POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master Degree Thesis

Design, development and integration of an Instructor Operator Station (IOS) for flight simulators

Advisor
prof. Riccardo Sisto

Candidate
Eugenio Sorbellini
matricola: 214631

Company Supervisor
Leonardo Aircraft Division
dott. ing. Gianfranco Bellora

Academic Year 2016-2017

Summary

This thesis was carried out in collaboration with the simulator department of Leonardo Aircraft Division.
On each simulator there is an interface where the instructor is able to control and modify aircraft parameters, external weather conditions and every other aspect relevant to the management of the simulation. This interface is known as Instructor Operator Station (IOS).
In Leonardo AD flight simulators are managed by a software called Simulation Framework, its duty is to read and write simulations variables and control the status of mathematical models. This Framework capable to interact with different simulators is in development and Leonardo AD is interested in an IOS that can be linked to it.
Usually IOSs are tailor-made for specific aircraft, the aim of this work is to define a generic solution that could be customized for different needs.
It is also an opportunity for the company to explore new technologies and evaluate their advantages.
The work flow was split in three phases:

- Collection of requirements common to different IOSs, by analysing existing software and interviewing pilot instructors.
- Definition of an architecture that can satisfy the requested modularity and other specific non functional requirements, such as compatibility with different operating systems (Linux and Windows at least) and with several screen sizes.
  The interface shall be suitable both for tablets and desktops, allowing the interaction with mouse and keyboard or touch-screens.
- Development of a prototype that implements a subset of the requirements found.

After the first phase, requirements were collected in a document and a Graphical User Interface (GUI) was designed. To provide a flexible GUI the IOS was built as a web application, taking advantage of HTML 5 and the Bootstrap library.
To speed up the development was used a web framework, different solutions were

analysed: Django, Spring, Laravel, Node.js, Xamarin and Ruby on Rails.
At the end was chosen Django (a web framework based on Python) for two main reasons: firstly the Simulation Framework had a Python API available and secondly Django relies on few external dependencies, has an integrated Object Relational Mapping (ORM) engine and comes with its own development server.
Those features are valuable in the simulator department, since for security reasons there is not an internet access. Every software packet must be provided off-line and root privileges are limited.
The application was developed using SQLite as database solution and the integrated server was used. The IOS runs from a folder and can be moved between different machines without worrying about databases and web servers configurations. At the same time professional solutions such as Apache web server and Oracle databases are fully supported, the ORM engine makes data migration easy, so the software can grow in complexity when needed.
Since the IOS shall display streams of data, WebSockets were introduced to break the synchronous paradigm of HTTP requests. A library was written to integrate the Python API with Django WebSockets (Channels) and to map the IOS variables to Simulation Framework ones.
Different simulators may have different variables names, the library and a set of tables allows to change the mapping by loading CSV configuration files from an administrative interface.
At the end of the development phase a user manual was produced, and the application was integrated on a research simulator.
Functionalities that provide support for all the IOS features, such as: reading of data streams, writings of variables and management of models were successfully tested and worked as expected.
Future developments may consider the integration of WebSockets support in the Simulation Framework, allowing to remove the overhead introduced by the Python API.

Contents

List of Figures

Chapter 1

Introduction

In this chapter basic concepts necessary to understand the thesis work are introduced.
In details are provided formal descriptions and classifications of Flight Simulators from the entities that regulate their development, then is explained what is an Instructor Operator Station (IOS), how it interact with simulators and what are the thesis objectives.

1.1 Flight Simulators and Instructor Operating Stations

Flight Simulators specifications are regulated by two main entities, the Federal Aviation Administration (FAA) and the European Aviation Safety Agency (EASA). Each authority has defined a set of categories in which simulators are divided depending on their functionalities and realism.
Without explaining all possible classifications with their peculiarities, here will be provided a general description of the two main categories in which both entities divide simulation devices and that are relevant for the IOS development.
Those categories are Full Flight Simulator (FFS) and Flight Training Device (FTD). FAA and EASA have different definitions of FFS and FTD [1] [2] [3], however they both agree on a set of general requirements that can be summarized as follow.

1.1.1 Full Flight Simulator

A FFS is a replica in full size of an actual aircraft deck and cockpit, it shall include all aircraft equipment and computers programs necessary to reproduce the behaviour of the aircraft on ground and flight operations, furthermore shall be present a visual system that emulates the out of flight cockpit view and a force cueing motion system that operates at least on 3 axes.
Then for FFS, both FAA and EASA provide a set of further categories, each category must respect an increasing number of characteristic, such as broader field of view angle or more axes for the motion system.

1.1.2 Flight Training Device

A FTD instead is an FFS where motion system and visual system are not required. The visual system is usually replaced by a screen.
It shall not reproduce the entire aircraft cabin, however the flight instruments and controls must be present.
There are rule and specification that simulators shall follow to be classified as FTD, but are less restrictive.
Both agencies as for FFS, have different categories to classify the level of realism of an FTD.

1.1.3 Instructor Operator Station

Both FFS and FTD are used for pilot training and shall provide an interface that allows the instructor pilot to control and modify aircraft parameters, external weather conditions and every other aspect relevant to the management of the simulation, this interface is generally called Instructor Operator Station, however in literature is also referred as Instructor Operating Station.
Both FFS and FTD shall have an IOS to comply with FAA and EASA specifications, although the complexity of the software is related to the complexity of the simulator and the training scenario it has been designed for.
The GUI can be displayed from a single screen up to several displays and the interaction can occur by means of mouse and keyboard or touch screens.
IOS in some scenario are also used for other purpose than pilot training. Aircraft engineers may use the IOS to inject variables in the simulator, this is usually done to test new components, before mounting them on real aircraft.

1.1.4 How simulator works

At the core of flight simulators there are mathematical models, those models emulate specific part of the aircraft or an aircraft subsystem such as the landing gear, the avionics, the hydraulic systems and so on.
They run in parallel, receiving and sending I/O signals to the flight instruments and to the IOS.
In Leonardo AD models are managed by a software called Framework or Simulation Framework. It is in charge of changing the state of models by running pausing and stopping them. It also allows the communication between models and the external environment and provides functionalities such as recording of variables in a specific moment or for a time interval.
Recordings are useful both for pilot instructors and avionic engineers, for instance an instructor can restart the simulation from a given time or show the flight record to pilots after an exercise.
Avionic engineers instead can use records of flight data to compare different aircraft configurations, and test changes on simulators before applying them to the real aircraft.
The IOS works on top of the Framework and interact with it through an API, although it is not just a GUI, it offers also extra functionalities that will be discussed in depth in the following chapters.

1.2 Objective of thesis

IOSs are usually commissioned together with simulators and tailored to them, offering the specific training functionalities the simulator is made for.
They are built to work on a predefined hardware and software, for instance a selected operating system, a fixed number of screen and resolutions, a predefined input device such as mouse/keyboards or touch screens.
While those are not limits for simulators that are made once and then operated for years as they are built, it can be a problem if project requirements change in time and if the IOS shall be able to control a range of devices instead than a single simulator.
Leonardo AD is involved in different research projects that requires flexibility in the IOS interface and functionalities. A Framework capable of interacting with a range of simulators is also being developed and maintained.
They need an IOS that can be configured to work with all the simulators controlled by the Framework, whose interface is flexible enough to be displayed on different devices, such as desktops with single or multiple screen and tablets, and who can be easily extended in functionalities depending on the needs of the simulation environment.

The development of a complete IOS requires time and resources that are beyond a 6 months thesis. The aim of this work is firstly to define what are the functionalities that a generic IOS shall implement, then explore possible IOS architectures that can satisfy the Leonardo AD needs and finally develop a working prototype.
Furthermore, is also an opportunity to work with software technologies outside the domain of existing business skills and discover which advantages they can bring to the company. Work flow can be summarized in those steps that will be discussed in the following chapters:

1. Comparison of different existing products and interviews with instructor pilots and Leonardo AD engineers to collect requirements.

2. Definition of a software architecture that can cover all the functional requirements collected and non functional requirements specific to Leonardo AD needs.

3. Implementation of a basic set of functionalities to show the capabilities of the designed architecture.

Chapter 2

Requirements

The requirement phase started with the analysis of several software solutions, the purpose was to identify a set of categories in which IOS functionalities are divided and which of them are in common between different simulators.
Most of the solutions seen were addressed to aircraft but there was also the opportunity to see products for helicopters.
In details: two IOS for the Alenia C27J Spartan, one developed internally by Leonardo AD and one acquired from L3 Brashear, an IOS developed by Leonardo Helicopters, that has also a tablet version, and some documentation about the IOS used to train Eurofighter and M346 AMI pilots.
All the Leonardo's IOS, both for aircraft and helicopters, were seen in action in FFS and FTD.
After an inspection of software and documentations the following functional categories have been identified:

1. Map & Generic Controls
2. Reposition & Time
3. Aircraft Settings & Fuel
4. Failures & Circuit Breakers
5. Weather
6. Payload
7. Ground Controlled Approach (GCA)
8. In-flight Refuel
9. Communications

The first 5 categories are common to all IOS even between aircraft and helicopters. Payload is relevant only for aircraft and helicopters that can bring a payload (people or equipment). GCA and In-flight refuel are functionalities specific to military aircraft. The Communications section can be more or less complex depending on the simulator equipment (headset, radios etc..).
Each category groups together one or more functions that operates in a same context. After comparing the various software GUIs an initial set of mock-up has been produced, both on paper and with MyBalsamiq software tool.
Mock-up have been shown to instructor pilots in order to be better refined in terms of functionalities and GUI usability.

2.1 Instructor interviews

Four instructors have been interviewed, all of them involved in the training of C27J pilots and with experience also on other simulators.
The interviews were repeated more times, after each interview mock-up were modified according to instructors advices.
During this process related features have been grouped together to divide the IOS interface in pages. Different options have been presented to instructors for menus, status-bar, controls and pages layouts. Instructors feelings about the existing IOS have been collected, in detail what they appreciate, and what they want to change. The interviews have made a fundamental contribution in the design process, allowing to discover many critical issues.
There are some informations and controls instructors want to have always available in all pages, those are: date and time of the simulation, a list of active failures, buttons to start, stop, pause the simulation, take snapshots, and eventually a list of those parameters that are not available in the aircraft instruments.
Informations to be displayed are related to the location of the IOS. If the station is separated from the aircraft cabin reproduction, instructors can't see the cockpit so flight instruments shall be replicated in the IOS.
Instead if the station is placed right behind pilots, instructors prefer to look directly at the simulator cockpit rather than watching instruments in their digital reproduction. This scenario apply to simulators for fighter jets and cargo aircraft. Fighter's cabin can usually accommodate only a pilot so the exercise has to be followed from outside, while in cargo aircraft simulators the IOS can be placed right behind the pilot's seats, so the instructor can see directly the cockpit.
Features that are common to all IOS may have a different implementation depending on the type of aircraft. For instance, snapshots are common to all IOS, they allow to restart the simulation from a given time. In most aircraft may be enough to have a button to take a snapshots once in a while. Instead in fighters shall be present a buffer that record last minutes of flight, so in any moment the simulation

can be rewind.
The GUI shall be designed with care, an interface that is good on desktop may not be easy to use on small touch-screens, this problem has been introduced recently with the spread of tablets, when existing desktop interfaces were ported on those devices.
Tablets are appreciated by instructors because they give them freedom to move in the simulated aircraft cabin, without losing control over the IOS interface. However, controls that occupy a whole screen on desktops may need to be split in more pages on tablets to make them usable.
Also touch controls such as sliders and wheels shall be designed with care, sometimes instructors have problems in setting right parameters because touch controls have a wrong sensitivity.

2.2 Functional categories

After the interviews for each category a list of requirements has been produced and a GUI layout has been designed.
In this section will be presented a general description of features present in each functional category identified.

2.2.1 Map & Generic Controls

This section displays a georeferenced map with a marker to show the current aircraft position and heading. The map may have different layers such as: terrain, airways, VFR.
Layers can be shown or hidden depending on exercise phase, for instance during landing the instructor can display a layer with the approach procedure, while in flight the terrain map can be more useful.
Then a monitor section for parameters of interest may be present depending on the exercise and instructor preference. In case of FFS, for safety reasons, are also available controls for the actuators and any other component that can potentially harm pilots.
If the simulator reproduces also aircraft sounds there are switches to set the volumes.

2.2.2 Reposition and Time

All IOS shall have some functionality for aircraft reposition, most common are by airport and runway selection, by providing ICAO codes or by selecting a custom position over the map.
Then may also be present a set of in-flight positions in proximity of runways that

allow to quickly repeat landing procedures.
A specific functionality desired by instructor is the customization of those positions by acting on aircraft heading, speed and altitude.
Finally, the instructor shall also be able to set date and time in the simulation to train pilots with different light conditions, choose the season and the moon phase.

2.2.3 Aircraft Settings & Fuel

Those are controls specific to the simulated aircraft, so this section can have different features depending on the aircraft type. Otherwise some settings are almost present in all IOS such as the selection of fuel quantity per each tank, buttons to refill consumable fluids, selection of icing conditions, controls to manage the engines start-up and ground procedures.

2.2.4 Failures & Circuit Breakers

This section allow the instructor to train pilots for the occurrence of a malfunction or more of them, such as engines fires, defective landing gears etc... In simple IOS there may be only buttons to activate or remove malfunctions, in most complex ones the instructor can specify sets of malfunctions to be activated together or define activation triggers, such as the overcoming of a certain speed or altitude.
In the circuit breaker section there is a graphical representation of all the cabin circuit breakers that the instructor can trigger. Circuit breakers are specific to an aircraft cabin so this part can change between simulators.

2.2.5 Weather

This section allows to control all aspects of weather such as clouds layers, clouds coverage, wind, gust direction and intensity, presence of storms with rain snow or hail, condition of the runways surface, definition of map areas where certain weather conditions apply. According to instructors this is one of the most complex section to operate, because of the number of parameters that can be set up, and the objective complexity in reproducing all weather conditions in a simulated environment.
In this section can also be present switches to set lights parameters of runways since instructors change them according to weather.

2.2.6 Payload

This section offer controls to set a specific payload configuration, move the centre of gravity, monitor the weight, the interaction can involve customization of payload by providing the type of loaded items and their disposition inside the aircraft or, in simpler solutions, buttons corresponding to predefined configurations.
If needed may also be present controls to emulate the Load Master activities such as opening/closing of cargo doors and load release.

2.2.7 Ground Controlled Approach

In this section the instructor can monitor the aircraft descent during landing, his horizontal and vertical deviation from the defined track. In practice the instructor impersonates the air traffic controller, providing instructions to pilots and guiding them in a correct approach path.
Sometimes the GCA is recorded by the IOS and plotted to a graph, so that it can be shown to pilots during the debriefing.

2.2.8 In-flight Refuel

This section is present only in simulators for military aircraft with the capability to be refuelled in-flight. The instructor can choose a type of refuel tanker, reposition the tanker in the simulated environment, set the light procedure, define a path that the tanker has to follow, control the drogue movements, start and stop the fuel flow.

2.2.9 Communications

In this section there is a list of navigational aid, with their type and status, the instructor can enable or disable them.
In FFS or FTD simulators that have a complete reproduction of the aircraft cabin there is also a section to control all the radios and pilot headsets, enable and disable the communication between 2 or more headset, choose communication frequencies, turn up and down headset volumes.

2.3 Engineers interview

The IOS is used also by avionic engineers to develop aircraft features.
Their needs are completely different from the pilots ones, usually the IOS section dedicated to engineers allows to make modifications on every simulation variable even the one that are not related to cockpit and deck controls.
This part of the IOS if present is hidden from the main interface because can potentially compromise the simulation. In this peculiar case where the IOS shall be able

to control multiple simulators this section will contain also all the configuration parameters that allows to change the appearance of the instructor interface and the simulation variables linked to the GUI.
Avionic and Aeromechanic Engineers have been interviewed to discover which features they expect from the IOS.
The main requirement is to have available a list of all simulation variables, the list shall allow to select a subset of parameters to be recorded and monitored on screen.
The collected data must be parsed in a human readable form, this involve the creation of graphs where variables are shown in function of time or other recorded parameters (eg. speed in function of altitude).
Plotting parameters such as: variables to plot per each graph, type and unit of measure of graph axis, must be configurable by engineers.
The IOS shall then allow to export the recorded data in a variety of formats, such as PDF for graphs or CSV for raw data that needs further elaborations.
There are many secondary requirements that will not be listed in the thesis, otherwise a complete requirement document both for engineers and instructor has been produced.

