Institute of Architecture of Application Systems

Internet of Things Patterns for Device Bootstrapping and Registration

Lukas Reinfurt[1,2], Uwe Breitenbücher[1], Michael Falkenthal[1], Frank Leymann[1], Andreas Riegg[2]

[1]Institute of Architecture of Application Systems, University of Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

[2]Daimler AG, Stuttgart, Germany
{firstname.lastname}@daimler.com

Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. 2017. Internet of Things Patterns for Device Bootstrapping and Registration. EuroPLoP'17, , Article (), 27 pages.
DOI: 10.1145/3147704.3147721

BIBTEX


```
@inproceedings{Reinfurt.2017,
  author    = {Reinfurt, Lukas and Breitenb{\"u}cher, Uwe and Falkenthal, Michael and Leymann, Frank and Riegg, Andreas},
  title     = {Internet of Things Patterns for Device Bootstrapping and Registration},
  booktitle = {Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP)},
  year      = {2017},
  publisher = {ACM},
  doi       = {10.1145/3147704.3147721}
}
```


© ACM 2017
This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version is available at ACM: http://dx.doi.org/10.1145/3147704.3147721

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Universität Stuttgart
Germany

Internet of Things Patterns for Device Bootstrapping and Registration

LUKAS REINFURT, Daimler AG
UWE BREITENBÜCHER, University of Stuttgart
MICHAEL FALKENTHAL, University of Stuttgart
FRANK LEYMANN, University of Stuttgart
ANDREAS RIEGG, Daimler AG

All kinds of large and small organizations are trying to find their place in the Internet of Things (IoT) space and keep expanding the portfolio of connected devices, platforms, applications, and services. But for these components to be able to communicate with each other they first have to be made aware of other components, their capabilities, and possible communication paths. Depending on the number and distribution of the devices this can become a complicated task. Several solutions are available, but the large number of existing and developing standards and technologies make selecting the right one confusing at times. We collected proven solution descriptions to reoccurring problems in the form of patterns to help Internet of Things architects and developers understand, design, and build systems in this space. We present ten new patterns which deal with initializing communication. Five of these patterns are described in detail in this paper. The patterns FACTORY BOOTSTRAP, MEDIUM-BASED BOOTSTRAP, and REMOTE BOOTSTRAP are used to bring information for setting up communication onto the device. Devices can be registered using the AUTOMATIC CLIENT-DRIVEN REGISTRATION, AUTOMATIC SERVER-DRIVEN REGISTRATION, or MANUAL USER-DRIVEN REGISTRATION patterns. During this process, a SERVER-DRIVEN MODEL, PRE-DEFINED DEVICE-DRIVEN MODEL, or DEVICE-DRIVEN MODEL is stored in a DEVICE REGISTRY to digitally represent the device.

CCS Concepts: • **Computer systems organization** → Embedded and cyber-physical systems; • **Software and its engineering** → Design Patterns

Additional Key Words and Phrases: Internet of Things, Device, Bootstrapping, Registration

ACM Reference Format:

Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. 2017. Internet of Things Patterns for Device Bootstrapping and Registration. *EuroPLoP'17*, , Article (), 27 pages.
DOI: 10.1145/3147704.3147721

1. INTRODUCTION

The Internet of Things (IoT) has been growing in recent years. More and more everyday objects and previously unconnected devices now talk as “things” to the internet, platforms, services, applications, and to other devices. These cyber-physical systems are used to make data collected by the devices accessible for monitoring and analytics and to remotely control these devices and their surroundings [Voas 2016].

The growth of the IoT is not controlled by a central entity, but rather happens organically. Large and small companies, research organizations, open-source projects, and governments are all trying to find a footing in this evolving field. Thus, the technologies and standards used vary [Atzori et al. 2010; Gubbi et al. 2013; Ishaq et al. 2013]. Additionally, solutions are often built with a particular market in mind,

Author's address: Lukas Reinfurt and Andreas Riegg: Epplestraße 225, 70546 Stuttgart, Germany; email: [firstname].[lastname]@daimler.com; Uwe Breitenbücher, Michael Falkenthal, and Frank Leymann: Universitätsstraße 38, 70569 Stuttgart, Germany; email: [firstname].[lastname]@iaas.uni-stuttgart.de.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
EuroPLoP '17, July 12–16, 2017, Irsee, Germany
© 2017 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-4848-5/17/07…$15.00
https://doi.org/10.1145/3147704.3147721

for example, home automation or smart factories, which has led to solution silos [Singh et al. 2016]. This results in a confusing collection of solutions which all basically do the same: i) send and collect data from devices, ii) monitor and analyze the devices and their surroundings through this data, and iii) use the resulting knowledge to remotely control devices and influence the environment.

We have identified and collected patterns in IoT systems with the goal of creating an IoT pattern language. These patterns describe abstract solutions to reoccurring problems and, thus, bring some clarity to the large collection of IoT solutions. The IoT pattern language will link these patterns to provide guidance for reading and applying IoT patterns. The patterns and the pattern language should help IoT architects and developers to select, use, and build IoT solutions. They may also give anyone interested in the IoT an overview of common problems and solutions. In our previous work, we described eight patterns for communication with and management of IoT devices [Reinfurt et al. 2016, 2017a], and six patterns for device energy supply and operation modes [Reinfurt et al. 2017b]. An overview of these and future IoT patterns can also be found online[2]. Some of these patterns make the assumption that the device is already connected and able to communicate with a backend server or another device. But at some point, any device has to undergo an initial setup procedure where these connections are configured. We divide this into two steps (also see Section 3): *Bootstrapping*, where the information required for the device to initiate communication is brought onto the device, and *registration*, where a device is made known to its communication partners. For this procedure we collected ten new patterns, five of which we present in detail in this paper.

The remainder of this paper is structured as follows: Section 2 presents related work. Section 3 gives a general overview of the device bootstrapping and registration process and explains the terminology. Section 4 shortly summarizes the pattern format we use and how we identified the patterns. Section 5 gives a short overview of all ten patterns, followed by a detailed description of five selected patterns. Section 6 summarizes and concludes the paper.

2. RELATED WORK

Our patterns are based on the concept first introduced by Alexander et al. for the domain of architecture [1977]. This pattern concept has since then been applied in other domains, including IT. Examples include the Messaging Patterns by Hohpe et al. [2004] or the Cloud Computing Patterns by Fehling et al. [2014]. Others have worked on the pattern writing process [Meszaros and Doble 1996, Harrison 2006b, 2006a; Wellhausen and Fießer 2012; Fehling et al. 2014; Fehling et al. 2015b]. There has also been work on improving the usability of abstract patterns. For example, linking abstract patterns to technology specific patterns [Falkenthal et al. 2016] or to solution implementations [Falkenthal et al. 2014a, 2014b] enables building solution languages [Falkenthal and Leymann 2017].

In our previous work, we introduced IoT Patterns for devices [Reinfurt et al. 2017b] and IoT communication and management [Reinfurt et al. 2016, 2017a]. The former publication contains patterns for different IoT device energy supply types and operation modes, while the latter are concerned with different solutions to handle intermittent and constrained communication and remote device management. But all these patterns assume that the initial configuration of communication channels has already happened.

Some other patterns for IoT or related topics exist. Eloranta et al. published patterns for constructing distributed control systems [Eloranta et al. 2014a]. These patterns are mainly concerned with reliability and fault-tolerance within large machines for foresting, mining, construction, etc., and not with small, constrained devices common in the IoT. But they describe three patterns for system startup:

[2] http://www.internetofthingspatterns.com

BOOTSTRAPPER, SYSTEM-START-UP, and START-UP NEGOTIATION. The BOOTSTRAPPER pattern is concerned with getting hardware into a defined state after every start-up by initializing memory and other buses and running tests to identify problems early, often in several stages. SYSTEM-START-UP divides multiple nodes within a system, which have different start-up times and resource requirements, into master and slaves to handle these dependencies and ensure a proper start-up process. START-UP NEGOTIATION, which is presented in more detail in [Eloranta et al. 2014b], describes a mechanism, where nodes announce their presence and a central negotiator gathers these announcements to get an overview of the state and available functionality of the overall system. These patterns are somewhat related to ours and could be combined with them, but are generally concerned with lower levels of bootstrapping. Qanbari et al. present four patterns in the IoT area for edge application provisioning, deployment, orchestration, and monitoring [Qanbari et al. 2016]. These patterns describe using existing technologies like Docker and Git to provision containers or code onto devices. For this to work, a connection between the device and some backend server has to be configured, which they assume has been done already. Thus, these patterns can be seen as a next step after bootstrapping and registering the device with our patterns presented in this paper. Another paper describes a pattern language for IoT applications, where existing patterns were taken and classified into layers according to the problems they solve. The resulting clusters or classes of patterns were put into relation to each other to form a pattern language which allows navigating between these clusters [Chandra 2016]. Most of the patterns used in this work are from blogs and do not resemble the patterns we are talking about, neither in form, nor scope. Besides, the resulting language only allows navigating between the very broad clusters, but not between the single patterns. Thus, its usability is limited.

3. TERMINOLOGY AND OVERVIEW

As described in more detail in [Reinfurt et al. 2017b], an IoT system is usually made up of a few different components, as shown in Figure 1. There are usually several *devices*, which contain sensors and actuators, as well as processing, communication, and energy components. These devices can be anything from environmental sensors to smart fridges, cars, or manufacturing machines. There is also a *backend server*, a central, powerful component which handles data processing, device management, and other tasks. There may also be *other components* which access devices or device data through the backend server.

A central aspect of IoT devices is that they are connected, in general to i) a backend server, ii) a DEVICE GATEWAY[3], or iii) to other devices. But such a connection has to be configured at some point in time. The patterns presented in this paper describe various aspects of this configuration process. By applying these patterns, a factory-new device without any information about its communication partners is turned into a device which i) knows how and with whom to communicate and ii) that is known to its communication partners. Figure 2 provides an overview of this process, which is divided into two parts: *Device Bootstrapping* and *Device Registration*.

