An Extendable Microservice Architecture for Remotely Coupled Online Laboratories

Johannes Nau[1] and Marcus Soll[2]

[1] Technische Universität Ilmenau
PF 10 05 65, 98684 Ilmenau, Germany
johannes.nau@tu-ilmenau.de
[2] NORDAKADEMIE gAG Hochschule der Wirtschaft
Köllner Chaussee 11, 25337 Elmshorn, Germany
marcus.soll@nordakademie.de

Abstract. The CrossLab project aims at creating a new online laboratory paradigm in which various parts of an experiment are modular and composable across institutional boundaries.
In this work, we show our progress on the development of the backend infrastructure that drives the distributed and remotely coupled online laboratory infrastructure developed in the CrossLab project. This infrastructure is a technical solution to enable the cross-elements and cross-universities aspects of the CrossLab project, i.e., it enables the composition of various laboratory devices across different universities into one experiment.
The backend infrastructure will be embedded in a larger system architecture which we describe briefly. The paper focuses on the details of the REST Interface that the system architecture uses.

Keywords: Virtual and Remote Labs, Hybrid Take-Home Laboratories, Software Development, Microservices

Source code: https://github.com/Cross-Lab-Project/crosslab

1 Introduction

In STEM education, hands-on training in laboratories is a crucial part of the curriculum because it allows the hands-on experience of the theoretical knowledge usually taught at universities [1]. Experience shows that these laboratories allow students to understand the knowledge taught in a course better [2].

Laboratory work has a few drawbacks regarding time, staff, and material needed compared to a classical lecture [3]. One method to partially overcome these drawbacks is to use online laboratories [4, 5], which are web-accessible versions of hands-on laboratories. The CrossLab project [6] aims at defining solutions at the technical, didactical, and organizational levels to create digital laboratory objects, that mix diverse types of laboratories (cross-types), enable the composition of different laboratory objects (cross-elements), mix various

disciplines (cross-disciplines), and are accessible by multiple universities (cross-universities). An exciting side benefit of the new paradigm developed in the CrossLab project is that it enables the development of hybrid take-home labs where actual hardware is located at the students' homes [7].

This paper presents the current state of the application programming interface (API) driving the CrossLab architecture. New laboratories can be created in this architecture by freely combining different laboratory devices through a uniform protocol. A laboratory device in this context is the abstract representation of inseparable elements of an experiment, i.e., a physical system that should be controlled, the user interface that interacts with or shows the experiment (including the user), or assessment systems that evaluate the user during the experiment [8].

Furthermore, this architecture includes all necessary services for running a remote laboratory while staying decentralized. The architecture closes a gap in the current protocol standards for remote laboratories [9].

2 State of the Art

This chapter briefly describes the current state of laboratory architectures and web architectures, which are needed to describe our proposed architecture.

2.1 Existing Architectures

There are several preexisting remote laboratory architectures. For example, the *GOLDi* lab [10] allows remote experiments, where experiments consist of a single control unit and a single electromechanical model. While the system can run at multiple locations and institutions, it tightly couples different components.

Another system widely used is the *LabsLand* system [11], which started as the open source *WebLab-Deusto* [12] and developed into a commercial service. LabsLand offers various remote and *ultraconcurrent*[3] experiments located at different institutions. However, it does not allow changing of configuration, i.e., connecting new devices to a preexisting experiment.

The *VISIR* project [13] aims to design electronics laboratory systems, including hardware and open-source software. Many universities have adopted VISIR worldwide [14]. While VISIR provides the actual experiment and the needed software, it does not provide any laboratory management and is often integrated into other Systems like WebLab-Deusto [15].

Many digital teaching projects often disappear or are not sustainable (for an overview of German projects, see [16]), which is also true for different remote laboratory systems: The LabShare [17] project has a dead website at the time of writing this paper, and the iLab project [18] closed in 2019 [19].

