FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Internet of Things Software Modules Marketplace

João Pedro Furriel de Moura Pinheiro

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Gil Manuel Magalhães de Andrade Gonçalves

Second Supervisors: Eliseu Moura Pereira and João Reis

July 23, 2020

Internet of Things Software Modules Marketplace

João Pedro Furriel de Moura Pinheiro

Mestrado Integrado em Engenharia Informática e Computação

July 23, 2020

Abstract

The advent of the Cyber-Physical Systems (CPS), a physical system representation through a virtual model, usually used to control a system or a process comes from the growing democratization of the computational power. Nowadays, virtually anything can be equipped with some kind of embedded processor to automate tasks, generate or consume some kind of data. In addition, the continuous development and improvement of the communication networks have helped leverage the concept of the Internet of Things (IoT) in which things are now, themselves, connected to the Internet, exchanging data with each other and with people.

In the industrial sector, CPS, also called Cyber-Physical Production Systems (CPPS) and the IoT are the main technological advances that originated the Fourth Industrial Revolution, commonly designated as Industry 4.0 in which the factory floor is no longer a centralized model where all the computation is done centrally but is now a decentralized model where industrial equipment have embedded devices to control, automate tasks and react in a dynamic and intelligent manner to the sensed physical environment.

Thereby, one of the keywords around the CPPSs is software. Software is no longer centralized and is now distributed through several devices that comprise the system. This new approach comes with significant changes and one of them is the reuse and distribution of the software. It is not viable to manually deploy and install software in hundreds or thousands of devices and not having a way of reusing the existing software. If, on the one hand, the desire is to develop a more intelligent process control system, on the other, flexibility, adaptability and simplicity are also convenient capabilities, or else intelligent manufacturing process control systems are built upon a lot of resources debt. Hence, the solution is to build standards, tools and frameworks that allow the reuse of software and its rapid deployment in the distributed devices.

One option, in the Industry 4.0 field, to cope with the software reuse issue in this kind of systems is the encapsulation of software in functional blocks, the Function Blocks (FBs) and their use in the function block programming paradigm, described in IEC-61499 standard. The functionality is abstracted away in the FBs and can be reused by just deploying them to the devices. This way, it is easier to manage a network by dragging and dropping these blocks, building complex applications centrally and deploy everything to the distributed embedded devices. However, the implementation of this standard to address the aforementioned problem brings, itself, other necessities such as managing the FBs, monitoring them and their previous download by the embedded devices.

This dissertation's main goal is the development of a marketplace to manage and monitor FBs in an IEC-61499 network envisioning the filling of the previously mentioned gaps in this kind of networks. The marketplace, integrated in an IEC-61499 global solution will enable the distribution of FBs among the embedded devices in a IEC-61499 compliant CPPS, functioning as a central repository of software components, having also monitoring features, allowing the

detection of flaws or malfunctions about FBs usage.

Keywords: Cyber-Physical Systems, Cyber-Physical Production Systems, Internet Of Things, IEC-61499, Function Block Programming, Software Marketplace

Resumo

O aparecimento dos *Sistemas Ciber-Físicos* (CPS), uma representação dum sistema físico através dum modelo virtual, normalmente usado para controlar um sistema ou um processo, teve origem na crescente democratização do processamento computacional. Hoje em dia quase todos os objetos são passíveis de serem apetrechados com algum poder computacional para automatizar determinada tarefa, gerar ou consumir algum tipo de dados. Além disso, a melhoria contínua das redes de comunicação levou ao advento do conceito da *Internet Of Things* (IoT) em que as coisas passam a estar ligadas, também elas, à rede.

Na área industrial, os CPS, também designados por Sistemas Ciber-Físicos de Produção (CPPS) e a IoT foram os grandes catalizadores do movimento Indústria 4.0 ou quarta revolução industrial em que, entre outros avanços, a gestão do chão de uma fábrica deixou de ser um modelo centralizado, onde a computação era feita centralmente e passou a ser um modelo descentralizado, onde os equipamentos industriais estão munidos de dispositivos embebidos para os controlarem, automatizar tarefas ou até responder dinâmica e inteligentemente a mudanças percepcionadas.

Uma das palavras-chave desta revolução é então o *software*. O software deixa de estar centralizado e passa a estar distribuído pelos vários dispositivos pertencentes ao sistema. Esta nova abordagem traz mudanças significativas e uma delas é a da reutilização e distribição do software. Não é viável instalar software manualmente em milhares de dispositivos, ou não reaproveitar software já desenvolvido para um determinado dispositivo, noutro, caso faça sentido. Se por um lado, o desejo é um controlo de processo mais inteligente, por outro flexibilidade, simplicidade de configuração e adaptabilidade são características desejadas sob pena de se conseguirem sistemas mais inteligentes mas com uma penalização grande em termos de tempo e mão-de-obra dispendidos. Desta forma, a solução passa pelo desenvolvimento de *standards*, ferramentas e *frameworks* que permitam a reutilização de software e a sua rápida instalação nos dispositivos da rede.

Uma das alternativas, surgida na área da indústria 4.0, para lidar com o problema da reutilização de código neste tipo de sistemas é o encapsulamento do software em blocos funcionais, os chamados *Function Blocks* (FBs) e a sua utilização em programação por blocos, que está descrita no standard IEC-61499. A funcionalidade é abstraída em FBs e pode ser reutilizada apenas instalando determinada função num determinado dispositivo. Desta forma, é possível gerir uma rede simplesmente arrastando blocos, construindo aplicações complexas e no fim apenas carregar num botão para iniciar o processo. No entanto, o surgimento deste standard para endereçar os problemas que esta arquitetura acarreta, traz ele próprio, outras necessidades, tais como a de gestão dos FBs, a própria monitorização dos mesmos aquando da sua execução nos nós da rede e o seu prévio download por parte dos dispositivos embebidos.

Esta dissertação tem, assim, como objetivo, o desenvolvimento de um *marketplace* para a gestão e monitoriação de FBs numa rede implementada segundo o standard IEC-61499, visando assim colmatar essa necessidade. Esta aplicação, integrada numa solução global IEC-61499, permitirá a rápida instalação de FBs nos nós da rede, atuando como repositório central de components

de software. Além disso, terá também uma componente de monitorização, permitindo detetar falhas ou malfuncionamentos no uso destes componentes.

Keywords: Sistemas Ciber-Físicos, Sistemas Ciber-Físicos de Produção, Internet Of Things, IEC-61499, Programção por Blocos, Marketplace de Software

Acknowledgements

I would like to thank my supervisors, professor Gil Gonçalves, Eliseu Pereira, and João Reis for their availability, support and friendship during this last year.

To the colleagues I have had the chance to meet over the last five years, it was a really long journey and it would have not been possible without their support and friendship.

To the ones who, in some way, encouraged my hard decision to get back to the University, despite of the unusual of the situation, it really meant a lot.

To my friends, who supported my decision since the very beginning.

To my brother, sister, and father, who always cheered me up with their love and caring.

A special thanks to my mother, who could not have done more to help me, not only in the last five years but also during my whole life.

And lastly to my girlfriend, my daily support, it would have been impossible to reach even the end of the first year of this long adventure without her presence, support, and love.

To all of them, my sincerest thanks.

João Pedro Furriel de Moura Pinheiro

"Talent is luck,
the important thing in life is courage."

Woody Allen

Contents

List of Figures

List of Tables

Abbreviations

CPS	Cyber-Physical System
CPPS	Cyber-Physical Production System
IoT	Internet of Things
IIoT	Industrial Internet of Things
FB	Function Block
DT	Digital Twin
GUI	Graphic User Interface
AAS	Asset Administration Shell
MQTT	MQ Telemetry Transport
PLC	Programmable Logic Controller
OPC-UA	OPC Unified Architecture
FBDK	Function Block Development Kit
FBRT	Function Block Runtime
MVC	Model-View-Control
I3	Intelligent IoT Integrator
CLI	Command-Line Interface
OS	Operating System
DBMS	Database Management System
IDE	Integrated Development Environment
RE	Runtime Environemt
PO	Product Owner
IEC	International Electrotechnical Commission
CRUD	Create, Read, Update and Delete Operations
HTTP	Hypertext Tranfer Protocol
UI	User Interface
DOM	Document Object Model
DTD	Document Type Definition
TSP	Traveling Salesman Problem

Chapter 1

Introduction

1.1 Context

The ability to connect things to a network, instead of just people, led to the advent of the commonly called Internet of Things (IoT) phenomena. If, in addition, these things are equipped with computational power, a whole new world can be explored, bringing the potential to change people and organizations' lives. These technological breakthroughs improved the efficiency and flexibility of process control systems in several fields, as things that otherwise were just standalone nodes of a system like sensors, actuators, industrial machinery among others, with no embedded intelligence and local controlled, are now capable of processing information, make decisions and communicate with each other. These advances led to the introduction of the so-called Cyber-Physical Systems (CPS) which are virtual representations of physical process control systems by means of their digital twin counterparts[35]. These systems have been implemented in several fields from Healthcare, Energy, Intelligent Transportation, or Manufacturing.

In the particular sector of manufacturing, these systems are commonly called Cyber-Physical Production Systems (CPPS). CPPSs have helped to leverage the Fourth Industrial Revolution[36], leading to paramount breakthroughs that are helping organizations and companies to succeed.

CPPSs provide production systems with capabilities of making industrial machinery more intelligent, flexible, and adaptable to make the production process more efficient, reducing its costs and increasing profits. CPPSs do this by providing typical industrial machinery with processing power with embedded systems to control them and adding a variety of sensors so these machines can sensor the physics around and react to them, turning the production system into a computational distributed one. This network of things and machines was possible thanks to the emergence of the IoT concept, particularly the Industrial Internet of Things (IIoT).

As previously mentioned, CPPSs aim to help the manufacturing process to become more flexible and adaptable by adding processing and sensing capabilities to shop floor equipment. With this, it is possible to control the process, reconfigure a piece of specific equipment, or even monitoring the state of the different devices.

CPPSs enable the creation of a virtual system, representing the physical system. The virtual system contains the digital twins of the physical equipment on the shop floor. With this approach, it is possible to control the functions assigned to each equipment, aiming to a quick reconfiguration of the overall process when is needed.

Moreover, besides giving the possibility of controlling the production process, with the implementation of CPPSs it is possible to collect data that otherwise would not be possible, and processing this data, locally or remotely, can give valuable insights into how the system is performing and how it might be improved.

With all this processing power and data, it is of crucial importance that the system could be easily re-programmable and configurable so the software installed and running in a device (to control specific equipment or a given algorithm) could be rapidly deployed to the different nodes of the system, forming a functioning overall process. One way to achieve this is to abstract and encapsulate functionalities into software modules that can be later installed in these devices. The key is the reusability and distribution of the modules so one does not need to know how the module is developed but only what it does, what are the inputs and outputs so it is easy to decide where the module needs to be deployed.

For this reason, several protocols and standards have been developed. One of those standards is the IEC-61499[16], which defines the concept of Function Block Programming in which a Function Block (FB) represents the building block of the system. These building blocks execute a specific task, communicate with others using data and event inputs and outputs and altogether define how the system function hence easing the tasks of managing the shop floor of a factory.

1.2 Motivation

Current solutions for CPPSs are more concerned with the flexibility and configurability of the manufacturing process so one part of the system can be quickly replaced, updated, or even reconfigured in a quick and intelligent way to tackle a specific requirement. The way they are doing this is by abstracting and reusing the software modules representing the functions that a piece of given equipment is performing.

However, software management and distribution in a CPPS is an important issue that should be addressed. It is important that devices that should run a specific software module have that module installed, configured and ready to run so, for instance, the integration of a new device could follow the plug-and-play paradigm, avoiding the process of assuring that the device is properly configured with all the software needed.

Although abstracted away in black boxes, it is important to monitor software modules so the engineer in charge can quickly locate a malfunctioning module and take action. It is not only a matter of adding, removing, or moving pieces of software around but also monitoring their state, otherwise, the flexibility of the system is penalized.

In summary, it is possible to contribute to the development of CPPS technology by providing means to tackle the aforementioned issues, improving the flexibility and configurability of a production system, facilitating the quick integration of software components and devices, decreasing the complexity of industrial automation management.

1.3 Objectives

With the previously mentioned approaches, although it becomes easier and flexible to configure and reconfigure the system, other issues emerge. Where to store all these software modules, how to transfer them to the target devices, and how to monitor possible malfunctions occurring are among those.

This dissertation aims to tackle these issues by means of developing a central repository, from a marketplace perspective. This repository will be comprised of a software components registry where the software modules code will reside and a database, storing the meta-information of the modules and their error reports. On the other hand, a protocol will be developed in order to communicate with the embedded devices so they can request the needed modules, these devices will also transmit their state using an industry well-established protocol. At last, a client application to manage the marketplace will also be developed.

In general terms, the main objectives are listed next.

- **Function Block Management** - The final solution needs to provide a way to manage the software modules (Function Blocks) available by uploading the code of the module and saving all the metadata that the standard requires like inputs and outputs as well description of what that component does. Hence this component should be a Graphic User Interface (GUI)

- **Automatic components download** - The distributed system is composed of different devices embedded in the physical components of the factory floor. These devices must communicate with a repository in order to download and install the software components needed to run.

- **Real-Time Monitoring** - The system must provide a way to monitor the devices' resources on the network, like the amount of CPU or RAM being used. With this functionality, it is possible to control which ones are performing according to what is expected as well as the ones not performing as expected and hence infer about the efficiency of the system or the modules installed in the device.

- **Error Reporting** - The system must provide a way so the embedded devices could report to the central repository possible error states and the central repository show those errors in the GUI.

1.4 Problem Definition

As mentioned before, the Industry 4.0 brought a great variety of tools to industrial automation in order to improve the efficiency and reduce the costs of production, turning the industrial automation in a very complex distributed system. In what a factory floor is concerned, a CPPS is the virtual representation of the physical equipment installed. Each of these has a digital twin mapped in that virtual representation and each of these digital twin is composed of a set of software modules, all articulated to perform a complex task. The ability of quickly redefine the way all the devices and software modules are built is a key enabler for achieving flexibility and making the process quickly adaptable to desired changes.

To accomplish this goal, reusable software is indispensable. There has to be a quick way of managing these pieces of software as black boxes, making them pieces of a puzzle, in order to configure the manufacturing process. In order to apply this concept, an industry-standard came along, the IEC-61499 which defines these black boxes as FBs. Each FB performs a very specific task, is controlled by an algorithm, ranging from simple input/output algorithm to a complex machine learning one and communicates with others via data and events.

A set of FBs defines an application which in turn, along with other applications composes the controlling system. This hierarchy is depicted in figure 1.1.

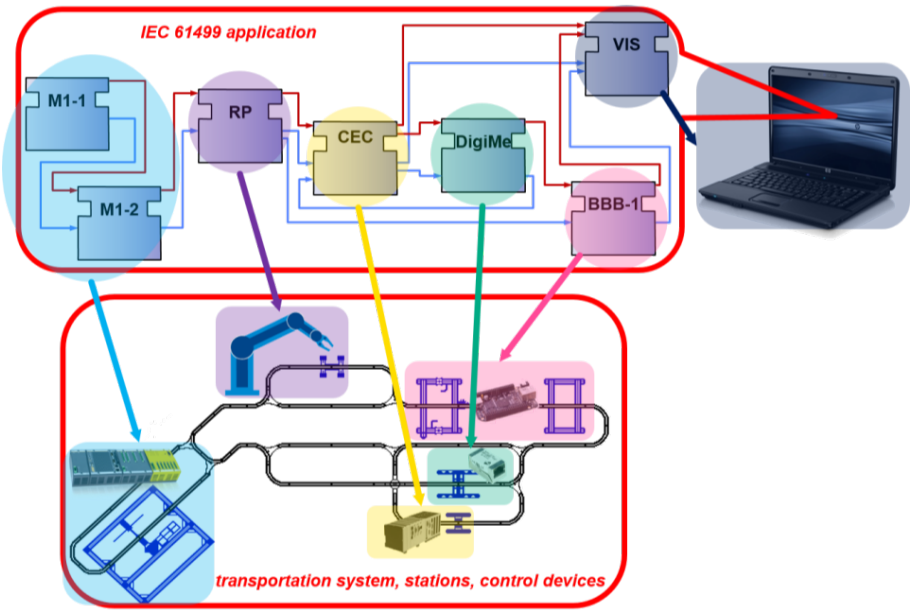


Figure 1.1: IEC-61499 System Design

To correct implement this standard, two software components are needed, an Integrated Development Environment (IDE) in order to build the configuration of the system and a Runtime

Environment (RE) which is the platform, installed in the embedded devices where the software will run. There are IEC-61499 compliant solutions described in section 2.3.1.

From an operational point of view, there is the assumption that these software modules are available and correctly installed and configured in all the system devices which can lead to difficulties when it comes to integrate a new device or develop a new module (in this case a FB). This difficulty constitutes the main problem that leads to the development of this work.

From a monitoring point of view, it is also important to have a mechanism to quickly locate malfunction FBs as well as it is convenient to have a tool keeping track of the resources usage by each device. The lack of this kind of tools and mechanisms also contributes to the problem definition and ultimately leads to an integrated solution to fill these gaps.

1.5 Proposed Solution - Function Block Marketplace

In order to address the requirements mentioned in section 1.3, a centralized repository of FBs will be developed, forming the marketplace and a representation of the proposed architecture can be seen in figure 1.2. A client application will be developed so the marketplace management can be made in a GUI. In terms of database, a relational model will be chosen as the standard IEC-61499 is strict in terms of the definition of FBs, so data integrity is an important requirement. The chosen technologies will come from the ones explored in section 2.4 and will be analyzed in-depth in the first phase of the project.

As the communication between the client and the server will be made by means of HTTP, the repository will consist in a web server and a database and the client will be a Web Application. The requests between these two components will be made by using an API of web services exposed by the Web Server. In addition, an FTP server will be deployed to serve the implementation files of the software modules upon request from the embedded devices.

In terms of IEC-61499 compliant software in which the final solution will be integrated, it will be used the 4Diac IDE along with Dinasore RE. The 4Diac IDE[7], more detailed in section 2.3.1, is the application used to actually build the CPPS, in terms of its building blocks, the FBs. The Dinasore RE[23], also detailed in section 2.3.1 is the software where the FBs will run on the embedded devices. An integration module for Dinasore RE will be developed so the runtime is aware of the marketplace in order to download all the required FBs.

At last, in terms of server-side integration, there will be an integration module between the webserver and the Dinasore RE by means of the OPC-UA protocol[22]. This protocol is an industry standard one for data exchange between different vendor industrial machines and is detailed in section 2.2.4.1. With this integration, the webserver can be notified of the status of the equipment and software and locate it, in case of malfunction. An integration with the 4Diac IDE will also be designed so it can be aware of the available FBs to configure the system.

The solution will be called Jurassic Park, in a reference to the Jurassic Park movie [44] as this platform will be used to control a set of Dinasores, the RE installed in the embedded devices of the distributed system.

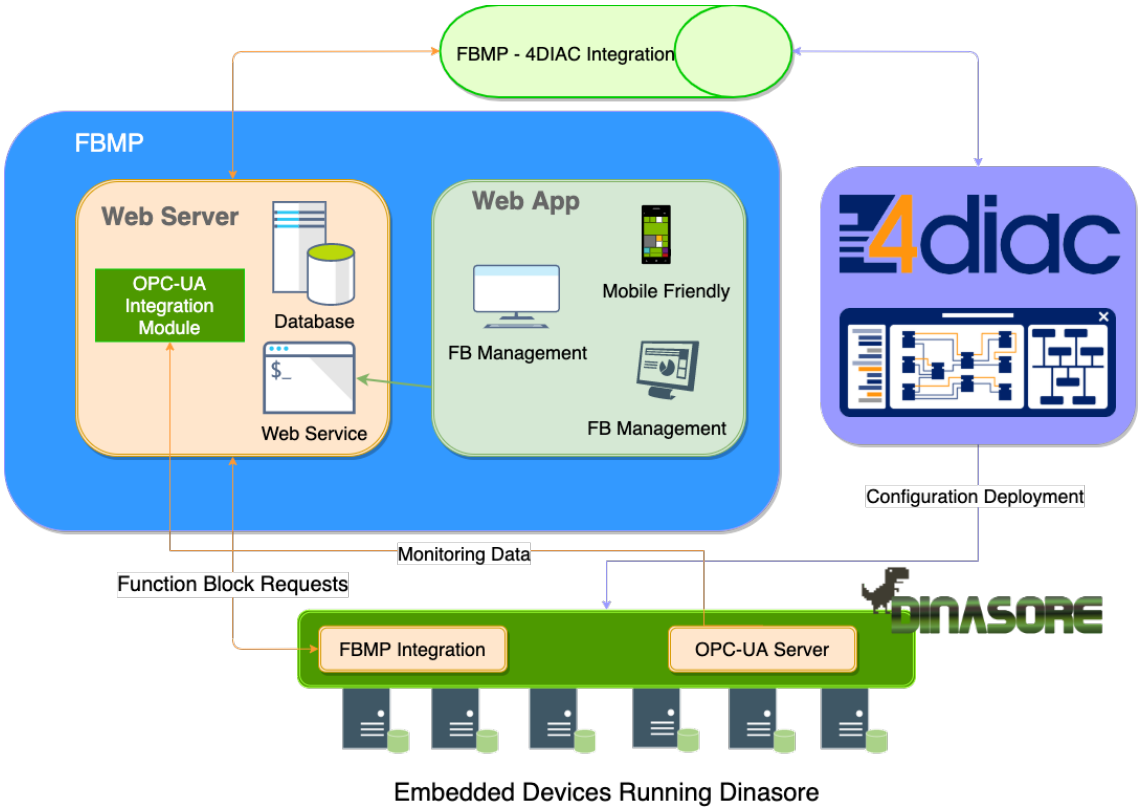


Figure 1.2: Proposed Solution

1.6 Document Structure

This document is divided into 6 chapters. Chapter 1 introduces the project context, motivation, and main objectives of the work. It also defines the problem and the envisioned solution to cope with it. At the end of the chapter, the document structure is described.

In chapter 2, a state of the art revision is made where the background of CPSs is explored, the technologies related with CPPSs and the technological options for technical requirements of the solution are revisited as well as some IoT Marketplaces proposed architectures are described and finally, the conclusions of this revision is presented.

Chapter 3 describes the final architecture of the solution and also explains the decisions taken in terms of technologies adopted for each of the system components.

In chapter 4 the whole implementation process is detailed, all the functionalities are described as well as the code structure explained.

In chapter 5 the tests conducted with the developed solution are presented. Moreover, the results of those tests are also described in this chapter.

Finally, in chapter 6 the final conclusions about the development of the work are detailed. In addition, some remarks on what might be the future work to build on the solution achieved.

Chapter 2

State of The Art

This chapter presents the state of the art and the context of this work. Section 2.1 aims to introduce the concepts behind the Cyber-Physical Systems to explain how these systems have become, nowadays, so important in many fields. The second section, 2.2, does an overview of the fields where Cyber-Physical Systems can be applied with a focus in Cyber-Physical Production Systems. The third section, 2.3 describes related work in the field, more specifically IEC-61499 tools, some purposed marketplace concepts in IoT, and well-established architectures and software distribution tools. The fourth section, 2.4, describes different technologies to implement the desired final solution of the project and the last section, 2.5, closes with the conclusions about what is the gap this work is trying to fill and what can be done to make these systems even more valuable.

2.1 Introduction

Cyber-Physical Systems (CPS) are integrations of computational/virtual entities with physical processes[35]. With the increasing access of people and organizations to computational enabled devices with low price and size, objects that otherwise were just standalone functional parts of a system can now be controlled. Moreover, the increasing speed and bandwidth of nowadays networks, along with an increasing capacity of processors paved the way for the advent of the so-called IoT. Thereby, those aforementioned objects not only have computational power but also the capability of communicating with each other as well as with humans. Among these objects, that we can call smart objects are sensors, industrial robots, medical devices, smart switches, and many others. These smart objects together can form a network to control some kind of process applied to many areas like healthcare, automobile, manufacturing, or energy, among others.

A more detailed definition of what could be a Cyber-Physical System can be found at [42]: "Cyber-physical systems (CPS) are physical and engineered systems whose operations are monitored, coordinated, controlled and integrated by a computing and communication core". With the combination of the previously mentioned causes (access to processing power and the improvement of networks), we can build this kind of systems. These systems are called Cyber-Physical Systems not only because of the capacity to interact with human agents but also because usually,

these systems try to replicate the physical process into a digital one, resulting in what is called the Digital Twin (DT).

In short, the advent of the Cyber-Physical Systems is a result of the proliferation of low-cost and increased-capability sensors of increasingly smaller form factor; the availability of low-cost, low-power, high-capacity, small-form-factor computing devices; the wireless communication revolution; abundant internet bandwidth; continuing improvements in energy capacity, alternative energy sources, and energy harvesting.[42].

The increasing research and applications of CPS come in a moment where Cloud Computing is also booming. "The Cloud Computing is the result of advancement of a few technologies, such as distributed computing, Internet technologies (service oriented architecture, web services), system management (autonomic computing), and hardware (virtualization, multicore chips)"[33]. This happens because although processing and storage capacity found in the components of a system are increasing, it is still necessary bigger support to store all the data coming from so many devices.

At last, it is important to state that CPSs, by their nature of generating and sending data, raise several issues regarding security, interoperability, robustness, and reliability of the system. As an example, one wouldn't like to know that the private data being measured about blood pressure is not being sent in a private and secure manner to the ones who should read and take action with the data.

2.2 Cyber-Physical Systems Applications

In this section, a few examples of areas where CPS can be applied are described. The last one, Cyber-Physical Production Systems in the context of this work and because of that, a deeper revision is made in order to explain the contribution of this project to this specific area.

2.2.1 Healthcare

The Healthcare field is probably among the ones with more potential in terms of CPS applications. If one thinks about the vast number of medical sensors that have been around for years, the possibilities are endless regarding the new capabilities to communicate and even process the data. Healthcare CPSs equip not only in-hospital applications but also, and maybe with more disruption power, in-home applications. Sensors can read the patient data and send this data to a remote location where clinicians can take action as well [33].

Typical applications of CPS in healthcare are Electronic Health Record, Smart Checklist, Vital Sign monitoring and alarm software, Medication Intake Applications, Medical Status Monitoring Applications, and Telemedicine.

2.2.2 Energy

The advantages of applying CPS in the Energy field are countless. One can think of an electric power transmission grid as one of the largest complex interconnected networks ever built [42]. It

becomes clear that applying data processing and intelligence to all these interconnected networks is crucial to improve the quality and to reduce the likelihood of power blackouts. The main application of the CPS in the energy is related to the concept of Smart Grid. "Smart grids are electric networks that employ advanced monitoring, control, and communication technologies to deliver reliable and secure energy supply, enhance operational efficiency for generators and distributors, and provide flexible choices for prosumers. Smart grids are a combination of complex physical network systems and cyber systems" and the contributions of CPS to the Smart Grid are the integration of the real and virtual worlds, the connection between physical and cyber systems, the processing of big data required to help deliver timely decisions for Smart Grid operations and the self-adaption, self-organization, and self-learning by which CPS can respond to faults, attacks, and emergencies, contributing to the resilience and safeness of the Smart Grid.[48].

