AutoIoT: a Framework based on User-driven MDE for Generating IoT Applications

Thiago Nepomuceno, Tiago Carneiro, Paulo Henrique Maia, Muhammad Adnan, Thalyson Nepomuceno, Alexander Martin

AutoIoT: a Framework based on User-driven MDE for Generating IoT Applications

Thiago Nepomuceno
Fraunhofer Center for Applied Research on Supply Chain Services
Germany
thiago.nepomuceno@scs.fhg.de

Tiago Carneiro
INRIA Lille - Nord Europe
France
tiago.carneiro-pessoa@inria.fr

Paulo Henrique Maia
State University of Ceará
Brazil
pauloh.maia@uece.br

Muhammad Adnan
Fraunhofer Center for Applied Research on Supply Chain Services
Germany
adnanmd@scs.fhg.de

Thalyson Nepomuceno
Federal Institute of Ceará
Brazil
thalyson.silva@ifce.edu.br

Alexander Martin
Friedrich-Alexander-Universität Erlangen-Nürnberg
Germany
alexander.martin@fau.de

ABSTRACT

Developing an Internet of Things (IoT) system requires knowledge in many different technologies like embedded programming, web technologies, and data science. Model-Driven Engineering (MDE) techniques have been used as a concrete alternative to boost IoT application development. However, the current MDE-to-IoT solutions require expertise from the end-users in MDE concepts and sometimes even in specific tools, such as the Eclipse Modelling Framework, which may hinder their adoption in a broader context. To tackle this problem, this work proposes AutoIoT, a framework for creating IoT applications based on a user-driven MDE approach. The proposed framework allows users to model their IoT systems using a simple JSON file and, through internal *model-to-model* and *model-to-text* transformations, generates a ready-to-use IoT server-side application. The proposed approach was evaluated through an experiment, in which 54 developers used AutoIoT to create a server-side application for a real-world IoT scenario and answered a post-study questionnaire. The experiment reports the efficacy of AutoIoT and user satisfaction of more than 80% through 6 out of 7 evaluated criteria.

CCS CONCEPTS

• **Computer systems organization** → *Sensors and actuators*; • **Software and its engineering** → *Software prototyping*;

KEYWORDS

Code Generation, User-driven, MDE, IoT application

ACM Reference Format:
Thiago Nepomuceno, Tiago Carneiro, Paulo Henrique Maia, Muhammad Adnan, Thalyson Nepomuceno, and Alexander Martin. 2020. AutoIoT: a Framework based on User-driven MDE for Generating IoT Applications. In *Proceedings of* . ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Over the past years, the interest in the Internet of Things (IoT) technologies has grown both in industry and academia due to its wide range of applications, including smart homes, smart healthcare, smart grids, smart cities, and smart factories [16]. IoT projects usually share some essential features, such as sending and receiving data to/from devices, analyzing and visualizing data, and providing a user management system [2].

A complete IoT system is composed of four different layers [10]: device layer, gateway, a server-side application, and data analysis. The usual data flow in an IoT system starts when a device sends it's sensor data using low-level communication protocols (e.g., sockets) to a gateway. Then, the gateway transforms the binary data received into a more human readable format (e.g., JSON) and sends it to the server-side application, which is responsible for collecting and storing the data in a database. Additionally, the server application can also provide features like a virtual representation of the devices, a Graphical User Interface (GUI) to visualize the data, a user management system, and ways to share collected data with third-party systems (e.g., via an API or a message queue). Finally, the data analysis subsystem accesses the data stored and produces useful insights to help the users of the IoT system to understand their data and to take decisions accordingly.

In some cases, an IoT system does not need all four layers. Examples include (i) when devices are powerful enough to support internet protocols a gateway is not needed; (ii) in prototypes or simpler systems the *data analysis* layer is not required; and (iii) in some cases, even the *server-side application* is not needed, since devices can communicate directly between them autonomously.

Developing a complete IoT system requires expertise in many different technologies like embedded programming, web technologies, and data science. Given the complexity of the task, some tools and platforms have been created to support developers. The better well-known solutions are the platforms created by big companies like Google[1], Amazon[2], and Microsoft[3]. However, costs associated

[1] https://cloud.google.com/iot-core/
[2] https://aws.amazon.com/iot/
[3] https://azure.microsoft.com/en-us/services/iot-hub/

with the provided services, as well as the vendor lock-in, may hinder the adoption of such solutions for some smaller or more specific IoT projects.

Alternatively, Model-Driven Engineering (MDE) has been gaining attention from both academia and industry in recent years as a concrete solution to ease the development of IoT systems through automatic code generation to different hardware devices [12] and server-side application platforms [3, 6, 11, 13]. Nonetheless, most of those approaches to build IoT applications present at least one of the following drawbacks: (i) they are based on a strong modeling phase with complex meta-models that are designed to cover a wide range of different scenarios, hence requiring time and modeling expertise; (ii) they generate only boilerplate code of the application, demanding developers to write most of the application logic by hand; and (iii) they require expertise in tools that are standard in the MDE community, but not well known in the industry, like the Eclipse Modeling Framework (EMF) suite.

According to a definition given by Abrahão *et. al.* [1], those work are considered as *technology-driven approaches*, in which the primary goal is to improve existing MDE techniques, usually making them more general in order to cover a broader range of use cases, and not to improve the usability or to ease the adoption by a general target audience. Most of those works require their audience to became MDE experts, learning theory and techniques commonly used in the MDE field. On the other hand, in a *user-driven approach* [1], the user knowledge and necessities guide the development of the new MDE approach. This way, such approach should provide tools that help to bridge the gap between what its users already know and what is expected from them to know to use the new tools/methods, making the adoption easier.

Previous work have already followed the premise behind user-driven MDE approaches. GenApp [15], for example, is a tool for building entirely functioning science gateways and standalone GUI applications from collections of definition files and libraries of code fragments, and Json-GUI [5] is an AngularJS front-end module that dynamically allows data scientists and web form users to generate form-based web interfaces. However, to the best of our knowledge, we have not found any work that applies the concepts of user-driven MDE for the design and development of IoT applications. This kind of approach would benefit not only experienced IoT developers to boost the application creation by reducing development time, but also introducing new ways to build IoT systems to developers and practitioners by decreasing the technology learning curve.

