Enabling the Automatic Generation of User Interfaces for Remote Laboratories

Wissam Halimi[1], Christophe Salzmann[2], Hagop Jamkojian[1], and Denis Gillet[1]

[1] EPFL, REACT, Station 11, CH-1015 Lausanne, Switzerland
wissam.halimi@epfl.ch, hagop.jamkojian@epfl.ch, denis.gillet@epfl.ch
[2] EPFL, Automatic Control Laboratory, Station 9, CH-1015 Lausanne, Switzerland
christophe.salzmann@epfl.ch

Abstract. Remote laboratories are an important component of blended and distance science and engineering education. By definition, they provide access to a physical lab in a distant location. Many architectures enabling remote laboratory systems exist, the most common of which are Client-Server based. In this context, the Server interfaces the physical setup and makes it software-accessible. The Smart Device Specifications revisit a Client-Server architecture, with the main aim of cancelling the dependencies which inherently exist between a Client and a Server. This is done by describing the Server as a set of services, which are exposed as well-defined APIs. If a remote laboratory is built following the Smart Device Specifications, any person with programming skills can create a personalized client application to access the lab. But in practice, teachers rely on the mediated contact with a lab provider to have information about what kind of experiment(s) the lab in question implements. Even though there is a complete description of the available sensors and actuators making up a lab and how to be accessed, it is not clear how they are connected (relationships). In this sense, a list of sensors and actuators are not enough to make a guided selection of components to create the interface to an experiment. Therefore, the aim of this work is to support teachers in choosing the experiments and creating the respective UI on their own, in a pedagogically oriented scenario and by taking into consideration the target online learning environment. This is done by revisiting the Smart Device Specifications and extending them, in addition to proposing a tool that will automatically generate the user interface of the chosen experiment(s).

Keywords: remote laboratories, online learning, cyber physical systems, user interfaces, personalisation

1 Introduction

Remote laboratories (RLs) are an important component of distance and blended learning for science and engineering education. They allow learners to experiment in order to validate or refute a hypothesis, accept or reject a taught subject. By definition, they provide remote access to hands-on sessions, which are essential for the process of learning and assimilating scientific concepts [**?**][**?**]. As some Web technologies emerged and

died, many architectures for remote laboratory systems have been proposed. These architectures range from being case-specific to more generalised. The most adopted architectures, are Client-Server based where typically the Server interfaces the physical equipment of a lab, and the Client provides a software application through which users can access the lab. With the birth of the 'separation of concerns' paradigm enabling Service Oriented Architectures (SOA), lab providers started building their laboratories in a more modular way [**?**][**?**]. With such architectures, the access to the remote setup is done through web services or APIs where the laboratory server is exposed as services [**?**][**?**][**?**]. The main aim of adopting a Service Oriented Architecture for RLs is to separate the tiers of the remote laboratory system. The Smart Device Specifications for remote labs describe the Server as services through well-defined interfaces as proposed in [**?**]. This approach is motivated by the complete separation of the Client from the Server encourages the broader sharing of remote labs. The Smart Device Paradigm decouples the Client and the Server further enabling the personalization of the client applications. When the Smart Device Specifications are adopted for an RL, the Server is exposed as services described by APIs enabling any person with programming skills to create user applications to connect to the labs. This is possible by "talking" to the APIs and understanding how client applications can access the services provided by a remote setup. With the Smart Device Specifications, the development and deployment of remote laboratories is much easier, faster, and modular for different stakeholders, namely the lab provider and the user client developer.