2.4 GUI design

At the end of the interview process both of instructors and engineers a GUI has been designed.
Depending on the number of controls categories have been divided in one or many pages. The designed GUI includes a menu bar and a status bar repeated on every page. The menu-bar is a vertical column with icons, it has been placed to the left border of the screen.
Instructors prefer to use icons than textual menus, since they are easier to locate inside the GUI, furthermore, on touch-screens icons occupy less space and are easier to click than textual menus. Each icon in the menu-bar correspond to one category in which functions are divided and it opens a different IOS page.
The status bar instead, is the place where are shown relevant information about the simulation and has been located on the top border of the screen.
The instructors asked also to have a set of controls always available, so the status bar has been split in two parts, a left side with the requested controls and a right side where are shown simulation informations and alerts.
At the centre of the screen are displayed IOS pages relative to a selected menu category, if a category requires more pages a tab menu is provided to move between them.

Figure 2.1. An interface sketch-up made with MyBalsamiq

Chapter 3

Architecture

In this chapter, various IOS's software architectures are discussed, then is presented a list of architectures and programming languages considered for the development and finally the chosen architecture.
Will be made references both to web frameworks and Leonardo AD Simulation Framework, to distinguish between them the latter will be always referred as Simulation Framework or Framework with first capital letter.

3.1 Existing architectures

This section will make an overview of the architectures regarding software solutions analysed, but in order to protect Leonardo's intellectual property is not possible to specify many implementation details.

3.1.1 Leonardo Aircraft Division

The IOS developed by Leonardo AD is in operation on C27J Spartan simulators. It controls both FFS and FTD simulators.
It is written in C++ and Qt. Most parts of simulators are written in C, because it is a language close to hardware and the main choice for real time applications.
The Qt library was chosen because it allows the creation of multi-platform GUI making the IOS compatible with both Linux and Windows machines.
The software is based on a Client/Server architecture, the IOS (client) receives and sends data to the Framework (server).
In this peculiar case, the communication does not happen with a network protocol but data is read and written on a shared memory.
The current Framework change this behaviour moving the data exchange to the TCP/IP network protocol.

3.1.2 Other IOS software analysed

For the other IOS software analysed can't be made assumptions on the programming languages or libraries used to build the GUI.
Although can be said that they all share the same client/server structure, where the exchange of information happens through TCP/IP or proprietary protocols.
They are usually targeted to a specific operating system with a fixed configuration, such as a predefined number of screens with a defined form factor and resolution.

3.1.3 Non functional requirements

For the development of the IOS a set of non functional requirements has been defined, those requirements have influenced the architectural choices so they will be listed below.
Requirements derive from research projects, where features are tested before their integrations on aircraft. Those projects operates on a variety of aircraft, the IOS shall adapt to them, and the interface shall be quickly editable to allow the management of tested features.

- The IOS shall interact with the new Framework Leonardo AD is developing.
- The IOS shall be configurable allowing to control different simulators.
- The IOS configurations shall be applicable without making use of programming skills.
- The IOS shall include an administration section, accessible only by the engineering department, where the IOS pages can be customized.
- The IOS shall make use as far as possible of open source software to contain costs.
- The IOS will be in operation in restricted environment with no internet access, this must be considered for what concern the software installation dependencies and the external resources needed by the implemented functionalities.
- The IOS shall support different screen sizes and resolutions.
- The IOS interface shall be usable both on desktop platforms and tablets.
- The interaction with the IOS can happen by mouse/keyboard or touch screen.
- The IOS interface shall operate on the largest number of devices and OS (Windows, Linux, Android, Apple iOS) and at least on Windows and Linux.
- The IOS shall be modular allowing to add new functionalities when needed.

3.1.4 Conclusions

The current IOS architecture offer some advantages:

- The interaction with the server is straightforward because there is a direct link with the Framework.
- C++ is a programming language that makes a good use of hardware resources.
- The client/server architecture is adequate and simple.
- It is easy to install the software in an off-line environment, where tracking and installing dependencies is not an easy task.

Although it has some disadvantage, changes to the IOS requires recompilation of code and installation on clients.
Different Operating Systems means different installer to be produced.
Even if Qt allows to design cross platform GUIs, they do not scale well with different screen sizes and two GUI version, one for tablets and one for desktops have to be produced.
Qt is not completely free, Qt technology has been sold and acquired multiple times, since 2016 the library is maintained by the Qt Company. To develop commercial products and have a full set of features, licenses must be acquired.
GUI development allows great flexibility although buttons, switches, sliders have to be customized by programmers, the choice of predefined GUI widgets is limited.
There is not a framework behind that offer built in functionalities such as interfaces for management of users or data, everything must be developed in C++ from scratch.

3.2 Chosen architecture

To overcame the limits of the current architecture the choice was to develop the software as a web application.
The IOS interface is shown in a web browser, this implies multiple advantages, the client side of the IOS is independent from the operating system of the device.
Any device that supports a web browser can show the IOS.
Web technologies allow the creation of flexible interfaces that dynamically adapt to different screen formats and resolutions. There is no need to reinstall the IOS on client devices, modifications made on server will be immediately available.
The disadvantages are an increased complexity of the architecture and a slower response time, although since the IOS shall not be a real-time application the compromise is acceptable.
Despite what happens in simulators, where delays are perceived by pilots, if the IOS latencies are in the order of tens of milliseconds instead of microseconds the

behaviour will be transparent to the instructor.
Although, there is still a problem that must be considered.
Requests and responses in a web application are usually performed on top of the HTTP protocol that is synchronous. The user interacts with the interface displayed in the browser, the browser makes a request to the server that sends back a response. This can be a limit for the implementation of the IOS. Lets imagine a scenario where the aircraft is moving on a map, if the instructor wants to see the updated position he will have to periodically refresh the IOS page to obtain the new coordinates.
Another problem can arise if the IOS is open on two different devices, for instance a tablet and a desktop computer.
In this case if a user modifies a parameter on a device the change will not be immediately visible on the other, a page refresh is needed.
This behaviour is not acceptable and there are 2 technologies that can be used to overcome those limits, AJAX and WebSockets, both have been used in the development of the IOS since they suit different use cases.
The IOS needs also to store and retrieve data, for instance the instructor may desire to save a set of custom coordinates for the aircraft reposition during an exercise.
There are many database solutions available both free and licensed, otherwise since the IOS must be as flexible as possible sticking to a single product may be a limit for the future, so support for ORM has been considered in the architectural choices.
AJAX, WebSockets and ORM are niche technologies that may not be familiar to the reader so a brief description is provided.

3.2.1 AJAX

Is an acronym for Asynchronous JavaScript and XML, the first description of the technology was published in 2005.
It is based on JavaScript and uses the *XMLHttpRequest* object to perform asynchronous requests.
AJAX does not define a protocol of his own, the interaction is built on top of the HTTP protocol.
Data is usually exchanged in XML or JSON format. While data is received the user is not blocked and can use the GUI.
The main use of AJAX is for updating small portion of the page dynamically, it is suitable to use when on server side is already present data that must be sent asynchronously. When data is generated on the fly and must be streamed to the browser a better alternative is to use WebSockets.

3.2.2 WebSockets

They define a protocol used to establish a full-duplex channel between the user browser and the server.
The API has been standardized by the W3C and the protocol by IETF (RFC 6455) in December 2011.
Full support from all mayor browser including mobile ones came only at the end of 2013. While this can be a problem in mainstream application, it is not a limit in the IOS environment where the browser is chosen by the system administrator.
The WebSocket protocol is different from HTTP, when a WebSocket connection is established an "upgrade request" is received by the browser that provide to switch to the new protocol.
The WebSocket API provide methods to connect, disconnect, send and receive messages. Connections are based on TCP protocol which is unicast, otherwise most high-level implementations of the protocol in WebSocket Servers allows to send data to multiple clients at the same time.
In theory WebSockets can replace all the AJAX functionalities and offer new capabilities, otherwise they must be supported both from the user browser than the server back-end.
In practice both technologies have their advantages and disadvantages, they coexist and are used for different needs.

3.2.3 ORM

Acronym for Object-Relational Mapping, are systems that allows to map classes from programming languages to SQL tables, the ORM take care of creating the right tables and relationships among them.
The programmer does not write SQL queries but reads and writes data to mapped classes.
ORM make the application independent from the underlying SQL database.
If the database technology is changed the ORM software will rebuild all tables and relationships.

3.3 Implementation analysis

A web based application with support for WebSockets should allow to design a fully functional IOS.
Although implementing everything from scratch is not a good solution since there are many web framework which provide a basic set of functionalities that can be extended.
Integration with the Simulation Framework must also be taken into consideration for the final choice. There are currently 3 API that allows the connection with

Framework. They are written in C, C++ and Python.
Many solutions have been explored, here will be presented a small summary for each analysed platform.

3.3.1 Spring MVC

The most popular web framework based on Java [5], it offers many APIs and rich documentation, ORM is supported by interaction with external libraries such as Hibernate. WebSockets are fully supported.
The disadvantages are related to the Java world, the JDK must be installed in the development environment and a servlet such as Tomcat must be present to run the application. After changes code must be recompiled and the application redeployed to the servlet.
Also the way libraries are fetched can be a problem. The preferred option to resolve dependencies is by tools such as Maven and Gradle that automatically recover needed libraries from internet, although our development environment will be off-line.
Of course there are ways to configure those tools to use local repositories, otherwise in Java web development is common to use tens of different libraries, fetching all of them manually is a long task.
Furthermore there is not a native framework API in Java and integration with C++ can lead to compatibility issues.

3.3.2 Laravel

The most popular framework for PHP development [6].
The basic requirements to develop with Laravel is having installed PHP7 and few other PHP extensions.
There is built in support for ORM and WebSockets. Dependencies are resolved by composer, a tool that download and install needed packets from internet, although there are ways to configure it to use a local repository.
Laravel offers also a built-in development web server that can be used to run the application without installing a dedicated one, such as Apache.
The only drawback is the lack of a native API for the Simulation Framework.

3.3.3 Ruby on Rails, Node.JS

The first is a framework that uses the Ruby programming language [8], it is a niche product with few documentation, for our usage does not offer advantages over Laravel and Spring so has not been considered a valid choice.
Node.JS is a JavaScript runtime environment [7]. It allows to use JavaScript as a server side language.

The advantage is in having a single language suitable both for the application front-end and back-end, although Node.JS is a recent technology, still gaining popularity. There are web frameworks based on Node that offer all the features needed for the IOS, but there is not a solution that has imposed as market leader yet, choosing one is risky for long time support.
Furthermore JavaScript gives great freedom in language syntax, this speeds up the development time but make it easy to write bad code, decreasing the software maintainability. For all those reasons even this solution has been discarded.

3.3.4 Xamarin

That is not a web framework but is worth mentioning.
It is an environment developed by Microsoft that allows to crate native user interfaces for Windows, Android and iOS that share the same C# code [9].
C# can integrate C++ code, so is possible to interact with the Simulation Framework. Xamarin is free to use and the SDK is released under MIT license.
Unfortunately, there is not Linux support yet, both for the generated applications than for the developing environment.
All the simulator department use Linux as main operating system and a switch to Windows is not cost effective.

3.3.5 Django

The most used web framework based on Python [4], the basic requirements to run Django is having installed python 2.7.
ORM functionalities are built in, the preferred way to fetch external Python libraries is through *pip*, a tool that can download and install packets from internet or local repositories, the advantage over Spring is that most functionalities like ORM are built in the framework, so few additional packets are needed.
Django like Laravel provides an integrated web-server for development. WebSockets are available with an external library called Django Channels.
The Simulation Framework can be accessed directly from the Python API already available. Python like PHP is an interpreted language, not requiring to recompile code.

3.3.6 Conclusions

The final choice is between Django, Laravel and Spring.
The question to answer is if there is something relevant to the IOS development, that makes Laravel or Spring a better solution and justify the effort of using the Framework interface without a native library.
The answer is no for Laravel that offer almost the same functionalities, but a comparison to Spring is more complex. Spring does not include only a web framework but is a collection of projects that cover a wider range of technologies, so it may be more suitable for future IOS developments.
Otherwise this flexibility comes at the cost of a more complex architecture a less agile development, due to the Java language verbosity, the need to recompile code after changes and a more difficult integration with the Simulation Framework.
In the worst case a new Java API had to be written and the effort does not justify the advantages.
Those are the main reasons for which Django has been chosen as the development platform, although there are also other benefits that will be presented in the next chapter where is provided a deeper description of the Django environment.

Chapter 4

Django, description and customization

In this chapter is described the Django framework, the MVC architecture that is at its core and the peculiar features that make it a good choice for the IOS development.
A complete description of the framework is out of the thesis scope, only informations about features used in the IOS or that are interesting for future developments are provided.

4.1 MVC pattern

This programming pattern has been introduced for the first time with the Smalltalk language in 1979. It is the architecture adopted by most web framework and in general the most common pattern used for web development.
Django makes no exception splitting the application logic in 3 parts:

1. **Model**: The component that manages the data, not the real data but the application modules that interact with it. In Django they are a set of files called *models.py*. Those files using Django classes and methods define the structure of the database that the ORM system translates into SQL tables.

2. **View**: The presentation layer, it defines the user interface. In Django those are the templates files. They are standard HTML files where a template language, made by a set of tags, is used to exchange data with the Controller.

3. **Controller**: It connects models with views, controlling the flow of information. In Django consists of a set of files called *views.py*.

Because of this naming conventions, where Controllers are called "views" and Views are templates, Django architecture is usually referred as MTV.

4.2 Project structure

Before proceeding into the description of a Django project structure is necessary to specify what in Python is a module and what is a package.
A module is a single Python file that may contain one or more functions.
A package is a collection of modules inside a folder.
A Django project is made of a packages collection, each package can be constituted by several modules.
There is a primary package where also application settings are defined and secondary packages usually referred as App. An App is a unit of the whole application. The Django project made of the primary package and of one or more Apps is then placed in a root directory.
Besides models, views and template files, there are two more mandatory files peculiar to Django framework: *settings.py* and *urls.py*, whose content is described in next sections.
By now can be said that if no coupling is maintained between Apps, parts of the application can be added or removed simply by commenting few lines in the *settings.py* and *urls.py* files placed in the main package.
This approach has been followed in the IOS development, where there is a set of packages that are the backbone of the IOS.
When an additional functionality is needed, it can be added to the project as an App and then linked to the main package by the settings and urls files.
Apps can be deployed on any webserver that supports the WSGI protocol. Or run directly with the integrated web server.
In a basic set-up, the application is made of a folder that contains the SQLite database (a file) and the Django project. This makes possible to run the application without installing a DBMS software or a dedicated web server, this increase the productivity because developers have not to deal with databases and servers configurations.
Having an application that runs from a local folder with few external dependencies is an important advantage in the simulator environment where software installations and system changes are limited and must be done only by authorized personnel.
When a new project is created (by an IDE or from the operating system console) in the root directory is generated also a *manage.py* file that gives access to a set of commands defined by the framework, it allows to perform several actions, such as launching the integrated web server, build, dump, load or migrate the database.

4.2.1 settings.py

This file contains all the information needed for the interaction between the main App and all the other Apps both internal than external to the project. It is unique and located only in the primary package.
Relevant sections of *settings.py* file are the following:

- **Installed Apps**: It is a list of all Django's packages used in the project, a package can be an internal app or an external library.
  For instance in the IOS project here are listed all the apps which constitute the application and the Channel package that offers support for WebSockets.

- **Middleware**: Here are listed all components that interact with requests an responses before they are delivered to a view.
  A common middleware performs user authentications, is provided in the Django default configuration and is used by the IOS to give different access privileges to instructors and engineers.

- **Databases**: Here are listed all the databases used, they can be more than one. If a database is changed the ORM system allows to migrate the data quickly by giving few commands scripted in the *manage.py* file.
  Since only the ORM interacts directly with the database tables, no changes to the application code are needed when a database switch is made. Supported databases are: SQLite, PostgreSQL, MySQL and Oracle products.

