acts as a proof of concept of a web application that interacts with a Tela Server and its scheduler, while it offers a solution addressing the needs of real end users

6.1.2 Development Process

As we have chosen to follow an iterative agile approach based on sprints, we will develop Tela Hawk in the same way. However, as it is fairly simple application with well-defined requirements, we will try to complete it in a small number of sprints.

6.2 Analysis of the System

6.2.1 Analysis of Use Cases

Users will interact with Tela Hawk to obtain a better understanding about how they popularity is changing over time, which include:

- Log in and log out using an Instagram account.
- Start / stop watching themselves.
- Monitor how his popularity has changed.

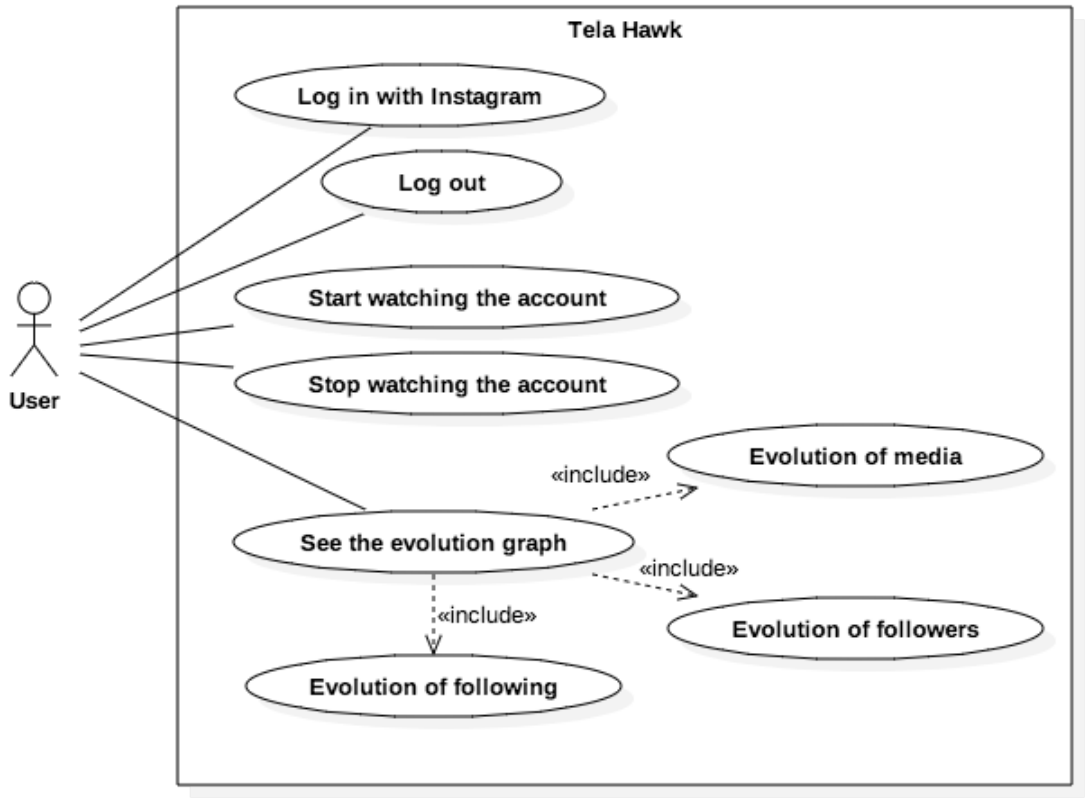


Figure 6.1 Tela Hawk Use Cases Diagram

6.2.2 Specification of System Requirements

From the use cases identified in the previous section, we will elaborate a list of functional and non-functional requirements the system should meet. This will be the base for the design phase, and our focus during testing.

6.2.2.1 Functional Requirements

ID	Name	Description
FR1	Authentication	
FR1.1	Log in	The user can log in with his Instagram account, and his user and name are displayed.
FR1.2	Log out	The user can log out the system.
FR2	Monitoring	
FR2.1	Start monitoring	Tela will schedule a task to obtain information of the user
FR2.2	Stop monitoring	Tela will cancel the scheduling of the task
FR2.3	See the results	Observe the evolution of the number of media, followers and following.

6.2.2.2 Non-Functional Requirements

ID	Description
NFR1.1	The website deployment will be easy to perform.
NFR1.2	The website should not depend on the operating system of the web server.
NFR1.3	The application will be based on Angular, all the logic and operations are done in the client side.

6.2.3 Design of Use Case Scenarios

Due to the limited scope of the following use case scenarios will omit the alternate flows.

6.2.3.1 Authentication (FR1)

Log in (FR1.1)	
Description	Log in with his Instagram account.
Primary Actors	User, Instagram & Tela.
Preconditions	The user has an Instagram account.
Post conditions	The system stores the access token of the Tela Session.
Trigger	The user clicks the log in button.
Flow	1. The user is redirected to the Instagram login page. 2. The user logs into Instagram. 3. Instagram redirects the user to the system, along with an Instagram access token. 4. The system sends a request to Tela to create a session with the Instagram access token. 5. Tela sends the access token of a new session back to the system. 6. The system stores the access token and redirects the user to the dashboard.

Log out (FR1.2)	
Description	Log out from the system.
Primary Actor	User & Server.
Preconditions	The user is logged in.
Post conditions	The connection is destroyed, in both the server and the local machine.
Trigger	The user clicks the log out button.
Flow	1. The system sends a request to the server to delete the session. 2. The server deletes the session and answers with a success message. 3. The system deletes the connection. 4. The user is redirected to the log in page.

6.2.3.2 Monitoring (FR2)

Start monitoring (FR2.1)	
Description	Schedule the retrieve of information of the logged user in Tela.
Primary Actors	User & Server.