In this context, invoking a service is equivalent to controlling actuators or retrieving data from sensors making up the laboratory setup. Provided the APIs, it is possible to personalize the client application accessing the labs by enabling the teachers to use the RLs in different ways, according to their educational needs, by designing their own experiments. This is the case of remote laboratories that are configurable and offer the flexibility of conducting different experiments, corresponding to different scientific phenomena. In this work, we refer to the activity which allows students to freely vary the parameters on lab equipment as an experiment, and we refer to the combination of sensors and actuators used in an experiment as a "configuration" from a lab owner point of view. Since the information provided by the APIs based on the Smart Device Specifications do not covey enough information that shows how the sensors and actuators are connected and dependant, usually the teachers resort to the mediated contact with a lab provider to have information about what kind of experiment(s) the lab in question implements. On another note, creating User Interfaces (UIs) is still largely reliant on disposing of a software developer, preventing teachers with no such privileges from personalizing their own applications. We recognise that it is important in such setups to give teachers the autonomy to select and create user interfaces for remote labs that fulfil their own pedagogical objectives, without the need to contact a lab provider through an application developer. Therefore, the aim of this work is to support teachers in choosing the experiments and creating the respective UI on their own, in a pedagogically oriented scenario and by taking into consideration the target online learning environment. This is done by revisiting the Smart Device Specifications and extending them, in addition to proposing a tool that will automatically generate the user interface of the chosen experiment(s).

This paper is structured as follows: we begin by providing on overview of existing remote labs architectures while identifying their pros and cons for the challenges at hand. In Section **??** we elaborate on our extension of the Smart Device Specifications to enable the automatic generation of UIs for configurable labs. In Section **??** we present our proposed tool for UI generation, and an accompanying example for a remote laboratory: the Mach-Zehnder Interferometer.

2 Related Work

In [**?**] the authors make their debut in defining Smart Devices (SDs) motivated by the need to move away from adopting proprietary technologies for building remote laboratories, and the need to converge towards common conventions for designing and building remote laboratory systems. Accordingly, they re-engineer the server side by implementing separate services for the different hardware access which are possible for their example lab. In parallel, instead of creating a complete web application or widget, they provide four separate ones for each of the accessible services: a graph tool, a video feed, a control panel for the system's parameters, and a tool for saving the experimental data. The users of a remote lab can choose any subset or all the provided widgets to use the lab in a 'metawidget'. While this effort is a move toward a personalization of the user client, it is still proprietary for the embedding web-based environment.

Later in [**?**] and [**?**], the authors elaborate in more detail about the Smart Device Paradigm and introduce the concept of LaaS (Lab as a Service). The Smart Device Paradigm revisits the reputed Client-Server architecture for remote labs by re-thinking the server side and equipping its component with some 'intelligence'. This is based on Thomson's definition of smart devices, as devices which have identity and kind, memory and status tracking, communication capabilities, and more [**?**]. Accordingly, the Smart Device Specifications extend this definition to support complex systems such as remote labs. Motivated by the need to completely separate the server and client sides to further enable the personalization of client applications, the mentioned specifications represent the behaviour of the connected sensors and actuators as services exposed through well-defined APIs. The services representing a sensor or actuator instance are fully described through 'metadata'. The 'metadata' provide a description of the considered lab through the General Metadata which tells the name of the lab, a short high-level description, a contact person, and licensing information. The API Metadata defines the supported services by the lab, by specifying the corresponding sensor and actuator requests and responses. Moreover, the Smart Device Specifications provide service descriptions for authorisation, which takes care of user authentication. This metadata category is of no interest to this work. It is claimed that the 'metadata' is enough for building user applications without the need for further information from a lab provider. While this might be true for laboratories supporting one experiment, it is not true for configurable labs which provide the possibility of conducting many experiments with the same connected equipment. This is due to the absence of a description of how the sensors and actuators are connected and which configurations are possible. The Smart Device specifications provide a description of services as independent units.

Other frameworks for the generation of remote lab user interfaces exist, such as the tool based on EjsS in [**?**]. In this work, the authors bring to importance the need for user interfaces which can be well integrated in web-based learning environments such as Moodle. Additionally they invoke the necessity to support open web technologies and move away from Java applets which are no longer supported by modern web browsers. While they provide a solution that is reusable, and prevents application developers from building UIs from scratch for each lab, this framework only supports the generation of UIs for labs which are compatible with their implementation of the presented app builder.

3 Revisited Smart Device Specifications

The API Metadata of the Smart Device Specifications specify the communication protocol and formats for sending requests and receiving responses from a remote laboratory. It is composed of two main sections: *apis* and *models*. The *apis* describe which services are implemented and how they can be accessed, by providing information on the adopted communication protocol, the type of requests to write and responses to receive specified by their corresponding *models*, the parameters to pass to the request, and the authorization schema to implement at the client side if applicable. The *models* section details the structure of the requests, responses, and data to be applied to the actuators or sensed by the sensors. It includes information on the unit, type, allowed ranges, range steps, last measured values, and the value update frequency.