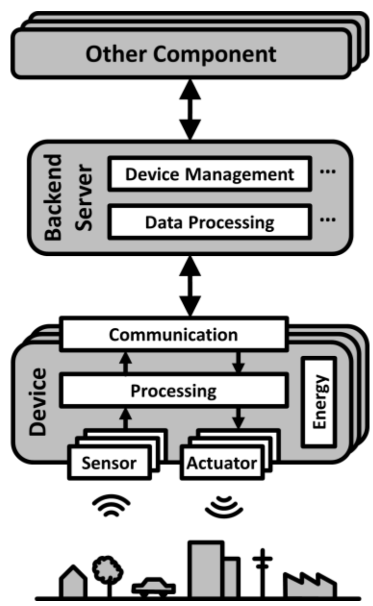

Figure 1. IoT Overview {Reinfurt 2017 #1367}.

[3] Some devices do not have the communication technology to connect directly to an existing network, such as the internet. These devices can be connected through a translating component, called a DEVICE GATEWAY [Reinfurt et al. 2016, 2017a].

The term *bootstrapping* is used in various fields, such as finance, statistics, linguistics, computing, etc., to describe a process where a simple system activates a more complicated system[4]. In the area of networked communication and computing, this commonly means the initial distribution of settings and configuration, such as identification and security information, in order to be able to create trust between different communication partners and to start communication [Kumar et al. 2009; Heer et al. 2011; IETF 2012]. Thus, in this paper, *Device Bootstrapping* refers the process wherein the bootstrap information, i.e., information which a device needs to initiate communication with another component, such as IP addresses and authentication data, is put on the device. As shown in Step 1 in Figure 2, we collected three patterns which describe how and when this information may be placed onto the device: FACTORY BOOTSTRAP, MEDIUM-BASED BOOTSTRAP, and REMOTE BOOTSTRAP.

Once the device has this bootstrap information, it may initiate communication with other components. But often, a second step is necessary: *Device Registration*. By being registered, the device is made known to its communication partners and is often bound to a specific user or company account, as well as its virtual representation, the device model. This allows other components to query a server or platform about the metadata, such as device ID, device type, manufacturer, etc., of all the connected devices, even if they are currently offline[5]. This step, for which we collected the three patterns AUTOMATIC SERVER-DRIVEN REGISTRATION, AUTOMATIC CLIENT-DRIVEN REGISTRATION, and MANUAL USER-DRIVEN REGISTRATION, is shown in Step 2 in Figure 2.

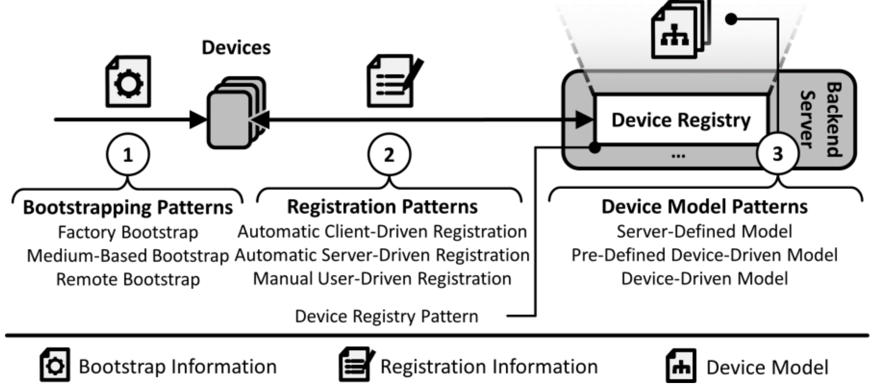

Figure 2. Overview of the device bootstrapping and registration process and associated patterns we present in this paper.

During this step, a *Device Model* instance, a virtual representation of the device, is stored in the DEVICE REGISTRY, as shown in Step 3 in Figure 2. This virtual representation contains static metadata about the device, such as name, type, id, manufacturer, etc., but may also include a description of operations that can be invoked on them (for example, operating an actuator, or restarting the device) or information that can be retrieved from them (for example, the most recent values of a sensor). The DEVICE REGISTRY allows other components in the system to query it about all registered devices if they need to. We identified three different types of *Device Models*, which we describe in the patterns SERVER-DEFINED MODEL, PRE-DEFINED DEVICE-DRIVEN MODEL, and DEVICE-DRIVEN MODEL. Once the registration process is completed, the device is now able to communicate and function as intended.

The previous description already shows that the application of these patterns follows a certain logical order. Figure 3 makes this order explicit. At the start, we decide between the bootstrapping patterns. As REMOTE BOOTSTRAP requires the address of a bootstrap server it has to be used together with another

[4] https://en.wikipedia.org/wiki/Bootstrapping
[5] In contrast, a DEVICE SHADOW goes a step further than only offering the static metadata of on- or offline devices. It allows other components to interact with offline devices as if they were online, by storing a virtual copy of all the latest data they offered and all commands send to the device but not yet transmitted [Reinfurt et al. 2016, 2017a].

bootstrap pattern to supply this information. After that, we can choose one of the registrations patterns, and, during this process, assign one of the model types to the registering device. At the end, this information is stored in the DEVICE REGISTRY.

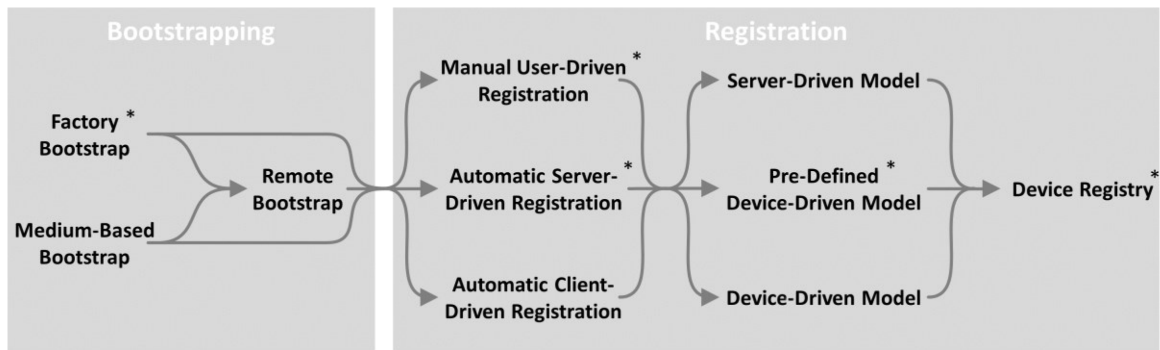

Figure 3. Graph of the pattern application order.

4. PATTERN FORMAT AND IDENTIFICATION PROCESS

The format used for the patterns in this paper is based on other existing pattern formats, as described in Section 2 and in our previous work [Reinfurt et al. 2016, 2017a], but with some small adjustments: The **Name** and the **Icon** provide a textual and visual way to identify the pattern, whereas other names under which it may be known are listed under **Aliases**. A short summary gives an overview of the pattern. The **Context** describes the circumstances in which a problem occurs. The **Problem** section describes the core of the problem that the pattern addresses. **Forces** list different aspects which have to be considered when choosing a solution to the problem. The **Solution** states the core steps which solve the problem considering the forces, while the **Solution Details** section adds more details and lists benefits and drawbacks. The **Variants** may contain variations of the pattern that do not warrant a separate pattern. **Related Patterns** are also listed to create interconnections between patterns which are used together or exclude each other. Finally, the **Known Uses** section (which was called *Examples* in our previous work) lists the sources of the pattern.

As described in more detail in our previous work [Reinfurt et al. 2016, 2017a], the patterns were identified and collected following the process of Fehling et. al [2015a]: We looked at product pages, user manuals, technical documentations, standards, whitepapers, and research papers of a sample of randomly selected IoT solutions for both businesses and consumers. We collected and categorized reoccurring solutions, which were then grouped by similarity into rough pattern candidates. We followed the rule of thumb described by Coplien [1996] and collected at least three different examples from different manufacturers per pattern before we authored the candidates into abstract patterns.

5. INTERNET OF THINGS PATTERNS FOR DEVICE BOOTSTRAPPING AND REGISTRATION

In this section, we present ten IoT Patterns for device bootstrapping and registration. Table 1 gives a short overview of all of them. The later subsections go into more detail on five of these patterns: MEDIUM-BASED BOOTSTRAP, REMOTE BOOTSTRAP, AUTOMATIC CLIENT-DRIVEN REGISTRATION, SERVER-DRIVEN MODEL, and DEVICE-DRIVEN MODEL.

Table 1: Overview of the presented patterns

Pattern Icon and Name	Short Pattern Description
Device Bootstrapping Patterns	
FACTORY BOOTSTRAP	The information a device requires to create the first connection to its communication partners is placed on the device during manufacturing.
MEDIUM-BASED BOOTSTRAP (p.7)	The information a device requires to create the first connection to its communication partners is placed on the device during deployment. A storage medium, such as a USB stick, is put into the device. The device uses the information stored on this medium to create the first connection.
REMOTE BOOTSTRAP (p.12)	The information a device requires to create the first connection to its communication partners is remotely sent to the device by a bootstrap server.
Device Registration Patterns	
AUTOMATIC CLIENT-DRIVEN REGISTRATION (p.15)	To make itself known to its communication partners, a device initiates a registration process: It calls a registration API on the backend server and supplies required information about itself. From this information, a device model instance is created and stored on the backend server for future reference.
AUTOMATIC SERVER-DRIVEN REGISTRATION	The backend server is informed about new devices by an out-of-band mechanism and initiates a registration procedure: It requests the required information from the devices. From the responses, it creates and stores a device model instance for future reference.
MANUAL USER-DRIVEN REGISTRATION	A user manually registers new devices using an API or GUI provided by the backend server. The user has to create a new or select an existing device model, create an instance of it and fill it with the metadata of the device. Once completed, the device can now connect to the backend server.
Device Model Patterns	
SERVER-DRIVEN MODEL (p.19)	Device models, which describe the attributes and functionalities of particular types of devices, are created and stored on the backend server. The backend server also assigns instances of these models to the respective devices.
PRE-DEFINED DEVICE-DRIVEN MODEL	Device models, which describe the attributes and functionalities of particular types of devices, are created and stored on the backend server. The devices themselves choose a model that fits them from the provided selection and fill an instance of it with their metadata.
DEVICE-DRIVEN MODEL (p.23)	Device models, which describe the attributes and functionalities of particular types of devices, are created and stored on the device itself. The device supplies this model instance to its communication partners.

Device Registry Pattern	
DEVICE REGISTRY	Device model instances of registered devices are permanently stored for future reference. This allows static device metadata to be retrieved even if the device is offline, and simplifies the management of registered devices.