2.2.3 Intelligent Transportation

Another field where CPSs are becoming an invaluable help is in Intelligent Transportation where Cyber Systems modeling aims to develop new ways of transport and improve traffic management. It is easy to think about the endless number of applications resulting in adding intelligence and data processing capacity to the physical elements of a transportation system like traffic lights, roads, and vehicles. Among those, we can find Collision Avoiding Systems, Automatic Road Enforcement, Emergency Vehicle Notification Systems, and Smart Traffic Lights.

2.2.4 Cyber-Physical Production Systems

A Cyber-Physical Production System (CPPS) consists of a digital representation of a factory floor communicating and exchanging data in real-time with the actual physical system. Cameras, sensors, controllers, robots among other components of the floor can be equipped with processing components in order to collect data and in some cases even process that data. These components alone would not have any use as well as if there was no way to represent the system as a whole. The CPSs along with the advent of IoT and Cloud Computing are the main causes of the Fourth Industrial Revolution, the buzz term Industry 4.0. "Industry 4.0 is a large German initiative that emphasises the extension of traditional manufacturing systems to full integration of physical, embedded and IT systems including the Internet"[36]. Industry 4.0 brings this new paradigm where machines are interconnected and exchange data in order to diagnoses themselves, react to some specific conditions of the physical system and ultimately work together to make products in an intelligent and efficient way. With this new approach, Production Systems will be easier to plan and control [39].

One of the key aspects of a CPPS is the configuration of the system. This kind of system is an example of a distributed one where we can have thousands of nodes communicating with each other and we need a representation of the system, and how it is configured. Not only the current market needs in manufacturing demands for a good configuration of a factory floor but also demands a quick reconfiguration to respond to the rapid changes in the consuming trends.[25].

This need introduces the terms Asset Administration Shell (AAS). "An AAS provides a machine accessible interface to any kind of asset - be it an automation device, a process description, or any other relevant entity. An AAS might contain administration data, runtime information, or even executable operations of a device."[46]. In fact, an AAS, according to the concept of Industry 4.0 can be viewed as "the bridge between the physical world and the IoT world or, in other words, the data model from where the digital twin stems from"[43]. When this term was coined, several protocols to implement a system like this have come to light and there are two that are being widely used in industry, especially in regards to the modeling of the system and to the machine-to-machine communication, namely IEC-61499 (evolution of the IEC 61131-3) for the asset configuration and system modeling and OPC-UA for the vendor-independent machine to machine communication. There is also a very widely used machine-to-machine transport protocol in the IoT domain called MQ Telemetry Transport (MQTT). These three protocols are described in the next section.

2.2.4.1 Important Industry 4.0 Protocols

"IEC-61499 has been developed to enable intelligent automation where the intelligence is genuinely decentralized and embedded into software components, which can be freely distributed across networked devices."[47]. This definition clearly states that, when conforming to the IEC-61499, an Asset Administration System is given the ability to control the execution of software in the different nodes of the system by means of deploying a software encapsulated and abstract element called Function Block (FB).

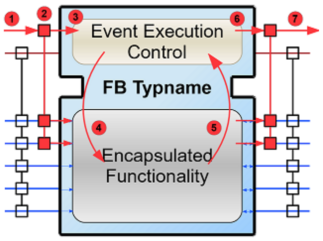


Figure 2.1: Function Block Example

The FB is a reusable abstraction of functionality that a device, usually a Programmable Logic Controller (PLC) can install and execute and is the main artifact of IEC-61499 specification. As can be seen in figure 2.1, a FB has both data and event inputs as well as data and event outputs. The event inputs represent the events that trigger the execution of the FB and the output events are the events that the FB might generate to other linked FBs. The data input is the data transmitted to the FB and data output is the data the FB will pass on to the next FBs. Usually, the control execution of the FB is made by means of a State Machine where the initial state is triggered by an input event and then the rest of the state is controlled either by the data generated or by sensing the physics around.

There are three types of FBs: Basic, Composite, and Service Interface. A basic FB is one with an Execution Control Chart, a state machine that defines the execution of the code, A composite FB is a configuration/pipeline of basic FBs and a Service Interface one is usually used for accessing I/O and communicate via network interfaces.[30].

With the implementation of this standard, the industry tries to easily control and configure a factory floor which represents a distributed system of processing nodes representing a variety of factory devices like sensors, robots, controllers, and so on. Even more important, and to respond quickly to a constantly changing market, the implementation of this standard aims to quickly reconfigure a given configuration of a factory floor. This standard is also intended to establish an open, component-oriented, and platform-independent development framework to improve reusability, reconfigurability, interoperability, portability, and distribution of control software for complex distributed systems [40].

Another important protocol for the correct implementation of an AAS is the OPC Unified Architecture (OPC-UA). This protocol aims to ease the communication and data exchange between different devices from different vendors. "OPC is a set of industrial standards for systems inter connectivity, providing a common interface for communications between different products from different vendors."[37]. While IEC-61499 is more concerned about a high-level configuration of the system and the functions of each part of it at a Process Control Level, OPC-UA is more concerned about the communication from machine to machine. OPC-UA can be used, for example, to monitor device malfunctions.

MQ Telemetry Transport (MQTT) is a lightweight transport protocol widely used in IoT to transfer data across devices. It is a publish/subscribe, simple message protocol designed for low-bandwidth, high latency networks, which make it ideal for machine-to-machine data transfer. It requires a component called Broker where the data producers deliver their messages and data consumers subscribe in order to receive the data, in accordance with the publish/subscribe model. [6]

2.3 Related Work

Usually, to implement the IEC-61499 standard, two major components are needed. On one hand, a visual tool to model the system, drag and dropping Function Blocks onto the devices comprising the system and usually containing a list of the Function Blocks available, what is called the Function Block IDE. On the other hand, a Runtime Environment (RE), running on the embedded devices, where the Function Blocks will run is also necessary. In this section, a comparison of some Software Tools and Runtime Environments is presented, as can be seen in figure 2.2.

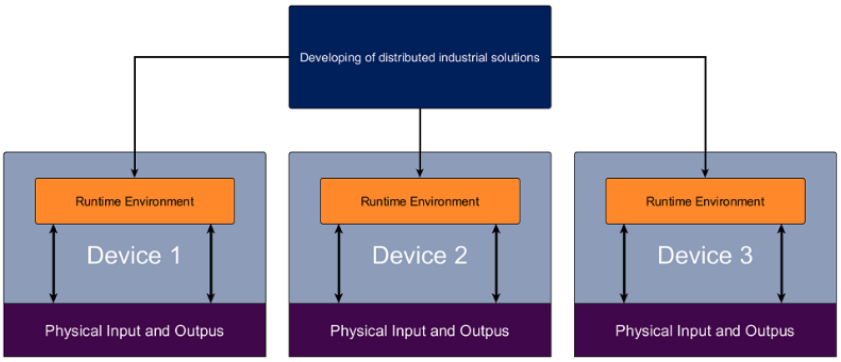


Figure 2.2: Function Block IDE and Runtime Environment[7]

2.3.1 IEC-61499 Software Tools and Runtime Environments

2.3.1.1 FBDK and FBRT

The Function Block Development Kit[10] (FBDK) was the first IEC-61499 developed tool and it has been used to demonstrate IEC-61499 protocol. This tool is developed as a simple Java Applet and it has been used for research purposes. Nowadays serves to verify if other solutions are IEC-61499 compliant. The Runtime developed for the tool is Function Block Runtime(FBRT), it was also developed in Java. However, it is portable to the 4Diac platform which means that it can deploy Function Blocks to devices running 4Diac FORTE or to FBRT. Nowadays, the embedded Runtime is no longer supported. However, a non-embedded runtime is still being used for teaching, research, and testing purposes [27]. FBRT implements a Non-Preemptive Multi-Threading Resource (NPMTR) execution model, which is based on a depth-first FB scheduling mechanism.[40].

2.3.1.2 4Diac IDE and FORTE

In this solution, the 4Diac IDE is the tool where one can actually configure a distributed system of Function Blocks whereas FORTE is the Runtime Environment to be installed in the embedded devices in order to run the Function Blocks assigned by the 4Diac IDE. FORTE Environment is written in C++ and is intended to execute Function Blocks Networks in small embedded devices and run on top of an Operating System, it is multi-threaded, low memory-consuming and has been tested in several Operating Systems. On the other hand, the IDE is written in JAVA, based on the Eclipse Framework. In this IDE, one can model the distributed system, create Function Blocks, applications (set of Function Blocks), and configure the devices.[7]. As stated in the 4Diac webpage, it is not enough to create the Function Blocks on the IDE, once they are created, one needs to manually deploy them to FORTE in the devices. As mentioned before, this solution is portable to FBRT, which means that one can configure a system including devices running FBRT and vice-versa. It is also important to state that 4Diac supports also MQTT and OPC-UA protocols.

2.3.1.3 ISaGRAF Workbench and ISaGRAF Runtime

The ISaGRAF Workbench was the first commercial implementation of the IEC-61499 and is developed by Rockwell Automation. It does not include portability to other tools and does not assure the configurability of any runtime environment other than the IsaGRAF runtime. It does not include the feature of creating and deploying new and customized Function Blocks. This solution, being a commercial one, is more focused on using IEC-61499 to leverage their tool capabilities instead of strictly implement the standard, resulting in a lack of portability and interoperability with other systems [47]. One of the first IEC-61499 implementations with this tool was in a shoe manufacturing factory where the level of flexibility achieved turned the individually tailored shoes cost the same as the mass-manufactured ones[26].

2.3.1.4 nxtSTUDIO and nxtRT61499F

This is also a commercial supported IEC-61499 development environment developed by nxtControl. It is being used by a significant number of device and machine vendors from the manufacturing domain. Its portability with FBRT and FORTE is being tested. It has an efficient cross-platform runtime environment (can run in different types of embedded systems) and is based on the open-source FORTE runtime but enhanced with extra features like OPC-UA servers and Web servers[27]. The development of this tool introduced a concept of Composite Automation Type (CAT), based on the Model View Control (MVC) design pattern where a machine is represented by a Function Block representing that machine parts, divided into Model, View, and Controller[47].

2.3.1.5 4Diac and Dinasore

The last solution is the only one to be developed by two different entities, the IDE is 4Diac, already described in section 2.3.1.2, and with the runtime environment being Dinasore standing for Dynamic INtelligent Architecture for Software and MOdular REconfiguration[23]. This runtime is developed by SYSTEC, a Faculdade de Engenharia da Universidade do Porto Laboratory for Systems and Technologies Research [11]. This runtime is developed in Python hence embedded devices need to have Python Environment installed in order to run Dinasore. However, being written in Python enables the development of Machine Learning algorithms as Function Blocks, making this a good solution to implement intelligent Process Control Systems.

2.3.2 Marketplaces in IoT Context

In recent years, there as been a growing trend of implementing digital marketplaces applications. These marketplaces can either be one-sided like clothes or food e-commerces or two-sided marketplaces where vendors could make deals with buyers in order to exchange products.

In the context of the IoT, the marketplace approach envision takes advantage of the growing number of interconnected devices capable of generating data as well as the big number of data demanding applications being developed nowadays. Although this kind of marketplaces cannot

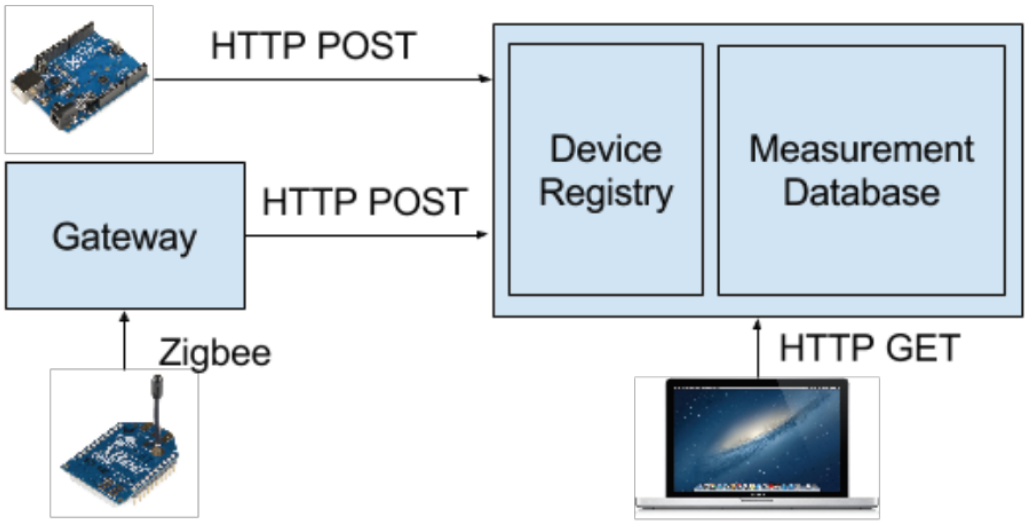


Figure 2.3: Data Marketplace Architecture [38]

be seen yet in use, there are several proposed models in order to implement the idea behind it. "The basic idea is that sensors will provide data, and a large number of users will be able to efficiently access it from anywhere using Internet."[38].

A model was proposed by [38] where device owners first register their devices with metadata about what they measure, where they are located, price, and data buyers would query the system about the devices available, prices, and measurement frequency needed as can be seen in figure 2.3.

Another approach was introduced by [34] in a project called IoT Marketplace for Smart Communities. They called it the Intelligent IoT Integrator (I3). Although the high-level model and objectives were similar, the implementation proposed is different in regard to the protocols used. Instead of using REST Web services, they proposed an architecture where data, after being negotiated between owners and buyers, would be transferred by means of the Publish-Subscribe Protocol where buyers subscribed to the channels representing the data streams of the data they bought. This architecture was defined with an MQTT Broker and a Django-based backend and MySQL Database. The overall architecture can be seen in figure 2.4.

Another model, proposed by [41] and coined as Sensing-as-Service envision a system where the marketplace is implemented following the current trends like infrastructure-as-a-service, platform-as-a-service or software-as-a-service, providing resources as a service, instead of owning those resources. According to this approach, the data consumer would have to pay only while data is needed and as a service, avoiding the negotiation with the data provider, which is abstracted away in the service layer, as can be seen in figure 2.5.

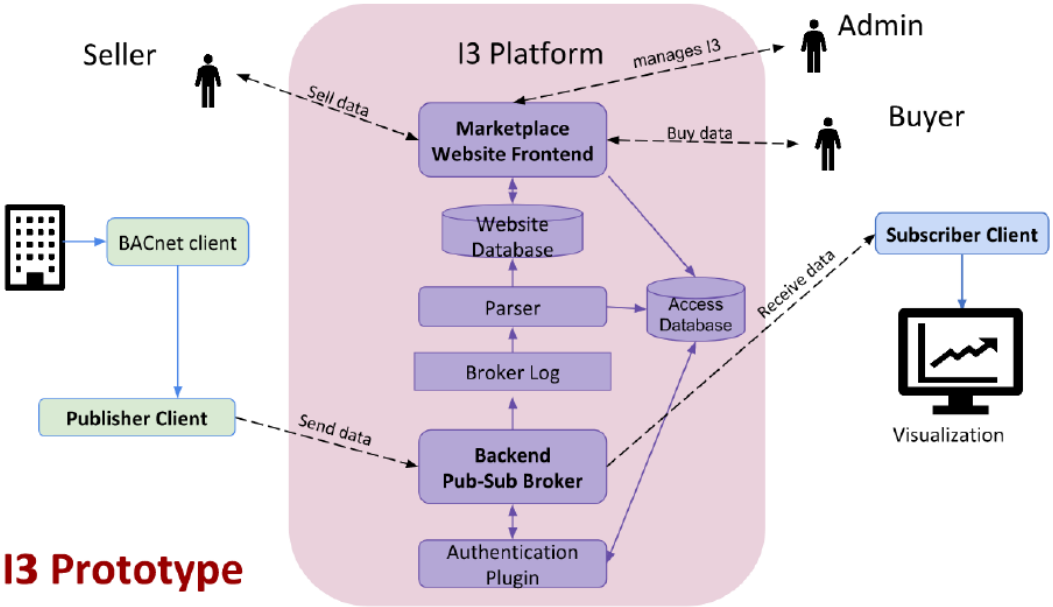


Figure 2.4: I3 Architecture[34]

2.3.3 Software Distribution Marketplaces

The proposed solution is based on a centralized repository of Function Blocks. This architecture is quite alike a package manager or an application store where there is a Software Module Database which stores not only the code necessary to run the application but also the meta-information about the application, as well as feedback information eg. how many times has been downloaded or some malfunction that might have been reported. According to this, several similar solutions for the same objective are described.

From a developer perspective, there are several platforms intended to distribute software packages in order to reuse functionalities already developed by other developers, saving time and increasing the reusability of code. Examples of these platforms are NPM [17], RubyGems [14], PiP [18] or Composer [5]. Usually, these platforms are for open-source redistribution of code. Hence, the commercial trade part of the marketplace paradigm is not applicable.

Generally, the architecture of these platforms includes three main components:

- **The Graphic User Interface** (GUI)
  This component is a Graphic User Interface providing users (developers) with the functionality of finding the package they are looking for as well as to give all the information about the components that can be installed like statistical insights, dependencies on other packages and possible known security or vulnerability issues. This part usually runs on the browser, as the GUI is usually a web-based one.

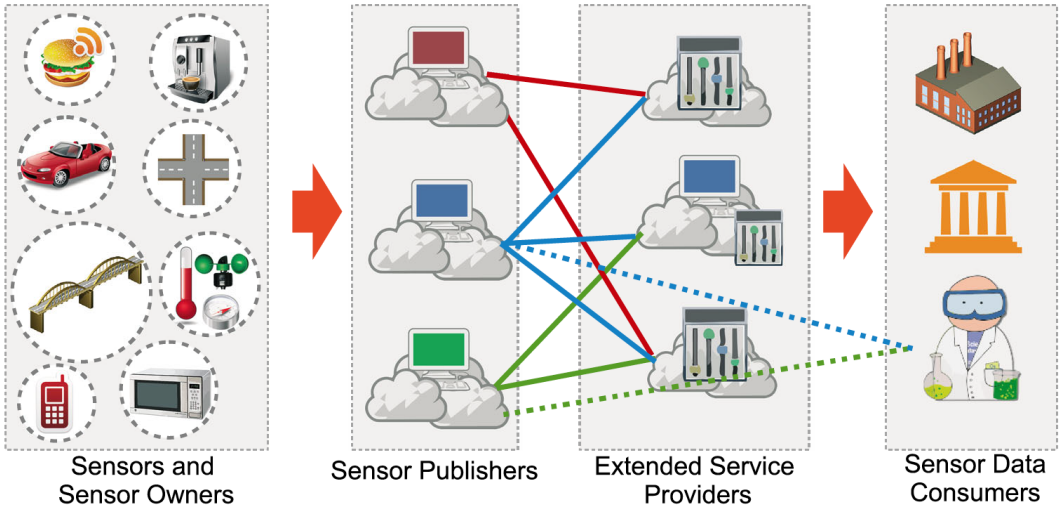


Figure 2.5: Sensing as Service[41]

- **The command-line interface** (CLI)
  Taking into account that this platform is for developers, this GUI-less application allows them to interact with the repository, downloading or uploading new packages and normally runs in the terminal of the developer Operating System (OS).

- **The registry**
  This is the part where the software modules are stored, it is a database containing the code and meta-information of the packages and its function resides on providing the packages when they are queried from the CLI as well as the information source to be displayed on the GUI.

From an end-user point-of-view, the distributed software is, instead of a language-specific software package, a complete application to be run in a device like a personal computer, mobile phone, or other general use computational device. Examples of this kind of platforms can be found in the Apple or Google ecosystems like App Store [1], Google Play [32] or Chrome Web Store [3].

Normally, like the software packages distribution platforms, these marketplaces contain three components.

- **The end-user application**
  This is the tool, running on the application target platform that allows users to search and download the desired application, an example would be Google Play application on an Android device.

- **The registry**
  This component stores the application executable code as well as the metadata related to the applications stored. It serves both the GUI and the application.

- **The GUI**
  Usually a web page to be platform agnostic, this tool allows users to search for a specific application and retrieve all the information regarding it, like the number of downloads, authors, versions, among others.

2.4 Technologies

The final result of this work involves the articulation of three software components, namely a GUI, a Database Management System (DBMS), and a Server-Side Framework capable of serving the database stored data and to communicate with the GUI and embedded devices, implementing OPC-UA protocol for that matter.

2.4.1 User Interface Technologies

"A user interface is the space where interactions between humans and machines occur" [45]. When keyboards and screens appeared, Command-Line Interfaces quickly became the standard of user interfaces. For many, this was already a breakthrough when comparing to the previous era, where there were no keyboards or even screens and only punch cards were used to first write the program and the data and then input them to the computer. [31].

When Personal Computers became standardized and computers started to enter in daily lives of non-technical people, either for recreational or work purposes, a need for a completely new and more friendly way to interact with computers arose. The Graphical User Interface came along as an implementation of the WIMP concept: Windows, Icon, Menus and Pointers[45]. This was the beginning of the GUIs era and from that moment onward most software applications have a GUI.

For this reason, programming languages used to create general use applications have started to provide specific frameworks to accommodate the need for creating GUIs. Some examples are mentioned in table 2.1.

Table 2.1: Native Applications GUI Frameworks

Framework	Language	Target Platform
Swing	Java	Windows, Linux, MacOS
JavaFX	Java	Windows, Linux, MacOS
Kivy	Python	Windows, Linux, MacOS, Android
Shoes	Ruby	Windows, Linux, MacOS
Qt	C++	Windows, Linux, MacOS, iOS, Android
Xcode Storyboards	Swift	iOS
Android Studio Layouts	Java/Kotlin	Android

Another mark in the user interfaces was the advent of touch screens and the need to build user interfaces for significantly smaller devices like mobile phones or tablets. These devices, especially mobile smartphones are quickly replacing laptops for day to day tasks like e-mail, social networking, or calendars. For that reason, developing GUIs that could work for both big and small screens usually requires to develop more than one application to run on different platforms. One of the solutions for this issue seems to be the development of web applications, running in browsers which are a medium that usually all platforms can run. For this reason, several frontend open-source web frameworks were developed to ease the design and implementation of web applications GUIs, avoiding the need of designing different layouts for different size devices and also providing more efficient methods of creating web applications.

Below, in 2.2 it is possible to see a brief summary of the most popular ones.

Table 2.2: Frontend Web Frameworks [2]

Framework	Language	Pros	Cons
React	Javascript	• Frequently Updated • High Performance	• Not SEO friendly • Extra packages for complex apps
Angular	Typescript	• Strong community • 2-way data binding	• Complex • Typescript Learning
Vue	Javascript	• Beginner friendly • Popularity • Simple Syntax • High Performance	• Young • Too Flexible

2.4.2 Database Management Systems Solutions

DBMS have become a crucial component of almost every application. The architecture of a system should be designed taking into account the specific requirements of the application data and in a way that can be quickly retrieved, saved, deleted, or modified.

One can find, essentially, two types of databases, the relational databases and non-relational databases also called SQL databases and NoSQL databases. The first type basically relies on data stored in tables and relations between them, thereby being called relational databases and non-relational databases for non-structured data where different types of data can be stored in one single document.[4].

Usually, when the data structure is rigid and the consistency and integrity of data is more important than efficiency, relational databases are the correct choice due to their ACID (Atomicity, Consistency, Isolation, Durability) properties. This model has been around for several years and most of the complex solutions where the data model of the system is of utmost importance are implemented with this kind of DBMS.

However, in recent years there has been a trend in using non-relational databases since in some cases, data structure and relationships cannot be modeled in advance and data consistency is not an indispensable requirement. Also, the scalability of this kind of DBMS is easier and usually involves adding more resources, while relational databases require a vertical scalability by increasing the power of existing resources. As an example, an e-commerce application, where data integrity is paramount and all the data modeling should be done before even launching the system, should go for the relational databases while an IoT application, with high latency sensors and a lot of unstructured data could be using a non-relational database.

A summary of the most used DBMSs can be seen in table 2.3.

Table 2.3: Database Management Systems [4]

DBMS	Type	Pros	Cons
MySQL	Relational	• Free version • Popularity	• Hard to scale • Not Fully SQL Standard
MariaDB	Relational	• Encryption • High Performance • Broad Set of Features	• Small Community
Oracle	Relational	• Strong tech support • Documentation • Large Capacity	• High Cost • Hard to Learn
PostgreSQL	Relational	• Scalable • Custom Data Types • Open-source • Large Community	• Poor Documentation • Lack of reporting and auditing tools
MongoDB	Non-Relational	• Simple Data Access • Easily Scalable	• Memory Consumption • Data Insecurity
Redis	Non-Relational	• Rapid Solution • Massive Data Processing	• Memory Consumption • No Query Language
Cassandra	Non-Relational	• Data Security • Flexibility	• Slow Reading
Elasticsearch	Non-Relational	• Easily Scalable • Fast Data Processing	• Lack of Multi-Language

2.4.3 Server-side Technologies

The rise of the distributed computation has led to several models. Mainly, it is possible to tell apart two main models for distributed systems. A centralized model, where there is a central node that usually contains the data and can also contain the logic of the system and the rest of the nodes

querying data or functionality to the centralized one. The second model is a decentralized one, where the functionality and data are distributed across all the nodes. Needless to state that the latter is more complex as it involves a lot of engineering efforts so all the nodes have the same state at any given point in time if that is a requirement which usually is. As the desired final solution of this project will be a centralized distributed system, this part is more concerned about this kind. Nevertheless, an example of a decentralized distributed system worth mentioning is the distributed ledger technology based on blockchain for crypto-currencies transactions, due to its growing popularity.[29].

In terms of the centralized distributed systems, perhaps the most known model is the client-server approach, where the central node is the server and it contains, at a given moment in time, all the information about the system state while the other nodes, client nodes, communicate with the server by means of some network channel, requesting information retrieval, updating, creating or deletion, usually on a permission basis.

Perhaps the most known client-server distributed model is the web model, where there is a server that usually executes the logic and contains the data to serve to clients, web browsers, which in turn, displays the retrieved information. On the other way, clients also request, by means of the browser, the creation or modification of the data. The requests are made through the network using the HTTP protocol.[15].

Currently, there are many server-side solutions, each one with advantages and drawbacks and as the proposed solution will be based on this model, a backend node communicating with a web-based client for managing the marketplace, serving static files to the embedded systems and communicating with those embedded systems by means of the OPC-UA protocol, a comparison among some of the most popular frameworks has been conducted and can be seen in [28], summarized in table 2.4.