- **Static Root/Static Dirs**: Those settings tells Django where static files such as CSS and JavaScript library have to be placed.
  A command scripted in the *manage.py* file provide to collect static files from packages folders listed in Installed Apps.

The *settings.py* file can also be customized by adding additional section, this feature is used in the IOS to enable or disable the Simulation Framework integration and change Map's settings.

4.2.2 urls.py

This file define the set of URL used by the application, each URL is mapped to a view function or class, which in turn serves a template.
References can be made recursively. A root *urls.py* file must be placed in the primary package. It can refer to secondary *urls.py* files defined in other Apps, this approach reduce the overall coupling.

4.2.3 Channels

Django does not offer WebSockets support out of the box, although it can be added with a package called Channels.
When Channels are linked to the application, two more files are needed to enable WebSockets: *routing.py* and *consumers.py*.
First one defines the urls on which channels are listening for connections, in a similar way as *urls.py* does for standard HTTP request.
The latter specify the logic to apply when a connection/disconnection occurs and when a message is received. In this file can be overwritten three methods: *connect*(), *disconnect*() and *receive*(), allowing to customize the application behaviour when those events are triggered.
Connections can be divided in groups, and thats the approach that has been followed. For each module in the IOS that needs asynchronous messages a channel group has been defined.
When a message is sent to the users browser, one or more target groups can be specified.
Channels are integrated directly with the provided Django development server, otherwise for application deployment is possible to split the incoming traffic in HTTP and WebSockets requests that are managed by different servers. Using a dedicated solution increase the performance, this option shall be considered if the IOS complexity grows in the future.
On client side, channels messages are received with the standard WebSocket JavaScript API and the output is shown on web pages.

4.2.4 Admin pages and GeoDjango

Django by default offers a web interface for the application administrators.
It is divided in sections corresponding to the various Apps that compose the project. For each App there is an admin page where is possible to interact with the models defined, by creating, deleting or updating items. Model pages are displayed as tables where each element occupy a row.
Tables can be also customized by modifying the arrangement in which items are displayed or by adding functionalities such as import/export or search fields.
GeoDjango is a GIS extension for Django that comes with the default distribution and integrates an API to work with geographical data such as shape files.
To use the GIS functionalities a database that support geographical types such as PostGIS or SpatialLite must be installed and linked to the application.
The GeoDjango API has not been used in the IOS but is an interesting feature for future developments, allowing to extend the set of functions related to maps and aircraft reposition.

4.2.5 Templates and Bootstrap

Templates are HTML pages with special tags used to display and elaborate informations coming from server back-end, each App has a template folder where all those files are collected. Django defines its own template language that has been used for the IOS development.
A relevant feature of template languages is inheritance, common front-end parts can be defined in a base template and this file can be extended in each package to provide specific functionalities. This feature is used in the IOS to define once all the common parts such as menus, status-bar, import of CSS and JavaScript libraries.
The base template links also all the dependencies needed by the Bootstrap tool-kit.
Bootstrap is a collection of CSS and JavaScrip files, originally developed by Twitter and now free to use and distributed under MIT license. It allows to quickly build responsive web pages providing a set of CSS classes used to style HTML elements.
Web pages are divided in rows and each row can display up to 12 columns.
This scheme simplify the creation of pages that dynamically adapt to different screen sizes.
During operation the size of the browser window is detected and a different number of columns per row are displayed.

4.2.6 Maps and geocoding

Usually IOSs provides maps, they can be used to display the actual position of the aircraft or to perform repositions.
To make this IOS as flexible as possible and decrease development times maps are not designed as a built-in functionality, maps are displayed with the OpenLayer JavaScript library that has been imported inside the project.
OpenLayers allows to put dynamic maps in web pages, it can interact with any map server that offer an interface to the library, it allows to display tiles from tile servers or directly render vector data in a variety of formats.
Each layer can contain a different data source. Layers can be displayed with different level of transparency or hidden.
There are also features to create points, draw lines and poligons, make animations, extract geographical coordinates from the loaded maps.
The IOS currently uses two layers, one to load a set of tiles from a remote map server and one to display the aircraft position as an icon.
During flights the icon is moved according to latitude longitude and heading coordinates coming from the simulator.

Map data and map servers

As stated before the IOS shall work in an offline environment, this puts several limits to the available choices in terms of map servers.
Google, Microsoft and Esri offer maps suitable for the IOS needs, otherwise they do not allow to export maps for off-line usage without a commercial license.
The OpenStreetMap project provide several Giga Bytes of geographical data covering all the world, although data is in a raw format, to be displayed on a chart must be converted, imported inside a geographical database such as PostGIS and a server must be set-up.
The server should install a software for rendering maps from the PostGIS data.
Mapanik is one of the most used tool for this operation and is released under LGPL license.
If the tile-server is a machine different than the IOS server, further software is needed to make maps accessible in the network by means of HTTP protocol and OpenLayers interface.
For this purpose, can be used Apache with the mod_tile extension or a more specific solution, there are several free tile-servers available on the market.
All those steps are expensive in terms of computing power, storage and time for having all the architecture correctly set-up and running.
For the thesis purpose a simpler solution has been found in the OpenMapTiles project that offers a collection of pre rendered tiles and a preconfigured tile-server.
Tiles are free to use for evaluation and non-commercial purpose, they are built from the OpenStreetMap data and include a subset of the available geographical information, they lack features such as hill-shading and contour lines, they are good for a basic usage, although if the IOS will become a commercial product, the set-up of a proper tile-server should be considered.

Chapter 5

Simulation Framework integration

Before discussing how the IOS has been connected to the Framework is necessary to explain what the Framework does and how it works.
During a simulation mathematical models read and write data on memory areas.
Each simulator has a set of memory areas, divided in several fields that identify a specific aircraft parameter.
Fields may have different data type, they can be integer float or arrays of values.
Simulators may have tens of memory areas, each one identified by a name and a version number. Each area may contain hundreds of fields in which input and output data is written.
The data structure that contains the fields for a specific memory area and version is called Interface Control Document (ICD). When a simulator is configured an ICDs database is created.
A legacy Framework is in operation on existing simulators, it uses special hardware to manage memory areas and is no longer maintained.
A new Framework is under development, it will be installed on new projects and is the solution the IOS will use.
It moves the interaction with memory areas to the TCP/IP protocol, making easier the integration of web technologies.
It is composed by a master node in charge of executing operations on areas and models, and by one ore more slave nodes that can exchange data and execute commands on the master.
The Simulation Framework has two main functionalities that are of interest for the IOS:

- It can manage the models involved in the simulation by starting, pausing and stopping them.
  For instance when the instructor performs a "Freeze", one or more simulation models have to be paused.

- It allows to read and write variables in memory areas.

There are memory areas for reading outputs from the simulation and memory areas for writing values in input to models. Some actions performed by the IOS may require using both functionalities together, some simulation data can be elaborated by models only after a freeze or a stop.

A memory area field is identified by a unique id, those ids are not consistent across different simulators, if the IOS uses directly fields ids, each time a new simulator is connected the code should be changed.
To overcome this limit data structures have been designed to map each Framework id of interest into an IOS id. In this way, when a new simulator is linked to the IOS, will be enough to load a new transition table to write and read data from memory areas.
To perform the mapping four tables have been defined as Django models:

- **IosName**: A list of all the names used inside the IOS code to read and write in memory area fields, each name is unique.
- **ICD**: The collection of all ICDs of interest. Each entry is identified by a triple: the area name, the area version and the id of the area field.
  The triple allows to store ICDs from different simulators inside this table.
- **IosRead and IosWrite**: They have the same structure, they map an IOS name to an ICD triple.
  As the name suggest one table is for reading data from the simulator and the other for writing data to it.

In principle, a single table with the same structure as *IosRead/IosWrite* would have been enough to perform the mapping.
In practice working with thousands of rows is easy to do mistakes.
With a single table an error will be discovered only at run time when a read/write occurs, and who configures the IOS shall know in advance the field names of ICDs to create a correct mapping.
This design instead will detect most errors as soon as the data is loaded in the application.
The *IosName* data is maintained by the programmer and is modified only if Python code is changed. ICDs can be loaded directly from the Framework when the IOS is connected, so they can be quickly refreshed if needed.
There is no limit to the number of ICDs that can be loaded, the loading can be performed one time per each simulator and then the table refreshed only when ICDs change.
The only tables that have to be modified per each simulator are *IosRead* and *IosWrite*.
When values are submitted, they are compared against the *IosName* and the *ICD* tables, so the user is notified if a mismatch occur and the wrong row is discarded.

The mapping division between reads and writes avoid mistakes both from the side of engineers that load data than programmers who define read and write functions.

5.1 Framework Manager API

In Leonardo AD is also developed and maintained a Python API that allows to write and read data to the Simulation Framework.
Otherwise because of the mapping mechanism and the use of WebSockets it cannot be used directly inside Django. A new set of functions has been written to perform easily writes to memory areas and stream readings to the connected clients.
The API is made of a set of methods defined inside a Python class called *FrameworkManager*.
FrameworkManager has been defined according to the singleton pattern, it is instantiated the first time it is imported inside the code, and any subsequent call will be referred to the same object.
By design to simplify maintainability the *FrameworkManager* class is the only point of contact between the IOS and the Framework API, any read or write must pass by its methods.
During the first (and only) instantiation a number of actions are performed:

1. Framework settings are read from the Django *settings.py* file
2. If the integration with the Simulation Framework is enabled a communication channel is opened and kept active for all the application life cycle.
3. All the different area names available in the *IosRead* and *IosWrite* tables are retrieved and the corresponding areas are opened in read or write mode.
4. If the management of models is enabled the IOS will read the list of models declared in the *settings.py* file, it will try to resume and stop all of them and set the IOS status to resumed and stopped.

After those steps *FrameworkManager* is ready to serve its methods.
Each method is defined as a Python "class method" and can be called with the syntax:

```
FrameworkManager.method_name(parameters)
```

In this section are described all the available methods, with their input parameters and the actions they perform.

5.1.1 Models management

Four methods have been defined to manage models, they act on the list of models declared in the *settings.py* file, they do not need any input parameter.

1. **pause()**: Pause models
2. **resume()**: Resume models
3. **start()**: Start models
4. **stop()**: Stops models

By the way the Simulation Framework works pause-resume and start-stop act independently, a model can be both paused and stopped, that means that if a paused and stopped model is started it will be in pause state.
The IOS uses two distinct variables to take track of models status, one for pause-resume the other for start-stop actions.
All the methods check the status of models after execution, errors are printed in the console.

5.1.2 Ios and connection management

Those are methods for retrieving informations about the IOS internal status and the framework connection.

1. **is_connected()**: Returns *True* if a connection with the framework has been established.
2. **get_server_hostname()**: Returns the hostname of the framework master node as a string
3. **get_start_stop_status()**: returns the start-stop status of the IOS as a string, respectively *"start"* or *"stop"*, it refers to a variable internal to the IOS and does not correspond necessary to the real models status (eg. if the integration with models has been disabled in the settings file)
4. **get_pause_resume_status()**: returns the pause-resume status of the IOS as a string, respectively *"pause"* or *"resume"*.
   Same considerations as the previous method apply.

5.1.3 Writings

Here will be listed all methods that perform writings on memory areas.

set_position(parameters)

This method sets all the memory area fields necessary to perform an aircraft reposition, it needs several parameters in input:

- **lat_dd**: latitude in decimal value as *degrees*
- **lon_dd**: longitude in decimal value as *degrees*
- **ft_m_sw**: a string, *"ft"* if altitude is expressed in $feet$, *"m"* if is expressed in $meters$.
- **altitude**: the desired altitude in decimal value, it can be expressed in $feet$ or $meters$, in Above Ground Level (AGL) or Mean Sea Level (MSL) value.
- **kts_ms_mach_sw**: a string, *"kts"* if speed is in $knots$, *"ms"* if speed is in m/s, *"mach"* if speed is expressed in $mach$
- **speed**: the desired speed as decimal value, can be expressed in $knots$, m/s or $mach$
- **heading**: heading (ψ) in $degrees$
- **pitch**: pitch (θ) in $degrees$
- **roll**: roll (ϕ) in $degrees$
- **ramp**: ramp (γ) in $degrees$
- **msl_agl_sw**: a string, *"msl"* if altitude is in MSL value, *"agl"* otherwise.

set_fuel(list)

This method sets the fuel quantity per each tank.
In input requires a list of fuel quantities expressed in *pound*, the list can contain at most 10 values.
Up to 10 memory area fields can be written by this method, each field correspond to the fuel quantity of a tank, depending on the simulator, if the aircraft has n tanks the first n memory area fields will be written.
An error is printed in the console if more than 10 fuel values are submitted, and no actions are performed.

set_date_time(datetime)

This method sets date and time in the simulation.
The input parameter is a Python *datetime* object. It must be initialized at least with year, month, day, hour and minutes values.
Errors are printed in the console if the *datetime* object is incomplete and no actions are performed.

write(list)

This method performs generic writes on memory areas, in input requires a list of lists, where each element is made of a string corresponding to an entry of the *IosName* table and a value to be written:

```
[[ios_name, value], [ios_name, value], ...]
```

The method will resolve the binding between the *IosName* table and memory area fields using the *IosWrite* table.
When a corresponding entry is not found in *IosName* or *IosWrite*, an error is printed in the console and that writing is skipped.

5.1.4 Readings

Here will be listed all methods that perform reads on memory areas.

read(list)

This method perform a generic read, in input requires a list, where each element is made of a string corresponding to an entry of the *IosName* table.
The method will resolve the binding between the *IosName* table and memory area fields using the *IosRead* table.
If corresponding entries are found a dictionary is returned, if a name is not found will return *None* and errors will be displayed in the console.
The returned dictionary will have as keys the strings submitted in input.
Following methods are more complex, they can be used to have a constant stream of data from Framework. Retrieved variables are routed to WebSockets and constantly refreshed on screen.
Three methods have been designed to make this operations as simple as possible, they provide to integrate the Simulation Framework API with Django Channels and shall be used inside the *consumers.py* files.
Those files are present in each module of the IOS that uses WebSockets.

init_websocket(group, ios_names, sleep)

- **group**: A string, is the name of a Django Channel group.
- **ios_names**: A list of IOS variable names, all the entries available in the *IosRead* table can be submitted.
- **sleep**: An optional float number.
  It is the time interval between readings expressed in *seconds* (eg. $0.5 = 500ms$). If not present is used the default value specified in the *settings.py* file

This function should be called at the beginning of *consumers.py*, outside the methods defined.
It creates a thread for the reader's group passed as argument. The thread provides to read values from the memory areas mapped to the list of IOS variables. Readings occurs at intervals of *sleep* seconds.
As soon as values are red, they are sent in JSON format to all clients listening to the Django Channel group. The JSON packet contains a dictionary, each entry is a pair *key: value*, where *key* is the IOS variable name.
The JSON packet can be red client side with the standard JavaScrip API for WebSockets.
The thread created by *init_websocket()* will exist for all the life cycle of the IOS.
By default will be in a paused state, this is done to preserve computing resources when not needed.
Following methods describe how to start the thread.

add_reader(group)/remove_reader(group)

In input they require the name of a Django Channel group as a string.
Those functions should be called in the *connect()* and *disconnect()* methods of *consumers.py* file.
The group passed as argument shall be the one present also in the corresponding *init_websocket()* call.
add_reader() increase a counter inside the corresponding thread.
remove_reader() decrease the counter.
Increasing and decreasing operations are thread safe respect to the counter value that will be always consistent. The thread will start running when counter is greater than 0 and pause again as soon as the counter returns to 0.
This ensure that threads runs only if there are browser windows opened on web pages requiring a data stream.