Preconditions	The user is logged in and is not currently monitoring himself.
Post conditions	The action has been scheduled in Tela.
Trigger	The user clicks the 'start watching' button.
Flow	1. The system sends a request to Tela to schedule his monitoring. 2. Tela does it, and sends a success message back. 3. The option is disabled.

Stop monitoring (FR2.2)	
Description	Cancel the scheduling of the monitoring task.
Primary Actors	User & Server.
Preconditions	The user is logged in and he is currently monitoring himself.
Post conditions	The monitoring of the user is not scheduled any longer.
Trigger	The user clicks the 'stop watching' button.
Flow	1. The system sends a request to Tela to cancel the scheduling. 2. Tela does it and sends a success message back. 3. The option is disabled.

See the results (FR2.3)	
Description	Show the friends of the logged user.
Primary Actors	User & Server.
Preconditions	The user is logged in.
Post conditions	-
Trigger	The user clicks the logs in.
Flow	1. The system sends a request to Tela to receive the monitored information. 2. The information is rendered.

6.2.4 Analysis of Components

As Tela Who?, this system will have two main components:

- The `login` component, which will be responsible to log the user into Instagram and create a Tela connection.
- The `hawk` component, which is the main one. It will be composed by a `toolbar` component, and a `dashboard` component.

6.3 Design

6.3.1 Design of Wireframes of the Visual Interface

First, this will be the login view:

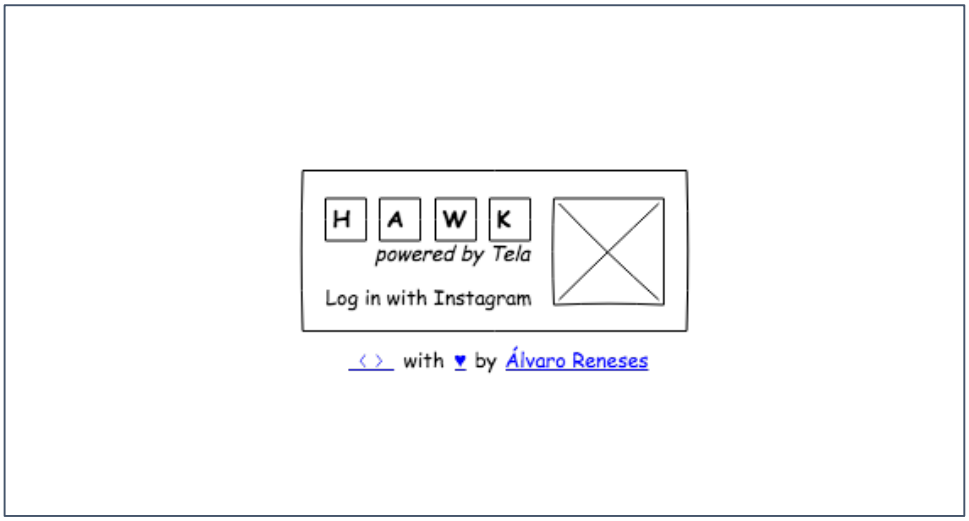


Figure 6.2 Tela Hawk Login Wireframe

Once the user clicks on Log in, he will be redirected to Instagram. Then, after authentication, the user will be redirected back to the dashboard:

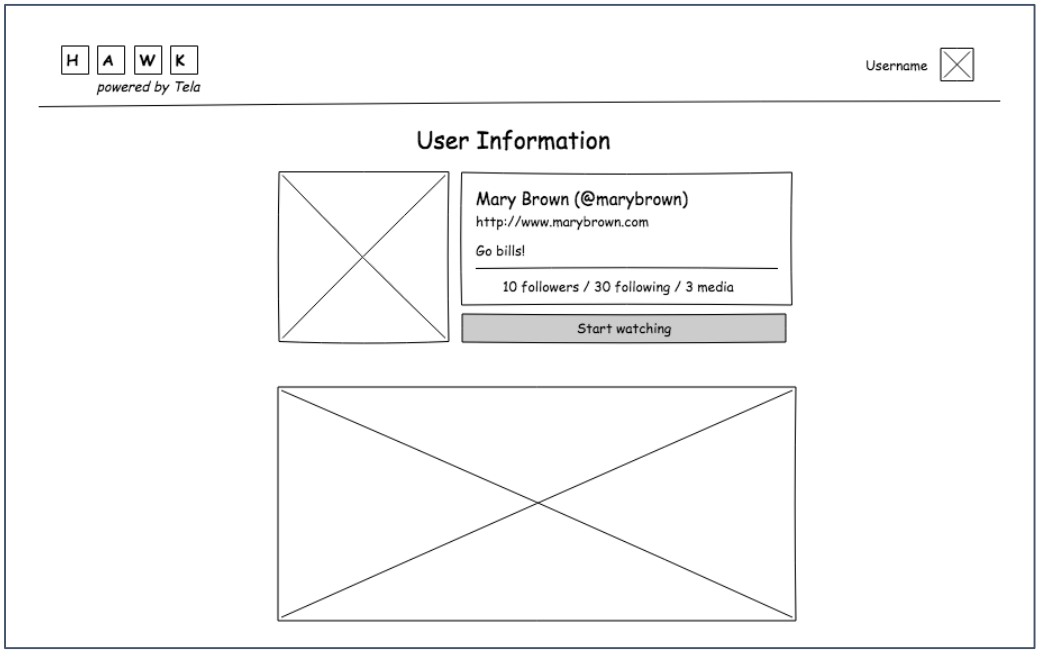


Figure 6.3 Tela Hawk Dashboard Wireframe

- If the user hovers or clicks on the username (in the toolbar), a logout button will appear.
- Clicking the "Start watching" button will the system perform the action, and it will be replaced by "stop watching".