In this realm, this work proposes a framework, called AutoIoT, for generating IoT server-side applications that relies on a user-driven MDE approach. This way, AutoIoT allows users to model their IoT system using a simple JSON file and through internal model-to-model transformation and code generation techniques, to create automatically the IoT server-side application. Furthermore, IoT developers that have some expertise in MDE can extend AutoIoT by creating specific components, called *Specialized Builders*, that are responsible for generating the source code of specific IoT applications. The proposed framework is evaluated with users with different backgrounds who modeled and developed a real IoT server-side application. The results indicate that the users were able to create the application correctly and in a short time with no or little knowledge in MDE.

The remainder of this document is organized as follows: Section 2 presents the most related work. Section 3 introduces the AutoIoT Framework, describing the user-driven process and implementation details. Next, Section 4 details the evaluation of AutoIoT in an industrial use case and its findings and Section 5 briefly discuss how to use AutoIoT in more complex use cases. Finally, the conclusions are outlined in Section 6.

2 RELATED WORK

Previous work have already used MDE technologies to ease the development of IoT server-side applications. We describe some of these approaches as follows.

Brambilla, Umuhoza, and Acerbis [3] propose a model-driven approach to design IoT Graphical User Interfaces (GUI). The work defines both specific components and design patterns using a visual modeling language for IoT applications and discusses the importance of having good GUI to improve the acceptance of the IoT system and to model the user interactions with the GUI. The paper discusses the back-end software and how it stores and retrieves information from a database, but it is not clear whether the proposed approach also generates those components or not. The approach requires users to have expertise in MDE and Interaction Flow Modeling Language (IFML).

ThingML (Internet of Things Modeling Language) [11] is an MDE approach that aims to cover a considerable amount of IoT use cases. It is already in a more advanced stage of development when compared with other MDE-to-IoT solutions and is especially attractive to teams with MDE experts. According to the authors, the main contribution of the project is a code generation framework and an associated methodology to give practitioners full control of the code by letting them easily customize compilers for their needs. To accomplish this, expertise in MDE and *Domain Specific Language* (DSL) is required.

Pramudianto *et al.* [13, 14] proposes IoTLink, which aims to hide many of the required expertise to develop an IoT server-side application behind a Flow-Based Programming interface. It targets inexperienced developers entering the IoT development, allowing them to create a ready-to-use application in minutes. While IoTLink makes an important step and is aware of the limited expertise of developers, it still requires developers to learn new skills to get started with the tool, like Flow-Based Programming and the Eclipse Modeling Framework. Additionally, the resulting system lacks some essential features usually requested in an IoT application, like a GUI.

All aforementioned work require some level of MDE expertise to create IoT applications, which may hinder their adoption. In contrast, AutoIoT has been designed based on a user-driven MDE approach and uses only technologies that its target audience already knows, like JSON representation and a general programming language[4]. This way, IoT and Web developers with no or little knowledge in MDE techniques can design and generate ready-to-use IoT server-side applications, thus reducing development effort and the technology learning curve.

[4]Currently the AutoIoT framework is available only in Python, but ports to other programming languages are planned

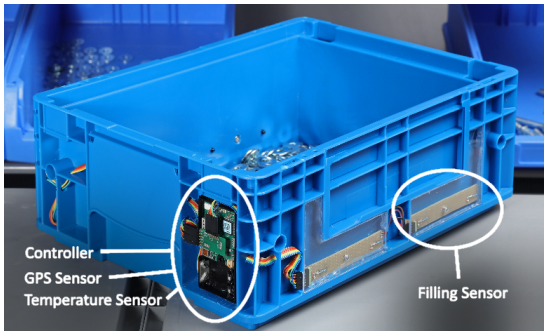

Figure 1: The hardware device and its sensors are attached to a container. The device includes a temperature, position (GPS) and filling status sensors. This picture shows a prototype of the smart container where all sensors are visible, not the final product.

Finally, AutoIoT does not intend to replace more complete MDE-to-IoT approaches (e.g., ThingML), since these approaches are designed to cover a broader range of IoT use cases. Instead, it proposes a solution that allows developers to get some of the benefits of MDE without the need of learning new technologies.

3 AUTOIOT FRAMEWORK

This section presents the proposed framework for generating IoT server-side applications. Initially, a simple motivating example that will be used throughout this section is presented and, subsequently, the user-driven MDE process is described. After that, one type of a *Specialized Builder* component is detailed. Finally, implementation issues about the framework are presented.

This work focuses on how AutoIoT can be used to create a prototype for the motivating example, but all the concepts presented can be expanded to cover a more complex IoT use case, as discussed in Section 5.

3.1 Motivating example

The use case presented in this section consists of smart containers that send information about location, temperature, and filling status (whether the container is empty or not). In this scenario, the IoT device is a hardware module attached to each container, as shown in Figure 1, that sends sensor data to a base station using a proprietary communication protocol called MIOTY, it is similar to LoRa protocol and has a range of 15 kilometers. Then, the base station, which is connected to the Internet and works as a gateway, transmits data to the server-side application through the MQTT protocol. In turn, the server-side application is also in the Internet, hosted on a cloud server. The data sent by the base station is codified in JSON format and contains the location of the box (GPS), temperature, and filling status.

The server software in this scenario needs to (i) create a virtual representation of the container. Additionally, it (ii) receives and processes messages from the base station (using MQTT protocol), and links received data to the correspondent virtual device. Furthermore, it also needs (iii) to store sensor data in a database and (iv) to provide a GUI for easing the user interaction with the system and visualization of data, both historical (from the database) and real-time data. Finally, the system should (v) provide an API that allows project partners (third-party systems) to access the stored data.

3.2 The User-driven MDE Process

The MDE process proposed by this work, as shown in Figure 2, is composed of three main phases: modeling the IoT Scenario, generating server-side application code, and extending the source code. The first two phases are mandatory, while the last one is optional. Each phase is detailed as follow.

3.2.1 Modeling the IoT Scenario. The process starts with the developer modeling the IoT scenario, which is defined according to the meta-model depicted by Figure 3. The meta-model defines an IoT project and its components, i.e., devices, sensors and the data sent by them, besides the system communication with devices or third-party applications through communication protocols (MQTT, HTTP, and WebSockets) and the connection to a database. Additionally, it can also describe the data visualization using dashboards and components, such as tables, charts, and maps.

The meta-model has been created based on our experience on developing several IoT projects for both research and industry in *Fraunhofer SCS*. It can not be considered complete and does not intend to cover all IoT use cases. Its main goal is to provide all information needed by the current *Specialized Builders*, and when new *Builders* are created, the meta-model will be updated accordingly.