Table 2.4: Backend Framework Comparison [28]

	Node.js	**PHP**	**Django**	**Rails**
Getting Started	Very Good	Excellent	Very Good	Good
Help and Support	Good	Excellent	Excellent	Excellent
Popularity	Very Good	Excellent	Fair	Very Good
Development Tools and Package Management Systems	Excellent	Good	Fair	Very Good
Environments	Excellent	Poor	Good	Good
Integrations with Databases	Excellent	Very Good	Fair	Very Good
Performance	Excellent	Fair	Very Good	Fair

2.5 Summary and Conclusions

All of these solutions are more focused on a higher level of Process Control and some of those currently only support a pre-defined set of Function Blocks. This is mainly because of the closed scope of the Function Blocks sets these tools have already built-in. However, there are other

solutions, like 4Diac with extended functionality of creating new, customized Function Blocks. It becomes clear that the deployment of the Function Blocks to the network nodes are assumed to be made outside of these tools scope. However, and as one of the objectives of the IEC-61499 is to reach the reconfigurability of the system[40] as a quick way to respond to the deployment of new devices or software or to change the way the product is being manufactured because of some market demand, if there is not a way to quickly deploy Function Blocks to the devices, as per configured through the aforementioned tools, the reconfigurability of the system can take longer and become a tedious and prone to errors task. This is due to the necessity to manually install the software (Function Blocks) needed to run the configuration drawn in the tool to the Runtime Environment where the software is going to run.

On the other hand, the marketplace approach can be seen in several, robust and well-established solutions, either from a developer perspective in software package managers like NPM, PiP, or RubyGems or from an end-user point of view in Applications Marketplaces like AppStore or GooglePlay.

At last, in the IoT context, three models to exchange data between sensors owners and data-driven applications developers were presented. These two projects prove that the marketplace paradigm is also entering the IoT context.

It was seen that current Distributed Control Systems tools for CPPSs have limitations addressing the issue of software reusability and distribution. Although the IEC-61499 standard aims for the standardization of this kind of systems and tries to make the programming of distributed applications more accessible, there is no option that provides the automatic installation of software modules in the embedded devices on a distributed network. For that reason, this works aims to tackle that issue, providing a solution to integrate automatic distribution of Function Blocks onto the RE of the devices that need to run the FBs, in the context of IEC-61499 standard. The proposed solution will embed some concepts of the three research areas in this chapter, the IEC-61499 tools, IoT Marketplaces, and Software Package Distribution Systems.

At last, a brief summary of the state-of-the-art tools based on the technological requirements was conducted so a clear picture of the available frameworks and options was described in order to support the best choice taking into account the proposed solution for this work.

Chapter 3

Solution Architecture

The architecture of the system has been defined taking into account the integrations needed to be made. At a higher level, the system should integrate with the FB IDE, in this case, the 4Diac IDE and, concerning the devices, its integration uses a RE, in particular, the Dinasore RE. One of the main functionalities and objectives of building such a system is to automate the management and the access of these software modules, the IEC-61499 FBs, used by the embedded systems.

As a result, the system actuates as a central repository, distributing software among clients upon requests. According to this, the system needs to be manageable through an application and needs to have meanings to serve files to other components.

The architecture defined has three main components, clearly visible in figure 3.1. One of these components, the Dinasore RE, already existed before the development of this solution. Nevertheless, it is important to form part of the schema, so one can easily see where this component fits in the overall final solution: the Jurassic Park Backend, the Smart Components, and the Web Application.

3.1 Jurassic Park Backend

This component contains the application servers, the database server and should run on the machine where the 4Diac is running.

The Application server runs a Node Application that exposes an API composed by a set of Web Services to manage the creation, edition, deletion, and retrieval of the FBs. This application also functions as an OPC-UA client connected to the different OPC-UA Servers running on the embedded devices so it can track them in real-time and serve this information to the Web Application through a TCP WebSocket connection. The Application Server also runs an FTP Application to serve the needed files to the different embedded systems running on the network. The Application Server runs on the machine where the instance of the 4Diac IDE is running as this integration is made through the machine file system.

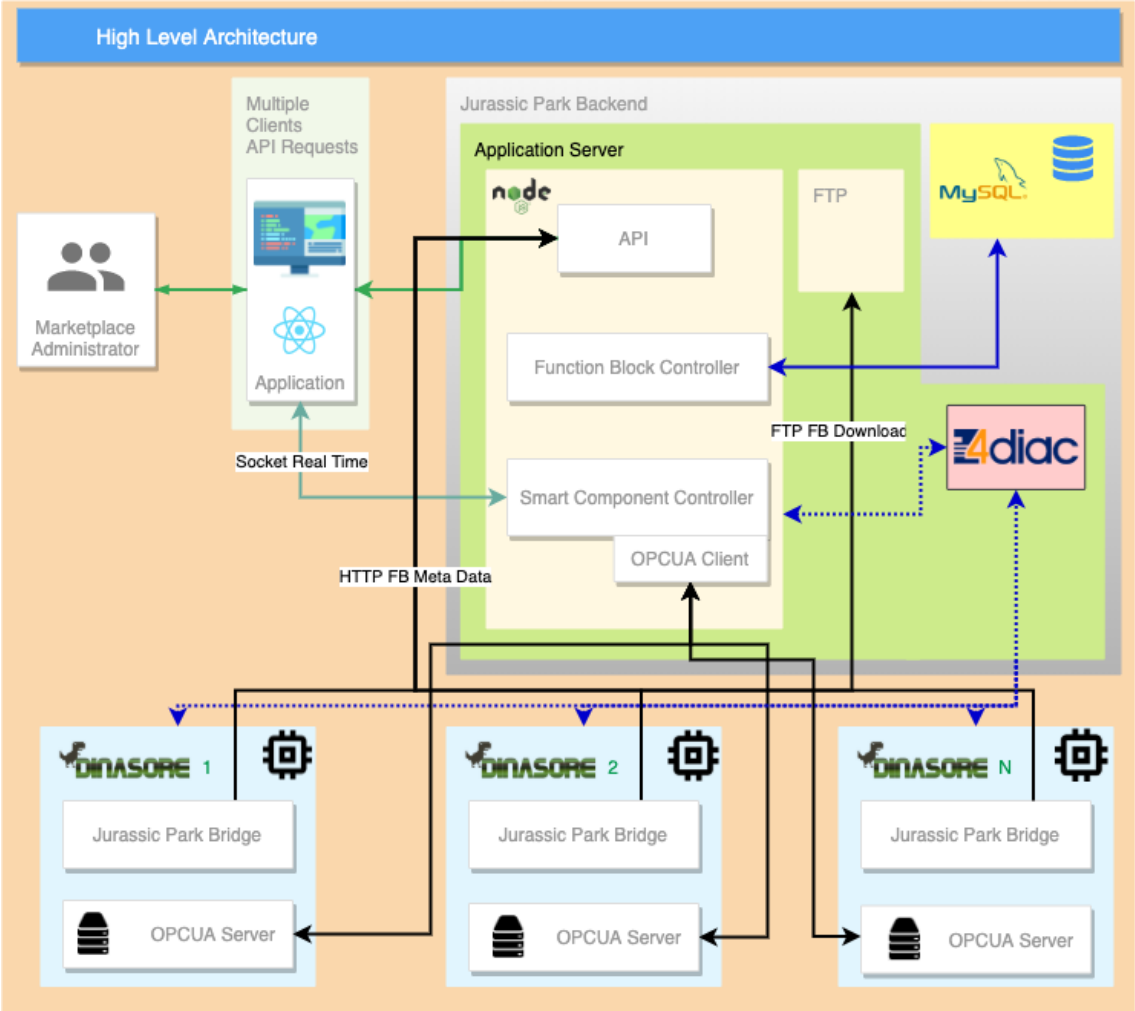


Figure 3.1: Overall Architecture

The Backend also comprises an instance of a MySQL Server Database, which stores all the meta-information related to the software modules. The Node application communicates with this server in order to perform the operations that manage the creation, edition, deletion, and retrieval of the modules.

It is also important to mention that the most important entity in the solution is the FB, a reusable piece of software that must be distributed through the different running devices on the network. After all, the final solution is a central repository of these modules. Each of these modules is composed by two main parts, the implementation part, consisting in a Python file where the logic of the FB is implemented, and a metadata part, an XML, which stores the meta-information about the module, like the data inputs, outputs, and name as explained previously in chapter 2.2.4.1. These are the two files that will be served every time a running device request a given module. Hence, each FB will have a folder in the Application Server where these files are stored to be served by the FTP server.

3.2 Smart Components (Dinasore RE)

The smart components are running the Dinasore Runtime, developed in Python by Systec, a Faculdade de Engenharia da Universidade do Porto laboratory. This is the runtime where the function blocks are running and, as it can be seen in figure 3.1, has now a new component, developed in the context of this project, the Jurassic Park Bridge that takes care of the communication with the Jurassic Park Backend. This communication is made both via HTTP and FTP in different moments to achieve different purposes, as it is explained in section 4.3.5.2. During the development process, and in a development environment, a Docker environment was used. With this approach is easy to simulate a network of several embedded devices.

3.3 Web Application

The third component in the architecture is a Web Application used both by the Jurassic Park Marketplace administrators and all the users that need to track the devices in real-time. This application allows, in a friendly way, all the operations required to manage the market (creation, edition, deletion, and retrieval of modules). These operations are accomplished by calling the web services exposed by the Application Server, more specifically in the Node Application.

3.4 Technologies

Before starting the project, and according to what has been described in chapter 2, there were some decisions to take in regard to the technologies to be used throughout the development of the project. Therefore, there are three main components that forms the final solution, the Backend, including the HTTP API for the management of the FBs and the Real-Time Engine to monitor the devices, the Database Server, and the Web Application, as the User Interface.

3.4.1 Backend Technologies

As mentioned in figure **??**, nowadays, there are four most used Backend frameworks for building web-related Applications, including only APIs or other Web Applications platforms. The choice had to be made taking into account that not only an API was going to be developed but also two other fundamental components had to form part of the final solution, namely the real-time communication with the Web Application to send live information about the devices, and the communication, via OPC-UA protocol, with the smart components, themselves. The decision taken was to use Node.js as the backend framework for this project, with the following advantages.

- **Experience**: Taking into account that the time to develop such a project was not much, the technologies to be chosen should not require a learning phase. Node.js is a JavaScript framework that allows the execution of JavaScript on server-side, outside the browser and is one where an adapting phase was not going to be needed.

- **Language Consistency**: The framework choice for the Web Application was going to be one of the most recent trends of the frontend web development. All of these frameworks use JavaScript has the programming language, since it is the browser natural language. Therefore, choosing a framework that utilizes the same language for the backend would ease a lot the development process, avoiding the overhead of being constantly changing the programming language, especially on an agile oriented development process is made by requirement, developing all the required parts for it to be fully working.

- **Tools and Package Management System**: The community that has been developed around Node.js led to the growth of NPM, the Node.js ecosystem package manager and therefore the number of libraries and packages available are huge and the probability of finding a specific tool or library to accomplish a specific requirement is very high.

- **Performance**: Node.js and JavaScript in general follow the paradigm of Single Thread Non-Block I/O which has been proved to be efficient when it comes to serving requests. The only downside of this technology is for heavy CPU computation tasks, which is not the case for this project and therefore making Node.js a good choice also from a performance point of view.

- **Portability**: Node.js is available for the most used Operating Systems, Linux, macOs, and Windows.

3.4.1.1 Third-Party Modules Used

As mentioned before in this section, one of the greatest advantages of using Node.js is the number of third-party modules built ready to be used to accomplish a specific desired task and not having to write all the code from scratch. The list of the main ones is describe next.

- **Express**: Express is a Web Application framework with a lot of functionalities. Among those are the routing of HTTP requests and parsing the payload of the requests.

- **Socket IO**: Socket IO is a JavaScript Library to build realtime web applications. It has two main parts, a Server and a Client. The server runs on the server-side and the client runs on the browser.

- **MySQL**: This is the package used to integrate with the MySQL Database Server allowing the execution of queries to the database.

- **Moment**: Moment is a very useful JavaScript package to handle operations with dates and times like convert to different formats and timezones.

- **Typescript**: Typescript is the module to allow writing the code in TypeScript language instead of writing in pure JavaScript. The package brings the transpiler that transpiles the code from TypeScript to runnable JavaScript. TypeScript is a superset of JavaScript and

adds the advantage of adding types to JavaScript language which makes the code easier to debug and maintain.

- **Node-Fetch**: This is a package to abstract the complexity of making HTTP requests. It uses the same API of the original window.fetch used in JavaScript for the browser. This way, making HTTP requests is exactly the same whether is made from the browser or from the server.

- **Node-Opcua**: This is the package that implements the OPC-UA protocol. The submodule used was the Node-Opcua Client to connect to the servers that are running in each instance of the Dinasore RE.

3.4.1.2 Database Server

As part of the backend side of the project, there was a need to store some data persistently. As mentioned in section 4.2, some entities must be saved in non-volatile memory. These entities are mostly related to the software modules to be served, the FBs. Like was mentioned in chapter 2, the choice of database technologies is vast and there are some characteristics of the system that influence one to choose between a relational model database and a non-relational model database. In this case, the consistency of the data is a paramount requirement, there are well-established relationship between the data entities and the management of the FBs require ACID properties, especially because the Web Application can be used by more than one user at the same time. According to these necessities, MySQL was chosen among the relational database options mainly because of the following advantages.

- **Experience**: As already mentioned before, the time to develop this project was not much hence there was a need to avoid choosing unknown technologies or tools that would require a learning phase. The developer had already worked before with this tool which made it the perfect choice to accommodate this condition.

- **Popularity**: The community around MySQL is large because it has a free open-source version. This could save time whenever a difficulty came around as the likelihood of another member of the community had already faced the same issue was high and therefore the issue would be easily overcome.

- **Documentation**: The documentation available is very detailed which can overcome the disadvantage of MySQL not being fully SQL standard, so sometimes a documentation consultation is needed in order to check the correct syntax for a specific case.

3.4.2 Smart Components

The RE in which this solution must be integrated is the Dinasore, written in Python, aiming the ease of the implementation of machine learning algorithms to improve the manufacturing production process in factories. For this reason, the most viable strategy to use here is to contribute for the

Dinasore project and integrate a new functionality package capable of making the bridge between the RE and the newly created platform allowing the communication and the information exchange between them, which is fundamental for the implementation of the desired requirements.

To tackle the low-level specification of HTTP requests, *requests* dependency package was added into the Dinasore project.

3.4.3 Web Application

There are several ways a GUI application can be built, as mentioned in section 2.4.1. This choice is highly dependent on the project characteristics and requirements. Nevertheless, nowadays, there is a trend for Web Applications, a Web Application is one that runs in a Web Browser, its building blocks are built with HTML language, styled with CSS, and makes use of JavaScript as a programming language to add logic to the application. The main advantage behind this is portability and almost all Operating Systems have a browser and they all render in the same way, the way that has been just explained before. In the past, the architecture of this kind of applications followed mostly a server-side render approach, with technologies like PHP or Java, which means that all the UI elements were rendered upon data transmitted by the client and sent to the browser client, the browser just add to render the HTML file received. This approach means that for every request, a new entire HTML page had to be rendered and sent to the client. There are still a lot of Web Applications built with this architecture. However, in the past years a new paradigm came to light. The idea of decoupling the backend just for the logic and data-generation from the frontend, to build the UI elements based on the data received. This new way of developing a Web Application is called SPA, standing for Single Page Application, where the application runs only on one page and whenever some part of the page needs to be updated with different UI elements, is the frontend JavaScript code that does the job. In a first load, the server sends all the JavaScript code necessary to all rendering of the UI elements, without rendering it, it is then the job of the browser to generate the HTML from the JavaScript received. From that moment, the frontend side of the application just hit the correct endpoints to collect or create data in the server and update the UI whenever is necessary. With this approach, it is even possible to create a Web Application without any server, if the data is only local or if there is no data whatsoever.

Following this trend, some UI JavaScript libraries came up, as revised in section 2.4.1. In the end, the library chosen was the React[13], a library to create GUI that follows a declarative programming paradigm, instead of a traditional imperative approach. This means that instead of directly manipulating the UI elements, the UI elements are automatically updated in a reaction of a change in the application's internal state. The program is written telling what should be the appearance of the UI elements for each internal state and the user interaction (or other external events) change that internal state.

3.4.3.1 Third-Party Modules Used

- **Experience**: For the same reason as in the backend, the developer has already some experience not only with the language (JavaScript/TypeScript) but also with this specific library. This is a huge advantage as it avoids a preliminary phase of learning the tool.
- **Performance**: Among all the libraries this is considered the most efficient one due to the way they build the interface and the concept of Virtual DOM, which is a copy of the real DOM tree currently being rendering in the browser. When something changes, instead of rerendering all the actual DOM, React compares the previous Virtual DOM with the new one and just rerender the DOM objects that actually changed, making the manipulation of the DOM really fast.
- **Community**: Among all the JavaScript frameworks, React is the one with the biggest community which makes it easy to find external tools and packages when needed, avoiding writing code from scratch for something that was already developed. It is possible to use the NPM manager to install JavaScript dependencies.
- **Typescript**: Typescript transpiler. Described in the previous section 3.4.1.1.
- **Material UI**: This is a UI component library ready to use. It contains buttons, forms, tables, and more that can be used in a react application. These components are also highly customizable.
- **React Google Chart**: A charts library developed by Google.
- **Socket IO**: the client-side of the Socket IO package, a high-level library to use Web Sockets for real-time communication.

3.4.4 FTP Server

To serve the files required by the smart components, an FTP server was used. The choice was the VSFTPD solution[21]. This component was chosen because of the simplicity of configuration, it is an FTP server for Linux Platform. However, the system is built in such a way that other kinds of FTP servers can be used as the only configuration needed in the other components is to set the port and address of the FTP server.

Chapter 4

Implementation

This chapter describes and details the development process of this project as well as the result of the development in terms of its components and it is divided into three sections, the development process, in section 4.1, the data model, in section 4.2 and a detailed explanation of all the system components in section 4.3.

4.1 Development Process

This project was managed through an agile development process, more specifically, following the SCRUM methodology. The following roles were defined:

- **Product Owner (PO)** - The co-supervisor, responsible for the global vision of the project.
- **Scum Master + Development Team** - In this case, as there is only one person developing the system, the scrum master and the development team roles were performed by the same person.

In the first phase, all the requirements were listed with the Product Owner taking the responsibility of ordering those requirements. Furthermore, the requirements were divided into four main categories: Backend, UI, and Integration as simple categories and User Story, as a set of simple requirements and representing a System Functionality.

Secondly, the requirements were grouped into milestones. These milestones represented the sprints where the requirements were developed.

At the end of each sprint, the functionalities were ready and presented to the PO. Moreover, in the end of each sprint, a retrospective has been made in order to improve or readjust the next milestones to come.

At last, the development process was made with the help of several tools like a Text Editor with debug capabilities and an integrated runtime, a version control system to keep track of the

versions being developed and with scrum management capabilities and also a browser extension to debug the web application.

At each moment, the requirements were at one of the next four states.

- **Product Backlog** - Requirement pending to be developed.
- **Sprint Backlog** - Requirement to be developed in the current sprint but not yet started.
- **In Progress** - Requirement being developed at the current moment.
- **Done** - Requirement finalized and ready to be integrated.

The aforementioned management can be seen in figure 4.1 and an example of a closed sprint with the requirements developed can be seen in figure 4.2

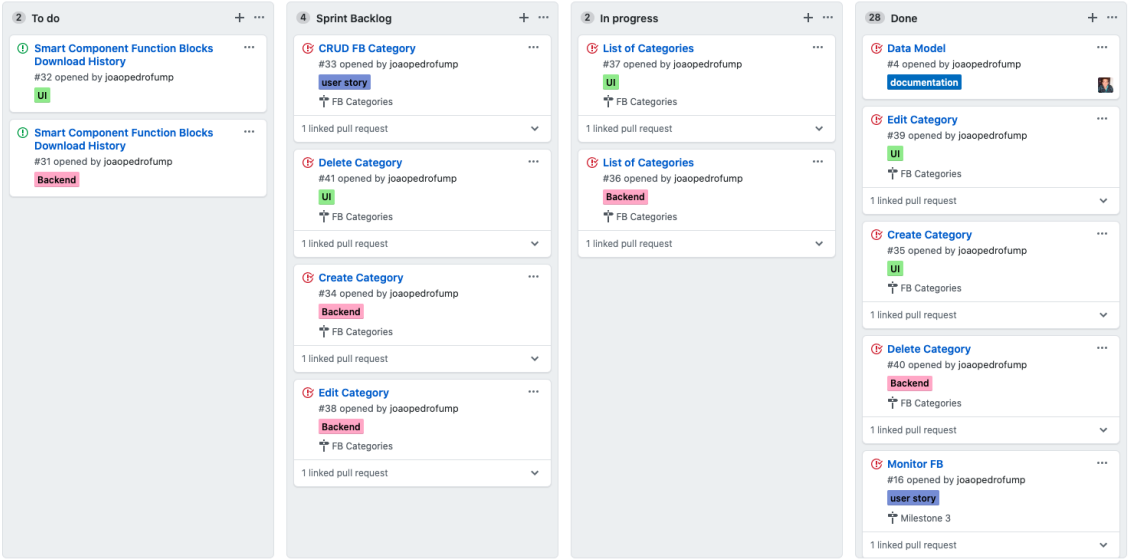


Figure 4.1: Requirements Management

The list of all system requirements can be found in appendix A.1.

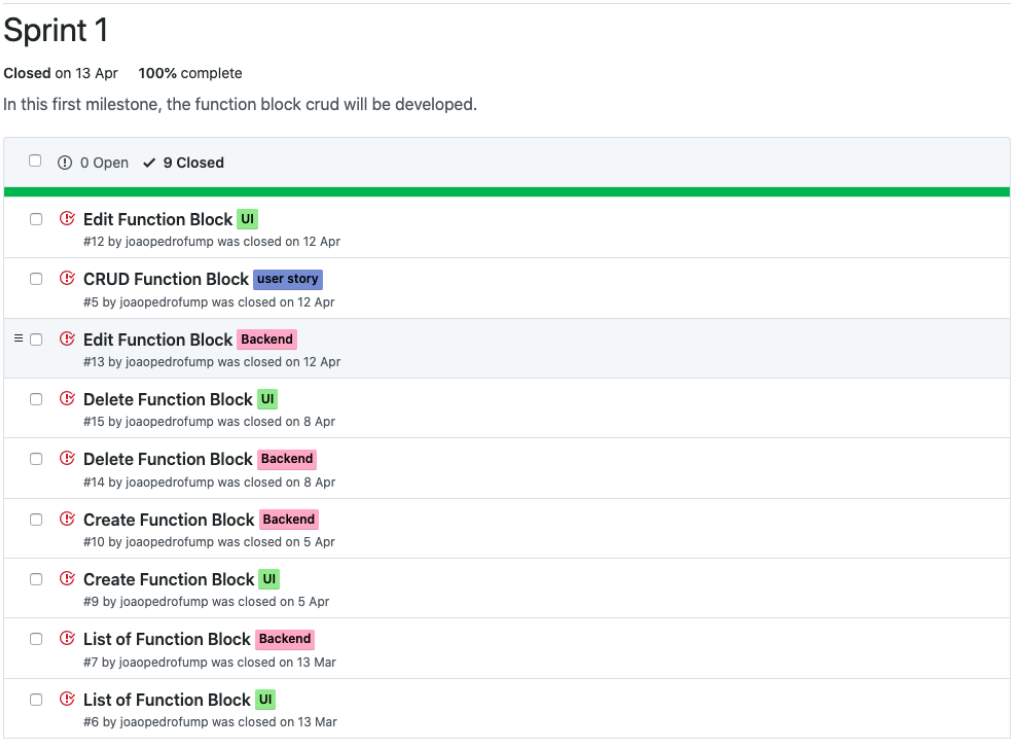


Figure 4.2: Sprint Example

4.2 Data Model

One of the essential parts of building an information system is to describe the model of that information. This model is depicted in figure 4.3.

For the sake of understanding, the entities represented in the data model are divided into two categories: Likely Changing Entity, Unlikely Changing Entity, summarized in table 4.1.

Table 4.1: Data Model Entity Categories

Entity	**Category**
Function Block	Unlikely Changing Entity
FB Category	Unlikely Changing Entity
Variable	Unlikely Changing Entity
Event	Unlikely Changing Entity
Function Block External Dependency	Unlikely Changing Entity
External Dependency	Unlikely Changing Entity
Smart Component	Likely Changing Entity
FB Instance	Likely Changing Entity

This distinction is important as it determined where the information is stored. For the entities unlikely to change much, this information is stored in a persistent database and is accessible even between the Application Server restarts. Most of this data is related to the meta-information of

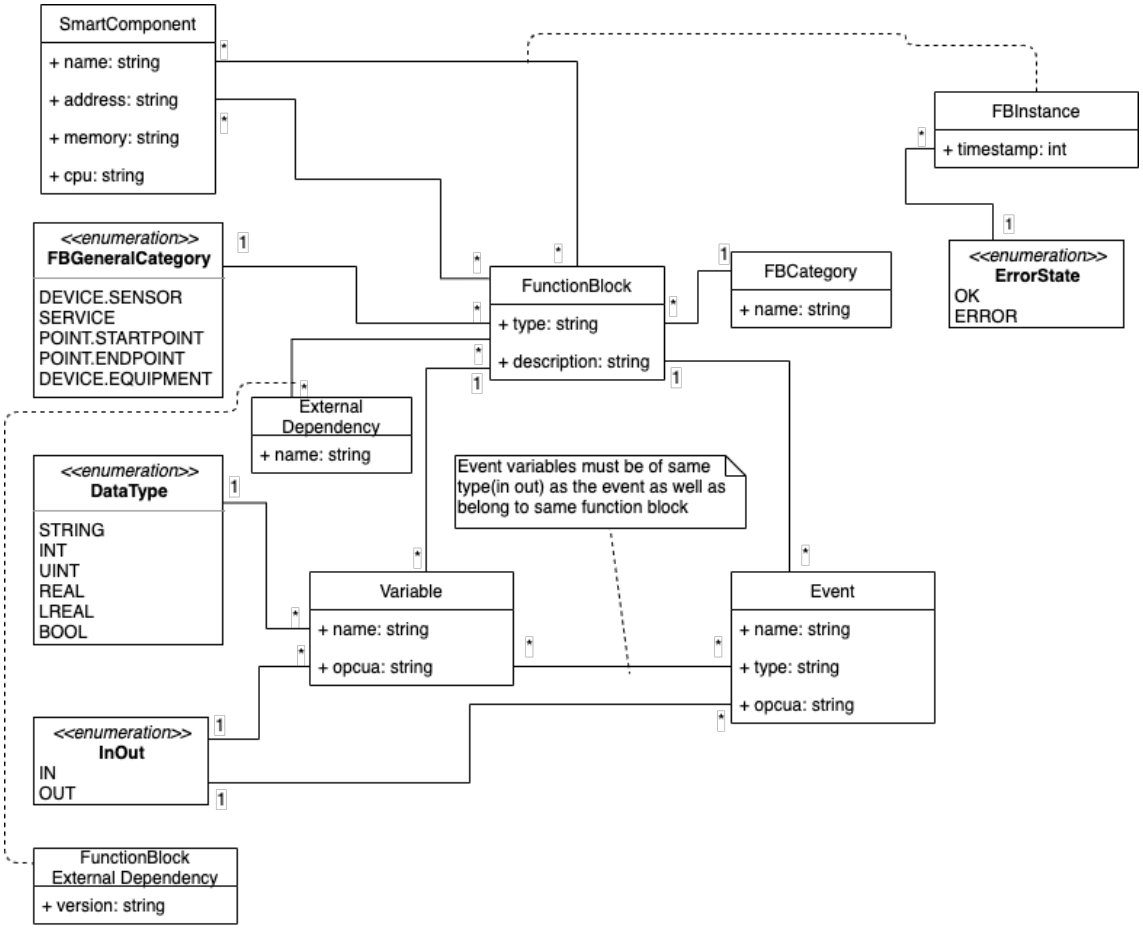


Figure 4.3: Jurassic Park Data Model

the modules, as can be seen in figure 4.3. Besides, the information about the repository, the actual modules, should be saved in a persistent way otherwise one would need to create all the modules every time there was a need to shut down the server. On the other hand, there are data that is likely to change a lot, the data related to the smart components running in a given moment, as well as their FB instances. As this information is changing at a fast pace, there is no point in storing it in a persistent database. As it is detailed in the next chapters, whenever a device is connected, it will establish a communication with the server, signaling that event, and from that moment, that device will be monitored and its status saved in volatile memory, always updating the Web Application whenever a change is detected.