5.1 Device Bootstrapping Patterns

As previously explained in Section 3, it is the aim of *Device Bootstrapping*, and, thus, also the aim of the following patterns, to get all information required to start communication onto the device. This *bootstrap information* may reach a device in different ways, which will be detailed in the following sections, which explain the MEDIUM-BASED BOOTSTRAP and the REMOTE BOOTSTRAP patterns in more detail. Figure 4 also puts these two patterns in the context of the other bootstrapping and registration patterns.

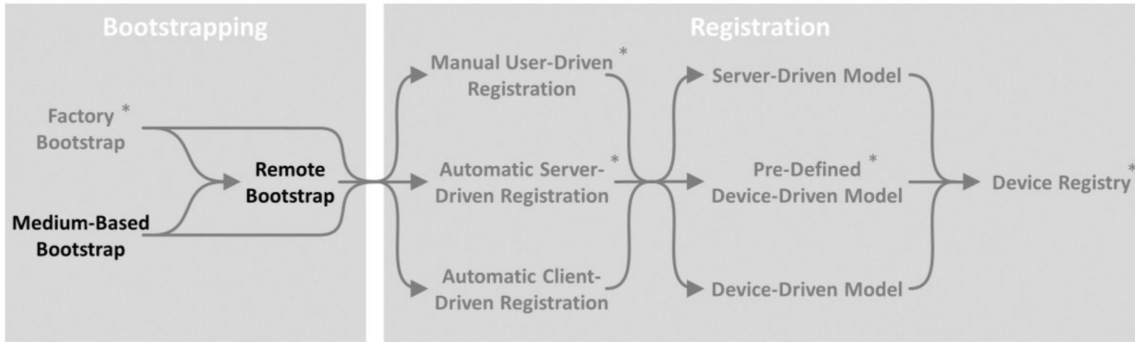

Figure 4. Context of the Bootstrapping patterns explained in Section 5.1.

5.1.1 MEDIUM-BASED BOOTSTRAP

The information a device requires to create the first connection to its communication partners is placed on the device during deployment. A storage medium, such as a USB stick, is put into the device. The device uses the information stored on this medium to create the first connection.

Context: You have a new device which has to communicate with other components to fulfill its purpose. The device is either connected directly to the other components, or it goes through a network and multiple intermediaries. Either way, the new device has to know how to create these connections.

Problem: A new device needs some basic information to be able to create connections and to start communication. This information may change over time. How do you get this information onto the device while keeping it as independent as possible?

Forces:

- **Independence:** The device should not depend on third parties for bootstrapping to avoid vendor lock-in.
- **Flexibility:** The details needed for first communication may change with time. You possibly need to adapt the device to these changes.
- **Choice:** You need a level of choice of how and to whom the device connects but baking this information into the device during production using FACTORY BOOTSTRAP means personalizing every device. It binds the device to this choice and makes it uninteresting for parties which have other needs.
- **Cost:** There may be solutions which offer more flexibility, but may require additional software or hardware components which could increase the overall cost of a device.
- **Security:** Security credentials for authentication, authorization, or encryption may be needed for communication, but these credentials have to be brought onto the device without an attacker being able to eavesdrop on them or alter them.
- **Scalability:** In some situations, large amounts of devices have to be bootstrapped, which can be a lot of work.
- **Physical Access:** Some devices may be hard to reach because they are set up in high places or at remote locations or rough terrain.
- **Resilience:** Some devices need to be built to withstand harsh conditions, but some components will decrease their ability to do so, for example, movable parts which break more easily or external ports which could let in water or dust.

Solution: Bootstrap, i.e., configure the device on-site from a replaceable storage medium, for example, a USB stick, that contains all necessary bootstrap information. When the device starts, let it read and use the information placed on this medium to start communication. Have the device copy its content for later use.

Solution Details: *Device Bootstrapping* is the process wherein the bootstrap information, i.e., information which a device needs to initiate communication with another component, is put on the device. MEDIUM-BASED BOOTSTRAP utilizes a removable storage medium to store bootstrap information, for example, in the form of a memory card or USB stick. So during manufacturing, no bootstrap information is placed on the device. Instead, the bootstrap information is put onto a bootstrap medium in a preparation step, as seen in Step 1 in Figure 5. This could be done by a communication service provider who then distributes these media to its customers, or by the device owner as preparation for the deployment. The medium is used as a *transport medium* to get the bootstrap information onto the device. An installation or maintenance worker inserts the storage medium into the device. The device reads the bootstrap information from the medium and stores it locally for future use (Step 2). The device should be able to identify, if a storage medium with different bootstrapping information as already saved on the device is inserted, and use this updated information to overwrite the old one. Afterwards, the worker removes the medium and may use it to bootstrap other devices. The device then gathers information on how to create a first connection from the bootstrap information copied from the medium and starts communication (Step 3).

The bootstrap information on the storage medium has to include an address for the device to contact, e.g., an IP address. Here, it also makes sense to provide multiple addresses, if possible, in case one of them is not reachable for some reason. If the communication has to be secure, the bootstrap information needs to store security credentials, for example, a pre-shared key or a certificate, or enable the device to get those credentials. Since this information is sensitive, it cannot be accessible to unauthorized parties. The device manufacturer has to take proper measurements to secure the channel between the device and the storage medium.

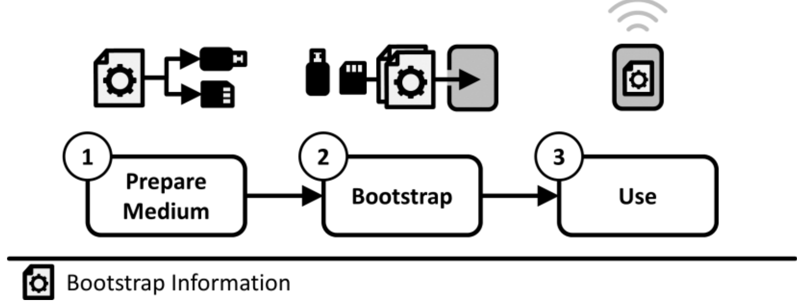

Figure 5. Sketch of the MEDIUM-BASED BOOTSTRAP pattern.

Benefits:

- **Independence:** MEDIUM-BASED BOOTSTRAP allows devices to be independent of any one organization. This simplifies device production, as one generic device is customizable with different storage media.
- **Flexibility:** The bootstrapping details can be updated by reinserting a storage medium. It can also be used to bootstrap different devices.
- **Choice:** Bootstrapping the device on-site during deployment allows you to choose how and to whom to connect the device until the last minute.
- **Security:** The bootstrap information is not transported over wired or wireless communication channels, where it could be copied or altered by an attacker.

Drawbacks:

- **Availability:** The device cannot create a connection if the target of the bootstrap information is temporarily not available. To avoid the device retrying to connect to this one target, provide a list of fallback targets.
- **Scalability:** Using a storage medium to bootstrap large numbers of devices does not scale. REMOTE BOOTSTRAP might be a better alternative in such situations.
- **Physical Access:** The device has to be physically accessible to be able to use a storage medium to bootstrap it. When the device's location is hard to reach, MEDIUM-BASED BOOTSTRAP should be done before the device is installed in its final location. REMOTE BOOTSTRAP is another alternative.
- **Inflexibility:** Having fixed bootstrap information is not suited for situations which need frequent changes to this information. One solution is to store a pointer to a bootstrap server for REMOTE BOOTSTRAP instead of the final communication partner. The bootstrap server dynamically manages the bootstrapping of devices, for example, to carry out load balancing.
- **Outdated Information:** The bootstrap information on the storage medium may become outdated with time. Updating the information is cumbersome: First, one has to know the storage media which contain outdated information. Next, one has to manually retrieve those media, update the information, and redistribute them to their original locations. One way to avoid this is by enabling Over-The-Air Updates to the bootstrap information. Outdated information is less of a problem when using REMOTE BOOTSTRAP.
- **Resilience:** Movable pieces are prone to fail in harsh conditions which involve vibration, shock, humidity, or extreme temperatures. This may make devices with a removable storage medium unusable for such purposes. Using FACTORY BOOTSTRAP or REMOTE BOOTSTRAP may be more suited in such situations.
- **Logistics:** You or a third party has to buy the storage media, fill it with the bootstrapping information, and bring it to the devices. This entails complex logistics and increases costs. REMOTE BOOTSTRAP simplifies this process.
- **Costs:** Devices need the appropriate hardware and software components to be able to use a storage medium for bootstrapping, which increases costs. Additionally, the manual labor required to get the medium to the device also adds to the costs. Designing the device so that these components can also be used for other tasks and integrating MEDIUM-BASED BOOTSTRAP into the deployment process helps to control these costs.
- **Security:** The storage medium could be stolen or copied. SYMMETRIC ENCRYPTION or ASYMMETRIC ENCRYPTION could be used so that the information on the device cannot be read by attackers without the proper key [Fernandez 2013].
- **Trust:** Any medium could be inserted into the device, but the device has to know if it can trust the information contained on the medium. Using a DIGITAL SIGNATURE WITH HASHING is one solution to prove that the content on the medium is genuine and was not tempered with [Fernandez 2013].

Variants:

- **IDENTIFICATION-MEDIUM-BASED BOOTSTRAP:** This variant is similar to MEDIUM-BASED BOOTSTRAP in that a storage medium with bootstrapping information is placed inside the device, as shown in Step 2 in Figure 6. But here the storage medium is used in a more permanent fashion as an *identification medium* for the device, similar to SIM cards used for mobile phones. It gives the device an identity and associated access rights as long as it stays in the device. The device does not save the bootstrap information (Step 3). It has to be aware if you remove the storage medium or change the information on it. In this case, the device rereads the information from the medium every time before it uses it, i.e., before it executes update and registration operations.

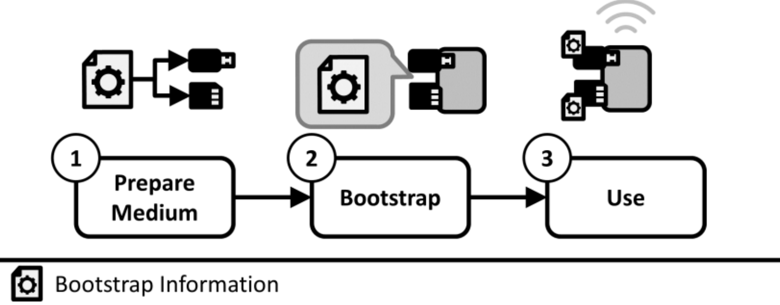

Figure 6. Sketch of the IDENTIFICATION-MEDIUM-BASED BOOTSTRAP variant.