The *apis* section is based on four main API calls: *getSensorMetadata*, *getSensorData*, *getActuatorMetadata*, and *setActuatorData*. *getSensorMetadata* is formatted as a *SensorMetadataRequest* model, returns a list of all sensors in the lab in a response formatted as a *SensorMetadataResponse* model. In the response to this request, the *sensorIds* are included to allow for separate calls to each. To read the data on a specific sensor, the UI calls the *getSensorData* request as modeled by a *SensorDataRequest* by including the corresponding *sensorId*, as a response the data captured by the sensor is returned in a *SensorDataResponse*. A *getActuatorMetadata* request sent as an *ActuatorMetadataRequest* returns a list of all actuators in the lab: the *actuatorIds* in an *ActuatorMetadataResponse*. The *actuatorIds* is an array which contains the *actuatorId* of separate actuators. To write data to an actuator, it is sufficient to invoke the *sendActuatorData* request formatted as a *SetActuatorData* request, providing an *actuatorId*.

With such information a basic automatic UI generator can be put in place. The API calls in addition to the requests and responses models are provided. It is worth mentioning that all exchanged requests and responses with the Smart Device are JSON-encoded, further facilitating the parsing of the API calls, which can be automatized. But what sensors to pick with which actuators? How are the sensors and actuators working together? Are there several possible experiments that can be conducted with the same setup, but using different sensor-actuator configuration? All of this information is not included in the existing SD Specifications. What differentiates remote laboratories from other cyber-physical systems is that they are built to fulfil an educational goal: conducting pre-defined experiments to reflect on certain topics. With no knowledge about the interconnections of the lab components, it is not possible to build a UI that interfaces

'pedagogically meaningful' experiments. In this section, we extend the Smart Device Specifications to describe the possible "configurations" or "experiments" of labs supporting one or various experiments, further enabling the auto generation of the user interface. We extend the detailed 'metadata' to add a service to the *apis* which returns the configurations or experiments supported by the remote lab, in addition to the requests and responses *models*. The extended SD Specifications with the *experiments* service provides enough information to enable the automatic generation of user interfaces without the need of the lab owner to confirm the possibility of conducting a particular experiment. Our proposed extension is two-fold:

1. Define the *models* for an *Experiment*, *SendExperimentsRequest*, *SendExperimentRequest*, *ExperiementsMetadataResponse*, and *ExperiementMetadataResponse*
2. Define the new api calls: *getExperiments* and *getExperiment*

Experiment model: An *Experiment* model is characterised by 2 fields common to all models: *id* and *properties*. The *id* characterises the model at hand, in this case its value is *Experiment*. This *id* field gives knowledge to the automatic generator about the format of an *Experiment* JSON object for further processing. The *properties* are made up of 5 sub-fields:

- experimentId: which can take any string value. The value of this field is defined by the lab provider.
- fullName: which contains a non-formal name of the experiment. It can take any string value.
- description: a human readable description of what the experiment is about. This field is meant to be informative for teachers, to get a high level description of the experiment.
- sensors: it is an array containing a list of the sensor ids used in a particular experiment. *sensorIds* can have any string value. The string values of *sensorIds* contained in this JSON object should be corresponding *sensorIds* defined in the metadata.
- actuators: it is an array containing a list of the actuator ids used in a particular experiment. *actuatorIds* can have any string value. The string values of *actuatorIds* contained in this JSON object should be corresponding *actuatorIds* defined in the metadata

