electronics

MDPI

Article

Enhancing EJsS with Extension Plugins

Jesus Chacon *, Eva Besada-Portas, Gonzalo Carazo-Barbero and Jose Antonio Lopez-Orozco

Department of Computer Architecture and Automation, Complutense University of Madrid, 28040 Madrid, Spain; ebesada@ucm.es (E.B.-P.); gocarazo@ucm.es (G.C.-B.); jalopez@ucm.es (J.A.L.-O.)
* Correspondence: jeschaco@ucm.es

Abstract: Easy JavaScript Simulations (EJsS) is an open-source tool that allows teachers with limited programming experience to straightforwardly bundle an interactive computer science or engineer simulation in an HTML+ JavaScript webpage. Its prominent place in Physics (where it has won several prizes) should not hinder its application in other fields (such as building the front-end of remote laboratories or learning analytics) after having adapted part of the functionality of EJsS to them. To facilitate the future inclusion of new functionalities in EJsS, this paper presents a new version of this tool that allows the enhancement of EJsS, letting it incorporate new tools and change its graphical user interface, by means of extension plugins (special software libraries). To illustrate the benefits of this distributable self-contained non-intrusive strategy, the paper (a) discusses the new methodological possibilities that the Plugins bring to EJsS developers and users, and (b) presents three plugins: one to support the plugin management and the others to easily set up a streamlined remote laboratory. Moreover, the paper also presents the main characteristics of that remote lab to allow readers take advantage of EJsS and the three plugins to set up new online experiments for their students quickly.

Keywords: open-source software; learning technologies; virtual and remote laboratories; Internet-based teaching; Easy JavaScript Simulations

Citation: Chacon, J.; Besada-Portas, E.; Carazo-Barbero, G.; Lopez-Orozco, J.A. Enhancing EJsS with Extension Plugins. *Electronics* **2021**, *10*, 242. https://doi.org/10.3390/electronics10030242

Academic Editor: Anna Rita Fasolino
Received: 24 November 2020
Accepted: 18 January 2021
Published: 21 January 2021

1. Introduction

Computer simulations are useful for teaching science and engineering, as they help students to establish relationships between the theoretical concepts and simplified/controlled versions of real-world scenarios [1,2]. Online pedagogical material and courses make some simulations leave the classroom and become interactive graphical tools that, being part of virtual and remote laboratories, allow students to understand the behavior of the system under study [3–5]. Ready-to-use simulations are also collected and shared, through different projects and web pages, such as the Interactive Simulations for Science and Math in [6], Open-Source Physics [7] or the University Network of Interactive Laboratories for Control Engineering [8].

Developing interactive graphical simulations usually requires the implementation of the *model* that defines the behavior of the system (e.g., the Ordinary Differential Equations —ODEs— in dynamic systems), apply a simulation strategy (e.g., an ODE integrator/solver) and build the *view* (i.e., the graphical user interface) of the simulation. To this end, different applications have already been used and/or combined, ranging from simulation commercial tools (e.g., Modelica, Matlab and EcosimPro) to video games simulation engines (e.g., Unity and Second Life) or generic programming environments (e.g., JAVA, JavaScript + HTML and Node.js), as the following labs/methodologies, to name a few, show [9–17]. Despite their success, these approaches require their developers to have a good knowledge of the tool and/or programming technique, hardening the creation of new virtual simulated labs.

Easy JavaScript Simulations [18,19] is an alternative free open-source tool that has earned a prominent place in physics education, with more than a thousand of simulations

available in the ComPADRE digital library [20] and several prizes (e.g., the American Physical Society 2020 Excellence in Physics Education Award and the 2015 Multimedia in Physics Teaching and Learning Award). The popularity of EJsS comes essentially from the fact that it simplifies the creation of an interactive learning simulation for people who have a limited programming experience, but who might have great ideas for teaching and experimenting. Within the advantages of EJsS, we consider three of them invaluable. First, it is easy to use due to a clear separation between the *model* behavior and the *view* graphical components, and to a trivial and interactive way to link them. Second, its bunch of built-in numerical solvers and powerful event handling mechanisms can cope with many simulation scenarios. Third, it automatically bundles an HTML + JavaScript webpage, which can be directly shared with the students. All these characteristics hide the complexity of the simulation implementation behind the scenes, letting its developers focus their attention on the problem itself and on the creation of an attractive interactive view.

Although the initial goal of EJsS was to facilitate the creation of interactive scientific computer simulations, over the years it has found application in other fields, such as building the front-end of different remote labs [21–33] or for learning analytics [34]. These new uses of EJsS have required that the developers of this tool, helped by some of its users with advanced programming capabilities, adapt/extend its functionality to/for the necessities of the new application domains. For example, in its remote lab applications the simulation of the *model* has been substituted by communications routines between the EJsS *view* and the remote system. Or for performing learning analytics with EJsS, different statistical tools have been incorporated in the newer version of EJsS. Although several of these functionality changes have been addressed in the past by EJsS capability to include new *elements* in the model (e.g., communication or learning analytics libraries), others (e.g., including new tools and view components, or supporting the edition of the code that interacts with the remote system) have required to modify directly the EJsS code, needing a deep understanding of the tool's programming, and causing issues in versions compatibility and distribution.

The objective of this work is to present a new standardized way of modifying the functionality and Graphical User Interface (GUI) of EJsS to facilitate its extension to new application domains. In other words, this paper presents a new version of EJsS that takes advantage of its existing capabilities, while adapting its aspect and tools to new types of users without modifying directly EJsS code. Moreover, the presented approach (based on self-contained extension plugins, already incorporated in EJsS version 6.0 and available from [35]) will allow a streamlined non-intrusive distribution of new functionalities and GUI elements of EJsS, and facilitate its maintenance in new releases of EJsS. Moreover, it has been especially designed to bring cohesion to future features of EJsS *model* and *view*, and to support the integration of new GUI elements within EJsS.

Furthermore, and to explain how the new version of the tool and methodology works, this paper also presents three of our extension plugins. The first is intended to help EJsS users to install/enable/disable new plugins. The other two plugins can help EJsS users to set up a streamlined remote laboratory. Considering the current COVID-19 teaching scenario, where easy-to-use online teaching resources are required to handle blended and remote learning, we have decided to include also in this work a description of the developed remote lab infrastructure for those readers interested in it. The plugins and the remote lab infrastructure are available from [36].

The rest of the document is structured as follows. Section 2 presents the background, discussing the main characteristics of EJsS, presenting the traditional extension mechanisms, and analyzing their benefits, drawbacks and limitations. Section 3 highlights the methodology changes supported from the version of EJsS that incorporates the extension mechanisms presented in this paper. Section 4 gives an overview of the characteristics of the new extension mechanisms, presents the changes performed in EJsS to support them, and describes the key-points that the features and the architecture of any extension plugin must fulfill. Section 5 illustrates, through case studies and from the EJsS perspective, how

the plugins can adapt EJsS to fit the needs of different developers and fields of application. Section 6 describes the main characteristics of the remote lab infrastructure developed to test the plugins, and shows how to set up a remote lab quickly with it and with the help of EJsS enhanced by the same plugins. Finally, Section 7 draws the conclusion and presents the future research lines.

2. Background

The contents of this section are three-fold. On one hand, it describes the main characteristics of EJsS, especially highlighting those ones that are required to understand the new extension mechanism presented in the paper. On the other one, it analyzes the traditional extension mechanisms already supported by EJsS, establishing its differences and limitations. Finally, and as a consequence of the previous analysis, it raises a set of questions about other possible ways of increasing EJsS functionality and making it compatible with future releases of EJsS.

2.1. Easy JavaScript Simulations Overview

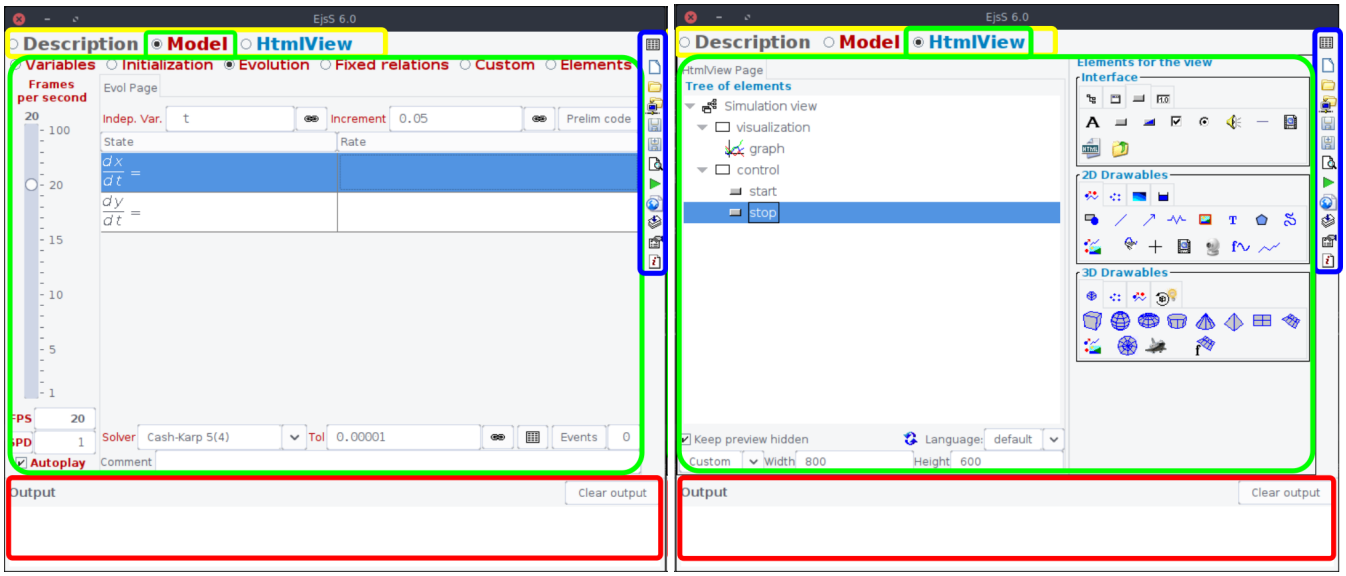
Essentially, EJsS [19] is a tool that allows users to define, in a decoupled way, a simulation *model* using ordinary (and delay) differential equations and the *view* that conforms its interactive and graphical visualization. Moreover, it takes the *model* and *view* defined by the users to generate the HTML + JavaScript code of the corresponding webpage simulation. To this end, EJsS makes use of different numerical integration methods (from fixed step algorithms such as the Euler method to more sophisticated ones such as the Cash–Karp algorithm), built-in event detection and handling procedures, and other sophisticated mechanisms that are useful to create and run complex simulations. Moreover, it incorporates a palette of visual and interactive elements that help users to design illustrative and attractive graphical environments for their simulations. All these features make EJsS a great tool to create interactive simulation and virtual laboratories.