Figure 3 depicts components represented by either dotted or continuous lines. A valid model that conforms to the proposed meta-model only needs to define the continuous-line components. The dotted-line ones are optional and will be generated by using a *model-to-model* transformation (M2M). This way, the *code* attribute in *Topic*, *Endpoint*, *WebSockets* and *Dashboard* does not need to be defined by the developer in the application model. Only a description of the system is required, removing the need for defining any business logic in the modeling phase.

To model the IoT scenario, users can write the JSON file manually or use a Web GUI that will generate such a file. Independently of the chosen method, the final result of the modeling phase is always a JSON file containing the model representing the system, and that is in accordance with the meta-model presented in Figure 3.

Considering the example of the smart container, a model used as input to AutoIoT can be seen in Listing 1. In this example, the developer only gives information about the continuous-line components: the project, database, MQTT broker configuration, and the device and its sensor. The device's *fields* in line 19 represents attributes of the device itself and are not supposed to change very often. The same happens for the sensor's *fields* in line 26. However, in addition to *fields*, sensors can have *data_fields* that represents the data that is regularly sent by this kind of sensor to the server application.

The JSON file is the only artifact created by the user during the modeling phase and the only expertise required is knowing the JSON syntax (if the file is written manually, without the GUI). Developers extensively use JSON to exchange data between systems and

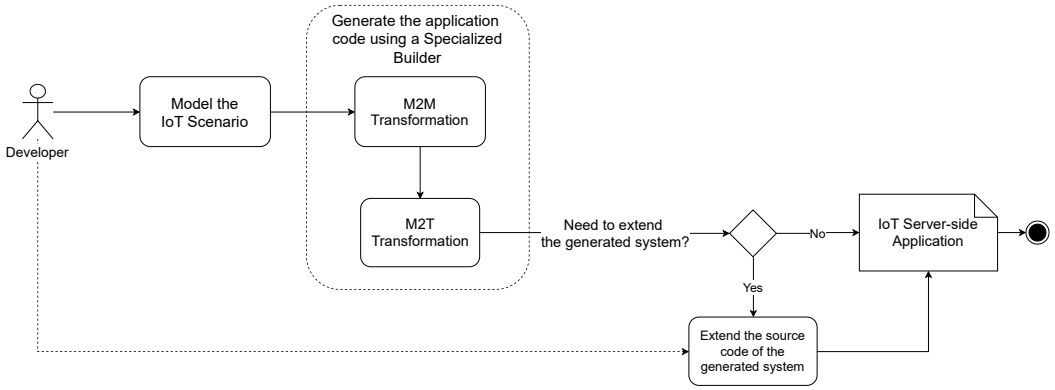

Figure 2: The developer starts the process by creating the model that represents the IoT scenario. It can be done using a GUI, programming library or a simple JSON file. The defined model is the input for a Specialized Builder that, through M2M and M2T transformation, generates the IoT server-side application source code. Optionally, the developer can also extend the generated application to fulfill further requirements.

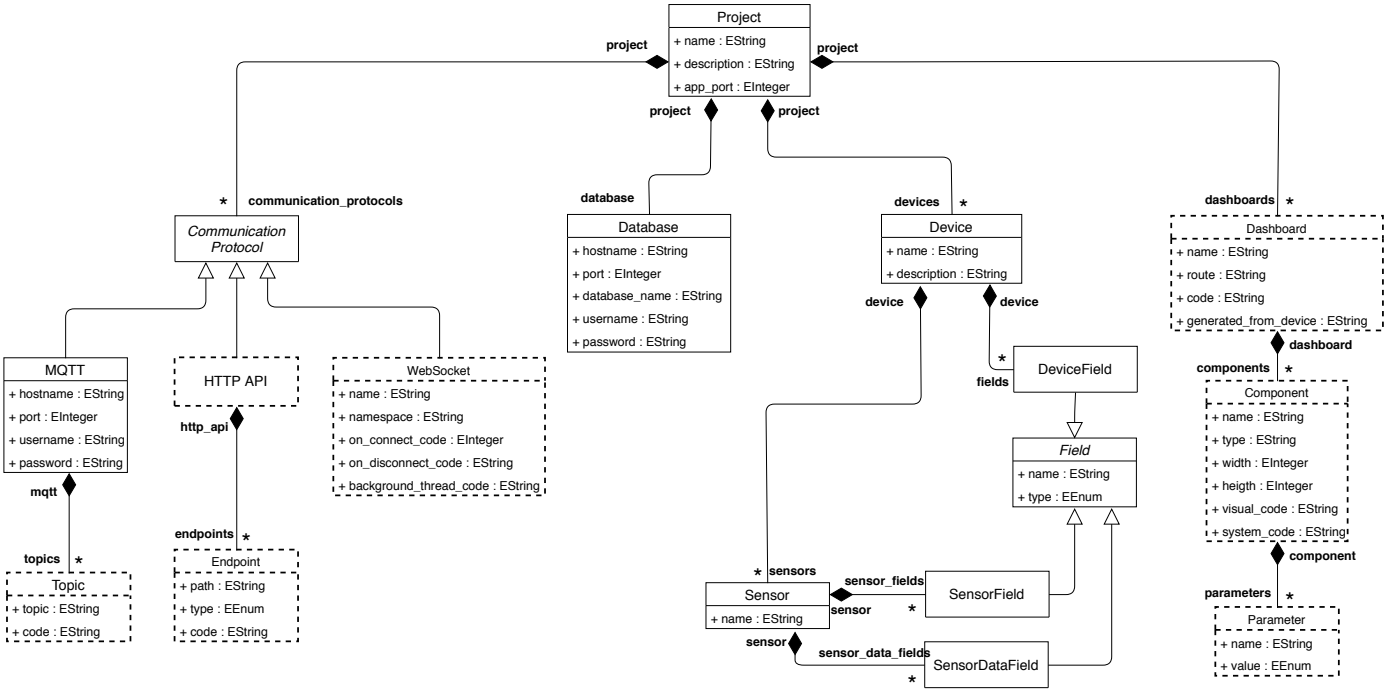

Figure 3: The meta-model that defines an IoT project and its components. Components represented by continuous line are obligatory and need to be present in the model created based on this meta-model. Components represented by dotted line are optional and will be generated using a *model-to-model transformation* (M2M) (explained in Section 3.2).

most of the *General Programming Languages* have native support to JSON.