A complete Experiment model is shown below:

```
"Experiment": {
    "id": "Experiment",
    "properties": {
    "experimentId": {
        "type": "string"
        },
    "fullName": {
        "type": "string"
        },
    "description": {
        "type": "string"
```

```
    },
    "sensors": {
        "type": "array",
        "items": {
          "id": "Sensor",
          "properties": {
            "sensorId": {
            "type": "string"
            }}}
    },
    "actuators": {
        "type": "array",
        "items": {
          "id": "Actuator",
          "properties": {
            "actuatorId": {
            "type": "string"
            }}}
    }
}
```

getExperiments api: The *getExperiments* api allows the retrieval of a list of supported experiments. The *nickname* of this call is "getExperiments" which means it needs to be used when initiating a request. *summary* and *notes* fields give a high level description of what this call does: answers with a JSON object containing the list of available experiments ids. The response of this call is formatted as an *ExperimentMetadataResponse* which will be detailed later in this section. As it can be deducted from the *properties* field, the request is formatted as a *SimpleRequest* defined in the original SD Specifications. The *authorization* field designates authentication mechanisms that the remote lab is using to permit users to access the lab, if empty it means no authorization needs to be done. *responseMessages* detail the possible responses that can be received at the requester end, in case an *ExperimentMetadataResponse* cannot be received.

```
{"method": "Send",
"nickname": "getExperiments",
"summary": "Returns a list of possible experiments",
"notes": "Returns a JSON array with all the ids of possible experiments",
"type": "ExperimentMetadataResponse",
"parameters": [{
        "name": "message",
        "description": "The payload for the getExperiments service.",
        "required": true,
        "paramType": "message",
        "type": "SimpleRequest",
        "allowMultiple": false
    }],
"authorizations": {},
"responseMessages": [{
        "code": 402,
```

```
            "message": "Too many users"},{
            "code": 404,
            "message": "Experiments not found"},{
            "code": 405,
            "message": "Method not allowed. The requested method is not allowed
            by this server."},{
            "code": 422,
            "message": "The request body is unprocessable"
        }]}
```

ExperimentRequest model: To retrieve the required *actuatorIds* and *sensorIds* for a particular experiment, an *ExperimentRequest* has to be sent to the Smart Device hosting the laboratory as shown hereafter. The *ExperimentRequest* should contain the *experimentId* of the desired experiment. A list of *experimentsId*s can be retrieved with the *getExperiments* call.

```
"ExperimentRequest": {
    "id": "ExperimentRequest",
    "required": ["method", "experimentId"],
    "properties": {
        "method": {
            "type": "string",
            "description": "The method should be equal to the nickname of one
            of the provided services."},
        "experimentId": {
            "type": "string"}
}
```

ExperiementMetadataResponse model: The response of an *ExperiemntRequest* is an *ExperimentMetadataResponse*. The *id* of this response tells the type of JSON object to expect at the receiving end. It is formatted as to contain the *Experiment* JSON object which defines an experiment. This should be enough for an auto generator to create a UI corresponding to the required request.

```
        "ExperimentMetadataResponse": {
            "id": "ExperimentMetadataResponse",
            "properties": {
                "method": {
                    "type": "string"},
                "experiments": {
                    "type": "array",
                    "items": {
                        "$ref": "Experiment"
        }}}}
```

In Sections **??** and **??** we present our automatic generator of user interfaces based on the extended Smart Device Specifications, to demonstrate their completeness for our purpose. In addition to providing the teachers with a tool that enables them to autonomously create basic UIs for remote labs, this automatic UI generator provides the

possibility of personalizing the UI for embedding in a platform of choice: the social media platform graasp[3], or an LTI consumer platform such as Moodle or edX[4]. The proposed approach is illustrated through a remote laboratory supporting multiple experiments: the remote Mach-Zehnder interferometer. This remote laboratory was presented as a work in progress in [**?**], where we build the lab using software templates according to the Smart Device Specifications, and in this paper we extend the implementation to support the personalised auto-generation of the user interface with the added configurations.

