

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO CEARÁ
PRÓ-REITORIA DE PESQUISA, PÓS-GRADUAÇÃO E INOVAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Jagni Dasa Horta Bezerra

**smAR2t: a Model-driven Architecture to Interact with the Web Of
Things using Augmented Reality**

Fortaleza – CE

9 de fevereiro de 2021

Jagni Dasa Horta Bezerra

smAR2t: a Model-driven Architecture to Interact with the Web Of Things using Augmented Reality

Thesis submitted to the Postgraduate Program in Computer Science of the Federal Institute of Education, Science and Technology of Ceará as a partial requirement to obtain a Master of Computer Science degree.

Line of Research: Software Engineering

Supervisor: Cidcley Teixeira de Souza

Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)

Pró-Reitoria de Pesquisa, Pós-Graduação e Inovação (PRPI)

Programa de Pós-Graduação em Ciência da Computação (PPGCC)

Fortaleza – CE

9 de fevereiro de 2021

Dados Internacionais de Catalogação na Publicação
Instituto Federal do Ceará - IFCE
Sistema de Bibliotecas - SIBI
Ficha catalográfica elaborada pelo SIBI/IFCE, com os dados fornecidos pelo(a) autor(a)

B574s Bezerra, Jagni.
smAR2t : A Model-driven Architecture to Interact with the Web Of Things using Augmented Reality /
Jagni Bezerra. - 2020.
54 f. : il.

Dissertação (Mestrado) - Instituto Federal do Ceará, Mestrado em Ciência da Computação, Campus
Fortaleza, 2020.

Orientação: Prof. Dr. Cidcley Teixeira de Souza.

1. Internet of Things. 2. Model Driven Engineering. 3. Augmented Reality. I. Título.

CDD 004



Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)
Pró-Reitoria de Pesquisa, Pós-Graduação e Inovação (PRPI)
Programa de Pós-Graduação em Ciência da Computação (PPGCC)

Jagni Dasa Horta Bezerra

This thesis was deemed appropriate for obtaining the title of Master in Computer Science, being approved by the Coordination of the Postgraduate Program in Computer Science of the Federal Institute of Education, Science and Technology of Ceará and by the examination board:

Supervisor: Cidcley Teixeira de Souza

Instituto Federal de Educação, Ciência e Tecnologia do
Ceará (IFCE)

Atslands Rego da Rocha

Instituto Federal de Educação, Ciência e Tecnologia do
Ceará (IFCE)

Edgar Marçal de Barros Filho

Universidade Federal do Ceará (UFC)

Fortaleza – CE
9 de fevereiro de 2021

Acknowledgements

To my father, Emanuel Alves Bezerra, an example of humility I look up to and would certainly be pleased to witness this new life cycle.

To my mother, Cicera Vanluzia Horta Bezerra, for her love, encouragement and support throughout my life.

To Dr. Cidcley Teixeira de Souza, for his excellent guidance and repeatedly reassuring me of my work's importance and quality.

To the PPGCC, for the offered disciplines, financial support to publish the papers and laboratory structure.

To my girlfriend, Letícia Rodrigues Costa, for her unconditional patience, love, emotional support, text/presentation reviews and effort to understand a subject that is so unfamiliar just to help me (and she did a great job).

To Dr. Edgar Marçal de Barros Filho, in partnership with EMBRAPPII, for allowing me to work with mobile development, thus providing for my financial necessities throughout the entire course.

To fellow scholars Alyson Noronha, Cristiano Coutinho, Joel Araújo and Régis Melgaço, who helped me open my mind to exploring different technologies, concepts and techniques throughout the course.

To my friends Mirella Alves, Paulo Rogério and Yana Castro, who helped me mature and become a better man through the last ten years of friendship and gave me the necessary emotional support through this journey.

Abstract

The Internet of Things (IoT) is an extension of the internet that has a high heterogeneity of devices, cloud services and communication protocols, comprising different research areas. There are also situations where things change their locations and networks, bringing even more complexity to this scenario. This thesis aims to propose an architecture (called smAR²t) that allows the creation of standardized virtual interaction environments which can help handle these heterogeneity related issues by using: the World Wide Web Consortium's Web of Things standards to tackle interoperability; Augmented Reality as an alternate User Interface (UI) for environments with high density of things; Model Driven Engineering techniques to create a model that can turn feasible generating reactive and customizable UIs for any compliant thing at runtime. For each one of these 3 aspects, a corresponding case study was designed and developed. They detail which implementation decisions, tools, languages and techniques were used, as well as which issues were found and their respective counter-measures. Even though the case studies were not integrated into a single application, their resulting artifacts serve as input to each other, demonstrating that smAR²t can provide a foundation for developers as well as important insights for scientists, contributing to the future of IoT.

Keywords: Internet of Things. Model Driven Engineering. Augmented Reality.

List of Figures

Figure 1 – IoT Concerns, their sources and relations.	
Source: Motta, Oliveira and Travassos (2018)	1
Figure 2 – Examples of current IoT tech stacks. Source: Author	2
Figure 3 – IoT Analytics chart with the number of currently connected devices.	
Source: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/	6
Figure 4 – Improved IoT stack with WoT. Source: Author	7
Figure 5 – Architectural aspects of a Thing covered by a TD.	
Source: https://www.w3.org/TR/2020/REC-wot-architecture-20200409	7
Figure 6 – An example AR application.	
Source: https://developer.apple.com/augmented-reality/	8
Figure 7 – Degrees of Freedom.	9
Figure 8 – Popular AR apps.	
Source: https://isl.co/2018/01/augmented-reality-everything-you-need-to-know-for-2018/	9
Figure 9 – MDE Structures and how they relate.	
Source: Jácome-Guerrero, Ferreira and Corral (2017).	11
Figure 10 – Reactive principles and how they relate.	
Source: https://www.reactivemanifesto.org	13
Figure 11 – React composition examples. Source: https://devbutze.com/p/93	14
Figure 12 – Flutter’s UI formula.	
Source: https://flutter.dev/docs/	15
Figure 13 – Vue’s reactivity diagram.	
Source: https://vuejs.org/v2/guide/reactivity.html	15
Figure 14 – smAR ² t’s modules. Source: Author	17
Figure 15 – smAR ² t’s discovery module. Source: Author	17
Figure 16 – smAR ² t’s augmented interface module. Source: Author	17
Figure 17 – smAR ² t’s actuation component. Source: Author	18
Figure 18 – Case Studies overview. Source: Author	20
Figure 19 – Thing Discovery and Augmented Reality overview diagram. Source: Author	22
Figure 20 – The image used inside QRcodes in case study 1. Source: Author	23
Figure 21 – A TD for a sample lamp WT. Source: Author	24
Figure 22 – Google Maps Tiling Scheme: the first three zoom levels, tiles and num- bering. Source: Stefanakis (2017)	24
Figure 23 – Geolocation to AR Space algorithm. Source: author	25

Figure 24 – AR Space Axes. Source: https://docs.viromedia.com/docs/scenes . . .	25
Figure 25 – Model Transformations overview diagram. Source: Author	27
Figure 26 – W3C TD Metamodel. Source: Author	29
Figure 27 – Mozilla TD Metamodel. Source: Author	30
Figure 28 – ThingUI Metamodel. Source: Author	31
Figure 29 – UI and Actuation overview diagram. Source: Author	35
Figure 30 – An example Flutter widget tree. Source: http://esolution-inc.com . . .	36
Figure 31 – An example widget tree with a Provider and Consumers. Source: www.raywenderlich.com	36
Figure 32 – The widget tree of case study 3 Flutter application. Source: Author .	37
Figure 33 – UML Class Diagram of case study 3. Source: Author	38
Figure 34 – Actuation controller UML Class Diagram. Source: Author	39
Figure 35 – Form ChangeNotifier UML Class Diagram. Source: Author	44
Figure 36 – Example of input customization. Source: author.	47

List of abbreviations and acronyms

AR	Augmented Reality
API	Application Programming Interface
CMR	Context Model Representation
DoF	Degrees of Freedom
DSL	Domain-Specific Language
IFCE	Instituto Federal de Educação, Ciência e Tecnologia do Ceará
IoT	Internet of Things
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
MDE	Model-Driven Engineering
M2M	Model to Model
M2T	Model to Text
MT	Model Transformation
NASH	Núcleo Avançado de Engenharia de Software Distribuído e Sistemas Hiperídia
smAR ² t	Smart Model-driven And Reactive Augmented Reality Things
TAI	Thing Augmented Interface
TD	Thing Description
TMR	Thing Model Representation
UI	User Interface
W3C	World Wide Web Consortium
WMP	Web Mercator Projection
WoT	Web of Things
WT	Web Thing

Contents

1	INTRODUCTION	1
1.1	Related Work	4
1.2	Thesis Structure	5
1.3	Scientific production	5
2	BACKGROUND	6
2.1	Web of Things	6
2.2	Augmented Reality	8
2.3	Model-Driven Engineering	9
2.4	Reactive Programming	11
2.5	Declarative UI	13
3	SMAR²T - SMART MODEL-DRIVEN AND REACTIVE AUGMENTED REALITY THINGS	16
3.1	Architecture	16
3.2	Case Studies	18
3.3	Example of Application Flow	21
4	CASE STUDY 1: THING DISCOVERY AND AUGMENTED RE- ALITY	22
4.1	QRcodes and Image Detection	23
4.2	Obtaining a Thing Description	23
4.3	Calculating Augmentation Position with Geolocation	24
4.4	Discussion	25
5	CASE STUDY 2: MODEL TRANSFORMATIONS	27
5.1	EMF-JSON	28
5.2	W3C Thing Description Metamodel	28
5.3	Mozilla Web Thing Description Metamodel	30
5.4	ThingUI Metamodel	31
5.4.1	ThingUI	31
5.4.2	Resource	32
5.4.3	Form	32
5.4.4	Actuation	32
5.4.5	Input	33
5.5	EMF Transformation Virtual Machine	33

5.6	Discussion	34
6	CASE STUDY 3: UI AND ACTUATION	35
6.1	Flutter Composition	35
6.2	Case Overview	36
6.3	JSON Parsing	38
6.4	Communicating with the WT	39
6.4.1	Actuation Controller	40
6.4.2	ActuationRequest and ActuationResponse	41
6.4.3	Actuator	41
6.4.4	ResourceListener	42
6.5	UI Management	43
6.5.1	Dashboard Widget	43
6.5.2	Form ChangeNotifier	43
6.5.3	Form Wrapper	44
6.5.4	Input Wrapper	45
6.5.5	Input Customization	46
6.6	Discussion	47
7	CONCLUSIONS	48
7.1	Objectives Review	48
7.1.1	Objective 1	48
7.1.2	Objective 2	49
7.1.3	Objective 3	50
7.2	Contributions	50
7.3	Future Work	51
	BIBLIOGRAPHY	52
	APPENDIX	55
	APPENDIX A – EXAMPLE APPLICATION V1	56
	APPENDIX B – EXAMPLE APPLICATION V2	57
	APPENDIX C – UI AND ACTUATION VUE PROTOTYPE	58
	APPENDIX D – PROJECT REPOSITORIES	59

1 Introduction

Santos et al. (2016) describe the Internet of Things(IoT) as an extension of the internet, which provides for smart objects allowing them to be remotely controlled, communicate with each other and be accessed by users as service providers. Sharma, Shamkuwar and Singh (2018) affirm that since the term was used for the first time in 1999, it is growing more popular and viable as different technologies arise and evolve, solving past technical issues. E.g., miniaturization of components, power-efficient electronics and component cost reduction. This evolution is moving it towards larger environments and more complex applications like Smart Cities and outer space sensoring.

However, as Li, Xu and Zhao (2014) state, IoT systems are highly heterogeneous in terms of protocols, networks, and services. Each application has broadly diversified hardware configurations, Quality of Service requirements, functionalities and goals. The technical improvements also introduced concepts that increase the complexity of IoT environments like the mobility of things, allowing devices to physically move, connect themselves to different networks or even create their networks. Motta, Oliveira and Travassos (2018) show these and many other challenges, aggregating them in 14 categories of IoT concerns. These were gathered from a literature review, a qualitative study with practitioners and Government reports. They can be seen in fig. 1, along with their source and relations.

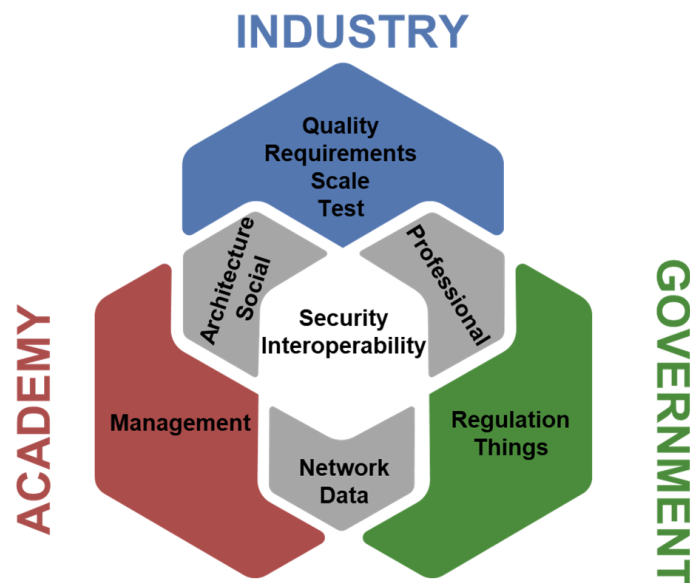


Figure 1 – IoT Concerns, their sources and relations.

Source: Motta, Oliveira and Travassos (2018)

Motta, Oliveira and Travassos (2018) also elucidate that the alignment of technology

with social areas is crucial in the IoT scenario and should provide new methods and tools. That statement has led our research to question which methods and tools could help create a standardized virtual interaction environment between users and smart objects even in heterogeneous physical environments, thus approaching 3 of the listed concerns: **Interoperability**, **Scale** and **Social**.

Interoperability is related to the challenge of making different systems, software, and things to interact for a purpose including standards and protocols. We believe that some of the issues can be addressed by applying to the Web of Things(WoT) standards and using the Thing Description(TD) formal model. The World Wide Web Consortium has been developing the Web of Things standards to address the IoT heterogeneity issues by using web protocols and the Thing Description formal model. Things that adhere to these standards are called Web Things(WT) and can expose their properties, actions and events in well known and optimized web protocols. We believe this standardization effort may also prove useful as a base to User Interface(UI) standardization.

With many devices in the same environment, the increasing **Scale** of IoT systems may raise UI challenges. To actuate upon or obtain data from smart objects and applications, users need to access a specific UI(e.g., mobile app, website). These UIs rely on navigating through a hierarchy of information (usually from general to more detailed) and are different for each application. They are heterogeneous not only visually but may also use various cloud services and tech stacks, which can be seen in Fig. 2.

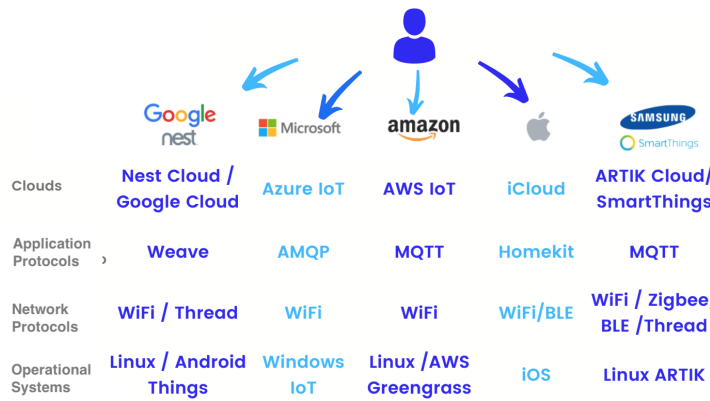


Figure 2 – Examples of current IoT tech stacks. **Source:** Author

These modern UIs usually have lists, maps or dashboards to provide an overview of available things in the user's surroundings. These work well if the user is in a context with a low density of Things and can easily label each of them (e.g., a smart home). However, in future high-density vicinities (e.g., a Smart City), these interfaces can be impractical as 2D maps may be unclear for devices that are too close to each other and labels may not be descriptive enough for similar devices (e.g., Streetlight 1, Streetlight 2).

We believe that users could benefit from using Augmented Reality(AR) as an

alternate UI in this scenario. Carmigniani and Furht (2011) define AR as a “direct or indirect real-time visualization of a physical ambient with the addition of virtual computer-generated information”. In the last few years, as smartphones have evolved in processing power and are also starting to include depth cameras, AR has gained momentum as an approach to provide a great range of interactive experiences. Norouzi et al. (2019) show in their survey that the integration of AR as an interface for IoT has proven to be very frequent in the scientific community and can provide positive results with its tangibility.

Social concerns involve understanding the situation of IoT users, their appliances and relations. We believe that Reactive UIs based on a standardized model obtained from TDs through Model-Driven Engineering(MDE) techniques could handle these ease-of-use and interactivity issues. After choosing which WT to interact with through AR, users would need access to a standardized UI. We believe that WT UIs can be adapted at runtime from a single code if they are reactive and use a model defined by us as a ThingUI. Using Model-Driven Engineering(MDE) to systematize the transformation of TDs into ThingUIs could be the way to accomplish this level of adaptation. Jácome-Guerrero, Ferreira and Corral (2017) define MDE as “a software development approach that considers the systematic use of models to develop software rather than using general-purpose programming languages”. The TD and ThingUI models would be specified with a Domain Specific Language(DSL) and TD models could be manipulated with Model Transformations(MT) to generate a ThingUI model.

This thesis general objective is to propose an architecture called smAR²t(Smart Model-driven And Reactive Augmented Reality Things). smAR²t allows the creation of a standardized virtual interaction environment between users and smart objects even in heterogeneous physical environments by integrating MDE, Reactive Programming, AR and WoT. This thesis specific objectives are:

- To discover which tools, languages and techniques are adequate for the proposed scenario;
- To demonstrate at least one example of how each smAR²t module can be implemented through descriptive case studies;
- To identify limitations and issues, describing how they can be addressed or avoided.