After the modeling is completed, AutoIoT loads the content of the model file using a function. The first step is to validate and transform the JSON file into Python objects using the Pydantic[5] library. After that, the framework finally delivers these objects to the appropriated *Builder* that will generate the application code.

3.2.2 Generating server-side application code. After the modeling phase is finished, the created model is used as input to a *Specialized Builder* that starts an *M2M transformation*. Then, the extended model goes to an *M2T transformation* phase that generates the source code of an IoT server-side application. The type of application generated depends on the *Builder* used. This work implements the *Prototype Builder* (PB), a *Specialized Builder* that produces an IoT system to manage IoT devices. The PB, the M2M, and the M2T transformations are detailed in Section 3.3.

[5]https://github.com/samuelcolvin/pydantic

```json
{
  "project": {
    "name": "Container Management Project",
    "description": "The Container Management Project.",
    "app_port": 5000
  },
  "database": {
    "type": "sqlite",
    "hostname": "localhost"
  },
  "mqtt": {
    "hostname": "iot.eclipse.org",
    "port": 1883
  },
  "devices": [
    {
      "name": "Container",
      "description": "An IoT device that sends temperature, position and filing status.",
      "fields": {
        "name": "String",
        "barcode": "String"
      },
      "sensors": [
        {
          "name": "MainSensor",
          "fields": {
            "send_interval": "String"
          },
          "data_fields": {
            "temperature": "Float",
            "filing_status": "Integer",
            "position": "Point"
          }
        }
      ]
    }
  ]
}
```

Listing 1: Example of a model represented in JSON format

Listing 2 presents a small example of the AutoIoT Framework being used to load a *project.json* file and shows how to generate the IoT server-side application using a *Specialized Builder*.

The *build* method is responsible for applying the M2M and M2T transformation to the model created by the *load_project* method. In turn, the parameters of the *build* method are the chosen *Specialized Builder*, the output folder and a dictionary with additional configuration. In this example the *Prototype Builder* is used, and the Docker[6] deployment is set to `false`.

3.2.3 Extending the generated code. After the source code generation, the developer has the choice to extend it to fulfill further requirements of the project or use the generated application as is. The generated application is already ready-to-use, and it can be executed locally or deployed on a cloud server.

3.3 Prototype Builder

AutoIoT provides an implemented *Specialized Builder* called *Prototype Builder* (PB). It has been chosen as the first *Specialized Builder* due to its wide use as a server-side application in several IoT projects, since every project usually needs a prototype and most IoT scenarios need to communicate with and manage IoT devices.

[6]Docker is a tool designed to make easier to deploy, run and maintain applications by using containers technology.

```python
from autoiot import AutoIoT
from autoiot.builders import PrototypeBuilder

autoiot = AutoIoT()
autoiot.load_project('project.json')
autoiot.build(PrototypeBuilder, 'output/project', {'docker': False})
```

Listing 2: Using AutoIoT to load a model file and generate an IoT server-side application

The PB generates a Flask application[7] written in Python, HTML, CSS, and Javascript. The generated server-side application communicates with IoT devices and third-party system through MQTT, Rest API, and WebSockets. Moreover, the generated application manages IoT devices (creates, edits and deletes them), stores and visualizes sensor data sent by them. Additionally, it also provides a user management feature that controls access to the stored data.

Figure 4 outlines more *Specialized Builders* that are planned for future work. Each *Specialized Builder* should follow the metamodel proposed by this work. However, the set of components that are mandatory (continuous line) or optional (dotted-line) can vary from one *Builder* to another. Each *Builder* has to implement an M2M transformation that extends the input model adding custom components (dotted-line components) and an M2T transformation that generates the final application source code.

To create the PB, the abstract class Builder is extended as shown in Figure 4. The Builder class has a set of abstract methods that need to be implemented by the *Prototype Builder* class. The most important ones are *extend*, which performs the M2M transformation, and *generate* that is responsible for the M2T transformation

[7]http://flask.pocoo.org/

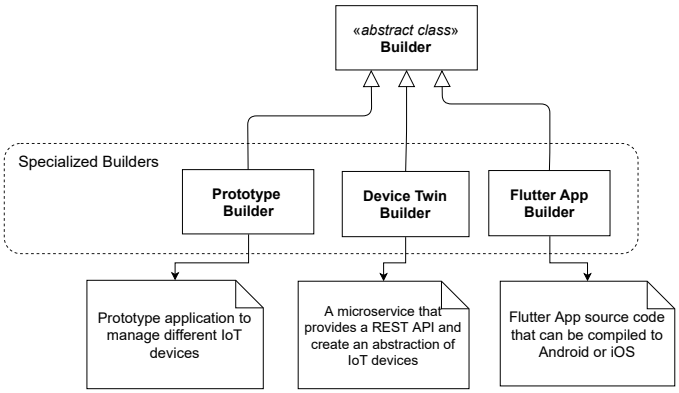

Figure 4: A developer can create *Specialized Builders* extending the *abstract class* Builder provided by AutoIoT. Each Builder receives the same model but produces different applications. The generated application can follows a monolith architecture like the application generated by the *Prototype Builder*, a microservice architecture like one generated by the *Device Twin Builder* or even create a complete mobile application if the *Flutter App Builder* is used.

(Section 3.3.1 and Section 3.3.2). Following the transformations executed by the PB are detailed.

3.3.1 M2M Transformation. When the model defined by the user is loaded by AutoIoT (line 5 of Listing 2) it is validated, processed and stored in memory as a graph, whose nodes are components of the model. The M2M transformation consists of visiting each node of the graph and checking its type. Examples of types include *Project*, *MQTT*, *Database*, *Device* and *Sensor*. For each type of component, the transformation creates specific dotted-line components and attaches them to the input model. For example, whenever the transformation finds a *Device* component in the original model, it creates a *Topic*, *HTTP API*, *WebSocket* and *Dashboard* components and includes them in the original model.

3.3.2 M2T Transformation. Similar to the process described in the M2M transformation (Section 3.3.1), in the M2T transformation the PB also visits each node of the graph and checks its type. For each different type, it triggers a different function that generates source code for the application using the templating technique. The default templating engine used by AutoIoT is Jinja[8], commonly used inside the Flask Framework to render HTML content.