In addition, as shown in [9], there is currently no protocol for connecting multiple single devices to a unified experiment. This means the configuration of an experiment is always preconfigured or limited to only a few possibilities.

[3] prerecorded experiments with actual data which can be replayed as wished

2.2 Microservices

Microservices can be understood to be small (i.e., doing only one thing) and autonomous (i.e., communication solely through network) services that work together [20]. Unfortunately, multiple definitions are used in literature [21]. Interest in Microservices has risen over the last few years (cf. [21, 22]).

According to [22], microservices offer many benefits by dividing the application into smaller, self-contained units. These smaller units can be understood and tested more efficiently, and the technology can be chosen according to the task a unit should solve. Deployment and management are easier since the different microservices can run independently.

The same authors [22] pointed out potential problems from microservices. For example, it is hard to find the correct dimension, so microservices are both small and still large enough to be useful. In addition, the distributed nature of microservices leads to different challenges, some of which are: APIs must be stable or versioned, the attack surface is larger due to more exposed API, data consistency issues can surface, and performance measurement is more challenging.

One crucial aspect is the run-time environment of microservices. Besides running on physical hardware, different technologies such as containers, virtual machines, containers over virtual machines, and serverless functions are described in literature[21].

2.3 API Paradigms

Jin et al. describe in [23] multiple paradigms for web APIs. They divide between Request–Response APIs (like REST, RPC, and GraphQL) and Event-Driven APIs (like WebHooks, WebSockets, and HTTP Streaming). We want to focus on the Request–Response APIs for this work.

According to [23], REST focuses on managing resources through create, read, update, and delete operations (called CRUD pattern). Each URL eighter represents a resource (e.g., a device in a laboratory). Ressources can be statelessly manipulated through standard HTML calls.

In contrast to REST, RPC describes actions [23]. Each URL typically represents an action, which can be called through HTTP containing different parameters and is answered by a response encoded in JSON or XML. There are other protocols available under which RPC can be used.

Finally, Jin et al. [23] describes GraphQL as an API paradigm where the client can define the data structure that the server should provide. Therefore, only a single endpoint is needed. While GraphQL has several advantages for the API user, it increases complexity for the service provider.

2.4 API Modelling Languages + API-First-Design

There are multiple API modeling languages available to describe (REST) APIs: RAML, API Blueprint, or OpenAPI [24]. While all API modeling languages are similar, their syntax and supported tools differ.

Although these languages differ in detail, all allow the usage of the API-First Design: Since all interaction of microservices and all (business) logic is accessible only through API, the modeling of well-designed APIs should be tackled early in microservice development [25]. An API-First Design has multiple advantages, for example, better testability (see [26] as an example) or loose coupling between different components (see [27] as an example). Unfortunately, while the API-First-Design is an emerging trend, there still needs to be more scientific research in this area [25].

3 Architectural Challenges

Our architecture aims to build a system for experiments that cross the line between different types of devices, different types of laboratories, different disciplines, and institutions (see [6] for more information). Because of this and the distributed nature of such a system, we collected the following major requirements. All requirements were collected together with shareholders from each participating institution.

1. **Ease of Implementation**: It should be easy for (new) participants to integrate their devices into the architecture. Therefore, it must be easy to implement the required software interface.
2. **Integrability**: It must be easy for institutes to integrate our architecture into their preexisting IT infrastructure (e.g., for authentication). The system must be flexible where parts of IT infrastructure can be interchangeable (like authentication or cloud services).
3. **Adaptability**: The system should be adaptable to new scenarios, like new experiments, new applications, or new classes of devices.
4. **Scalability**: It should be easy to add new devices or institutions to the system. Newly added devices/institutions should be fully integrable.
5. **Partition tolerant**: The system should be fully functional for an institution even if any (or all) other institutions stop using it (except institution-spanning experiments).

4 High-Level Software Architecture

As the architecture is implementing the new remote laboratory paradigm from [8], this work is heavily influenced by the architecture presented there.