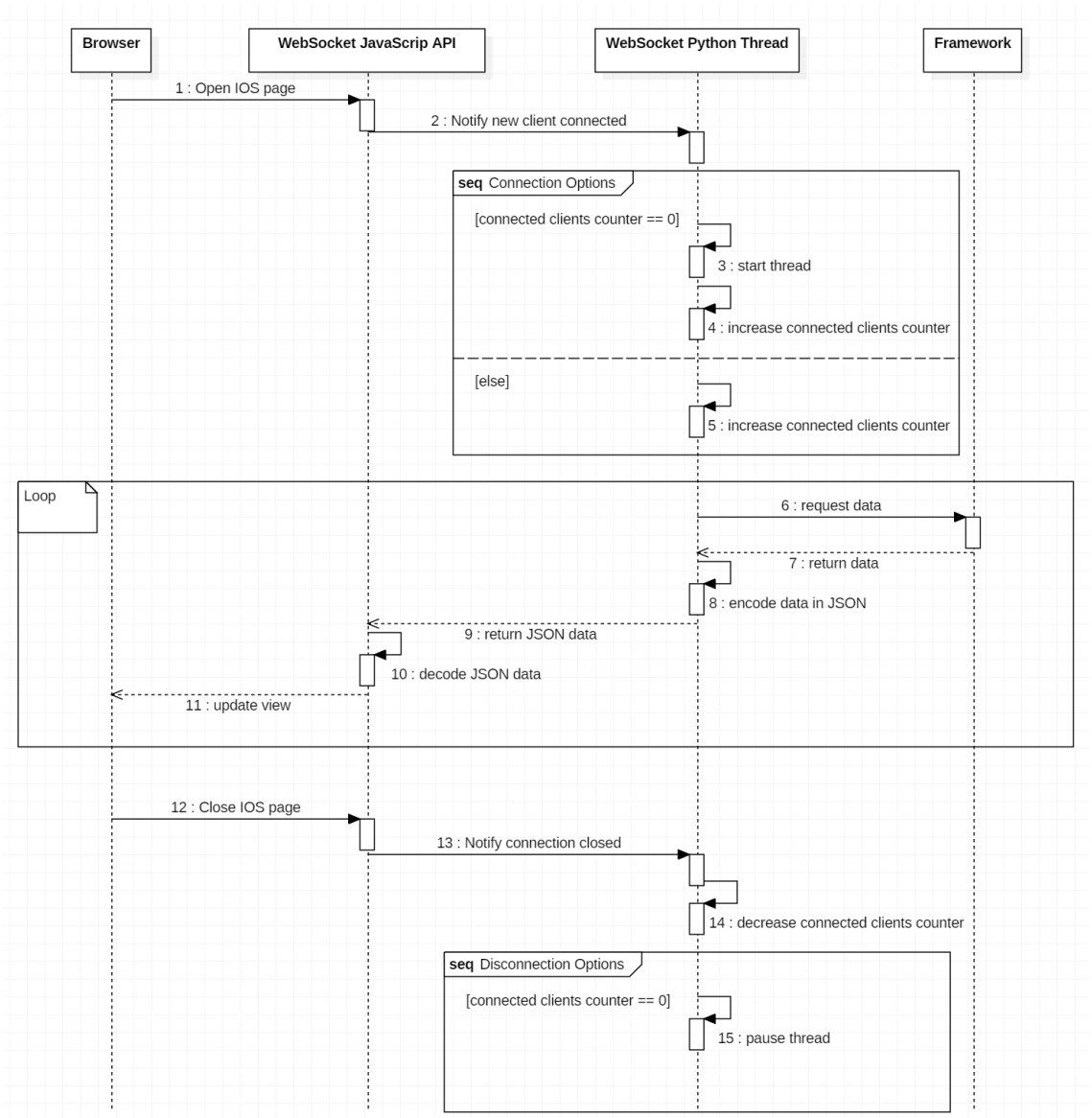

Figure 5.1. Sequence diagram (simplified) for a WebSocket thread

Chapter 6

IOS prototype

The IOS prototype implements a subset of the functional categories found in the first thesis part.
The application consists of 5 sections: Map, Reposition, Aircraft, Failures, Administration. Each section correspond to one of the identified categories, with the exception of the Administration part that provides controls for managing the IOS data, configure the software for different simulators and change sections appearance.
The application can run on the integrated Django web server, it does not require a dedicated solution. Formerly, for performance and security reason, the Django web server shall be used only in development. In practice the IOS will run in an offline environment and the only user will be the pilot instructor, so there are no particular concerns in using it.
Otherwise, if the application development will continue in the future, a switch to a more professional web server is strongly recommended, to achieve both better performance with WebSockets streams and a greater reliability.
SQLite has been chosen as the database platform, because it allows to store all the data in a single file inside the main application folder.
Those implementation choices make the application portable, it can be moved from one machine to another by simply coping the root folder, without worrying about databases and web servers configurations.

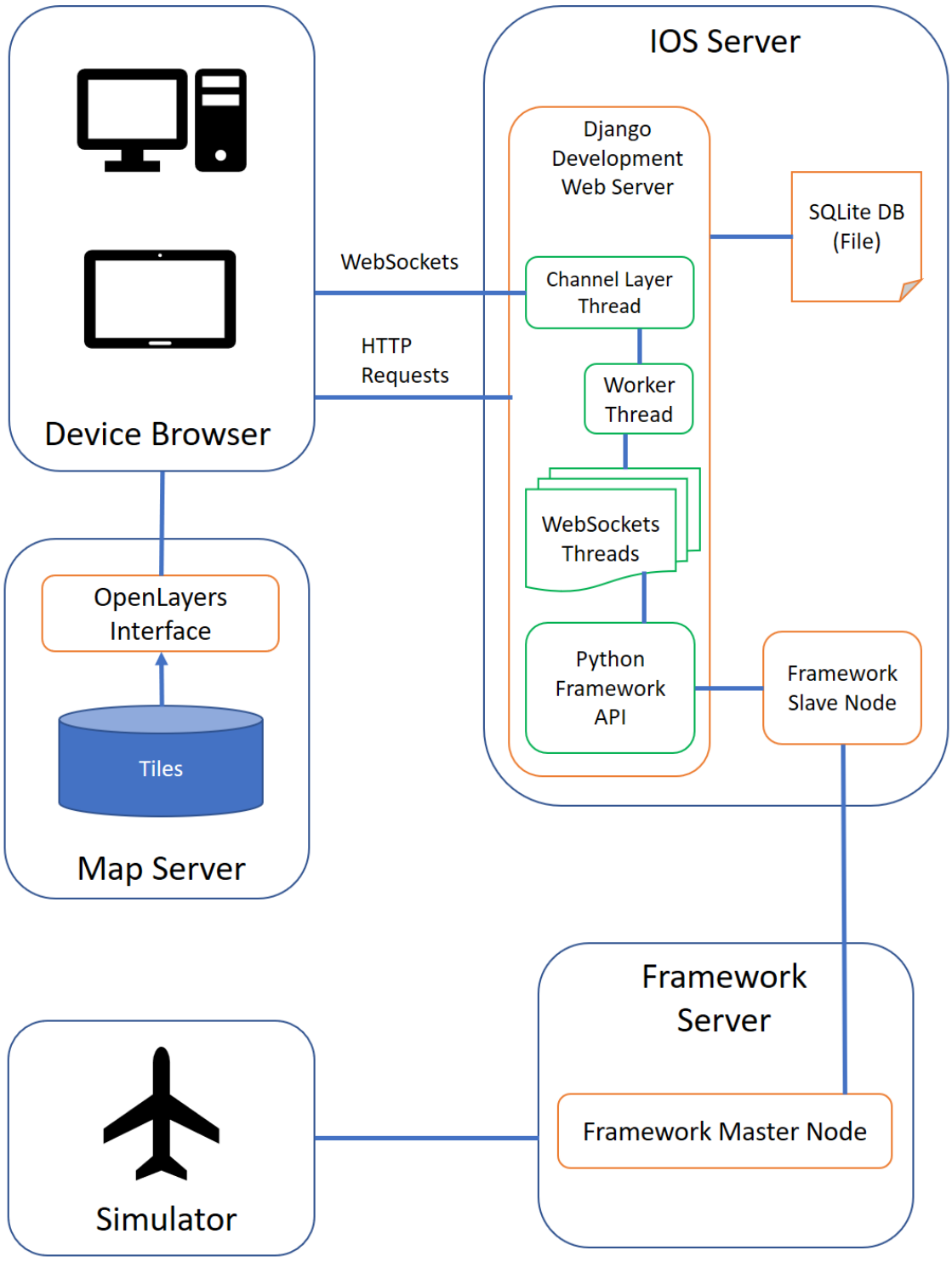

Figure 6.1. Current system architecture, optimize portability at performance expense, HTTP requests and WebSockets are managed by the Django development server

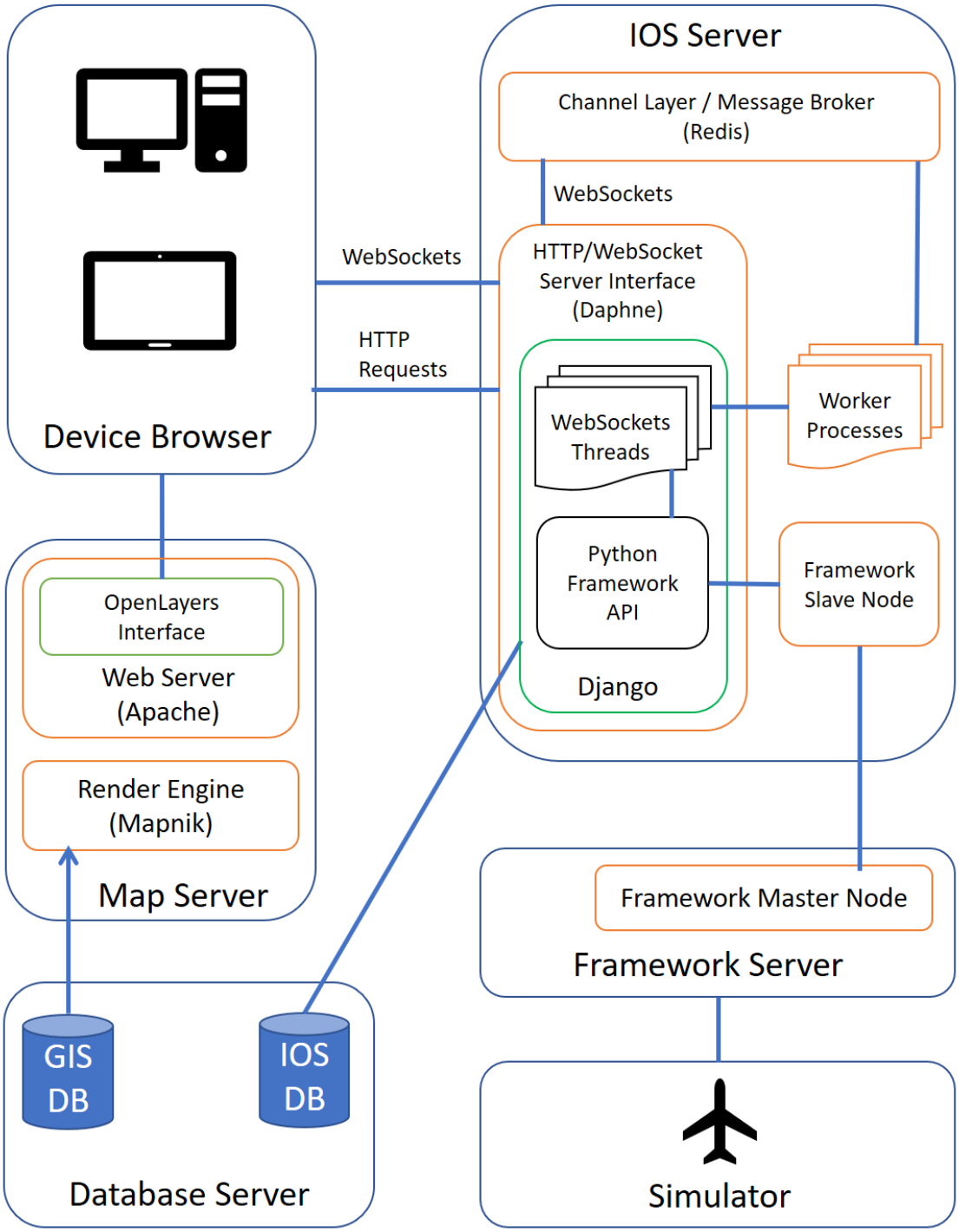

Figure 6.2. A possible system architecture for production, performances are greater, WebSockets are managed by a dedicated component (Redis), although the application will not be portable.

The application is packaged in a zip file with the following structure:

```
ios.zip
ios
    installation
    virtual_machines
    django_doc
    ios_root
        map
        reposition
        aircraft
        failures
        ios
        csv_export
        csv_store
        fixtures
        static
        framework_test.py
        manage.py
        ios_database
```

Proceeding with order: *installation* is the folder that contains all the libraries needed to install the IOS.
virtual_machines contains 2 virtual machines, one with a map server and the other with a preconfigured IOS.
django_doc is an off-line version of the official Django and Channels documentation.
The folder *ios_root* contains all the IOS files. After the installation steps it can be moved from one system to another.
Folders: map, reposition, aircraft and failures are Django Apps, they correspond to the first 4 sections in which the IOS is divided. The *ios* folder is the application core, it contains the Django *settings.py* file, the *FrameworkManager* API and the definition of data structures that allow to communicate with the Framework.
Folders *csv_export* and *csv_store* are related to the collection of CSV files necessary to pair the IOS with different simulators.
The *fixtures* folder contains a backups of the SQLite database in different configurations.
The *static* folder is a collection of all static files used by IOS pages (CSS, JavaScrip libraries, images..).
The last 3 files are: the database used at runtime (*ios_datatbase*), the python script that allows to run Django administrative tasks (*manage.py*) and a suit of tests for the Python Framework API (*framework_tests.py*).
As is evident there is not a reference to the Administration section, because it is not a Django App defined inside the IOS, but rather a customized version of the

administration functionalities already provided by Django.

6.1 Provided virtual machines

Two virtual machines have been set-up, one with an IOS and the other with a map server, the IOS is already configured to work with a simulator used for research purposes and known as Generic Regional Aircraft (GRA).
Virtual machines have been created with VirtualBox 5.1.28, they have IP addresses statically configured, *x.x.x.121* for the Map server and *x.x.x.120* for the IOS. If addresses are changed the IOS has to be reconfigured properly.
Credentials *.txt* files with login informations are available in VirtualBox directories.
The following procedure will launch a working version of the IOS, accessible on any node of the GRA network with a browser.
Firstly start the map server virtual machine, open a terminal inside it and give the following commands:

```
$ sudo iptables -F
$ tileserver-gl italy.mbtiles -b 0.0.0.0
```

Now the map server should be accessible on any GRA node using this URL:

```
http://<map_server_ip>:8080
```

Then the GRA Framework master node has to be started together with the model master and model scheduler processes.
Models have to be put on stop state.
When everything is working start the IOS virtual machine, open a terminal inside it and give the following commands:

```
$ cd $FRAMEWORK_PATH
$ ./bin/ncfs2_slave <gra_master_ip>
```

Open a second terminal and give the following commands:

```
$ sudo iptables -F
$ source ios_virtual_env/env/bin/activate
$ cd  ios_virtual_env/ios_root
$ python manage.py runserver 0.0.0.0:8000
```

Now the IOS shall be accessible from any browser in the GRA network at the address:

```
http://<ios_ip>:8000
```

6.2 Installation procedure

The application can be installed both on Windows and Linux, minimum requirements is having installed Python in version 2.7, GCC and the Python development packages, needed to build Python C extensions.
The Framework is necessary only if a connection with the simulator has to be established, in this case to exchange data the application requires that a slave node is running.
Otherwise for the purpose of loading data or configuring pages the IOS can be launched without an active connection.
In this section will be explained how to install the IOS on CentOS 7 since is the reference Operating System for the simulator department, installation steps should be similar on Windows systems, will only be necessary to download the Windows versions of the libraries available in the installation directory.
The procedure has been tested with CentOS 7.4 (1708) with GNOME desktop environment, and will illustrate all the steps necessary to have a fully working solution from a stock installation.
CentOS 7 already satisfy most of the needed requirements, the only missing packets are python-devel and gcc, they can be installed with administrator privileges executing the following commands:

```
$ sudo su
# yum install python-devel
# yum install gcc
```

Those are the only packets that have to be provided by the system administrator, and they are included in the official CentOS repositories. All the following steps can be executed in user mode.
Firstly environment variables have to be set, this can be done by adding a line to the *.bashrc* file located in the home directory.
Open a terminal and execute the following command:

```
$ gedit .bashrc
```