Each triggered function receives as input a component defined in the model and generates part of the IoT application source code. For example, whenever the search finds a *Device* type component, the *found_device* function is triggered. It receives the *Device* object as input and generates a Python class that represents this type of *Device*. Furthermore, the *found_device* function also generates a controller class to feed the GUI with information from the database concerning this type of device. Additionally, it also generates HTML files that list the devices stored in the database and performs the CRUD operations (Create, Read, Update and Delete). However, in case the graph search finds a *Dashboard* instead of a *Device*, it triggers a different function that generates HTML, CSS, and Javascript files that compounds the GUI.

At the end of the M2T transformation process, the PB generates the source code of a Flask project. This project is written in Python, HTML, CSS, and Javascript, communicates with IoT devices, and stores its sensor data in a database. The Flask project supports different communication protocols (e.g., HTTP, Web Sockets and MQTT) that allows third-party systems to communicate using various web technologies. Furthermore, it provides a GUI that can be used to manage the IoT devices, and visualizes sensor data using tables, charts, and maps updated in real time. Finally, it also provides a user management system to allows different levels of access to the stored data. Figure 5 depicts a dashboard page of the system generated for the smart container motivating example.

3.4 Implementation Details

AutoIoT Framework is implemented using Python 3 and uses the Pydantic library to provide model (JSON file) validation and processing.

The Jinja is the default template engine used for the M2T transformation (but other options can be used). It is the standard template engine used by the Flask framework, which is used to render HTML pages. Since developers are already familiar with its syntax, they can easily change the templates if there is a need for it.

To help developers implement both transformations (M2M and M2T), the *Builder* class provides a function that goes through the whole model and triggers functions depending on the type of the component found during the search as already described in Section 3.3. After the implementation of the callback functions, which are triggered when the search find each type of component, the *Specialized Builder* became very simple as shown in Listing 3.

```
1  from .callback_functions import found_device, found_project,
   ↪  found_database
2
3  class MySpecializedBuilder(Builder):
4
5      def __init__(self, project, output_path, config):
6          self.project = project
7          self.output_path = output_path
8          self.config = config
9
10     def extend(self):
11         self.register_callback(Project, found_project)
12
13         self.search()
14
15     def generate(self):
16         self.register_callback(Device, found_device)
17         self.register_callback(Database, found_database)
18
19         self.search()
```

Listing 3: A simplified example of a Specialized Builder class. This is not a working source code, some parts like libraries importing and inheritance specific details were removed to increase the overall readability.

Other *Specialized Builders* can be created by developers and incorporated to AutoIoT (as shown in Figure 4). The only restriction is that it receives as input a JSON object containing a model that follows the proposed meta-model as shown in Listing 2. There are no restrictions about the type of system the *Specialized Builder* can generate (Web, Mobile or Desktop) or the underlying programming language used to implement it. Even that anyone can create a *Builder* and incorporate to AutoIoT, in a normal workflow developers do not need to create any *Specialized Builders* since most of the *Builders* are provided by the AutoIoT Framework itself or its community, similar to how it happens when choosing a third-party programming library.

4 EVALUATION

To evaluate the AutoIoT framework an experiment has been conducted. It consists of using AutoIoT to generate the server-side application of an IoT use case, extracted from one of our industrial projects. After trying AutoIoT, the developers answered a post-study questionnaire to evaluate their experience. This helps us to understand what technologies should be used when developing tools for this target audience: IoT and Web developers.

In the rest of this section the use case, the experiment, and the results are detailed.

[8] http://jinja.pocoo.org/

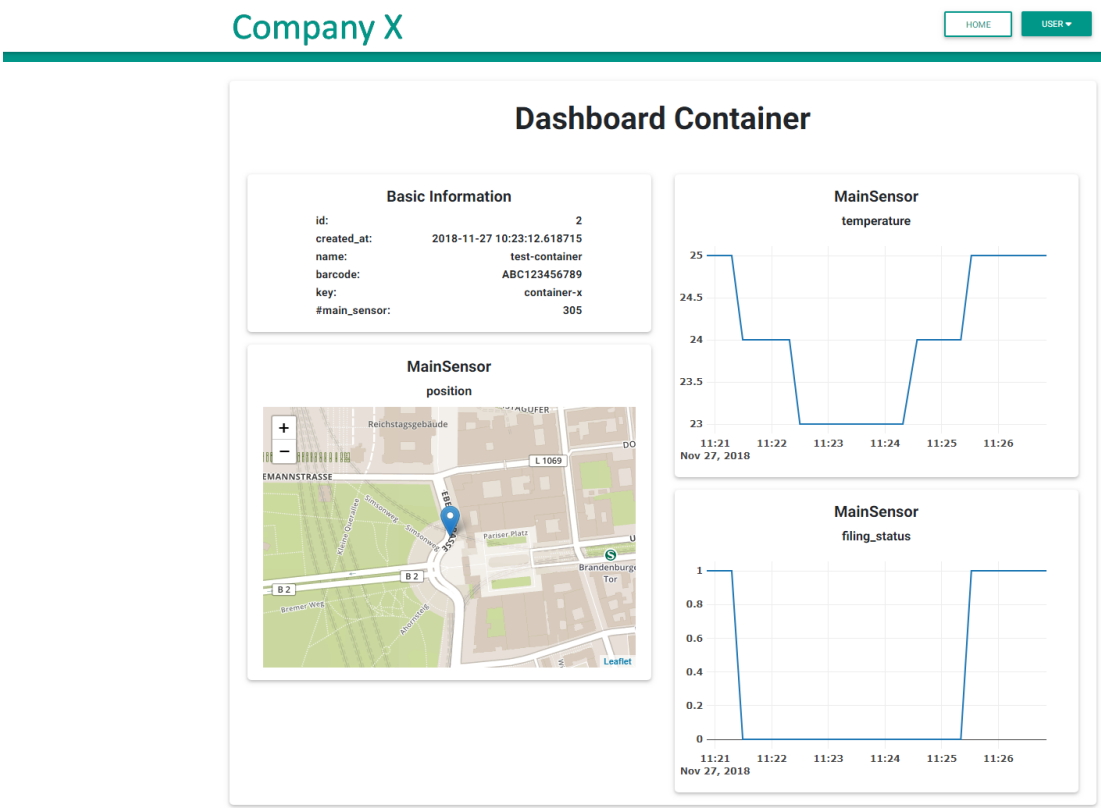

Figure 5: A dashboard page of the prototype application generated by the *Prototype Builder* for the motivating scenario. It allows visualisation of the data sent by a container, including basic information about the container and its sensor data (temperature, position and filing status).