To facilitate the interactions of the developers of the simulations or applications in EJsS, its functionality is also distributed into several areas that group conceptually related elements:

- The *Main Editors Panel*, placed at the top of EJsS and framed in yellow at Figure 1, gives access, by default, to the three main tools required to define all the aspects of a new simulation. On one hand, the *Model Editor*, framed in green at Figure 1a, allows to define the variables and the differential equations of the model, to write fixed relations and custom code, and to use extension elements. On the other, the *HtmlView Editor*, framed in green at Figure 1b, contains, at its left side, the tree of elements that the user has already combined to obtain the current *view* of the simulation and, at its right side, the palette of available visual and interactive elements (e.g., buttons, sliders, text fields, 2D plots, or 3D drawables). Finally, the *Description Editor* (which is not displayed in Figure 1 because its functionality has not been modified in the new version of EJsS) is used to let the simulations or applications developers provide helpful information to its final users.
- The *Buttons Bar*, placed on the right of EJsS and framed in blue at Figure 1, contains shortcuts to common actions, such as building and deploying the application, accessing to global configuration parameters, etc. These buttons can directly trigger an action, or open a dialog that groups several actions or configurations.
- The *Output Message Area*, placed at the bottom of EJsS and framed in red at Figure 1, serves as a log, where EJsS notifies its direct users (i.e., the simulations or applications developers) of potential errors and of other events that occur during their interaction with the tool.

Finally, for those readers familiar with Easy *Java* Simulations (EJS, [37]) is worth noting that Easy *JavaScript* Simulation (EJsS) appeared to complement the former. Moreover, both follow the same decoupled definition methodology, but while the new EJsS encapsulates the simulations as HTML+JavaScript webpages, the old EJS generates a Java applet. For

this reason, The arrival of EJsS in 2015 was acclaimed since webservers were removing its applet support for security reasons, and hence EJS applets could no longer be hosted in educational webpages.



(**a**) After selecting the Model Editor (**b**) After selecting the HtmlView Editor

Figure 1. Screenshots of EJsS standard GUI.

2.2. Traditional EJsS Extension Mechanisms

Adapting EJsS functionality to other application fields was already possible due to the following decisive factors:

1. *The accessibility to the code of EJsS*, which is an open-source tool available in a public repository, *allows anyone with enough programming knowledge and time to study and customize it to his/her needs*. Although this approach undoubtedly offers an absolute control and flexibility, it is not always the best alternative. On one hand, introducing changes into EJsS requires a deep knowledge of its code. Moreover, the implications of a change in other parts of the code are not always obvious, due to the complexity and size of EJsS. On the other one, it is not efficient to modify EJsS with every feature you want to add, because when a new version of EJsS will be released, probably the same changes will have to be reworked.
2. *The possibility of creating new View Elements* to satisfy the needs of new types of users (e.g., to include the visual representation of a frequently used device). To do it, the user must modify the EJsS code locally, implementing the interface of a `ViewElement` (which will allow EJsS to be aware of its existence and to be able to use it), assigning a new icon to the element and arranging it into a group of the palette components. The new *View Element* can afterwards be used in the graphical interface of the user application, similarly to the default EJsS *View Elements*. Although this approach is less code-intrusive than the previous, the new *View Element* will only be available in the EJsS application where its code has been modified (and hence, it is not available for other users or should be included again in future releases of EJsS).
3. *The possibility of adding new Model Elements*. To allow EJsS to support more behaviors than the usually defined with ordinary/delay differential equations and event handling mechanisms, EJsS allows to encapsulate and reuse other functionality from the initially empty *Model Elements Palette*. Again, the new *Model Element* must implement the interface `ModelElement` to let EJsS be aware of its existence and to be able to access its functionality. However, instead of incorporating this new code straightaway into the code of EJsS, *Model Elements* can be provided as external libraries of new function-

ality (e.g., the communication routines of the remote labs) that, once deployed in an appropriated folder, EJsS can load to make them available for the users that require them. This approach helps to easily share the new functionality between users and quickly reuse it in new version of the code.

2.3. Questioning the Extension Mechanisms

It is worth noting that the previous functionality adaptation mechanisms are presented in this paper by decreasing programming difficulty and increasing decoupling with EJsS code. Moreover, although a *View Element* and a *Model Element* are conceptually different (the first implements visual components, while the second encapsulates behaviors), they (1) share the mechanism that makes them usable within EJsS (which consists on implementing an interface of a given class) and (2) diverge in the way that they are actually included within EJsS (because the first requires the modification of the EJsS code, while the second can be implemented as an external-loadable library).

While wondering about the reason of the last difference, new questions arose. For instance, could we (or would be useful to) also modify, by using external-loadable mechanisms, other parts of EJsS, such as the tools available from the *Buttons Bar* or the editors from the *Main Editor Panel*? Is there a better reason to make some tools accessible from one group of elements of the EJsS interface or from another? Could we bundle modifications in different parts of EJsS in a unique external plugin? Could we distribute the modifications, according to their final purpose, in multiple simultaneously loadable plugins?

To be able to give positive answers to the previous questions we have developed the new extension plugin mechanism for EJsS that is presented in the following sections of this paper. With it, we expect to: (a) provide an elegant strategy to facilitate the improvement of EJsS, (b) let developers with advanced programming capabilities customize or extend the EJsS interface and functionality for its potential final users, (c) support an easy-to-use update-independent distribution method, and (d) let users choose, according to their needs, the specific extensions that they want to incorporate into EJsS.

3. Methodology Overview

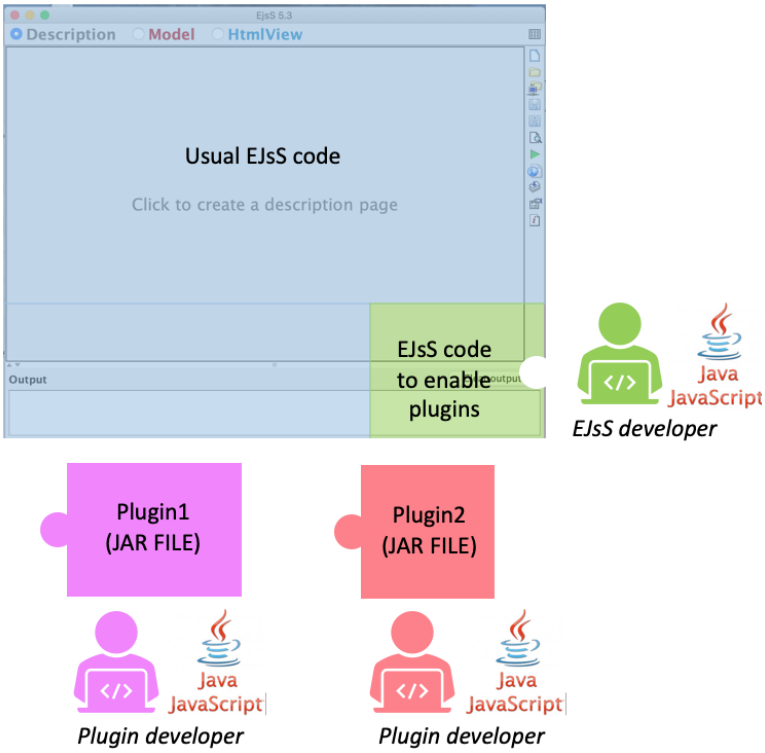
In this section, we describe different methodological aspects of the work presented in this paper. In particular, we first describe how the plugins are to be used from the point of view of different types of EJsS users and afterwards, we explain the methodology followed to determined which changes and which benefits should be supported by the plugins.

3.1. Methodology from the Point of View of EJsS Developers and Users

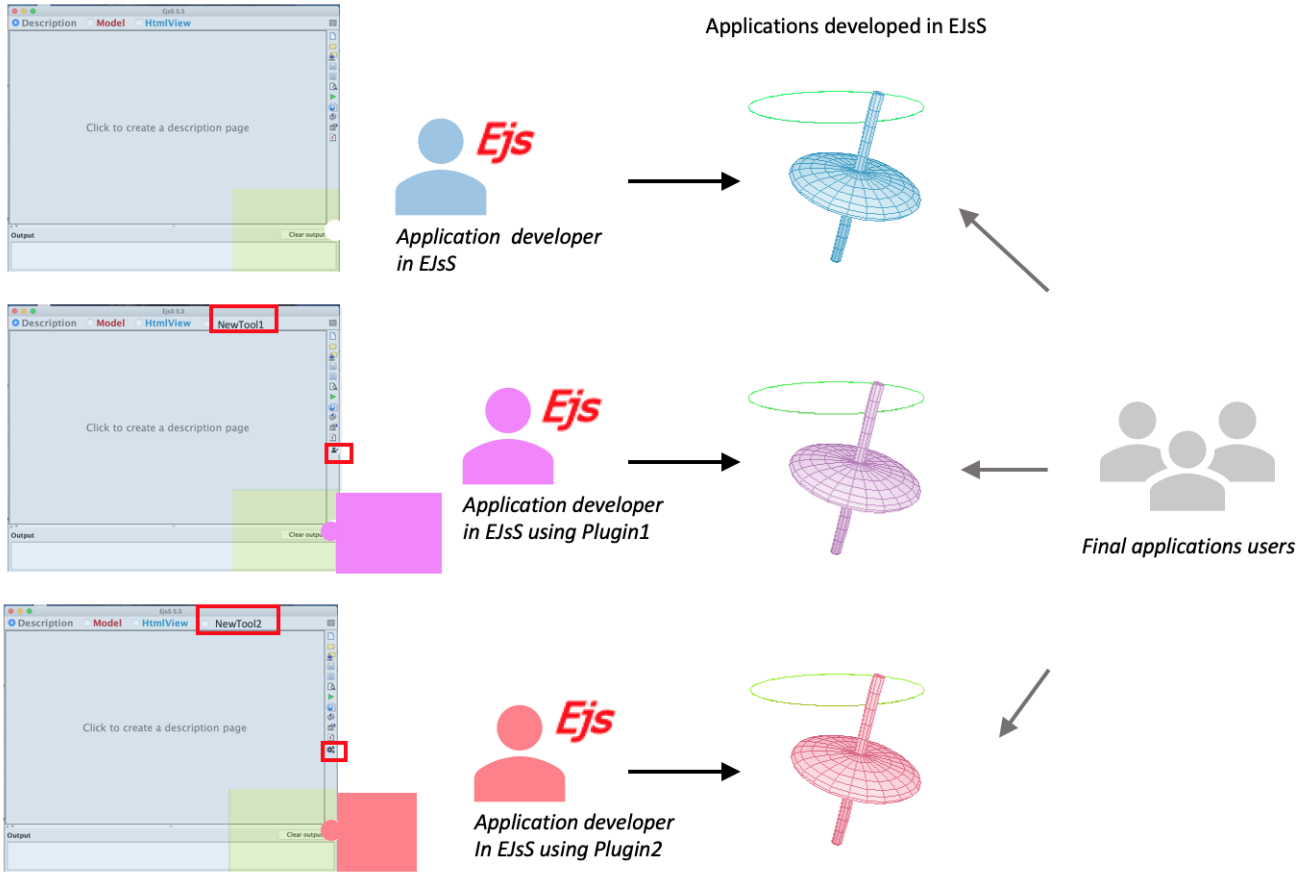
The methodology presented in this paper is intended to allow the adaptation of the functionality and GUI of EJSs to new domains. To be able to do it, two different programming processes are required.