6.3.2 Design of Components

Based on the previous wireframes, we will identify all the components we could decompose the application in:

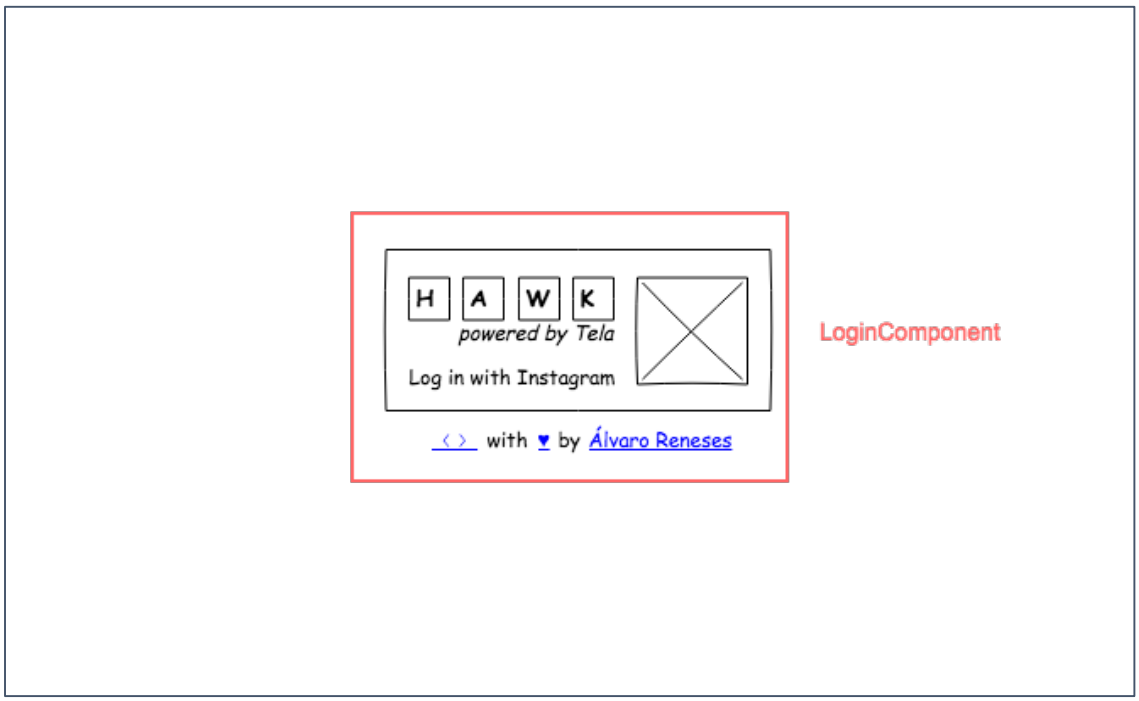


Figure 6.4 Tela Hawk Login Components

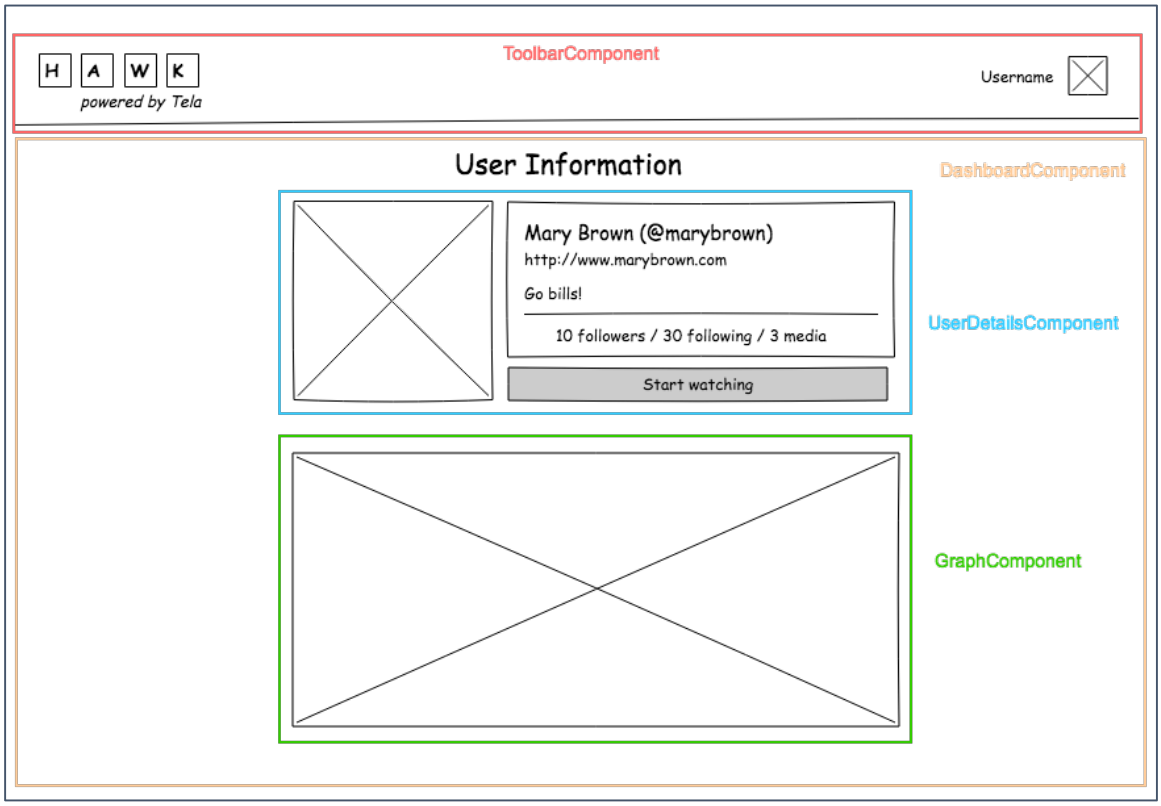


Figure 6.5 Tela Hawk Dashboard Components

These components could be represented as a component tree, so that we obtain a better view of how each component depends on the rest:

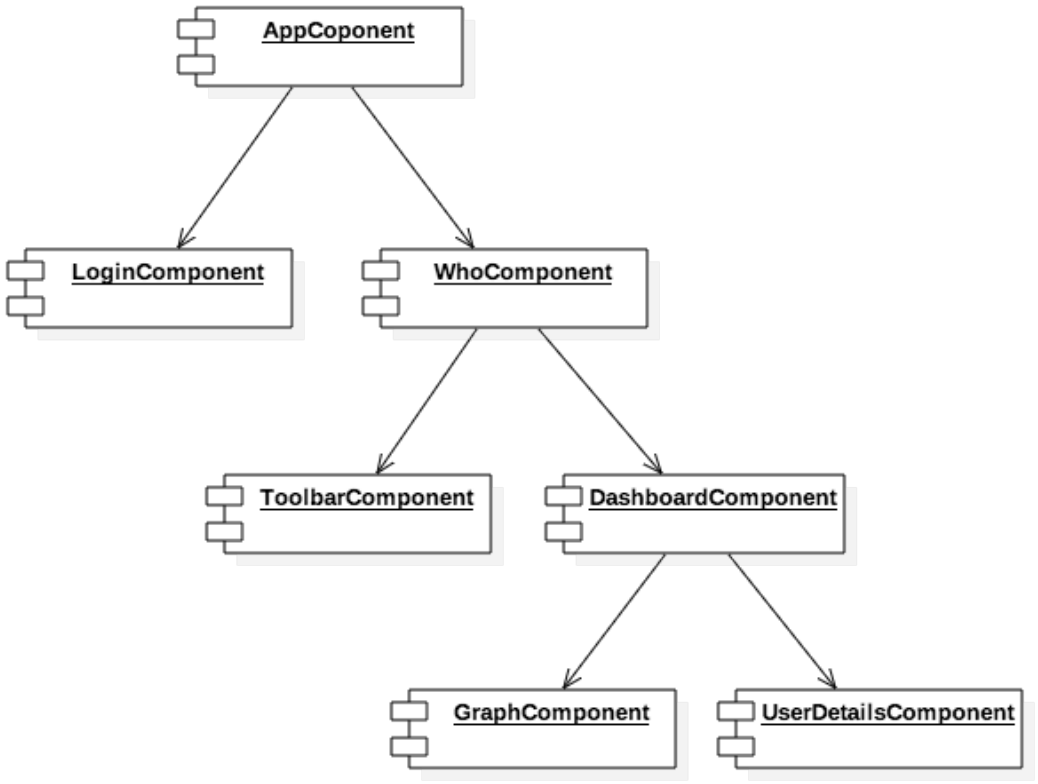


Figure 6.6 Tela Hawk Component Diagram

6.4 Test Plan

6.4.1 Unit Tests

Due to the small size of the system, and the limited scoped of it as it acts as a proof of concept for a client application interacting with the Tela Server, unit tests have been omitted.

6.4.2 Integration Testing

Testing scripts will be designed outlining a sequence of tasks to perform in order to check the compliance with the functional and non-functional requirements:

Requirement FR1: Authentication	
Test case: TC1.1.1	
Description	The user logs in with his Instagram Account
Result expected	The user is redirected to the dashboard, and his username and image are displayed in the toolbar
Test case: TC1.1.2	
Description	Once logged, the user logs out
Result expected	The session is destroyed and the user is redirected to the login page
Requirement FR2: Monitoring	
Test case: TC2.1.1	
Description	The user clicks “start watching”
Result expected	The action is scheduled in Tela, and the button changes to “stop watching”
Test case: TC2.1.2	
Description	The user clicks on “stop watching”
Result expected	The action is canceled in Tela, and the button changes to “start watching”
Test case: TC2.1.3	
Description	The user logs in into the dashboard, after “start watching”
Result expected	Results are shown in the graph

6.4.3 Usability Tests

As explained in the previous system, Tela Who?, prototype evaluation tests should be realized. In order to do it, we will choose the most important tasks, and ask subject users to complete them:

- Log into the system with Instagram, and then log out.
- Start monitoring himself, and then stop doing it.
- See the monitoring results.

6.5 System Implementation

The programming language and technologies employed, coding style, tools and programs used are the same than in the previous system, Tela Who?

6.5.1 Dependencies

In addition to dotenv, already commented in Tela Who?, Tela Hawk has two more dependencies.

6.5.1.1 Highcharts + jQuery

Highcharts[46] is a JavaScript library for chart generation. It has been used to show the monitoring results, along with jQuery[47]. Version: 5.0.2.