Related Patterns:

- **FACTORY BOOTSTRAP:** For situations where you need rugged devices, FACTORY BOOTSTRAP is more suited.
- **REMOTE BOOTSTRAP:** Pointing the device to a bootstrap server instead of its final communication partner increases flexibility and resilience.
- **AUTOMATIC DEVICE-DRIVEN REGISTRATION, AUTOMATIC SERVER-DRIVEN REGISTRATION, and MANUAL USER-DRIVEN REGISTRATION:** After an MEDIUM-BASED BOOTSTRAP, the next step is often registering the device with one of these registration patterns.

Known Uses: Bootstrapping via removable medium is familiar to average consumers in the form of Subscriber Identification Module (SIM) cards put in mobile phones. But this form of bootstrapping is not restricted to SIMs and mobile phones. For example, Gemalto offers other types of media, such as Machine Identification Modules (MIMs), to bootstrap other devices [Gemalto 2016b]. OMA LWM2M defines a bootstrapping interface which includes bootstrapping from a Smartcard. If necessary, it uses a secure channel between the device and the smart card to protect the bootstrapping information. If you insert such a smart card, the client has to use it as a primary bootstrapping source. Before each register or update operation, the client has to check if you removed the smart card or changed the information on it. Afterward, the device has to update or delete the information it has stored to stay consistent [Open Mobile Alliance 2015]. The interface is able to remotely update the bootstrapping information stored on the smartcard [3GGP 2003]. Many products offer the ability to update the software on the product (and, thus, the bootstrap information) by exchanging a USB stick or SD-Card. Examples include Intel's Galileo board [Intel 2017], the Raspberry Pi [Raspberry Pi Foundation 2017], or Devialet's Phantom connected speaker [Devialet 2016].

5.1.2 REMOTE BOOTSTRAP

The information a device requires to create the first connection to its communication partners is remotely sent to the device by a bootstrap server.

Context: You have a new device which has to communicate with other components to fulfill its purpose. The device is either connected directly to the other components, or it goes through a network and multiple intermediaries. Either way, the new device has to know how to create these connections. The device is located in a remote location where it is hard to reach for maintenance.

Problem: A new device needs some basic information to be able to create connections and to start communication. This information is not the same for all devices and it may change from time to time. How do you get this information onto the device while retaining flexibility and allowing for a robust construction, all while the device is hard to reach?

Forces:

- **Security:** Security credentials for authentication, authorization, or encryption may be needed for communication but these credentials have to be brought onto the device and have to be secure themselves.
- **Simplicity:** The device has to work without any further actions required by its owner. For example, the end-user does not have the ability to do any required setup or a company wants to install a large number of devices.
- **Size or Cost Constraints:** The device's design, i.e., its form factor and the number of components, has to be small and simple because it needs to fit into size or cost limitations. Adding components just for bootstrapping, such as a memory card slot or USB connector for MEDIUM-BASED BOOTSTRAP may not make sense.
- **Robustness:** You intend to use the device in harsh environments and you need it to be durable. For example, you want to seal it to be water tight, or you want to put into a rugged enclosure which is intentionally hard to access to prevent any adverse effects from the outside. But this limits access to the device for legitimate maintenance purposes.
- **Flexibility:** The details needed for first communication change with time. You need to adapt the device to these changes.
- **Physical Access:** You want to place the device in a location that is hard to reach, which makes it difficult or dangerous to do a MEDIUM-BASED BOOTSTRAP.
- **Scalability:** You have to set up communication for a large number of devices, which costs time and resources if done manually.

Solution: Store the bootstrapping information on a bootstrap server. Provide the device with details on how to get to this server by FACTORY BOOTSTRAP or MEDIUM-BASED BOOTSTRAP or have the server informed about new devices. Download the bootstrap information from the bootstrap server onto the device. Use the bootstrap information to start communication.

Solution Details: *Device Bootstrapping* is the process wherein the bootstrap information, i.e., information which a device needs to initiate communication with another component, is put on the device. In the case of a REMOTE BOOTSTRAP, no bootstrap information for its final communication partner is initially placed on the device. The device has either no bootstrap information or has the contact information for a bootstrapping server stored.

In the first case, the bootstrap server has to execute a server-initiated REMOTE BOOTSTRAP. This requires a mechanism which informs the bootstrap server of new devices. This happens through an out-of-band communication channel and allows the server to directly connect to the devices to bootstrap them, as shown in Step 2 in Figure 7.

In the second case, the device is able to execute a client-initiated REMOTE BOOTSTRAP. Here, the device uses the stored information (from a FACTORY BOOTSTRAP or MEDIUM-BASED BOOTSTRAP) to connect to a bootstrap server and requests a bootstrap (Step 1). In this case, it makes sense to provide the device with multiple bootstrap servers to select from, in case one of them is not reachable for some reason. In either case, the bootstrap server now sends bootstrap information to the device, as shown in Step 2 in Figure 7, which the device uses to start normal communication (Step 3).

The bootstrap information has to include an address for the device to contact, e.g., an IP address. Here, it also makes sense to provide multiple addresses, if possible, in case one of them is not reachable for some reason. If the communication has to be secure, the information needs to store security credentials, for example, a pre-shared key or a certificate, or enable the device to get those credentials. Since this information is sensitive, it cannot be accessible to unauthorized parties. The person or organization responsible for the bootstrapping infrastructure has to take proper measurements to secure the communication channel between the device and the server.

The bootstrap server is able to offer extra functionality such as security or load-balancing features. For example, the bootstrap server may randomly assign new devices to backend servers to spread the load evenly across them. Alternative assignment methods may be implemented which use the device type or any other metadata.

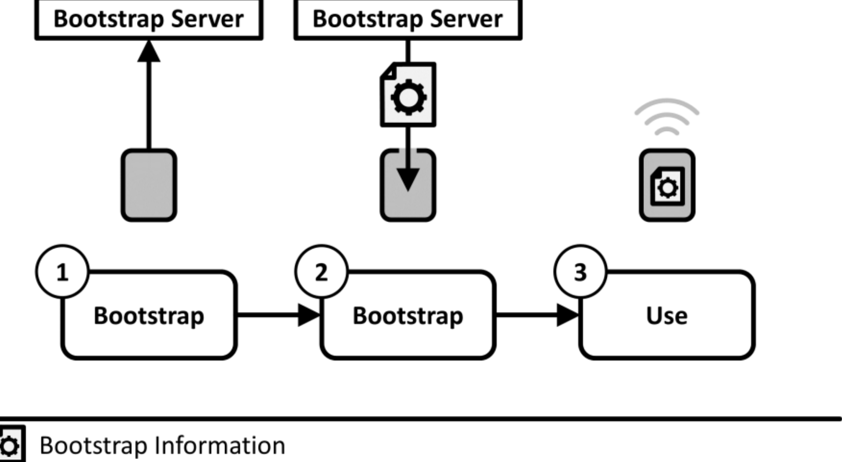

Figure 7. Sketch of the REMOTE BOOTSTRAP pattern.

Benefits:

- **Flexibility:** You are able to alter the bootstrap information if required by changing circumstances. The information is sent to the device just-in-time when it starts.
- **Genericity:** The devices stays generic in terms of their communication information until it is remotely bootstrapped. In cases where the bootstrap server belongs to one organization the device stays generic in the boundaries of this organization and could be used everywhere inside this boundary. In cases where a third party provides the bootstrapping server, the devices are generic even across organizations until remotely bootstrapped.
- **Robustness:** Movable parts, such as cards and card slots, etc., are prone to failure. When the bootstrap information is sent remotely after the manufacturing process, there do not have to be movable parts. Access to the device enclosure is not required. This allows the manufacturer to seal or ruggedize devices to make them water- or dust-resistant or resilient against vibrations and shock.
- **Simplified Logistics:** The bootstrap information does not have to be available during production and it is not required to handle physical media. This simplifies logistics.
- **Scalability:** A large number of devices can be bootstrapped with little effort.
- **Physical Access:** No physical access is required to the device in order to bootstrap it.
- **Size Constraints:** The device does not need additional hardware components, so the size can be kept small.
- **Simplicity:** Bootstrapping happens automatically without further action required by users.
- **Added Functionality:** The bootstrap server is able to offer extra functionality, such as security or load-balancing features, for which normally a separate server would be used. For example, by first returning the addresses of the most underutilized backend servers to a device, the bootstrap server acts as a load balancer who ensures that devices are registered evenly across the backend servers.

Drawbacks:

- **Cost/Effort:** REMOTE BOOTSTRAP requires an infrastructure with bootstrapping servers to work. Each device operator providing bootstrapping infrastructure for themselves means added cost and effort. Using a third party for bootstrapping services allows this third party to pool resources and work more efficiently.
- **Dependence:** REMOTE BOOTSTRAP needs infrastructure which is able to send bootstrap information to the devices. This infrastructure is either self-hosted or run by a third party. Regardless, infrastructure fails and companies go out of business. Being able to use MEDIUM-BASED BOOTSTRAP as a backup is a solution in such cases.
- **Initiation:** For client-initiated REMOTE BOOTSTRAP the information required to contact the bootstrap server has to be brought onto the device. This can be done during manufacturing with FACTORY BOOTSTRAP or with MEDIUM-BASED BOOTSTRAP, which would place the generic contact information of the bootstrap server on the device. The bootstrap server would then provide the specific bootstrapping information to the device.
- **Security:** The bootstrapping information is send remotely to the device which makes it a potential target for hacking attempts. Use SYMMETRIC ENCRYPTION or ASYMMETRIC ENCRYPTION to keep the information private and DIGITAL SIGNATURE WITH HASHING to proof it is genuine and has not been tampered with [Fernandez 2013].

Related Patterns:

- **MEDIUM-BASED BOOTSTRAP:** Consider MEDIUM-BASED BOOTSTRAP as a backup solution if REMOTE BOOTSTRAP is not working. It could also be used to get the information required by the device to contact the bootstrap server.
- **FACTORY BOOTSTRAP:** To get the initial information required to contact the bootstrap server from the device, FACTORY BOOTSTRAP is one option.