We developed 3 case studies, each with different architecture modules: **Thing Discovery + AR**, **Model Transformations** and **UI + Actuation**. They were successfully created and integrated into a Flutter environment, allowing users to actuate upon a sample Web Thing.

1.1 Related Work

Zachariah and Dutta (2019) discussed how an AR browsing model could be developed for IoT. They developed a proof-of-concept application where users could place an augmentation on the vicinity; Interact with it, showing a web page served on the local Wi-Fi network; Turn a lamp on or off by interacting with this web page. This work emphasized how this could be a compelling use case for AR by providing a software-based interaction that might, one day, feel intuitive, accessible and familiar as the physical interfaces we commonly encounter in our daily lives.

Schwab (2018) master thesis details the development of a gateway called Universal Explorer. It generates a RESTful API with OpenAPI documentation, a WebSocket API and a JavaScript (TypeScript) API from a W3CTD or MozillaTD. By registering TDs, any developer can configure the Universal Explorer and provide a documented and standardized bridge between Web-based client applications and Smart Objects of a vicinity. This work reinforces that the WoT can be a reasonable standard to deal with heterogeneity and can be further explored to create state-of-the-art interactions or applications.

Lan et al. (2019) propose a different resource model altogether instead of using W3C's TD. Their resource model is also ontology-based but uses RDF/XML. They also present a unified fog-based IoT access platform as a way to centralize and share resources. It is responsible for gathering and processing data from vicinity devices that are described through the resource model. By interacting with the access platform through an API, the applications can access various heterogeneous sensing devices using the unified interface.

Libert (2020) master thesis describes the implementation of an IoT middleware that uses AR as a usability amplifier. It involves adding a server to the local network (implemented in Python with the Flask framework). The server is responsible for storing TDs and providing them to a client application implemented in Javascript. This client application uses the aruco.js library to identify markers with the device's camera and generates a web page by parsing the TD, allowing users to interact with the desired thing. Even though Libert only had access to our two papers at the time, he states that the smAR2t architecture was a great inspiration. We believe this thesis's more detailed specification can also help other researchers and guide developers that want to create more complex applications.

Table 1 shows an overview of what technologies and topics each related work approached. Zachariah and Dutta affirm that the creation of an AR-based browsing architecture would primarily consist of putting the right pieces together and making concrete decisions on standards. We believe that smAR²t can fulfill that role while offering developers flexibility and customization through modularization. That is important because security strategies, communication protocols and interaction affordances can be different

for each Thing and developers need to be able to opt-in or out. Besides that, we also aim to offer developers the possibility of customizing the UI elements according to their own custom rules.

We believe that Schwab’s and Maxime contributions can be further improved by including TD metadata and using MDE instead of manual parsing. Defining a TD metamodel and transformation rules opens room for the generation of different artifacts, model validation and integration with other models. That is important because W3C’s TD may not be adhered by all IoT systems and it is not clear if legacy systems will transition to it. The ideal situation would be to have only one description model, but having the flexibility to generate UIs from different models like the one proposed by Lan et al. may prove useful while in this transitional state.

Authors	WoT	TD	Reactive	AR	MDE
Lan et al.	✓	✗	✗	✗	✓
Libert	✓	✓	✗	✓	✗
Schwab	✓	✓	✗	✗	✗
Zachariah and Dutta	✓	✗	✗	✓	✗

Table 1 – Related work overview

1.2 Thesis Structure

The next chapters are organized as follows: Chapter 2 introduces essential concepts such as IoT, Web Of Things, Model-Driven Engineering, Declarative UI and Augmented Reality; Chapter 3 describes in detail the smAR2t architecture proposal and provides an overview of the case studies which are then detailed in chapters 4, 5 and 6, each with a final section that contains a summarized discussion about the case; Finally, chapter 7 has conclusions, considerations and comments about future work.

1.3 Scientific production

- BEZERRA, Jagni Dasa Horta; DE SOUZA, Cidcley Teixeira. **smAR2t: a Models at Runtime Architecture to Interact with the Web Of Things using Augmented Reality**. In: *Insightful Ideas and Emerging Results - Simpósio Brasileiro de Engenharia de Software (SBES)*, 33. 2019, Salvador.
- BEZERRA, Jagni Dasa Horta; DE SOUZA, Cidcley Teixeira. **A model-based approach to generate reactive and customizable user interfaces for the web of things**. In: *Proceedings of the 25th Brazillian Symposium on Multimedia and the Web*. 2019, Rio de Janeiro. p. 57-60.

2 Background

2.1 Web of Things

Santos et al. (2016) describe the Internet of Things (IoT) as an extension of the internet which provides for smart objects (objects that previously would be a day-to-day item but that now have computational and communicational capacity), allowing them to be remotely controlled and accessed as service providers. An IoT application usually involves sensors that periodically gather information about its environment connected to a uniquely identified embedded system. The collected data is then processed and made available to be consumed by other smart objects, sent to cloud servers or relayed to a local network using some communication technology (e.g., WiFi, LoraWan, Bluetooth).

According to IoT Analytics, there were 7 billion smart devices connected to the internet in 2018 and this number tends to grow. However, this ascension also brought in a high level of heterogeneity in protocols, services and hardware. IoT researchers and developers are aware of the problems this heterogeneity can cause and are trying to deal with them differently.

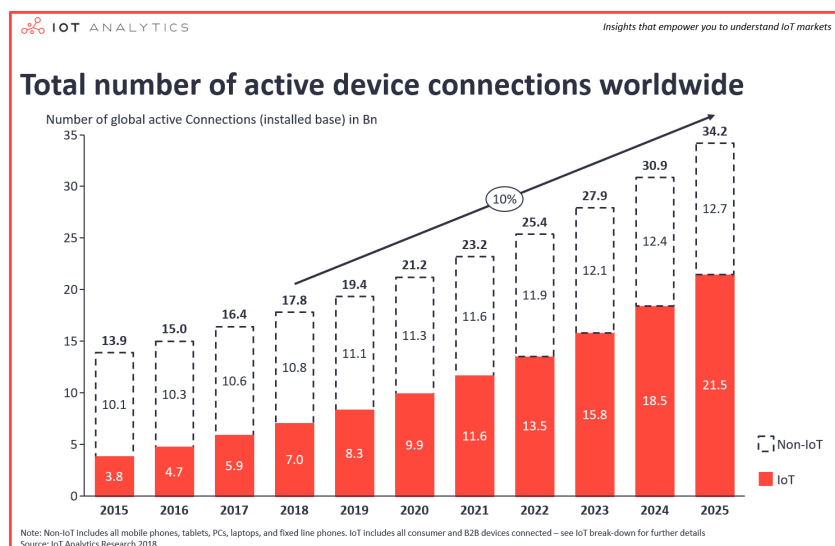


Figure 3 – IoT Analytics chart with the number of currently connected devices.

Source: <https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/>

Since 2014 the W3C has invested in standardizing and turning feasible the idea of allowing real-world objects to use existing and well-known Web standards by creating the Web of Things Working Group. Mozilla¹ describes the WoT as a unifying application

¹ <https://hacks.mozilla.org/2017/06/building-the-web-of-things/>

layer for the IoT, linking together multiple underlying IoT protocols using existing web technologies. Fig. 4 depicts how WoT aims to improve it, not by providing a new vertical stack but rather an horizontal stack above current protocols and Operational Systems.

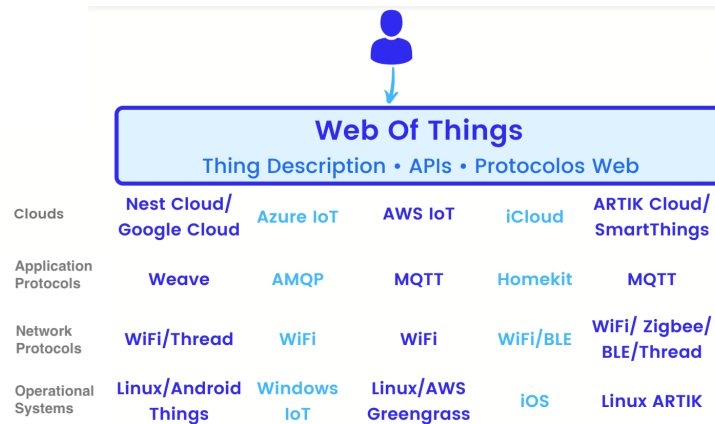


Figure 4 – Improved IoT stack with WoT. **Source:** Author

The foundation of W3C’s WoT lies upon a formal model called Thing Description. A TD is a JSON-LD encoded document that describes the Interaction Affordances, Data Schemes, Security Configuration and Protocol Bindings of a Web Thing, as seen in Fig. 5 . Thus, a TD allows a client (e.g.: an app, website, another WT, etc.) to know how to exchange information, monitor, send commands, link WT to services and require authorization to do these actions.

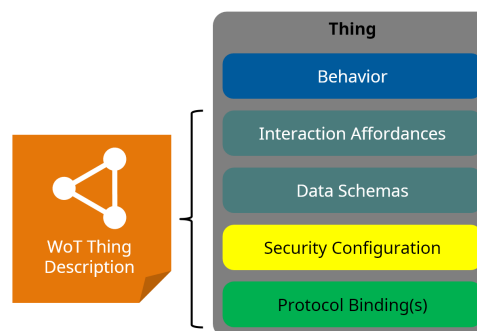


Figure 5 – Architectural aspects of a Thing covered by a TD.

Source: <https://www.w3.org/TR/2020/REC-wot-architecture-20200409>

The WoT defines that three communication methods may be used to obtain a WT’s TD or actuate upon it:

- **Directly** - Using a protocol (e.g.: Bluetooth, HTTP) to request a TD from the WT itself within its physical range.
- **By Gateway** - Gateways are network elements that implement all the communication interfaces of its IoT subnetwork and connecting itself to the Internet, acting as a middle agent that can provide TDs and relay messages to the WTs.

- **By Cloud** - Communicating with WTs by cloud usually involves interacting with a stack of services from a specific provider (e.g. Google, Amazon, Apple), but can also be more similar to a direct communication if the Wt can use IPV6, for example.

Mozilla also developed an open-source implementation of the WoT: Mozilla WebThings. It includes the WebThings Gateway and the WebThings framework, proposing a different TD data model. Mozilla's TD is formatted as a plain JSON to allow a less complicated serialization and simplify the implementation of web clients. The framework also handles W3C's API requirements like the communication with non-web communication protocols(e.g.: Bluetooth).

2.2 Augmented Reality

Carmigniani and Furht (2011) define AR as a "direct or indirect real-time visualization of a physical ambient with the addition of virtual computer-generated information". In the last few years, as smartphones have evolved in processing power and are also starting to include depth cameras, AR has gained great recognition as an engaging and practical approach for smart environments.

AR applications often rely on image and video processing techniques to give an immersive experience, detecting where an augmentation should be seen and positioned. Some methods require image markers or QRCodes, while others use the device's motion sensors to calculate how the augmentations should be integrated with the real world. There are two main native mobile frameworks to create mobile AR experiences: ARCore for Android and ARKit for iOS. These frameworks allow developers to "anchor" virtual scenes and objects; recognize surfaces and image markers; create user-shared scenes; face recognition. Fig. 6 shows an ARKit example of its new feature: people occlusion.

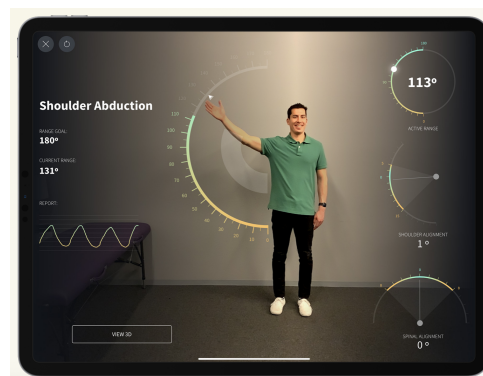


Figure 6 – An example AR application.

Source: <https://developer.apple.com/augmented-reality/>

Mobile AR experiences are shown in a 3D coordinate system where the user has

6 Degrees of Freedom (DoF), shown in Fig. 7. Degrees of freedom refers to the six basic ways a rigid object can move through 3D space. Three correspond to rotational movement around the x, y, and z axes, commonly termed pitch, yaw, and roll. The other three correspond to translational motion along those axes, which can be thought of as moving forward or backward, moving left or right, and moving up or down.

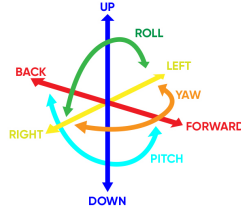


Figure 7 – Degrees of Freedom.

It is a technology that is continuously being improved through Google and Apple efforts. There are already many games, educational apps and navigation utilities that integrate this technology to provide users with more immersive experiences. Some examples can be seen in Fig. 8: Snapchat, Ikea Place, Pokémon Go and Google Maps.

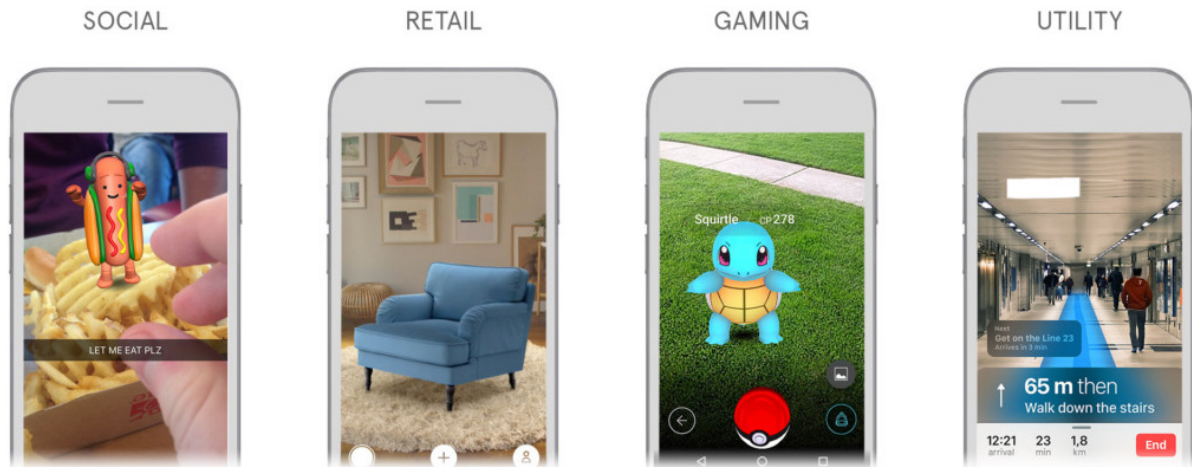


Figure 8 – Popular AR apps.

Source: <https://isl.co/2018/01/augmented-reality-everything-you-need-to-know-for-2018/>

2.3 Model-Driven Engineering

Model-Driven Engineering (MDE) is defined by Jácome-Guerrero, Ferreira and Corral (2017) as “a software development approach that considers the systematic use of models to develop software rather than using general-purpose programming languages”.

Bezin and Gerbe (2001) and Seidewitz (2003) define model as a simplified representation (or an abstract description) of a part of the world named the system. A

model is useful if it helps to gain a better understanding of the system. In an engineering context, a model is useful if it helps to decide the appropriate actions that need to be taken to reach and maintain the system's goal.

MDE applications involve the specification of a Domain Specific Language (DSL) and Model Transformations (MT) to accomplish the systematic usage of models, thus automatizing the development process entirely or partly by updating the models.

Deursen, Klint and Visser (2000) define **DSL** as “a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain”. This definition emphasizes the difference between DSLs and General Purpose Languages like C, Java or Python. As they are usually restricted to a particular problem domain, the way that notations are written are usually more abstract and thus closer to the human language in a logical way. An abstraction of the model known as metamodel is used to dictate structure and well-formedness rules of the language in which the models should be expressed.

More abstract notations can help turn development learning curves shallower, widening the range of people who can contribute to a project. For example, someone with an expertise about the context of use of the application can focus on representing real-world situations and solutions with the models rather than specifying them to a developer.

The resulting models expressed by a DSL can then be manipulated through **Model Transformations** to produce other models or to be refined. Transformations can be defined as such:

A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language. (KLEPPE, 2003)

Mens and Gorp (2006) also established a model transformation taxonomy to further detail them according to the abstraction level and number of models on both source and target. When dealing with source code as models, transformations can allow several useful operations like refactoring, optimization, formal refinement, language migration and code generation. On a broader scope, a transformation may receive a model source and produce another model (Model to Model or M2M) or produce plain text (Model to Text or M2T). Fig. 9 summarizes and relates models, DSLs and transformations.

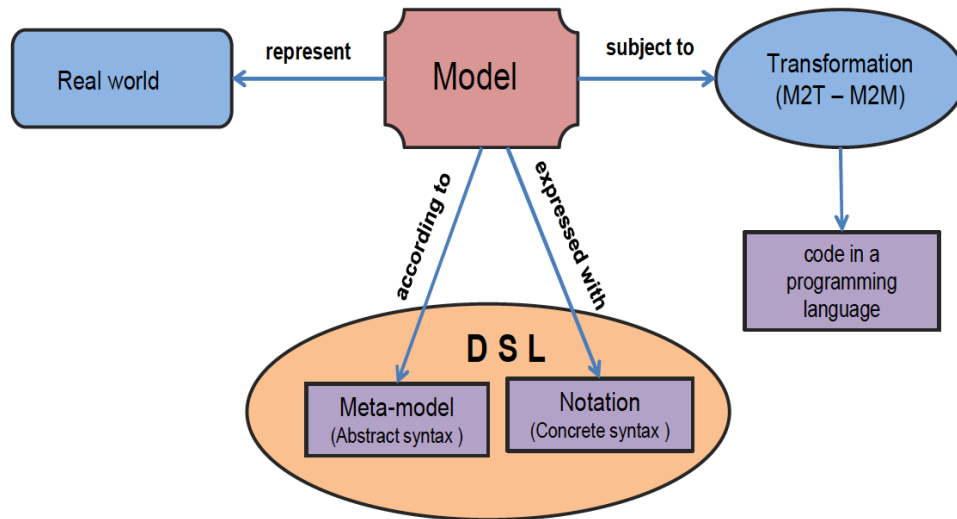


Figure 9 – MDE Structures and how they relate.