6.5.2 Visual Interface

In this section we will include screenshots of how the final visual interface looks. First, the login page:

Figure 6.7 Tela Hawk Login Page

[46] http://www.highcharts.com

[47] https://jquery.com

Once we are connected, we are redirected to the dashboard, showing the monitoring results:

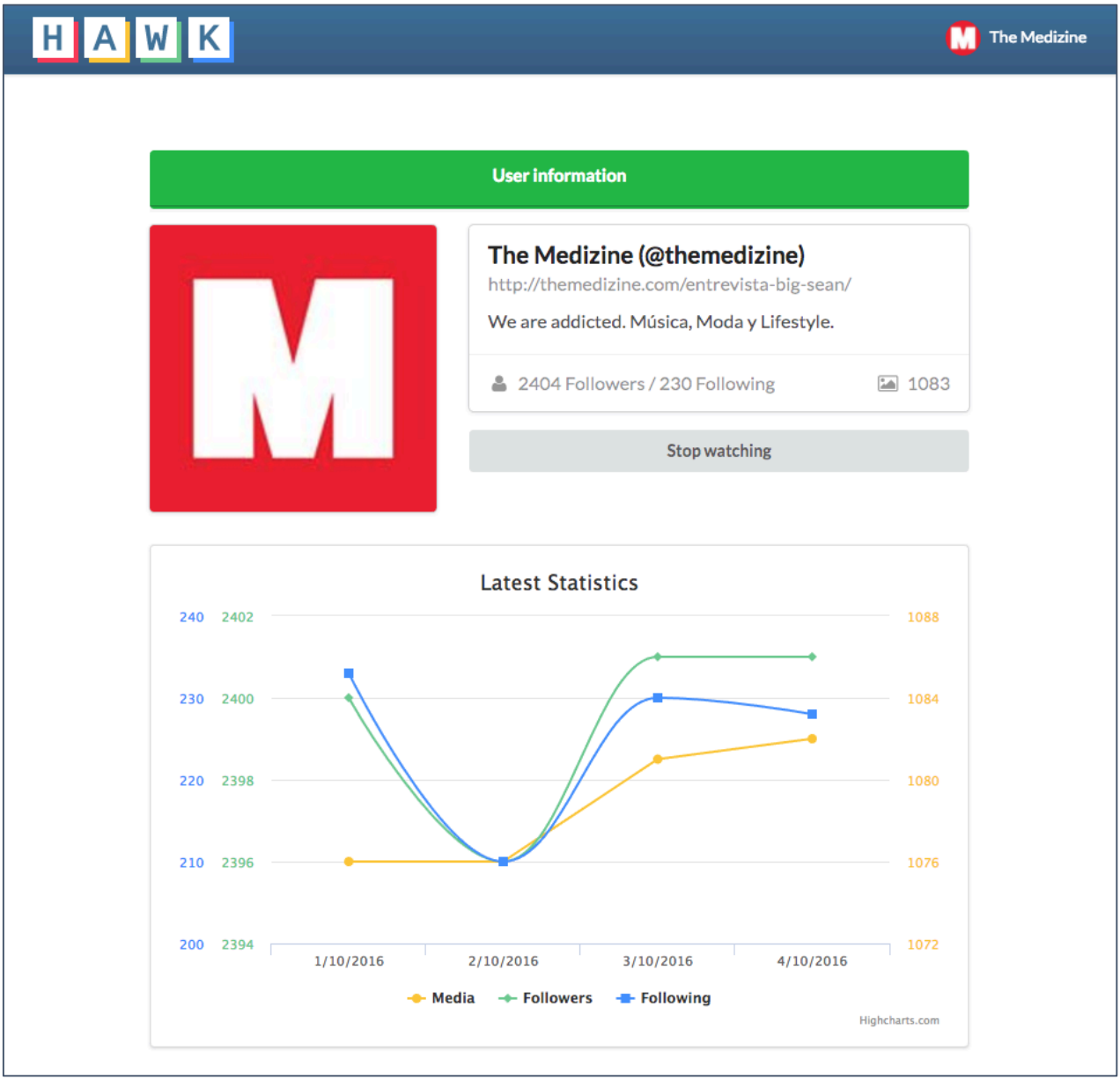


Figure 6.8 Tela Hawk Dashboard Page

6.5.3 Description of Components

- **AppComponent**: Main component, contains the router outlet which will select the corresponding view.
- **login/LoginComponent**: Login view.
- **login/AuthComponent:** Receives the callback from Instagram, extracts the access token, creates the session and redirects the user to the dashboard.
- **who/HawkComponent**: Dashboard wrapper, containing the `ToolbarComponent` and the `DashboardComponent`.
- **who/toolbar/ToolbarComponent**: Toolbar in the top of the application.
- **who/dashboard/DashboardComponent**: Dashboard with the main functionality.
- **who/dashboard/GraphComponent**: Graph that will display the results.
- **who/dashboard/UserDetailsComponent**: User analysis.
- **services/AuthService**: Service in charge of authentication management.
- **services/TelaService**: Service in charge of interacting with the Tela server.

6.5.4 Implementation Details

Most part of the implementation is identical to Tela Who?, and will not be covered. We will only focus on the most important components with implementation differences:

6.5.4.1 TelaService

Although most part of the implementation of `TelaService` is identical to Tela Who?, the methods offered by the server differ:

```
create(token: string): Observable<String> { ... }
self(token: string): Observable<User> { ... }
user(token: string, username: string): Observable<User> { ... }
userById(token: string, userId: number): Observable<User> { ... }
getScheduled(token: string): Observable<Scheduled[]> { ... }
schedule(token: string): Observable<Scheduled> { ... }
cancel(token: string, scheduledId: number): Observable<string> { ... }
counts(token: string, userId: number): Observable<Counts[]> { ... }
```

6.5.4.2 Dashboard

The `DashboardComponent` initializes the current user and its counts at the moment it is constructed:

```
export class DashboardComponent {

    private token: string;

    user: User;
    isLoadingUser: boolean;
    counts: Counts[];

    constructor(private auth: AuthService, private tela: TelaService) {
        this.token = this.auth.getToken();
        let userId = auth.getUser().id;
        this.getUser(userId);
        this.getCounts(userId);
    }
 }
```