4.1 Structural Description

As shown in Fig. 1, the system consists of a backend, a frontend, and multiple so-called laboratory devices that connect to the backend and – when the experiment is running – to each other.

The backend system uses a gateway to authenticate and route the requests to the microservices that are only available through this gateway. The current architecture uses four services:

- **Device Service** – This service is exclusively responsible for interacting with the devices that will register with the backend.
- **Identity Management Service** – This service will hold all user-relevant data and authenticate them at the gateway's request.
- **Experiment Service** – This service lists, creates, and executes experiments.
- **Booking Service** – This service is responsible for booking devices and selecting an appropriate one if a device from a device group is requested.

The structure is separated in this way to decouple the microservices from each other. Ideally, each service will be containerized and run on a platform for container orchestration. This approach will ensure good scalability towards a more significant network load. The containerized applications also allow for effortless deployment using tools like docker-compose. That way, the application will be deployable for a wide range of people with varying technical expertise.

At the same time, this microservice architecture allows for easy replacement of different services, making the architecture easy to change and evolve. E.g., a common issue will be that different institutions want their respective identity

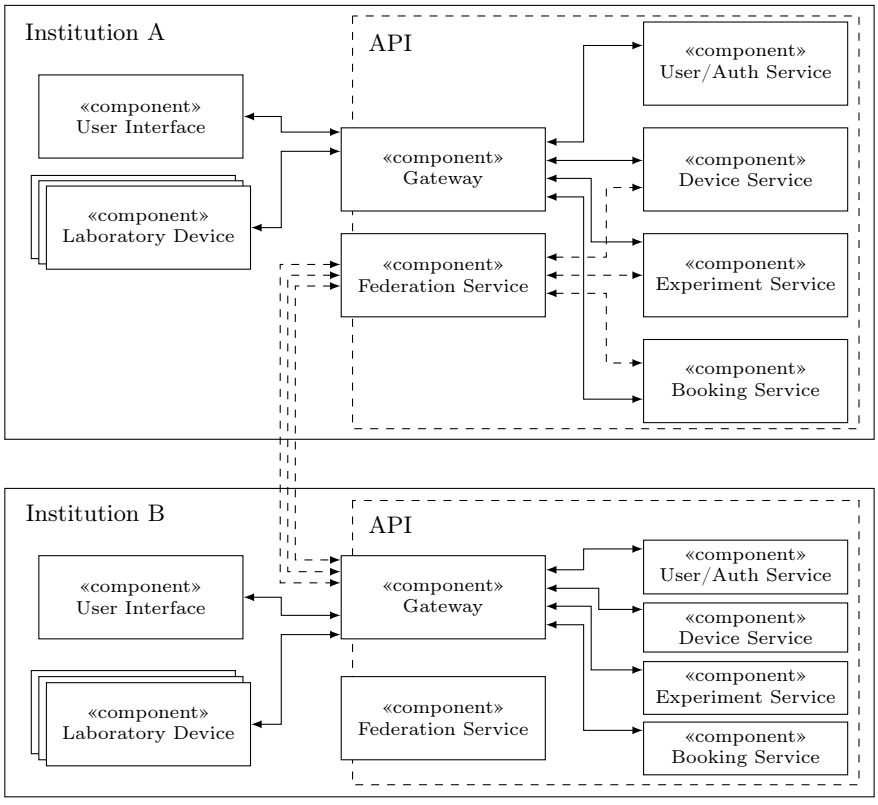

Fig. 1. Crosslab architecture

management systems. As the systems are decoupled, it is possible to change a component without changes to the rest of the architecture.

Connecting to the backend are one or multiple frontends, usually implemented as a website but could also be implemented as a plugin to a learning management system (LMS) such as Moodle. The frontends allow users to create, modify, and run different experiment setups. Because of the new paradigm from [8] that we use here, prior to the execution of an experiment, the front end will also create all necessary laboratory devices for a user that are necessary to view and interact with the other devices in the experiment.