4 An Example: Light Interference Experiences

4.1 The Mach-Zehnder Interferometer

As mentioned earlier, we are especially interested in reconfigurable experiments. The Mach-Zehnder Interferometer is an example of devices which are used to study different subjects ranging from light interference to optical telecommunication, in classical and also in quantum physics [**?**] [**?**]. The Mach-Zehnder interferometer considered in this paper has a layout shown in Fig.**??**. The apparatus is composed of a monochromatic light beam, two half-mirrors, two complete mirrors, two beam splitters, a density filter, and a detection screen mounted with a photo diode. With this layout, some light interference characteristics and resulting phenomena can be studied by repeatedly reflecting the light beam on the mirrors and half-mirrors before its arrival on the detection screen. In the next subsection we describe two of the possible experiments that can be conducted with this equipment. For visualization purposes, two cameras are placed in the lab in order to reflect the status of real environment: a camera placed in front of the detector screen to see the resulting incident light, and an infrared camera (because the experiments are conducted in the dark) that shows the whole setup.

4.2 The Experiments

The mind map in Fig.**??** shows two possible experiments that can be done with the Mach-Zehnder interferometer, upon which we will base our explanation of the implementation and function of the automatic UI generator in Section **??**.

The first and second experiments are conducted in a high light intensity setup, meaning that the density filter is not attenuating the intensity of the light coming from the monochromatic light beam. The first experiment enables the users to quantitatively understand light interference, by visualizing the resulting fringes on the screen, and/or also the feed from the infrared camera, in addition to depicting the direction in which the fringes move when the mirror mounted with a piezo actuator manually controlled with a voltage which is increasing or decreasing in value. In the second experiment, the students can quantitatively study light interference by observing the emitted signal from the photo diode as the piezo is controlled with a triangular signal causing a translation motion.

[3] http://graasp.eu/
[4] https://www.edx.org/

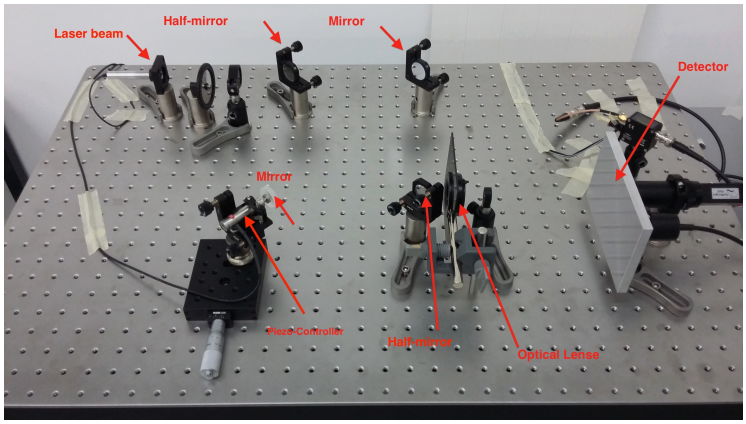

Fig. 1. The Mach-Zehnder Interferometer Layout

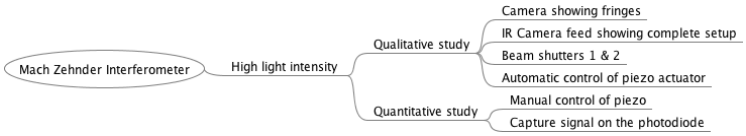

Fig. 2. Mindmap of the Mach-Zehnder Experiments

When a *getExperiments* api call is sent to the Smart Device hosting this laboratory, the following response is received:

```
{"method": "getExperiments",
"experiments": [{
    "experimentId": "qualitative",
    "fullName": "Qualitative Study",
    "description": "Observing light interference on the screen",
    "sensors": [{"sensorId": "Video"}, {"sensorId": "VideoIR"}],
    "actuators": [{"actuatorId": "laser"},{"actuatorId": "piezo"},
    {"actuatorId": "bs1"},{"actuatorId": "bs2"}]
    },{
    "experimentId": "quantitative",
    "fullName": "Quantitative Study",
    "description": "Studying the  signal provided by the photodiode",
    "sensors": [{"sensorId": "photodiode"}],
    "actuators": [{"actuatorId": "laser"},{"actuatorId": "piezo"},
    {"actuatorId": "bs1"},{"actuatorId": "bs2"}]
}]}
```

The response shows that there are two possible experiments with the *experimentId*s “qualitative” and “quantitative”. Accordingly, the list of sensors and actuators for each of the experiments can be either used from this response, or retrieved by a separate call to *getExperiment* while passing the corresponding *experimentId*.