The `GraphComponent` is monitoring this property, and will render the graph at the moment it changes:

```
ngOnChanges(changes: SimpleChanges): void {
    if (changes['counts'] && this.counts) {
        let dates: string[] = [];
        let media: number[] = [];
        let followers: number[] = [];
        let following: number[] = [];
        let lastDate: string;
        console.log(this.counts);
        this.counts
            .sort((c1, c2) => c1.createdAt.getTime() - c2.createdAt.getTime())
            .forEach(c => {
            let date =
`${c.createdAt.getDay()}/${c.createdAt.getMonth()}/${c.createdAt.getFullYear()}`;
            if (lastDate === date) {
                dates.pop();
                media.pop();
                followers.pop();
                following.pop();
            }
            dates.push(date);
            media.push(c.numberOfMedia);
            followers.push(c.numberOfFollowers);
            following.push(c.numberOfFollowing);
            lastDate = date;
        });
        this.renderChart(dates, media, followers, following);
    }
}
```

6.5.5 Test Results

- After each iteration, the outcome of them was manually tested, ensuring that the developed functionality met the expectations and requirements.
- At the end of its development, the twelve tests were manually performed, ensuring that the final version met the expectations and was completely ready to use.
- As with Tela Who?, prototype evaluation usability tests were performed with two subject users. Both of them completed the tasks we ask them to do without any problem, so we might conclude that this early prototype is finished and ready for its further development –which is out of the scope of this project.

6.6 System Manuals

The following manual and can be found (in Markdown format) at `/tela-hawk/README.md`.

6.6.1 Building

Tela Hawk building requires **npm** and **Webpack**. In order to generate the production files, execute:

```
npm install
```

The command will automatically execute Webpack, and the final files will be placed at the `dist/` folder.

6.6.2 Running the Built-In Server

Tela Hawk is shipped with a Node server, so that you can start the application right after building it.

In order to do it, just execute:

```
npm install  # If not executed before
npm start
```

6.6.3 Deploying Tela Hawk in a Server

You can deploy Tela Hawk in any type of server, as long as it redirects all the not found requests to the `index.html` file.

In the built-in Node server, this is done by including the following lines:

```
var publicPath = path.resolve(__dirname, 'dist');
app.use(express.static(publicPath));
app.all('*', function (req, res) {
    res.status(200).sendFile(path.join(publicPath, 'index.html'));
});
```

6.7 System Outcome and Extensions

6.7.1 System Outcome

As Tela Who?, the outcome of this system is a functional prototype of our web application, which fulfils all our requirements, limited by the system scope. However, it should be remarked that this is a prototype and not a final product. Therefore, this should not be used for production purposes nor real users; unless it is for testing purposes.

This application is deployed at: https://tela-hawk.herokuapp.com.

However, as commented in the report, only sandbox users can log in due to the restrictions of the Instagram Sandbox Mode.

6.7.2 Extensions

This system and its functionality is quite basic. If we would like to further develop it and obtain a final system, we should:

- Develop automated tests.
- Improve the error handling, checking corner cases.
- Perform usability tests with real users over the final version of the application.

6.8 Content delivered

The source code of the project can be found at the `tela-who` folder. Within it, we find:

Folder: /tela-hawk	
Directories	
Folder	**Contents**
/config	Configuration files to build the application
/dist	Production files (if built)
/node_modules	Development dependencies (npm has install the project)
/public	Static files, like images and CSS files.
/src	Source code
Files	
File	**Description**
.env.sample	Sample of .env file
.gitignore	Content ignored by Git
package.json	npm main file
README.md	Instructions for building and deployment
server.js	Built-in Node server
tsconfig.json	Typescript compiler configuration
webpack.config.js	Webpack configuration

7 Global Problems Found

7.1 Instagram Disables Relationships Endpoints

The Instagram endpoints to retrieve the followers and following of the authenticated user are:

```
GET /users/self/follows
GET /users/self/followed-by
```

At the beginning of the project, it was possible to also retrieve the followers and following of any user, replacing 'self' by the user ID. For example, to retrieve the followers of the user 123:

```
GET /users/123/followed-by
```

Although these endpoints were not officially documented, there were developers and applications using them. As the documentation of the API has some problems, I assumed that it was a simply lack of documentation.

However, two months before the end of the project, Instagram banned the access to those endpoints.

At that moment, the Instagram module was already built, and I had to modify it, along with the repositories, the database, and so on.

Worst yet, the only client application which was already built, Tela Link, was completely based on them. This is a screenshot of it:

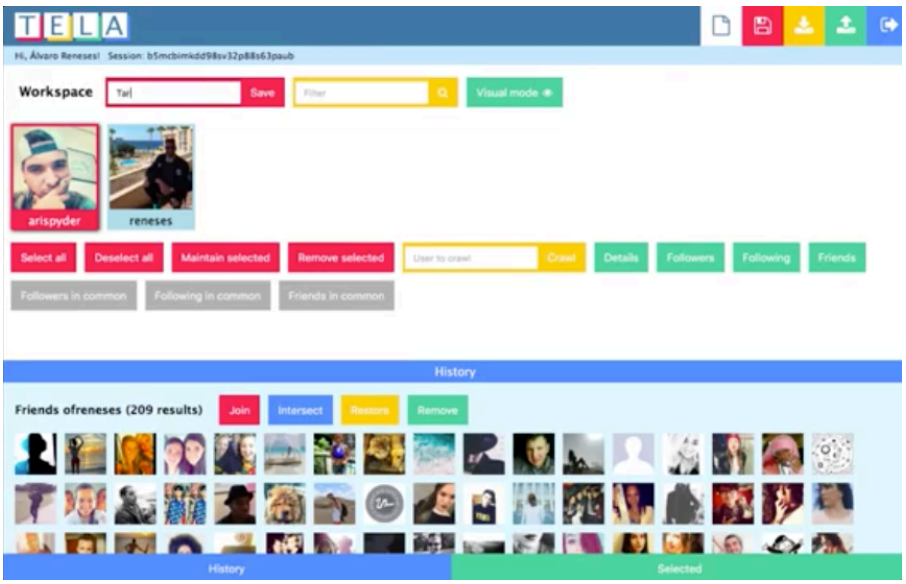


Figure 7.1 Tela Link

This application offered advanced analysis of Instagram relationships. The code is not runnable anymore, as Tela had to be adapted to the changes. However, I have attached a video called 'tela-link.mp4' to the files delivered, which shows the application in action (it was recorded before the endpoints were disabled).