If *PYTHONPATH* is not already defined in the environment variables, past the following line at the end of the document, otherwise redefine the *PYTHONPATH* line.

```
export PYTHONPATH=$PYTHONPATH:$HOME/.local/lib/python2.7/site-packages
```

Save and close the file. This ensure that python packets installed in user space are seen by the IOS.
Next unzip *ios.zip* to the home directory, open a new terminal, and give the following commands:

```
$ cd ios/installation/packages/pip
$ python setup.py install --user
```

```
$ cd ../pytz
$ python setup.py install --user
$ cd ../six
$ python setup.py install --user
$ cd ../virtualenv
$ python setup.py install --user
$ cd ..
$ pip install --user --no-index -f ./ incremental
$ pip install --user --no-index -f ./ django
$ pip install --user --no-index -f ./ channels
```

Now all packages needed by the IOS shall be available on the system.
Next steps will create an environment where the IOS can run.

```
$ cd $HOME
$ mkdir ios_virtual_env
$ cd ios_virtual_env
$ virtualenv env
```

Virtualenv can take several minutes to build the environment

```
$ cd $HOME
$ cp -R ios/ios_root ios_virtual_env/
```

To activate the IOS environment give the following command:

```
source ios_virtual_env/env/bin/activate
```

Next commands will use the Django *manage.py* script, they are needed to set-up an initial database.

```
(env) $ cd ios_virtual_env/ios_root
(env) $ python manage.py makemigrations
(env) $ python manage.py migrate
```

To set-up a database with some example data:

```
(env) $ python manage.py loaddata fixtures/all.xml
```

Instead, for a minimal working configuration:

```
(env) $ python manage.py loaddata fixtures/min.xml
```

Now everything is ready, the Django server can be started with the following command:

```
(env) $ python manage.py runserver 0.0.0.0:8000
```

The IOS can be used by opening a browser to:

```
http://localhost:8000
```

Login is required, initial credentials can be found in the *credentials.txt* file available in the installation folder.
To make the IOS accessible from any node in the network is necessary to edit the *settings.py* file.

```
$ cd $HOME
$ gedit ios_virtual_env/ios_root/ios/settings.py
```

In the following line add the IP address of the IOS machine.

```
ALLOWED_HOSTS = ['localhost', '127.0.0.1']
```

Ensure that the machine has a valid network configuration. By default CentOS will block incoming connections, the system administrator should edit default iptables rules.
To temporary allow connections (until reboot), give the following command:

```
$ sudo iptables –F
```

The IOS will be accessible on any network node at the following URL:

```
http://<ios_ip>:8000
```

Notice that the IOS is not already linked with the Framework, in the following section will be explained how to do it.

6.2.1 Link with Framework

This section will list all the steps necessary to link the IOS with the Framework, this guide will assume that the IOS is installed, there is a proper network configuration and the Framework is configured with all the necessary environment variables.
A preliminary check can be performed by launching the Framework with the following commands.
If the machine is also the master node:

```
$ cd $FRAMEWORK_PATH
$ ./bin/ncfs2_master
```

otherwise:

```
$ cd $FRAMEWORK_PATH
$ ./bin/ncfs2_slave <master_hostname>
```

From now on, all the steps will assume that the Framework is running.
For a proper management of models is also necessary to launch the model master and the model scheduler processes on the master node and put all models on stop state.
The IOS uses the Python Framework interface, the following command will ensure that it works:

```
$ cd ios_virtual_env/ios_root
$ python framework_test.py <master_hostname>
```

If error messages are returned there is a Framework misconfiguration, the IOS will not work or will have limited functionalities.
When everything is correctly set-up, the *settings.py* file has to be edited.

```
$ gedit ios/settings.py
```

The following section has to be filled properly:

```
# framework settings
FRAMEWORK = {
 # True if you want to connect the Framework, False otherwise
 'ACTIVE': True,
 # The Framework master node hostname
 'MASTER_HOSTNAME': 'localhost',
 # True if you want to manage models, False otherwise
 'MODEL_SCHEDULER': True,
 # Here the names of all model you want control with the IOS
 'TOT_FREEZE_MODELS': ['dummy', ],
 # Sleep time between WebSockets updates
 # 1 second is a safe option, on fast machines <= 0.5 is OK
 'THR_SLEEP': 1,
}
```

Then the IOS can be started:

```
$ cd $HOME
$ source ios_virtual_env/env/bin/activate
$ cd ios_virtual_env/ios_root
(env) $ python manage.py runserver 0.0.0.0:8000
```

Now the IOS will establish a connection with the Framework on start-up, otherwise, to control a simulator the IOS variables have to be linked to the simulator ones, in the IOS section of the Administrator Functionalities the complete procedure is explained.

6.2.2 Link with map server

The IOS is not dependent from a map server, any map server who expose a web interface compatible with web tiles *XYZ* format can be used.
To link a Map Server with the IOS is enough to modify the settings.py file:

```
$ gedit ios_virtual_env/ios_root/ios/settings.py
```

and edit the following section:

```
# map server settings
# change the URL with your own Map Server,
MAP_SERVER = {
  # provide an URL compatible with XYZ tile format
  'URL': 'http://<map_server_ip>:<port>/{z}/{x}/{y}.png'
  # projection standard used by the tile server
  'PROJECTION': 'EPSG:3857',
  # max tile server zoom level
  'MAX_ZOOM': '18',
  # min tile server zoom level
  'MIN_ZOOM': '0',
  # The pixel ratio used by the tile server.
  # For example, if the tile service advertizes
  # 256px by 256px tiles but actually
  # sends 512px by 512px
  # images (for retina/hidpi devices)
  # then tilePixelRatio should be set to 2.
  'TILE_PIX_RATIO': '1',
  # Tile size used by the tile server
  'TILE_SIZE': '[256, 256]',
}
```

In case is used the provided map server virtual machine, will be enough to change the URL to:

```
'http://<vm_ip_addr>:8080/styles/osm-bright/{z}/{x}/{y}.png'
```

6.3 Administrator functionalities

This part of the manual illustrates all the functionalities of the Administrator part of the IOS and is addressed to the engineering department who configures the software for instructor usage.
The Administrator section is accessible by clicking on the "wrench icon" in the menu-bar or directly at the following URL:

```
http://<ios_ip>:8000/admin/
```

The Administrator part is divided in several sections: IOS, Authentication, Aircraft, Failures, Map, Reposition.

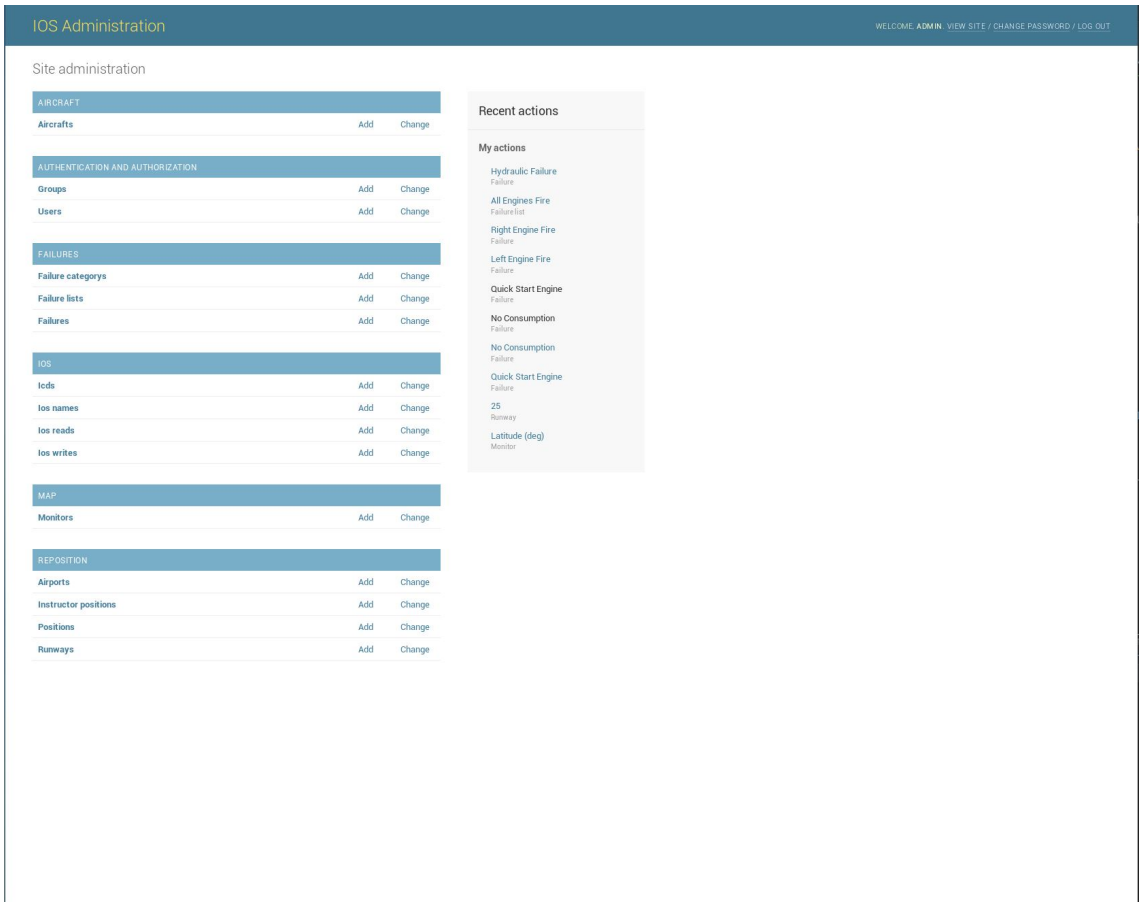

Figure 6.3. Admin page, view of the index

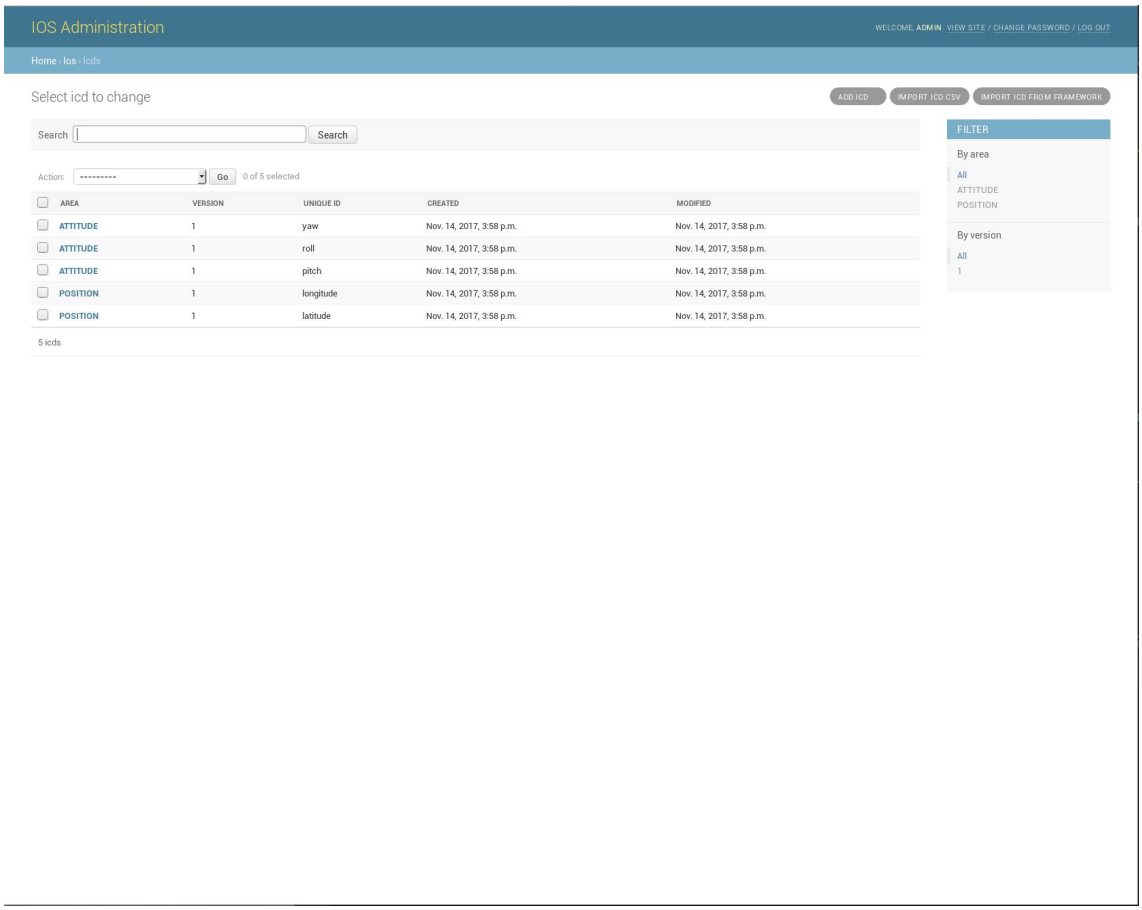

Figure 6.4. Admin page, view of a table (ICD)

6.3.1 IOS

This section allows to bind the IOS variables to the one managed by the Framework, there are 4 tables: *ICD*, *IosName*, *IosRead*, *IosWrite*.
All those tables allows to import and export rows in CSV. To import rows from CSVs there is an "import" button at the top right corner of each table's page.
Example CSVs files are available in the *csv_store* directory.
To export rows individually or in group, select them with a tick and then click on the export functionality available in the drop-down menu at the page top.
Generated CSVs will be stored in the *csv_export* directory of the IOS.

ICD

This table contains unique IDs of all the Framework variables that will be read or written by the IOS.
Each row has the following fields:

- **Area**: Memory area the variable belongs to.
- **Version**: Memory area version.
- **Unique ID**: The Framework variable id.
- **Created**: Date and time of row first creation.
- **Updated**: Date and time of last row update.

Beside the standard import/export functionalities, the interface allows to load rows directly from Framework by providing a list of comma separated area names.

Ios Name

This table has a list of names corresponding to IOS variables, each row has a single field, *Name*, that is a string of text.
After the IOS installation this table should be already populated with a set of variables used by the IOS functionalities, if not, import the provided *ios_names.csv*.
Names of this table should be changed only when the Python code of the IOS is modified or new functionalities are implemented.

Ios Read/Ios Write

Those tables perform the binding between ICD Framework variables and IOS variables, respectively for reading and writing operations.
Each row has the following fields:

- **Ios name**: A foreign key to an antry of the *IosName* table
- **ICD**: A foreign key to an entry of the *ICD* table
- **Scale factor**: An optional number that will be multiplied by the value that is retrieved or written.

Simulator first set-up

This section explains how to perform a first variable mapping.
The described procedure refers to the GRA simulator for which CSV files are already provided.
In case of a different simulator the procedure is the same but *ios_read.csv* and *ios_write.csv* files must be changed with a correct mapping.

1. Open the *ios_read.csv* and *ios_write.csv* files available in the *csv_store* directory of the IOS, and take notice of the different memory area names that appear in files, (they should be 3 or 4 area names).
   From the Administrator page click on *icds* and then on *IMPORT ICD FROM FRAMEWORK*.
   In the import page click on *IMPORT FROM FRAMEWORK* button.
   A list of all available memory area should be displayed, now in the input box write down the area names collected from the CSV and click again on the *IMPORT FROM FRAMEWORK* button.
   All the Framework variables available in the selected memory areas will be imported.
   At the end of the process click on *icds*.
   The imported variables should be available in the *ICD* table.

2. From the Administrator page click on *Ios names*, a list of IOS variables should be already available.
   If not, click on *IMPORT IOS NAME CSV*, select the *ios_name.csv* file from the *csv_store* directory of the IOS and click the *SUBMIT* button.

3. From the Administrator page click on *Ios reads*, then on *IMPORT IOS READ CSV*, select the *ios_read.csv* from the *csv_store* directory of the IOS and click the *SUBMIT* button.

4. From the Administrator page click on *Ios writes*, then on *IMPORT IOS WRITE CSV*, select the *ios_write.csv* from the *csv_store* directory of the IOS and click the *SUBMIT* button.