Lastly, the architecture's structure includes a variety of laboratory devices. As indicated above, each device will register with the *Device Service* and then keeps a bidirectional connection to that service to receive instructions for configuration and connection to other devices to form an active experiment.

4.2 Project Organisation

We use an *API first* approach (see Sec. 2.4) because it enables the development of services by multiple institutions in parallel while not waiting for the service implementations of the other institutions. In addition, this approach enables us to fixate the API early and reduces the tedious work of updating the API, especially with different institutions working on the architecture.

With an agreed API specification, it is also possible to validate implementations and ensure that all different implementations of this architecture are working together in a federated setup.

The reference implementation of the architecture will be organized in a mono repository [28]. This approach is chosen over multiple repositories because, in that way, it is easier to keep track of all relevant components and manage access. It should be noted that this decision is based on the fact that the architecture is developed in small academic teams scattered over multiple institutions.

The mono repository is structured in the folder structure seen in Fig. 2. The *docs* directory will contain all the documentation for the architecture in a structure that is translateable by Jekyll to be hosted on a website [4].

[4] This is already automated on the popular source code hosting website GitHub.

```
/
├── docs ..... Jekyll compatible documentation
├── clients ..... Clients for use in user interfaces and devices
├── backend-services
│   ├── auth ..... Authentification and User Management Service
│   ├── booking ..... Booking Service
│   ├── device ..... Device Service
│   ├── experiment ..... Experiment Service
│   ├── federation ..... Federation Service
│   └── gateway ..... Gateway Service
├── helper ..... Helper Tools which are used during development and deployment
│   └── openapi-codegeneration ..... Tool to transform OpenAPI documents to code
└── scripts ..... Scripts for the developer and CI/CD solutions
```

Fig. 2. File Tree for the Monorepository

The *clients* directory will contain all the clients used to interact with the architecture from user interfaces or individual laboratory devices. The *backend-services* directory will contain all the services described above. In this directory, each folder represents a single service. Each service can have its own structure. However, we require a Dockerfile in its root to build a complete Docker container for each service.

The *helper* directory will contain all projects that are not directly deployable for the application. That includes code generation tools and utility libraries that might be used in multiple services. In this directory, each folder represents a single package. The *scripts* directory will contain all scripts used by the developer or the CI/CD solution. These might include test, build, and deployment scripts, as well as scripts for often-needed actions.

5 Fine Level Software Architecture

The Domain Model for the architecture is shown in Fig. 3. We use a role-based access control (RBAC) model, where each user has a role with a set of scopes (e.g., add new devices, list devices, etc.). The laboratory devices can have different types: A single physical or logical device has the type `device`. Our Domain Model also allows for the grouping of devices; the resulting group can also be seen as a device. Lastly, we provide the ability to create devices on the fly and distinguish between devices created in the cloud or at the edge device of the user. The most prominent example of an edge instantiable device is the user interface that shows the webcam and experiment controls to the user in the browser. Each device can have a set of services that it offers; these services generally map to the device's capabilities.

When an experiment starts, all involved devices are connected to each other with peer-to-peer connections. `Peerconnection` objects represent these Connections.

Additionally, each `Peerconnection` holds the service configuration for each service transmitted over the peer-to-peer connection it represents.

Each experiment comprises a set of roles representing one or multiple actual devices, as well as a set of configurations for individual services. At the level of an experiment, each (service-) configuration can have multiple participants, which will be resolved to a peer-to-peer connection by the experiment service when an experiment gets started. This means it is possible to connect, e.g., a webcam to multiple participants. The participants of such configuration are roles that are defined in the experiment. This indirection allows multiple devices to take the same role in an experiment, which is useful when multiple users are involved in an experiment.