- On one hand, we (as EJsS users with advanced programming capabilities and previous experience in the development of EJsS code) have already modified the core of EJsS to let it load and include the new functions and visual elements that will be bundled in a *Plugin* into EJS.
- On the other one, users of EJsS with advanced programming capabilities require writing the code of the *Plugin*, (a) considering the new functionality and visual elements/windows that they want EJsS to have, and (b) adhering to the guidelines that will be presented in the following sections.

Both programming processes are schematized in Figure 2a, using light blue to represent the part of the EJsS code that already existed (previously to our changes), light-green for the code that we have included into EJsS to handle the *Plugins*, and purple and red for the code associated with two different *Plugins*, which are bundled as JAR files. In addition, the code developers (named in Figure 2a EJsS developers and *Plugin* developers accordingly to the type of coding they have performed) are represented by different human icons colored as the code they have created.



(**a**) Programming process for the new *Plugins*



(**b**) Perspective of normal users of EJsS

Figure 2. Methodology from the point of view of EJsS developers and users.

From another perspective, in Figure 2b we show how the normal users of EJsS (i.e., those without advanced programming capabilities that develop applications—including simulations—in EJsS and that are therefore called application developers in the figure) benefit from the existence of the new *Plugins*. At the top, regular EJsS users (in light blue) that want to develop simulations with the normal EJsS functionality use EJsS as always, without taking advantage of the *Plugins*. At the middle and at the bottom, new types of EJsS users (in purple and red) exploit, with the purpose of developing new types of applications in EJsS, the new *Plugins*, using the new functionality that appears in EJsS GUI (framed in red) after only putting the *Plugin* JAR files in a specific folder of EJsS. Moreover, it is worth noting that an application developer can use several *Plugins* simultaneously to create its applications. At last, the final users (in gray) of any applications developed within EJsS are not aware of how it has been developed and can take advantage of all the properties that their developers (in EJsS) have included in them.

3.2. Methodology to Determine Which Changes and Benefits Should Be Supported by the Plugins

We started our research wondering how we could improve the way people interact with EJsS. That apparently simple question is easier to state than to answer, but we took it as an inspiration to search for ideas and tips that can help us understand how other people use EJsS, and identify the weaknesses that we could address to enhance the tool. In the search for an answer, we extracted information and conclusions from different sources:

- The authors' experience of many years using EJsS from different perspectives: as normal users (developing simulations and remote labs front ends [23–27,38]) and as software developers (either modifying EJsS, creating *Model Elements* or integrating with external libraries).
- An informal extraction of needs and requirements of EJsS users, from direct talks with people involved at different levels, and through the study of *Model Elements* of other developers, looking at what they have done and how. This is somewhat subjective, but an invaluable source of knowledge.
- Based on the study of many remote lab applications that were gathered and analyzed for different literature reviews [39,40].

With all the previous information, we identified some patterns or difficulties that many people faced in their use of EJsS. We cite here the most relevant that will be discussed in the rest of the paper.

- *Version conflicts*. We found out that it is frequent to have problems due to the use of different versions of EJsS, or extensions that provide extended functionality which are not up to date.
- *Distribution*. Moreover, sometimes updating such extensions can be problematic, in particular when the user must copy one or more files manually to a specific folder.
- *Unspecialized interface*. Though EJsS is easy to use, and the *Model Elements* allow the provision of new functionality, sometimes it can be difficult to completely integrate them in the interface of EJsS, what was degrading the experience of some users.

The rationale of the extension mechanism that we will explain in detail in the following sections is to address the identified difficulties and provide a better interaction with EJsS. Moreover, in Section 4.3, we explain the methodology that the developers of the new *Plugins* must follow, and in Section 5 we discuss and illustrate how the *Plugins* are created and used through several case studies, with the intent that newcomers can confidently face these tasks.

4. Extension Plugins to Enhance EJsS

In this section, we overview the main characteristics and allowed adaptation of the extension mechanisms presented in this paper, describe how we have modified EJsS to support them, and explain how to implement them properly.

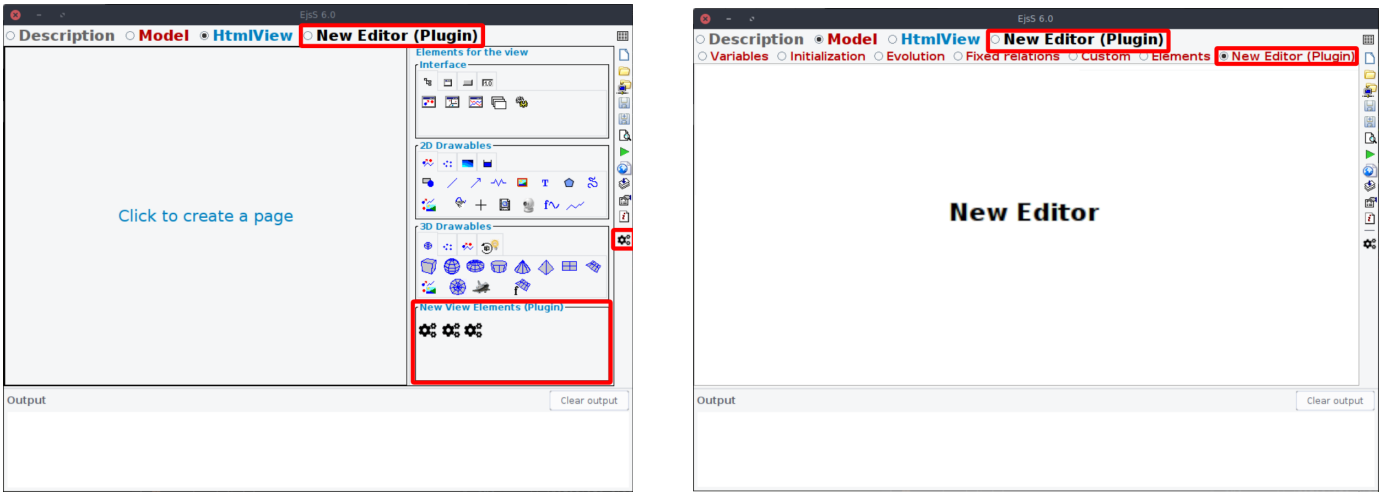
4.1. Characteristics Overview and Allowed EJsS Adaptations

The extension mechanism presented in this paper is based on the concept of *Plugins* that must allow (1) to centralize the addition of different capabilities in EJsS, and (2) to customizet/adapt the GUI of EJsS to users with different needs.

Moreover, to ensure the usability of the approach, the new plugin mechanism must also fulfill the following requirements:

1. *Simple distribution*, to facilitate the plugin download and installation.
2. *Self-contained*, to ensure that a plugin contains all the libraries and components it needs to work and that it does not depend on additional actions in the side of the user.
3. *Non-intrusive*, to be able to completely revert its installation if needed.

Before describing the new *Plugins* mechanism presented in this paper is worth noting that it is a generalization of the EJsS *Extensions* provided in [41], which encapsulate different *Model Elements* in different EJsS extension bundles. The novelty of our approach consists of allowing the modification, at some extent, of other parts of the EJsS layout, by providing new *Main Editors*, or new elements that can be nested inside the *Model* and *HTMLView Editors*, or inside the *Buttons Bar* (as the highlighted red framed areas of the GUI of the new version of EJsS presented in Figure 3 show). In other words, we inherit the possibility of bundling new functionality within EJsS, by not only supporting the inclusion of new *Model Elements* as previously, but also by giving access to other tools that can be helpful for some EJsS users.



(**a**) New elements in the Main Editor Menu, View Menu and Button Bar

(**b**) New elements in the Main Editor Menu and in the Model Menu

Figure 3. Customizing options of EJsS GUI and functionality with Extension Plugins.

The benefit of the approach, if used properly, will allow the adoption of EJsS as a tool to centralize different tasks that the users should perform for their projects. Let us illustrate it with an example: using EJsS as a central tool to set up a remote laboratory. In this case, the *Main Editor* menu can include a new tab to give access to a panel that facilitates the configuration of the communications with the remote system. It is also possible to add new *Buttons* that give instant access to actions that the instructor performs frequently, such as configuring the authentication credentials to connect with the remote lab. Of course, new *Model Elements* (which encapsulate functionality relevant to the remote lab) or new *View Elements* (with graphical representations often used in it) can also be added to the EJsS GUI and grouped in a distributable *Plugin*, bundled in a JAR file.

4.2. Supporting the New Plugins Infrastructure

During the initialization process of EJsS, it creates all the elements of its interface that will be available to the user. In fact, since 2016, it already performs a minor customization, because it allows users to select between two possible versions (Java and JavaScript), each one with a different *View* panel and slight differences.

To add support for the *Plugin* mechanism, we have modified EJS initialization process, to make it incorporate, roughly, the following steps:

1. *Searching for Plugins*, i.e., the plugin JAR files that the users have placed in folder `workspace/config/CustomPlugins`.
2. For each plugin JAR file:
   (a) Loading resources (panels, buttons, *View Elements*, and *Model Elements*).
   (b) Adding panels to the *Main Editor*.
   (c) Adding buttons to the *Button Bar*.
   (d) Adding *View Elements* to the *HTML View Editor*.
   (e) Adding *Model Elements* to the *Model Editor*.

From a conceptual point of view, there is a clear separation between the functionalities that the *Plugin* provides and the structures that must adhere to with the purpose of being integrated within EJsS. Regarding its functionalities, they obviously depend on the particular aims of each *Plugin* and cannot be discussed here. With respect to the integration, however, it is mostly a systematic task that does not depend on the nature of the *Plugin*, but rather on the type and quantity of elements that is composed of.

As explained before, the elements the plugin aggregates are instances of the following ones: *Main Editor*, *Bar Button*, *Model Element* and *View Element*. Moreover, we also allow them to be instances of the *Nested Editor*, to support the inclusion of multiple editors within a new *Main Editor* tab. Additionally, each *Plugin* must implement a JAVA interface, named `Plugin`, that informs of and lets EJsS use its capabilities. For each element included in the *Plugin*, it must also provide the implementation of an interface or extend a base class dependent on the type of element. Moreover, *Plugins* are distributed in the form of single JAR files that contain everything that is needed for running them inside EJsS, a way of proceeding that fits perfectly with the simple distribution and self-containment requirements stated at the previous section. Finally, although the installation of a new *Plugin* is extremely easy too (and consists of copying the JAR file into a specific folder), the Plugin Manager, presented as the first case of study of Section 5, provides extended features to manage *Plugins* within the EJsS interface.