7.2 Instagram Forcing Sandbox Mode and Toughening App Reviews

Once this project was already started, Instagram announced that all the new applications will automatically be put in Sandbox mode (Instagram Developer Documentation, 2016). On the other hand, existing applications would have some months to pass the Permissions Reviews.

In addition to that, they decided to toughen the app reviews, restricting the permissions, so that users would only use the official applications (Spencer, 2015) (Purba, 2015).

This affected the project mainly in three ways:

- The applications could not be tested in the same way. As the users have to grant explicit permissions, and only the content belonging to the sandbox user appear, most part of the API calls returned empty results.
- The new API limit for sandbox users is quite easy to reach, having to wait up to one hour in order to be able to continue testing it.
- Due to the new toughened permissions review, the original planning for client applications would not be accepted.
- Even if they are, the new process takes several days and their blocking reasons are quite opaque (Instagram Developer Documentation, 2016).

7.3 Interaction with the Instagram and Twitter APIs

The usage of Instagram and Twitter APIs turn out to be quite challenging.

On one hand, the Instagram one is quite easy to work with as it offers authentication based on Bearer tokens. However, its documentation is not very complete and the sandbox mode where Instagram puts all the non-reviewed applications is way too restrictive.

On the other hand, the Twitter documentation is very detailed. However, its authentication flow based on OAuth is quite more complex and requires external libraries and extra steps.

8 Conclusions

The three systems that we wanted to develop were successfully produced, although with different implementation levels:

- Tela is a complete system, with robust test suites, complete functionality, and production ready.
- Tela CLI is a functional system, although it should be reviewed and further developed before using it in productions environments.
- Tela Who? and Tela Hawk is prototypes of web applications, which should not be used by end users.

Regarding the development process, we had several problems that have been explained in the previous section. What we learned from them is:

- "Software always change". Even in a project like this, that does not have an external source of requirements, there are external forces that may affect the system. In our case, Instagram disabling those endpoints make us throw most part of the project already designed and implemented, and plan again all the systems that we would implement.
- Agile methods help to overcome unexpected problems. We had to throw most part of the project away. However, if we would have been used a waterfall model, (almost) the whole project would have been useless.
- It is dangerous to base our applications on external APIs, whose policies might change. With the change of policies and restricting the permissions, a lot of real applications that were using the Instagram API had to pivot to other services, or to stop offering their services. In the same way, Tela Link was not feasible any longer.

9 Alphabetical Index

10 Bibliography

- Analytictech. (2016). *What is Social Network Analysis?*. [online] Available at: http://www.analytictech.com/networks/whatis.htm [Accessed 2 Nov. 2016].
- Angular. (2016). *Architecture Overview*. [online] Available at: https://angular.io/docs/ts/latest/guide/architecture.html [Accessed 2 Nov. 2016].
- Angular. (2016). *HTTP Client - ts - RxJS library*. [online] Available at: https://angular.io/docs/ts/latest/guide/server-communication.html#!#rxjs [Accessed 1 Nov. 2016].
- Beach, C. (2016). *What is an API?*. [online] Quora. Available at: https://www.quora.com/What-is-an-API-4 [Accessed 1 Nov. 2016].
- Beal, V. (2016). *What is API - Application Program Interface?*. [online] Webopedia. Available at: http://www.webopedia.com/TERM/A/API.html [Accessed 2 Nov. 2016].
- Christensson, P. (2013). *Framework Definition*. [online] TechTerms. Available at: http://techterms.com/definition/framework [Accessed 2 Nov. 2016].
- Cuanto Gana. (2015). *Cuanto Cobra un Ingeniero Informático en España?*. [online] Available at: http://www.cuanto-gana.com/cuanto-cobra-un-ingeniero-informatico-en-espana/ [Accessed 2 Nov. 2016].
- DB-Engines Ranking. (2016). *Popularity ranking of graph DBMS*. [online] Available at: http://db-engines.com/en/ranking/graph+dbms [Accessed 2 Nov. 2016].
- Deering, S. (2012). *Do you know what a REST API is?*. [online] SitePoint. Available at: https://www.sitepoint.com/developers-rest-api/ [Accessed 2 Nov. 2016].
- Dictionary.com. (2016). *Definition of Social Network*. [online] Available at: http://www.dictionary.com/browse/social--network [Accessed 1 Nov. 2016].
- Dotson, C. (2014). *Node.js vs Python vs PyPy – A Simple Performance Comparison*. [online] Chad Dotson. Available at: http://www.cdotson.com/2014/08/nodejs-vs-python-vs-pypy-a-simple-performance-comparison/ [Accessed 4 Nov. 2016].
- ESLint. (2016). *no-console - Rules*. [online] Available at: http://eslint.org/docs/rules/no-console [Accessed 4 Nov. 2016].
- Franks, J. (1999). *RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication*. [online] IETF. Available at: https://tools.ietf.org/html/rfc2617 [Accessed 4 Nov. 2016].
- Google. (2016). *Google Java Style Guide*. [online] Available at: http://google.github.io/styleguide/javaguide.html [Accessed 4 Nov. 2016].
- Instagram Developer Documentation. (2016). *Permissions Review*. [online] Available at: https://www.instagram.com/developer/review/ [Accessed 1 Nov. 2016].
- Instagram Developer Documentation. (2016). *Sandbox Mode*. [online] Available at: https://www.instagram.com/developer/sandbox/ [Accessed 1 Nov. 2016].
- Instagram. (2016). *FAQ*. [online] Available at: https://www.instagram.com/about/faq/ [Accessed 30 Oct. 2016].
- Johansson, M. (2016). *Why are IT systems in big enterprises usually built using Java, instead of Python or JavaScript?*. [online] Quora. Available at: https://www.quora.com/Why-are-IT-systems-in-big-enterprises-usually-built-using-Java-instead-of-Python-or-JavaScript [Accessed 4 Nov. 2016].