Source: Jácome-Guerrero, Ferreira and Corral (2017).

There are multiple tools, languages and frameworks that may focus on different types of transformations. The OMG’s MetaObject Facility Specification, Eclipse’s Ecore/OCL and Miso’s MetaDepth are examples of current state-of-the-art metamodeling frameworks. They have integrations with different transformation languages and programming paradigms, with ECore being the more generalist and popular one. There are also alternatives like Fouquet et al. (2012) Kevoree and the Runtime System API that focuses on supporting Models At Runtime. Models At Runtime is a related research field that investigates the use of models to represent, manage and adapt software. As Szvetits and Zdun (2016) explain, information about the running system is fed back to the software models to support reasoning and to take corrective actions at the model level. The running system then uses a self-representation based on models connected to it, transitioning to a new state.

2.4 Reactive Programming

As Kohler and Salvaneschi (2019) stated, asynchronous programming has become critical over the last years to allow different software types – web applications, mobile apps or IoT applications – to stay responsive to user interaction and environmental input. Many programming languages offer ways to achieve asynchronous features. Still, many are limited or low level, raising complexity and making code harder to maintain and comprehend. In the last years, Reactive Programming (RP) gained popularity as a practical approach to describe asynchronous programs in a declarative way. Salvaneschi’s (2014, 2017) researches show that RP increases the correctness of program comprehension while requiring neither more time to analyze it nor advanced programming skills, which may be why this approach has gained so much visibility. It can be defined as:

A programming paradigm that is built around the notion of continuous time-varying values and propagation of change. It facilitates the declarative development of event-driven applications by allowing developers to express programs in terms of what to do, and let the language automatically manage when to do it. In this paradigm, state changes are automatically and efficiently propagated across the network of dependent computations by the underlying execution model. (BAINOMUGISHA et al., 2013)

The Reactive Manifesto² is the result of contributions from various organizations to standardize Reactive development. It describes how to design Reactive systems based on four principles shown below. Fig. 10 shows an overview of these principles and how they relate.

- **Responsive** - The system responds promptly if at all possible and stays responsive to all user commands. The system should detect problems quickly and effectively deal with them by providing a consistent response even during problem times.
- **Resilient** - The system stays responsive in the face of failure. That applies not only to highly-available, mission-critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary.
- **Elastic** - The system stays responsive under varying workloads. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. That implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them.
- **Message Driven** - Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency. This boundary also provides the means to delegate failures as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

There are various libraries (e.g., ReactiveX, Bacon.js, JDK's Flow) that have been developed to allow developers to apply these principles by expressing computations

² <https://www.reactivemanifesto.org>

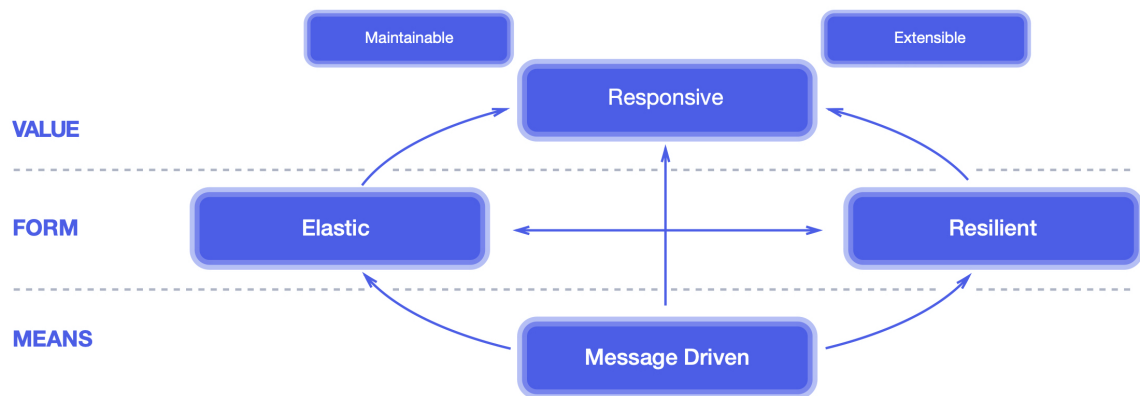


Figure 10 – Reactive principles and how they relate.

Source: <https://www.reactivemanifesto.org>

easily on event streams that can be chained and combined using high-order functional operators. They usually involve using the Observer and Iterator pattern along with functional programming concepts. There are also declarative UI frameworks that integrate reactivity in their ecosystem through their syntax, usage of components and state management strategies.

2.5 Declarative UI

As Lloyd (1994) explains, the declarative programming paradigm focuses on stating the logic of a computation without necessarily describing its control flow. This means using more abstract statements, that describe what is wanted without defining how to produce it. This paradigm has received greater notoriety in recent years as an effective way of composing UIs with a more readable syntax and simplicity. It has been adopted by a number of popular frameworks (Vue³, React⁴, Flutter⁵) and more recently by native mobile SDK's like Jetpack Compose⁶ for Android or SwiftUI⁷ for iOS.

However, the idea of Declarative UI has been around for a long time, with Microsoft's XAML⁸ being an example since 2008 and Oracle's FXML⁹ since 2011. These implementations were not so similar to modern ones but already allowed developers to compose UI through pre-defined components and focus on describing what they wanted to present instead of how. The existing UI elements could be referenced to enable interaction imperatively or connected to dynamic resources through listeners. Hanus and Kluß (2008) propositions offered improvements upon these strategies using functional programming

³ <https://vuejs.org>

⁴ <https://reactjs.org>

⁵ <https://flutter.dev>

⁶ <https://developer.android.com/jetpack/compose>

⁷ <https://developer.apple.com/xcode/swiftui/>

⁸ <https://docs.microsoft.com/dotnet/desktopwpf/fundamentals/xaml>

⁹ https://docs.oracle.com/javase/8/javafx/api/javafx/fxml/docfiles/introduction_to_fx.html

and have a significant number of similarities with current Declarative UI implementations like composition and state-driven development.

In current Declarative UI frameworks, developers define their UI through components that act like building blocks. They embrace composition over inheritance, allowing developers to create custom UI elements out of smaller ones that can be reused, combined or nested within each other, composing a tree (as seen in Fig. 11). They can share data and be customized through their state.

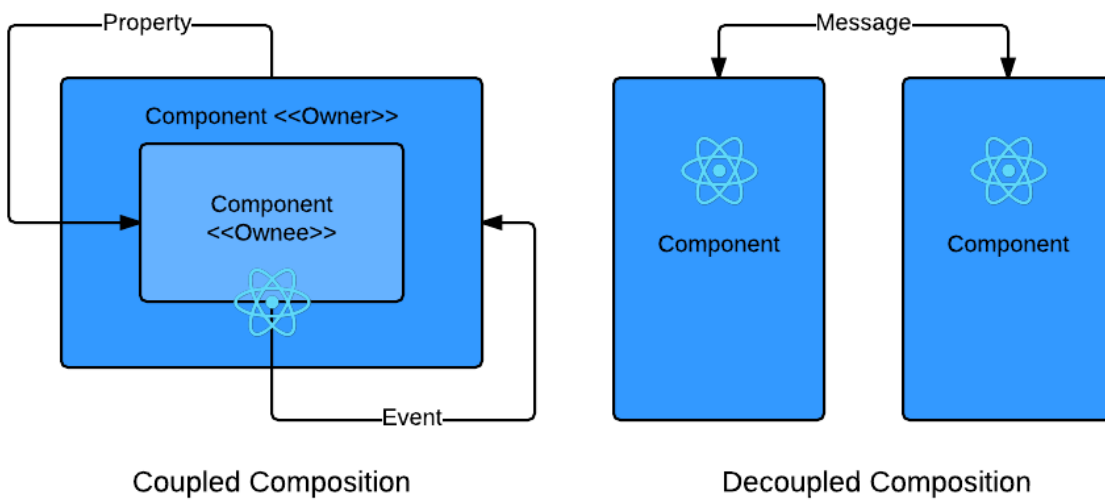


Figure 11 – React composition examples. **Source:** <https://devbutze.com/p/93>

The components are re-rendered whenever a state they relate to is updated. There are local states and the app state, with the app state being shared by different components while the local ones are singular to each of them and influence their children. Flutter's concise overview of state can be seen in Fig. 12 and an example of how Vue handles state changes can be seen in Fig. 13. Using state removes the need for referencing UI elements and manipulating their properties directly in an imperative manner. Instead, it leaves developers with the freedom to manage it as a one-source-of-truth in their preferred state management approach. For more complex applications, Flutter's documentation recommends the BLoC approach while Vue and React pick Redux. Still, developers have the freedom to choose from a range of different strategies (e.g., MVVM, Provider, MobX) despite the framework they are using.



Figure 12 – Flutter’s UI formula.

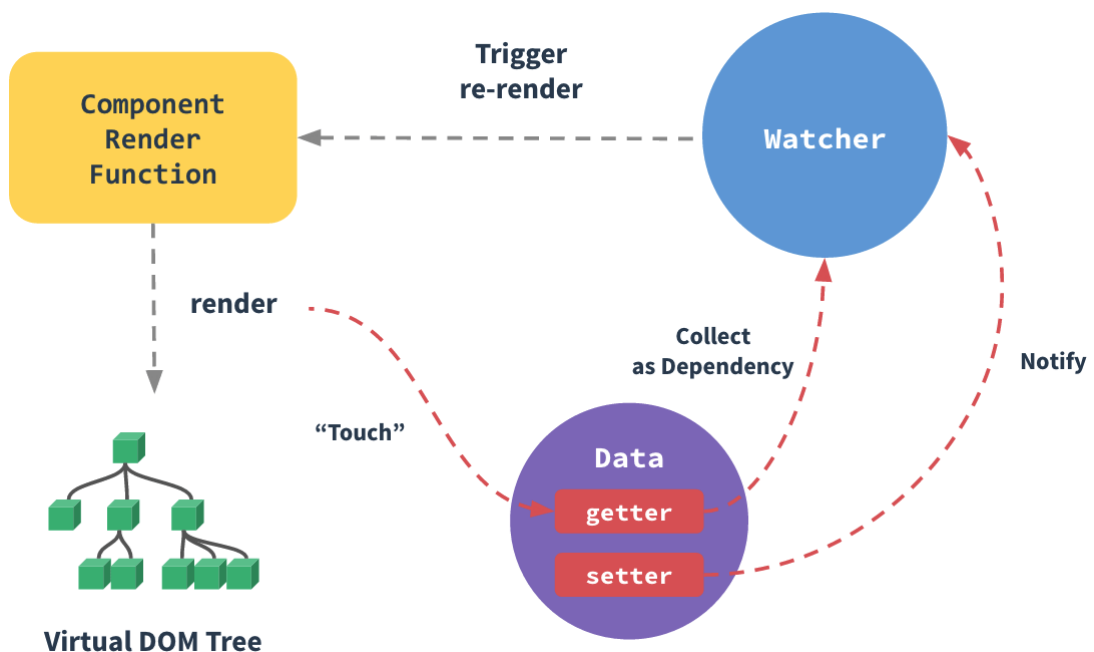
Source: <https://flutter.dev/docs/>

Figure 13 – Vue’s reactivity diagram.

Source: <https://vuejs.org/v2/guide/reactivity.html>

3 smAR²t - Smart Model-driven And Reactive Augmented Reality Things

This chapter comprises the proposed case studies and improvements mentioned in section 1.1, thoroughly describing the Smart Model-driven And Reactive Augmented Reality Things architecture with diagrams and explanations.

3.1 Architecture

The smAR²t architecture proposes a set of modules: **Thing Discovery**, **Models**, **Augmented Interface** and **Actuation**. Each module contains a set of components that can communicate, control or create each other.

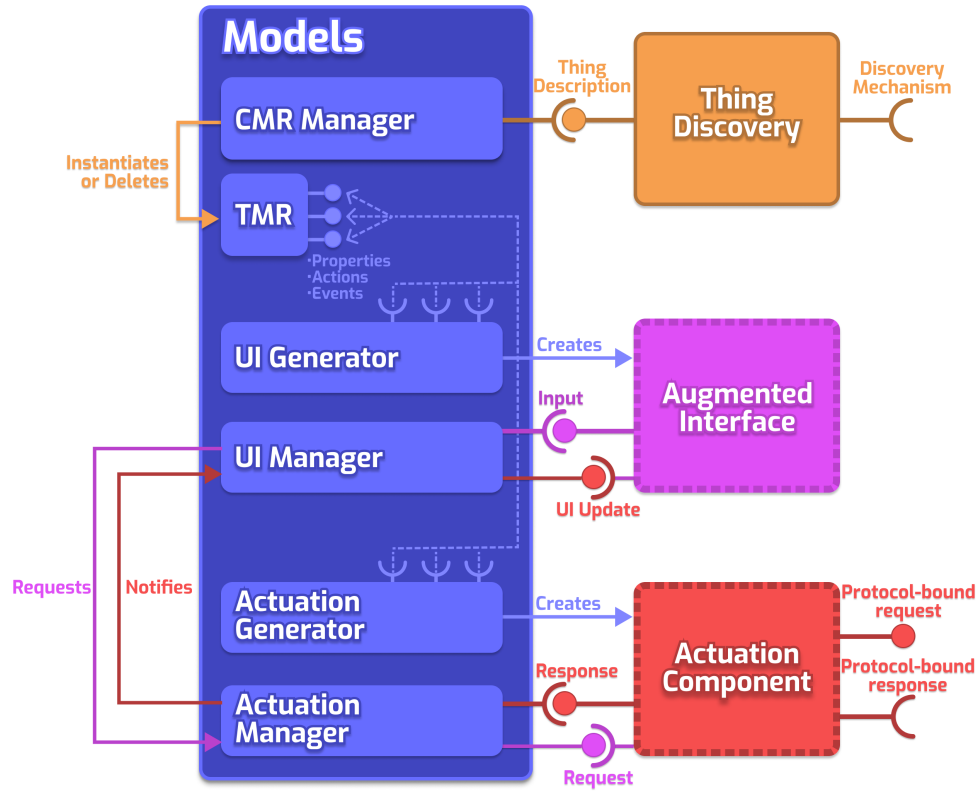
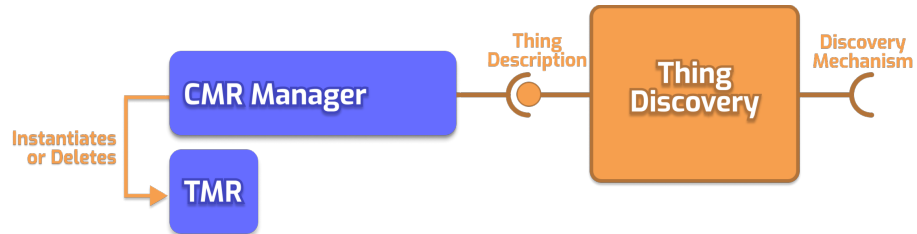
The Models module acts as a base that creates the necessary components for the Augmented Interface and Actuation modules with data received from the Thing Discovery module. That is done by managing a Context Model Representation (CMR) and a number of Thing Model Representations (TMR). Fig. 14 shows a diagram that gives an overview of its components and how they relate to other modules.

The CMR is in constant modification and represents the user's context by using these properties: geolocation, date, time of day, reachable networks and a list of available WTs. A TMR represents each WT of this list with the latest obtained values of properties and its location.

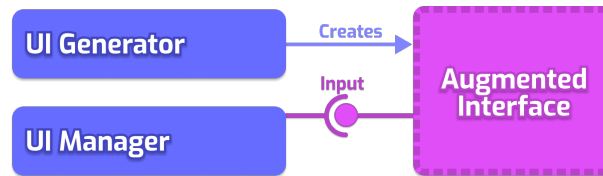
As shown in fig. 15, the Discovery module is responsible for scanning the environment, detecting when the CMR can be updated with the removal or addition of TMRs and providing each of their correspondent TDs for the CMR. This scanning can be made through network protocols (mDNS, UPnP, NSD) or physical means (QR Codes, computer vision techniques, image processing).

The Discovery and Model modules components are static, while the Augmented Interface and Actuation modules components only exist at runtime and only when the model representations exist. Changes that occur to the CMR and any TMR can be watched with reactive programming implementations and will propagate to create, modify or delete components of the Augmented Interface and Actuation modules.

The Augmented Interface module is responsible for using each TMR to create a Thing User Interface (ThingUI), updated according to its associated thing. A ThingUI also has a navigation scheme defined by the TD specifications, indicating which protocols

Figure 14 – smAR²t's modules. **Source:** AuthorFigure 15 – smAR²t's discovery module. **Source:** Author

and data schemes will be used when the user proceeds with an interaction. The navigation scheme also considers the authentication methods listed in the TD, redirecting users before allowing any further interaction. Fig. 16 illustrates the Augmented Interface module.

Figure 16 – smAR²t's augmented interface module. **Source:** Author

When users request interactions using the Augmented Interface, the Actuation component (represented in fig. 17) is responsible for creating the necessary requests, thus allowing real-world interactions within an augmented world, following the rules established

on the TD. It relies on the thing’s correspondent TMR’s information to adapt its behavior according to the necessary protocol bindings and data schemes.