To support the extension, we have had to modify the EJsS code, as discussed in Sections 2.2 and 3.1. However, this is a one-time modification of EJsS, already incorporated in version 6.0, that makes it capable of supporting the *Plugin* infrastructure, i.e., of discovering, loading and using new *Plugins*. Hence, the impact in future changes of EJsS are bounded to that part of the code, reducing the maintenance effort. The other side of the coin are the *Plugins*, which would adapt EJsS to a specific application, following an implementation template that provides the means to add new functionality and customize EJsS interface. Since the *Plugin* architecture requires the user to be able to modify how things are presented in EJsS, it is clear that we must introduce some changes in the way EJsS builds its interface. However, we have tried to be so little intrusive as possible, because we expect the major changes to be done at the *Plugin* programming side.

4.3. Plugin Implementation and Architecture

A *Plugin* implementation is composed of JAVA and JavaScript code, which follows a predefined structure that allows EJsS to load, instantiate, and communicate with it. That way, the tool is aware of the *Plugin* and can benefit from the capabilities supplied by its code.

Every *Plugin* must have a main class which implements the JAVA interface `Plugin`, providing a set of methods required by EJsS. This is the only essential one, but there are many EJsS-provided JAVA classes and interfaces that are at disposal of the developer who

wants to build an extension. The most frequently used are the class `Editor`, which provides a main or nested editor, and the class `AbstractModelElement`, which provides a template to create new *Model Elements*. The architectures of the examples described in the following section are later depicted within it.

Regarding the *Plugin* functionalities, we should remember that EJsS is a tool originally intended to design simulations and generate its corresponding running HTML and JavaScript code. Hence, the *Plugin* can provide actions that will run either during the designing or running phases, the former coded in JAVA (to be used within the EJsS layout/functionality) while the latter implemented in JavaScript (to be used during the simulations).

Moreover, the development of a new *Plugin* generally consists of two differentiated tasks: the implementation of the functionalities that the plugin adds to EJsS, and the incorporation of that functionalities into the template that EJsS provides through its extension mechanisms. Although the functionalities are highly *Plugin* dependent, their incorporation into EJsS is mainly a repetitive task. It is important to distinguish clearly between both tasks, since doing so properly will facilitate the creation of the *Plugin* and improve the maintainability of its code.

Independently of the complexity of the *Plugin*, the development of any new one adheres to the following steps:

1. *Integrating it with EJsS*: creating a new class that implements the JAVA interface `Plugin`.
2. *Developing its Graphical User Interface*: creating the *Plugin* GUI elements (e.g., editors and buttons).
3. *Implementing its core functionalities*: programming the tools provided by the *Plugin* to EJsS.

That methodology is followed by the case studies that are discussed in the following section.

5. Case Studies

This section presents three case studies of increasing complexity that illustrate the philosophy of the *Plugin* adaptation mechanism and give a practical insight on a *Plugin* development. The first two cases have been selected to show the benefits from adding a simple functionality and a small GUI change to EJsS, while the third is used to explain how to revamp it to present a more friendly interface to a specific type of user.

Prior to each *Plugin* implementation, there must be a design phase to specify its functionalities and identify the EJsS GUI modifications and components to add. Hence, each case study begins with a statement of the necessities that motivate each extension, follows with an informal specification of the requirements that gather the important technical and functional aspects to be considered, and ends with the actual implementation of the JAVA and JavaScript code that materialize the *Plugin*.

Finally, although it is out of the scope of this work, we want to remark that an essential part of a successful *Plugin* design (which consequently should not be neglected) is how the EJsS GUI should be modified for each new application. In other words, the modifications implemented in each *Plugin* will make the difference between enhancing or degrading the experience of the users of the adapted versions of EJsS.

5.1. Plugin Manager

The Plugin Manager is a simple yet useful *Plugin* that illustrates the philosophy of our approach. As we have already explained, each plugin is contained in a JAR file that must be put in a certain folder to let EJsS use it. Though the process is straightforward, it still requires the intervention of the user outside the tool. Clearly a better option would be to have a Plugin Manager integrated within the EJsS development environment.

To develop the Plugin Manager, we start listing the basic functionalities it should provide:

- *Installing or uninstalling a new Plugin*, provided in a JAR file, automatically. In other words, it must let users copy/delete, from EJsS GUI and in/from EJsS designated *Plugin* folder, each *Plugin* JAR file.

- *Enabling and disabling Plugins*, without uninstalling them, to let users decide if a *Plugin* should or not be loaded into EJsS, without needing to delete its corresponding JAR file.
- *Showing the list of installed Plugins*, with detailed information about each of them.

We want to make these functionalities accessible through a simple interface that consists of: a list that shows the installed *Plugins* and lets the user enable or disable them, an area to show detailed info on a specific *Plugin*, and buttons to install/uninstall *Plugins*. With these three elements, we have the ingredients to build the graphical interface of the Plugin Manager, but still need to find the better place to put it inside EJsS. Since the Plugin Manager that we are developing is directly related to the EJsS environment, rather than to a simulation, it is appropriate to make its GUI accessible from the *Buttons Bar*, by including a new button that opens a dialog window with the interface of the Plugin Manager.

After identifying the graphical elements that the *Plugin* must provide (the button and dialog window), we must implement several classes. The class `PluginManager` implements the interface `Plugin` that EJsS needs to be aware of the *Plugin*, and provides a `PluginInstallerButton` that builds the GUI that allows the interaction with the *Plugin*, instantiating a `PluginInstallerEditor`. The classes `PluginInstaller` and `PluginModel` implement its core functionality: the former provides methods to actually handle the installation, uninstallation, enabling or disabling of *Plugins*; while the latter loads and parses the information (authors, purpose, how to use it,...) of each *Plugin*. In Figure 4, we can observe the relationships among the classes, and that there is a clear separation among the code that provides the functionality (implemented in `PluginInstaller` and `PluginModel`), the code that builds the GUI (`PluginInstallerButton` and `PluginInstallerEditor`), and the code that allows the plugin integration with EJsS (`PluginManager and Plugin`). This division will be followed in all the examples of the paper, because it is a good programming practice for developing and maintaining the *Plugins*.

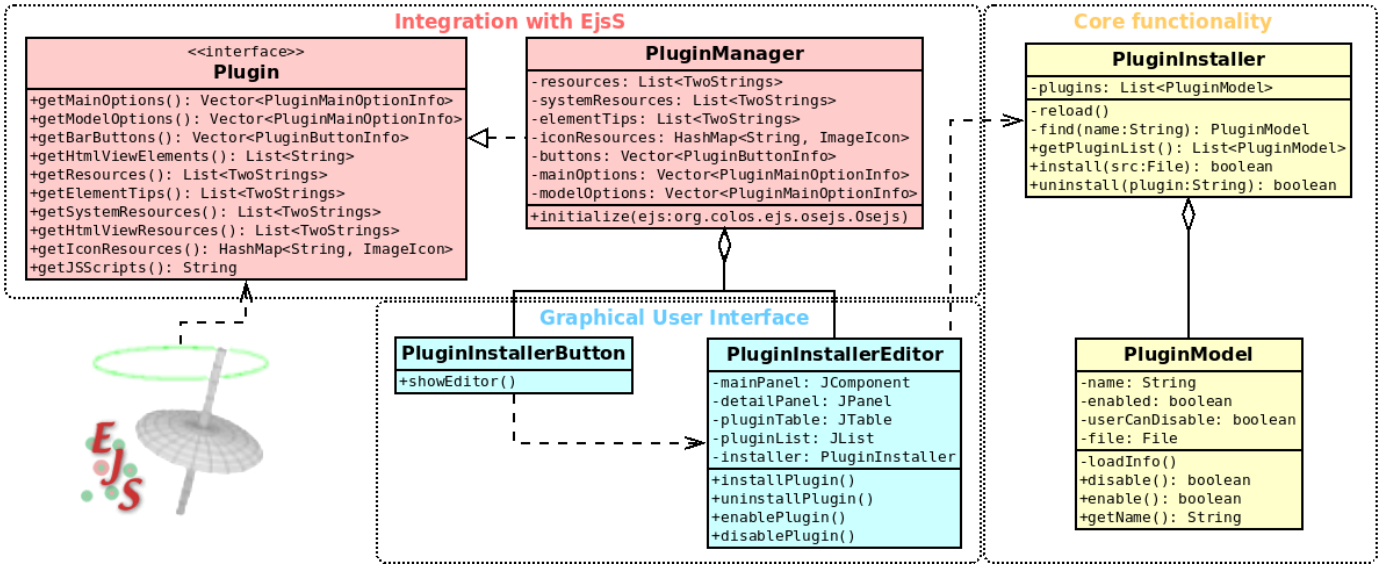
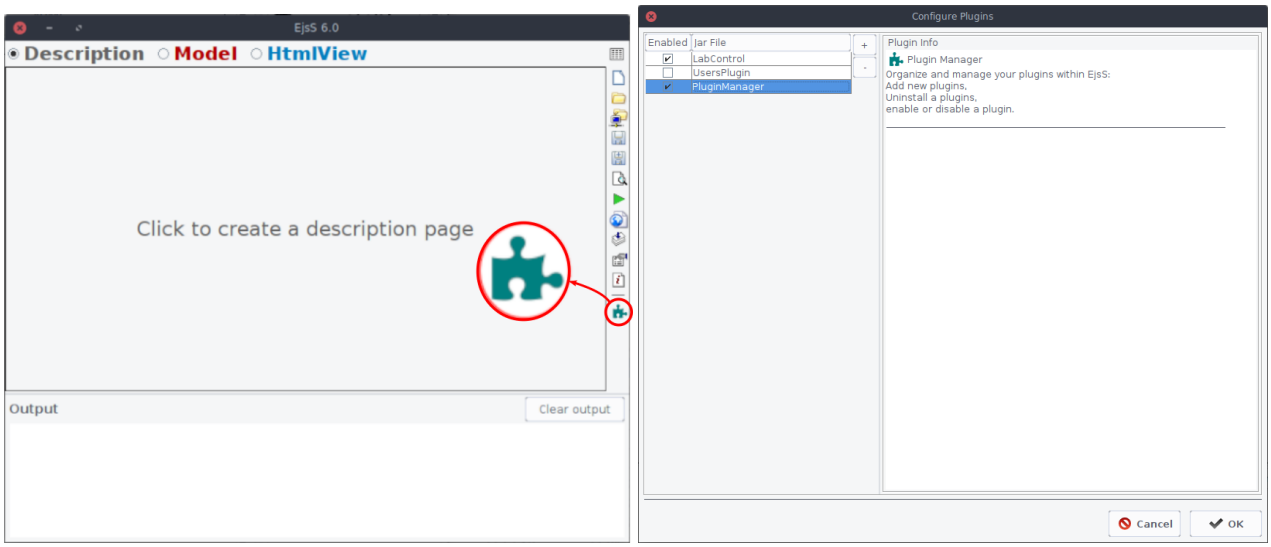


Figure 4. UML class diagram of the Plugin Manager.

Finally, and after implementing the different classes, bundling them in the archive `PluginManager.jar`, copying it in the appropriate folder and launching EJsS, EJsS main window includes, as Figure 5a shows, the plugin button, which opens the dialog window displayed at Figure 5b. Besides, it is worth noting that the Plugin Manager allows the enabling/disabling of any of the *Plugins* installed in the designated folder except itself, because disable *Plugins* are not loaded in EJsS, and therefore a disable Plugin Manager will not be available to let the user enable/disable itself or any other *Plugin*.