4.3 System Components

In this section, the details of the system components will be described. There are two main components, the first is the Application Server, containing both the Node application server and an FTP server, the Node application exposes the web services to communicate both with the Web Application and the Smart Components and the FTP server to serve files upon requests from the smart components. The second component is a Web Application, built with React, a JavaScript library to build User Interfaces for the Web.

Furthermore, two integrations have been made to close the whole solution. As said in chapter 2, IEC-61499 comprises two areas. First, the IDE, which enables the draw of FBs pipelines/-configurations, and second, the RE where the FBs actually execute, distributed among the smart devices of the network controlling the production system. Hence, two integrations have been made with these two parts, as explained in the next chapters.

4.3.1 Node Server

This component is the main part of the Jurassic Park Backend, it has two main responsibilities. The first one is to expose an HTTP REST API both to the Web Application and to the Smart Components. Each of the API endpoints will be described in the next chapter and a request to each of these endpoints will trigger a specific operation. This schema can be seen in figure 4.4

The second responsibility is the exchange of real-time data between both the Smart Components and the Web Application. The communication with the Smart Devices is done through an OPC-UA Client/Server Communication, an industry-standard protocol to track industrial equipment state [22]. The server side runs in each of the Smart Components in the Dinasore RE and it was already developed outside of this project context. The communication with the Web Application to exchange the data is done through a WebSocket channel. Therefore and summarizing, the Node Server establishes one communication channel with each of the Smart Component connected in the network via OPC-UA Client/Server Protocol, functioning as the client of each of the servers and another communication channel with the application through a persistent TCP

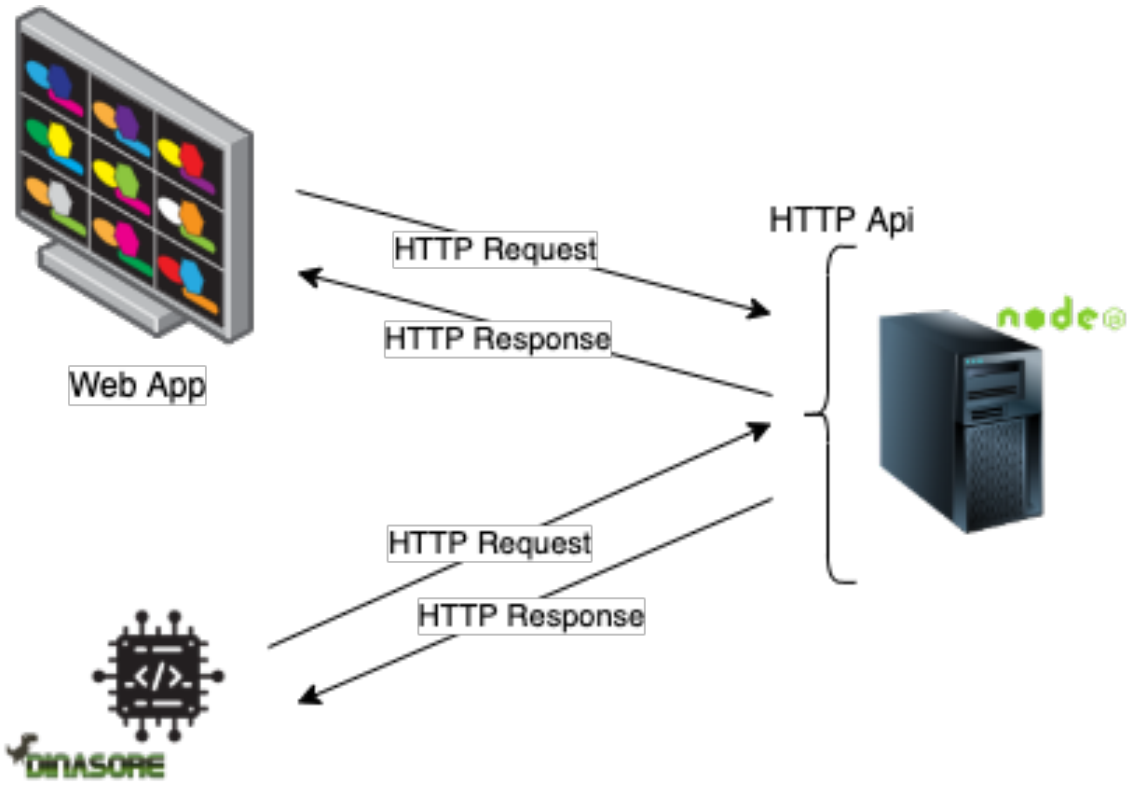


Figure 4.4: HTTP Communication

WebSocket. The package used to manage this communication was Socket.IO [20], a javascript package to abstract the low-level complexity of TCP Web sockets. This communication schema can be seen in figure 4.5.

To accomplish these two functionalities the Node Server has been divided into several components. These components will be described in the next sections and depicted in the diagram of the components that can be seen in figure 4.6.

4.3.1.1 API

The API component uses a JavaScript package to handle and correctly route the HTTP requests: Express [9]. This package can abstract away the complexity of HTTP requests, handling the methods and the information sent with the requests among other functionalities on top of a simple HTTP server that node has already built-in.

This component is a composition of other components, which were called API modules. Each of these modules handles a list of requests and the set of all modules forms the full HTTP Rest API.

There are two API modules, the first one is the Function Block Module, where the endpoints to manage the CRUD operations of the Jurassic Park Marketplace software modules are defined,

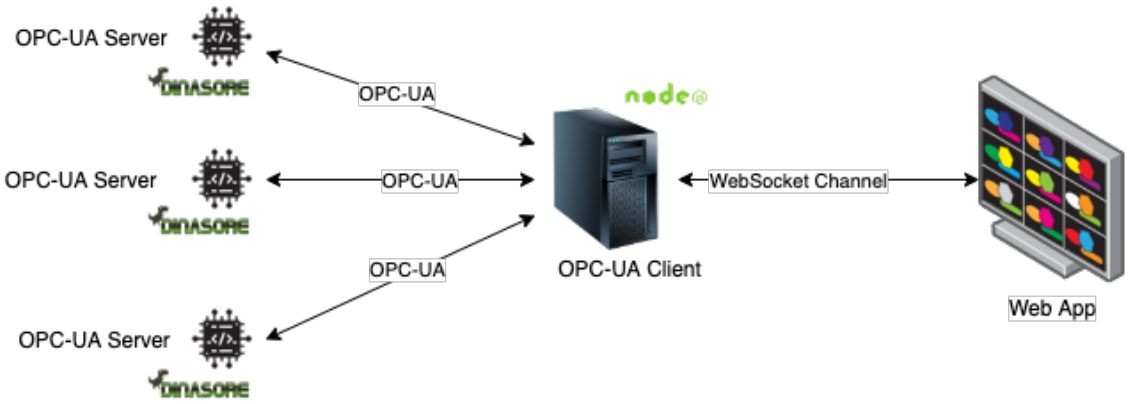


Figure 4.5: Real-Time Communication

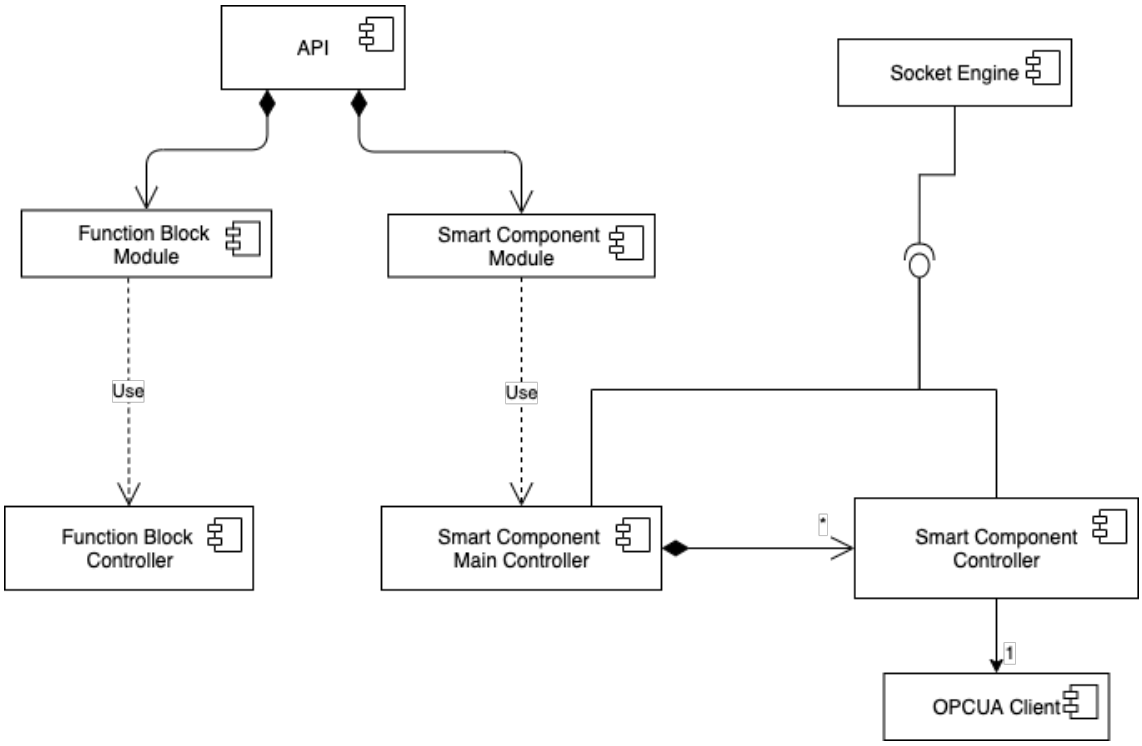


Figure 4.6: Components Diagram

and a second API module, responsible to expose an endpoint for the Dinasore RE running Smart Components.

In the next paragraphs, the endpoints are described. All the endpoints are expecting data in JSON format and will respond with data in the same format.

- **Function Block API Module Endpoints**
  - **GET /function-block**: This endpoint returns a list of all the FBs of the marketplace. This is just the metadata of the FBs and is to be used by the web application.
  - **POST /function-block/**: Calling this endpoint will create a new FB in the market. In order to correctly create the function block, the structure needs to be compliant with the model described in section 4.2. It is important to notice that one of the fields of the FB is a base64 representation of the Python implementation file.
  - **PUT /function-block/:id**: The API also provides the capability of editing a specific FB, in this case, identified by the FB id, present in the URL parameters. All the fields can be updated and even the implementation file can be changed, as long as it is presented in the body of the request.
  - **DELETE /function-block/:id**: This is the endpoint to be called to delete the FB identified by the URL parameter.
  - **GET /function-block-category/**: This is the endpoint that retrieves a list of all created categories in the system.
  - **POST /function-block-category/**: Calling this endpoint will create a new FB category in the marketplace.
  - **PUT /function-block-category/:id** This endpoint edits the category identified by the id present in the URL parameters.
  - **DELETE /function-block-category/:id**: This endpoint deletes the category identified by the id present in the URL parameters. It is important to notice that the integrity constraint in the database only allows the deletion of a category as long as there is no FB with that category.
- **Smart Components API Module Endpoints**:
  - **POST /smart-component/** This endpoint is to be used by the Smart Component devices in order to announce their connection to the network. In this communication, the Smart Device sends its address and the port where its OPC-UA Server is running so the Marketplace can try to establish a new connection to the device.
  - **GET /function-block/:type**: This endpoint is to be used by the Smart Components in order to get the detail of a given FB that they might need to download or know the dependencies. This FB is identified by its type name, present in the URL parameters.

It is important to mention that, as can be seen in figure 4.6, the API modules are just the public interface to the external world of Jurassic Park. These components make use of the specific controllers to accomplish the required tasks. These controllers are described in section 4.3.1.3.

4.3.1.2 Socket Engine

Among the main objectives of the project, there are two that are closely related to the concept of real-time communication, the monitoring of the Smart Components, and the error detection in their running FB instances. Real-time communication cannot be achieved with a REST API, a REST API functions in an HTTP request basis where an HTTP request is sent, the responses are sent back from the server, either successfully or with errors, and the connection is closed.

To establish real-time communication between the Web Application running in a browser and a server there has to be a permanent communication channel between these two entities. In order to achieve this, the approach was to use WebSocket technology. As its documentation page [20] states, this technology "enables Web applications to maintain bidirectional communications with server-side processes".

To abstract away the complexity of having to manage one single connection with every client running the Web Application, a third-party package was used. The Socket IO package, among other functionalities, handles re-connections and multiplexing. It has a simple API that enables sending data to all connected sockets and to create socket namespaces to organize the set of sockets connected into areas of the application. The namespaces in the Socket Engine context can be seen as the equivalents of the REST API endpoints. In the context of this project, there two kinds of namespaces.

- **/smart-component**: The sockets connected to this namespace will receive live data about all the main information of all the smart devices connected. This is helpful for creating a live list of connected devices and their status. It is described in more detail in section 4.3.4.

- **/smart-component/:id**: The sockets connected to this namespace will receive real-time data about the Smart Component identified with the id. This namespace will be used by the part of the Web Application responsible to show more detailed information about a specific Smart Component. It is also shown in section 4.3.4.

As it is shown in figure 4.6, this component provides an interface used by both the Smart Component Main Controller and the Smart Component Controller in order to send data to the connected sockets functioning as a bridge between the controllers and the client connected and running the Web Application on a browser.

4.3.1.3 Controllers

Two main controllers were developed. Each controller is meant to control a specific area of the application. The first one is the Function Block Controller to control the CRUD Operations of

the FBs and FB Categories and the second controller is the Smart Component Controller, a more complex one, to control the smart component state at any given moment, as depicted in figure 4.5.

The Function Block Controller was developed as a singleton class with methods to create, update, delete and retrieve function blocks and categories. This controller is also dependent on some utility functions to integrate with the database and with the file system. This behavior is related to the fact that this controller needs to manage the metadata of the FB and save it on the database with the implementation and metadata file. For this reason, the public folder of the server is organized in the following manner. Each of the categories created has a folder, inside that folder, there is a folder containing both the Python implementation file and the XML file with the meta-information of the FB. This can be seen in figure 4.7.

Figure 4.7: Function Block Folder Structure

It is also important to notice that this component will read the Jurassic Park configuration file, described in section 4.3.1.5.

The second controller of the Node application is the Smart Component Main Controller. This controller is more complex than the Function Block Controller as can be seen in the Component Diagram described in figure 4.6. This controller, just like the Function Block Controller, is a singleton class. However, this controller will contain a list of sub-components, Smart Component Controllers, one for each smart component device connected. This component will read a

hidden file which contains a list of devices connected when the server was last shut down and create one Smart Component Controller for each of those devices. At any moment, when a new device connects to the network, a new Smart Component Controller is created. Each of the Smart Component Controller creates an OPC-UA client to connect to the OPC-UA server running on the smart device. This OPC-UA client subscribes to changes in the OPC-UA server, meaning that is notified when the data of the Smart Component changes, like its resources usage or the state of FB instances running in the device.

Both the Smart Component Main Controller and each of Smart Component Controllers communicate with the browser clients through the Socket Engine, sending them updated information about the state of the devices. This state includes resource consumption and the error state of the FB instances that are currently running in the device. The UI part of these functionalities can be seen in section 4.3.4.

4.3.1.4 Models

When building a REST API, a crucial part is the design of the models. These models, in the programming language used, need to correctly mirror the models designed in the database so the data is always correctly structured to send to the clients. Doing so also allows the avoidance of misinterpretations when reading the data. This can only be done with a typed programming language and was one of the reasons TypeScript was chosen instead of pure JavaScript.

The models were designed by defining interfaces, specifying not only the mandatory or optional fields but also the linking between entities. Instead of defining simple interfaces, they could have been defined as classes. However, this is just a simple mapping between the data representation in the database language and the server-side language and the introduction of classes could only make sense if there was the need to add methods to the models, which was not the case. For the models defined as a set of predefined constants, an enumeration has been used.

It is important to notice that not only the data exchanged via the REST API will follow the structure of these models but also the data exchanged via the Socket Engine. After all, all the data transmitted in the communication of all the system components will be done following the structure of the entities defined in these models.

- FB General Category
- External Dependency
- Function Block
- FB Instance
- Inout Type
- Data Type
- Event

- Variable
- Event Variable
- Smart Component

The details of the implementation of these models are described in Appendix A.2. These are all the entities comprising the system in terms of data. This is the actual representation of the information saved and transmitted between the different parts of the solution and, after all, constitute the language the different components speak when they need to communicate. All the models depicted in the previous listings are self-explanatory of the different entities that compose the system but are important to understand how the data is linked and how the information is exchanged among the components of the solution.

4.3.1.5 Configuration and Startup

In order to correctly startup the server some configurations were set up. These configurations are not only related to the internal functioning of the server but also with the communication with the other components, namely the database, the Web Application clients, and the Smart Components.

The first part of the configuration parameters is the environment variables used that are described next.

- **APP_PORT**: This is the port where the Node server is running and listening for requests.
- **SOCKET_PORT**: This is the port where the socket engine is running and listening for client requests in order to establish socket communication with them.
- **MYSQL_DATABASE_HOST**: This is the address or the nameserver where the instance of MySQL Database is running.
- **MYSQL_DATABASE_PORT**: This is the port where the instance of the MySQL database is running on the corresponding server.
- **MYSQL_DATABASE_NAME**: This is the name of the database used to store the data.
- **MYSQL_DATABASE_USER**: This is the username of a user with access to the database specified in the previous environment variable.
- **MYSQL_DATABASE_PWD**: This is the password of a user with access to the database specified in the previous environment variable.

Defining environment variables is a way of avoiding to change the source code when some of these variables have to change. As an example, if the database server changes its location, the only thing to change in the configuration is the corresponding environment variable, without the need to recompile the source code. The second part of the configuration is in the file *jurassic.config.JSON*.

This is a file written in *JSON* and specifies some extra configuration properties. This file was created thinking about the evolution of the solution. Adding new configurations to this file is easier than keep adding new environment variables as the project grows and needs more configuration parameters. Summarizing, the technical configurations regarding the database and the server are done via the environment variables whereas the configuration related to high-level Jurassic Park functionalities is done through a configuration file. Therefore, the current configurations present in the file are described next.

- **_4diacLib**: This the path to the folder where 4diac fetches the meta-data of the available FBs. In this path, only the XML metadata file of each FB will be saved.
- **functionBlocksFolder**: This is the path where both Python implementation file and the XML file will be saved. There is a folder for each category, and inside each category one folder for each FB, containing the files.

As mentioned before, as the system evolves with new functionalities in the future, the configurations can be added here. The source code of the project has a function to read this configuration file and create an object with all the configuration. As this is a *JSON*, nested configurations are also possible instead of a simple flat list of variables. Being a *JSON* file, the conversion to a JavaScript object is straightforward.

4.3.2 FTP Server

One of the main objectives of this project is to ease the process of pre-configuration of a production system. By this, the installation of the software modules, the FBs, in all the devices controlled by the process, the Smart Components. In this scenario, the first requirement is to have them centralized and accessible, which can be accomplished with all the functionalities to manage the CRUD operations, the storage both in the database and in the File System. The second requirement is to distribute these packages, comprising the XML file and the Python implementation file among all the Smart Components that need to have them installed in order to execute them. For this reason, the approach was to run, in the same machine an instance of an FTP server so the Smart Components could download the files. The choice was the VSFTPD standing for *Very Secure FTP Daemon*. As this is not coupled to the main Node Application, the FTP server can change in the future and the only requirements for the configuration are described next.

- **Share Folder**: Define the share folder as the same defined in the Node application configuration as the folder where the FBs files are stored so the FTP clients can download them.
- **Credentials**: Define the username and password to query the server for downloading the files.

A high-level process representation of a FB download process is represented in figure 4.8. A detailed sequence diagram of a Function Block Download is described in section 5.1.2.

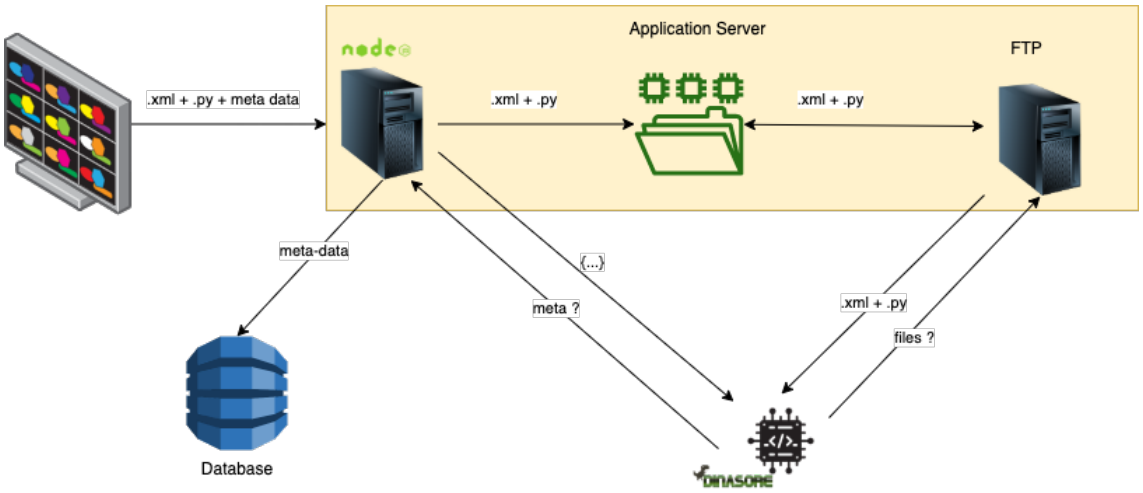


Figure 4.8: Function Block FTP Download

4.3.3 Database

The Database Server is a dedicated server running in a different machine from the Application Server which contains both the Node and the FTP. This server is responsible for the storage, in a relational database, of all the data related to the FBs. A schema for the database was designed according to the specifications of the metadata of the FBs. The schema of a FB is standardized in a DTD file that can be found in Appendix A.3 and accessed in [12]. Besides the structure found in the FB DTD, there was also the need to include a new entity, a category so the solution could have the functionality of separate FBs by categories defined also within the system.

A database schema was created to accommodate the data requirements and can be seen in appendix A.4. The schema contains the required tables and also triggers to assure the consistency of the data. According to what can be seen in figure 4.3, there is a constraint that says that event variables must be of the same type as the event as well as belong to the same function block. Therefore, the triggers created assure that if this constraint is not verified, an invalid flag is set in the event variable entity.

In contrast with the FTP server, the Database server is not flexible enough to be changed in the future. In fact, there is a driver installed in the Node application to connect to the MySQL database instance, the *MySql* node third-party package mentioned in section 3.4.1.1. In order to make this possible, it would have been necessary to build an adaptor in the Node source code so different types of DBMSs could be used. This was out of the scope of the project.

Nevertheless, it is possible to change the location and port of the running Database server as these are configuration parameters of the Node application as it is explained in section 4.3.1.5.

In fact, in order to make that configuration work, the configuration of the MySQL database instance needs also to be configured with the follow parameters that should be pass on to the Node application configuration:

- Database Name

- Database User
  - Username
  - Password

It is also necessary to give permissions to the created user so it has rights to the database.

4.3.4 Web Application

This is the most visible part of the whole project. As previously mentioned in section 3.4.3, React is a JavaScript library developed by Facebook that allows the building of Web Applications by switching from the imperative style of programming to a declarative one. This way, the application becomes much more modular. It is even possible to reuse some components so they can act as a sort of template. The Web Application structure has been divided in the following categories, each category is a folder containing other sub-folders or the actual implementation file in React.

- **Components**: This category contains the actual UI components of the application, these components were subdivided in Function Block folder, with all the UI elements to manage the Function Blocks, the Smart Components folder, to monitor the Smart Components connected to the network, the Function Block Categories folder, containing the UI elements to allow the users to manage the Function Block Categories and the templates folder, containing the UI elements meant to be reused in the other components like charts, navigators and tables.

- **Services**: This category of files was created to accommodate the services needed to communicate with external system components, in this case, with the Jurassic Park backend. Inside this category, there are two sub-categories, the HTTP module, with all the functions to fetch and send data to the backend, and the Socket module, with functions to communicate via TCP sockets with the backend, using Socket IO external package like mentioned in section 3.4.3.

- **State**: The paradigm of declarative programming and React in particular is very simple and based on a UI change in a reaction to an internal change in the component state. This state can be a simple variable like a string, a number or an array or even a set of these. This state, once changed, triggers an update in all the UI elements that are dependent on that state. However, for complex applications with more than one component, there should be a place to centralize all the state variables used by different components. Therefore, the functions built inside this module, are able to extract or change the state, so any component that needs to change this centralized state can only do that by using this API. As an example, the Function Block categories fetched from Jurassic Park Backend should be centralized here so the Function Block components and the Function Block Categories components can make use of them to build their respective UI.

When the application is first rendered, it is possible to check if the backend is up and running by checking the icon on the right side of the application top bar, as it can be seen in figure 4.9. The green light indicates that a connection could be established, red color indicates otherwise. This functionality is achieved by establishing a permanent Web Socket connection to the backend. As mentioned in section 3.4.3, the client-side of the Socket IO library was used in order to establish the connection to the server-side part of this library, running on the backend.