- **AUTOMATIC DEVICE-DRIVEN REGISTRATION, AUTOMATIC SERVER-DRIVEN REGISTRATION, and MANUAL USER-DRIVEN REGISTRATION:** After a REMOTE BOOTSTRAP, the next step is often registering the device with one of these registration patterns.

Known Uses: OMA LWM2M defines a bootstrapping interface which offers client- or server-initiated remote bootstrapping. A pre-provisioned remote bootstrapping server stores the necessary information and configures clients by calling their write or delete functionality. [Open Mobile Alliance 2015]. The Kaa IoT platform uses one or multiple bootstrap services. A Software Development Kit (SDK) automatically provisions devices with a list of bootstrap service URIs during development. The device randomly contacts one of these to get a list of available operations services. The bootstrap service selects and orders this list by using different load-balancing strategies [CyberVision 2016b]. Gemalto offers an On-Demand Provisioning Service (OPS) for their UICCs. When a user activates a device for the first time, it automatically connects to the OPS platform. The OPS downloads a customized provider profile to the device to finish bootstrapping [Gemalto 2016a].

5.2 Device Registration Patterns

After applying one of the patterns described in the previous section to get the initial information required for communication onto the devices, they are now ready to communicate. But as described earlier in Section 3, this may not be the only step required before the devices are ready to work as intended. Often, the next step is that they register themselves at a backend server, which we called *Device Registration*. This step allows them to be linked to user or company accounts and to a virtual representation of themselves, an instance of a *Device Model*. The following section describes one of the registration patterns, AUTOMATIC CLIENT-DRIVEN REGISTRATION, in more detail. Figure 8 puts this patterns into context compared to the other bootstrapping and registration patterns.

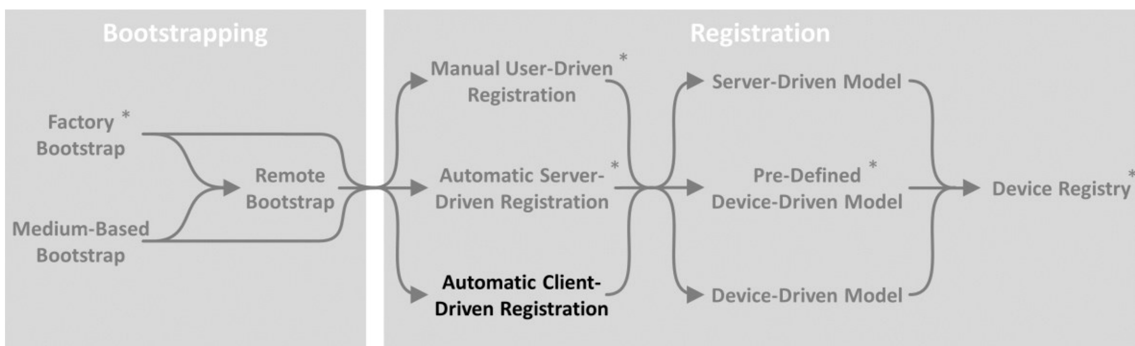

Figure 8. Context of the Bootstrapping patterns explained in Section 5.2.

5.2.1 AUTOMATIC CLIENT-DRIVEN REGISTRATION

To make itself known to its communication partners, a device initiates a registration process: It calls a registration API on the backend server and supplies required information about itself. From this information, a device model instance is created and stored on the backend server for future reference.

Aliases: Self-registering Devices

Context: Some devices should be connected to a backend server. Often, the backend server has to allocate resources for each device, for example, communication queues or storage space for stored events.

Problem: Usually, many devices will be connected to one backend server, but for security or privacy reasons and to prevent misusage not any device should be able to connect at will. Some information about the authorized devices has to be known to the backend in order to be able to contact and manage them. How can this be done if the devices are autonomous and very dynamic?

Forces:

- **Scalability:** In some cases, a large number of devices might be intended to connect to one backend server. Allocating the necessary resources and managing the registration by hand would involve a lot of work.
- **Dynamics:** Devices might move between locations and backend servers in a very dynamic fashion. Managing these changing associations manually would be a lot of work or even impossible in highly dynamic scenarios.
- **Security:** Not every device should be allowed to connect to a specific platform. There have to be some rules that govern which devices are allowed.

Solution: Allow unknown devices to register themselves at the backend server via an API. The devices should provide at least a minimal set of metadata in the API call for successful registration. This should include the device id, details on how to communicate with the device, and some kind of authentication information, such as a token. Check the metadata to prevent illegitimate registration attempts, for example by comparing metadata against a list of allowed values or by validating the authentication information. If successful, place this metadata in a DEVICE REGISTRY so that the device can be identified later on.

Solution Details: *Device Registration* is the process of making a device known to its communication partners, and often binding it to a specific user or company account, as well as its virtual representation, the device model. AUTOMATIC CLIENT-DRIVEN REGISTRATION allows devices to register themselves with a backend server that they didn't have any contact before without human intervention. For this to work, the device has to know the details of how it can contact the backend server and how it can invoke the registration process. These details have to be given to the device during the bootstrapping process, for example by doing FACTORY BOOTSTRAP, MEDIUM-BASED BOOTSTRAP, or REMOTE BOOTSTRAP.

Once the device wants to register with a backend server, e.g., when it is started and has no associated backend or cannot connect to an associated backend because it is not yet registered, as shown in Step 1 in Figure 9, it contacts the backend server and invokes the registration process as specified in the bootstrapping information (Step 2). This usually involves passing along some arguments in a registration API call. A minimal set of arguments for successful registration has to be defined so that normal operation is possible if only these arguments are provided during the registration process. This set usually includes a device ID, the device communication details, such as the IP address, and some kind of authentication and authorization information. Tokens[6] are a common way to provide this information.

The backend server should now check if the provided information satisfies the minimal requirements. If they do and if the authentication and authorization information are valid, the device metadata is added to the list of registered devices in the DEVICE REGISTRY (Step 3). Additional checks can be implemented with rules or policies to provide fine-grained control over connecting devices. The backend server now also can set up and allocate all the resources which it needs to properly integrate the device, such as communication queues or other storage, etc. Now, the registration process is complete and the device is able to access the full functionality of the platform (Step 4).

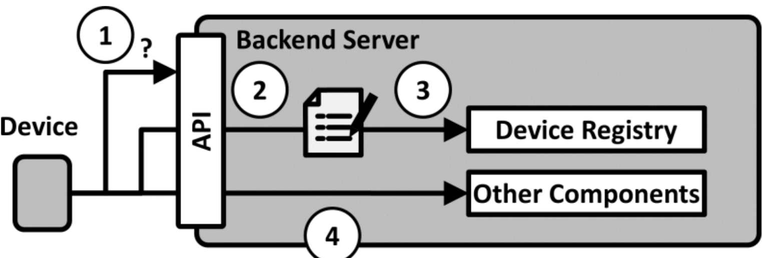

Figure 9. Sketch of the AUTOMATIC CLIENT-DRIVEN REGISTRATION pattern.

Benefits:

- **Scalability:** The registration process is automated and does not require human intervention. This allows a large number of devices to be registered very fast.
- **Security:** Checking the provided registration data allows the backend server to filter out unwanted requests and prevents unnecessary resource allocation.
- **Dynamics:** Devices can be dynamic and move between different locations and backend servers on their own without the need for manual work.

Drawbacks:

- **Reachability:** The registration service might not be reachable due to some unforeseen event, in which case the registration attempts will fail. To prevent such cases, a list of multiple registration servers could be given to the device so that it can try to find a working one. Another option is to provide a highly available registration service with automatic failover.
- **Out-of-Date:** The connection information for registration stored on the device might be out of date once the device tries to register. For example, the information was put on the device during manufacturing with FACTORY BOOTSTRAP, but the device is first started a few months later when the information is already outdated. In such cases, it should be possible to manually register the device so that it can connect to the backend, for example by using a MANUAL USER-DRIVEN

[6] Tokens are used to store and pass along information about an entity in a tamper-proof way. The information usually includes an ID and some claims about the entity but may also include additional information. A signature of this information is also part of the token. By applying a cryptographic algorithm to the information and comparing the result to the signature, the recipient of the token is able to validate the claims.

REGISTRATION process. Once connected, the outdated information should be detected and updated automatically by the backend.

- **Dynamics:** In highly dynamic environments, devices may only register for a short time of activity, then go offline and seldom or never reconnect. Storing all these obsolete registrations may become a problem over time. Thus, registrations should be removed after a certain amount of time without any activity has passed.

Related Patterns:

- **FACTORY BOOTSTRAP, MEDIUM-BASED BOOTSTRAP, and REMOTE BOOTSTRAP:** The device needs some initial information to be able to communicate and register with a backend server. This information may be provided with FACTORY BOOTSTRAP, MEDIUM-BASED BOOTSTRAP, or REMOTE BOOTSTRAP.
- **SERVER-DRIVEN MODEL, PRE-DEFINE DEVICE-DRIVEN MODEL, and DEVICE-DRIVEN MODEL:** An instance of one of these device model types is created and filled with metadata during the AUTOMATIC CLIENT-DRIVEN REGISTRATION process.
- **DEVICE REGISTRY:** The device model instance created during the AUTOMATIC CLIENT-DRIVEN REGISTRATION process is stored in a DEVICE REGISTRY. Other components can query this registry to get an overview of or static metadata about registered devices.
- **DEVICE WAKEUP TRIGGER:** A device has to be registered so that its metadata is available when it is offline and can be used for a DEVICE WAKEUP TRIGGER[7] [REINFURT ET AL. 2016].

Known Uses: OMA LWM2M allows clients to register with one or more servers through the client registration interface. The client has to provide the server with information that allows it to contact the client and maintain the registration. The client also has to periodically execute an update operation, or otherwise, the server will remove the registration [Open Mobile Alliance 2015]. SiteWhere allows devices to self-register by sending a unique hardware id and a specification token. If the device registers for the first time, a new device record will be created, otherwise, an existing record will be selected by SiteWhere [SiteWhere 2015]. With the Kaa IoT middleware, devices use the endpoint registration process to send their DEVICE-DRIVEN MODEL along with security credentials to the Kaa cluster for initial registration. The device itself does not know the address of the servers where it will be registered. Instead, it has been given a list of bootstrapping servers during development, which will then use load-balancing strategies to handle the registration of the device to a specific server [CyberVision 2016b]. Autodesk SeeControl also has an auto-provisioning mechanism where devices can register themselves by sending messages to the platform and passing rules and policy checks [Autodesk 2016].