According to the REST pattern, everything in our architecture (including devices, experiments, connections between devices, and bookings) is mapped as a resource. Each resource is identified as a unique URL (Uniform Resource Locator) [29]; uniqueness between different institutions is guaranteed because the URL contains the host, which typically is the institution's web address.

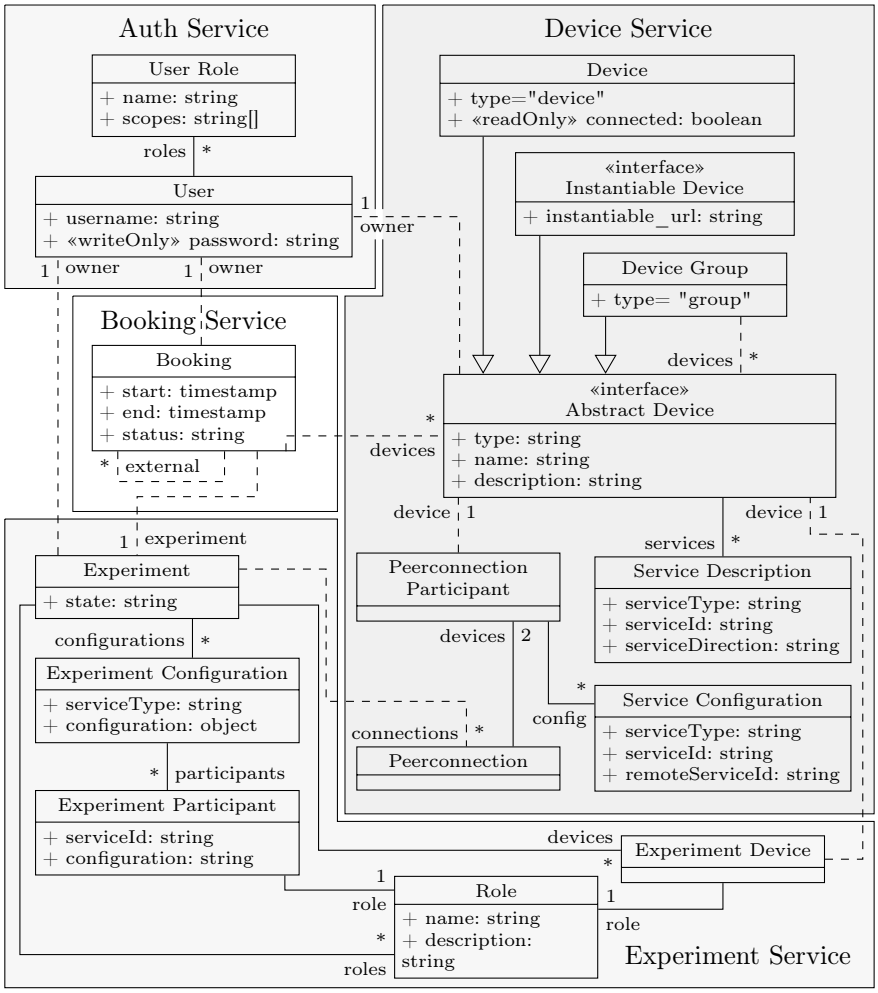

Fig. 3. Domain Model of the API. Dashed associations are realized by an URL reference and, therefore, can be federated over multiple institutions.

All resources can be managed through the CRUD pattern:

- **Create:** New resources can be created using *HTTP POST* request. The body must contain all relevant information for the resource.
- **Read:** Ressources can be read using a *HTTP GET* request to the URL of the resource.
- **Update:** An existing resource can be updated using a *HTTP PATCH* request. Only the attributes that should be changed must be provided, and not all attributes can be changed.
- **Delete:** Ressources can be deleted using a *HTTP DELETE* request to the URL of the resource.

Mapping the domain model from Fig. 3 to resources, we arrive at the following service endpoints described briefly below.

5.1 Authentication Service

The authentication service is responsible for the authentication and management of users and access tokens. For this, it maps the resources `users` and, as a subresource of each user, the resources `roles`. The other services relying on the authentication service will define the roles available.