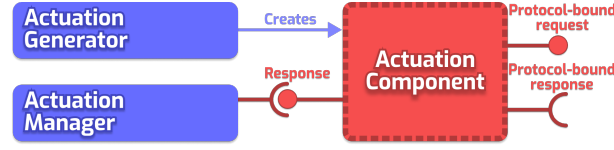


Figure 17 – smAR²t’s actuation component. **Source:** Author

3.2 Case Studies

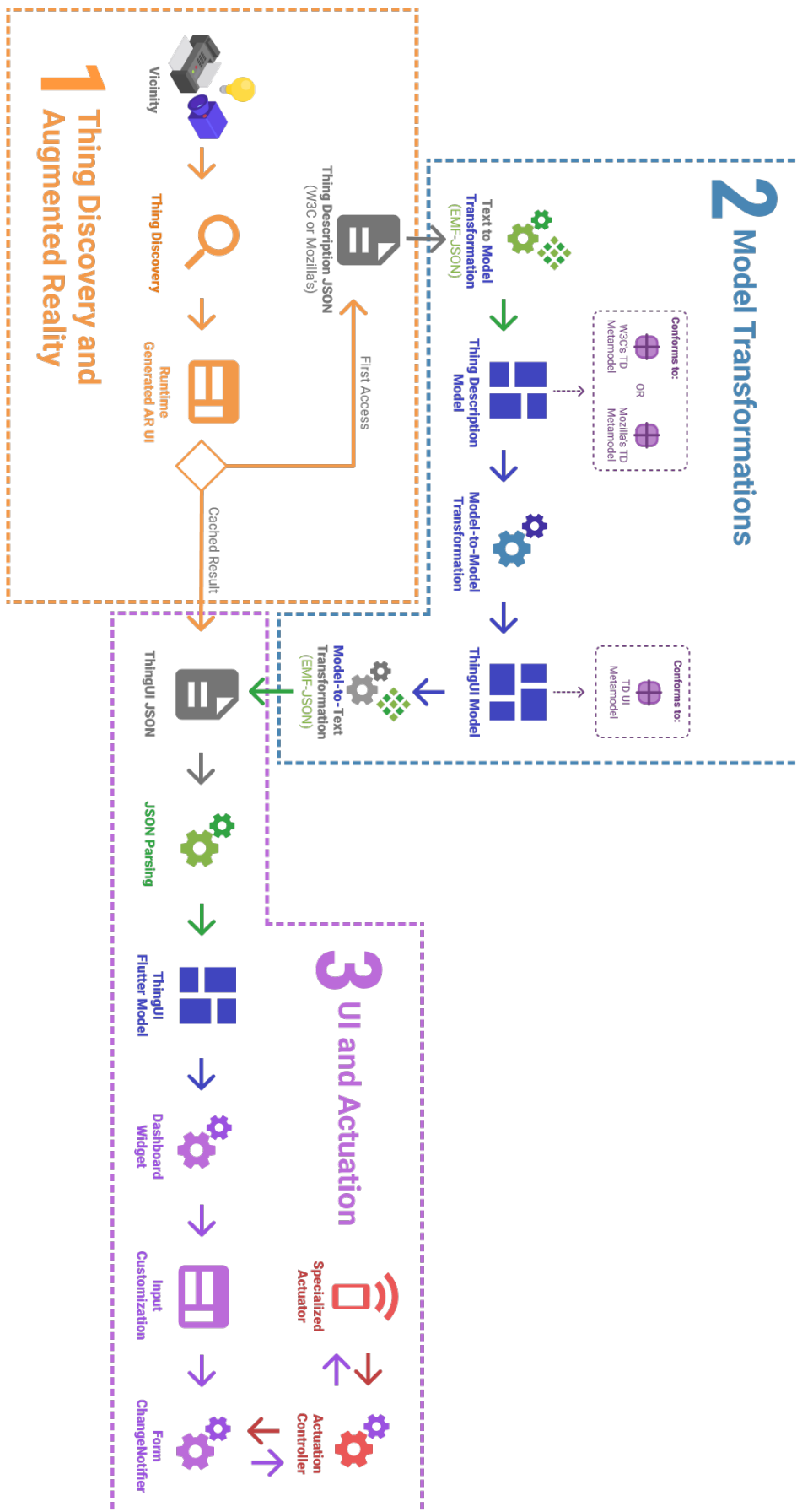
Linn and Clancy (1992) define case studies as a narrative description of the process used to solve a problem. They also affirm that these effectively communicate the complex, multi-dimensional decisions necessary for designing an optimized computer program. It offers a wide range of instructional provisions to create more effective programs, thus meeting this thesis’ objective by opening other researchers’ way to understand, explore and improve smAR²t.

We implemented three cases and all of them are part of a single smAR²t application prototype. The prototype is a Thing Browser, allowing users to explore their vicinities and interact with any WoT compliant Thing that is marked with the application’s QRCode. Fig. 18 shows an overview of all the case studies and how they relate. The implemented case studies were the following:

- **1 - Thing Discovery and Augmented Reality.** Describes how the prototype can: use AR to explore the vicinity; calculate augmentation positions; obtain a TD.
- **2 - Model Transformations.** Covers MDE aspects of the prototype: the definition of metamodels; T2M, M2M and M2T transformations of a TD to a ThingUI; execution of transformations at runtime.
- **3 - UI and Actuation.** Implementation details of TMR management, actuation procedures and user interactions.

We present the case studies in this order for more clarity as it follows the application flow. However, the implementation moved back and forth between cases as we explored different technologies and environments. For example, Case 3 was the first to be implemented and used Vue, but it was then remade with Flutter because of issues found while implementing case 1. These changes, problems and decisions are detailed in each of the following chapters, always on their last section called *Discussion*. There are also appendixes A and B that show the first drafts of the application diagram and the main changes that were made during the research. The case studies were developed from June 2019 to January 2020 on a lab called NASH (Núcleo Avançado de Engenharia de Software

Distribuído e Sistemas Hiperfídia) at IFCE (Instituto Federal de Educação, Ciência e Tecnologia do Ceará).

Figure 18 – Case Studies overview. **Source:** Author

3.3 Example of Application Flow

As it may be noticed, managing a context with a large number of Things can become complicated. There are two interface states to deal with that and save resources: **Exploration** and **Thing Focus**.

During the **Exploration** state, the user can move the device's camera and see the TMRs that are represented by anchored AR elements above any smart object discovered within the vicinity. During this state, TMRs will not be constantly updated.

If users are in a new vicinity, they must locate the QRcodes close to the WT's. Each QRcode content is a JSON indicating how to obtain the WT's TD and all of them have an image that is used to track their location on the AR space. More details are given in section 4.1.

After obtaining the TD (more details in section 4.2), it is registered in the CMR Manager along with its unique identifier and the user's current location (latitude and longitude). An augmentation is then positioned on top of the QRcode.

If a QRcode of a previously registered WT is scanned again, the CMR Manager updates the correspondent registry. That can be important to some WT's because: they can be mobile, thus their location must be corrected; they may have received remote updates, changing their TD; their TD is obtained directly and the WT has connected to a different network, changing its base URL.

When the user is in a vicinity with registered WT's, each of their registered geolocations is used to determine where the augmentations should be positioned without needing to find the Augmented Images again (see section 4.3 for more details). The users can also refine their experience if the sensor accuracy is low by only showing augmentations anchored to Augmented Images.

When the user taps an augmentation, the CMR Manager checks if the TD has already been converted to a ThingUI and cached. If not, the TD is sent to an MT service to create a ThingUI (as seen in chapter 5). If so, the state is changed to **Thing Focus**.

In this state, a customized generated interface is shown and the user can actuate upon the WT (as seen in chapter 6). The corresponding TMR protocol bindings will be established to continuously watch for updates, rebuilding the relevant UI elements whenever a change occurs. When the user finishes interacting with a WT, smAR²t goes back to the exploration state and disables automatic updates for the previously focused TMR.

4 Case Study 1: Thing Discovery and Augmented Reality

This case study comprises an example application flow that guides users to explore their vicinity, obtain TDs and choose which one they want to interact with. As shown in Fig. 19, there are two possible outputs: a Thing Description or a Thing UI, leading to case 2 or 3, respectively.

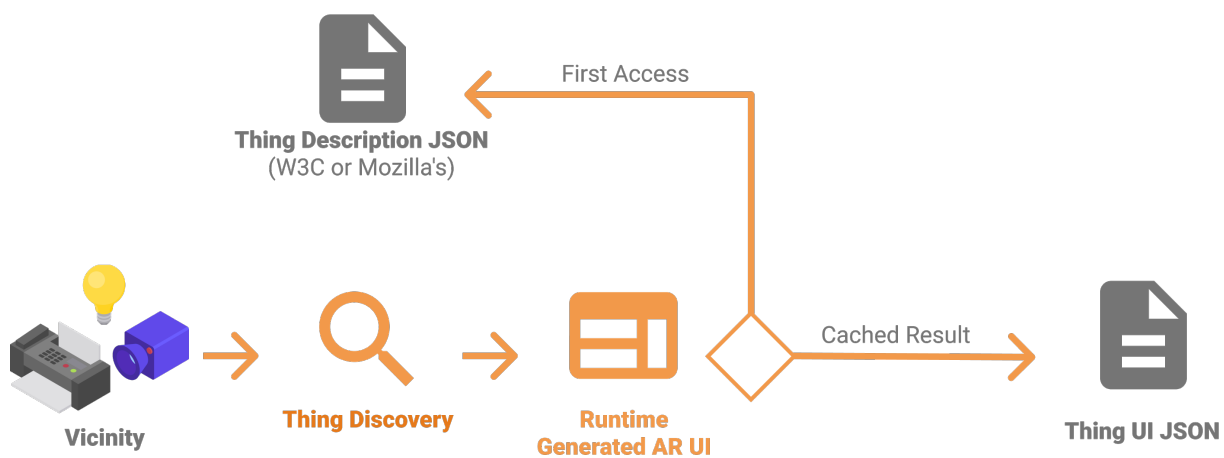


Figure 19 – Thing Discovery and Augmented Reality overview diagram. **Source:** Author

For this AR experience to be successful, we needed a way to define where each augmentation should be positioned in relation to the user so they would be at least close to their location in the real world. Smartphones have sensors to determine a user's geolocation and compass orientation, but the accuracy is usually lower around buildings and electronics.

As Szot et al. (2019) have tested, modern smartphones' location accuracy can vary from 1 to 14m. Novakova and Pavlis (2017) tested Android smartphone compasses while Allmendinger, Siron and Scott (2017) tested iOS compasses. Both iOS and Android optimal devices achieved an error margin of 50. Still, Novakova also found that a significant number of their Android devices were simply unusable because they would reach 800 of discrepancy.

In our context, an accuracy considered low for vicinity interaction may be acceptable if the user wants to interact with WTs from afar (like controlling something at their houses from work). Having in mind these variations and inaccuracies we decided to use image detection, calculate augmentation positions with Web Mercator Projection (MP) and allow the user to choose which method to use.

4.1 QR Codes and Image Detection

ARCore and ARKit have image detection solutions that can respond to 2D images in the user's environment. After an augmented image begins to be tracked, the frameworks provide continuously refined estimates for position and orientation. These estimates have enough data to allow an augmentation to be anchored in the physical environment and even maintain their location after the image cannot be seen anymore. In this case study, we tracked fig. 20, positioned inside the QR Codes.



Figure 20 – The image used inside QR Codes in case study 1. **Source:** Author

To work with any image size and infer their physical size, the user must get close enough to fill at least 70% of the device's screen width or height. After the image is detected, the user is prompted to move away in order to scan the QR Code and know which obtaining method, protocol and identifier should be used to get the TD.

4.2 Obtaining a Thing Description

In this case study, the TD is obtained directly from an ESP32 development board. We used Mozilla's webthing-arduino¹⁰ API to create a local web server and generate a TD, which can be seen in fig. 21. This way, devices on the same network can obtain the TD through an HTTP GET call, either using the local IP address (webserver) or the configured hostname (mDNS). It is a sample Webthing that has three settable properties called: *on*, *level* and *limitTime*. Sending PUT HTTP requests with a valid JSON WoT notation to the *on* href toggles the board's built-in led, emitting a serial log. The other properties are mockups that only emit serial logs.

¹⁰ <https://github.com/mozilla-iot/webthing-arduino>

```

{
  "name": "Built-in LED",
  "href": "/things/led", "@context": "https://iot.mozilla.org/schemas",
  "@type": [ "OnOffSwitch", "Light" ],
  "properties": {
    "on": {
      "type": "boolean", "@type": "OnOffProperty",
      "href": "/things/led/properties/on"
    },
    "level": {
      "type": "number", "@type": "BrightnessProperty",
      "min": 0, "max": 100,
      "href": "/things/led/properties/level"
    },
    "limitTime": {
      "type": "number", "@type": "LimitTimeProperty",
      "href": "/things/led/properties/level"
    }
  }
}

```

Figure 21 – A TD for a sample lamp WT. **Source:** Author

4.3 Calculating Augmentation Position with Geolocation

As Stefanakis (2017) explains, the Mercator is a cylindrical projection of Earth widely used by map provider applications because of Google's adaptation: the Web Mercator Projection (WMP). It uses the spherical formulas at all scales, whereas conventional large-scale Mercator maps typically use the ellipsoidal form of the projection.

WMP's world coordinates are represented with X and Y axes, adjusted such that the upper left corner is (0, 0) and the lower right corner is (256, 256). Fig. 22 shows Google's strategy of tiling and zooming factor to navigate from large to small scales.

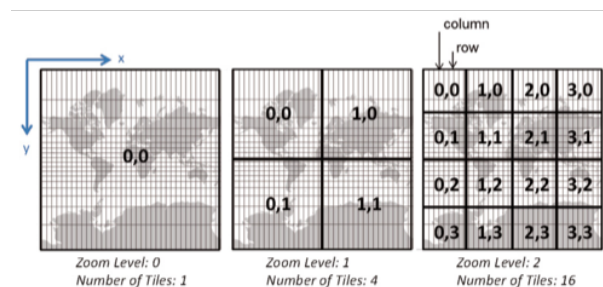


Figure 22 – Google Maps Tiling Scheme: the first three zoom levels, tiles and numbering. **Source:** Stefanakis (2017)

Fig. 23 gives an overview of how the AR space position of an augmentation can be calculated based on the user's current location.

If we place a user facing North and intersect the AR space axes (shown in fig. 24) with the WMP axes, AR's X corresponds to WMP's X and AR's Z corresponds to WMP's Y. This happens because the Mercator projection's Grid North coincides with True North.

```

alignZAxisToTrueNorth()
userMercatorPosition ← latLngToMercator(userLat, userLng)
wtMercatorPosition ← latLngToMercator(wtLat, wtLng)

xPosition ← wtMercatorPosition.x - userMercatorPosition.x
yPosition ← 0
zPosition ← wtMercatorPosition.y - userMercatorPosition.y

showAugmentation(xPosition, yPosition, zPosition)

```

Figure 23 – Geolocation to AR Space algorithm. **Source:** author

AR's Y-axis would represent altitude, but we did not consider it in our calculations and always present augmentations to users on the Y value that they started the AR session.

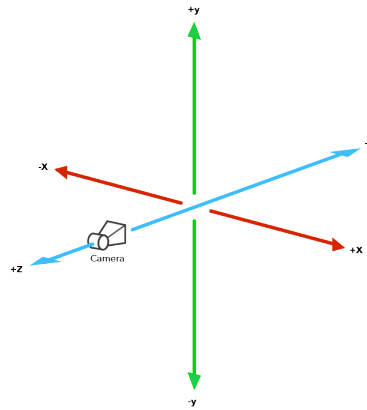


Figure 24 – AR Space Axes. **Source:** <https://docs.viromedia.com/docs/scenes>

4.4 Discussion

At the first stages of implementation, the WebXR framework was tested, aiming to support the Web environment because, at the time, case 3's first prototype was implemented with Vue. However, WebXR was very unstable and only worked on a specific build of Chrome (Canary 72.0.3592.2) because ARCore support was removed after this version.

Because of that we decided to migrate to another reactive programming framework called Flutter¹¹. Google's UI toolkit for building natively compiled mobile applications (web and desktop in beta) from a single codebase. While developing this case study we made contributions to an open-source project called *ARCore_Flutter_Plugin*¹² in a Github fork¹³. For now, the AR experience only runs on Android devices but can be made available for iOS if the same contributions are integrated to the *ARKit_Flutter_Plugin*¹⁴. Supporting

¹¹ <https://flutter.dev>

¹² https://github.com/giandifra/arcore_flutter_plugin

¹³ https://github.com/smar2t/arcore_flutter_plugin

¹⁴ https://github.com/olexale/arkit_flutter_plugin

these projects can also open the way to a unified ARCore, ARKit and WebXR plugin solution.

To scan QR Codes, we implemented a way to capture ArFrames from the ArSession while maintaining it active. The received frames are then converted to an image file analyzed by FirebaseVision, obtaining the QR Code content.

To store augmentation locations, we also added the feature of obtaining the world coordinates of Augmented Images when they are found or updated. Besides the location data, this also allows further control over augmentations. They can be positioned and transformed relatively instead of restricting them to the Augmented Image's exact location and angle.

We successfully obtained a TD from an ESP32 development board that used Mozilla's webthing-arduino API. A QR Code is positioned by the WTs side containing its mDNS hostname. The `mdns_plugin`¹⁵ package is then used to lookup and resolve it, obtaining the IP and port the request must be sent to. Through an HTTP GET request, the TD is received and can now be used to generate a ThingUI.

The `localstorage`¹⁶ package was used to persist a map of TMRs in a local JSON database. The CMR successfully managed to filter which TMRs were in the vicinity by making distance comparisons between their geolocations. However, the augmentation positioning strategy proved to be imprecise for managing close quarters, as we expected. It can be adequate for a general overview and to filter WTs in the vicinity but is not reliable if there is a need to precisely position a WT's augmentation. Because of that, we offered users the option to only show WTs that they scan on a single session, placing augmentations over their QR Code.