(a) Backend Up and Running (b) Backend Down

Figure 4.9: Backend State

It is important to state that each of the application components described next will run on a different page of the Web Application. This can be seen in the URL address bar in the browser. One could think that, for this reason, this is not a Single Page Application. However, it still is. The address bar URL is not a backend endpoint, is just a way the frontend has to organize the navigation, so when that URL changes in the browser, there is no communication with the backend, it is just a change in the component that is being rendered at the moment. All the communications with the backend are done asynchronously and the data received is always plain *JSON* data. The frontend is decoupled from the backend and this will never send HTML content to the client. Instead, it is the client browser the one responsible to render all the UI content.

The main pages of the application are the following ones, with the respective URL address.

```
{
  home: '/',
  functionBlockList: '/function-block',
  editFunctionBlock: '/function-block/:id',
  newFunctionBlock: '/new-function-block',
  smartComponentList: '/smart-component',
  smartComponentDetail: '/smart-component/:id',
  functionBlockCategories: '/function-block-category',
}
```

Listing 4.1: Web Application Main Pages

4.3.4.1 Function Block Components

In this area of the application, all the functionalities regarding managing the FB repository can be performed. The main ones are the CRUD operations. The communication between this area of the application and the Jurassic Park Backend is established by means of HTTP requests. The endpoints consumed are the ones described in section 4.3.1.1. The invocation of these endpoints

is made with high-level functions developed in the HTTP service interface, described in section 4.3.4.5.

The first one is the list of all the FBs available in the marketplace and can be seen in figure 4.10, available through the path */function-block*.

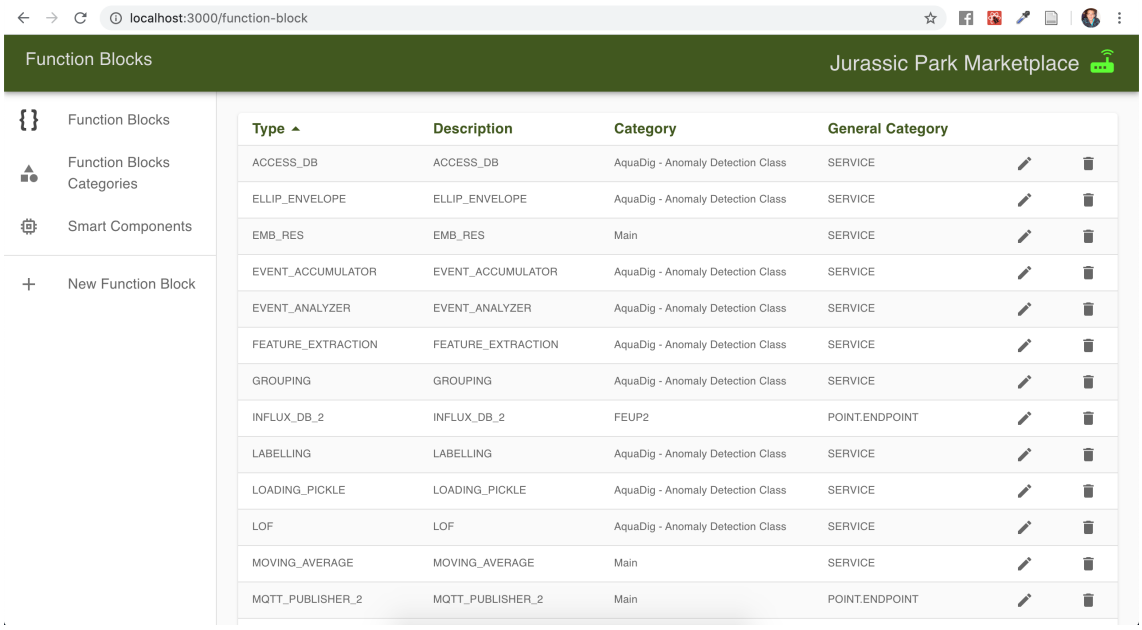


Figure 4.10: Function Block List

In this list, it is possible to see all the main characteristics of each FB, including its type, description, category, and general category. This list makes use of the Jurassic Park Backend endpoint '*GET /function-block*', fetching all the FBs present in the database. Furthermore, after fetching the data, it will save it locally, so no further fetching is needed, neither in this component nor in any other that uses this data.

The second functionality of this area is the creation of a new FB. The following form, depicted in figure 4.11 and rendered in a separate page, in the path */new-function-block*, contains all the fields required to create a FB.

This form contains all the fields for creating a new FB. It is important to refer to the model described in section 4.2. An example of this validation can be seen in figure 4.12.

With this validation, the application assures the correct creation of the FB, making sure that the structure described in the models is followed. It is also worth pointing out that the implementation file in Python is also a mandatory field. In fact, this is one of the most important parts of creating the FB and it is where all the logic of the software module is implemented to be run in the Smart Component RE. Furthermore, there can't be more than one FB with the same name, and any attempt to create a FB with the name of any other existing one will result in an error returned from the backend, and shown in the application, as can be seen in figure 4.13.

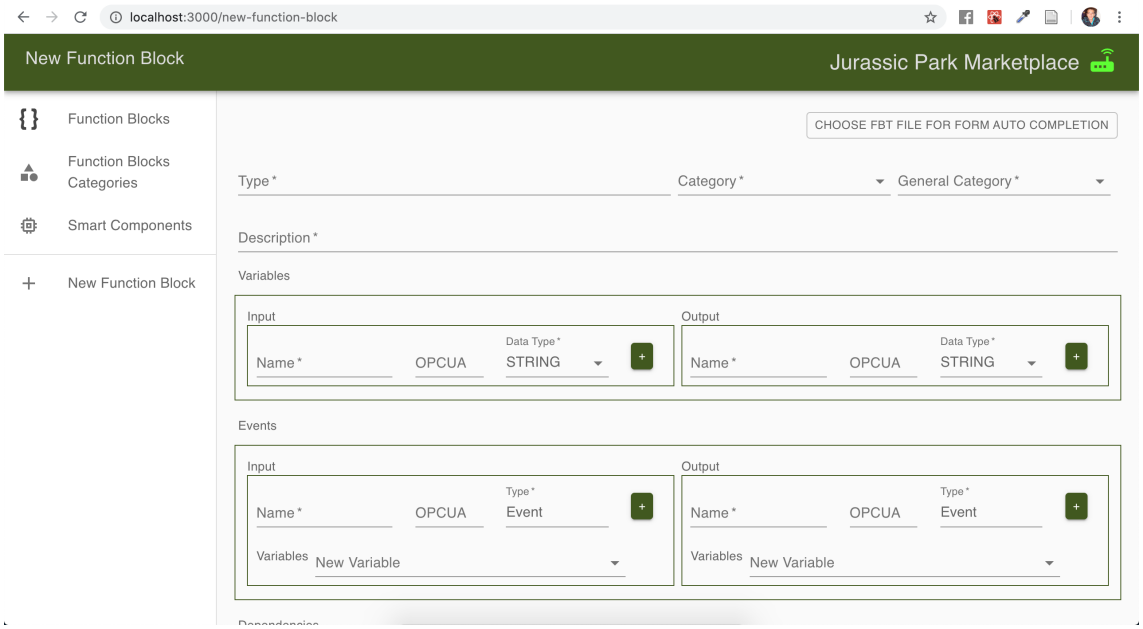


Figure 4.11: New Function Block Form

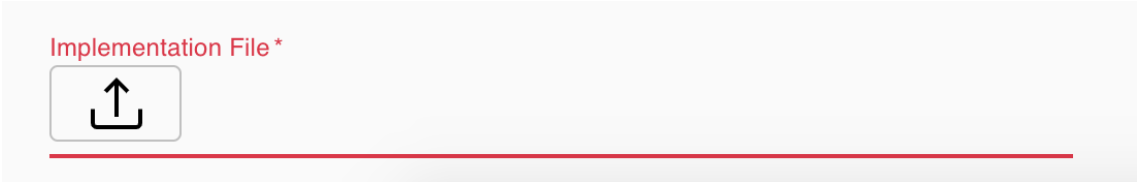


Figure 4.12: Function Block Validation Example

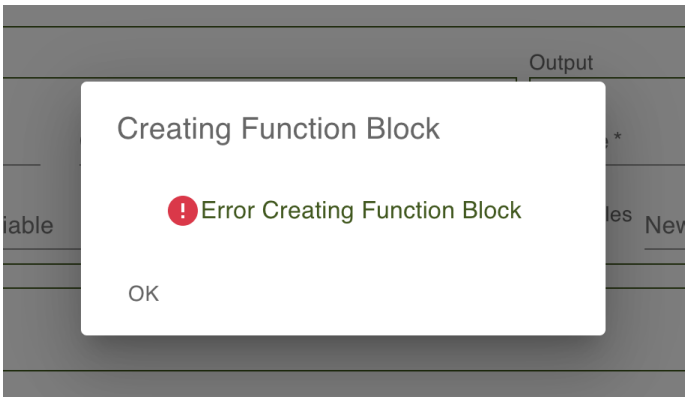


Figure 4.13: Function Block Error

After the correct filling of all the required fields, the Web Application, before sending the data to the Jurassic Park Backend, creates the XML file of the FB metadata, following the specification found in appendix A.3. After that, it will hit the API endpoint '*POST /function-block*', already described in 4.3.1.1. After receiving a successful response from the backend, the FB will then be saved to the internal state of the application so it is available to the rest of the components without having to fetch all data from the server again.

There are two ways of using this component to create a new FB. The first one was just described in the previous paragraph, but there is a more convenient way to create a FB for those cases in which the FB is already created as an XML metadata file. In that case, it is possible to drag that file and drop it into the form. The application will parse the XML file and automatically fill all the fields of the form. It is also possible to drag and drop the implementation file along with the XML file. This way, it is even faster to create a new FB in these situations.

The third functionality of the Function Block area is the edition of a FB. According to what has been said about React, this is a library that allows the creation of modular components which means that when building applications with this library, it is possible to reuse some of the components. That is what happened when this functionality was developed. The FB form had already been built, so it was just a matter of reusing it. The difference is the way it is loaded when the form is loaded, the data of an already existing FB is passed on. This component is available through the path */function-block/:id* where the URL parameter corresponding to the id of the FB is extracted in order to get the FB data. When this page is loaded, a first search in the state is done in order to check if the FB is already loaded locally. Otherwise, a call to the backend API *GET /function-block* is made, fetching and updating the local state with all the FBs available. An example of an already filled FB form can be found in figure 4.14.

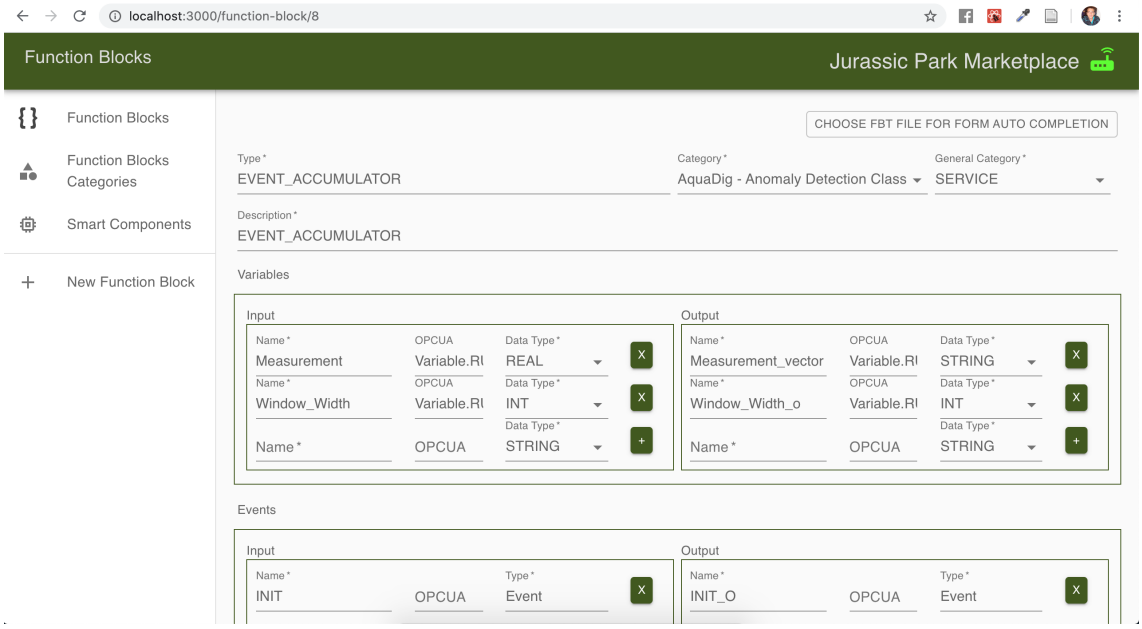
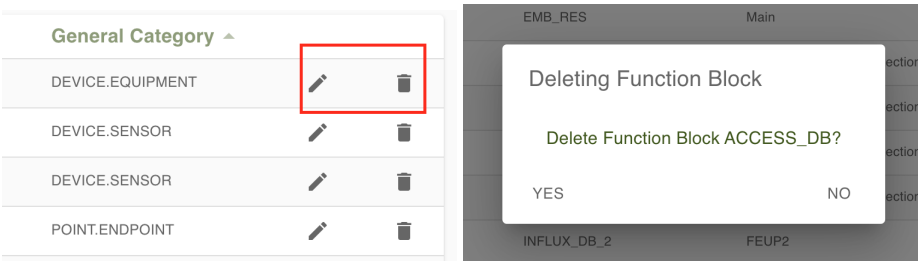


Figure 4.14: Function Block Editing

With this approach, it was possible to avoid the redundancy of the code, reusing the component already built before for creating a new FB. In terms of mandatory fields, the only one that is not mandatory while editing an already existing FB is the Python implementation file as there is already one uploaded when the FB was first created.

After modifying the FB, the new data is transferred to the backend by calling the endpoint '*PUT /function-block/:id*' where the URL parameter id is the id of the FB. After receiving a successful response from the server, the local state is updated with the updated FB.

The last operation a user can perform in this area is the deletion of a FB. This functionality is available from the list, as there was no need to create a separate page for it. From the list of the FBs, there are two buttons in each of the FB. The first one is a link to the edition page just described in the previous paragraph. The second one is the button to delete the FB, where after the click, the application prompts the user for confirmation. This can be seen in figure 4.15.



(a) Action Buttons (b) Deletion Confirmation.

Figure 4.15: Function Block Deleting

After confirming the deletion, the endpoint '*DELETE /function-block/:id*', updating the local state upon a successful response from the API.

With these four functionalities, all the CRUD operations can be done, and the marketplace managers are capable of managing the FBs database without having to manually create metadata files.

4.3.4.2 Function Block Categories Components

In this area of the application, it is possible to manage the FBs categories. These categories are not related to the IEC-61499 specification. The concept was designed in order to better organize the software modules in the repository. With this, related FBs can be grouped by category. As examples, these categories can represent projects for which the FBs were developed, algorithm types that the FB implements among others.

Unlike the Function Block area of the application, all functionalities of this area are available in a single page, through the path */function-block-category*. This happens because there is no need to have a complex form to create a category, as the only needed field to create a category is the name of the category.

The first functionality of this area is the list of available categories. For each category created, the application shows the name of the category and a list of FBs in that category. The categories

managed here are available in the FB form, when creating or editing. This can be seen in figure 4.16 and figure 4.17.

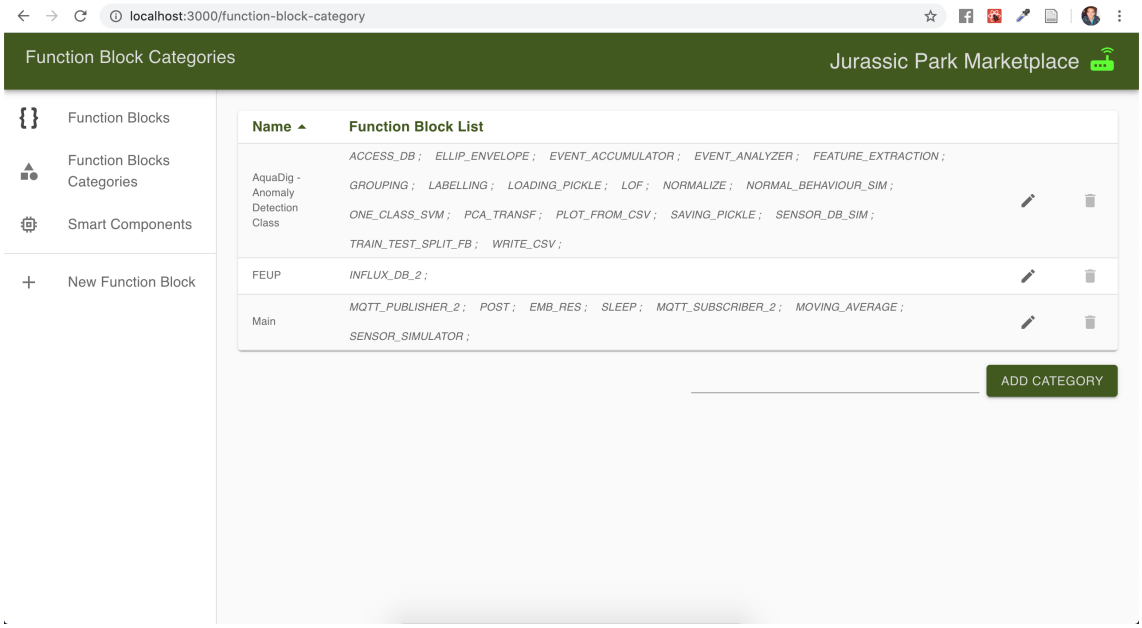


Figure 4.16: Function Block Categories List

The data presented in this list is an array of categories. This entity is described in section 4.3.1.4. When loading this component, the application will first check the local state for the list of categories. If they were previously fetched, the list is built right away. On the contrary, data will be fetched from the API endpoint '*GET /function-block-category*'. After fetching the data, the component will be rendered and the data saved in the local state, avoiding further communications with the backend for this purpose.

To create a new FB category, the second functionality in this area, there is a simple field and button for confirmation. As mentioned before in the section, the category entity is very simple in terms of data required by the user. These two UI elements are placed under the list of categories, and to create a new FB category there is no need to navigate away to another page, as it is visible in figure 4.16.

In order to create the FB category, the Web Application will send an HTTP request to the backend endpoint '*POST /function-block-category*' and after the correct answer, the category will be saved locally, being then available for other components. The list is also updated with the newly created category. Like with the FB entity, there can't be more several categories with the same name and if that is the case, the application will alert the user, after receiving an error from the backend.

The edition of a category is also possible. As depicted in figure 4.16, there is a button, in each category of the list that will prompt a dialog box where the user can edit the category. This dialog box can be seen in figure 4.18.

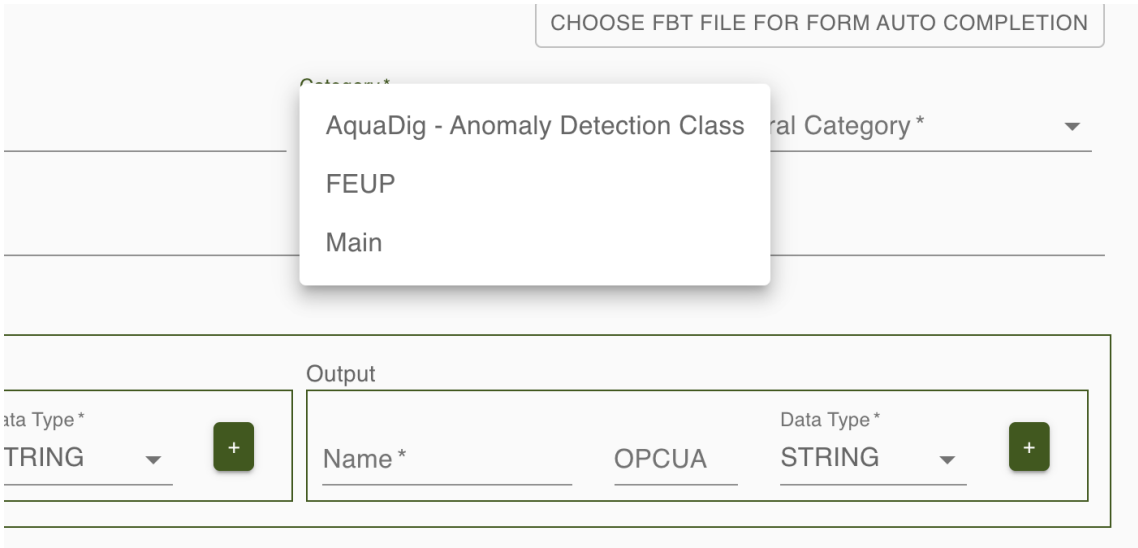


Figure 4.17: Function Block Categories List in Function Block Form

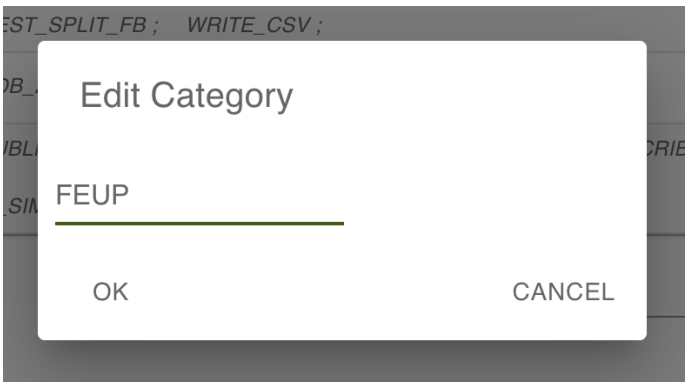


Figure 4.18: Function Block Category Edition

This action will call the backend endpoint '*PUT /function-block-category/:id*', where the id URL parameter identifies the category to be edited. The edition will be saved locally upon a good response from the backend.

As when creating a new category, the edition of a category can return an error in case of changing the category name to one that already exists.

The last functionality is the category deletion. Like in the category edition, there is a button, in each of the categories lines, that triggers that category deletion. It is important to note that only a category without any FB associated can be deleted. In terms of Web Application, this button will be disabled for those categories that can't be deleted, as can be seen in figure 4.19. Regarding the backend, there is also a constraint in the database that avoids the deletion of a FB category in case there are FBs in that category.

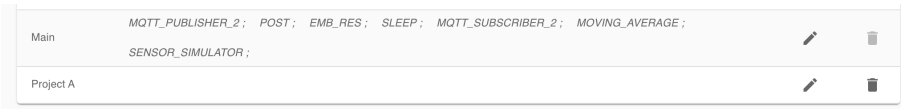


Figure 4.19: Function Block Category Deletion

This action will make the application send an HTTP request to the backend endpoint '*DELETE /function-block-category/:id*', identifying the category to be deleted with the URL parameter. After the correct deletion, the list will be updated and the category deleted from the local state.

4.3.4.3 Smart Devices Components

This area of the application is concerned with the monitoring of the Smart Components. In contrast with the Function Block area, the communication between the application and the backend is established via a permanent TCP socket connection. To avoid the complexity of the low-level specification of the Web Sockets API, the client-side of the Socket IO JavaScript library was used in order to establish that communication. This library allows the creation of namespaces which are equivalent to the API endpoints. With this approach, is possible to separate concerns within the application, separating the communication channels. Therefore, the namespaces used to establish the connections are the ones detailed in section 4.3.1.2. All the communication is made with the high-level functions developed in the socket service interface described in section 4.3.4.5.

The first page of this area is the list of the Smart Components connected to the network. This UI component can be accessed through the path */smart-component* and establishes a connection to the backend through the socket '*/smart-component*' namespace. This list shows the information of each of the devices that have already established a connection to the marketplace. Hence, a Smart Component can be either connected or disconnected, as the green or red light indicates, and as it is possible to see in figure 4.20. The list also shows, for each component, its name, address, port, CPU percentage current usage, and memory used. As this component is permanently connected to the backend, this information is being updated in real-time so when a new device is connected

or when a running device is shut down, the information is updated accordingly without the need to refresh the page.

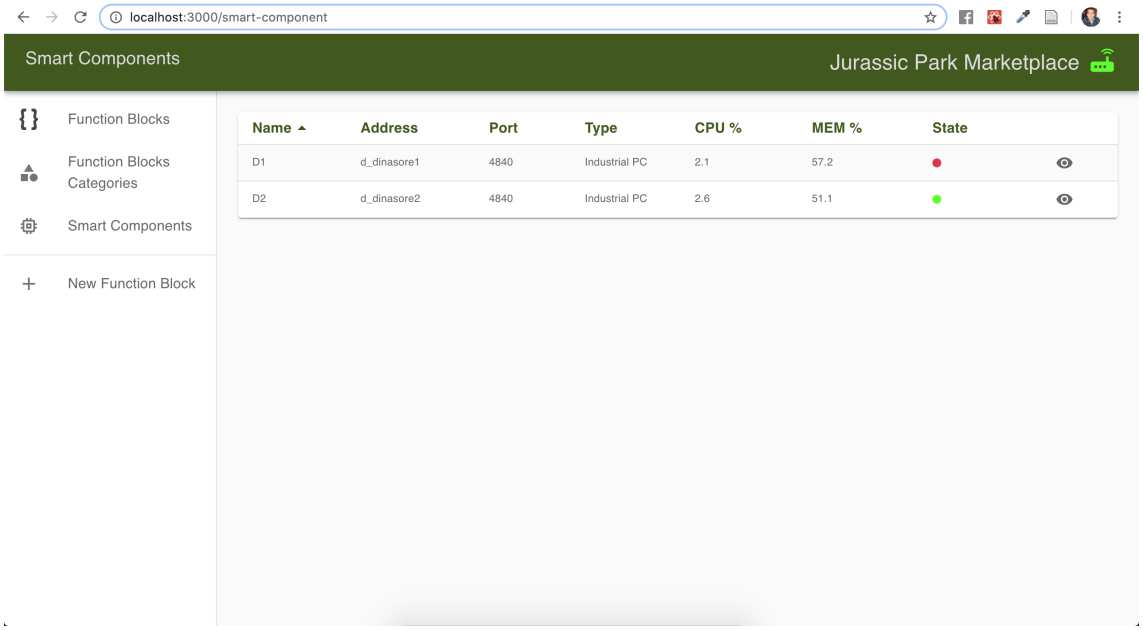


Figure 4.20: List of Smart Components

The second functionality of this area, accessed through the path */smart-component/:id*, is a page with the detailed information of the Smart Component. The id in the URL parameters identifies the Smart Component to be rendered. Like in the previous page, the load of this page triggers a socket connection between the browser and the backend, to the '*/smart-component:id*' namespace. The data being received in real-time can be seen in figure 4.21.

One of the main objectives of the project was the monitoring of the FB state, running on the devices. In this page, when some FB instance is not working properly, it is possible to see what is that FB instance and in which Smart Component is running. This is achieved by checking, in the list of running FB of the Smart Component, the state of the FB instance. The green icon means the FB is running correctly and when it goes to red, it stopped working. Like the previous component, this page is being dynamically updated, making this page a live dashboard to monitor the Smart Component.