Now the mapping is complete, to make it effective and use the IOS to control the simulator is necessary to reboot the Django server.

6.3.2 Aircraft

This section allows to create a configuration for a specific aircraft.
If the IOS has been installed loading the *all.xml* file, there is already an entry for the ATR-72 of the GRA simulator.
To create a new Aircraft configuration click on the *ADD AIRCRAFT* button on the top right corner of the page and fill the following fields:

- **Name**: Name of the aircraft (must be unique)
- **Active**: Put a tick to make this configuration active.
  Only an aircraft at a time can be active, if there is another active aircraft an error is returned.
- **Fuel Tanks**: An aircraft can have up to 10 fuel tanks.
  To add a fuel tank click on *Add another fuel tank* and fill the following fields:
  - **Name**: Name of the tank that will be displayed on the IOS Aircraft page.
  - **Max Capacity**: Maximum tank capacity in *pound*.
  - **Initial Fuel**: Initial tank fuel in *pound*.

The IOS Aircraft page will display the current active configuration.

6.3.3 Authentication

This section allows to create new IOS users.
After the installation is already available the *admin* user that has full access to the Administrator page.
To create a new user click on *ADD USER* button, provide a username and a password and click on *SAVE*.
The new user will have access to all IOS pages except the Administration section.

6.3.4 Failures

This section allows to define failures.
It is divided in 4 tables: *FailureCategories*, *Failures*, *FailureLists*.
For each failure triggers can be defined, otherwise there is not a thread that checks for trigger occurrences and activate failures. This is left to future IOS developments. By now are available all the data structures necessary to perform trigger checks.

Failure categories

Is a collection of categories in which failures can be grouped.
A category can be created by clicking on *ADD FAILURE CATEGORY* button and providing a name.

Failures

Is the collection of available failures, to add a new failure click on *ADD FAILURE* button and fill the following fields:

- **Category**: A foreign key to an item of the category table
- **Name**: Name given to the failure, it will be visible in the Failure IOS page. Each name must be unique.
- **Active**: A Boolean value, is *True* when the failure is activated.
- **Description**: An optional description for the failure, if present will be displayed in the Failures page.
- **Trigger Type**: Failures can be activated manually or by events, here are defined all the events that trigger a failure, options are:
  - **None**: No trigger assigned.
  - **Altitude Above**: Failure is triggered when the altitude is above a specified value.
  - **Altitude Below**: Failure is triggered when the altitude goes below a specified value.
  - **Speed Above**: Failure is triggered when the speed is above a specified value.
  - **Speed Below**: Failure is triggered when the speed goes below a specified value.
  - **Weight on wheels**: Failure is triggered when the landing gear touches the ground.

- **Altitude**: Required only when a trigger of type *Altitude Above* or *Altitude Below* is selected, it is expressed in $feet$.
- **Speed**: Required only when a trigger of type *Speed Above* or *Speed Below* is selected, is expressed in *knots*.
- **Failure Parameters**: A set of failure variables and corresponding values written in memory areas when a failure is activated or disabled.
  To Add a new variable click on *Add another failure parameter* and fill the following fields:
  - **Ios Name**: A drop down list with all the available IOS variables ids, select one to write.
  - **Value On**: The value to be written in the corresponding memory area variable when the failure is enabled.
  - **Value Off**: The value to be written in the corresponding memory area variable when the failure is disabled.

  There is no limit to the number of variables that a failure can write.

Failures can be imported and exported in XML format in a similar way as tables of the IOS section are imported and exported in CSV.
The XML file should have this structure.

```
<?xml version="1.0"?>
<data>
  <failure category="cat_name" name="fail_name">
    <parameter ios_name="ios_name" off="val" on="val"/>
  </failure>
  <failure category="cat_name" name="fail_name">
    <parameter ios_name="ios_name" off="val" on="val"/>
  </failure>
  ...
</data>
```

All the failures imported in this way will have by default the trigger field set to *None*.

Failure lists

This section allows to create lists of failures, to create a list click on *ADD FAILURE LIST* and follow the procedure.
Failure lists will be displayed in the Failures IOS section and can be loaded from a drop-down menu.

6.3.5 Map

This section has a single table *Monitor*, it allows to create a list of variables that will be monitored on the Map IOS page.
To add a new variable click on *ADD MONITOR* and fill the following fields:

- **Name**: A label to be displayed for the monitored variable.
- **Variable**: A drop down list that refers to the row of the *IosName* table. Choose a variable name to be monitored.

6.3.6 Reposition

This section allows to defines coordinates and other parameters that are written when a reposition is performed.
It is divided in the following tables: *Airport*, *Position*, *InstructorPosition*, *Runway*.

Airport

This table is made of a list of names, to add a new airport click on *ADD AIRPORT* button and enter an airport name.

Position

This table contains all the values that identify an aircraft position and that are written in Framework variables during a reposition.
To add a new position click on the *ADD POSITION* button and fill the following fields:

- **Name**: A name for the position, (eg. "airport"-"runway").
- **Latitude**: A float value in *degrees*.
- **Longitude**: A float value in *degrees*.
- **Heading**: A float value in *degrees*.
- **Pitch**: A float value in *degrees*.
- **Roll**: A float value in *degrees*.
- **Ramp**: A float value in *degrees*.
- **Altitude unit**: A drop-down list, available choices are *meters* or *feet*
- **Altitude type**: A drop-down list, available choices are *MSL* or *AGL*

- **Altitude**: A float value in *meters* or *feet*
- **Speed unit**: A drop-down list, available choices are: *m/s*, *knots* or *mach*.
- **Speed**: A float value in: *m/s*, *knots* or *mach*.

In this table coordinates are expressed in decimal degrees, if a coordinate is in Degree Minutes Seconds (DMS) format can be loaded with the following procedure:

- Open the Reposition IOS page (not the administrator section) and enter the DMS coordinates in the *Customize Position* section.
  Give a name and click on *SAVE*
- The saved position will be available in the *Position* table where it can be furtherly customized.
  A reference to the position is also present in the *InstructorPosition* table, it can be deleted, the position will be preserved.

Instructor positions

Those are positions saved by instructors while they use the IOS, otherwise they can be created manually by clicking on *ADD INSTRUCTOR POSITION* button and filling the following fields:

- **Name**: A name to be given to the instructor position.
- **Position**: A drop down list that refers to the items of the *Position* table.

Runway

This table allows to define parameters for airport runways, such as the take off position or a list of in-flight positions to train pilots for landings.
A new table row can be created by clicking on *ADD RUNWAY* button and filling the following fields:

- **Airport**: A drop-down list that refers to items of the *Airport* table.
- **Name**: Name assigned to the runway, a character string.
- **Take off position**: The start position for the aircraft take-off, is a drop-down list referring to items of the *Position* table.
- **Landing Positions**: A set of in-flight positions to train pilots for landings, a new *Landing Position* can be added by clicking on *Add another landing position* and filling the following fields:
  - **Name**: Name for the landing position.

- **Category**: A drop-down list, available choices are: *Front Center*, *Front Right*, *Front Left*, *Reverse Left*, *Reverse Right*.
  Those choices will define in which category the item will be displayed inside the Landing Reposition IOS page.
- **Position**: A drop-down list referring to items of the *Position* table.

There is no limit to the number of in-flight positions that can be defined per each runway.

6.3.7 Data management

Django provides a set of commands for loading and exporting data, for a single App or for the whole project. Data exports are called fixtures.
The following commands must be given in an active environment and from the *ios_root* folder.
To export all data in XML give the following command:

```
(env) $ python manage.py dumpdata --indent 2 --format xml > filename.xml
```

To export data for a single App (eg. Aircraft, Reposition, Failure..) in XML give the following command:

```
(env) $ python manage.py dumpdata app_name --indent 2 --format xml > filename.xml
```

To load data from an exported XML file:

```
(env) $ python manage.py loaddata fixture_name.xml
```

To delete the project database:

```
(env) $ find . -path "*/migrations/*.py" -not -name "__init__.py" -delete
(env) $ find . -path "*/migrations/*.pyc" -delete
(env) $ rm -rf ios_database
```

After the reset is necessary to set-up again the initial database as explained in the installation procedure.

6.4 User functionalities

This section is addressed to instructor pilots or other operators that will use the IOS to control the simulator.
The IOS interface is composed of a top bar where are located controls to start stop pause and resume the simulation, the current date and time of the simulation, the time elapsed since the start button has been pressed and a drop down menu with a counter where are listed important events.
An event can be an active failure or a setting that changes the normal aircraft behaviour, such as the activation of "Quick Start Engine" and "No Consumption" modes.

On the left side of the screen there is a menu-bar that gives access to different IOS pages. Relevant sections are: Home and Map (House Icon), Reposition (Globe icon), Aircraft (aircraft icon) and Failures (flash icon).

6.4.1 Home and Map

In this section there is a Map where a marker shows the current aircraft position and heading.
The aircraft can be locked or unlocked to the map.
When locked, the marker will be fixed at screen centre, the map will move and rotate according to the simulation coordinates.
When unlocked, the map can be panned and zoomed, the aircraft will move independently.
Depending on the settings applied by the Administrator may be also available a section to monitor relevant simulation variables.

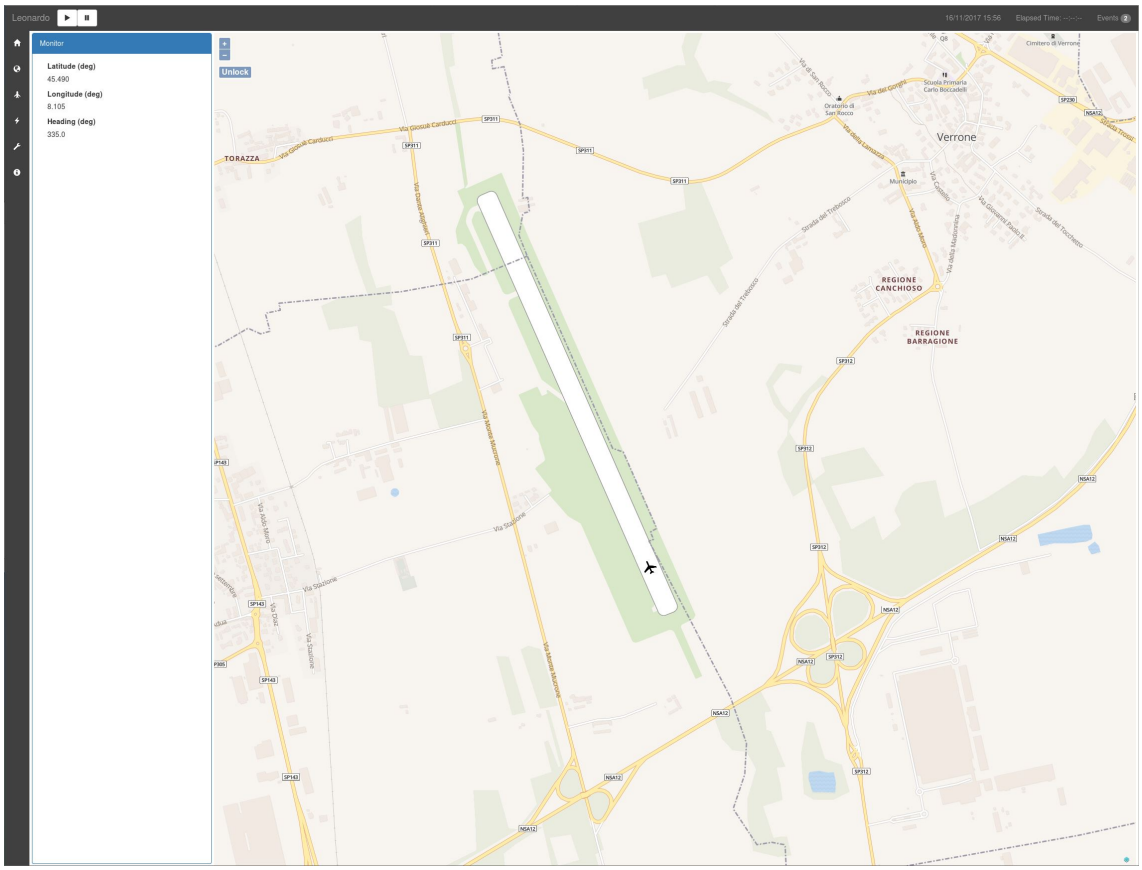

Figure 6.5. Home and Map page as is displayed on a desktop screen

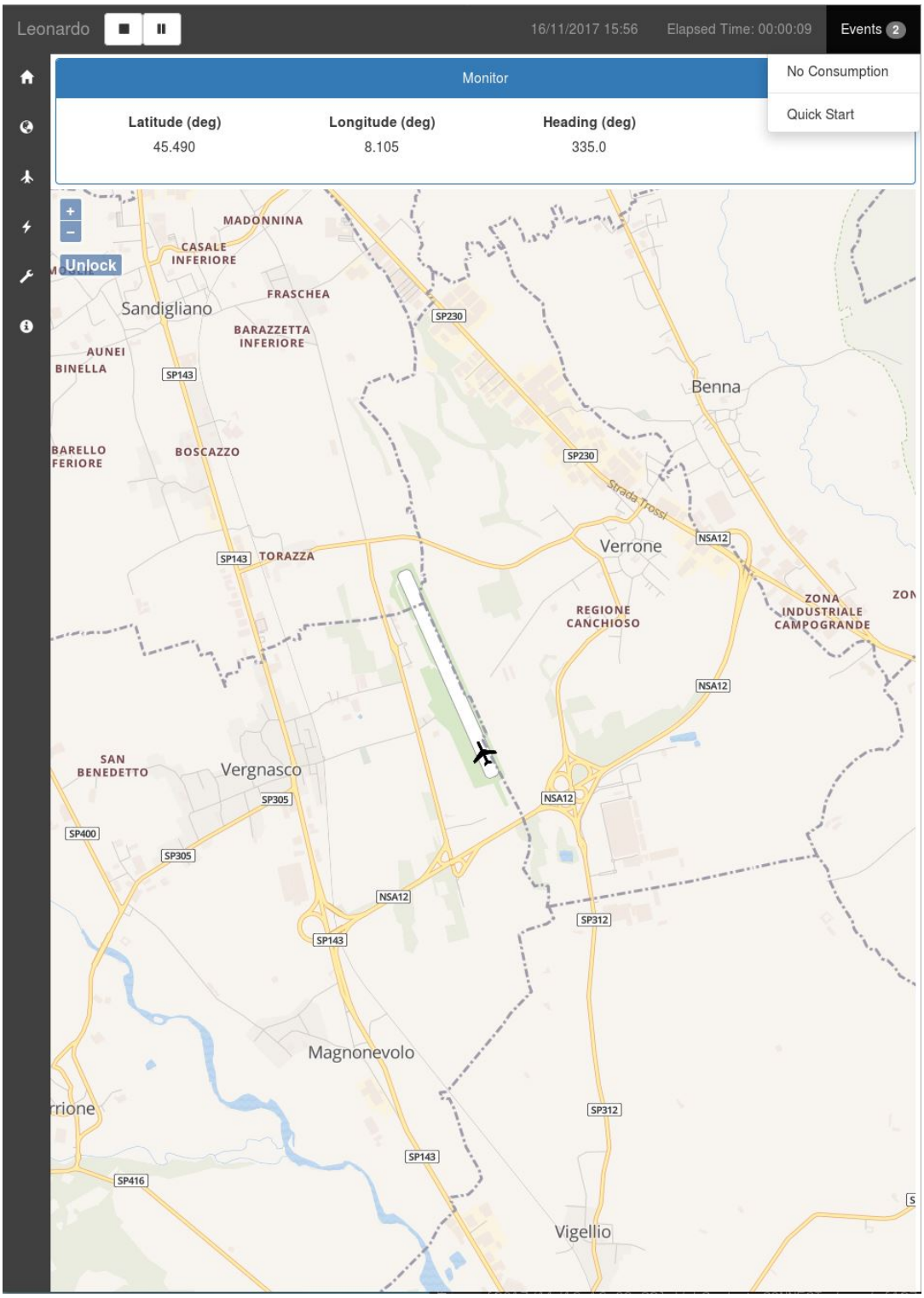

Figure 6.6. Home and Map page as is displayed on a tablet screen, the simulation is started and the "Events" list is open.