(**a**) EJsS screenshot with the new button (**b**) Dialog window to interact with the plugin

Figure 5. EJsS adapted with the Plugin Manager.

5.2. Users Manager

The Users Manager Plugin is another example of a *Plugin* that extends the capabilities of EJsS, in this case to provide EJsS users with a tool to manage the database of users of a server that provides access to a remote laboratory. This database is used, for security reasons, to control the access of users who can connect, carry out experiments, monitor the lab or perform maintenance tasks. With every new academic year, the teacher (or person in charge of the lab maintenance) must edit the user database, revoke permissions from students that have passed the course and incorporate the new students to the access list. To facilitate all these tasks, it seems reasonable to integrate the user database editing into the EJsS workflow, making instructors life easier, a tedious process more friendly and less error prone.

To develop the Users Manager, we start again stating what this *Plugin* must support:

- *Retrieving the user database from a remote server*.
- *Displaying the user information* (username, permissions, etc.).
- *Modifying user data.*
- *Importing users from an external file.*
- *Sending database updates to the remote server.*

After determining the *Plugin* functionality, we focus on its interface: the user needs to be able to configure the server access (i.e., to provide the *URL* of the server and the *credentials* of the users), to request/update the user database from the remote server, and to edit the user data. All these elements can be put in the same dialog window that should be accessible from EJsS. In this case, we consider that the best option is to make them accessible from a new tab added to the *Main Editor Panel*, since from the remote laboratory perspective it can be equally relevant to handle its *HTMLView* correctly and to update its users efficiently.

To implement this *Plugin*, we need to take into account the lab server implementation, as it determines the communication protocols and Application Programming Interface (API) that will be used to create it. However, since a thorough discussion of the lab server implementation is not essential to understand this section (and can entangle the vision we want to provide through the examples), we detail it in Section 6, for those readers who are interested in using the remote lab infrastructure that we have developed to test the new *Plugins*.

During the implementation of this *Plugin* we have developed the classes that are schematized in Figure 6. In particular, the class `UsersManagerPlugin` implements the

interface `Plugin` (to let EJsS be aware of it) and registers the class `UsersEditor`, which is responsible for building the layout of the new main Users editor/manager. Moreover, the class `UserUpdater` communicates with the server (using *Apache HttpComponents*), parses and generates the exchanged JSON structures, and imports the database CSV file (using *Apache Commons CSV*). Finally, the class `UserModel` encapsulates the user data information (username, password, etc.).

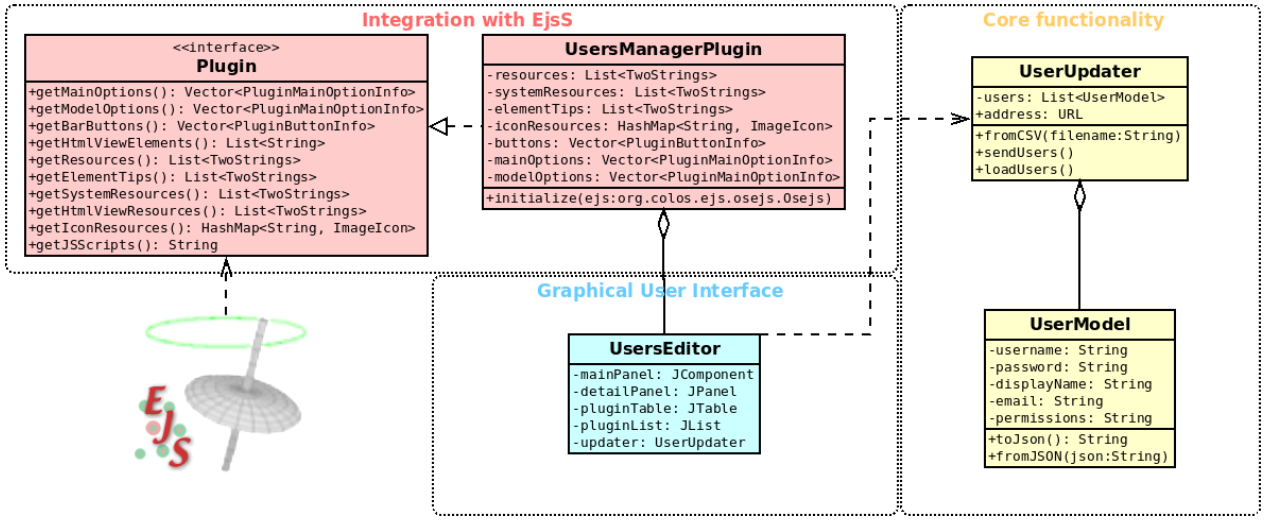


Figure 6. UML class diagram of the Users Manager.

Finally, after the *Plugin* is implemented and installed, EJsS displays, as Figure 7 shows, the new Users tab in the *Main Editor Panel* and, after selecting it, the corresponding Users Editor window. Hence, from the point of view of the person in charge of the remote lab, EJsS is now offering a specialized environment that allows the management of the remote laboratory users, avoiding the necessity of additional software tools to do it.

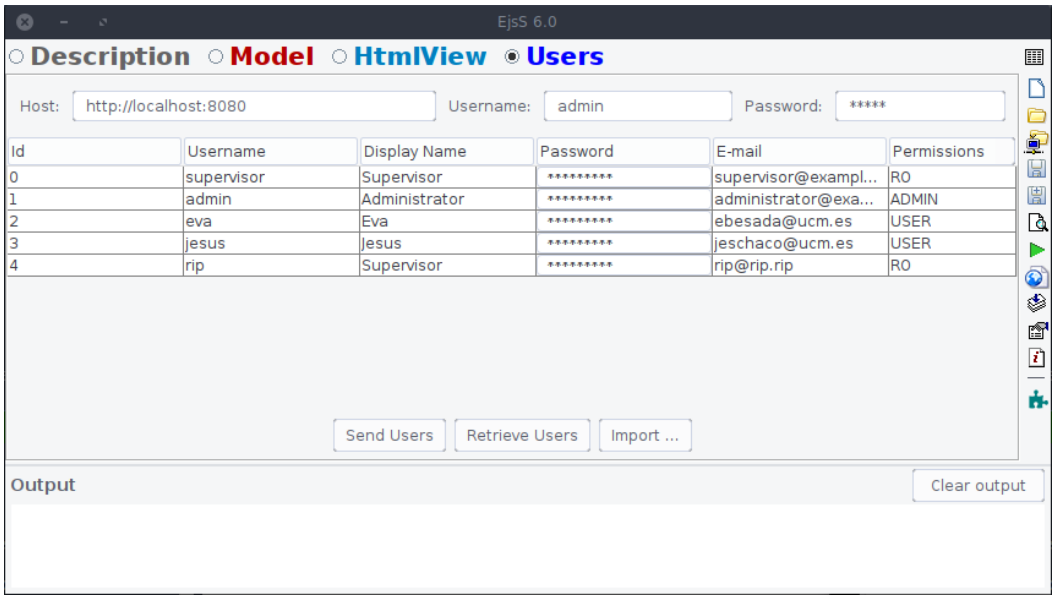


Figure 7. EJsS adapted with the Users Manager Plugin.

5.3. Remote Laboratory Management

The next case presents a more complex *Plugin* to use EJsS as an integrated development environment for setting up experiments supported by the remote lab server presented in Section 6. In this case, the *Plugin* modifies several elements in EJsS GUI and functionality

to make it a friendly interface for the lab designers, who will be able to do from it the following tasks:

- *Configuring the connection* (url, credentials, ...) *of different lab instances*, to be able to gain access to them.
- *Exchanging data with the lab instances during the experiments*, to be able to interact with the components of the remote lab, monitoring the evolution of its variables and modifying its configurable parameters.
- *Reusing visual and interactive elements designed to automatize and homogenize the experiments interface*. The rationale behind this task is that remote labs (especially those included in the same courses) can share many elements of their graphical interface (e.g., login area, execution buttons or data visualizers) that can be reused to (1) facilitate the lab developer task and to (2) present visually coherent experiences to the students (to optimize their time of study and with the system).

Regarding the *Plugin* interface, we must decide which set of visual elements will provide the previous functionality. To show the remaining types of *Plugin* extensions supported by the new version of EJsS:

- We have created a new subpanel, named *Remote Labs* and implemented with class `LabControlModelEditor`, inside the main *Model Editor*, where the lab developer must provide the connection information (server IP address and connection port) of the remote labs that would be accessible from EJsS. Moreover, after connecting with the lab, this panel will also show information about which variables of the remote lab will be accessible (readable and/or writable) from EJsS.
- We have developed a new *Model Element*, named *Lab* and implemented with class `LabControlModelElement`, to create, for each of the accessible remote labs, a variable that encapsulate the methods that give access to the remote lab. Thus, different labs will be represented by different instances of the `LabControlModelElement`, that will be accessible from different parts of EJsS (e.g., from *Model Custom* code or *View* elements). Arguably, the `LabControlModelElement` for the lab instances could introduce redundancy in the remote lab configuration. To avoid duplicity, the configuration of the lab is only included in the `LabControlModelEditor`. Besides their purpose is clearly different: the `LabControlModelEditor` is used to know which variables of the lab are accessible from EJsS, while the instances of `LabControlModelElement` are actually used to access them.
- We have developed two new *View Elements*. The first one is a button bar to control the execution of the remote lab during each student experiment (i.e., it starts, stops and resets the experience, and connects and disconnects from it). The second one is a grouped set of labels and editable fields to let the student provide its user and password.

To implement this *Plugin* we have developed the classes that are represented in Figure 8. First, the Java class `LabControlModelElement` (and its homonym JavaScript object that provides the runtime functionality) implement a *Model Element* following the traditional extension method (see Section 2.2). Hence, they not depend on the new *Plugin* mechanism presented in this paper, and could actually be used without it. However, the *Plugin* extends the functionality of the *Model Element* to provide an enhanced interface to manage/build remote labs within EJsS. In fact, this approach could be a good way to reuse existent model elements, while providing a better EJsS integration layer. Second, the class `LabControlPlugin` implements the interface `Plugin` and is responsible for instantiating and registering the other components: a `LabControlModelEditor` (which provides the Remote Laboratory configuration panel) and two *View Elements* (`LabControl` and `LabLogin`, for the lab control buttons bar and the student login information). Please note that while the previous components (`LabControlPlugin` and `LabControlModelEditor`) are implemented in Java, the latter ones (`LabControl` and `LabLogin`) are JavaScript objects that will be directly included into the simulation, which allows for more control and flexibility. Since this *Plugin*

adds more functionalities and graphical elements to EJsS than the *Plugins* presented in the previous sections, it has also grown in complexity and number of classes/components.