4.1 Experiment

To evaluate the proposed framework an experiment was conducted. First, 5 developers from both academy and industry were invited to use AutoIoT to develop the application for the use case described in Section 3.1 and answer the post-study questionnaire (available on Google Forms[9]). This initial phase was used to improve both the experiment and the questionnaire itself.

In the second phase of the experiment, developers from different companies and research labs around the world were invited to try AutoIoT. The only requirement needed was to have previous experience with IoT or Web development. In total, 54 developers finished the experiment and completed the questionnaire. The evaluation was performed by $17 - 56$ years old participants ($Mdn = 28.81$, $Std = 7.56$), with qualification ranging from Doctorate to no technical qualification at all. The two most common qualification levels have a Bachelor and Master degree, corresponding to 62% of the total of participants (17 participants has each qualification). Most of the participants declared to have between 5 to 8 years of experience in programming (38.89% of the participants).

The experiment consisted of downloading a zip file containing *AutoIoT*, a device *Simulator* software and a tutorial that describes the same scenario introduced in Section 3.1. This very simple scenario was chosen to make the evaluation process simpler, since some of the participants are Web developers with no expertise in IoT, a simpler use case allows them to understand and evaluate the framework better. In Section 5 is discussed briefly how to use AutoIoT in more complex IoT use cases.

The tutorial also describe the steps required to accomplish the task. The task consists of (i) using AutoIoT to generate the application source code for the given scenario, (ii) running the generated software and (iii) simulate the smart containers through the provided *Simulator* software.

After completing the tutorial, the participants were asked to answer a post-study questionnaire. The questions are presented as statements (S1-7) and are divided into two groups. The first group evaluates AutoIoT itself, and the participants should choose the answer from a five-point Likert scale ranging from *agree* to *disagree*.

- S1 - The system generated by AutoIoT cover the requirements of the scenario.
- S2 - After reading the project description file I could easily change the configuration to cover a scenario with different IoT devices.
- S3 - I would spend more than 40 working hours to manually codify the same system generated by AutoIoT.

[9]https://forms.gle/m2BBKmWJfNxiiexA6

- S4 - The code generated by AutoIoT is well organized and easy to understand.
- S5 - If I had to change the generated code and change the way that the sensor data is handled, I would know which file I should change. (Example: Instead of just store the received data in the database, inspect it and check if the "filling_status" is equal to zero, if so send an email/message to administrators informing that the container is empty).
- S6 - Use AutoIoT to generate the server application was easy.
- S7 - I could use AutoIoT to generate the server-side application in some of my IoT projects in the future.

The questionnaire was designed to evaluate whether the code generated by AutoIoT works and cover the requirements of the proposed scenario (S1). Additionally, it was evaluated whether the developers could use AutoIoT to cover different IoT scenarios, whether the generated source code was easy to understand, and whether developers were able to extend the code to fulfill further project requirements (S2, S4, and S5, respectively). One of the main benefits of using an MDE approach with code generation is to save development time, this aspect was also evaluated (S3). Finally, developers were asked if their experience using AutoIoT was easy and if they could use the framework to develop future IoT projects (S6 and S7, respectively).

The main objective of this work is to provide a tool that uses technologies that developers already know, removing the need for having a long learning phase and easing its adoption. Additionally to the evaluation of AutoIoT itself, it is important to know the expertise of the developers regarding different technologies. Therefore, the second group of questions asked how much experience the participants had regarding the following technologies.

- General Programming Language (e.g. Python, Java, C++, etc.)
- Internet of Things Development (e.g. MQTT Protocol, IoT projects, etc.)
- Domain Specific Languages (e.g. WebRatio, OCL, ThingML, etc.)
- Flow-Based Programming (e.g. Node-RED)
- Model Driven Engineering (e.g. models, system modeling, model transformations, etc.)

After gathering feedback from the participants, the evaluation of the results was conducted.

4.2 Results

Since the group of developers that participated in the experiment consists of professional Web developers and IoT researchers, it was expected that most of them show expertise in *General Programming Languages* and *Internet of Things technologies* as depicted in Figure 6. Few developers reported expertise in *Domain Specific Languages* and *Model-Driven Engineering*. This finding supports our choice of not including any of these technologies, at least not in a way visible to developers. Results also show that few developers have expertise in *Flow-Based programming*, a technology commonly employed to quickly model the behavior of a system (used by Pramudianto [14] and Node-Red, for example).

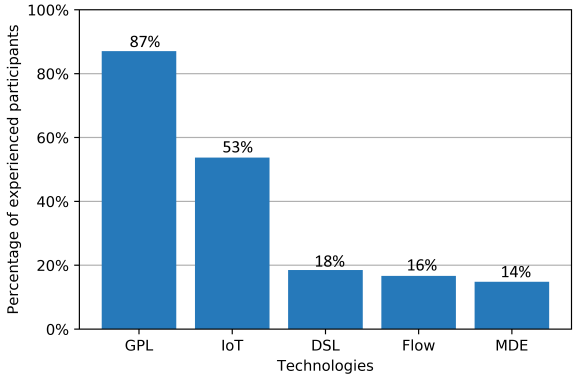

Figure 6: The percentage of developers that declared to have experience 4 or 5 (in a 1-5 scale) in a given technology.

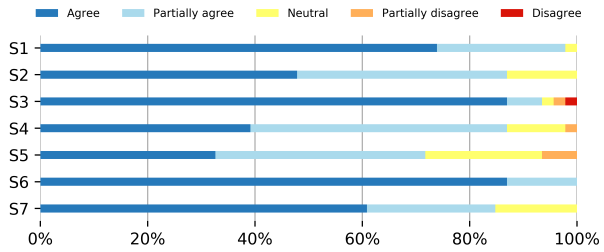

Figure 7: Questionnaire results from the participants with no expertise in MDE.

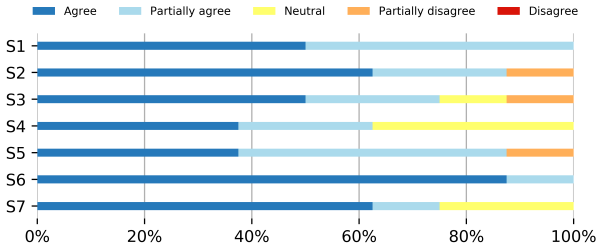

Figure 8: Questionnaire results from the participants with expertise in MDE.