¹⁵ https://pub.flutter-io.cn/packages/mdns_plugin

¹⁶ <https://pub.flutter-io.cn/packages/localstorage>

5 Case Study 2: Model Transformations

This case study details an example MDE application that creates ThingUIs from TDs using the Eclipse Modeling Framework (EMF). Eclipse's documentation¹⁷ shows that EMF provides tools and runtime support to produce a set of Java classes from a model specification described in XML. There is also a set of adapter classes that enable viewing and command-based editing of the model. EMF includes a metamodel called Ecore for describing models. With it, we defined three metamodels: W3C Thing Description, Mozilla Web Thing Description and ThingUI.

The EMF-JSON Jackson framework¹⁸ was used to make the T2M and M2T transformations (JSON to TD or MozillaTD model; ThingUI model to JSON). The EMF Transformation Virtual Machine (EMFTVM) was used to execute M2M runtime transformations with the ATL transformation language¹⁹ (TD or MozillaTD model to ThingUI model). Fig. 25 illustrates an overview of the necessary steps to transform a TD JSON into a ThingUI JSON. Each step of transformations will be detailed in the following subsections.

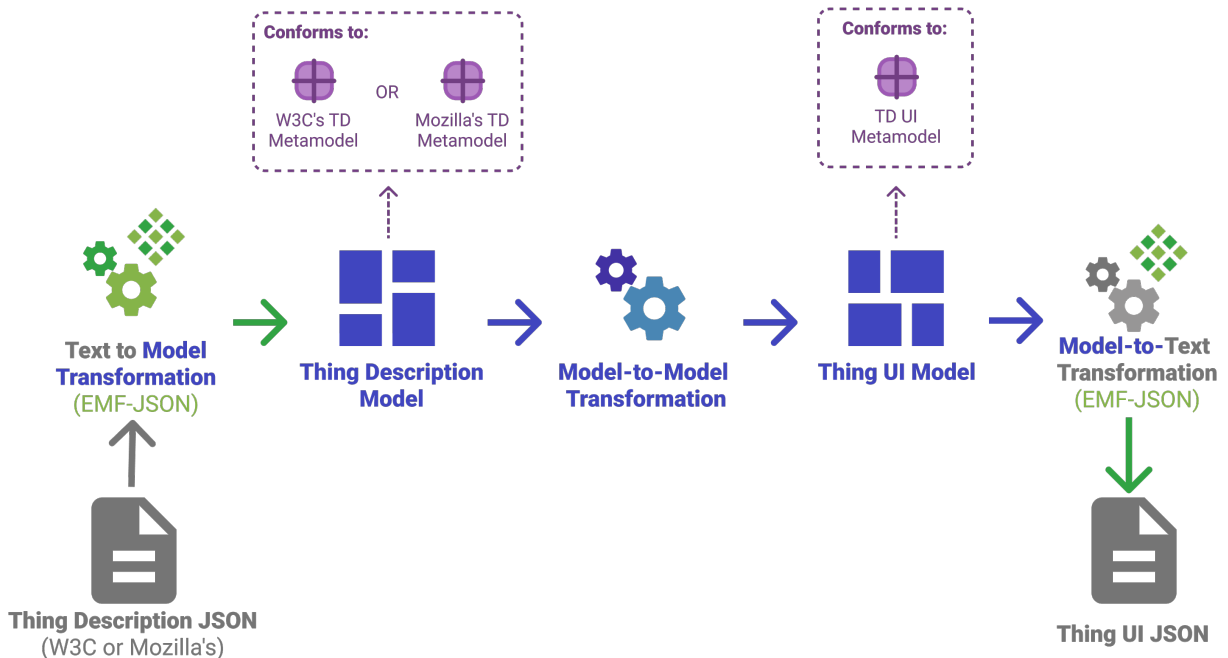


Figure 25 – Model Transformations overview diagram. **Source:** Author

¹⁷ <https://www.eclipse.org/modeling/emf/>

¹⁸ <https://github.com/emfjson/emfjson-jackson>

¹⁹ <https://www.eclipse.org/atl/>

5.1 EMF-JSON

The EMF-JSON is an umbrella of projects that aim to provide support for the JSON format and associated technologies to EMF as an alternative to XMI. We use one of its subprojects called EMF-JSON Jackson (a high-performance JSON processor for Java) to serialize and deserialize our EMF models as JSONs.

Our theory on this study case was that by defining W3C and Mozilla TD metamodels thoroughly and correctly, their serialization with EMF-JSON Jackson would match an actual TD. That also means a model could be instantiated directly by deserializing a TD. After defining the metamodels and fixing found issues, this theory was confirmed with the example TDs provided by Mozilla²⁰ and partially successful with TDs provided by W3C²¹. Models can be instantiated directly and generate the same base TDs back after being serialized. However, W3C's TD Context Extensions imported name-value pairs are not supported and are then omitted (explained in Subsection 5.2).

TDs are validated with W3C's JSON schema²² to determine which metamodel they conform to. Mozilla TDs do not match this schema and thus, the Mozilla metamodel is used if the validation fails. If a Mozilla TD model cannot be created, then the TD is considered invalid.

5.2 W3C Thing Description Metamodel

The W3C TD metamodel can be seen in Fig. 26 and was created according to the TD Information Model definitions²³. The TD Information Model is built upon the following independent Vocabularies:

- **Core TD:** which reflects the Interaction Model with the Properties, Actions, and Events Interaction Affordances.
- **Data Schema:** includes (a subset of) the terms defined by JSON Schema.
- **WoT Security:** identifies security mechanisms and requirements for their configuration.
- **Hypermedia Controls:** encodes the main principles of RESTful communication using Web links and forms.

It is important to notice that every relationship between classes in the metamodel is a containment. That was done to ensure that all of them are nested when serialized by EMF-JSON Jackson.

²⁰ <https://iot.mozilla.org/wot/#web-thing-description>

²¹ <https://www.w3.org/TR/wot-thing-description/#example-serialization>

²² <https://www.w3.org/TR/wot-thing-description/#json-schema-for-validation>

²³ <https://www.w3.org/TR/wot-thing-description/#sec-vocabulary-definition>



to that type's schema's constraints. That may include certain types of property, action and event the device can support. As the Attribute class can represent every extension name-value pair, every Mozilla TD can be defined with this metamodel.

It was more thoroughly explored than the W3C Metamodel because case study 1 uses a Mozilla WT(can be seen in Chapter 4, section 4.4). Because of that, we made more tests, assuring that serialization and deserialization work correctly with any valid Mozilla Web TD. We created the ThingUI, keeping in mind this metamodel's specifications to include every feature of it.

5.4 ThingUI Metamodel

The ThingUI metamodel can be seen in 28. We designed it with default suggested inputs for each type of data and including a type property on Thing, Input and Resource classes to allow customization. The main classes and their relationships are described in the following subsections.

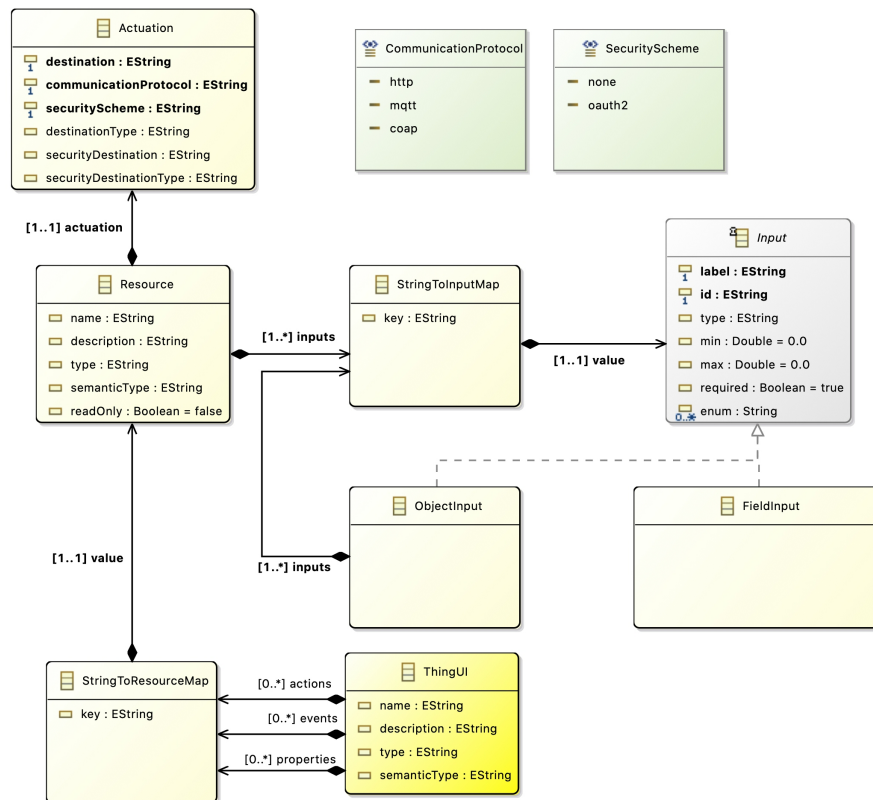


Figure 28 – ThingUI Metamodel. **Source:** Author

5.4.1 ThingUI

The ThingUI is the main class that represents the Thing itself. It is the JSON root and has maps for *properties*, *actions* and *events*, represented by the Resource class. It

includes properties found both on W3C and Mozilla TDs:

Title: A string shown in the augmentation and across multiple screens while the user is interacting with the WT; **Description:** A string optionally displayed in the augmentation and the WT introduction screen, usually containing a summary of the WT capabilities; **Type:** Corresponds to *semanticType* of W3C and Mozilla metamodels Thing classes. Can be used to determine which icon can represent the WT and indicate to the user which interactions they can expect; **PropertyOrder, ActionOrder and EventOrder:** Arrays that contain the keys of their correspondent Resource maps. That allows users to customize their own UIs by reordering the elements of their interface.

5.4.2 Resource

The Resource class represents properties, actions and events of a Thing. All resources contain a Form that indicates whether they can be edited or interacted with. It has these properties:

Title: A string that should be shown with emphasis within the Inputs of this resource; **Description:** A string that usually contains instructions about the interaction and what the resource represents; **Type:** Indicates whether this Resource is a property, an action or an event; **SemanticType:** Corresponds to *semanticType* of the Mozilla metamodel Property, Action and Event classes. It can indicate which Input elements should be shown for this resource and provide developers margin for customization.

5.4.3 Form

A Form is a container for one or more Inputs that will be shown for a Resource and contains an Actuation when needed. It includes the following properties:

ReadOnly: If this property is true, it indicates this form's Inputs must not be editable and that interactions should not actuate upon the WT. If this property is false, then the form Inputs must be editable and it must contain an Actuation; **ShowsSubmitButton:** When this property is false, it indicates that every edition of an Input in this form will trigger an actuation. When true, it states that the actuation is only triggered after confirming with a specific button.

5.4.4 Actuation

An Actuation contains the necessary data to perform an actuation upon a WT Resource. It has these properties:

CommunicationProtocol: A string that indicates which protocol must be used to perform the actuation, (e.g.: HTTP, MQTT, COAP); **Destination:** A string that shows

where the protocol calls must be directed; **DestinationType**: A string that indicates if the *destination* property is a URL, IP Address, BLE Address, etc; **SecurityScheme**: A string indicating which Security Scheme is used by the TD; **SecurityDestination**: A string that indicates where the security protocol calls must be directed during setup; **SecurityDestinationType**: A string that indicates if the *securityDestination* property is a URL, IP Address, BLE Address

5.4.5 Input

Each Input of a Form represents an interface element that allows users to check the current status of a Resource and possibly interact with them. Input is an interface which **ObjectInput**, **TextInput**, **NumberInput** and **BoolInput** implement. These are the Input properties:

Id: The key this Input is paired within the Form’s map of Inputs; **Label**: Optional string shown above the Input; **Placeholder**: Optional string shown to the user when the input value is empty; **EditingValue**: The Resource value that the user interacts with. That is the value that will be sent when an actuation must be made. If the Form’s *readOnly* property is true, this value will always be equal to *thingValue*; **ThingValue**: The most recent value of the associated Resource received from the WT; **Type**: An optional string that can be used to perform customizations on a specific input. Usually associated with the Resource’s *semanticType*.

TextInputs, **NumberInputs** and **BoolInputs** have their own *inputType*. This property is used to indicate which default input should be shown to the user. **TextInputTypes**: plain, email, multiline, phone and datetime. **NumberInputTypes**: dial, slider, keyboard and stepper. **BoolInputTypes**: switch, checkbox and button.

NumberInputs additionally have the *minimum* and *maximum* properties used to control a range of values the user can choose. **ObjectInputs** contain a map of Inputs that represent each of the object’s properties.

5.5 EMF Transformation Virtual Machine

Wagelaar et al. (2011) created The EMFTVM. As they explain, it is a stack-based virtual machine that aims to provide an execution semantics standard for implementing rule-based MT languages. Existing MT languages that target the EMFTVM include ATL and SimpleGT.

We used Victor Guana’s ATLauncher project²⁴ as a reference to setup the ATL + EMFTVM environment. We created the W3CMapper, MozillaMapper and ThingUIMapper

²⁴ <https://github.com/guana/ATLauncher>

Java classes to configure serialization and deserialization of each model. They also configure EMFTVM with the necessary input model, output model and transformation ruleset specifications during each step.

We designed two sets of ATL rules to transform W3CTD and MozillaWTD models into ThingUI models. These rules are transformed into the EMFTVM semantics and executed at runtime, resulting in a ThingUI model that can be serialized into a JSON. This JSON is sent as a response to the client application, which can now use it to present a UI to the user and let them interact with the WT (as shown in Chapter 6).

5.6 Discussion

In the first version of this study case, the approach involved storing model resource files on the server (XMI or JSON), monitoring model changes with a framework like Viatra²⁵. Clients would access the models through a REST API when the transformation was done using the EMF-REST library²⁶. While that is a valid approach, we decided to use EMFTVM after discovering it in our bibliographical research and creating some toy applications.

We used the EMFTVM approach mainly to provide developers with greater flexibility on what MT Language to use and simplify the server tasks by executing the transformations at runtime. This flexibility is provided because EMFTVM maps MT languages to the same implementation. Therefore, it has an executable semantics standard that allows developers to use their preferred approach with any compatible MT Language. The runtime execution of transformations simplifies server tasks because the models can be stored within memory, formatted and then sent to the client. This way, the server does not need to store, manage and distribute model resource files to users.

We would also implement JSON parsers to deserialize TDs into models and ThingUI models to JSON had we not found the EMF-JSON project. The TD metamodels we created successfully represented W3C TDs that did not have context extensions and Mozilla TDs. We successfully coordinated all of the following processes through a single execution: parsing the JSON obtained in case 1 directly to a TD model using EMF-JSON; transforming this model into a ThingUi model through an ATL runtime environment; parsing this ThingUi model into a JSON using EMF-JSON.

²⁵ <https://www.eclipse.org/viatra/>

²⁶ <https://somresearch.uoc.edu/tools/emfrest/>

6 Case Study 3: UI and Actuation

This case study details an example Flutter application that can interact with a WT according to the specifications of a ThingUI JSON.

The JSON is parsed into a ThingUI model used by the application to define its behavior and presentation. We designed it to allow developers to use customized widgets instead of the default ones if they wish so.

The user interactions trigger events to a FormChangeNotifier, that relays requests to an Actuation Controller, responsible for validating and propagating these requests to an Actuator. According to its specified communication protocol and security scheme, the ThingUI's Actuation object defines which specialized actuator will be instantiated.

These Actuators are used to format protocol-bound messages that effectively perform an actuation upon the WT according to user interactions. ResourceListeners also use them to update the associated TMR in realtime according to events that are received from the WT.

The actuation response and the most recent TMR go back through the Actuation Controller to the FormChangeNotifier, which triggers any necessary UI feedback and updates. That allows users to know if their actuation was successful and to watch a WT status in realtime.

Fig. 29 illustrates a general overview of this case study. A brief introduction to how Flutter approaches composition is presented and then each component will be detailed in the following sections.

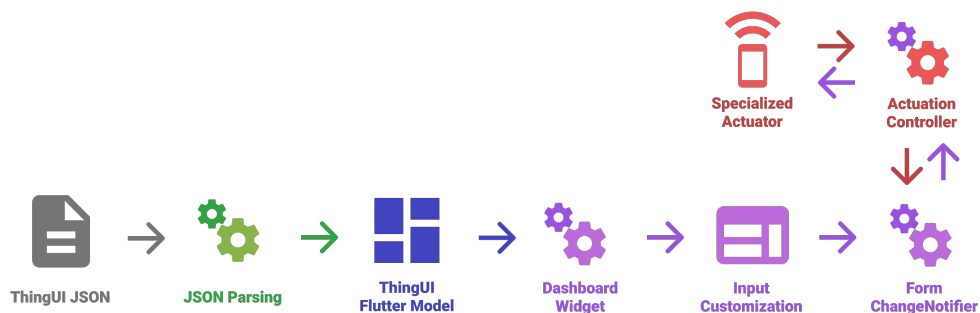


Figure 29 – UI and Actuation overview diagram. **Source:** Author

6.1 Flutter Composition

In Flutter, each UI component is called a widget, which may have a state and a number of children widgets. A widget state is called ephemeral and should only contain

data that is exclusively relevant to the UI, rebuilt with every state change. Opposed to that, there is the app state, which contains data shared across many parts of the application and may be stored between user sessions.

Managing the app state can be a complicated task and there are many different approaches. In this case study, we used Providers to propagate data along the Widget Tree. A Widget Tree is a representation of a UI screen that is composed of coupling widgets together. An example of a Widget Tree is shown in fig. 30.