4.3.4.4 Other Components

During the development of the Web Application, there was a need to develop some components to be used across the application. Doing this, it was possible to save several lines of code in the application, just by isolating these components and use them inside others, passing the needed values to the correct rendering. The list of these components, which were called templates, is described next.

- **Bars**: To draw a bars chart, passing the minimum, the maximum, the data and the colors.

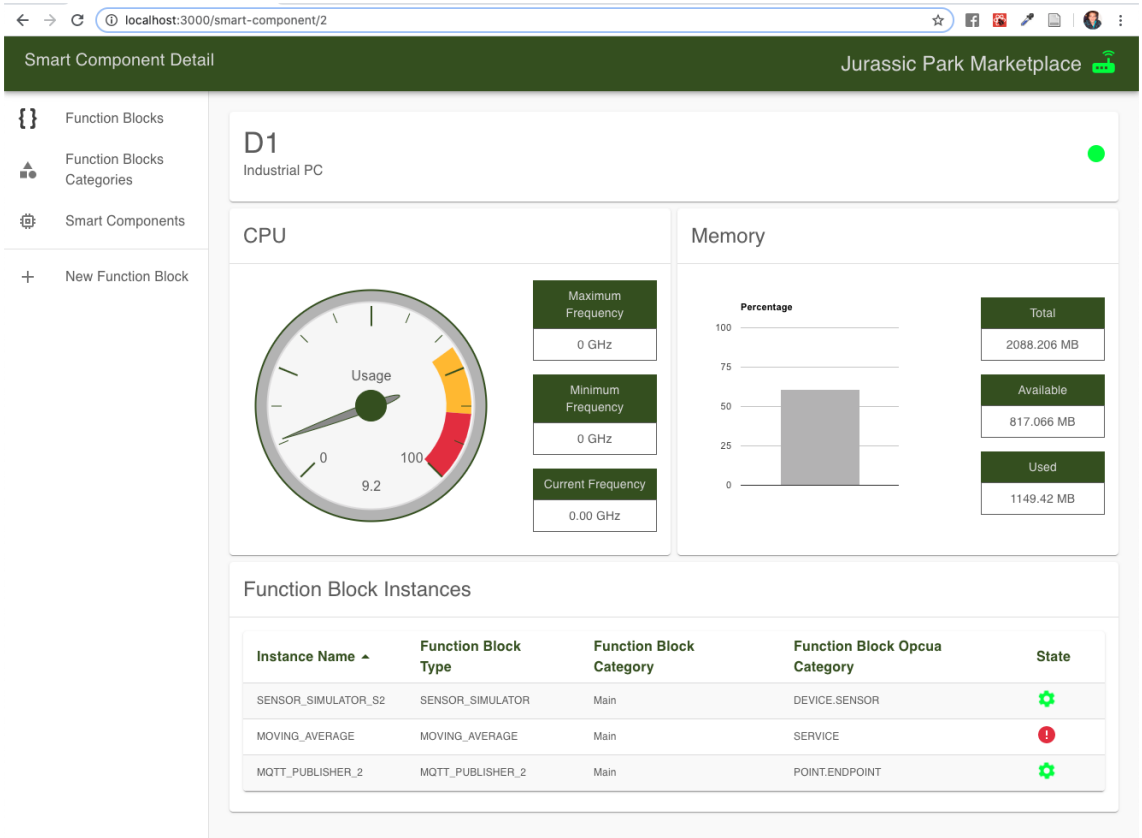


Figure 4.21: Smart Component Detail

- **ConfirmAction**: A component to manage asynchronous calls and confirmation actions. This component is a state machine with a pre-state where prompts the user to confirm the action, the executing state that makes the asynchronous action, usually a call to the backend API and a success state, and an error state as the final state. This component is rendered in a dialog box.

- **LazyComponent**: This is a component wrapper for the components that need data loaded asynchronously. The component wrapped inside this will just render when the data is received. Meanwhile, a circular progress bar is shown.

- **Navigator**: This the component that wraps the left navigation bar and a main component. It is used to navigate between pages of the application.

4.3.4.5 Services

The organization of the Web Application was made separating the concerns. As mentioned before in this section, the three main modules of the application are the UI components, the services, and the Local State.

This section of the code contains high-level entities to cope with communication to external services. In the context of the project, the only external entity to communicate is the Jurassic Park Backend. Nevertheless, the backend provides two different ways of communication, an HTTP API and a Socket engine, to establish real-time communication channels.

Therefore, this section of the code is divided into two subsections, the HTTP service interface, and the socket service interface. Both are briefly described next.

- **HTTP Service Interface**
  This service uses the fetch browser API in order to send HTTP requests to the backend. For the sake of standardization of all backend responses via the HTTP API, the following interfaces were created.

```
interface RequestResponseState {
    error: boolean
    msg: string
    errorCode: number
    extra: any
}

interface RequestResponse {
    state:RequestResponseState
    result: any
}
```

Listing 4.2: Request - Response Structure

Thus, all the requests made to the backend API are passed back to the service caller as a RequestResponse that contains the state and the data returned by the backend.

Furthermore, the service provides an even higher level function for the specific needs of the application. The service provides high-level functions for the following actions.

- Get FBs
- Create a FB
- Edit a FB
- Delete a FB
- Get FB Categories
- Create a FB Category
- Edit a FB Category
- Delete a FB Category

The result type depends on the entities being fetched for the get requests. In regards to other types of requests, the state of the response will determine the success or error of the request.

- **Socket Service Interface**
  This service is a wrapper class of the Socket class provided by the Socket IO library. An instance of this class is built by providing the namespace where the connection is to be established. The connection is made calling the main function of the class and initially passing the callback functions for the main socket events.

  - *Connect*: Callback executed when the connection is established
  - *Disconnect*: Callback executed when the connection is terminated
  - *Connect_Error*: Callback executed when there is an error when establishing the connection
  - *Error*: Callback executed when there is an error in the connection
  - *Connect_Timeout*: Callback executed when there is a timeout when establishing the connection
  - *Reconnect*: Callback executed when the connection is reestablished
  - *Reconnecting*: Callback executed when the connection is being reestablished
  - *On_Initial _Data*: Callback executed when the connection is established and the backend sends the initial data

  Among these callbacks only the connect and disconnect callbacks are mandatory to pass on to the service when connecting to the socket. Furthermore, the service also provides a function to install a listener for a specific event, as seen above.

```
1     public addListener(event: SOCKET_EVENT, listener:(data:any) => void) {
2        this.socket?.on(event,listener)
3    }
```

Listing 4.3: Adding a listener to a socket event

This way, the user of the service can be notified when an event was received and retrieve the data transmitted in that event.

The Smart Components area of the application makes use of these services in order to update their respective UI whenever new data is received.

4.3.5 Integrations

The last subsection of this chapter describes the integrations needed to be developed in order to bring all the subsystems together and accomplish the goals of the project. Before the development of this project, the set of the Dinasore RE plus the 4Diac IDE was already a running system, being effectively used to build process control systems for the manufacturing industry. In fact, it was the usage of the system that brought on other kinds of necessities which ultimately lead to the development of this project.

Thus, the last task developed in the project was to build a way of interconnecting the existing system with the newly developed subsystem, comprising the Jurassic Park backend which contains the servers and the database and the Web Application. This task is usually called System Integration.

It became clear since the beginning that two integrations were necessary as the running system is composed of two subsystems, the RE, installed in the Smart Components to actually run the FB deployed by the second subsystems, the IDE, in this case, the 4Diac IDE.

4.3.5.1 4Diac IDE Integration

In terms of the 4Diac IDE, the approach was very simple and the integration just consisted of a shared File System between the Jurassic Park backend and the 4Diac IDE. In other words, the backend part of the new system needs a way to access the File System where the 4Diac IDE is running. For the sake of clarity, the diagram of figure 4.22 depicts this scenario.

The 4Diac IDE has a local database of FBs meant to be used to program a process control system. This local database is a set of files with the meta-information of the FBs. The same files that the backend now is able to produce when a FB is created. For this reason, and also because the 4Diac is developed by a third-party, outside the context of the laboratory where the project was developed, this was the easiest and more straightforward way to integrate with the IDE. Furthermore, if in the future, there is a need to change the IDE, the integration is done in the same way, avoiding specific integrations for a particular IDE.

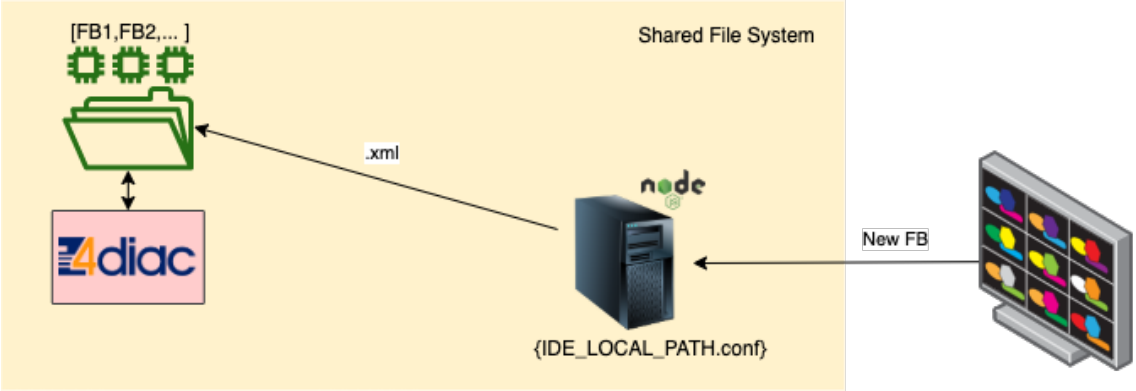


Figure 4.22: 4Diac IDE Integration

As a result, and as also mentioned in section 4.3.1.5, the configuration of the backend as a parameter defining the path to the folder where the IDE is storing the FBs meta-information files. Thus, when a FB is created, a copy of the metadata file in XML is written in that specific folder, being available to the IDE. When a FB is modified or deleted, that change also takes effect in the IDE local database.

The easiest way to achieve this is to make sure that the IDE and the backend server are running in the same machine, making the integration easy and straightforward. Nevertheless, if this is not the case or in the future, there is a need to change the infrastructure, there are network protocols to create network file systems like Network File System, Samba, and Common Internet File System, among others.

4.3.5.2 Dinasore Integration

The RE environment is an application developed in Python and installed in all the Smart Components. This application allows each Smart Component to run the configuration received from the IDE, or in other words, which software modules are supposed to run and how. For that, each of the Smart Component has a TCP server running so whenever the IDE needs to deploy a new configuration of the system, it establishes a TCP connection per each Smart Component and deploys its configuration, telling what are the FBs that need to run and how they are linked with each other. After that, the TCP connection is then closed. The set of all the Smart Components and running FBs forms the whole process control system. Previously to this project, these FBs needed to be already installed, otherwise, the deployment of the system in the Smart Components would return an error to the IDE, stating that the Smart Component didn't know about a specific FB. According to this, the integration needed to be made was to avoid this error. Once the main server and the repository of these modules were complete, it was time to put the RE talking to the repository.

The idea was to make sure that, for every new configuration deployment, the Smart Components could retrieve the meta-information of the FBs they will run as well as their implementation,

so no previous installations were needed to be made. As a further requirement, the Jurassic Park platform should be also able to monitor the Smart Components.

The Dinasore RE is structured in four modules:

- **Communication**: This module contains the classes to take care of the communication between the RE and the IDE.
- **Core**: This module comprises the fundamental part of the application, the classes responsible to start the FBs execution.
- **Data Model**: This module is related to the data structure of the FBs and how they are related together as a whole in the RE.
- **OPC-UA**: This module contains all the classes to create an OPC-UA server that handles OPC-UA client connections in order to exchange information about the Smart Component to monitor its state.

As described in section 4.3.1.1, the backend exposes an HTTP REST API. A part of that API was developed to be consumed by the RE, as mentioned in that section. Besides using this HTTP API, the RE needs also to download, via FTP protocol, the FB implementation files.

The integration was made by creating a new class, inside the communication module, to take care of the communication between the RE and the Jurassic Park. This class was developed with all the methods to call the endpoints necessary to the integration and to download, from the FTP server, the implementation files.

This new communication module class developed has three main responsibilities, detailed next.

- **Announcement**: The first responsibility of the integration class is to announce itself to the Jurassic Park marketplace. This announcement is made by calling the HTTP endpoint '*/POST smart-component*'. The payload of the request contains the address of the Smart Component and the port where the OPC-UA server is running. This announcement is made every time the Smart Component is started or restarted. With this, the Jurassic Park backend adds the device to the list of currently monitored devices and tries to establish a connection to the OPC-UA server to monitor the Smart Component.
- **Retrieve FB meta-information**: The second functionality of the integration is related to the download of the metadata of the FB. For each FB deployed by the IDE, the Dinasore will retrieve the FB metadata calling the endpoint '*GET /function-block/:fbType*'. The meta-information is a plain *JSON* text, converted to a Python dictionary, following the model described in section 4.3.1.4. This model contains, among other data, two essential fields:
  - FB category: the category of the FB is used to build the path where the FTP download will fetch the FB implementation file.

  - FB External Dependencies: When creating the FB through the application, it is possible and desirable to add all the Python external packages needed to be installed in the device in order to properly run the FB. After retrieving this list, the RE will check if the dependencies are installed and will install the missing ones from the python repository.

- **FB download**: The last part of this integration class contains methods to download, via FTP, of each of the FB files, the implementation, and the metadata file. The path to files will follow the pattern '*/home/[FB_Category]/[FB_Type]/[FB_Type].py*' for the implementation file and '*/home/[FB_Category]/[FB_Type]/[FB_Type].fbt*' for the XML file containing the metadata of the FB.

These integration points are well visible in the overall architecture depicted in figure 3.1.

In terms of configuration, there is a need to pass some new parameters to the RE when it starts to run. Before this project, there were already some variables passed through the command line when starting up the RE. After this integration, the following ones were added.

- *mha*: the Jurassic Park marketplace HTTP server address
- *mhp*: the Jurassic Park marketplace HTTP server port
- *mfa*: the Jurassic Park marketplace FTP server address
- *mfp*: the Jurassic Park marketplace FTP server port

To better understand the several scenarios in which the RE contacts the Jurassic Park marketplace, the section 5.1 contains the sequence diagrams that explain these scenarios.

Chapter 5

Experiments and Results

In this chapter, the testing and experimentation of the solution are described. In the first section, the most used functionalities of the system are documented in sequence diagrams, explaining the interactions among the several parts of the whole system. In the second section, the results of several tests are described, starting from a simple proof of concept and then increasing the complexity of the scenarios.

5.1 Main Functionalities

5.1.1 Creating a Function Block

To create a new IEC-61499 compliant Function Block, the user will fill in the required fields. These fields are described in 4.3.1.4 and the specification is detailed in a DTD file which can be consulted in A.3. This can be done filling the manual form that can be found in the application or automatically, dragging and dropping an XML file into the form. In both cases, the Web Application will perform a validation on the data, making sure that the FB is IEC-61499 compliant and has all the required fields, as well as the information is consistent. After the validation, the data and the FB implementation file are sent to the Jurassic Park backend, calling a specific endpoint that id described in section 4.3.1.1. The Jurassic Park backend performs also a validation on the data. In case of a successful validation, the FB metadata is saved in the database. After that, a metadata file is generated, and together with the implementation file, it is saved in the FTP serving folder. A copy of the metadata file is also saved in the 4Diac library folder.

The process ends with a response message from the Jurassic Park backend to the application, alerting the user of the correct creation and saving of the FB. In case of an unsuccessful validation, an error is returned and none of the above backend actions is performed.

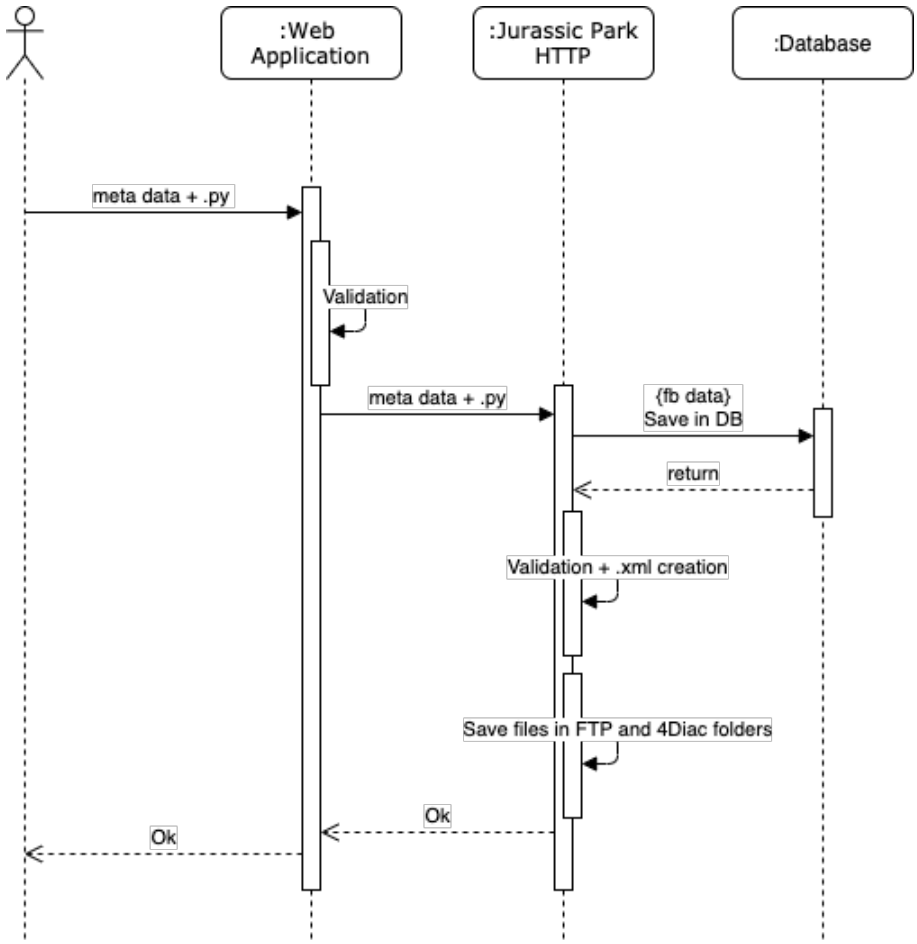


Figure 5.1: Function Block Creation Sequence

5.1.2 Downloading Function Blocks

This is the most important goal of the whole solution. One of the major drawbacks of the IEC-61499 is the fact that there is little or no concern about the implementation and storage of the FBs. The standard is more concerned about the system as a whole and how the different pieces of software are linked together in order to design a functioning process control system. This turns the compliance to this standard sometimes a bit cumbersome because there is a whole previous work of getting all the software ready to run on the devices or even a tedious process of debugging and deploying the software among all the components.

Therefore, before this solution came up, whether there were two or two hundred Smart Components comprising a system, the programmer usually transferred manually or via SSH the needed modules to every Smart Component. After that, if there was something missing or a bug found out in a FB, the process of installing the FB had to start over again.

With this solution, the idea was to turn the Smart Components more autonomous and independents from human intervention. Thus, with the Jurassic Park marketplace and with the integration made on the Dinasore RE now the RE is capable of checking the needed software resources and download them automatically whenever it receives a new configuration from the IDE, or in other words, whenever it receives a new set of required FBs. The whole sequence is depicted in the diagram in figure 5.2.

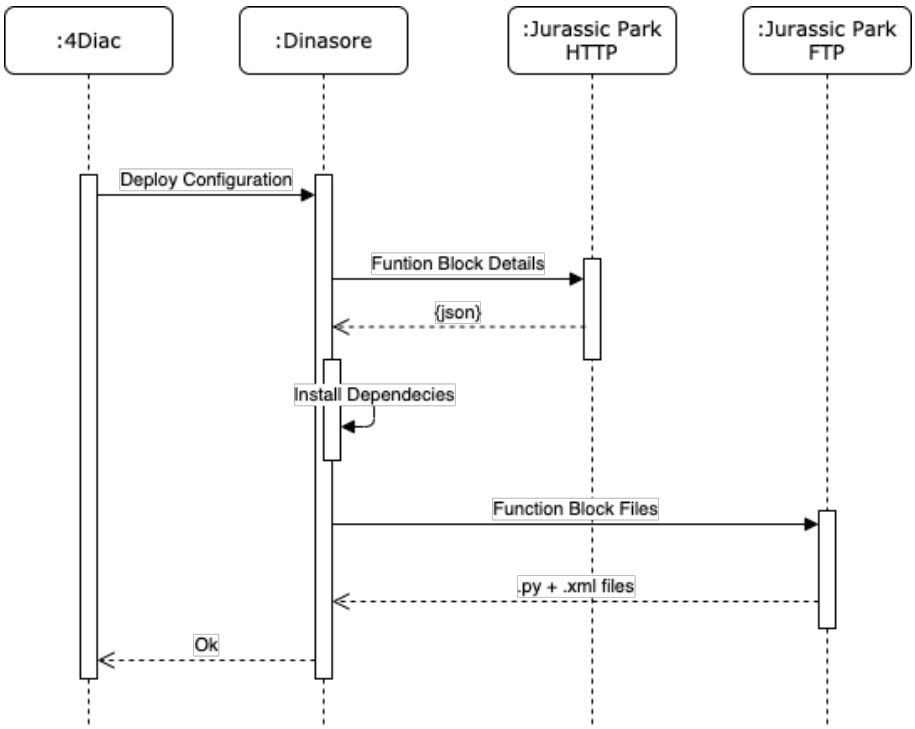


Figure 5.2: Function Block Download Sequence

The process starts when the IEC-61499 programmer uses the 4Diac to deploy a configuration, this will make the IDE send the list of FBs to be used by each of the Smart Components. The

Smart Component running the Dinasore RE will then contact the Jurassic Park backend to get the details of each of those FBs. After receiving these details, it will check the dependencies for every FBs to download, if there are missing dependencies, the RE will download and install them via PIP. After that, the RE will then download the FB via the Jurassic Park FTP server. The sequence is closed with the response from the Dinasore RE to the 4Diac IDE.

5.1.3 Starting up a Smart Component

The developed solution does not require that all the Smart Components have to be turned on and running in order to communicate with the marketplace. This means that whenever a new device has to be added to the system, the Jurassic Park can cope with that situation. For that, the Smart Component will announce itself to the marketplace, informing it about the address and the port where the OPC-UA server is running. The Jurassic Park will then try to connect to the OPC-UA server running in the Dinasore RE and send back the response, acknowledging that the connection was established correctly. From that moment, the marketplace can then start to monitor the Smart Component, both in terms of the hardware resources consumption and in terms of software errors on the running FB instances.

This sequence is well detailed in the diagram of figure 5.3. It should also be noted that the sequence depicted in the diagram of figure 5.2 and the sequence explained in this paragraph can be performed together in some situations, as explained next.

- *Fresh start of the device*: In this case, if the Smart Component has no configuration saved before, so it will just announce itself to the marketplace to be monitored.

- *Restart of the device*: When restarting a device that had already been used before, a configuration is saved on the device, with the information of the FB needed to be installed there. Therefore, both sequences will be performed, which means that besides the announcement, the Smart Component will also download the required FBs.

- *New configuration deployment*: When there is a new configuration deployment from the 4Diac IDE, the Dinasore RE will resend an announcement to the marketplace so the marketplace can reconnect to the OPC-UA server and refresh all the resources being used. Besides this, the FB will be downloaded again because of the deployment of the new configuration.

This approach gives a lot of flexibility to the IEC-61499 programmer. On the one hand, when new resources are needed, there is no need to manually add the device information into the system. On the other hand, if the software needs of a device also change, the required software will also be downloaded automatically.

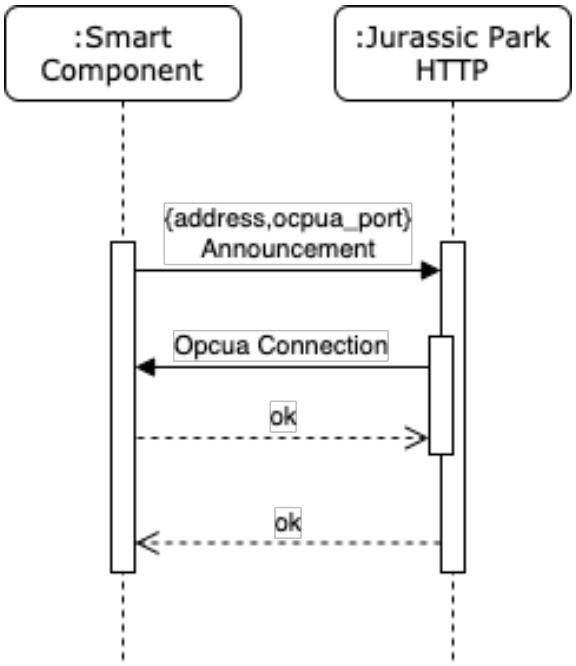


Figure 5.3: Starting Up Smart Component Sequence

5.2 Usability Validation

Prior to this platform, when a new FB had to be created, besides the actual programming of the implementation file, in Python, the programmer had also to manually create an XML file, compliant with the IEC-61499 specification. After that, both files had to be transferred to the devices where the FB was to be used.

With the Jurassic Park marketplace, the development of the IEC-61499 compliant XML file is no longer necessary and the FB management is done through the developed GUI. The usability validation was made with the implementation of a sensorization simulation scenario. This scenario simulates the acquisition and storage of data from a sensor and its moving average through time. The data is generated in a simulator FB in one Smart Component and saved in another Smart Component, on a local database, with the data being transferred via an MQTT broker.

The system designed for this validation is depicted in figure 5.4, a configuration designed in the 4Diac IDE. As it is possible to see, the system comprises two Smart Components, the FB instances marked in green deployed in one, and the FB instances in orange in a second device.

To accomplish this scenario, several FBs were developed. The list is described next.

- SENSOR SIMULATOR: This FB is responsible to generate a dummy value and send it as the only output variable.
- MOVING AVERAGE: This FB receives an input value, and together with a value defined for the window, calculates a moving average of the last values and sends it as the output variable.