6.4.2 Reposition

This section allows to move the aircraft in the virtual environment and to set a date and time.
Repositions can be applied only when the simulation is stopped. Settings will become effective as soon as the start button in pressed.
There are three ways to select a position:

- By manually filling coordinates, heading, speed and altitude in the "Customize Position" section.
- By selecting airport and runway from a drop down menu, in this case the "Customize Position" section will be automatically filled
- By clicking on the displayed map. In this case the "Customize Position" section will be filled with the coordinates of the point clicked.

The "Customize Position" section has also buttons to change the coordinates directions *north* and *south* for latitude *east* and *west* for longitude.
The speed unit: $knots$, m/s or $mach$.
The altitude unit: $feet$ or $meters$.
The altitude type: MSL or AGL.
Positions can also be saved and then reloaded. There is a *Save Position* button to memorize a position and a *Load Position* drop-down menu where saved positions can be retrieved.
When all the fields are filled, clicking on *Set Position* will send the coordinates to the simulator.
In this page there are also controls to set the date and time. A date can be chosen in the following ways:

- Manually, providing a string in the format: dd/mm/yyyy hh:mm
- By an interactive widget displayed when the calendar icon is clicked.
- By selecting a season from a drop down list, each season when selected will load a different date-time string.

The selected time will become effective only when the *Set date and Time* button is clicked.
On the page top there is a tab, it gives access to the *Landing Reposition* section.
Here can be selected an in-flight position near an airport runway, the procedure is the following:

- Stop the simulation
- Select an airport and a runway, 5 drop down menus will be populated with the available in-flight positions.

- Apply customizations if needed by changing heading, altitude and speed.

- Click the set position button to apply settings and start the simulation to see changes.

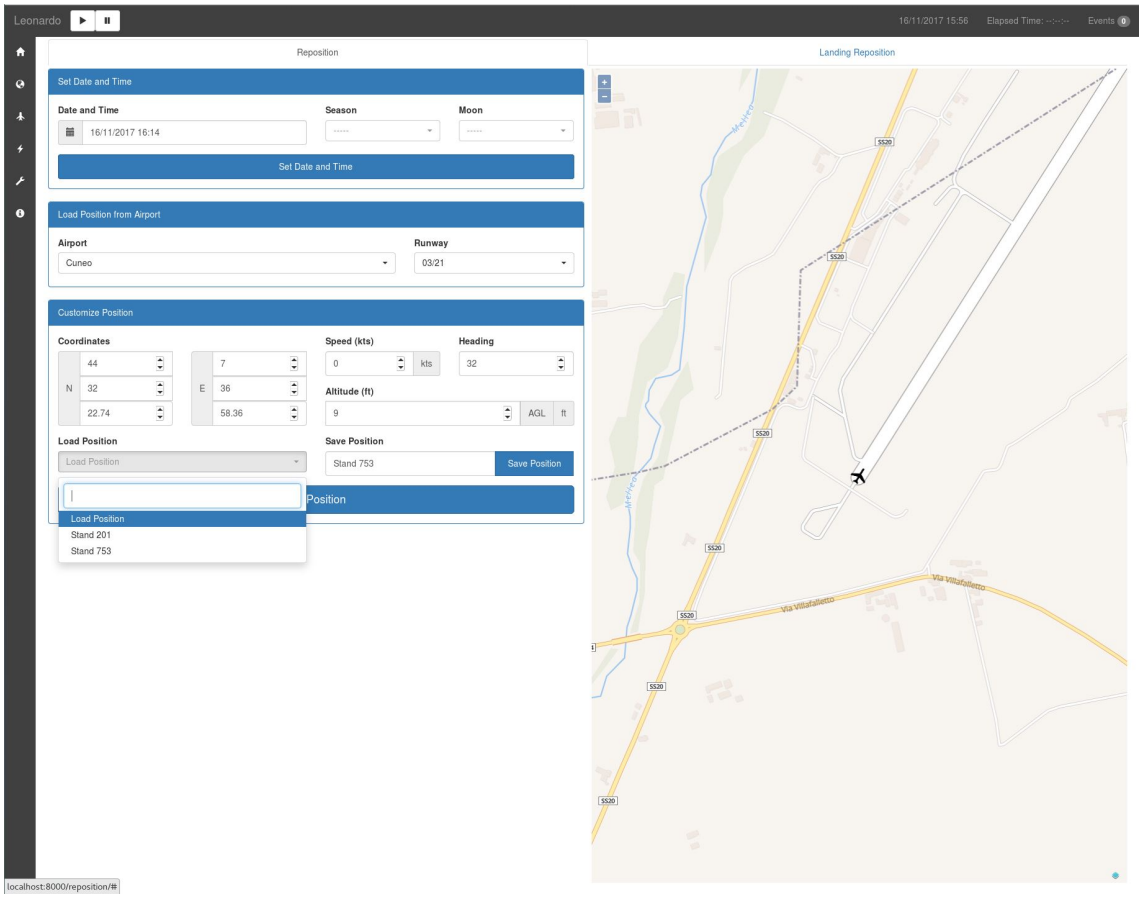

Figure 6.7. Reposition page as is displayed on a desktop screen, the "Load Position" menu is opened

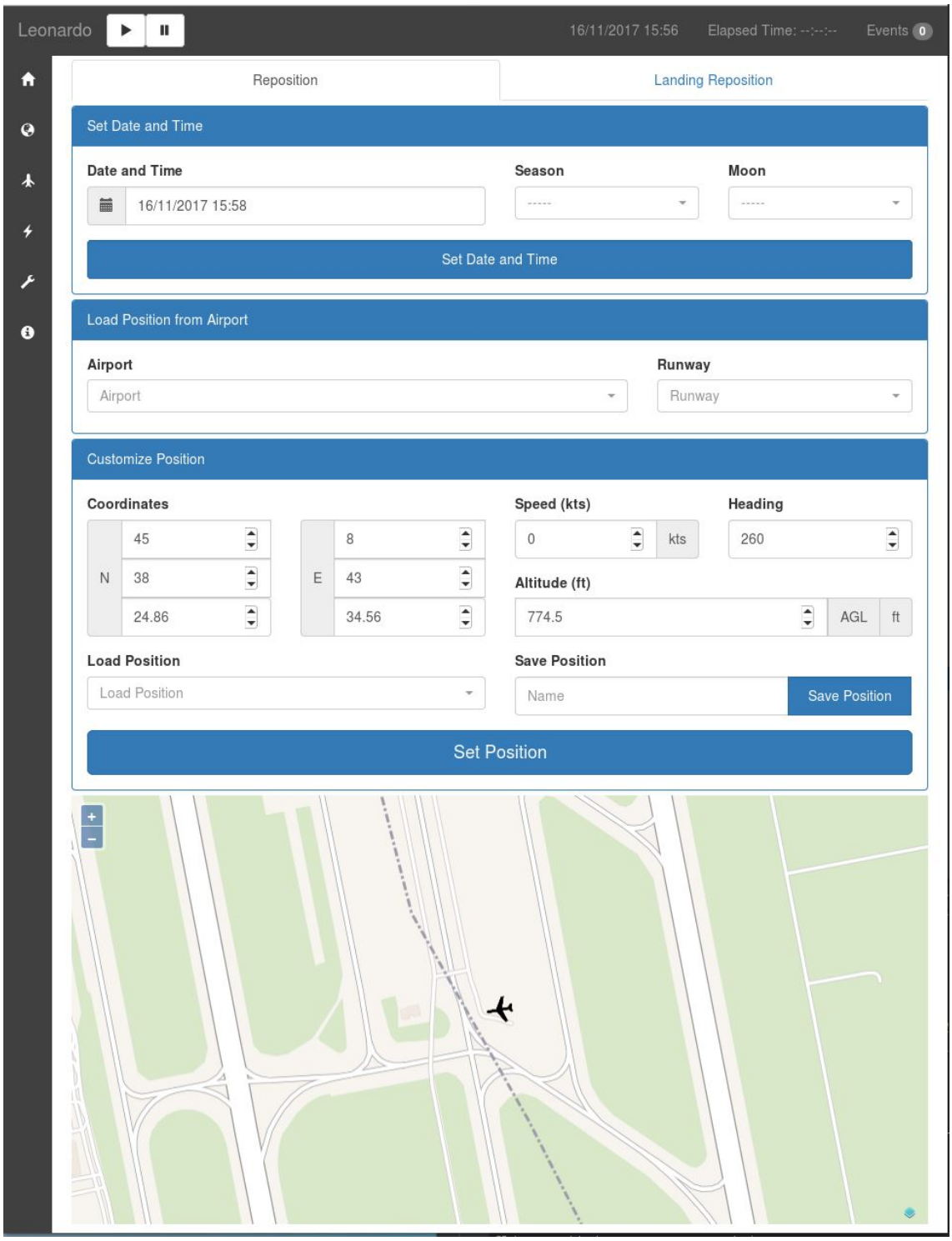

Figure 6.8. Reposition page as is displayed on a tablet screen

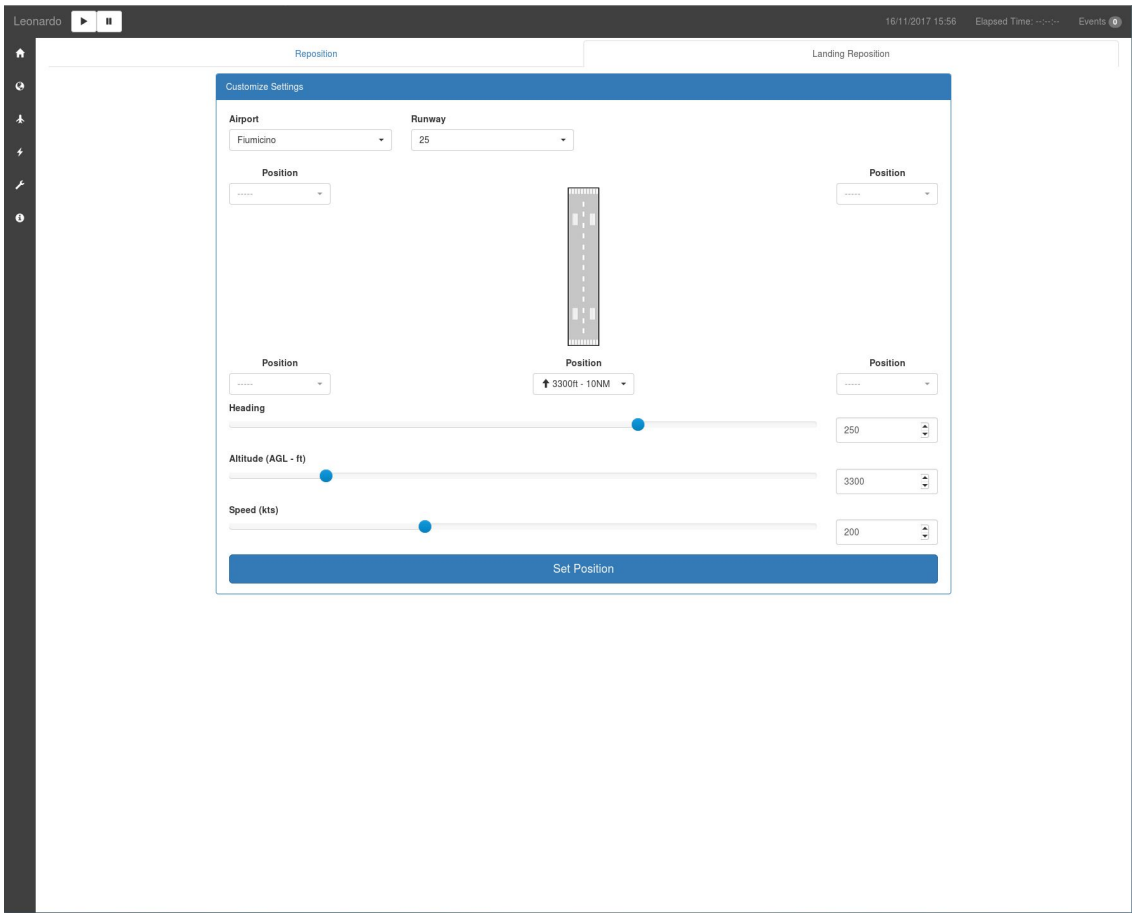

Figure 6.9. Landing Reposition page

6.4.3 Aircraft

In this section there are all the settings that apply to the simulated aircraft such as fuel quantities and switches.
The fuel quantity can be controlled per each tank. When the simulation is running fuel level is shown both in a numeric and a graphical interface where aircraft tanks are reproduced.
When simulation models are stopped or paused is possible to adjust the fuel quantity per each tank, by moving sliders or by entering the fuel as a number in *pound*. There are then other settings with on/off switches that allows to control aircraft parameters.

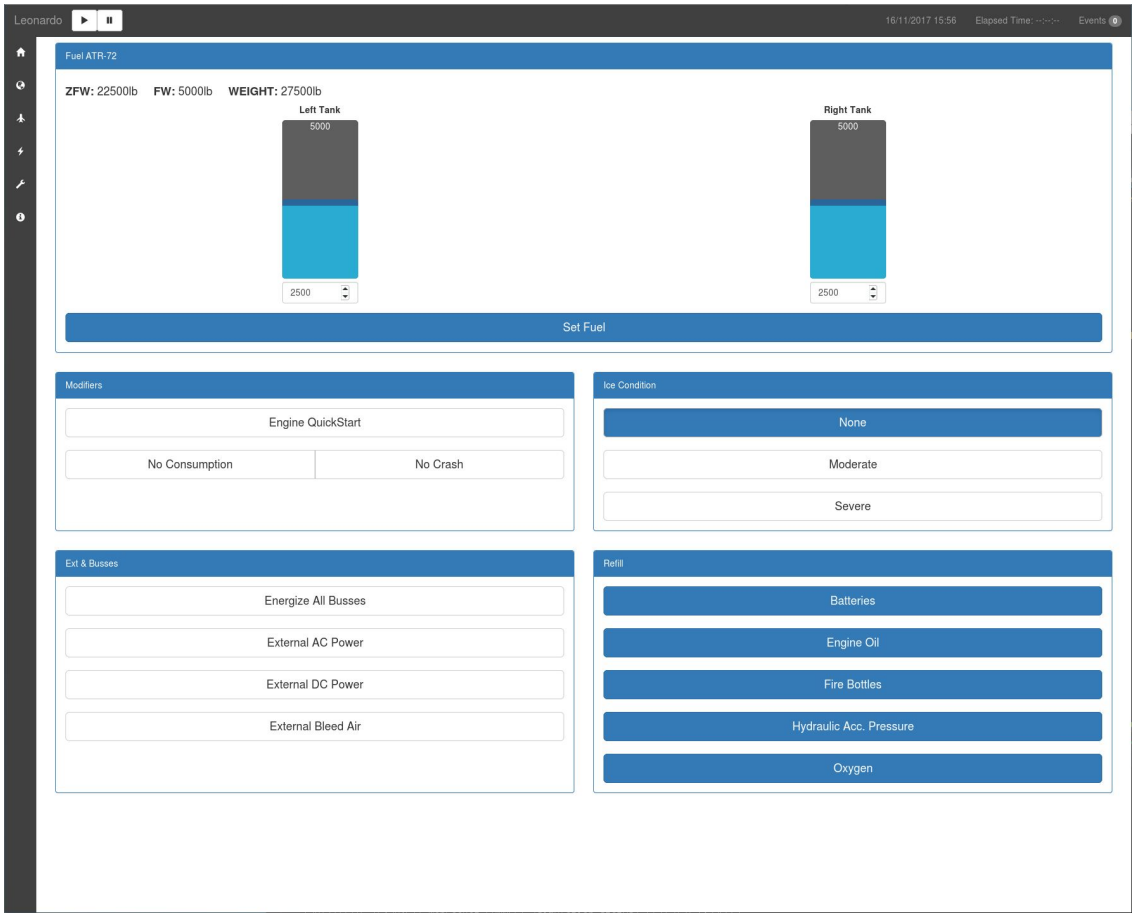

Figure 6.10. Aircraft page

6.4.4 Failures

This module contains all settings relative to failures.
Failures can be displayed in a single list or divided by category.
In the failure title there are two buttons, one to enable/disable the failure and the other to add the failure to a custom list.
Clicking the failure title will open the body section. Here a description is provided, if present, and a form for triggering the failure is shown.
The trigger type can be selected from a drop-down list, depending on the selected trigger type may be required to enter additional values for speed ($knots$) or altitude ($feet$).
Failures added to a custom list can be enabled, disabled and triggered all together.
Lists can be customized by adding or removing failures and saved in database by providing a name and clicking on *save* button.
Saved lists can be loaded from a drop-down menu.