5 The Automatic UI Generator

5.1 Design Considerations

In most cases, a remote laboratory is part of a learning activity comprising other educational resources such as documents, videos, etc... The learning activity is usually hosted by a MOOC platform such as edX, or social media platform such as graasp. When a remote lab is used in such contexts, it is important to take into consideration the following points to insure the integration of the RL user client in the platform on several levels: knowledge about user identity (awareness), the context, and having access to the storage resources of the platform.

Awareness about user identity is necessary for several purposes: authentication with the RL when required, saving and retrieving the data, and capturing user interaction with the RL user application. It is necessary to associate this data to the platform users for personalization purposes. Additionally, when conducting an experiment a lot of data is generated. Usually, when students are doing their experiments in physical labs, they save the data in files to be used for processing or take note of certain parameters. In all cases, these assets are saved for later reference or post processing. It is primordial to provide such facilities to the students, where keeping and retrieving their data can be accomplished within the platform. We take into account these considerations when implementing the automatic UI generator as detailed next.

5.2 Implementation

The automatic UI generator is a tool that enables the creation of a fully functional remote lab web client with a few clicks. The teacher needs to know the IP address and the port number over which a Smart Device is serving the desired remote lab. Using this information, the tool initiates a WebSocket connection with the lab server, and subsequently call the *getExperiments* service, which returns an array describing each experimental configuration supported by the Smart Device. As mentioned in Section **??**, each experiment is described by: the *experimentId* that uniquely identifies each experiment, the *fullname* and *description* of the experiment, in addition to the sensors and actuators arrays that contain the ids of all the respective sensors and actuators used by each experimental configuration. These configurations are displayed as checkboxes having the full name and the description of the experiment as their labels. The teachers can then select one or more of the presented possible configurations according to their educational goals. After performing this selection, the auto generator knows the ids of all the different sensors and actuators required for each experiment, and will thus send *getAcutatorMedata* and *getSensorMetadata* requests to the lab server in order to acquire the necessary information about each (See Fig.**??**).

For actuator access, the auto generator makes use of some of the fields obtained from the actuator metadata, in order to generate the necessary UI components. It uses the *actuatorId* which uniquely identifies each actuator, to populate the *actuatorId* field of the request packet which is sent to the Smart Device whenever a user of the generated lab client alters the state of an actuator, thus making a call to the *sendActuatorData* service. The auto generator also uses the *values* field of the metadata, which is an array

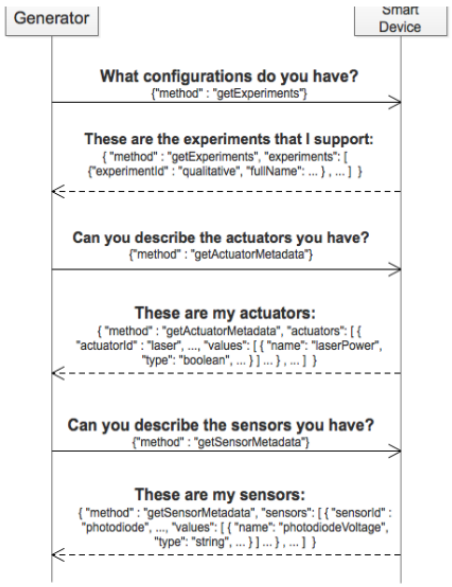

Fig. 3. How the automatic UI generator interacts with the Smart Device to build the UI

of all the measurement values each actuator contains. Each actuator value will be represented as a separate UI component in the generated widget. The auto generator uses the following fields from the metadata of each value:

- name: used to differentiate among the multiple values of an actuator.
- type: used to decide what type of UI component needs to be created for each value. For instance, a value of type boolean will be represented as a button that can be turned on or off by the user. Moreover, a value of type float will be represented as a numeric slider.
- rangeMinimum and rangeMaximum: used by the auto generator to specify the boundaries of the numeric slider that is created for a value of type float.