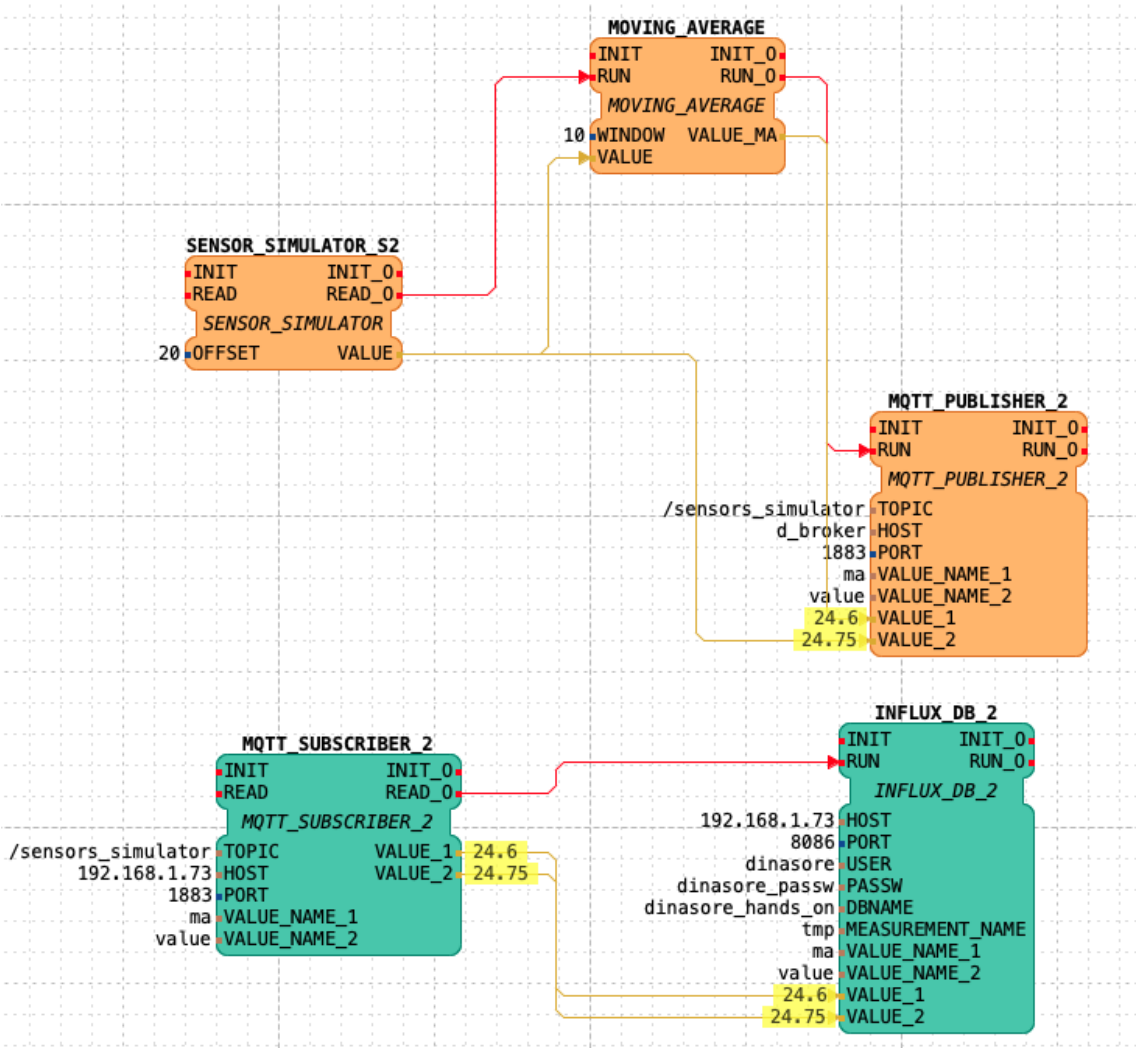


Figure 5.4: Usability Validation Scenario

- MQTT PUBLISHER: The MQTT Publisher FB is responsible to publish variables to an MQTT broker. In this specific case, the FB was developed to publish two variables and in the current configuration sends the current value simulated by the Sensor Simulator FB and the Moving Average calculated in the Moving Average FB.
- MQTT SUBSCRIBER: This FB subscribes to the topic where the previous one publishes the values to acquire the data, this FB outputs the values captured to the next FB.
- INFLUX DB. This is the database writer FB, its mission is to receive the values as the input values and write them in a local database.

5.2.1 Results

The validation of this first experiment is embodied in tables 5.1 and 5.2. A comparison between the different steps involved in the whole process has been done and two lists were produced. The first list with the steps involved prior to the development of the Jurassic Park Marketplace and a second one highlighting the suppressed steps.

Table 5.1: System deployment steps before Jurassic Park Marketplace

1.	Develop FBs logic in Python
2.	Write the XML files compliant with the IEC-61499 DTD
3.	Access each of the Smart Components directly or via SSH
4.	Transfer both the Python and XML files to each of the devices for all the FBs needed
5.	Copy the XML file to the 4Diac library
6.	Design the system
7.	Deploy the system to the several Smart Components

As it was possible to confirm, the results of the first experiment were the expected. It was possible to remove four steps out of a seven-step process to deploy a system.

5.3 System Monitoring

The second experiment conducted aimed to validate the monitoring of the Smart Components and the response that the system provides the user whenever something goes outside of what is expected. The scenario used in this experiment is the simulation of a system where there are LEDs which turns on and off depending on the value of temperature and voltage read by two

Table 5.2: System deployment steps after Jurassic Park Marketplace

1.	Develop FB logic in Python	
2	~~Write the XML file compliant with the IEC 61499 DTD~~ ~~Access each of the Smart Components directly or via SSH~~ ~~Transfer both the Python and XML file to each of the devices for all the FBs needed~~ ~~Copy the XML file to the 4Diac library~~	Create meta-information and upload implementation files to the Jurassic Park marketplace
3.	Design the system	
4.	Deploy the system to the several Smart Components	

sensors. The system is depicted in figure 5.5. There are sensor simulator FBs for the temperature and voltage. These FBs, in a real system, would represent the digital twins of the sensors, in this scenario, are just simulators and receive the simulated values through an MQTT broker. The values of the sensors are then passed on to the next FBs. These FBs will output a value depending on the actual value of the sensor, -1 if it is below the range, 0 in the range, and 1 above the range. Afterward, the LED controller FBs will control the actual value of the led by comparing the value received from the previous FB with its internal value represented as a FB variable and turn the LED on, in equality case. In this case, as it is possible to see in the figure, the LED1 turns on when the temperature rises above the maximum value (1), the LED2 will turn on when the temperature drops below the minimum (-1) value and the LED3 turns on when the voltage is higher than the maximum value(1). After all, if this is was a real scenario, the LED FBs would publish the computed value through an MQTT broker and the physical LEDs would subscribe, turning on and off according to the transmitted data. For this scenario, all the FBs were deployed in only one Smart Device. Besides these three FBs, there is another FB responsible to provide the others with an MQTT client so the FBs can subscribe to topics and also publish values.

Therefore, the following FBs were created through the Jurassic Park Marketplace application:

- MQTT CONNECTOR: This FB is responsible to create and provide other FBs with an MQTT client so they can publish to the MQTT broker and to generate and also publish the dummy generated data for the sensors.

- SENSOR MQTT: This is the sensor simulator. This FB receives the data generated by the MQTT connector by subscribing to the topic where the MQTT connector publishes, extract the required data and pass it on to the next FB. This FB is the sensor digital twin, in a real scenario.
- CONTROL CHART: This FB is used to output a value according to a comparison of the value received by the sensor and a pre-defined range, -1 below the range, 0 inside the range, and 1 above the range. This value will then be passed on to the LED FB.
- LED MQTT: This FB receives the value from the CONTROL CHART FB and according to that value will turn on and off the led by comparing it with the control value, publishing the result to a topic on the MQTT broker so the actual led can subscribe and turn on or off. Both the CONTROL CHART and LED FB represent the controller of the led and would be its digital twin in a real scenario.

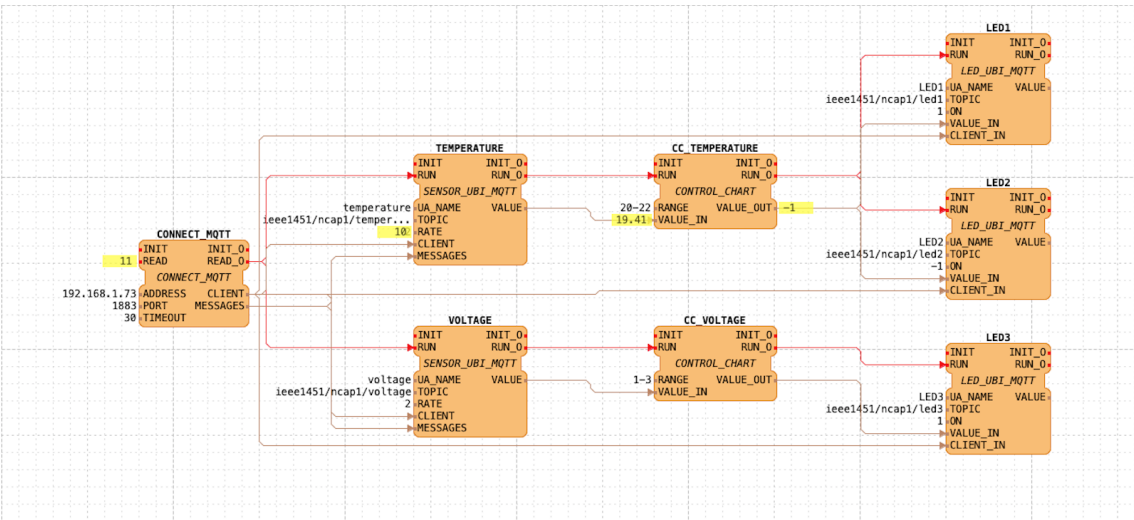


Figure 5.5: System Monitoring Scenario

This experiment aimed to validate the four main visual monitoring elements on the application:

- State of the Marketplace: As explained in chapter 4, the application as a visual icon indicates the status of the server. Whenever the marketplace is down, the icon will indicate that situation. This is accomplished through a web socket permanent connection between the marketplace and the browser. Whenever this socket connection is broken, the icon indicates that there is a problem connecting to the Jurassic Park marketplace server.
- State of a Smart Component: The state of a Smart Component running the Dinasore RE is also monitored and shown in the application in the list of Smart Devices. The monitoring component uses an OPC-UA connection between the marketplace and the Smart Component. Whenever this connection is broken, this information is sent from the marketplace server to the application running in the browser, through the web socket connection.

- State of a FB instance running on a Smart Component: All the FB instances have an OPC-UA monitoring value. This means that whenever a FB instance enters in an error state, this variable will indicate so. The criteria of a FB instance being in an error state is outside of this project scope. However, this method analyses the functional behavior of the FB using patterns generated by the FB events. An OPC-UA connection allows the monitoring of the OPC-UA variables of a device and with this, it is possible to monitor the variable which reflects the error state of the FB instance and communicate its state to the application, running in the browser.

- Resource Consumption of a Smart Component: As explained in chapter 4, the system also monitors the resources being used by the Smart Components.

5.3.1 Results

This scenario was chosen for this experiment because it is easy to manually induce errors in the FBs, the reason is outside of the scope of the project as this criterion is defined when developing the actual FB. The other monitoring elements were easily validated by just turning off a connected device and turning off the server itself. The results of the experiments are demonstrated in the next figures.

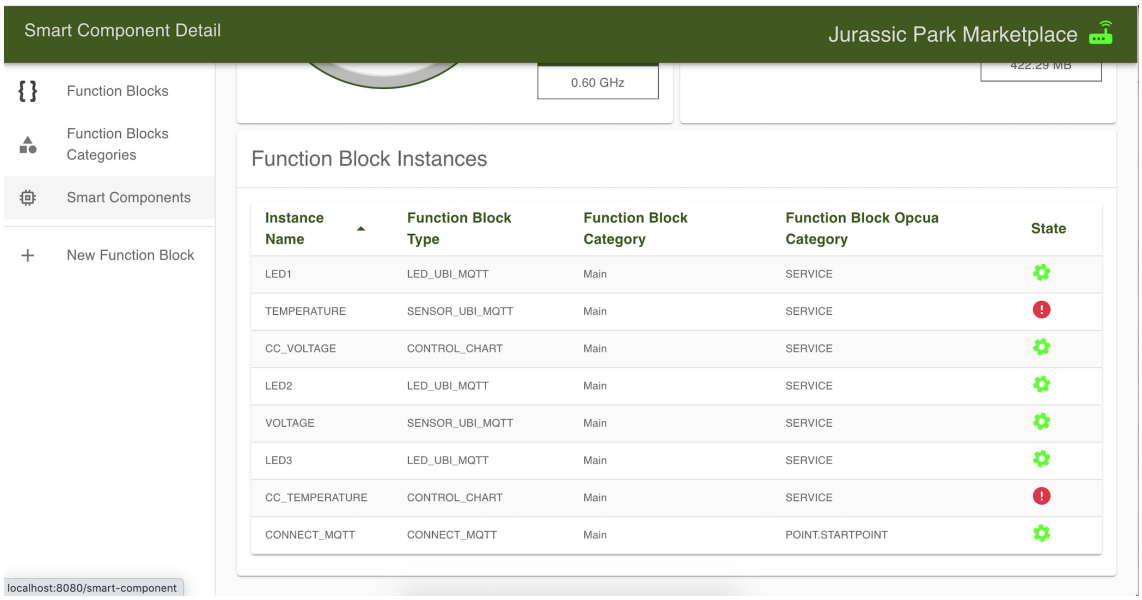


Figure 5.6: Function Block Instances Errors

As seen in figure 5.6, there are two FBs where the errors were manually induced. This state is not only correctly shown in the application but also immediately seen when the manual error inducing was done, which proves that this monitoring is done in real-time. This was also confirmed using an external industrial OPC-UA client application (PROSYS Opcua Client [19]) where this

variable was also monitored. Whenever the state goes back to a non-error state, the icon immediately turned green again, also without refreshing the browser page.

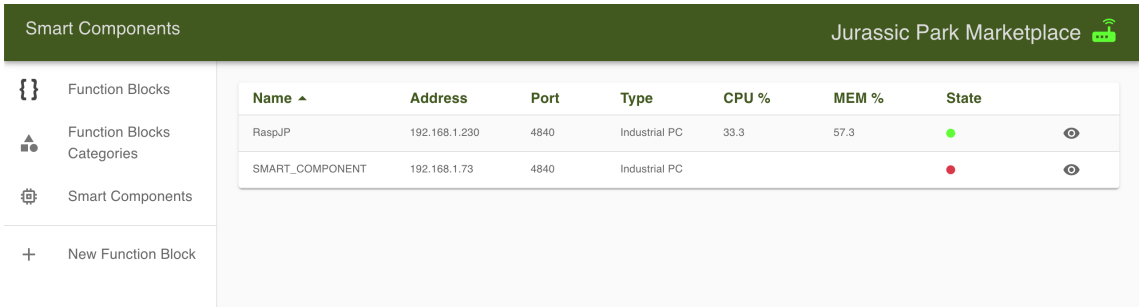


Figure 5.7: Turning off Smart Component

The second monitoring element was tested by simply disconnect one of the Smart Components connected to the marketplace. Like figure 5.7 depicts, the icon turned red on the device disconnection. The fact that the browser application is permanently connected to the Jurassic Park marketplace makes this monitoring a real-time one, without the need to refresh the page. Therefore, the change in the icon is performed immediately after the device is turned off. When the device is turned on again, the action is also perceived through the application, which changes back the icon to green for that device.

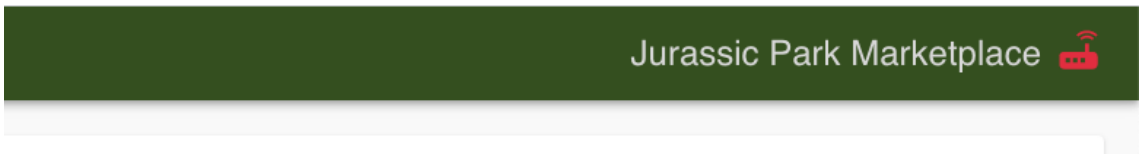


Figure 5.8: Turning off Jurassic Park

A validation on the marketplace state was also performed by shutting down the server. As it is visible in figure 5.8, the icon in the top bar of the application immediately turns red on the server shutdown. The breaking in the web socket connection caused by the server shut down immediately triggers the change in the interface if the web application. In the same manner, when the server is up again, the icon also indicates that by turning back into green color.

At last, the resource usage monitoring has also been tested. In the beginning, when the Smart Component was in an idle state, not running any instances of FBs, the resource consumption was low, with only the minimum necessary to run the Operating System and the RE. However, when the FBs were transferred to the Smart Component and the RE started to execute, it was possible to see these indicators to rise, indicating that the Smart Components resource consumption has increased. In a large distributed control system, this is a very important feature as it enables the user in charge of controlling the system to evaluate the need of adding more resources or even to have an idea of what are the FBs that are spending more resources.

5.4 Performance And Scalability

The last validation experiment was conducted with the aim of evaluating the performance and scalability of the platform. For this purpose, the scenario used was somehow different from the previous ones in terms of applicability. The IEC-61499 application developed for this experiment is not directly related to simulations in manufacturing scenarios but it is, instead, an attempt to solve a distributed optimization and simulation problem using the IEC-61499 function block programming paradigm. The problem is the classic graph theory Traveling Salesman Problem (TSP)[24] where there are a number of locations and the aim is to find the optimal path to visit each one of those locations. In this case, an evolutionary algorithm was implemented, with each of the FBs implementing a specific part of the algorithm. The system developed can be divided into two main parts and those parts can be seen in figures 5.9 and 5.10.

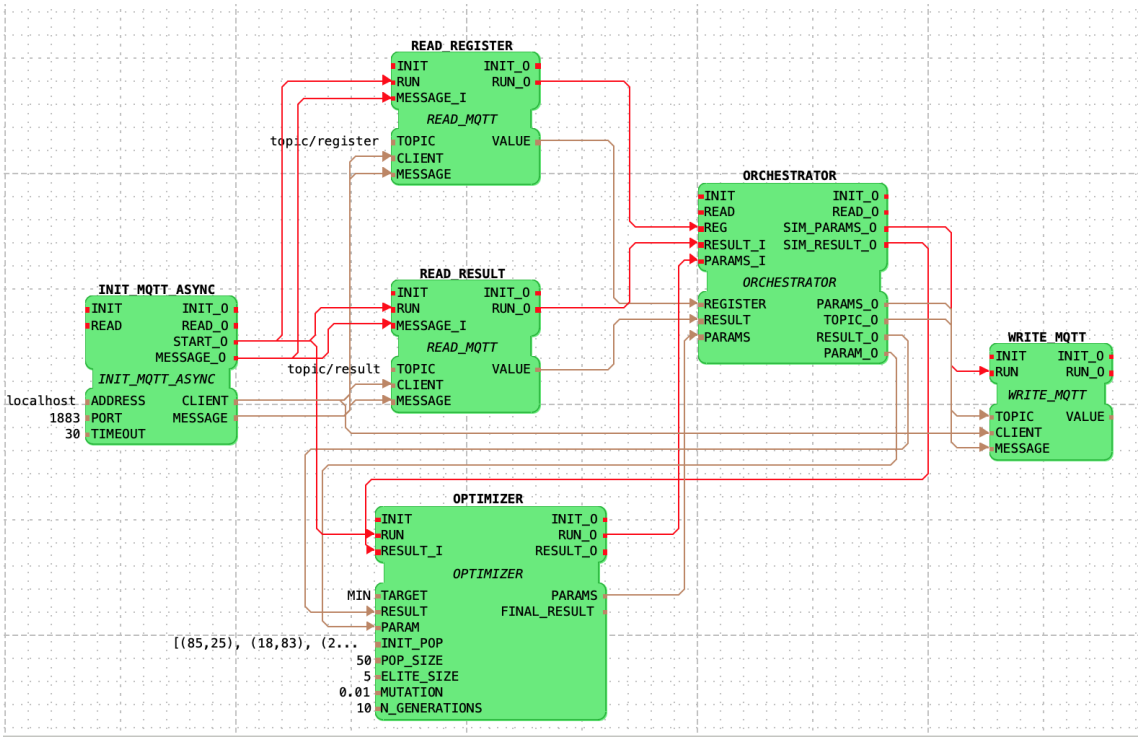


Figure 5.9: TSP Evolutionary Algorithm - Orchestrator

The main idea is to have a distributed system where there is a central orchestrator of the resources and the actual resources executing the evaluation of the solutions generated by this central component, and reporting back the results to the orchestrator part. The orchestrator part is the set of FBs shown in figure 5.9, meant to be deployed in one Smart Component and the simulator part is represented by the FBs depicted in figure 5.10 and to be deployed in each of the available resources (Smart Components as well). As an example scenario, there can be one Smart Component running the orchestrator and several Smart Components running the simulator to execute the evaluation of the solutions generated by the orchestrator (new populations in the

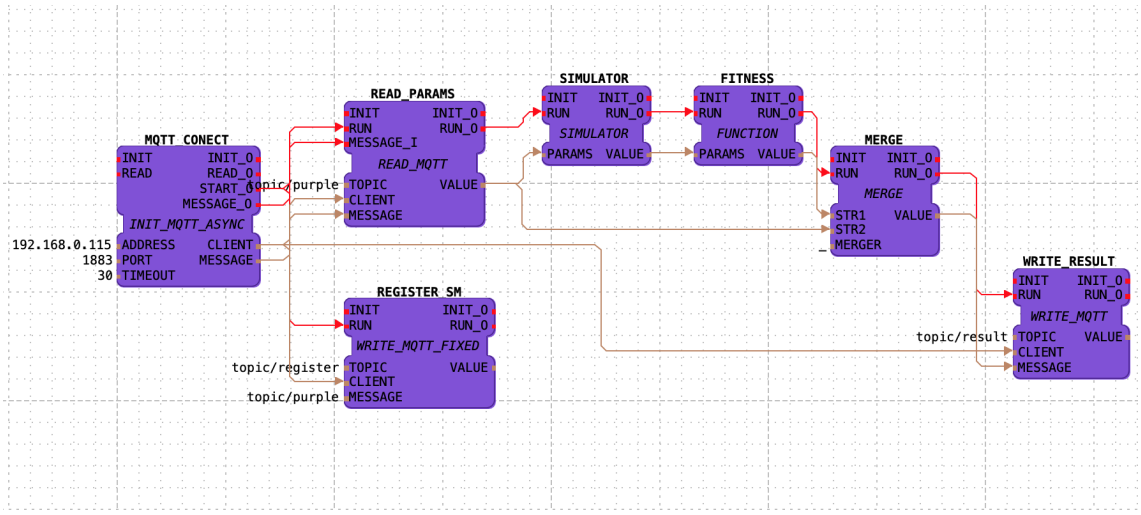


Figure 5.10: TSP Evolutionary Algorithm - Simulator

evolutionary algorithm) and report back the results to the orchestrator. In the end, the orchestrator will select the best solution computed by the simulators. With this system, it is easy to test the scalability and performance of the system without having a very complex system. It is just a matter of adding resources and deploy the simulator part into them.

To build and deploy the system, the Jurassic Park Marketplace application user had to create the following FBs:

- Orchestrator Part
  - INIT MQTT ASYNC: This FB is responsible to create an MQTT client connection and provide it to other FBs as well as read the messages published on the broker by the simulators and pass them on to the next FBs.
  - READ MQTT: This FB is used to receive values read from an MQTT broker. There are two instances of this FB, one to read the registrations of the simulators, and another to read the results of the evaluation performed by them. Both these readings are passed on to the orchestrator, the registration message makes the orchestrator add the simulator to the list of available ones and the result of the evaluation is used by the orchestrator to forward to the optimizer.
  - ORCHESTRATOR: This FB receives a new result, passes it to the Optimizer which compares the solution with previous ones. This FB also receives new solutions from the optimizer and sends them to the WRITE MQTT FB which, in turn, publish them in a topic where the simulator evaluates the solution, knowing that it must start a new evaluation with the provided solutions.
  - OPTIMIZER: The Optimizer receives the solutions sent by the Orchestrator, stops the processing if it finds the optimal solution or generate new solutions (population in a

evolutionary algorithm) and pass them on to the Orchestrator FB, which publish a message stating which is the simulator to evaluate these new solutions. It has some input variables related to the algorithm like the population size, the number of mutations, etc.

  - MQTT WRITER: This FB will publish the new solutions generated in the topic where the simulator to evaluate those solutions is subscribed. This is a round-robin in the list of the available simulators.

- Simulator Part
  - INIT MQTT ASYNC: This FB is used in the simulator part for the same reason as in the orchestrator part, to connect to a broker, read messages and pass them on to the next FBs, which will read and interpret the messages.
  - READ MQTT: In the simulator part, there is an instance of the READ MQTT FB. It is used to read the generated solutions sent by the orchestrator (new generations of populations in the evolutionary algorithm). These solutions are then passed to the next FB which is the simulator.
  - WRITE MQTT FIXED: This FB is used to publish the message that makes the simulator to be registered in the orchestrator. This message contains the id of the simulator.
  - SIMULATOR, FITNESS and MERGE: These FBs are responsible for the evaluation of the solutions and are related to the actual implementation of the algorithm. The result is then passed on to the WRITE MQTT FB which will publish it to the broker. The orchestrator, which is subscribed to the topic, will receive the result of the evaluation.
  - WRITE MQTT: There is an instance of the WRITE MQTT to send the result of the simulation back to the orchestrator via the MQTT broker.

5.4.1 Results

By conducting this experiment, it was possible to evaluate the performance of the platform when the number of Smart Components increased. The tests were performed in terms of the time that the Smart Components have taken to download the required FBs. The tests were performed varying the number of simulators. In other words, with this platform, this is the time taken to deploy the whole system.

5.4.1.1 Download Times

By measuring the times the Smart Controller needed to download the required FBs, it was possible to evaluate if this time started to increase when the number of devices connected to the marketplace grows. The experiment was performed with one Smart Device as the Orchestrator and varying the number of the simulators starting with only one until a test in which there were eight simulators. The results are expressed in terms of the time taken to download one FB.

$$\frac{Total\ Time}{Number\ of\ Simulator\ FBs \times Number\ of\ Simulators + Number\ of\ Orchestrator\ FBs} \tag{5.1}$$

The formula used to calculate the download time was the one described in 5.1.

- 1 Orchestrator, 1 Simulator

$$FBDownloadTime = \frac{24}{7 \times 1 + 6} = 1.85\ seconds \tag{5.2}$$

- 1 Orchestrator, 2 Simulators

$$FBDownloadTime = \frac{31}{7 \times 2 + 6} = 1.55\ seconds \tag{5.3}$$

- 1 Orchestrator, 4 Simulators

$$FBDownloadTime = \frac{61}{7 \times 4 + 6} = 1.79\ seconds \tag{5.4}$$

- 1 Orchestrator, 8 Simulators

$$FBDownloadTime = \frac{103}{7 \times 8 + 6} = 1.66\ seconds \tag{5.5}$$

At the end, the system was performing with one Smart Component running the orchestrator part and with eight Smart Components, each one running one simulator part, as shown in figure 5.11.

As expected, the deployment time increases as the number of Smart Components used also increases due to the fact that more Smart Components had to download the required FBs. However, this is a linear growth, meaning that the download of a FB is roughly the same, even if the complexity of the system and the number of embedded devices increases. This fact is well depicted in the chart in figure 5.12.