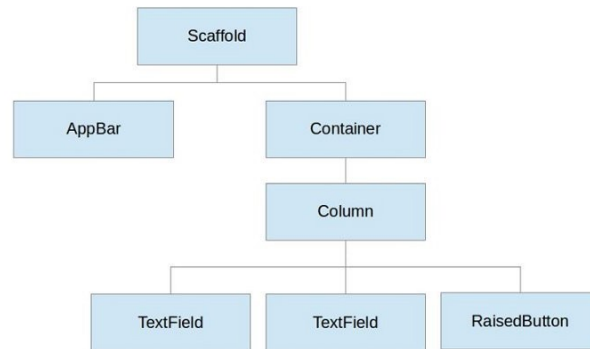


Figure 30 – An example Flutter widget tree. **Source:** <http://esolution-inc.com>

The Provider package offers a wrapper around InheritedWidget to make them easier to use and more reusable. With it, developers can include the data into the Widget Tree, propagating it to all descendant widgets. The widgets can access the data according to its type through a Consumer widget if they are stateless or using the `Provider.of<type>` constructor if they are stateful, obtaining the data from the nearest Provider ancestor. Fig. 31 shows an example with Consumer widgets.

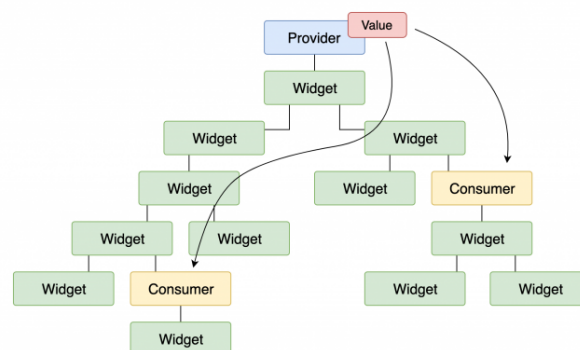


Figure 31 – An example widget tree with a Provider and Consumers.

Source: www.raywenderlich.com

6.2 Case Overview

The data-driven UI we propose is composed of 3 primary widgets: a Dashboard, Forms and Inputs. A Widget Tree is shown in Fig. 32 to provide an overview of how

these widgets are organized and related. The Dashboard acts as a customizable container, responsible for presenting a Form for each ThingUI Resource the user opted to see. Each Form then presents the necessary Inputs for each field an actuation request may need to interact with the associated Resource. The Inputs are the final UI elements shown to the user, who can now trigger actions by interacting with them.

Two objects manage business logic and are propagated through providers: *ActuationController* and *FormChangeNotifier*. They are detailed in the following sections with UML Class Diagrams that narrow the scope on classes and relations that are more relevant for each of them. However, to give an overview of the relation between them and the models they handle, a diagram with a broader scope can be seen in Fig. 33.

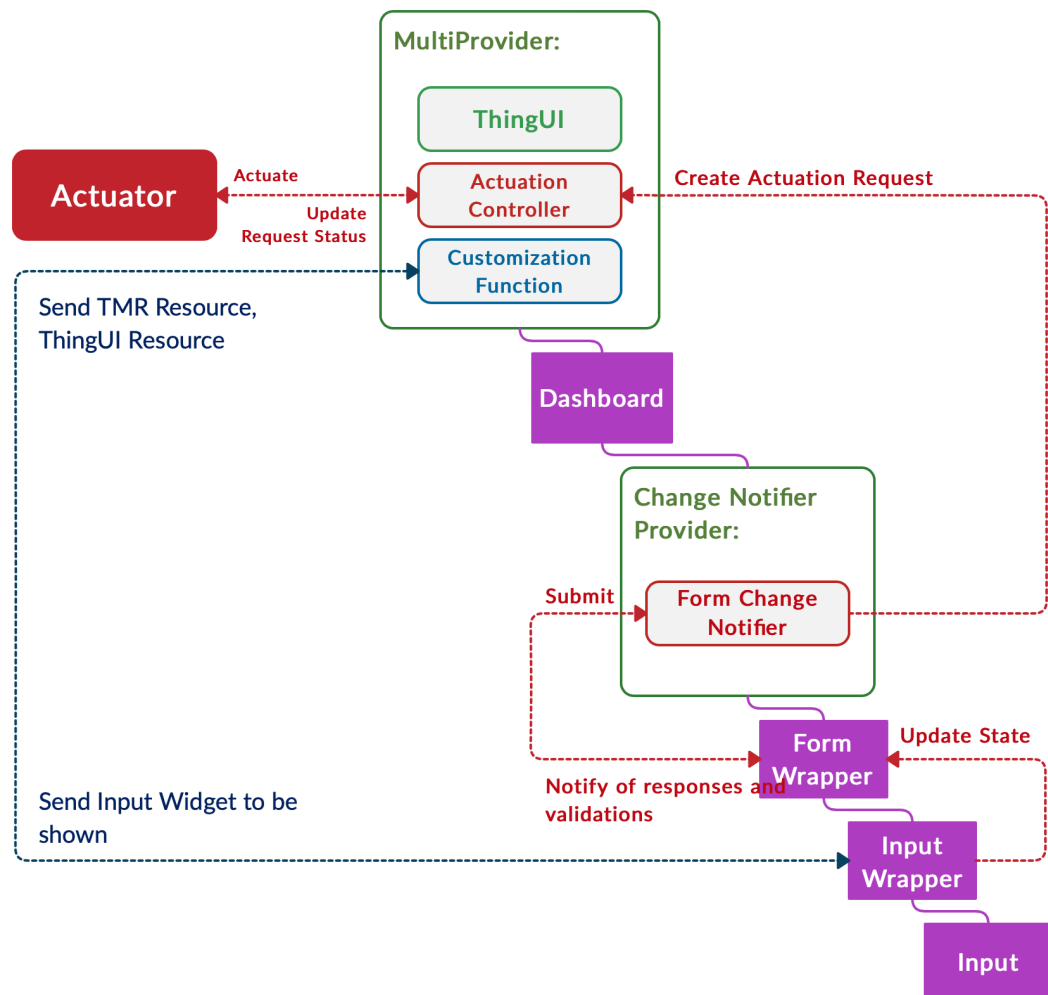
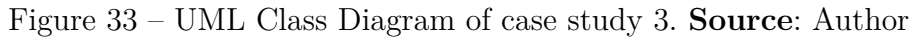


Figure 32 – The widget tree of case study 3 Flutter application. **Source:** Author



After receiving a ThingUI JSON, the application must then parse it to Flutter ThingUI model. As JSON is a well-known and established web standard, most programming languages have APIs available to help with this task.

The ThingUI model makes it possible to know which components should be presented to the user for each resource while the correspondent TMR model shows which data these components should receive.

²⁷ https://pub.dev/packages/json_serializable

6.4 Communicating with the WT

Every communication made with a WT is managed by a single *ActuationController*, an object that coordinates multiple *Actuators*. The actuators send messages to the WT when an *ActuationRequest* is received from a *FormChangeNotifier* or a *ResourceListener*.

The requests made by *FormChangeNotifiers* are actuations that change some of the WT's properties or make it execute specific procedures. On the other hand, requests made by *ResourceListeners* are queries to obtain the latest value of a property or event.

Fig. 34 contains a UML class diagram that describes the relevant models and relations that will be detailed in the next subsections.

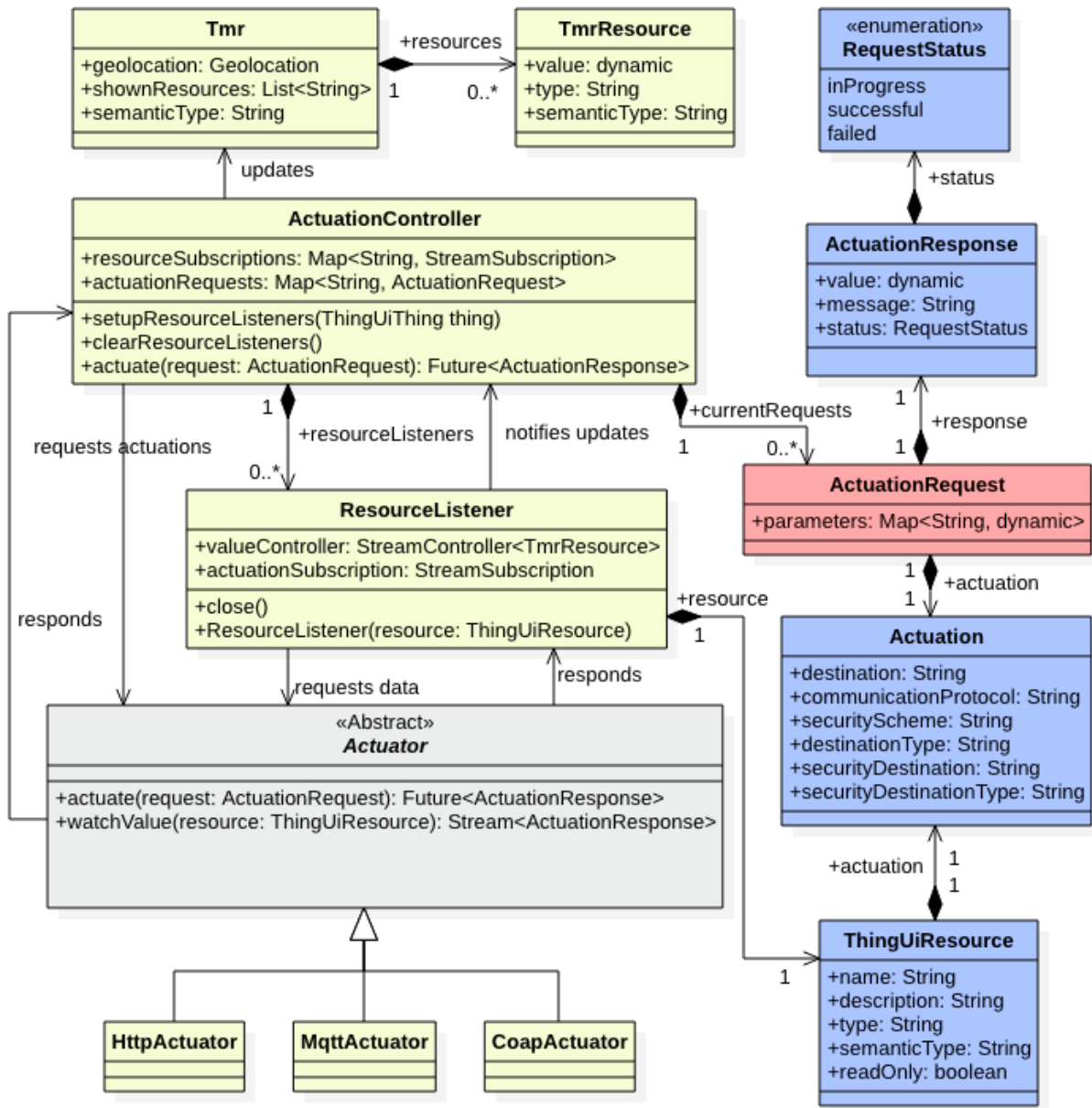


Figure 34 – Actuation controller UML Class Diagram. **Source:** Author

6.4.1 Actuation Controller

The Actuation Controller is instantiated when the user chooses a WT to interact with. A *MultiProvider* propagates it along with the *ThingUI* and an optional *Input Customization Function*. The *MultiProvider* is a direct ancestor of the *Dashboard* widget and uses its lifecycle to determine when the *ActuationController* must be set up and when it must be disposed of.

The *ActuationController* contains the following attributes: **SetupResourceListeners** - a method that receives a *ThingUI* and that is called by the *Dashboard* on its *initState* method; **ClearResourceListeners** - a method that is called by the *Dashboard* on its *dispose* method; **Actuate** - a method with two parameters: *actuationRequest(ActuationRequest)* and *resourceID(String)*; **actuationRequests** - a property that is a map with *String* keys and *ActuationRequest* values, used to avoid redundant requests; **resourceListeners** - a list that stores every *ResourceListener* created on the *setupResourceListeners* method.

The **setupResourceListeners** method instantiates the necessary *ResourceListeners* to receive updated values on the properties that are being shown to the user. The *ActuationController* subscribes to each of their *viewController's stream* property, creating an entry in the *resourceSubscriptions* map with the *resourceID* as a key and the *StreamSubscription* as a value. Every function passed to the streams *listen* method updates that specific *TMR Resource* whenever they receive data events. The TMR is also stored locally as a JSON using the *localStorage* package²⁸.

The **clearResourceListeners** method iterates through the *resourceListeners* list, calling each of their *close* method, and through the *resourceSubscription* map, calling each of their *cancel* method. This method is called whenever the *Dashboard* is disposed of: if the user decides to go back to the **Exploration** application state (explained in section 3.3), closes the app or powers off the smartphone.

When an *ActuationRequest* and its resource identifier are received from a *FormChangeNotifier* through the **actuate** method, they are added as an entry in the *actuationRequests* map and an *Actuator* is instantiated to send the message asynchronously. We do that to avoid redundant requests, guaranteeing there is only one request per resource at a time. Which *Actuator* is instantiated is determined by the *communicationProtocol* property of the *Actuation*. Whenever the message receives a response, fails to be sent or is timed out, the related *actuationRequests* entry is removed from the map and an *ActuationResponse* is sent back to the *FormChangeNotifier*. The *actuate* method returns an *ActuationResponse* within Flutter's *Future* class so it can work asynchronously. Receivers of a *Future* can register callbacks that handle the result once it is available. The *ActuationController* uses

²⁸ <https://pub.dev/packages/localstorage>

this not only to await an *Actuator* response, but also to apply timeout and retry criteria.

6.4.2 ActuationRequest and ActuationResponse

The ***ActuationRequest*** object contains the following properties: **Actuation** - a *ThingUiActuation* object (defined in section 5.4.4); **Parameters** - a map with String keys and dynamic values; **Response** - an *ActuationResponse* object.

The *ActuationRequest* is used to represent both data requests and actuation requests. Data requests are sent by *ResourceListeners*, using an actuator's *watchValue* method, while Actuation requests are sent by *FormChangeNotifiers*, calling an actuator's *actuate* method.

The ***ActuationResponse*** object contains the following properties: **Status** - a *RequestStatus* enum value, indicating whether the request is: *inProgress*, *successful* or *failed*; **Message** - a String that may be a confirmation message after a request was successful or detail why a request was not successful. **Value** - a dynamic property that contains the result of a data request.

6.4.3 Actuator

The *Actuator*'s responsibility is to process an *ActuationRequest*, creating the necessary protocol-bound message sent to interact with a WT. They can retrieve updated resource values to make the UI reflect the WT's current state or send data requesting an actuation according to the user's input.

Communication may be made **directly** with the WT, mediated by a **gateway** or a **cloud** service (as seen in section 4.2). The *ActuationRequest* contains a *ThingUiActuation* object, which includes the necessary data to know: which communication method must be used; which address the messages must be sent to; security protocols and procedures that must be applied.

We defined ***Actuator*** as an abstract class that contains the following properties: **actuate** - a method that receives an *ActuationRequest* as a parameter, returning an *ActuationResponse* asynchronously. **WatchValue** - a method that receives a *ThingUiResource* and returns a Stream of *ActuationResponses*.

We created concrete classes for each supported communication protocol (*HttpActuator*²⁹, *MqttActuator*³⁰ and *CoapActuator*³¹). They iterate through the *ActuationRequest* parameters to format them according to its *Actuation* communication protocol (e.g. JSON, XML, *UrlEncoded*).

²⁹ <https://pub.dev/packages/http>

³⁰ https://pub.dev/packages/mqtt_client

³¹ <https://pub.dev/packages/coap>

On the *actuate* method, each concrete actuator class has its specific error handling but must always return a *Future ActuationResponse* with a *successful* or *failed* status value and an optional message.

The **watchValue** method uses Flutter's Stream class to represent the multiple asynchronous events of receiving *ActuationResponses*. The Stream class is Flutter's implementation of observable sequences and there are two kinds of streams: *single-subscription* and *broadcast*. A *single-subscription* stream allows only a single listener during its lifetime. It does not start generating events until it has a listener, and it stops sending events when the listener is unsubscribed, even if the source of events could still provide more. A *broadcast* stream allows any number of listeners, and it fires its events when they are ready, whether there are listeners or not.

We decided to use only single-subscription streams so they can be managed and disposed of more easily when needed. Each concrete actuator defines its strategy to keep the stream updated. An actuator that uses HTTP or COAP may do this by Long Polling when available or by sending periodic GET requests, for example. In contrast, an actuator that uses MQTT may do this by merely subscribing to a specific topic.

6.4.4 ResourceListener

A *ResourceListener* is a class that sets up an *Actuator* with the necessary data request to keep a specific TMR resource up to date. We do that by filtering and treating the Stream of *ActuationRequests* the *Actuator* returns from the *watchValue* method.

They contain the following attributes: **ActuationSubscription** - a *StreamSubscription* to an *Actuator*'s *watchValue* returned stream; **Close** - a method that calls the *streamSubscription*'s *cancel* method; **ValueController** - a *StreamController* of dynamic values; **Resource** - a *ThingUiResource* instance;

A *ResourceListener* must be instantiated with a *ThingUiResource*, so the correct *Actuator* can be set up with its *ThingUiActuation*. They use *ThingUiActuations* in a similar manner to how an *ActuationController* uses them. The *communicationProtocol* property determines which type of *Actuator* should be instantiated, but instead of sending a request to the *actuate* method, its *ThingUiResource* is sent to *watchValue*.

After instantiating an *Actuator* and calling its *watchValue* method, the *ResourceListener* passes a function to the stream's *listen* method. This function analyzes the received *ActuationResponse* and converts its *value* into a *TmrResource* if possible. When this conversion is possible, the value is sent as an event through the *valueController*'s *add* method. When it is not possible, the *addError* method is called.