For sensor requests, the UI generator uses the *sensorId*, which uniquely identifies each sensor, to populate the *sensorId* field of the request packet that is sent to the Smart Device whenever the lab client makes a call to the *getSensorData* service. The generator also takes into consideration the *webSocketType* field of the sensor metadata to check whether a given sensor requires a text or a binary WebSocket. In case of a binary WebSocket, the generator assumes that it is a video feed and creates a UI component that displays the video. In the case of a text WebSocket, the generator uses the *values* field of the sensor metadata and represents each value as a separate UI component in the generated gadget. The auto generator uses the following fields from the metadata of each value:

- name: used to differentiate among the multiple values of a sensor.

– type: used to decide what type of UI component should to be created for each sensor value. For instance, a value of type boolean will be represented as a LED indicator. Moreover, a value of type string will be represented as a text value.
– unit: used by the generator to append a unit symbol to the retrieved sensor value.

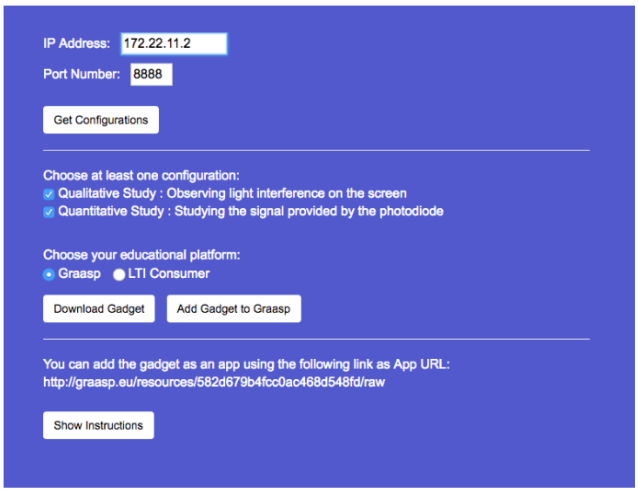

Fig. 4. The landing page of the automatic UI generator showing the two available experiment configurations for the Mach-Zehnder lab

Furthermore, the teacher has to choose an educational platform in which the generated UI will be embedded (See Fig. **??**). Currently, the automatic UI generator provides UIs which can be embedded in graasp, or in an LTI consumer platform (such as Moodle, or edX). If an LTI hosting platform is chosen as the target platform, then the resulting lab client application will automatically instantiate a WebSocket connection with the Smart Device whenever a user accesses the lab client, update the UI components of the sensors upon receiving new sensor values, handle the actuator changes performed by the user and send the new actuator data to the Smart Device. According to the teacher's selection of one or more experiments, the application will contain one or more tabs. Each tab represents a selected experimental configuration. Clicking on a tab in the client application will result in accessing the corresponding experiment, and displaying all the sensors and actuators associated with that setup. In this case, the resulting UI is an html file which can be used by the teacher to embed in the LTI consumer platform.

On the other hand, if graasp is chosen as the target educational platform, then the generated lab client application is an OpenSocial (OS) widget which can be embedded in the platform. Graasp supports OpenSocial widgets through its own implementation of the Shindig Apache server, enabling third party applications to access its database for user information, and for saving and retrieving files, actions, and other platform specific

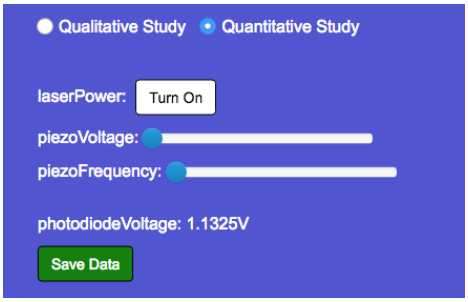

Fig. 5. The generated remote lab client application on graasp for the Mach-Zehnder lab

data [**?**]. Consequently, the OS widget will have all the aforementioned features of the LTI-targeted application, in addition to the following (See Fig. 5):

- Action logging: the generated graasp gadget will use the ActionLogger library[5], which provides an easy mechanism for logging the activities of the students. Interactions with the different UI components are saved as Activity Streams that have the actor-verb-object format. The logged activities can later be used to perform learning analytics.
- Saving experimental data: The lab application allows students to save the actuator and sensor data that were acquired while conducting the experiment. The data is saved in a specific format, that allows students to use it in other applications on the platform. For example, the students can have a graphical view of the experimental results using the Data Viewer application[6].