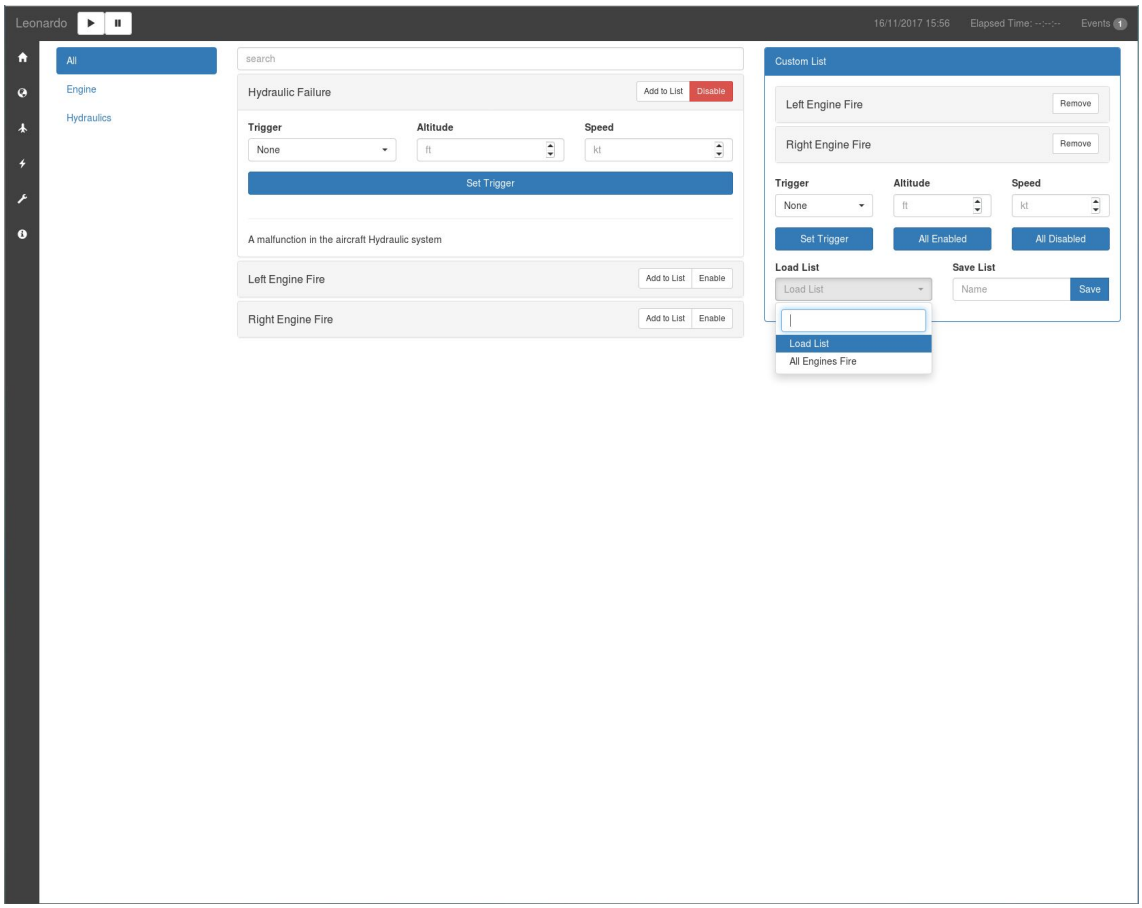

Figure 6.11. Failure page as is displayed on a desktop screen, The "Hydraulic Failure" is active and its section is expanded, also the "Load List" drop-down is opened

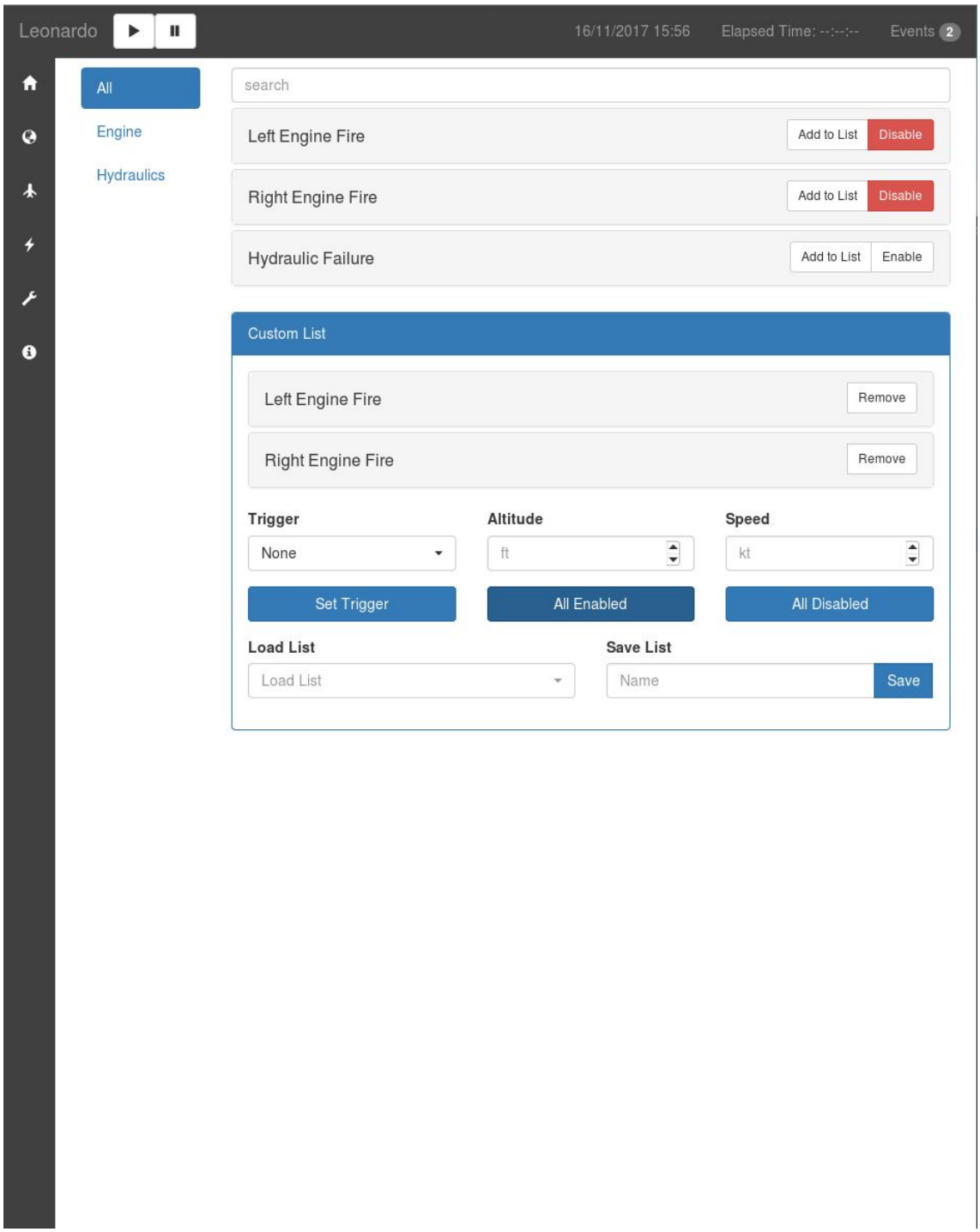

Figure 6.12. Failure page as is displayed on a tablet screen, all failures loaded in the custom list have been activated

Chapter 7

Conclusions

The IOS was tested on a FTD called GRA used by Leonardo AD for research purposes.
The simulator can emulate different aircraft of the ATR family. The IOS was configured for the ATR-72, it allowed a complete control of simulation models.
Start, stop, pause and resume operations were correctly performed.
Was also possible to move the aircraft using the Reposition page. Different airports and runways coordinates have been loaded and tested.
For what concerns aircraft controls was possible to set fuel levels and enable/disable the QuickStart Engine function.
Quick Start allows to turn on engines without executing a complete start-up procedure on the simulator.
Readings of parameters such as aircraft coordinates and heading were performed during flight and the live position was correctly displayed as a moving marker on the map. Also Fuel levels and weights were displayed and live updated.
Was not possible to inject failures, because they are not provided by the GRA simulator, otherwise every action performed by the IOS relies on reading/writing variables and managing models, this capabilities have been tested, so they are expected to work when supported.
The new Simulation Framework is under development and the IOS integration was also an opportunity to find bugs. During the IOS testing, was discovered and fixed a problem that prevented the correct loading of variables when models were stopped and restarted.

7.1 Future developments

By now the integration of the Simulation Framework with web technologies requires an external library to exchange data, in this peculiar case was the Python Framework API that allowed Django to read and write variables.
The C++ Framework and the Django environment are two distinct entities, this implies a delay in data exchanges and a fairly complex architecture to be managed, there is from one side the Framework, written in C++, and on the other the web infrastructure (web server, Django, SQLite, JavaScript, HTML pages, CSS).
During the application development was noticed that sometimes the engineering department needs to implement quickly basic functionalities such as collection of buttons or drop-down lists for which the complexity of a web framework is not needed.
The current architecture can be simplified by integrating the WebSockets management inside the Simulation Framework.
WebSockets do not require a web server, the connection is opened client-side by JavaScript when HTML pages are displayed in browsers. If the Framework can send and receive data with the WebSockets protocol, the Python/Django part can be removed.
Some functionalities such as management of user sessions and ORM support will be lost, otherwise portable web interfaces will be easily created.
All the files (HTML pages JavaScript and CSS) will be collected in a folder, and copied where the IOS is needed.
The software will behave like a client application, but without the need to be compiled and installed on different platforms.

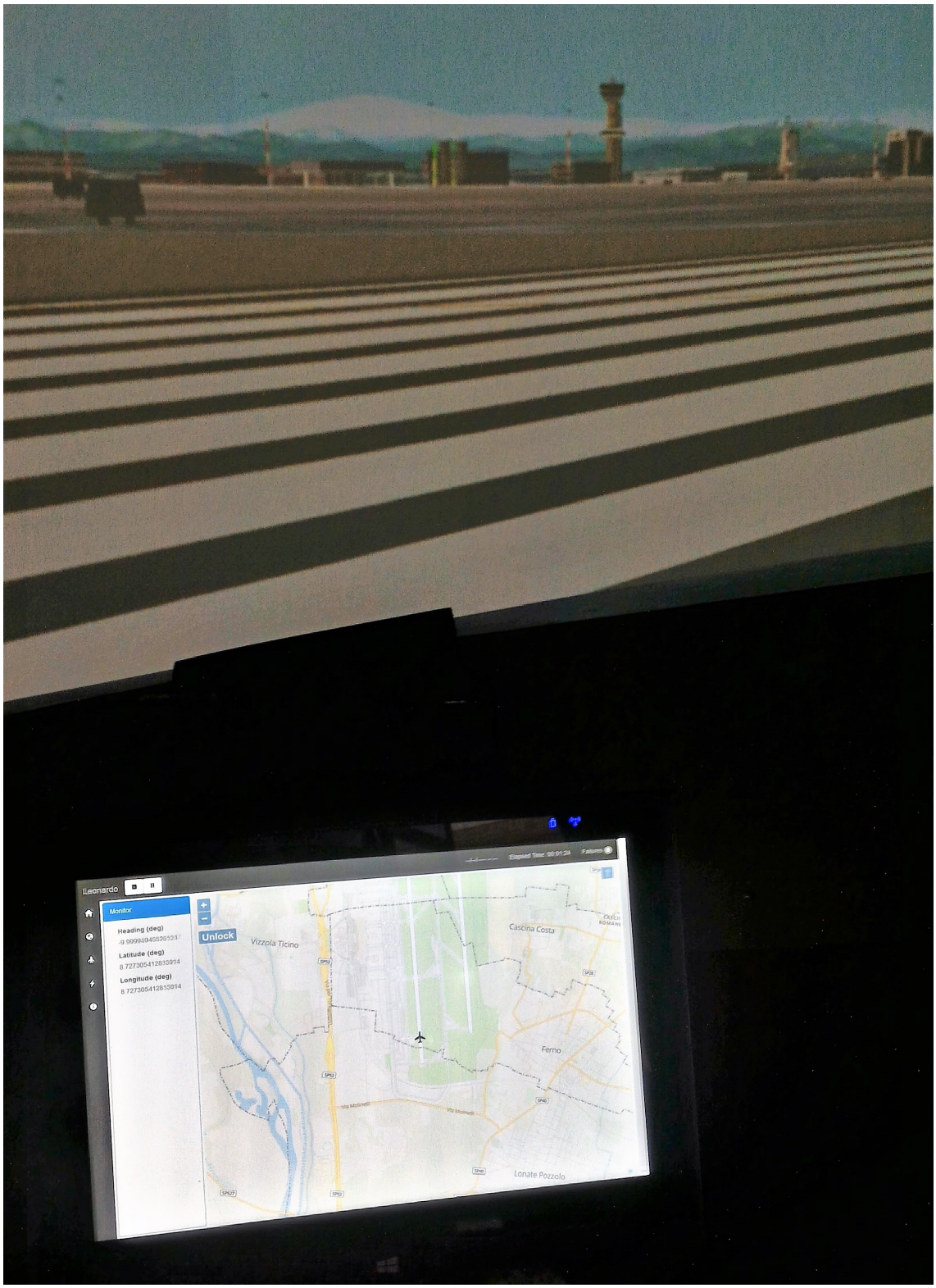

Figure 7.1. View from the GRA simulator cockpit, the IOS is in operation on a tablet

Appendix A

Acronyms

IOS Instructor Operator Station

GUI Graphical User Interface

HTML HyperText Markup Language

API Application Programming Interface

ORM Object Relational Mapping

HTTP HyperText Transfer Protocol

CSV Comma Separated Values

FAA Federal Aviation Administration

EASA European Aviation Safety Agency

FFS Full Flight Simulator

FTD Flight Training Device

I/O Input/Output

AMI Aeronautica Militare Italiana

GCA Ground Controlled Approach

VFR Visual Flight Rules

ICAO International Civil Aviation Organization

PDF Portable Document Format

TCP Transmission Control Protocol

IP Internet Protocol

OS Operating System

AJAX Asynchronous JavaScript and XML

XML eXtensible Markup Language

JSON JavaScript Object Notation

W3C World Wide Web Consortium

IETF Internet Engineering Task Force

RFC Request For Comments

SQL Structured Query Language

JDK Java Development Kit

SDK Software Development Kit

MVC Model View Controller

MTV Model Template View

WSGI Web Server Gateway Interface

DBMS Database Management System

IDE Integrated Development Environment

CSS Cascading Style Sheets

URL Uniform Resource Locator

GIS Geographic Information system

ICD Interface Control Document

AGL Above Ground Level

MSL Mean Sea Level

GRA Generic Regional Aircraft

GCC GNU Compiler Collectiont

DMS Degree Minutes Seconds

ATR Aerei di Trasporto Regionale

Bibliography

[1] Federal Aviation Administration, *AC 120-40B - Airplane Simulator Qualification*, July 29 1991

[2] Federal Aviation Administration, *AC 61-136A - FAA Approval of Aviation Training Devices and Their Use for Training and Experience*, November 17 2014

[3] European Aviation Safety Agency, *Certification Specifications for Aeroplane Flight Simulation Training Devices*, July 4 2012

[4] https://www.djangoproject.com/ *Django Project*, November 15 2017

[5] https://spring.io/ *Spring*, November 15 2017

[6] https://laravel.com/ *Laravel*, November 15 2017

[7] https://nodejs.org/it/ *Node.js*, November 15 2017

[8] http://rubyonrails.org/ *Ruby on Rails*, November 15 2017

[9] https://www.xamarin.com/ *Xamarin*, November 15 2017