The *valueController* is a *StreamController*. It is a Flutter class that allows sending data, error and done events to a newly instantiated stream. That stream is the one made

available to the *ActuationController*. Another option would be to map the *Actuator*'s stream of *ActuationResponses* to *TmrResources* directly. However, we opted to include error events so the *ActuationController* could decide how to handle them.

6.5 UI Management

6.5.1 Dashboard Widget

The Dashboard acts as a customizable container, responsible for stacking *FormChangeNotifiers* for each *Resource* vertically within a *ListView*. By default, the order of *Resources* is obtained by iterating through a *ThingUi*'s properties, actions and events respectively. However, we decided to allow users to choose which *Resources* will be shown in the dashboard and choose their preferred presenting order because they may have different needs and interact with WTs in different ways. That is done by including an edition button that changes the Dashboard's state, swapping the *ListView* with a *ReorderableListView*.

The *ReorderableListView* is a Flutter widget that allows users to reorder its descendants by dragging them across the screen interactively. The order of resources is stored in the TMR model as a list of strings, representing each Resource key. While in this state, the dashboard shows every resource of the *ThingUi* and the user can mark the ones which will be omitted, storing this information in the TMR model. After filtering and ordering the *Resources* list, the dashboard then iterates it, instantiating a *FormChangeNotifier* for each one of them.

6.5.2 Form ChangeNotifier

In Flutter, a *ChangeNotifier* is a class that can be extended or mixed in that provides a change notification API using *VoidCallback* for notifications. It is one way to encapsulate the application state and can be propagated through the widget tree with a *ChangeNotifierProvider*, which makes consumers rebuild every time that a notification is received. They are deeper within the widget tree so the scope does not get too broad and to avoid unnecessary rebuilds.

For example: if there were a hypothetical "*ThingChangeNotifier*" at the same level as the *ActuationController*, a change to a single resource would trigger a rebuild of the entire Dashboard. That is why we decided not to make *ActuationController* a *ChangeNotifier* and why *FormChangeNotifiers* are associated with a single resource. This way, whenever there are updates to a specific resource, only the descendant *FormWrapper* will be rebuilt.

The *FormChangeNotifier* is responsible for propagating updated data of a specific TMR Resource and the *ActuationRequest* that may be active, notifying descendant widgets whenever they change. It also validates and sends user submissions through an *Actuation-*

Request sent to the *ActuationController*. It contains a *Form Wrapper* as a direct descendant. Fig. 35 contains a UML class diagram that gives an overview of the *FormChangeNotifier* and all the data it obtains.

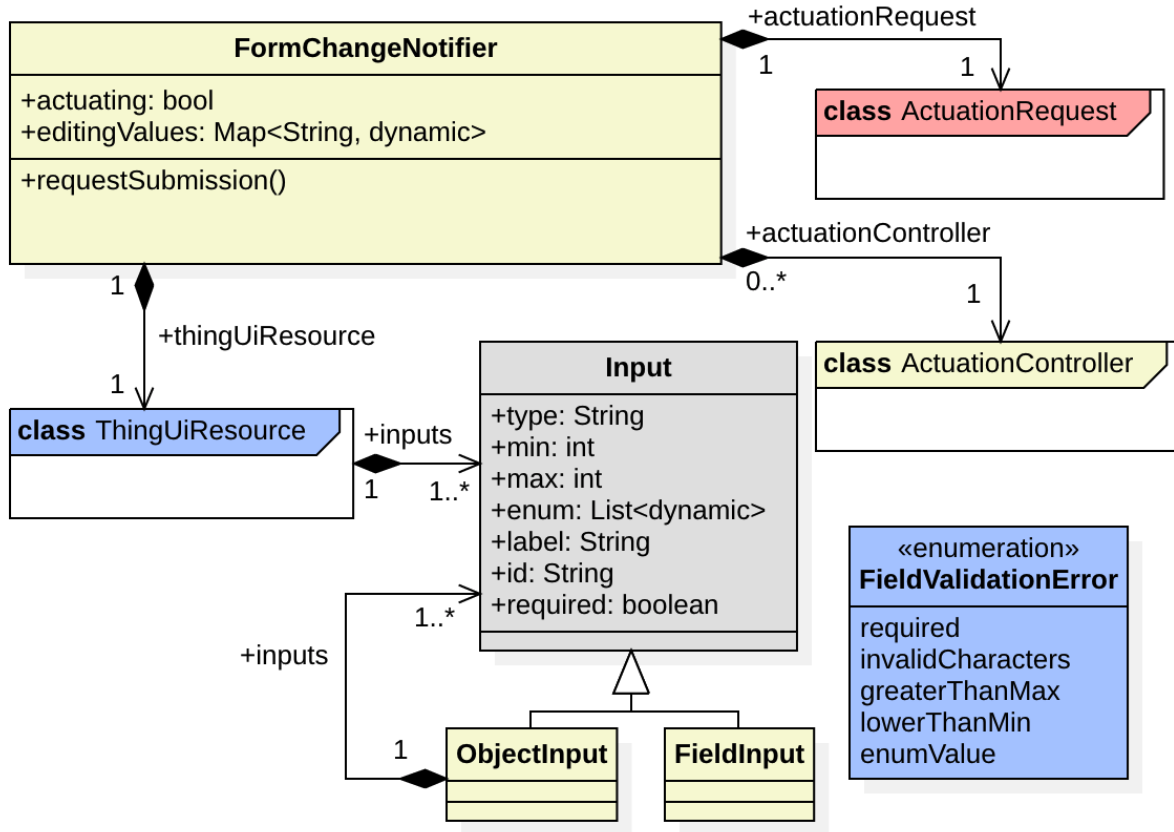


Figure 35 – Form ChangeNotifier UML Class Diagram. **Source:** Author

The *FormChangeNotifier* contains these attributes: **ResourceID** - the unique identifier string of the associated *Resource*; **ActuationRequest** - a computed variable obtained using the *resourceID* as a key in the ancestor *ActuationController*'s *actuationRequests* map. The *ChangeNotifier* method *notifyListeners* is called whenever this variable changes. **TmrResource** - a computed variable obtained using the *resourceID* as a key in the ancestor *TMR* resources map. **ThingUiResource** - a computed variable obtained using the *resourceID* as a key in the ancestor *ThingUI* properties, actions or events map. **RequestSubmission** - a method called after the user submits a form through a *FormWrapper*. An *ActuationRequest* is instantiated with the *editingValues* map and sent to the *ActuationController*'s *actuate* method. The *FormChangeNotifier* then awaits the response, notifying its descendants when it is available.

6.5.3 Form Wrapper

The *FormWrapper* is a container for a Flutter *Form* widget. The children of this widget are built by iterating through the *ThingUI* Form *inputs*, creating an *InputWrapper*

for each one of them. Each `InputWrapper` then contains a *FormField*, a widget that maintains the current state of a field, so that updates and validation errors are visually reflected in the UI. The *Form* widget provides a *FormState* that can help submit, reset and validate every descendant *FormField*.

The *FormState* can be obtained from any descendant to query or manipulate the form data as a whole. For example, calling `FormState.save` will invoke each *FormField*'s `onSaved` callback in turn. In our implementation, the *FormState.validate* method is called first and if every field is valid, then *FormState.save* is called.

Each `onSaved` callback creates an entry in the *FormChangeNotifier*'s *editingValues* map, with the field's identifier as a key and the field's data as a value. Then the ancestor *FormChangeNotifier*'s *requestSubmission* method is called, creating an *ActuationRequest* and sending it to the *ActuationController*.

Besides showing a list of Input Wrappers, the Form Wrapper may also include a submission button at the end of the list and an information button in the top right corner if the resource contains a description string. Tapping the information button presents an `AlertDialog` with the description text.

6.5.4 Input Wrapper

The *InputWrapper* widget is a container for a *FormField*. The wrapper contains the default criteria in the *buildInput* method to determine which widget will be shown to represent the form field, taking into account its `ThingUI` Input model.

A **DropDownButton** is shown whenever an Input contains **enum** values; a **TextField** is shown for **text** and **number** types, each with their respective validator. For **dates**, a **RaisedButton** with a `Text` as a child is shown. Its `onTap` method calls Flutter's default `showDatePicker`. A **Switch** is shown for *boolean* inputs. **Object** inputs are meant to be composed of multiple form fields and their value is always a map with `String` keys and dynamic values. They present a **Form** that automatically validates and submits whenever there are changes. The form fields of an **Object** input are also determined by calling the *InputWrapper*'s *buildInput* method.

Each input is validated according to their *ThingUI* Input counterpart. The `FieldValidationError` enum lists the possible errors an input may have: **required** - triggered when the user does not fill an input but the input *required* attribute is true; **enum** - triggered when the input only allows a set of values and the user informs one that does not belong to the set; **invalidCharacters** - triggered when the user includes a character that is not accepted for the input type (e.g. letters when the input type is *number*); **maxCharacters** - exclusive to the *text* input type and triggered when the user inserted a `String` with more characters than the input *max* property; **greaterThanMax**, **lowerThanMin** - exclusive

to *number* and *date* input types and triggered when the user informs a value that is not within the allowed range;

6.5.5 Input Customization

Developers can customize all inputs by providing a *CustomInput* function. The *buildInput* method always checks if an ancestor *CustomInput* Provider exists. If so, then it sends its specific *ThingUiResource* and *ThingUiInput* as parameters. If the *CustomInput* function returns a widget, it is used instead of the default ones described in the last section. We created two mixins called *ThingUiCustomInput* and *ThingUiCustomInputState* to allow developers to create custom widgets that conform to the expected behavior easily. The mixin has methods to update the *FormChangeNotifier*, obtain the WT's current resource value from the TMR and check the current *ActuationRequest*.

This strategy gives developers the freedom to show their custom widgets either in particular cases or replace all default widget occurrences altogether. We implemented two custom widgets for number inputs to test the customization: a Slider and a Stepper. We created a *CustomInput* function to consider the *semanticTypes* that the TD obtained in case study 1 has.

The TD has these 2 properties: **Level**, with type="number" and @type="Brightness Property"; **LimitTime**, with type="number" and @type="Limit Time Property". By default, the *InputWrapper* would show the same component for both as their *type* is the same. Fig. 36 shows how including a *CustomInput* can change that according to the *@type* of a property, showing a slider for **Level** and a stepper for **LimitTime**.

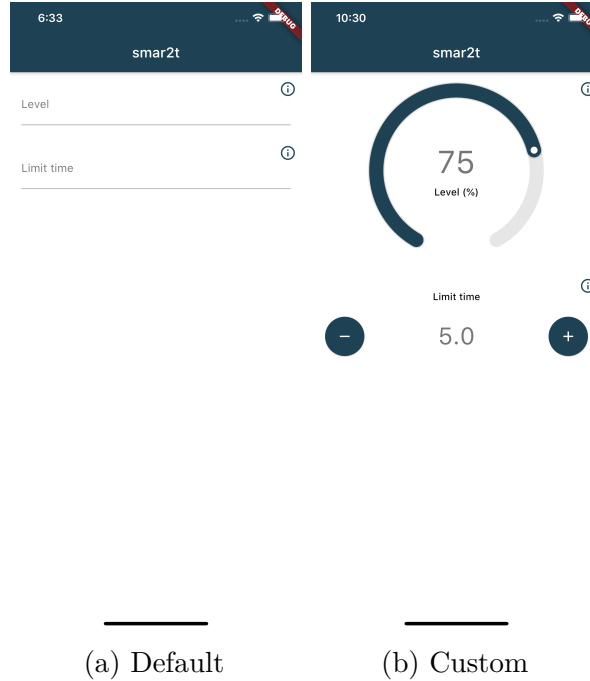


Figure 36 – Example of input customization. Source: author.

6.6 Discussion

Using a Declarative UI framework proved to be useful for defining customizations and easily adapting model-driven components. The first prototype was implemented in Vue, a progressive Web framework. More details about this implementation can be seen in Appendix C. Vue could have been a valid alternative were not for the WebXR issue in case study 1, explained in chapter 4, section 4.4. We also decided to migrate to Flutter because of its future support for web and desktop.

In Flutter, we successfully generated a UI according to a ThingUi JSON. That involved creating: models and their respective JSON Parsers by using the `json_serializable` package; actuators to manage the creation of protocol-bound messages and handle communication methods; a middleware class to manage actuators, resource listeners and avoid redundant requests; a user-customizable widget dashboard that lists properties, actions and events; a middleware class to relay user submissions and update the UI according to request responses; input wrappers that manage customization.

The UI successfully actuated upon and obtained data from the sample ESP32 Mozilla WT. The customization strategy worked well for input widgets, which remained consistent, accessed the expected providers and correctly sent the *ActuationRequests*.

The case study also contains certain limitations as it only tested an `HttpActuator` and a Mozilla TD. Further tests with `CoapActuator` and `MqttActuator` as well as a W3C TD are needed to affirm it is a complete solution concretely, but the preliminary successful results indicate that these are feasible.

7 Conclusions

This thesis proposed an architecture called smAR²t(Smart Model-driven And Reactive Augmented Reality Things) and detailed the development process of it's modules through 3 case studies, successfully integrating Model Driven Engineering, Reactive Programming and Augmented Reality while adhering to the Web of Things specifications. The results of each case study experiments reinforce our theory that smAR²t can allow the creation of a standardized virtual interaction environment between users and smart objects even in heterogeneous physical environments.

7.1 Objectives Review

7.1.1 Objective 1

To discover which tools, languages and techniques are adequate for the proposed scenario.

These were chosen after we experimented creating the 3 case studies in different languages and frameworks (Vue, ReactNative and SwiftUI). We decided to use Flutter because of it's current native support to iOS and Android (and future Web and Desktop support), to experiment with it's reactive state management strategies and to help with contributions to it's AR packages. The final version of our approach used the following tools, languages and techniques:

- WebThing
 - **Language / Framework:**
 - * C and C++ / Arduino
 - **Tools:**
 - * ESP32 Development Board
 - * ArduinoMDNS
 - * Webthing-Arduino Framework
 - **Techniques:**
 - Procedural imperative programming

- MDE Transformation Server
 - **Languages/Framework:**
 - * Java, ATL / EMF
 - **Tools:**
 - * EMF-Json
 - * EMFTVM
 - **Techniques:**
 - * MDE
- Client application
 - **Language / Framework:**
 - * Dart / Flutter
 - **Tools:**
 - * ARCore
 - * ARKit
 - * mDNS
 - * State Management: ChangeNotifiers and Provider
 - **Techniques:**
 - * Reactive Programming
 - * Declarative UI

7.1.2 Objective 2

To demonstrate at least one example of how each smAR2t module can be implemented through descriptive case studies.

We successfully implemented 3 case studies: 1 - Thing Discovery and Augmented Reality; 2 - Model Transformations; 3 - UI and Actuation.

Implementing them helped enlighten development details, requirements and issues. As can be seen on appendixes A and B, they also helped refine and enhance the proposed example application. Even though the case studies were not integrated into a single application, their resulting artifacts serve as input to each other.

Case 1 either obtains a TD that will be sent to case 2 or a cached ThingUI that will be sent to case 3. Case 2 transforms a TD into a ThingUI, that is then sent to Case 3. Case 3 performs actuations upon and obtains data from a WT. When the user finishes interacting with the selected WT and wants to enter Exploration mode, then case 1 would be shown again, repeating the cycle.

This connection between the cases demonstrated that smAR2t is feasible and we believe it can provide a foundation for developers as well as important insights for scientists, contributing to the future of IoT. We also hope that the discussions in each chapter can

help incite debates about the most adequate ways of handling heterogeneity and providing users with meaningful, practical solutions.

7.1.3 Objective 3

To identify limitations and issues, describing how they can be addressed or avoided.

Device geolocation precision - Our initial draft considered using only the device's sensors to determine the augmentation locations, proposing a way to store their positions and recover them seamlessly. After a bibliographical research and development experimentations, we found out that a great part of mobile devices that are used today do not provide the level of precision needed to create the expected experience. Thus, that led us to use Augmented Images as a more adequate approach

Current Flutter AR packages - During the development of case 3, we found out that the Flutter ARCore package had limitations that did not allow it to offer features that the ARKit package had. To fully implement case 3 in both platforms, we made contributions adding the following features: capture ARSession camera image; obtain world coordinates of detected Augmented Images.

TD Context Extensions - W3C's Web Thing Model includes support to more detailed vocabularies by referencing a JSON-LD context through HTTP Link Headers. Our proposed metamodel currently does not include it. We believe that the addition of TD Context Extensions external name-value pairs could be supported by 2 different strategies that other researches may prove:

- Defining individual metamodels for each external vocabulary and a way to integrate these with the base W3C TD Metamodel.
- Defining a metamodel that generically represents external vocabularies and a new metamodel that merges the W3C TD Metamodel with it, along with specific transformation rules to create a ThingUI.

7.2 Contributions

This thesis conclusions and findings can help developers focus on their problem domain instead of dealing with boilerplate code, aiming to conform to a specific tech stack. This is done by providing these contributions:

- A detailed application architecture and flow, that can serve as a foundation for future designs.
- Flutter ARCore plugin improvements.

- W3C and Mozilla TD metamodels along with parsers to automatically create models from a JSON.
- A model called ThingUi that contains the necessary data needed to generate a UI.
- An effective example of actuation and UI management with a ThingUi that can be adapted to other environments.

However, currently smAR²t does not completely embrace every aspect of an IoT application, so there is still margin for improvements and corrections. By implementing some of them, it could serve as a template like the ATLauncher project. Also, if the community gets engaged with the idea, the project is structured with modularity in mind, allowing them to separately contribute in a variety of ways: custom inputs; additional actuator concrete classes; custom input function criteria; security and routines. More possibilities are presented in the following section.