6 Teacher Customization

The automatic UI generator provides a basic and fully functional client application for operating a remote lab. The UI components are very basic, and might not be visually attractive. Using the generated code, the teachers can further personalize the UI appearance to fit their taste and needs. For example, a teacher in the Gymnase de Morges in Switzerland, chose to customize the UI to be embedded in graasp as shown in Fig.**??**.

In this widget, there are two tabs to switch between two possible experiments. In the Quantitative Study tab, there is a simulation diagram which allows students to control the lab by clicking on the corresponding image of a component. For example, to turn the laser beam ON/OFF it is enough to click on the box representing the light source. On the diagram are also present the placements of the IR camera and the normal camera allowing the student to know about the perspective of the video feeds. In this widget, the teacher chose to only display the video coming from Camera 2 showing the fringes on the screen. Next to it is a graphing tool that shows the signal captured by the photo diode in real-time. Since the teacher doesn't want the students to have to scroll, and

[5] https://github.com/go-lab/ils/wiki/ActionLogger
[6] http://go-lab.gw.utwente.nl/production/dataViewer/build/dataViewerTool.xml

since the simulation diagram conveys a real-time status of the lab, he decided that there are enough UI components for the students to conduct the experiment while having a good user experience.

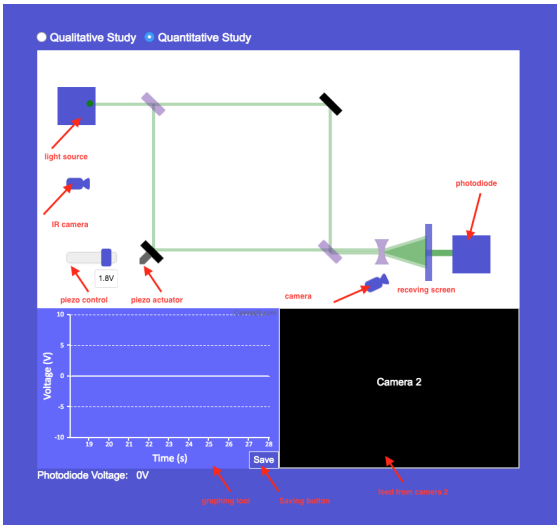

Fig. 6. Example of personalized Mach-Zehnder OS widget in graasp

Of course, the UI could have been customized otherwise to show the UI components differently, or to resize them in a different way. For example, an input box to control the piezo actuator could have been a replacement for the slider control. Also, instead of only showing the feed of Camera 2, both feeds from Camera 1 and Camera 2 could have been shown, in addition to the graphing tool. All of this is possible by starting from the code provided by the automatic UI generator. This alleviates the burden of establishing connections and parsing the remote lab APIs, making it more easy to personalize the appearance of user client according to a desired user experience.

7 Conclusion & Future Work

To describe a remote laboratory and enable the automatic generation of user interfaces, it is not enough to solely rely on describing the services making up laboratories. In this work we presented the extended Smart Device Specifications to support the description of the different configurations remote laboratories provide. This extension further enables the automatic generation of user interfaces. We also proposed a generator tool which helps teachers in autonomously creating client applications for different target platforms: graasp or an LTI-consumer environment. In our implementation of the tool, we take into consideration the need for full-integration with a target platform hosting the UI by providing an integration layer already embedded in the application supporting user identification, activity tracking, saving and retrieving experimental data.

To the best of our knowledge, this is the first automatic UI generator for remote lab clients, which is also integrating data storage for the target embedding learning environment. To ease the adoption of the Smart Device Specifications and to enable the targeting of other learning environments, the UI generator application is openly shared under the CC-BY-NC[7] creative commons licenses, on this link: `http://shindig2.epfl.ch/gadget/automatic_gadget_generator/`

Acknowledgment. This research is partially funded by the European Union in the context of Go-Lab (grant no. 317601) project under the ICT theme of the 7th Framework Programme for R&D (FP7).

[7] https://creativecommons.org/licenses/by-nc/2.0/