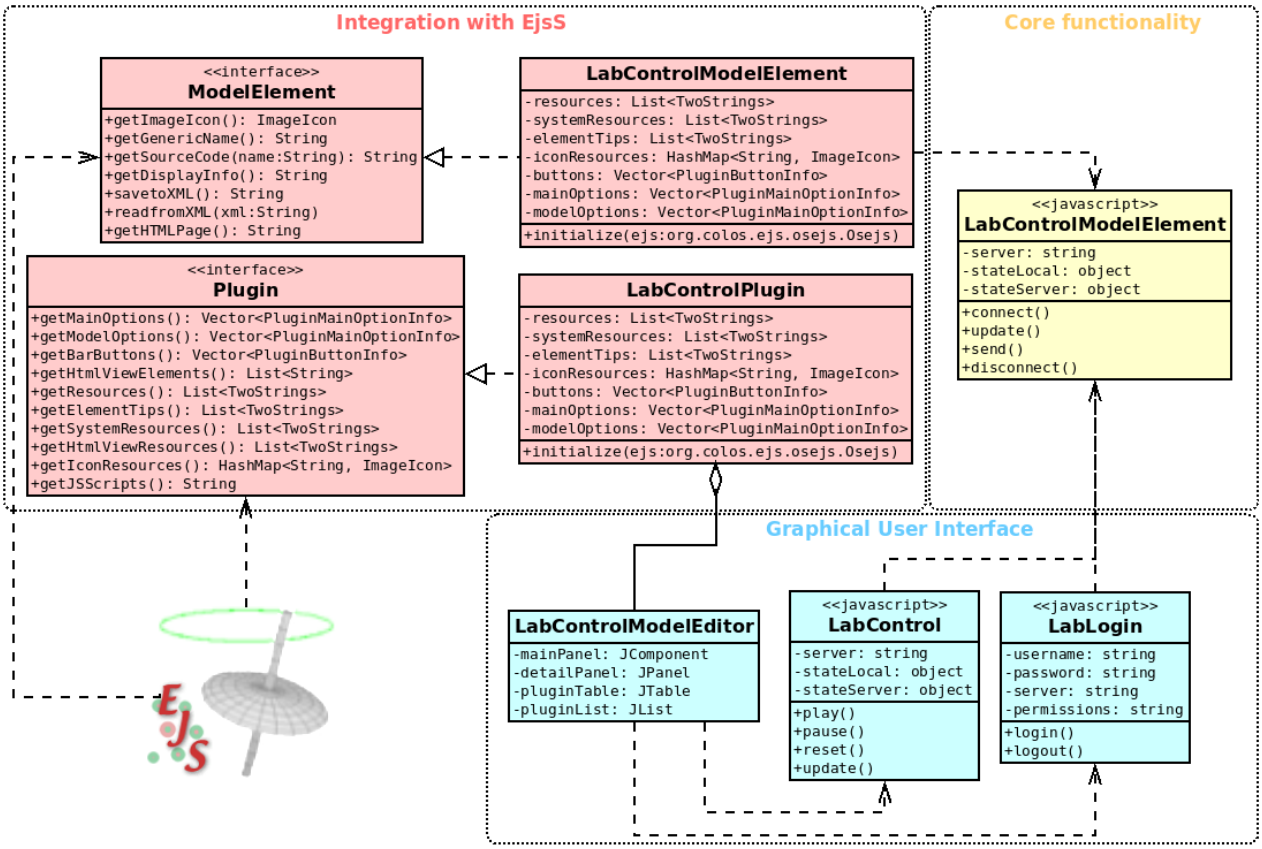


Figure 8. UML class diagram of the Remote Laboratory Manager.

After implementing and installing the Remote Laboratory Management plugin, EJsS layout looks different, as the screenshots of Figure 9 show. On the front screenshot we can see, on the left, that the information of three remote labs has been provided under the *Remote Labs* tab, and on the right, the details (connection and variables) corresponding to the selected lab instance on the left. On the back-left screenshot, we can observe the instances of the three *Lab Model Elements*, which can be used in the code and visual elements of the application that the user will develop with EJsS. Finally, on the top right screenshot, the *HtmlView Editor* shows, on the left that its tree of elements incorporates a student login interface and a lab control button bar, and, on the right, the new View Elements that are available to the developers.

Finally, this study case demonstrates the versatility of the new *Plugin* extension mechanism: it lets *Plugins* developers accommodate the EJsS interface and functionality to new users and scopes, which will benefit from the usual simulation and visualization tools of EJsS, and from the extended capabilities supported by each *Plugin*.

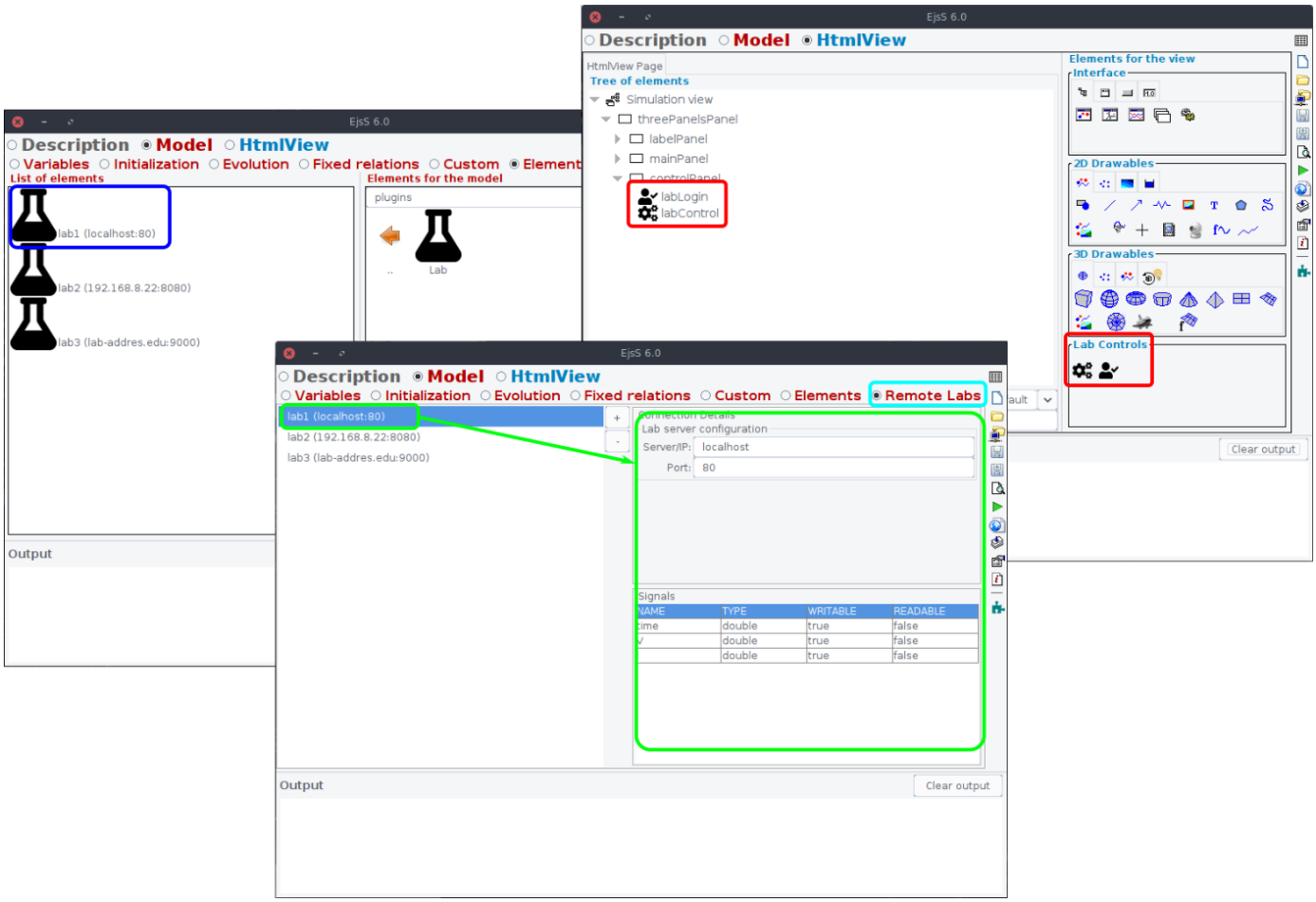


Figure 9. EJsS adapted with the Remote Laboratory Management.

6. The Remote Laboratory

To let readers take advantage of the developed *Plugins*, this section (1) describes the main characteristics of the remote laboratory which has been specifically implemented to test the Users and Remote Laboratory plugins (described in Sections 5.2 and 5.3), (2) outlines how to adapt the code of the Lab server for different applications, and (3) explains how to set up the laboratory from EJsS.

6.1. Overview of the Remote Laboratory

The interactions with the remote laboratory are supported by a lab server that we have implemented in Python [42], using the lightweight web development framework Flask [43] and a REpresentational State Transfer (REST) API [44]. We have implemented the server in Python because it is a convenient, popular and open-source programming language, used by a huge community of software developers and supported by many software libraries of different purposes, including several ones that facilitate the interaction with the hardware of a remote lab through different communication ports. We have selected Flask, instead of other popular and featureful web development frameworks such as Django, Tornado and CherryPy, since Flask is a popular framework for web applications developed in Python; it allows the easy set-up, testing and debugging of a minimal server; and can be deployed in different OS and platforms, such as a PC or a Raspberry PI, either as a standalone server or integrated into another web server such as Apache or Nginx. We have selected the REST API for letting the client — EJsS — interact with the lab server that we have developed, since it provides a standardized way to support HTTP requests, which is already used in other remote lab implementations [45–47]. Finally, as a discussion

of laboratories technologies is out of the scope of this paper, we suggest the following works [4,40] to those readers specifically interested in them.

The remote lab server must provide the functionality required (1) by the User Manager and Remote Laboratory Manager plugins, and (2) by the experiment itself (dependent on the purpose of each lab).

More in detail, to cover the functionalities related with the *Plugins* (which have been explained in Sections 5.2 and 5.3), the server API must expose several capabilities, including the users database management required by the Users Manager plugin, and reading/writing the values of the server objects (e.g., inputs and output signals, server variables, parameters) that need to be accessible from the Remote Laboratory Manager plugin. These features are encapsulated into the API GET/POST methods summarized in Table 1, which are invoked as web services from EJsS.

Additionally, from the perspective of the experiments themselves, the code of the server provides the backbone to create a functional remote laboratory. Essentially, it acts as a middleware that provides the means to connect the client (EJsS) with the code that accesses the hardware, including the student authentication and the management of the signals. The task of the laboratory developers is to fill the gaps between the middleware (server) and the laboratory hardware with the adequate code. For example, a remote laboratory deployed on a Raspberry PI could use the module `wiringPi` to access its peripheral ports (i.e., its general-purpose inputs/outputs) or `pyserial` to communicate with serial/USB devices.

Table 1. Methods of the REST API exposed by the remote lab server.