The authentification service must provide a `/auth` endpoint, which the gateway uses to authenticate users. The endpoint is expected to return a JWT Token [30], which is internally forwarded to the other services by the gateway.

Because of the simplicity of this service, it is easy to replace or extend it with different authentication methods without changing the overall system.

5.2 Device Service

The Device Service manages two resources `devices` and `peerconnections`. The `devices` resource contains the collection of all registered devices independently of their type. Service Descriptions are mapped as an array directly to the device resource.

The `peerconnections` resource contains a collection of `Peerconnection`s. The participants are mapped as an array together with their respective service configurations.

The CRUD mapping of actions to resources is straightforward; this service, however, also provides another `signaling` endpoint for each device. This endpoint allows sending messages to the device via a side channel (WebSockets), which is used to build peer-to-peer connections later on.

5.3 Experiment Service

The experiment service only manages the resources `experiments`. All other objects from Fig. 3 are mapped as an array in the experiment object. The primary purpose of this service is to keep track of all experiments, manage their state (created, booked, setup, running, and finished), and create all necessary `Peerconnection`s when an experiment is started.

5.4 Booking Service

The booking service allows users (here: persons or other services) to book one or multiple devices. All users only access the booking system at their institution, even if they want to book devices from other institutions. The booking service will manage the cross-institutions bookings for the user.

When the user requests a valid booking, it is always accepted and put to the status *pending*. If the booking contains any device group, it is automatically

resolved, and an appropriate device is chosen (devices from their own institutions are prioritized). Once all devices are reserved, the booking changes to *booked*. If any device is unavailable, it changes to *reject* instead. If, at any point, a device is not available anymore, there are two options: If it is from a device group, another device will be booked; else, the booking will change to *rejected*, thus freeing the rest of the devices. When a user wants to start an experiment, he needs to lock the booking, thus preventing most changes (e.g., adding devices is allowed, removing devices is not allowed) until the experiment finishes.

Currently, the booking service only allows for reservations beforehand. However, we later want to support other types of booking (e.g., spontaneous booking, and priority booking, where others can use the devices but lose access if you need them).

5.5 Federation

The federation service is not shown in the domain model but has an important role. Because experiments and devices can be shared across multiple instances of this system, each service must be able to communicate with other instances. As this communication must be authenticated, the authentication service is used to save the credentials for known instances and as an authenticating proxy server for all other instances. (cf. Fig. 1).

6 Takeaways and Future Work

This paper presented the API used to implement the new remotely coupled web experiment paradigm in the Crosslab project. The chosen microservice architecture allows for easy system extension, while the formal specification of the API allows for interoperability between different system instances. The API uses the OpenAPI specification and is used to generate the server code as well as the API client. The decision for the *API first* approach, as well as using the API documentation, has already proven valuable. Using OpenAPI as the specification language allowed us to generate different artifacts out of it: Automatically updated documentation, generation of an API client, and the generation of vast amounts of server code, including automatic validation of incoming requests. In the future, we plan to include automatic tests based on the API specification.

However, there are a few downsides to this we experienced. While most of our initial API design has held up, there were minor details (like the representation of different kinds of devices or the way specific endpoints had to be called) that we did not get right in the first draft. We updated the API, but that changed the generated code leading to problems across the existing code base (some of which were hard to spot). The API is currently in active development and will be extended and refined in the future. The source code and complete documentation are publicly available on Github.