7.3 Future Work

- Explore alternate and experimental ways of calculating augmentation positions (like persistent cloud anchors³²)
- Explore different ways to improve geolocalization accuracy also including altitude (beacons, WiFi triangulation services).
- Implement MQTT and COAP actuators, as well as testing multiple W3C TDs besides a single Mozilla TD.
- Structure and integrate the case studies as a single application, making it openly available so developers and researchers can easily reproduce and improve it.
- Make more contributions to the Flutter ARCore package or even create the necessary abstractions to develop a single package that can manage AR sessions in iOS, Android and Web environments.
- Providing example applications in different programming languages and frameworks like React or Vue.

³² https://developers.google.com/ar/develop/callforcollaborators#persistent_cloud_anchors

Bibliography

- ALLMENDINGER, R. W.; SIRON, C. R.; SCOTT, C. P. Structural data collection with mobile devices: Accuracy, redundancy, and best practices. *Journal of Structural Geology*, Elsevier BV, v. 102, p. 98–112, set. 2017. Available at: <<https://doi.org/10.1016/j.jsg.2017.07.011>>. Cited on page 22.
- BAINOMUGISHA, E. et al. A survey on reactive programming. *ACM Computing Surveys*, Association for Computing Machinery (ACM), v. 45, n. 4, p. 1–34, ago. 2013. Available at: <<https://doi.org/10.1145/2501654.2501666>>. Cited on page 12.
- BEZIVIN, J.; GERBE, O. Towards a precise definition of the OMG/MDA framework. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE Comput. Soc, 2001. Available at: <<https://doi.org/10.1109/ase.2001.989813>>. Cited on page 9.
- CARMIGNIANI, J.; FURHT, B. Augmented reality: An overview. *Handbook of Augmented Reality*, Springer Science & Business Media, Berlin, Germany, p. 3–46, August 2011. Cited 2 times on page 3 and 8.
- DEURSEN, A. van; KLINT, P.; VISSER, J. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, v. 35, n. 6, p. 26–36, Jun 2000. Cited on page 10.
- FOUQUET, F. et al. An eclipse modelling framework alternative to meet the models@runtime requirements. In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2012. p. 87–101. Available at: <https://doi.org/10.1007/978-3-642-33666-9_7>. Cited on page 11.
- HANUS, M.; KLUSS, C. Declarative programming of user interfaces. In: *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 2008. p. 16–30. Available at: <https://doi.org/10.1007/978-3-540-92995-6_2>. Cited on page 13.
- JáCOME-GUERRERO, S. P.; FERREIRA, M.; CORRAL, A. Software development tools in model-driven engineering. In: *CONISOFT, The International Conference on Software Engineering Research and Innovation*. 2017. Cited 3 times on page 9, 3, and 11.
- KLEPPE, A. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003. ISBN 032119442X. Available at: <<https://www.xarg.org/ref/a/032119442X/>>. Cited on page 10.
- KOHLER, M.; SALVANESCHI, G. Automated refactoring to reactive programming. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019. Available at: <<https://doi.org/10.1109/ase.2019.00082>>. Cited on page 11.
- LAN, L. et al. An IoT unified access platform for heterogeneity sensing devices based on edge computing. *IEEE Access*, Institute of Electrical and Electronics Engineers (IEEE), v. 7, p. 44199–44211, 2019. Available at: <<https://doi.org/10.1109/access.2019.2908684>>. Cited on page 4.

- LI, S.; XU, L. D.; ZHAO, S. The internet of things: a survey. *Information Systems Frontiers*, Springer Science and Business Media LLC, v. 17, n. 2, p. 243–259, abr. 2014. Available at: <<https://doi.org/10.1007/s10796-014-9492-7>>. Cited on page 1.
- LIBERT, M. *Using Augmented Reality to Control Device Interaction in a Web of Things Environments*. Thesis (Master) — Vrije Universiteit Brussel, 2020. Cited on page 4.
- LINN, M. C.; CLANCY, M. J. The case for case studies of programming problems. *Communications of the ACM*, Association for Computing Machinery, Inc., v. 35, n. 3, p. 121–133, 1992. Cited on page 18.
- LLOYD, J. W. Practical advantages of declarative programming. In: *GULP-PRODE*. 1994. p. 18–30. Cited on page 13.
- MENS, T.; GORP, P. V. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, Elsevier BV, v. 152, p. 125–142, mar. 2006. Available at: <<https://doi.org/10.1016/j.entcs.2005.10.021>>. Cited on page 10.
- MOTTA, R. C.; OLIVEIRA, K. M. de; TRAVASSOS, G. H. On challenges in engineering IoT software systems. In: *XXXII Brazilian Symposium on Software Engineering - SBES '18*. 2018. p. 42–51. ISBN 9781450365031. Cited 2 times on page 9 and 1.
- NOROUZI, N. et al. A Systematic Review of the Convergence of Augmented Reality, Intelligent Virtual Agents, and the Internet of Things. p. 1–24, 2019. Cited on page 3.
- NOVAKOVA, L.; PAVLIS, T. L. Assessment of the precision of smart phones and tablets for measurement of planar orientations: A case study. *Journal of Structural Geology*, Elsevier BV, v. 97, p. 93–103, abr. 2017. Available at: <<https://doi.org/10.1016/j.jsg.2017.02.015>>. Cited on page 22.
- SALVANESCHI, G. et al. An empirical study on program comprehension with reactive programming. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, 2014. Available at: <<https://doi.org/10.1145/2635868.2635895>>. Cited on page 11.
- SALVANESCHI, G. et al. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, Institute of Electrical and Electronics Engineers (IEEE), v. 43, n. 12, p. 1125–1143, dez. 2017. Available at: <<https://doi.org/10.1109/tse.2017.2655524>>. Cited on page 11.
- SANTOS, B. P. et al. Internet das coisas: da teoria à prática. In: *Simpósio Brasileiro de Redes de Computadores*. 2016. Cited 2 times on page 1 and 6.
- SCHWAB, L. *Universal Explorer for the Web of Things*. Thesis (Master) — Faculty of Science University of Bern, 2018. Cited on page 4.
- SEIDEWITZ, E. What models mean. *IEEE Software*, Institute of Electrical and Electronics Engineers (IEEE), v. 20, n. 5, p. 26–32, set. 2003. Available at: <<https://doi.org/10.1109/ms.2003.1231147>>. Cited on page 9.
- SHARMA, N.; SHAMKUWAR, M.; SINGH, I. The history, present and future with IoT. In: *Intelligent Systems Reference Library*. Springer International Publishing, 2018. p. 27–51. Available at: <https://doi.org/10.1007/978-3-030-04203-5_3>. Cited on page 1.

STEFANAKIS, E. Web mercator and raster tile maps: two cornerstones of online map service providers. *Geomatica*, NRC Research Press, v. 71, n. 2, p. 100–109, 2017. Cited 2 times on page 9 and 24.

SZOT, T. et al. Comparative analysis of positioning accuracy of samsung galaxy smartphones in stationary measurements. *PLOS ONE*, Public Library of Science (PLOS), v. 14, n. 4, p. e0215562, abr. 2019. Available at: <<https://doi.org/10.1371/journal.pone.0215562>>. Cited on page 22.

SZVETITS, M.; ZDUN, U. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software and Systems Modeling*, v. 15, n. 1, p. 31–69, 2016. ISSN 16191374. Cited on page 11.

WAGELAAR, D. et al. Towards a general composition semantics for rule-based model transformation. In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2011. p. 623–637. Available at: <https://doi.org/10.1007/978-3-642-24485-8_46>. Cited on page 33.

ZACHARIAH, T.; DUTTA, P. Browsing the Web of Things in Mobile Augmented Reality. *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, p. 129–134, 2019. Cited on page 4.

Appendix

APPENDIX A – Example application V1

This was the first design, created to provide an overview of the idea for the first drafts and was made before any extensive bibliographic study. The main differences between what was designed in this version and what was implemented in the final version were the following:

WOT API - W3C's Javascript API would be used, however it was not yet available when the implementation started and is currently under specification process;

WebThing Types - Only WT's with connection by cloud would be supported, disregarding direct and by gateway types;

WebThing Discovery - The discovery process would consist of searching for TD Repositories that supported geolocation and querying them, filtering results according to location proximity;

Models - TMR and ThingUi would be a single class; the metamodels would be created in the client application using the System Runtime API³³; TDs would be manually parsed in Javascript.

Augmented Interaction - WebXR would be used and all the UI would be presented within an augmentation, with a set of 3D components.

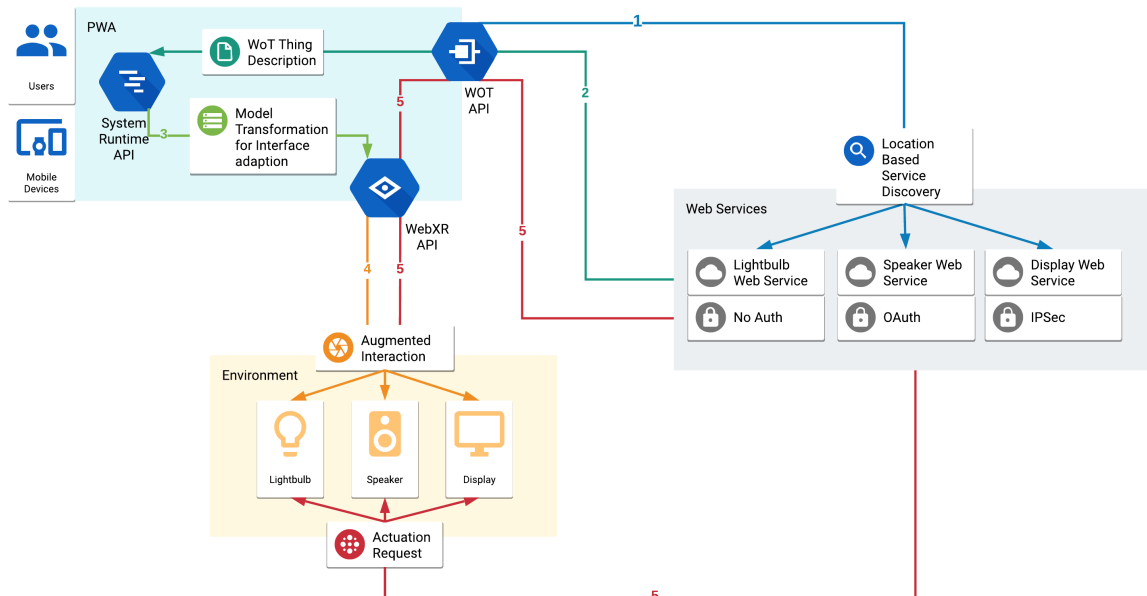


Figure 37 – First concept of the example application. **Source** - Author

³³ <https://github.com/design-first/system-runtime>

APPENDIX B – Example application V2

This diagram was published in *Insightful Ideas and Emerging Results - Simpósio Brasileiro de Engenharia de Software (SBES)*, 33. 2019, Salvador. At this time a more extensive bibliographic review had been made, which helped mature some concepts and demanded some as the WoT API was not yet available, like including different WT types and using a different TD discovery method.

The main differences between what was designed in this version and what was implemented in the final version were the following:

WebThing Discovery - The discovery process already included QRCode and Geolocation, but relied on an idea of hosting TDs in the application's server which would be updated by every user. That because the EMF-REST strategy would be applied (explained in chapter 5 section 5.6).

Models - The TMR model concept was created, but not the ThingUi. The client application would still manage metamodels and transformations.

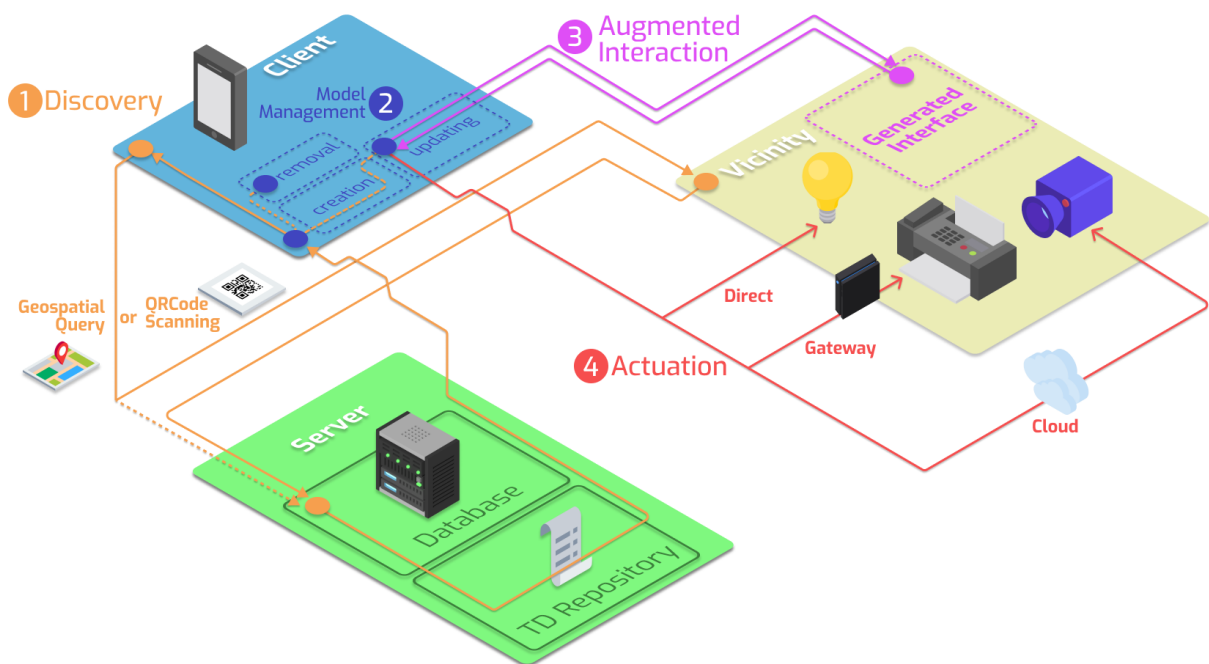


Figure 38 – Second proposal of example application. Source: Author

APPENDIX C – UI and Actuation Vue prototype

This component diagram was published in Proceedings of the 25th Brazilian Symposium on Multimedia and the Web. 2019, Rio de Janeiro. p. 57-60. It was developed in Vue, a front-end progressive web framework that allows creating of components with template models. It helps keep UI updated, reflecting any changes that occur to a corresponding watcher instance. The main differences between what was designed in this version and what was implemented in the final version were the following:

Parsing - TDs were directly parsed in Javascript through a Factory, lacking flexibility and reusability.

Models - The ThingUi was still not defined and the TMR would contain both data (metadata used to create components and current values of a thing).

Actuation - The *ActuationController* class started to be defined, but was overloaded with roles because important models and classes like *ActuationRequest*, *ResourceListener* and *Actuator* were not yet present.

Customization - The input customization idea was introduced, using Vue's *component* tag and concepts that are also in the final version, like a customization function and an input mixin. However, lacking the composability found in the final version, inputs also needed to communicate directly with the *ActuationController* and there were different mixins for properties, action and events.

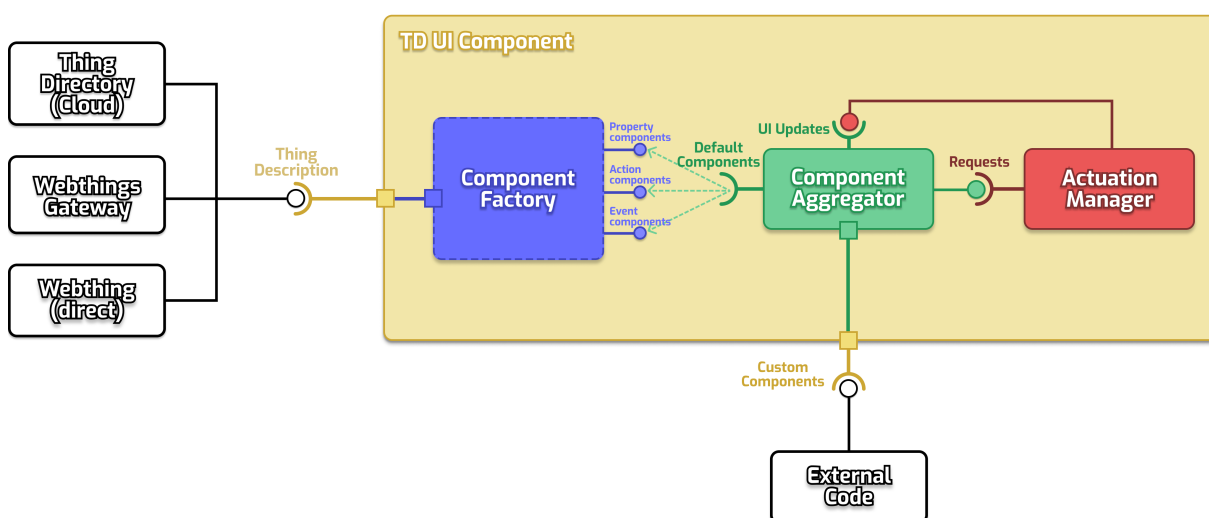


Figure 39 – Second proposal of example application. Source: Author

APPENDIX D – Project Repositories

All the code that was implemented during this research is open source and available under the permissive MIT License on Github. A smar2t organization page was created to gather all of the repositories: <<https://github.com/smar2t>>

Case study 1: <https://github.com/smar2t/ar_td_discovery>

Case study 2: <https://github.com/smar2t/smar2t_models>

Case study 3: <https://github.com/smar2t/td_driven_ui>

The ESP32 code can be accessed in: <https://github.com/smar2t/esp32_example>

Our fork with contributions to the Flutter ARCore plugin can be found in: <https://github.com/smar2t/arcore_flutter_plugin>

Appendix C's Vue UI is available on this link: <https://github.com/smar2t/td_interface_builder>