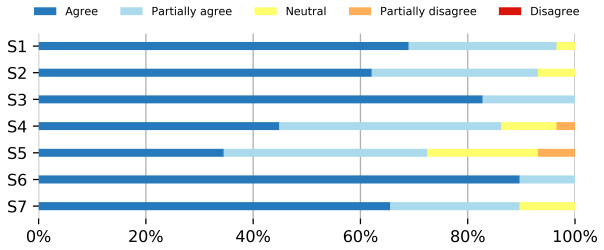

Figure 9: Questionnaire results from the participants with expertise in IoT.

To better understand the groups with different expertise, the participants were divided in three groups (with possible intersections): (i) participants with no expertise in MDE, (ii) participants with expertise in MDE and (iii) IoT experts.

The participants with no expertise in MDE represent 86% of the total and reacted positively to all aspects evaluated. 97% of the participants agreed that the system generated by AutoIoT cover the requirements of the scenario (S1) and all of them agreed that using AutoIoT to generate the server application was easy (S6), as can be seen in Figure 7. In turn, 71.73% of the participants agreed that it is easy to use AutoIoT in a different IoT scenario (S2) and 84.78% agreed that they could use AutoIoT in a future IoT project (S7). Only 4.34% and 6.52% of the participants had difficulties in understanding the generated code (S4) and changing the behavior of the generated system (S5), respectively. Finally, 93.47% of the participants agree that AutoIoT saves development time and that replicating the software produced by AutoIoT manually would cost them more than 40 hours of work (S3).

No participants of the group containing MDE experts disagreed with S1, S4, S6, and S7, but 12.5% of them disagree about the time necessary to create the same application generated by AutoIoT (S3), the easiness of change the JSON file for a different scenario (S2) and about how easy would be to change the behaviour of the generated system (S5).

In the last group (Figure 9), containing participants with expertise in IoT development, 96.55% of them agreed that the generated application cover all requirements of the scenario (S1). All of them agreed that develop the generated system manually would take more than 40 hours (S3) and that use AutoIoT was easy (S6). 93.10% of them agreed that they would easily apply AutoIoT in a scenario with different devices (S2) and 89.65% agreed that they could use AutoIoT in future IoT projects (S7). Finally, only 3.44% thinks that the generated code was not well organized (S4), and 6.89% of them were not able to extend the generated system (S5).

4.3 Discussion

Overall, all participants were satisfied with the system generated by AutoIoT and its easy(and ready)-to-use feature. The results of the evaluation show that the user-driven approach proposed by this work and the AutoIoT framework accomplish its goals, providing a tool that helps developers quickly create IoT server-side applications with no need to learn new technologies, as shown by the group with no MDE expertise (Figure 7).

The experiment also shows that the group with MDE expertise is slightly more resistant to the AutoIoT approach (as shown in Figure 8), 12.5% of them had difficulties changing the model (the JSON file) and extending the generated source code. This probably happens because they are used to other kind of model representation and it is not common, in a traditional MDE approach, to ask users to change the generated source code. Usually the user rarely update the generate code and, if a different output is desired, the models or the transformation usually need to be changed.

While working with models is natural to system designers and MDE experts, it is mainly avoided by common developers. Mostly of the participants of the experiment showed no modeling expertise and are used to work in the source code directly (Figure 6), using AutoIoT as a tool to speed up the development in the initial phase of the project. Additionally, 12.5% of the participants with MDE expertise disagree that they would take more than 40 hours of work to manually develop the system generated by AutoIoT.

It is possible that the participants were already considering tools that they know that could speed up the development process. This is a preliminary result, only 8 out of the 54 participants that participated in the experiment declared to have expertise in MDE. This is expected since MDE experts are not the target audience for this work. To better understand this group, further studies including more MDE experts are needed.

The group with expertise in IoT mostly contains no MDE experts, so its evaluation is very similar to the first group. The main difference between both groups is that IoT developers are familiar with the difficulties of developing the application to the proposed scenario. All of them agreed that they would take more than 40 hours of work to implement the application generated by AutoIoT.

This work also shows that Web and IoT developers do not have expertise in many of the technologies required to use conventional MDE-to-IoT approaches. This finding goes in accordance with [7], which found that less than 11% of a total of 3785 surveyed professional developers use any form of modeling (even sketches in a whiteboard) during the development, and most of them do not know or do not remember formal notations, like UML.

Finally, it is important to highlight that none of the developers had any assistance other than the step-by-step tutorial provided during the experiment. Personal assistance or further documentation about AutoIoT were not provided. The lack of documentation may be the reason behind some of the participants (7.40% of all participants) were not sure about how to extend the generated code (S5).

5 MORE COMPLEX USE CASES

The system generated by the *Prototype Builder* (PB) is excellent for prototyping development or scenarios that are very similar to the one presented in Section 3.1, but since it is based on a monolith architecture, it would be hard to adapt its source code to more complex IoT scenarios.

In bigger IoT projects, it is common to use a microservice architecture [17], in which the complete system is composed of multiple small independent services that communicate to each other using lightweight mechanisms. In those scenarios, AutoIoT could be used to generate some of these services, like a *Device Twin Service*, *MQTT Message Processing Service*, *Report Generator Service* or a GUI, for example. This way, the complete solution would be composed by manually written software in conjunction with the ones generated automatically by AutoIoT.

The main advantage in use AutoIoT compared to hand-written software is that AutoIoT automatically adapts to the characteristics of the devices and sensors that are going to be used in the new project. This means that changing one file (the JSON file that contains the model) AutoIoT is able to generate multiple ready-to-use microservices that can be automatically incorporated into the new project, with little-to-no change in the generated source code. Without AutoIoT, either the microservices would need to be implemented or previous source code would need to be adapted, increasing the cost of the project.

Furthermore, the microservices created by the *Specialized Builders* could also work in collaboration with other open-source projects like Node-RED, Eclipse Hono and Eclipse Ditto, for example.

6 CONCLUSION

With the growth of IoT applications, new IoT-specific technologies have also arisen. In order to avoid the overload of such technologies, many approaches have advocated the adoption of MDE with a promise of long term benefits, but most of these work achieved limited success due to the use of tools and concepts very well known in the MDE community, but less used outside of it [4, 8, 9].

This work proposed AutoIoT, a framework based on a user-driven MDE approach to develop IoT server-side applications. By using it, users can model their IoT scenarios either graphically (Web GUI) or textually (manually writing a JSON file) and, through internal M2M and M2T transformations carried by *Specialized Builder* components, the source code of the IoT application is automatically created. During the modelling developers only need to use technologies that they are used to work with (JSON representation and a general programming language), thus decreasing the learning phase and easing the framework adoption.