Acknowledgement. This work as part of the project CrossLab [6] is funded by Stiftung Innovation in der Hochschullehre

References

1. Satterthwait, D.: Why are 'hands-on' science activities so effective for student learning? Teaching Science 56(2), 7–10 (2010), doi: 10.3316/aeipt.182048
2. Forcino, F.L.: The Importance of a Laboratory Section on Student Learning Outcomes in a University Introductory Earth Science Course. Journal of Geoscience Education 61(2), 213–221 (2013), doi: 10.5408/12-412.1
3. Bretz, S.L.: Evidence for the importance of laboratory courses. Journal of Chemical Education 96(2), 193–195 (2019), doi: 10.1021/acs.jchemed.8b00874
4. Powell, R.M., Anderson, H., Van der Spiegel, J., Pope, D.P.: Using web-based technology in laboratory instruction to reduce costs. Computer Applications in Engineering Education 10(4), 204–214 (2002), doi: 10.1002/cae.10029
5. Keleş, D., Bulgurcu, A., Demir, E.F., Şemin, I.M.: The effect of virtual laboratory simulations on medical laboratory techniques students' knowledge and vocational laboratory education. Turkish Journal of Biochemistry 47(4), 529–537 (2022), doi: 10.1515/tjb-2020-0619
6. Aubel, I., Zug, S., André, D., Nau, J., Henke, K., Helbing, P., Streitferdt, D., Terkowsky, C., Boettcher, K., Ortelt, T.R., Schade, M., Kockmann, N., Haertel, T., Wilkesmann, U., Finck, M., Haase, J., Herrmann, F., Kobras, L., Meussen, B., Soll, M., Versick, D.: Adaptable digital labs - motivation and vision of the crosslab project. In: IEEE German Education Conference 2022. Berlin (2022), in press
7. Henke, K., Nau, J., Streitferdt, D.: Hybrid take-home labs for the stem education of the future. In: Uskov, V.L., Howlett, R.J., Jain, L.C. (eds.) Smart Education and e-Learning - Smart Pedagogy. pp. 17–26. Springer Nature Singapore, Singapore (2022), doi: 10.1007/978-981-19-3112-3_2
8. Nau, J., Henke, K., Streitferdt, D.: New ways for distributed remote web experiments. In: Auer, M.E., Pester, A., May, D. (eds.) Learning with Technologies and Technologies in Learning. Springer International Publishing, Cham (2022), doi: 10.1007/978-3-031-04286-7_13
9. Soll, M., Haase, J., Helbing, P., Nau, J.: What are we missing for effective remote laboratories? In: IEEE German Education Conference 2022. Berlin (2022), in press
10. Henke, K., Vietzke, T., Hutschenreuter, R., Wuttke, H.-D.: The remote lab cloud GOLDi-labs.net. In: 2016 13th International Conference on Remote Engineering and Virtual Instrumentation (REV). pp. 37–42 (2016), doi: 10.1109/REV.2016.7444437
11. Orduña, P., Rodriguez-Gil, L., Garcia-Zubia, J., Angulo, I., Hernandez, U., Azcuenaga, E.: LabsLand: A sharing economy platform to promote educational remote laboratories maintainability, sustainability and adoption. In: 2016 IEEE Frontiers in Education Conference (FIE). pp. 1–6 (2016), doi: 10.1109/FIE.2016.7757579
12. Orduña, P., Garcia-Zubia, J., Rodriguez-Gil, L., Angulo, I., Hernandez-Jayo, U., Dziabenko, O., López-de Ipiña, D.: The WebLab-Deusto Remote Laboratory Management System Architecture: Achieving Scalability, Interoperability, and Federation of Remote Experimentation, pp. 17–42. Springer International Publishing, Cham (2018), doi: 10.1007/978-3-319-76935-6_2
13. Gustavsson, I., Zackrisson, J., Håkansson, L., Claesson, I., Lagö, T.: The VISIR project – an Open Source Software Initiative for Distributed Online Laboratories. In: 2007 International Conference on Remote Engineering and Virtual Instrumentation (REV). pp. 1–6 (2007)
14. May, D., Reeves, B., Trudgen, M., Alweshah, A.: The remote laboratory visir - introducing online laboratory equipment in electrical engineering classes. In:

2020 IEEE Frontiers in Education Conference (FIE). pp. 1–9 (Oct 2020), doi: 10.1109/FIE44824.2020.9274121
15. Rodriguez-Gil, L., Orduña, P., García-Zubia, J., López-de Ipiña, D.: Advanced integration of OpenLabs VISIR (Virtual Instrument Systems in Reality) with Weblab-Deusto. In: 2012 9th International Conference on Remote Engineering and Virtual Instrumentation (REV). pp. 1–7 (2012), doi: 10.1109/REV.2012.6293150
16. Haug, S., Wedekind, J.: "Adresse nicht gefunden" – Auf den digitalen Spuren der E-Teaching-Förderprojekte. In: E-Learning: Eine Zwischenbilanz. Kritischer Rückblick als Basis eines Aufbruchs, pp. 19–37 (2009), doi: 10.25656/01:3215
17. Lowe, D., Murray, S., Weber, L., de la Villefromoy, M., Johnston, A., Lindsay, E., Nageswaran, W., Nafalski, A.: LabShare: Towards a National Approach to Laboratory Sharing. In: Proceedings of the 20th Annual Conference for the Australasian Association for Engineering Education (2009)
18. Harward, V.J., del Alamo, J.A., Lerman, S.R., Bailey, P.H., Carpenter, J., DeLong, K., Felknor, C., Hardison, J., Harrison, B., Jabbour, I., Long, P.D., Mao, T., Naamani, L., Northridge, J., Schulz, M., Talavera, D., Varadharajan, C., Wang, S., Yehia, K., Zbib, R., Zych, D.: The ilab shared architecture: A web services infrastructure to build communities of internet accessible laboratories. Proceedings of the IEEE 96(6), 931–950 (June 2008), doi: 10.1109/JPROC.2008.921607
19. iLabs, https://icampus.mit.edu/projects/ilabs/, last accessed: 21.10.2022 9:59
20. Newman, S.: Building Microservices. O'Reilly Media, Inc., Gravenstein Highway North, Sebastopol, 2 edn. (2021)
21. Di Francesco, P., Lago, P., Malavolta, I.: Architecting with microservices: A systematic mapping study. Journal of Systems and Software 150, 77–97 (2019), doi: 10.1016/j.jss.2019.01.001
22. Soldani, J., Tamburri, D.A., Van Den Heuvel, W.J.: The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software 146, 215–232 (2018), doi: 10.1016/j.jss.2018.09.082
23. Jin, B., Sahni, S., Shevat, A.: Designing Web APIs - Building APIs That Developers Love. O'Reilly Media, Inc., Gravenstein Highway North, Sebastopol, 1 edn. (2018)
24. Surwase, V.: Rest api modeling languages - a developer's perspective. International Journal of Science Technology & Engineering 2(10), 634–637 (2016)
25. Beaulieu, N., Dascalu, S.M., Hand, E.: Api-first design: A survey of the state of academia and industry. In: Latifi, S. (ed.) ITNG 2022 19th International Conference on Information Technology-New Generations. pp. 73–79. Springer International Publishing, Cham (2022), doi: 10.1007/978-3-030-97652-1_10
26. Bennett, B.E.: A practical method for api testing in the context of continuous delivery and behavior driven development. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 44–47 (April 2021), doi: 10.1109/ICSTW52544.2021.00020
27. Dudjak, M., Martinović, G.: An api-first methodology for designing a microservice-based backend as a service platform. Information Technology And Control 49(2), 206–223 (Jun 2020), doi: 10.5755/j01.itc.49.2.23757
28. Potvin, R., Levenberg, J.: Why google stores billions of lines of code in a single repository. Commun. ACM 59(7), 78–87 (jun 2016), doi: 10.1145/2854146
29. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform resource identifier (uri): Generic syntax. RFC 3986 (2005), doi: 10.17487/RFC3986
30. Jones, M., Bradley, J., Sakimura, N.: Json web token (jwt). RFC 7519 (2015), doi: 10.17487/RFC7519