[7] A DEVICE WAKEUP TRIGGER uses a secondary low-power communication channel to tell a currently offline device to enable its main communication channel and reconnect to the backend server [Reinfurt et al. 2016, 2017a].

5.3 Device Model Patterns

In Section 5.1 we have seen, how different *Device Bootstrapping* patterns may be used to bring the information required to start communication onto devices. In Section 5.2 we then argued, that this may not be sufficient to allow a new device to fully function as intended, and that devices may have to register with a backend server beforehand by taking part in a *Device Registration* procedure. During this procedure, an instance of a *Device Model*, a virtual representation of the device, is stored in a *Device Registry*. Such a model instance contains various metadata about a registered device, which allows other components to search a backend server for and interact with registered devices based on this data. The following patterns describe two different kinds of *Device Models* which may be used in more detail, namely the SERVER-DRIVEN MODEL and the DEVICE-DRIVEN MODEL. Figure 10 puts these pattern into the context of the other bootstrapping and registration patterns.

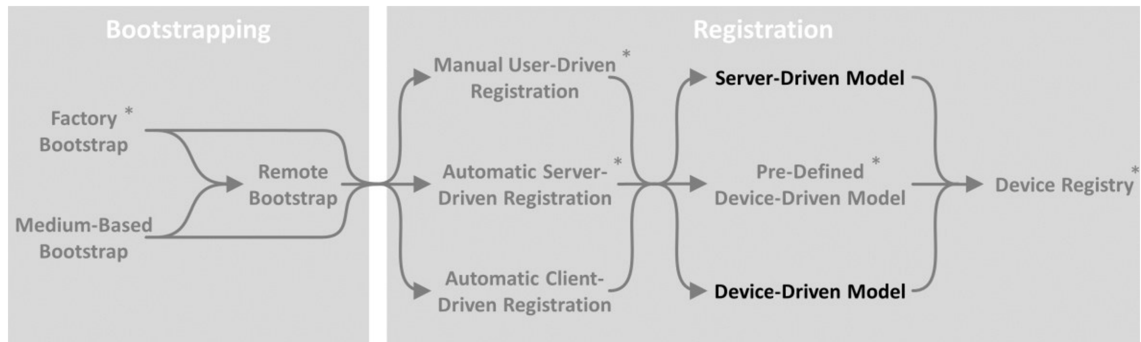

Figure 10. Context of the Bootstrapping patterns explained in Section 5.3.

5.3.1 SERVER-DRIVEN MODEL

Device models, which describe the attributes and functionalities of particular types of devices, are created and stored on the backend server. The backend server also assigns instances of these models to the respective devices.

Aliases: Service-Driven Model

Context: You have devices that each offer a specific set of functionality to other components, such as operations that can be invoked on them (for example, operating an actuator, or restarting the device) or information that can be retrieved from them (for example, the most recent values of a sensor). Each device is also described by a set of metadata, which could include its name, type, id, manufacturer, etc. You have other components, like other devices, a backend server, an application, etc., that need to interact with these devices.

Problem: Other components need to know the data and functionality which a specific device provides so that they can interact with it. But not all devices come with the ability to provide or store this information.

Forces:

- **Compatibility:** For other components to be able to interact with a device they have to somehow have a common understanding of the available functionality.
- **Unmodeled Devices:** Not all devices will come with a DEVICE-DRIVEN MODEL or have the ability to store such a model, or a pointer to a PRE-DEFINED DEVICE-DRIVEN MODEL. Especially legacy devices or very constrained devices might not be able to store and provide such information themselves. It should be possible to also integrate these devices.
- **Flexibility:** The abilities of a device might change through firmware updates. Other components have to be made aware of and adapt to these changes. This will be hard if changes have to be made on all other components, especially if changes have to be made by hand.
- **Diversity:** Many different kinds of devices with different functionality exist. Storing and managing descriptions of all of these devices centrally might be very hard.

Solution: Create a device model that describes the device and its functionality. Store it in a model database on the backend server. Assign a model instance to the device on the backend server during registration. Store the device's model instance in a DEVICE REGISTRY.

Solution Details: A *Device Model* is a virtual representation of the device, which contains various metadata about a registered device that allows other components to search a backend server for and interact with registered devices based on this data. SERVER-DRIVEN MODELS are device models that are stored and assigned to a device on the server side. They are usually created before or during the AUTOMATIC SERVER-DRIVEN REGISTRATION or MANUAL USER-DRIVEN REGISTRATION process by an administrator using a graphical user interface or an API, as shown in Step 1 in Figure 11. They can be an alteration or combination of other already existing models, which allows for some reusability and higher efficiency.

They must at least contain some kind of identifier of the device so that they can be linked to the device. They usually also contain additional static metadata, such as manufacturer, owner, version numbers, creation date, etc., which allow the models, and therefore the connected devices, to be filtered by various criteria. More complex models might describe attributes of the device that can be read or written by other components. For example, the location attribute of the device might be updated by the device itself using GPS and could be only readable for other components. For a device that does not have GPS, the location attribute could be set to writable for other components so that the location can be updated manually. The model could also describe the operations that other components might execute on the device, how they can be executed, i.e., necessary parameters, and what can be expected as a result, i.e., possible return values.

Once a registration process is started, which could either be AUTOMATIC CLIENT-DRIVEN REGISTRATION, AUTOMATIC SERVER-DRIVEN REGISTRATION, or MANUAL USER-DRIVEN REGISTRATION, the backend server fetches a matching device model from the device model database, as shown in Step 2 in Figure 11, and assigns it to the device (Step 3). This could either be done based on an assignment made by an administrator or based on some rules previously defined, which associate some metadata available about the device with a specific device model. The concrete device model instance assigned to the device is usually stored in a DEVICE REGISTRY on the backend server, as shown in Step 4. Other components can query the DEVICE REGISTRY to get an overview of all registered devices or retrieve the metadata of

specific devices they are interested in (Step 5). The devices themselves have no knowledge of the models at all. As a result, a SERVER-DRIVEN MODEL is the only instance of the device metadata and its single source of truth. In this case, all components that want to access this data have to go through the DEVICE REGISTRY.

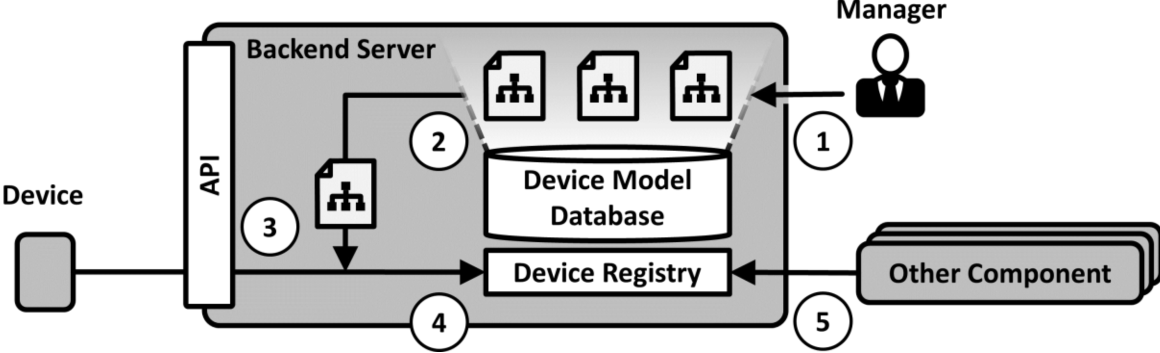

Figure 11. Sketch of the SERVER-DRIVEN MODEL pattern.

Benefits:

- **Compatibility:** A Server-Driven Model provides information about what data and functionality a device provides, so that other components can properly interact with the device.
- **Unmodeled Devices**: SERVER-DRIVEN MODELS are able to provide device models for devices which are otherwise not able to provide or store their own device model.
- **Flexibility**: When a device and therefor its device model changes these changes can be distributed centrally from the backend server.
- **Efficiency:** Because SERVER-DRIVEN MODELS are stored at one central place it is usually easy to make their creation and usage more efficient, for example by providing templates, reusing existing models, or allowing composition of multiple models.
- **Management:** Being stored in one central place can make the management of SERVER-DRIVEN MODELS easier than that of DEVICE-DRIVEN MODELS. The latter ones would have to be changed on the devices where they originated, while the former ones can all be managed in one place. This also allows manufacturers to easily provide new device models.

Drawbacks:

- **Choice:** Device users or developers do not have a choice about which model the server assigns to the device. The server may assign a wrong model, which may alter or impede functionality. If more choice is required, PRE-DEFINED DEVICE-DRIVEN MODELS or DEVICE-DRIVEN MODELS have to be used.
- **Diversity:** Providing SERVER-DRIVEN MODELS for all kinds of different devices is a lot of work and may be unrealistic. In situations where the number of device types is small and does not change often, this may not be a problem. If a lot of devices are involved and the situation changes frequently, DEVICE-DRIVEN MODELS may be a better choice. They can also be combined with SERVER-DRIVEN MODELS so that the SERVER-DRIVEN MODELS act as backup for devices without a DEVICE-DRIVEN MODEL.
- **Static Information:** A SERVER-DRIVEN MODEL offers only static metadata about the device it describes. If other components have to interact with offline devices based on dynamic information, a DEVICE SHADOW has to be used [Reinfurt et al. 2016].

Related Patterns:

- **AUTOMATIC DEVICE-DRIVEN REGISTRATION, AUTOMATIC SERVER-DRIVEN REGISTRATION, and MANUAL USER-DRIVEN REGISTRATION:** SERVER-DRIVEN MODELS can be assigned to devices during all of these registration options.
- **PRE-DEFINED DEVICE-DRIVEN MODEL:** PRE-DEFINED DEVICE-DRIVEN MODELS are similar to SERVER-DRIVEN MODELS in that they are defined and stored on the backend server. But unlike SERVER-DRIVEN MODELS, PRE-DEFINED DEVICE-DRIVEN MODELS are selected by the device, or respectively, its developer. This is useful for situations where device developers or users do not have access to the backend server.
- **DEVICE-DRIVEN MODEL:** SERVER-DRIVEN MODELS may be used as a backup for situations where devices do not or are not able to provide a DEVICE-DRIVEN MODEL.
- **DEVICE REGISTRY:** The actual instances of SERVER-DRIVEN MODELS created during the registration process are stored in a DEVICE REGISTRY. If other components want to retrieve the static metadata of a particular device or get an overview of registered devices, they can query the DEVICE REGISTRY.
- **DEVICE SHADOW:** A DEVICE SHADOW[5] offers other components access to the last known state of offline devices and allows them to send commands to these devices. The state and commands will be synced once the offline device reconnects [Reinfurt et al. 2016]. Thus, it can offer more dynamic information than device models stored in a DEVICE REGISTRY.