According to the initial evaluation, AutoIoT can successfully be used to speed up the development of IoT server-side applications and has been well received by Web and IoT developers. The participants of the evaluation reported more than 80% of satisfaction in 6 out of 7 evaluated criteria. Due to lack of documentation about the framework and the system generated by it during the experiment, only 74% of the participants reported being able to extend the generated system. This problem will be addressed in future versions.

This work shows how to design an user-driven MDE approach with very simplified modeling phase, while still giving users some freedom and control over the system to be generated. Additionally, it demonstrates that hiding MDE concepts behind technologies that users already know has a great impact on user satisfaction and adoption of the approach. Furthermore, it does not intend to replace existing MDE-to-IoT approaches. Instead, it gives developers an option to get some benefits of MDE methods without the need of learning MDE theory or using very specific MDE tools. Finally, this work does not tackle all layers of a complete IoT system development, but rather focuses only on developing the server-side application part. It does not generate device, gateway or data analysis source code in the current stage.

As future work it is planned the release of more *Specialized Builders* that will allow AutoIoT to be used to create IoT systems following the microservice architecture and tackle more complex IoT use cases. Additionally, it is intended to create a better AutoIoT ecosystem, publishing the source code of the framework at the Github and creating a website where users can share information and custom created *Specialized Builders*.

Further information about AutoIoT Framework and its related projects can be found on Github[10].

7 ACKNOWLEDGEMENT

The research was partially supported by the Bavarian State Ministry of Economic Affairs, Regional Development and Energy (StMWi) through the Center for Analytics – Data – Applications (ADA-Center) and as part of the lead project "Technologies and Solutions for Digitalized Value Creation" within the framework of „BAYERN DIGITAL II“

[10] https://github.com/AutoIoT

REFERENCES

[1] S. Abrahão, F. Bourdeleau, B. Cheng, S. Kokaly, R. Paige, H. Stöerrle, and J. Whittle. 2017. User Experience for Model-Driven Engineering: Challenges and Future Directions. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 229–236. https://doi.org/10.1109/MODELS.2017.5

[2] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. 2015. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys & Tutorials* 17, 4 (2015), 2347–2376. https://doi.org/10.1109/COMST.2015.2444095

[3] Marco Brambilla, Eric Umuhoza, and Roberto Acerbis. 2017. Model-driven development of user interfaces for IoT systems via domain-specific components and patterns. *Journal of Internet Services and Applications* 8, 1 (26 Sep 2017), 14. https://doi.org/10.1186/s13174-017-0064-1

[4] Jesús Sánchez Cuadrado, Javier Luis Cánovas Izquierdo, and Jesús García Molina. 2014. Applying model-driven engineering in small software enterprises. *Science of Computer Programming* 89 (2014), 176 – 198. https://doi.org/10.1016/j.scico.2013.04.007 Special issue on Success Stories in Model Driven Engineering.

[5] Antonella Galizia, Gabriele Zereik, Luca Roverelli, Emanuele Danovaro, Andrea Clematis, and Daniele D'Agostino. 2019. Json-GUI—A module for the dynamic generation of form-based web interfaces. *SoftwareX* 9 (2019), 28 – 34. https://doi.org/10.1016/j.softx.2018.11.007

[6] Francesco Gianni, Simone Mora, and Monica Divitini. 2018. RapIoT toolkit: Rapid prototyping of collaborative Internet of Things applications. *Future Generation Computer Systems* (2018). https://doi.org/10.1016/j.future.2018.02.030

[7] Tony Gorschek, Ewan Tempero, and Lefteris Angelis. 2014. On the use of software design models in software development practice: An empirical investigation. *Journal of Systems and Software* 95 (2014), 176 – 193. https://doi.org/10.1016/j.jss.2014.03.082

[8] A. Hamou-Lhadj, A. Gherbi, and J. Nandigam. 2009. The Impact of the Model-Driven Approach to Software Engineering on Software Engineering Education. In *2009 Sixth International Conference on Information Technology: New Generations*. 719–724. https://doi.org/10.1109/ITNG.2009.160

[9] John Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming* 89 (2014), 144 – 161. https://doi.org/10.1016/j.scico.2013.03.017 Special issue on Success Stories in Model Driven Engineering.

[10] S. Mora, F. Gianni, and M. Divitini. 2016. RapIoT Toolkit: Rapid Prototyping of Collaborative Internet of Things Applications. In *2016 International Conference on Collaboration Technologies and Systems (CTS)*. 438–445. https://doi.org/10.1109/CTS.2016.0083

[11] B. Morin, N. Harrand, and F. Fleurey. 2017. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software* 34, 1 (Jan 2017), 30–36. https://doi.org/10.1109/MS.2017.11

[12] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs. 2015. FRASAD: A framework for model-driven IoT Application Development. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. 387–392. https://doi.org/10.1109/WF-IoT.2015.7389085

[13] F. Pramudianto, M. Eisenhauer, C. A. Kamienski, D. Sadok, and E. J. Souto. 2016. Connecting the Internet of Things rapidly through a model driven approach. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. 135–140. https://doi.org/10.1109/WF-IoT.2016.7845416

[14] F. Pramudianto, C. A. Kamienski, E. Souto, F. Borelli, L. L. Gomes, D. Sadok, and M. Jarke. 2014. IoT Link: An Internet of Things Prototyping Toolkit. In *2014 IEEE 11th Intl Conf on Ubiquitous Intelligence and Computing and 2014 IEEE 11th Intl Conf on Autonomic and Trusted Computing and 2014 IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops*. 1–9. https://doi.org/10.1109/UIC-ATC-ScalCom.2014.95

[15] Alexey Savelyev and Emre Brookes. 2019. GenApp: Extensible tool for rapid generation of web and native GUI applications. *Future Generation Computer Systems* 94 (2019), 929 – 936. https://doi.org/10.1016/j.future.2017.09.069

[16] C. S. Shih, J. J. Chou, N. Reijers, and T. W. Kuo. 2016. Designing CPS/IoT applications for smart buildings and cities. *IET Cyber-Physical Systems: Theory Applications* 1, 1 (2016), 3–12. https://doi.org/10.1049/iet-cps.2016.0025

[17] K. Vandikas and V. Tsiatsis. 2016. Microservices in IoT clouds. In *2016 Cloudification of the Internet of Things (CIoT)*. 1–6. https://doi.org/10.1109/CIOT.2016.7872912