Method	Path	Purpose	Used by
GET	/users/get	Request the user database	Users Manager Plugin
POST	/users/set	Update the user database	
GET	/signals/info	Request signals description	Remote Lab Manager Plugin
GET	/signals/get	Request the values of one or more signals	
POST	/signals/set	Update the values of one or more of signals	

6.2. Adapting the Server Code for a Particular Application

To set up the laboratory server, we need to provide the code that implements the specific functionality of each lab, and then integrate it with the generic functionality of the server by providing, in the lab configuration code, the information (name, type and accessibility) of the experiment signals that will be exchanged with EJsS and the handler to the function that the server will invoke when it receives a request to read/write any of those variables.

In order to illustrate this process, we will set up a remote lab that controls an analog single-input single-output system. To do it, we first develop a Python module (named `HardwareInterface`) that communicates with a Digital-Analog and an Analog Digital Converter (DAC and ADC), which are both connected to an I2C port. For that purpose, our Python module provides two functions: `readADC():value` and `writeDAC(value)`. Next, we map in the lab configuration file the hardware signals to the server signals that will be exposed to EJsS. For that purpose, and as Figure 10 shows, we create (1) a `SIGNALS` array that contains an object for each signal (in particular, `input` and `output`) with the fields (`name`, `type`, `read` and `write`) and (2) a `HANDLERS` array that contains the mapping between each variable and the functions that will actually do the reading or writing (in particular, `input` is mapped to `readADC` and `output` to `writeADC`). Hence, when the server receives a request to read the variable `input` it will call the `readADC` function.

Moreover, another thing that needs to be configured is the user database. To do it we have two options: (1) do it from EJsS (exploiting the Users Manager plugin and the fact that our server includes by default and Admin user) or (2) modify it by hand, editing the file `users.db` which is a human-readable JSON object.

```
from control.HardwareInterface import readADC, writeDAC

class Config(object):
    DEBUG = False
    TESTING = False

    SIGNALS = [
        {'name':'input', 'type':'double', 'read':True, 'write':False},
        {'name':'output', 'type':'double', 'read':False, 'write':True},
    ]
    HANDLERS = [
        'input': {'read': readADC},
        'output': {'write': writeDAC}
    ]
}
```

Figure 10. Configuration file of the remote laboratory server.

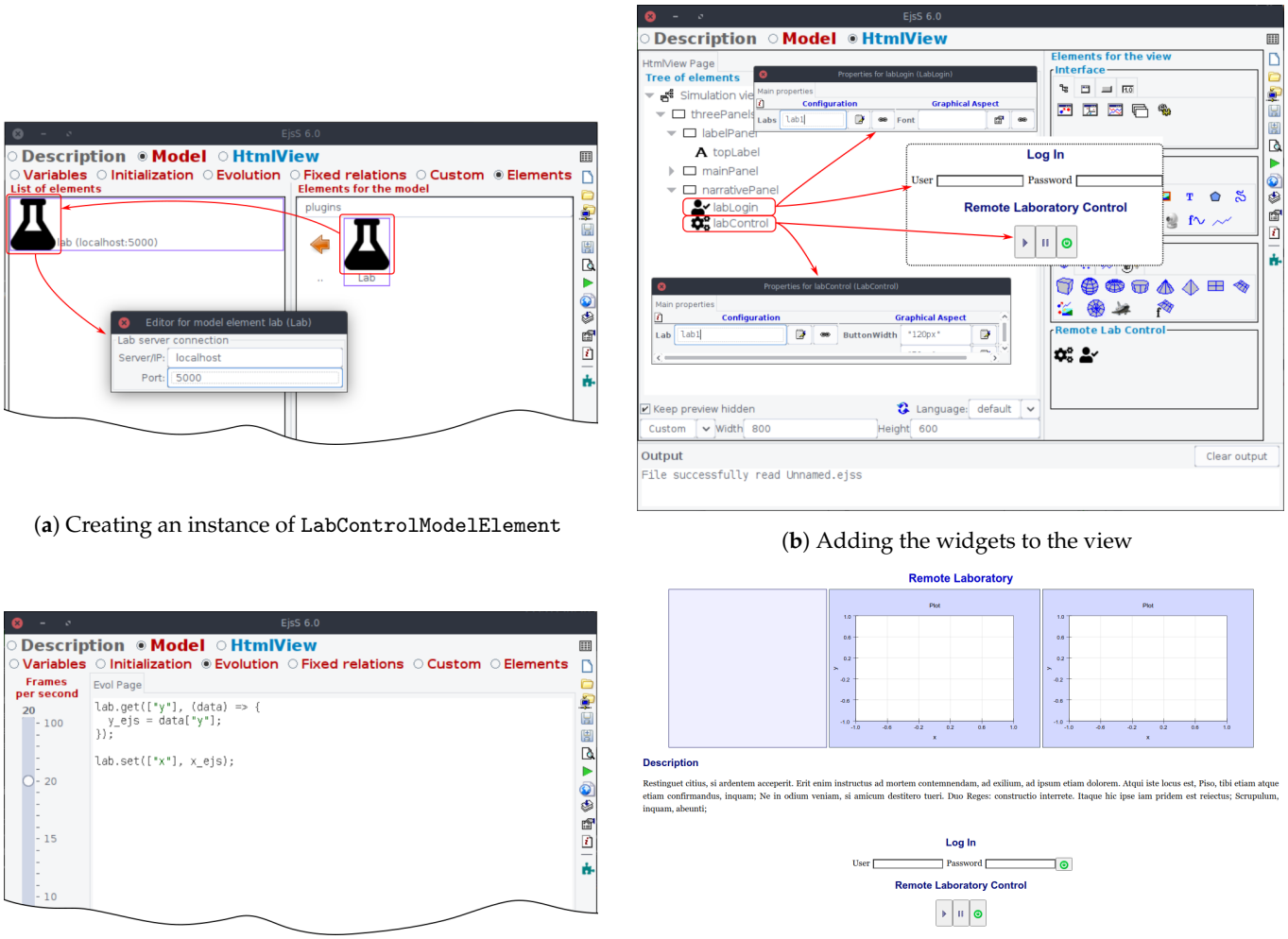
6.3. Setting up the Lab from EJsS

Once the lab server is well configured and put in production, laboratory developers need to configure the lab client that is used by the students to interact with the lab hardware. We do it from EJsS, enhanced with the brand-new Remote Lab Management and User Management plugins.

The first step is to open EJsS and create a new project or load the existent simulation that we want to use as the front-end of our new lab. Then, and as Figure 11a illustrates, we configure a new lab instance that represents our remote lab by instantiating a `LabControlModelElement` that encapsulates, as explained in Section 5.3, the API exposed by the lab server. Now that EJsS knows where the lab is located, we may want to look at the `LabControlModelEditor` that shows the info retrieved from the lab (including the signals that the server exposes, which types they have and whether they can be read, written or both). Should we want to have access to more than one lab server (e.g., two or more replicas of the lab to increase the availability of the experiences from the client side) we could repeat this process to instantiate more labs.

Next, we must develop (or adapt) the client GUI. Hence, we go to the *HTMLView Editor* and add the elements that allow the interaction with the lab to a new view (or to an existent view, in case we already have one). With that purpose, and as Figure 11b shows, we add a new login widget to the laboratory front-end/client that we are developing, with the purpose of letting the students authenticate and access to the remote laboratory. This widget, `LabLogin`, which relies on the lab object instantiated in the previous step, provides to the finally developed view, displayed partially in Figure 11b and completely in Figure 11d, with two fields for the credentials (*user* and *password*).

At this point, we already can generate a web application that is able to connect to the remote lab. However, it is still not very useful. To improve it, we add, as Figure 11b depicts, the lab control bar. Again, its functionality is provided by the instance of the lab model element created in the first step of this section. After adding the lab control bar to the tree of elements of the view, the view itself contains a bar with 3 buttons (`start`, `stop` and `connect/disconnect`). Their names are very explicit, and do not need much explanation. One cannot send or receive data unless the experiment has been started. One can finish the data exchange when the experiment is stopped. Finally, the `connect/disconnect` button is included in the lab control bar for convenience to allow connection and disconnection directly from the lab interface.



(**a**) Creating an instance of `LabControlModelElement`

(**b**) Adding the widgets to the view

(**c**) Adding the code to synchronize variables

(**d**) Student GUI of the lab developed with EJsS

Figure 11. How to set up the Lab from EJsS.

There is still one important thing to do to keep the web application synchronized with the server: reading and writing the values of the server objects. To that end, the `LabControlModelElement` provides the following two methods: `lab.get(vars, callback)` and `lab.set(vars, values, callback)`. Generally, the best place to invoke these methods is within an evolution page that runs periodically, as Figure 11c presents.

At this stage, and once everything has been set up, the laboratory developer uses EJsS *play* button to run the application, or EJsS *package* button to generate the client application (whose complete front-end appears in Figure 11d) that can be distributed to the students or hosted in a web server.

Finally, after performing all the previous steps, the student database should be updated (if it has not been done manually during the adaptation of the lab server), taking advantage of the Users plugin for that tasks. To do it from EJsS, we only need, as Figure 7 shows, to provide the lab address and user login, retrieve the database from the lab, modify the student information, and send it back to the lab server.

7. Conclusions

EJsS is a popular open-source tool to develop computational interactive simulations for science and engineering. Its versatility has been exploited in other fields, after extending its functionality.

This works presents a standardized way to enhance EJsS functionality and GUI to new users based on *Plugins*. The main contribution of the new extension mechanism is that it

lets software developers customize EJsS to their workflow or to the workflow of other users that exploit EJsS to develop applications (including simulations), in the way that better fulfills their needs. Through several examples we show the versatility of our proposal and encourage all types of EJsS users to analyze how they use EJsS and to determine what could be improved and adapted in their processes.

At this point, we want to highlight the differences in methodology between the standard approach to enhance EJsS and the one proposed in the paper. The traditional *Elements* extension mechanism has a particular and limited scope, while our *Plugins* can also adapt EJsS GUI to specific applications and modify the building environment to expose new functionalities better.

Finally, as two of the illustrative presented *Plugins* are directly related with the management of remote laboratories, to let their prospective users to exploit the possibility of setting up part of the lab from EJsS, we also provide the lab back-end, a streamlined remote laboratory server that can be easily adapted to different types of experiments.

In the future, and if the plugin architecture finds acceptance between EJsS users, the next step would be to reduce the vanilla EJsS version to a minimal core that provides the base functionality (ODEs, Model/View, ...) and the extensibility mechanism, and move the non-essential EJsS components to *Plugins* that focus on specific applications, such as process control, robotics, remote laboratories, or whatever a user would find useful. Moreover, we would be happy to provide support during the development of new *Plugins*, to help users with limited programming capabilities customize EJsS to their needs.

Author Contributions: Conceptualization and methodology, J.C., E.B.-P. and J.A.L.-O.; software, J.C.; validation, G.C.-B.; writing—original draft preparation, J.C. and E.B.-P.; writing—review and editing, G.C.-B. and J.A.L.-O.; funding acquisition, E.B.-P. and J.A.L.-O. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the University Complutense de Madrid through its 2019-2020 program of Innovation Educational and Teaching Improvement Projects and its POCOSIN Specific Research Grant.