Known Uses: IBM's Internet of Things Foundation service stores device models, which describe metadata and management characteristics of devices, in a device database. This database is the master source of device information. Device type templates are used to pre-define attributes that can then be overridden by device-specific attributes later on [IBM 2015b]. These device models and types can be created using a web dashboard or a REST API [IBM 2015a]. Bosch's IoT Suite has an M2M component that uses information models to describe devices. Models can be reused for devices of the same type and can also be combined to describe complex systems [Bosch Software Innovations 2015]. SiteWhere allows administrators to manage device specification models on the backend using the admin UI or a REST interface. These specifications contain a list of commands that can be invoked by SiteWhere on the devices [SiteWhere 2015]. The Ayla IoT Cloud stores virtual device definitions in the cloud [Ayla Networks 2015].

5.3.2 DEVICE-DRIVEN MODEL

Device models, which describe the attributes and functionalities of particular types of devices, are created and stored on the device itself. The device supplies this model instance to its communication partners.

Aliases: Client-side Endpoint Profile

Context: You have devices that each offers a specific set of information and operations to other components. You have other components, like other devices, a backend server, an application, etc., that need to interact with these devices. To be able to interact they need to know what functionality each device offers.

Problem: Other components need to know the data and functionality which a specific device provides so that they can interact with it. But some use cases have a high diversity of devices and may work in a decentralized fashion.

Forces:

- **Compatibility:** For other components to be able to interact with a device they have to somehow have a common understanding of the available functionality.
- **Flexibility:** The abilities of a device might change through firmware updates. Other components have to be made aware of and adapt to these changes. This will be hard if changes have to be made on all other components, especially if changes have to be made by hand.
- **Diversity:** Many different kinds of device with different functionality exist. Storing and managing descriptions of all of these devices centrally might be very hard.
- **Decentralization:** Devices may want to communicate in a decentralized fashion, without any central component involved. Thus, using SERVER-DRIVEN MODELS or PRE-DEFINED DEVICE-DRIVEN MODELS would not be possible.

Solution: Let the developers of a device themselves define a model of the device. The model contains the device properties that can be read or written and the device commands that can be executed. Store the model on the device and make it available to other components that want to interact with the device.

Solution Details: A *Device Model* is a virtual representation of the device, which contains various metadata about a registered device that allows other components to search for and interact with registered devices based on this data. A DEVICE-DRIVEN MODEL is usually defined by the developer of the device as part of the device creation process and stored on the device, as shown in Step 1 in Figure 12. During this process, the developer adds metadata to the device model that describes the device and its capabilities. This metadata might include general metadata, like the device name, manufacturer, ver-

sion numbers, manufacturing date, etc. It also includes a description of the properties that other components can access on the device. These properties might be readable or writable. Additionally, operations that can be executed by other components on the device are also described.

The model then has to be shared with other components that want to interact with the device. This could be done during the initial registration process, which could either be AUTOMATIC CLIENT-DRIVEN REGISTRATION, AUTOMATIC SERVER-DRIVEN REGISTRATION, or MANUAL USER-DRIVEN REGISTRATION (Step 2a). DEVICE-DRIVEN MODELS may also be communicated directly to other device or components if no central backend server is involved in the system (Step 2b). To be able to understand the format of the model, both the device and the other components have to use a common model format. This format could be agreed on beforehand. To allow for more flexibility, the device could offer its model in various different formats, and components could be able to interpret various different formats. A format negotiation mechanism could then be used to agree on a mutually supported model format. In some cases, it might be necessary to use an additional model translator to reach a common format. During the registration process with a central backend server, the model is usually stored in a DEVICE REGISTRY (Step 3), from where it may then be accessed by other components.

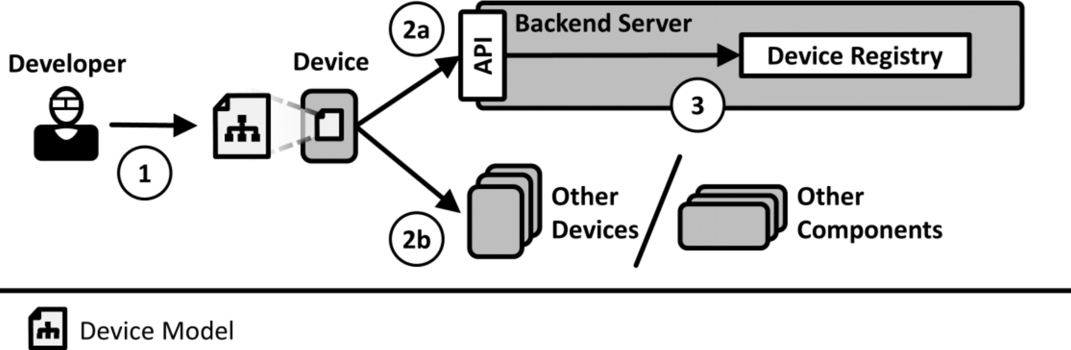

Figure 12. Sketch of the DEVICE-DRIVEN MODEL pattern.

Benefits:

- **Compatibility:** A DEVICE-DRIVEN MODEL provides information about what data and functionality a device provides, so that other components can properly interact with the device.
- **Flexibility:** The device developer has great flexibility in designing the device.
- **Diversity:** Each device can have its own specific model and does not have to be constricted to some more generic model that might not fully describe the device.
- **Decentralization:** No central component has to be involved, as all devices provide their own device model to other components.

Drawbacks:

- **Complexity:** A large number of devices, each with their own model, might make implementation very complex. A selection of PRE-DEFINED DEVICE-DRIVEN MODELS could be offered for common devices. If none of these fit a particular device, a DEVICE-DRIVEN MODEL could still be used.
- **Manageability:** The device model is stored on the device. The device has to be accessed each time a change to the model is made. REMOTE DEVICE MANAGEMENT could be used to make this process easier [Reinfurt et al. 2017a].
- **Availability:** Not all devices may be able to offer a DEVICE-DRIVEN MODEL. PRE-DEFINED DEVICE-DRIVEN MODELS or SERVER-DRIVEN MODELS may be used as backup solutions for these devices.

- **Static Information:** A DEVICE-DRIVEN MODEL offers only static metadata about the device it describes. If other components have to interact with offline devices based on dynamic information, a DEVICE SHADOW has to be used [Reinfurt et al. 2016].
- **Unmodeled Devices:** There may be some device which are not able to provide or store a DEVICE-DRIVEN MODEL, for example, because of very limited storage space. PRE-DEFINED DEVICE-DRIVEN MODELS or SERVER-DRIVEN MODELS have to be used for these devices.

Related Patterns:

- **AUTOMATIC DEVICE-DRIVEN REGISTRATION, AUTOMATIC SERVER-DRIVEN REGISTRATION, and MANUAL USER-DRIVEN REGISTRATION:** DEVICE-DRIVEN MODELS can be used in all of these registration options.
- **PRE-DEFINED DEVICE-DRIVEN MODEL:** DEVICE-DRIVEN MODELS could be combined with PRE-DEFINED DEVICE-DRIVEN MODELS to offer developers a basic selection of pre-defined models. If one of those models already fits the device, a developer could just use the pre-defined model instead of defining another one. If no pre-defined model satisfies the developer's requirements he could still define his own model.
- **SERVER-DRIVEN MODEL:** DEVICE-DRIVEN MODELS could be combined with SERVER-DRIVEN MODELS for cases where a device does not provide its own model. The service could assign a pre-defined model to that device.
- **DEVICE REGISTRY:** The actual instances of DEVICE-DRIVEN MODELS created during the registration process are stored in a DEVICE REGISTRY. If other components want to retrieve the static metadata of a particular device or get an overview of registered devices, they can query the DEVICE REGISTRY.
- **DEVICE SHADOW:** A DEVICE SHADOW[5] offers other components access to the last known state of offline devices and allows them to send commands to these devices. The state and commands will be synced once the offline device reconnects [Reinfurt et al. 2016]. Thus, it can offer more dynamic information than device models stored in a DEVICE REGISTRY.
- **REMOTE DEVICE MANAGEMENT:** Device models may change from time to time. REMOTE DEVICE MANAGEMENT can be used to apply these changes remotely to the device [Reinfurt et al. 2017a].

Known Uses: To integrate a device into the Kii IoT Platform, developers have to define a schema which describes the commands, i.e., actions and action results, and states, i.e. the attributes that define the device. This schema is not managed by the platform, but registered at the platform and used by other devices or mobile applications to identify actions they can execute [Kii 2015]. The Kaa IoT middleware uses client-side endpoint profiles that are created during development to describe device characteristics. These profiles are unidirectionally synchronized with the Kaa cluster. Updated profiles are detected on the client by comparing hashes and send to the server if necessary [CyberVision 2016a]. Devices registering at the Oracle IoT Cloud can declare new models during activation. Before being used, these draft models have to be activated, which will also update them on the device that has declared them [Oracle 2016].