- McDonnell, M. (2010). *How to Implement Interfaces in JavaScript*. [online] JavaScriptBank.com. Available at: http://www.javascriptbank.com/how-implement-interfaces-in-javascript.html [Accessed 4 Nov. 2016].
- Neo4j. (2016). *What is a Graph Database? A Property Graph Model Intro*. [online] Available at: https://neo4j.com/developer/graph-database/ [Accessed 2 Nov. 2016].
- Node.js. (2016). *Node.js*. [online] Available at: https://nodejs.org/en/ [Accessed 4 Nov. 2016].
- npm. (2016). *tslint*. [online] Available at: https://www.npmjs.com/package/tslint [Accessed 4 Nov. 2016].
- OrientDB Manual. (2016). *Installation*. [online] Available at: http://orientdb.com/docs/2.2/Tutorial-Installation.html#building-a-single-executable-jar-with-orientdb [Accessed 4 Nov. 2016].
- OrientDB Manual. (2016). *SQL Reference*. [online] Available at: http://orientdb.com/docs/2.1/SQL.html [Accessed 4 Nov. 2016].
- OrientDB. (2016). *OrientDB vs Neo4j - More Than Just a Mere Graph Database*. [online] Available at: http://orientdb.com/orientdb-vs-neo4j/ [Accessed 4 Nov. 2016].
- Padmanaban, R. (2014). *Regression Testing in the Agile World*. [online] TechWell. Available at: https://www.techwell.com/techwell-insights/2014/04/regression-testing-agile-world [Accessed 2 Dec. 2016].
- Purba, N. (2015). *Instagram's new API policy toughens access to its feed*. [online] WeLiveSecurity. Available at: http://www.welivesecurity.com/2015/11/18/instagrams-new-api-policy-toughens-access-feed/ [Accessed 2 Nov. 2016].
- PYPL. (2016). *PopularitY of Programming Language index*. [online] Available at: http://pypl.github.io/PYPL.html [Accessed 4 Nov. 2016].
- Rajkumar, J. (2016). *Automated Regression Testing Challenges in Agile Environment*. [online] Software Testing Help. Available at: http://www.softwaretestinghelp.com/automated-regression-testing-challenges-in-agile-testing-environment/ [Accessed 3 Nov. 2016].
- Redis.io. (2016). *Introduction to Redis*. [online] Available at: http://redis.io/topics/introduction [Accessed 4 Nov. 2016].
- Richardson, C. (2014). *Microservices Architecture pattern*. [online] Microservices.io. Available at: http://microservices.io/patterns/microservices.html [Accessed 21 Oct. 2016].
- Rouse, M. (2006). *What is database? - Definition from WhatIs.com*. [online] SearchSQLServer. Available at: http://searchsqlserver.techtarget.com/definition/database [Accessed 1 Nov. 2016].
- Sanfilippo, S. (2011). *How to take advantage of Redis just adding it to your stack*. [online] Antirez. Available at: http://oldblog.antirez.com/post/take-advantage-of-redis-adding-it-to-your-stack.html [Accessed 4 Nov. 2016].
- Semantic UI. (2016). *Semantic UI*. [online] Available at: http://semantic-ui.com [Accessed 1 Nov. 2016].
- Slf4j. (2016). *SLF4J*. [online] Available at: http://www.slf4j.org [Accessed 4 Nov. 2016].
- SonarQube. (2016). *SonarQube*. [online] Available at: http://www.sonarqube.org [Accessed 4 Nov. 2016].
- Spencer, G. (2015). *Instagram Updates Its Platform Policy, Prohibits Third Party Instagram Feed Apps*. [online] MacStories. Available at: https://www.macstories.net/news/instagram-updates-its-platform-policy-prohibits-third-party-instagram-feed-apps/ [Accessed 2 Nov. 2016].

- TIOBE. (2016). *TIOBE Index*. [online] Available at: http://www.tiobe.com/tiobe-index/ [Accessed 4 Nov. 2016].
- Twitter Help Center. (2016). *New user FAQs*. [online] Available at: https://support.twitter.com/articles/13920 [Accessed 2 Nov. 2016].
- UsabilityNet. (2006). *Evaluate prototype*. [online] Available at: http://www.usabilitynet.org/tools/evaluate.htm [Accessed 2 Nov. 2016].
- Webpack. (2016). *What is Webpack*. [online] Available at: http://webpack.github.io/docs/what-is-webpack.html [Accessed 2 Nov. 2016].
- Winkels, M. (2010). *Regression Testing with an Agile Mindset*. [online] Xebia Blog. Available at: http://blog.xebia.com/regression-testing-with-an-agile-mindset/ [Accessed 3 Nov. 2016].
- Zborowski, S. (2016). *Mini REST+JSON benchmark: Python 3.5.1 vs Node.js vs C++*. [online] Programming Soup. Available at: http://szborows.blogspot.com.es/2016/03/mini-restjson-benchmark-python-351-vs.html [Accessed 4 Nov. 2016].