Acknowledgments: The authors want to acknowledge Professor Francisco Esquembre for his suggestions during the development of this work. They also want to mention Mr Iñigo Aizpuru Rueda for his initial software developments.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rutten, N.; van Joolingen, W.R.; van der Veen, J.T. The learning effects of computer simulations in science education. *Comput. Educ.* **2012**, *58*, 136–153. [CrossRef]
2. de Jong, T.; Linn, M.C.; Zacharia, Z.C. Physical and Virtual Laboratories in Science and Engineering Education. *Science* **2013**, *340*, 305–308. [CrossRef]
3. Ma, J.; Nickerson, J. Hands-On, Simulated and Remote Laboratories: A Comparative Literature Review. *ACM Comput. Surv.* **2006**, *38*. [CrossRef]
4. Chen, X.; Song, G.; Zhang, Y. Virtual and Remote Laboratory Development: A Review. In Proceedings of the 12th International Conference on Engineering, Science, Construction, and Operations in Challenging Environments 2010, Honolulu, HI, USA, 14–17 March 2010.
5. Potkonjak, V.; Gardner, M.; Callaghan, V.; Mattila, P.; Guetl, C.; Petroviä, V.M.; Jovanoviä, K. Virtual laboratories for education in science, technology, and engineering: A review. *Comput. Educ.* **2016**, *95*, 309–327. [CrossRef]
6. PhET Interactive Simulations for Science and Math Webpage. Available online: https://phet.colorado.edu/ (accessed on 10 January 2021).
7. Open Source Physics Webpage. Available online: http://www.compadre.org/osp (accessed on 10 January 2021).
8. UNILABs Webpage. Available online: https://unilabs.dia.uned.es (accessed on 10 January 2021).
9. Martin-Villalba, C.; Urquia, A.; Senichenkov, Y.; Kolesov, Y. Two approaches to facilitate virtual lab implementation. *Comput. Sci. Eng.* **2014**, *16*, 78–86. [CrossRef]
10. Christian, W.; Belloni, M.; Esquembre, F.; Mason, B.A.; Barbato, L.; Riggsbee, M. The Physlet Approach to Simulation Design. *Phys. Teach.* **2015**, *53*, 419. [CrossRef]
11. Farah, S.; Benachenhou, A.; Neveux, G.; Barataud, D.; Andrieu, G.; Fredon, T. Flexible and real-time remote laboratory architecture based on Node.js server. In Proceedings of the 3rd Experiment International Conference, Ponta Delgada, Portugal, 2–4 June 2015.

12. Burke, R.D.; De Jonge, N.; Avola, C.; Forte, B. A virtual engine laboratory for teaching powertrain engineering. *Comput. Appl. Eng. Educ.* **2017**, *25*. [CrossRef]
13. Cambronero-Lopez, F.; Gomez-Varela, A.; Bao-Varela, C. Designing an ultrafast laser virtual laboratory using MATLAB GUIDE. *Eur. J. Phys.* **2017**, *38*, 034006. [CrossRef]
14. Gonzalez, J.D.; Escobar, J.H.; Sanchez, H.; la Hoz, J.D.; Beltran, J.R. 2D and 3D virtual interactive laboratories of physics on Unity platform. *J. Phys. Conf. Ser.* **2017**, *935*, 012069. [CrossRef]
15. Marcos, M.P.; de Prada, C.; Pitarch, J.L. Desarrollo e implementacion de un sistema de control en una planta piloto hibrida. In *XXXVIII Jornadas de Automática*; Servicio de Publicaciones de la Universidad de Oviedo, Campus de Humanidades: Oviedo, Asturias, Spain, 2017.
16. Zapata Rivera, L.F.; Larrondo-Petrie, M.M.; Ribeiro da Silva, L. Implementation of cloud-based smart adaptive remote laboratories for education. In Proceedings of the IEEE Frontiers in Education Conference, Indianapolis, IN, USA, 18–21 October 2017.
17. Zhu, Q.; Wang, T.; Jia, Y. Second Life: A New Platform for Education. In Proceedings of the First IEEE International Symposium on Information Technologies and Applications in Education, Kunming, China, 23–25 November 2017; pp. 201–204.
18. EJsS Webpage. Available online: http://fem.um.es/Ejs (accessed on 10 January 2021).
19. Esquembre, F. Facilitating the Creation of Virtual and Remote Laboratories for Science and Engineering Education. *IFAC Workshop Internet Based Control. Educ.* **2015**, *48*, 49–58.
20. ComPADRE Resources for Services for Physics Education. Available online: https://www.compadre.org (accessed on 10 January 2021).
21. Jara, C.A.; Candelas-Herías, F.A.; Torres, F. RobUaLab.ejs: A New Tool for Robotics e-Learning. Available online: http://hdl.handle.net/10045/10157 (accessed on 10 January 2021).
22. Casals-Torrens, P. Virtual Laboratory for Learning Asynchronous Motors in Engineering Degrees. *IEEE Rev. Iberoam. Tecnol. Aprendiz.* **2013**, *8*, 71–76. [CrossRef]
23. Chaos, D.; Chacon, J.; Lopez-Orozco, J.A.; Dormido, S. Virtual and Remote Robotic Laboratory Using EJS, MATLAB and LabVIEW. *Sensors* **2013**, *13*, 2595–2612. [CrossRef] [PubMed]
24. Bermudez-Ortega, J.; Besada-Portas, E.; Lopez-Orozco, J.A.; Bonache-Seco, J.A.; de la Cruz, J.M. Remote Web-based Control Laboratory for Mobile Devices based on EJsS, Raspberry Pi and Node.js. In Proceedings of the IFAC Workshop on Internet Based Control Education, Brescia, Italy, 4–6 November 2015.
25. Saenz, J.; Esquembre, F.; Garcia, F.J.; de la Torre, L.; Dormido, S. An Architecture to use Easy Java-Javascript Simulations in New Devices. In Proceedings of the IFAC Workshop on Internet Based Control Education, Brescia, Italy, 4–6 November 2015.
26. Bermudez-Ortega, J.; Besada-Portas, E.; Lopez-Orozco, J.A.; Chacon, J.; de la Cruz, J.M. Developing web TwinCAT PLC-based remote Control laboratories for modern web-browsers or mobile devices. In Proceedings of the 2016 IEEE Conference on Control Applications, Buenos Aires, Argentina, 19–22 September 2016; pp. 810–815.
27. Besada-Portas, E.; Bermudez-Ortega, J.; de la Torre, L.; Lopez-Orozco, J.; de la Cruz, J. Lightweight Node.js & EJsS-based Web Server for Remote Control Laboratories. In Proceedings of the 11th IFAC Symposium on Advances in Control Education, Bratislava, Slovakia, 1–3 June 2016.
28. de la Torre, L.; Sánchez, J.; Andrade, T.F.; Restivo, M.T. Easy creation and deployment of Javascript remote labs with EjsS and Moodle. In Proceedings of the 13th International Conference on Remote Engineering and Virtual Instrumentation, Madrid, Spain, 24–26 February 2016.
29. de la Torre, L.; Andrade, T.F.; Pedro Sousa, J.S.; Restivo, M.T. Assisted Creation and Deployment of Javascript Remote Experiments. *Int. J. Online Biomed. Eng.* **2016**, *12*. [CrossRef]
30. Mejías, A.; Herrera, R.S.; Márquez, M.A.; Calderón, A.J.; González, I.; Andújar, J.M. Easy Handling of Sensors and Actuators over TCP/IP Networks by Open Source Hardware/Software. *Sensors* **2017**, *17*, 94. [CrossRef]
31. Galan, D.; Isaksson, O.; Rostedt, M.; Enger, J.; Hanstorp, D.; de la Torre, L. A remote laboratory for optical levitation of charged droplets. *Eur. J. Phys.* **2018**, *39*, 045301. [CrossRef]
32. Reyes, M.; Sanchez-Herrera, S.; Mejias, A.; Marquez, M.; Andujar, J. A fully integrated open solution for the remote operation of pilot plants. *IEEE Trans. Ind. Inform.* **2018**, *15*, 3943–3951.
33. Torres, A.; Jara, C.; Seguí, B.; García, G.; Pomares, J. Development of hybrid laboratories of industrial systems for interactive learning of automation and control. In Proceedings of the 11th International Conference on Education and New Learning Technologies, Palma de Mallorca, Spain, 1–3 July 2019; pp. 8442–8446.
34. Esquembre, F.; García Clemente, F.J.; Chicón, R.; Wee, L.; Kwang, L.; Tan, D. Easy Java/JavaScript Simulations as a tool for Learning Analytics. In Proceedings of the International Conference on Computers in Education, Kenting, Taiwan, 2–6 December 2019.
35. UCM-DACYA-Labs. EJsS with Plugin Support. Available online: https://github.com/jcsombria/ejss (accessed on 10 January 2021).
36. UCM-DACYA-Labs. EJsS Plugins. Available online: https://github.com/jcsombria/ejss-plugins (accessed on 10 January 2021).
37. Esquembre, F. Easy Java Simulations: A software tool to create scientific simulations in Java. *Comput. Phys. Commun.* **2004**, *156*, 199–204. [CrossRef]
38. de la Torre, L.; Chacon, J.; Chaos, D.; Dormido, S.; Sánchez, J. A Master Course on Automatic Control with Remote Labs. In Proceedings of the 12th IFAC Symposium on Advances in Control Education, Philadelphia, PA, USA, 7–9 July 2019; pp. 56–61.

39. Heradio, R.; Torre, L.D.L.; Galan, D.; Cabrerizo, F.; Herrera-Viedma, E.; Dormido, S. Virtual and remote labs in education: A bibliometric analysis. *Comput. Educ.* **2016**, *98*, 14–38. [CrossRef]
40. Heradio, R.; de la Torre, L.; Dormido, S. Virtual and remote labs in control education: A survey. *Annu. Rev. Control* **2016**, *42*, 1–10. [CrossRef]
41. EjsS Extensions. Available online: https://github.com/UNEDLabs (accessed on 10 January 2021).
42. Python Webpage. Available online: https://www.python.org/ (accessed on 10 January 2021).
43. Flask Documentation Webpage. Available online: https://flask.palletsprojects.com/en/1.1.x/ (accessed on 10 January 2021).
44. REST API Tutorial Webpage. Available online: https://restfulapi.net/ (accessed on 10 January 2021).
45. Parkhomenko, A.; Gladkova, O.; Sokolyanskii, A.; Shepelenko, V.; Zalyubovskiy, Y. Implementation of reusable solutions for remote laboratory development. *Int. J. Online Eng.* **2016**, *12*, 24. [CrossRef]
46. de la Torre, L.; Chacon, J.; Chaos, D. Remote Interoperability Protocol Specification. Available online: https://doi.org/10.5281/zenodo.2644242 (accessed on 10 January 2021).
47. Cornetta, G.; Mateos, J.; Touhafi, A.; Muntean, G.M. Design, simulation and testing of a cloud platform for sharing digital fabrication resources for education. *J. Cloud Comput.* **2019**, *8*, 12. [CrossRef]