6. SUMMARY AND OUTLOOK

The IoT is growing from a vision to reality and with it, the number of technologies and standards used and created, as well as the solutions build with them, grow more and more confusing. To bring some order into this field and to provide IoT architects and developers some guidance, we started collecting IoT Patterns. In our previous work, we presented patterns for IoT devices [Reinfurt et al. 2017b] and IoT device communication and management [Reinfurt et al. 2016, 2017a]. In this paper, we added ten new patterns handling device bootstrapping and registration, of which five where described in detail. FACTORY BOOTSTRAP describes how initial communication can be set up during device production, while

MEDIUM-BASED BOOTSTRAP and REMOTE BOOTSTRAP handle this issue later, either manually in the field, or remotely. Next follows the device registration process, where devices are made known to their communication partners. This process may be triggered by the device itself using AUTOMATIC CLIENT-DRIVEN REGISTRATION, by another component via AUTOMATIC SERVER-DRIVEN REGISTRATION, or manually via MANUAL USER-DRIVEN REGISTRATION. During this process, a model of the device is created and stored in a DEVICE REGISTRY. This model may be defined by the device itself as a DEVICE-DRIVEN MODEL, or it is defined by the server and selected by the device (PRE-DEFINED DEVICE-DRIVEN MODEL) or assign to the device by the server (SERVER-DEFINED MODEL).
We have already collected more of these patterns and plan to combine them into a pattern language for IoT systems, which would further elaborate on the interconnections and relationships between these patterns.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Uwe van Heesch, for the discussions and comments that helped to improve this paper. This work was partially funded by the BMWi projects SePiA.Pro (01MD16013F) and SmartOrchestra (01MD16001F).

REFERENCES

3GGP. 2003. *Over-The-Air (OTA) technology*. (2003). Retrieved June 20, 2016 from ftp://www.3gpp.org/tsg_sa/WG3_Security/TSGS3_30_Povoa/Docs/PDF/S3-030534.pdf.

Alexander, C., Ishikawa, S., and Silverstein, M. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

Atzori, L., Iera, A., and Morabito, G. 2010. The internet of things: A survey. *Computer networks 54*, 15, 2787–2805.

Autodesk. 2016. *DEVICES - Autodesk SeeControl*. (2016). Retrieved May 2, 2016 from http://www.seecontrol.com/devices/.

Ayla Networks. 2015. *Key Capabilities*. (2015). Retrieved December 9, 2015 from https://www.aylanetworks.com/platform/key-capabilities.

Bosch Software Innovations. 2015. *The Bosch IoT Suite. Technology for a Connected World*. (2015). Retrieved December 3, 2015 from https://www.bosch-si.com/media/en/bosch_software_innovations/documents/brochure/products_2/bosch_iot_suite/brochure.pdf.

Chandra, G.S. 2016. *Pattern language for IoT applications*. (2016).

Coplien, J.O. 1996. *Software Patterns*. SIGS management briefings. SIGS, New York, NY.

CyberVision. 2016a. *Endpoint profiling - Kaa - Kaa documentation*. (2016). Retrieved April 7, 2016 from http://docs.kaaproject.org/display/KAA/Endpoint+profiling.

CyberVision. 2016b. *Endpoint registration - Kaa - Kaa documentation*. (2016). Retrieved April 7, 2016 from http://docs.kaaproject.org/display/KAA/Endpoint+registration.

Devialet. 2016. *How to upgrade the internal firmware of your Devialet*. (2016). Retrieved January 16, 2017 from https://help.devialet.com/hc/en-us/articles/203402221-How-to-upgrade-the-internal-firmware-of-your-Devialet.

Eloranta, V.-P., Koskinen, J., Leppänen, M., and Reijonen, V. 2014a. *Designing distributed control systems. A pattern language approach*. Wiley series in software design patterns. Wiley, Hoboken, NJ.

Eloranta, V.-P., Koskinen, J., Leppänen, M., and Reijonen, V. 2014b. *Patterns for the Companion Website*. (2014). Retrieved August 15, 2016 from http://media.wiley.com/product_ancillary/55/11186941/DOWNLOAD/website_patterns.pdf.

Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., and Leymann, F. 2014a. Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains. *International Journal on Advances in Software 7*, 3&4, 710–726.

Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., and Leymann, F. 2014b. From Pattern Languages to Solution Implementations. In *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. IARIA, Wilmington, DE, 12–21.

Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., Leymann, F., Hadjakos, A., Hentschel, F., and Schulze, H. 2016. Leveraging Pattern Application via Pattern Refinement. In *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC)*.

Falkenthal, M., and Leymann, F. 2017. Easing Pattern Application by Means of Solution Languages. In *Proceedings of the Ninth International Conferences on Pervasive Patterns and Applications (PATTERNS) 2017*. Xpert Publishing Services, 58–64.

Fehling, C., Barzen, J., Breitenbücher, U., and Leymann, F. 2015a. A Process for Pattern Identification, Authoring, and Application. In *Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2721956.2721976.

Fehling, C., Barzen, J., Falkenthal, M., and Leymann, F. 2015b. PatternPedia - Collaborative Pattern Identification and Authoring. In *PURPLSOC (In Pursuit of Pattern Languages for Societal Change): The Workshop 2014*. epubli GmbH, Berlin, 252–284.

Fehling, C., Leymann, F., Retter, R., Schupeck, W., and Arbitter, P. 2014. *Cloud Computing Patterns. Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, Wien.

Fernandez, E.B. 2013. *Security Patterns in Practice. Designing Secure Architectures Using Software Patterns*. Wiley.

Gemalto. 2016a. *LinqUs On-Demand Provisioning Service*. (2016).

Gemalto. 2016b. *On-Demand Provisioning Service*. (2016).

Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. 2013. Internet of Things (IoT). A vision, architectural elements, and future directions. *Future Generation Computer Systems 29*, 7, 1645–1660. DOI:http://dx.doi.org/10.1016/j.future.2013.01.010.

Harrison, N.B. 2006a. Advanced Pattern Writing. Patterns for Experienced Pattern Authors. In *Pattern languages of program design 5*. Addison-Wesley, Upper Saddler River, NJ, 433–452.

Harrison, N.B. 2006b. The Language of Shepherding. A Pattern Language for Shepherds and Sheep. In *Pattern languages of program design 5*. Addison-Wesley, Upper Saddler River, NJ, 507–530.

Heer, T., Garcia-Morchon, O., Hummen, R., Loong Keoh, S., Kumar, S.S., and Wehrle, K. 2011. Security Challenges in the IP-based Internet of Things. *Wireless Personal Communications 61*, 3, 527–542. DOI:http://dx.doi.org/10.1007/s11277-011-0385-5.

Hohpe, G., and Woolf, B. 2004. *Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, Massachusetts.

IBM. 2015a. *Device Management*. (2015). Retrieved December 14, 2015 from https://docs.internetofthings.ibmcloud.com/reference/device_mgmt.html.

IBM. 2015b. *Device Model*. (2015). Retrieved December 14, 2015 from https://docs.internetofthings.ibmcloud.com/reference/device_model.html.

IETF. 2012. *Security Bootstrapping Solution for Resource-Constrained Devices*. (2012). Retrieved January 16, 2017 from https://tools.ietf.org/id/draft-sarikaya-core-sbootstrapping-04.txt.

Intel. 2017. *Step 1: Make a bootable micro SD card*. (2017). Retrieved January 16, 2017 from https://software.intel.com/en-us/get-started-galileo-windows-step1.

Ishaq, I., Carels, D., Teklemariam, G., Hoebeke, J., Abeele, F., Poorter, E., Moerman, I., and Demeester, P. 2013. IETF Standardization in the Field of the Internet of Things (IoT). A Survey. *JSAN 2*, 2, 235–287. DOI:http://dx.doi.org/10.3390/jsan2020235.

Kii. 2015. *Schema*. (2015). Retrieved December 18, 2015 from http://documentation.kii.com/en/starts/thingifsdk/schema/.

Kumar, A., Saxena, N., Tsudik, G., and Uzun, E. 2009. Caveat eptor: A comparative study of secure device pairing methods. In *2009 IEEE International Conference on Pervasive Computing and Communications*. Institute of Electrical and Electronics Engineers (IEEE). DOI:http://dx.doi.org/10.1109/percom.2009.4912753.

Meszaros, G., and Doble, J. 1996. Metapatterns: A Pattern Language for Pattern Writing. In *Third Pattern Languages of Programming Conference*. Addison-Wesley.

Open Mobile Alliance. 2015. *Lightweight Machine to Machine Technical Specification*. (2015). Retrieved December 4, 2015 from http://technical.openmobilealliance.org/Technical/Release_Program/docs/LightweightM2M/V1_0-20151030-C/OMA-TS-LightweightM2M-V1_0-20151030-C.pd.

Oracle. 2016. *Using Oracle Internet of Things Cloud Service*. (2016). Retrieved May 2, 2016 from http://docs.oracle.com/cloud/latest/iot/IOTGS/IOTGS.pdf.

Qanbari, S., Pezeshki, S., Raisi, R., Mahdizadeh, S., Rahimzadeh, R., Behinaein, N., Mahmoudi, F., Ayoubzadeh, S., Fazlali, P., Roshani, K., Yaghini, A., Amiri, M., Farivarmoheb, A., Zamani, A., and Dustdar, S. 2016. IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications. In *Proceedings of the First International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 277–282. DOI:http://dx.doi.org/10.1109/IoTDI.2015.18.

Raspberry Pi Foundation. 2017. *Installing operating system images*. (2017). Retrieved January 16, 2017 from https://www.raspberrypi.org/documentation/installation/installing-images/README.md.

Reinfurt, L., Breitenbücher, U., Falkenthal, M., Leymann, F., and Riegg, A. 2016. Internet of Things Patterns. In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM.

Reinfurt, L., Breitenbücher, U., Falkenthal, M., Leymann, F., and Riegg, A. 2017a. Internet of Things Patterns for Communication and Management. *LNCS Transactions on Pattern Languages of Programming*.

Reinfurt, L., Breitenbücher, U., Falkenthal, M., Leymann, F., and Riegg, A. 2017b. Internet of Things Patterns for Devices. In *Proceedings of the Ninth International Conferences on Pervasive Patterns and Applications (PATTERNS) 2017*. Xpert Publishing Services, 117–126.

Singh, J., Pasquier, T., Bacon, J., Ko, H., and Eyers, D. 2016. Twenty Security Considerations for Cloud-Supported Internet of Things. *IEEE Internet Things J. 3*, 3, 269–284. DOI:http://dx.doi.org/10.1109/JIOT.2015.2460333.

SiteWhere. 2015. *System Architecture*. (2015). Retrieved December 8, 2015 from http://documentation.sitewhere.org/architecture.html.

Voas, J. 2016. Networks of 'Things'. *NIST Special Publication 800*, 183.

Wellhausen, T., and Fießer, A. 2012. How to write a pattern? A rough guide for first-time pattern authors. In *Proceedings of the 16th European Conference on Pattern Languages of Programs*. ACM, New York, NY.