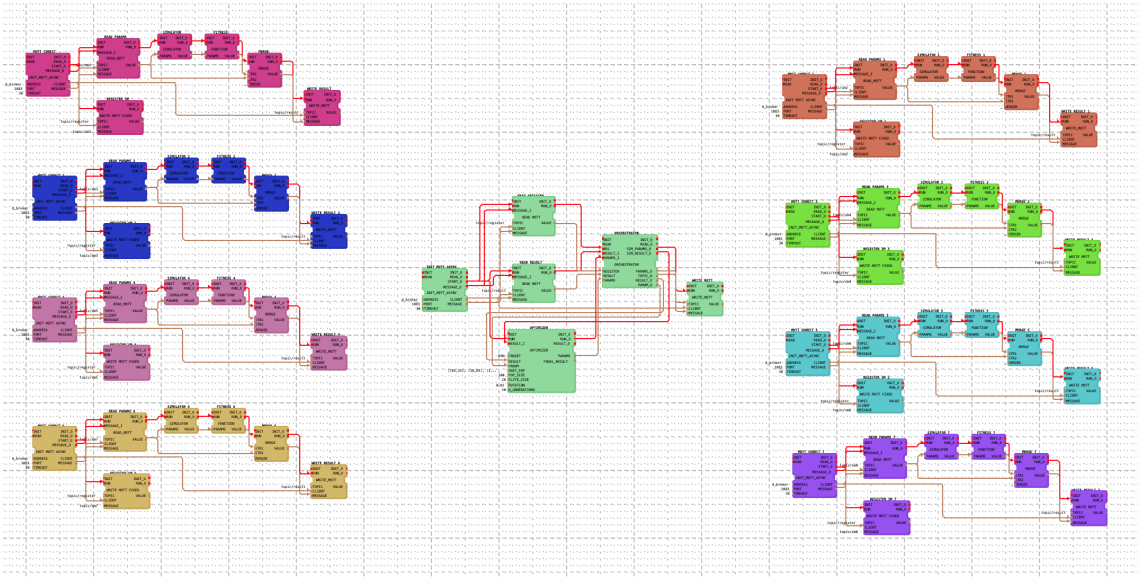


Figure 5.11: Simulators and Orchestrator

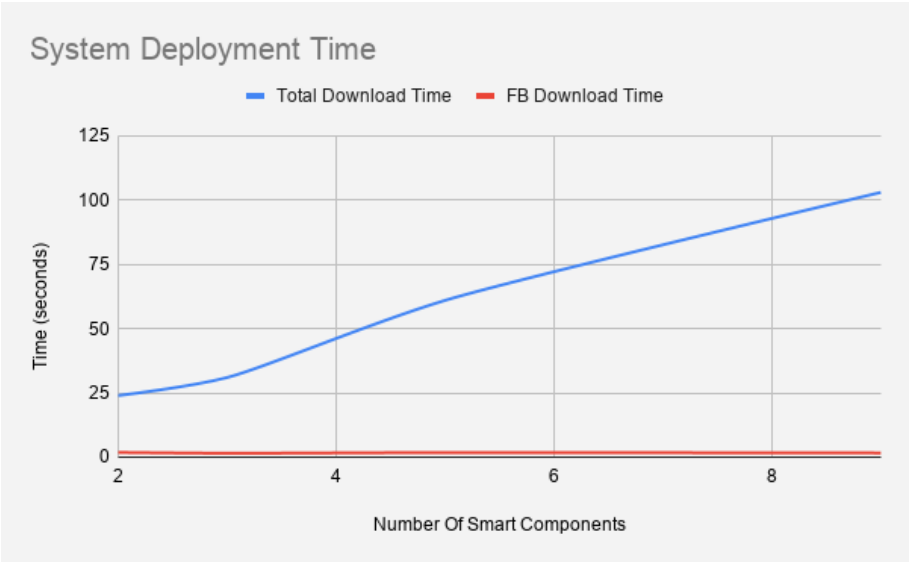


Figure 5.12: System Deployment Times

Chapter 6

Conclusions and Future Work

In this last section, it is time to discuss the developed solution and the results obtained. It is important to compare the initial requirements with the developed platform, the Jurassic Park marketplace, and to what extent it helps to smooth the process of deploying IEC-61499 systems in the context of CPSs.

6.1 Contributions

The objective of this dissertation, as mentioned in chapter 1 was to build a centralized repository of IoT software modules. These software modules are the so-called Function Blocks, the smallest piece of software that an IEC-61499 standard for developing CPPS comprises. As mentioned in chapter 2, there are a number of solutions to deploy this kind of systems. All of them are based in the main idea of splitting the process into two main parts, an IDE for developing the system, connecting the FBs to build a system that is capable of mimic and control a real CPS by means of their digital twin counterparts and a RE, installed in the embedded devices to actually run the software and resemble the system. These tools are developed taking into consideration the previous development and distribution of the software pieces. It was this idea that led to the development of this dissertation.

Therefore, the main objective of the dissertation was to build a third component to add on to the RE and the IDE and with the main goal of centralizing and store the core components of the CPPSs compliant with the IEC-61499 function block programming parading, the actual FBs. Furthermore, as this would function as a centralized module, monitoring functionalities would also be very useful to control the process. Currently, the IDEs do not have these functionalities.

According to the initial requirements, the three main goals of the dissertation were achieved.

- **Centralized repository of IoT Software Modules**
- **Integration with both the IDE and the RE**

- **Smart Component Monitoring**

On top of this, an application was developed so it can be used to manage all the functionalities provided by the platform, this application has the huge advantage of being a web application, which means that there is no need for a particular kind of software to run it, just a simple web browser, present in all modern OSs. Furthermore, by decoupling the frontend from the backend, making the HTML be rendered on the client-side, it was possible to reduce the number of communications needed as well as the size of the payloads of those communications, creating a smooth and fast user experience.

On one hand, the development of this platform will help the daily work of the factory managers, by easing the process of managing the software to be installed in the embedded devices, and, on the other hand, will also save time to the developers of the software, as they no longer need to worry about compliance and storing issues as these problems are now addressed by the platform created, the Jurassic Park marketplace. With these advantages, it becomes clear that the time to experiment new configurations in a factory floor decreases, which is crucial in a very competitive environment where a little change in the manufacturing process can have a huge impact on the final product. This way, it is very important to have tools to help the decision-makers to quickly test and change the process flow. With the development of this project, by adding a new layer on the technology stack, it is possible to increase even more the flexibility and configurability of IEC-61499 CPPS.

This platform is not only a simple software distribution system but also a sharing platform, where the developers of the software modules can store and share their work among them. The monitoring capability of the web application is also a big improvement as it helps to minimize the time to find errors and malfunctions. Without this developed feature, when the system stopped working or was not working as expected, that would require a debug session on all of the Smart Components to try to find the one with the defect and even the module that was not performing well. With the platform developed, this time is saved as it is possible to check the current state of all the system parts.

It is possible to conclude that there is still a lot of work that is possible to be done in order to improve the process of deploying CPPSs and to control their digital twins. The work developed throughout this dissertation is a contribution to that, filling the gap that was detected between the development of the software and the actual deployment in a distributed system like a Cyber-Physical Production one.

Parallel to the work developed in this dissertation, a scientific paper has been submitted to the Eclipse SAM IoT 2020 conference, a conference on Security, Artificial Intelligence, and Modeling for IoT.[8].

6.2 Future Work

As it happens with every project and with a new technology, there is always space for new features, specially after the platform starts to be actually used. Nevertheless, there are already now some identified possible new feature to develop on top of what was already built.

- **Version Control**: The version control of the FBs could potentially improve the marketplace and the way the Smart Controllers communicate with it. Currently, when a deployment is made, the Smart Components will always download the FBs required, fetching the FBs as they are stored in the marketplace. With the introduction of FB versions, the Smart Components could compare the version installed with the newest version in the marketplace and only download it in case of having an outdated version of the required FB. This would decrease even more the time to deploy the system.

- **Error History**: The error monitoring functionality is a very useful one and constitutes a major improvement for the professionals in charge of controlling a CPPS, as they can now see, in real time, the Smart Components that are not working correctly and which part of the system is failing. However, build an error history report on top of this could make them benefit even more by using the platform.

- **IDE Integration**: The integration with the IEC-61499 IDE used, the 4Diac IDE can be further improved. Currently, there is a need to close the IDE and open it again in order to have the newly created FBs available to use. This improvement would require to better investigate the IDE source code.

In general, any kind of Information System needs to be constantly updated. This platform is no exception and, as said previously, as users start to work with it, there will certainly be more significant features to improve, helping to build on the paradigm of function block programming in the context of CPPSs. Nevertheless, the main functionalities that led to the filling of the discovered gaps were developed with excellent results already demonstrated by the conducted tests. The journey of building such a system in a very complex environment, having to integrate with two subsystems already in use was a challenging one and, with it, it has been proven that even with the major benefits achieved, there is still space to improve and to make the process of managing a manufacturing process more and more flexible, with great impact in the industry.

Appendix A

Appendix

A.1 System Requirements

A.2 Data Models

- CRUD FBs
  - Create FB API endpoint
  - Create FB Frontend
  - Edit FB API endpoint
  - Edit FB Frontend
  - Delete FB API endpoint
  - Delete FB Frontend
  - List of FBs API endpoint
  - List FBs Frontend
- CRUD FB Categories
  - Create FB Category API endpoint
  - Create FB Category Frontend
  - Edit FB Category API endpoint
  - Edit FB Category Frontend
  - Delete Category FB API endpoint
  - Delete Category FB Frontend
  - List of FB Categories API endpoint
  - List FB Categories Frontend
- Upload FB metadata file to 4Diac Folder

- Serve FB files
- Dinasore RE integration with Jurassic Park
  - Download FB metadata
  - Download FB files
  - Download and install FB dependencies
- List of Smart Components Backend
- List of Smart Components Frontend
- Detail of Smart Component Backend
- Detail of Smart Component Frontend
- List of running FB instances Backend
- List of running FB instances Frontend
- OPC-UA integration between Jurassic Park and Dinasore RE
- Monitoring of FB instances

```
1 export enum FBGeneralCategory {
2     sensor='DEVICE.SENSOR',
3     service='SERVICE',
4     startPoint='POINT.STARTPOINT',
5     endPoint='POINT.ENDPOINT',
6     equipment='DEVICE.EQUIPMENT'
7 }
```

Listing A.1: Function Block General Category Model

```
1 export interface ExternalDependency {
2     edId?: number
3     edName: string
4     edVersion?: string
5 }
```

Listing A.2: Function Block External Dependency Model

```
1 export interface FunctionBlock {
2     fbId?: number
3     fbType:string
```

```
    fbDescription:string
    fbCategory?: FBCategory
    fbFbcId: number
    fbCategoryName?: string
    fbUser?: User
    fbUserId: number
    fbGeneralCategory: '' | FBGeneralCategory
    fbInputEvents: Event[]
    fbOutputEvents: Event[]
    fbInputVariables: Variable[]
    fbOutputVariables: Variable[]
    fbExternalDependencies: ExternalDependency[]
    fbFile?: string
}
```

Listing A.3: Function Block Model

```
export interface FbInstance {
    id:string
    fbType: string
    fbCategory?: string
    fbGeneralCategory?: FBGeneralCategory | ''
    state: number
}
```

Listing A.4: Function Block Instance Model

```
export enum InOutType {
    in= 'IN',
    out= 'OUT'
}
```

Listing A.5: Event and Variable Type Model

```
export enum DataType {
    dtString='STRING',
    dtInt='INT',
    dtUint='UINT',
    dtReal='REAL',
    dtLReal='LREAL',
    dtBool='BOOL'
}
```

Listing A.6: Data Type Model

```
export interface Event {
    eventId?: number
    eventType: string
    eventName: string
    eventOPC-UA?: string
    eventInoutType: InOutType
    functionBlock?: FunctionBlock
    eventVariables: EventVariable[]
    eventFbId?: number
}
```

Listing A.7: Event Model

```
export interface Variable {
    variableId?: number
    variableName: string
    variableOPC-UA?: string
    variableInoutType: InOutType
    functionBlock?: FunctionBlock
    variableDataType: '' | DataType
    variableFbId?: number
}
```

Listing A.8: Variable Model

```
export interface EventVariable {
    evEventId?: number
    evVariableId?: number
    evVariableName: string
    evEventName: string
    evValid?: boolean
}
```

Listing A.9: Event Variable Model

```
export interface SmartComponent {
    scId?: number
    scName:string
    scAddress:string
    scPort:number
    scState?: string
    scType?: string,
    cpuPercent?: number
    cpuFreqCurrent?: number
```

```
10     cpuFreqMin?: number
11     cpuFreqMax?: number
12     cpuFreq?: number
13     memAvailable?: number
14     memCached?: number
15     memPercentage?: number
16     memShared?: number
17     memTotal?: number
18     memUsed?: number
19     fbInstances?: FbInstance []
20 }
```

Listing A.10: Smart Component Model

A.3 Function Block DTD

```
1  <!-- Common elements -->
2  <!ELEMENT Identification EMPTY>
3  <!ATTLIST Identification
4    Standard CDATA #IMPLIED
5    Classification CDATA #IMPLIED
6    ApplicationDomain CDATA #IMPLIED
7    Function CDATA #IMPLIED
8    Type CDATA #IMPLIED
9    Description CDATA #IMPLIED >
10 <!ELEMENT VersionInfo EMPTY>
11 <!ATTLIST VersionInfo
12   Organization CDATA #REQUIRED
13   Version CDATA #REQUIRED
14   Author CDATA #REQUIRED
15   Date CDATA #REQUIRED
16   Remarks CDATA #IMPLIED >
17 <!ELEMENT CompilerInfo (Compiler*)>
18 <!ATTLIST CompilerInfo
19   header CDATA #IMPLIED
20   classdef CDATA #IMPLIED >
21 <!ELEMENT Compiler EMPTY>
22 <!ATTLIST Compiler
23   Language (Java | Cpp | C | Other) #REQUIRED
24   Vendor CDATA #REQUIRED
25   Product CDATA #REQUIRED
26   Version CDATA #REQUIRED >
27 <!ELEMENT FBNetwork (FB*,EventConnections?,DataConnections?,AdapterConnections?)>
28 <!ELEMENT FB (Parameter*)>
29 <!ATTLIST FB
30   Name CDATA #REQUIRED
```

```
31   Type CDATA #REQUIRED
32   Comment CDATA #IMPLIED
33    x CDATA #IMPLIED
34    y CDATA #IMPLIED >
35 <!ELEMENT EventConnections (Connection+)>
36 <!ELEMENT DataConnections (Connection+)>
37 <!ELEMENT AdapterConnections (Connection+)>
38 <!ELEMENT Connection EMPTY>
39 <!ATTLIST Connection
40   Source CDATA #REQUIRED
41   Destination CDATA #REQUIRED
42   Comment CDATA #IMPLIED
43   dx1 CDATA #IMPLIED
44   dx2 CDATA #IMPLIED
45   dy CDATA #IMPLIED >
46
47 <!-- FBType elements -->
48 <!ELEMENT FBType (Identification?,VersionInfo+,CompilerInfo?,InterfaceList, (
       BasicFB | SimpleFB | FBNetwork)?, Service?) >
49 <!ATTLIST FBType
50   Name CDATA #REQUIRED
51   Comment CDATA #IMPLIED >
52 <!ELEMENT InterfaceList (EventInputs?,EventOutputs?,InputVars?,OutputVars?,
       Sockets?, Plugs?)>
53 <!ELEMENT EventInputs (Event+)>
54 <!ELEMENT EventOutputs (Event+)>
55 <!ELEMENT InputVars (VarDeclaration+)>
56 <!ELEMENT OutputVars (VarDeclaration+)>
57 <!ELEMENT Sockets (AdapterDeclaration+)>
58 <!ELEMENT Plugs (AdapterDeclaration+)>
59 <!ELEMENT Event (With*)>
60 <!ATTLIST Event
61   Name CDATA #REQUIRED
62   Type CDATA #IMPLIED
63   Comment CDATA #IMPLIED >
64 <!ELEMENT With EMPTY>
65 <!ATTLIST With
66   Var CDATA #REQUIRED >
67 <!ELEMENT VarDeclaration EMPTY>
68 <!ATTLIST VarDeclaration
69   Name CDATA #REQUIRED
70   Type CDATA #REQUIRED
71   ArraySize CDATA #IMPLIED
72   InitialValue CDATA #IMPLIED
73   Comment CDATA #IMPLIED >
74 <!ELEMENT AdapterDeclaration (Parameter*)>
75 <!ATTLIST AdapterDeclaration
76    Name CDATA #REQUIRED
77   Type CDATA #REQUIRED
```

```
78    Comment CDATA #IMPLIED
79    x CDATA #IMPLIED
80    y CDATA #IMPLIED >
81  <!ELEMENT BasicFB (InternalVars?,ECC,Algorithm*)>
82  <!ELEMENT SimpleFB (InternalVars?,ECState+,Algorithm*)>
83  <!ELEMENT InternalVars (VarDeclaration+)>
84  <!ELEMENT ECC (ECState+,ECTransition+) >
85  <!ELEMENT ECState (ECAction*)>
86  <!ATTLIST ECState
87    Name CDATA #REQUIRED
88    Comment CDATA #IMPLIED
89    x CDATA #IMPLIED
90    y CDATA #IMPLIED >
91  <!ELEMENT ECTransition EMPTY>
92  <!ATTLIST ECTransition
93    Source CDATA #REQUIRED
94    Destination CDATA #REQUIRED
95    Condition CDATA #REQUIRED
96    Comment CDATA #IMPLIED
97    x CDATA #IMPLIED
98    y CDATA #IMPLIED >
99  <!ELEMENT ECAction EMPTY>
100 <!ATTLIST ECAction
101   Algorithm CDATA #IMPLIED
102   Output CDATA #IMPLIED >
103 <!ELEMENT Algorithm (VarDeclaration*,(FBD | ST | LD | Other))>
104 <!ATTLIST Algorithm
105   Name CDATA #REQUIRED
106   Comment CDATA #IMPLIED >
107 <!ELEMENT FBD (FB+,DataConnections) >
108 <!ELEMENT ST (#PCDATA)>
109 <!ATTLIST ST
110   Text CDATA #IMPLIED >
111 <!ELEMENT LD (Rung+) >
112 <!ELEMENT Rung (#PCDATA)>
113 <!ATTLIST Rung
114   Output CDATA #REQUIRED
115   Expression CDATA #IMPLIED
116   Comment CDATA #IMPLIED >
117 <!ELEMENT Other (#PCDATA)>
118 <!ATTLIST Other
119   Language CDATA #REQUIRED
120   Text CDATA #IMPLIED >
121 <!ELEMENT Service (ServiceSequence+)>
122 <!ATTLIST Service
123   RightInterface CDATA #REQUIRED
124   LeftInterface CDATA #REQUIRED
125   Comment CDATA #IMPLIED >
126 <!ELEMENT ServiceSequence (ServiceTransaction*)>
```

```
127 <!ATTLIST ServiceSequence
128   Name CDATA #REQUIRED
129   Comment CDATA #IMPLIED >
130 <!ELEMENT ServiceTransaction (InputPrimitive?, OutputPrimitive*)>
131 <!ELEMENT InputPrimitive EMPTY>
132 <!ATTLIST InputPrimitive
133   Interface CDATA #REQUIRED
134   Event CDATA #REQUIRED
135   Parameters CDATA #IMPLIED >
136 <!ELEMENT OutputPrimitive EMPTY>
137 <!ATTLIST OutputPrimitive
138   Interface CDATA #REQUIRED
139   Event CDATA #REQUIRED
140   Parameters CDATA #IMPLIED >
141
142 <!-- AdapterType elements -->
143 <!ELEMENT AdapterType (Identification?,VersionInfo+,CompilerInfo?,InterfaceList,
      Service?)>
144 <!ATTLIST AdapterType
145   Name CDATA #REQUIRED
146   Comment CDATA #IMPLIED >
147
148 <!-- ResourceType elements -->
149 <!ELEMENT ResourceType (Identification?,VersionInfo+, CompilerInfo?, FBTypeName*,
      VarDeclaration*, FBNetwork)>
150 <!ATTLIST ResourceType
151   Name CDATA #REQUIRED
152   Comment CDATA #IMPLIED >
153 <!ELEMENT FBTypeName EMPTY>
154 <!ATTLIST FBTypeName
155   Name CDATA #REQUIRED >
156
157 <!-- DeviceType elements -->
158 <!ELEMENT DeviceType (Identification?,VersionInfo+, CompilerInfo?, VarDeclaration*,
       ResourceTypeName*, Resource*, FBNetwork?)>
159 <!ATTLIST DeviceType
160   Name CDATA #REQUIRED
161   Comment CDATA #IMPLIED >
162 <!ELEMENT ResourceTypeName EMPTY>
163 <!ATTLIST ResourceTypeName
164   Name CDATA #REQUIRED >
165 <!ELEMENT Resource (Parameter*,FBNetwork?)>
166 <!ATTLIST Resource
167   Name CDATA #REQUIRED
168   Type CDATA #REQUIRED
169   Comment CDATA #IMPLIED
170   x CDATA #IMPLIED
171   y CDATA #IMPLIED >
172
```

```
173 <!-- System elements -->
174 <!ELEMENT System (Identification?, VersionInfo+, Application*, Device+,  Mapping*,
        Segment*, Link*)>
175 <!ATTLIST System
176   Name CDATA #REQUIRED
177   Comment CDATA #IMPLIED >
178 <!ELEMENT Application (SubAppNetwork)>
179 <!ATTLIST Application
180   Name CDATA #REQUIRED
181   Comment CDATA #IMPLIED >
182 <!ELEMENT Mapping EMPTY>
183 <!ATTLIST Mapping
184   From CDATA #REQUIRED
185   To CDATA #REQUIRED >
186 <!ELEMENT Device (Parameter*,Resource*,FBNetwork?)>
187 <!ATTLIST Device
188   Name CDATA #REQUIRED
189   Type CDATA #REQUIRED
190   Comment CDATA #IMPLIED
191   x CDATA #IMPLIED
192   y CDATA #IMPLIED >
193
194 <!-- SubAppType elements -->
195 <!ELEMENT SubAppType (Identification?,VersionInfo+,CompilerInfo?,
        SubAppInterfaceList, SubAppNetwork?)> <!ATTLIST SubAppType
196   Name CDATA #REQUIRED
197   Comment CDATA #IMPLIED >
198 <!ELEMENT SubAppInterfaceList (SubAppEventInputs?,SubAppEventOutputs?,InputVars?,
        OutputVars?)>
199 <!ELEMENT SubAppEventInputs (SubAppEvent+)>
200 <!ELEMENT SubAppEventOutputs (SubAppEvent+)>
201 <!ELEMENT SubAppEvent EMPTY>
202 <!ATTLIST SubAppEvent
203   Name CDATA #REQUIRED
204   Type CDATA #IMPLIED
205   Comment CDATA #IMPLIED >
206 <!ELEMENT SubAppNetwork (SubApp*,FB*,EventConnections?,DataConnections?)>
207 <!ELEMENT SubApp EMPTY>
208 <!ATTLIST SubApp
209   Name CDATA #REQUIRED
210   Type CDATA #REQUIRED
211   Comment CDATA #IMPLIED
212    x CDATA #IMPLIED
213    y CDATA #IMPLIED >
214
215 <!-- Network elements -->
216 <!ELEMENT SegmentType (Identification?,VersionInfo+, CompilerInfo?, VarDeclaration
        *)>
217 <!ATTLIST SegmentType
```

```
218   Name CDATA #REQUIRED
219   Comment CDATA #IMPLIED >
220 <!ELEMENT Segment (Parameter*)>
221 <!ATTLIST Segment
222   Name CDATA #REQUIRED
223   Type CDATA #REQUIRED
224   Comment CDATA #IMPLIED
225   x CDATA #IMPLIED
226   y CDATA #IMPLIED
227   dx1 CDATA #IMPLIED>
228 <!ELEMENT Parameter EMPTY>
229 <!ATTLIST Parameter
230   Name CDATA #REQUIRED
231   Value CDATA #REQUIRED
232   Comment CDATA #IMPLIED>
233 <!ELEMENT Link (Parameter*)>
234 <!ATTLIST Link
235   SegmentName CDATA #REQUIRED
236   CommResource CDATA #REQUIRED
237   Comment CDATA #IMPLIED >
```

Listing A.11: Function Block DTD

A.4 Database Schema

```
1  CREATE TABLE FBCategory(
2
3      fbcId INT PRIMARY KEY AUTO_INCREMENT,
4      fbcName VARCHAR(50) NOT NULL UNIQUE,
5      fbcUserId INT,
6
7      FOREIGN KEY(fbcUserId) references User(userId) ON DELETE SET NULL ON UPDATE
           CASCADE
8  );
9
10 CREATE TABLE FunctionBlock(
11
12     fbId INT PRIMARY KEY AUTO_INCREMENT,
13     fbType VARCHAR(50) NOT NULL UNIQUE,
14     fbDescription VARCHAR(200) NOT NULL,
15     fbUserId INT,
16     fbFbcId INT NOT NULL,
17     fbGeneralCategory ENUM('DEVICE.SENSOR','SERVICE','POINT.STARTPOINT','POINT.
           ENDPOINT','DEVICE.EQUIPMENT') NOT NULL,
18     fbSize INT,
19
```

```
20     FOREIGN KEY(fbFbcId) references FBCategory(fbcId) ON UPDATE CASCADE,
21     FOREIGN KEY(fbUserId) references User(userId) ON DELETE SET NULL ON UPDATE
           CASCADE
22
23 );
24
25 CREATE TABLE Event(
26
27     eventId INT PRIMARY KEY AUTO_INCREMENT,
28     eventName VARCHAR(50) NOT NULL,
29     eventType VARCHAR(50) NOT NULL,
30     eventOPC-UA VARCHAR(75),
31     eventInoutType ENUM('IN', 'OUT'),
32     eventFbId INT NOT NULL,
33
34     UNIQUE KEY(eventName,eventFbId),
35
36     FOREIGN KEY(eventFbId) references FunctionBlock(fbId) ON DELETE CASCADE ON
           UPDATE CASCADE
37
38 );
39
40 CREATE TABLE Variable(
41
42     variableId INT PRIMARY KEY AUTO_INCREMENT,
43     variableName VARCHAR(50) NOT NULL,
44     variableOPC-UA VARCHAR(75),
45     variableInoutType ENUM('IN', 'OUT'),
46     variableDataType ENUM('STRING','INT','UINT','REAL','LREAL','BOOL'),
47     variableFbId INT NOT NULL,
48
49     UNIQUE(variableName,variableFbId),
50
51     FOREIGN KEY(variableFbId) references FunctionBlock(fbId) ON DELETE CASCADE ON
           UPDATE CASCADE
52
53 );
54
55 CREATE TABLE EventVariable (
56
57     evEventId INT,
58     evVariableId INT,
59     evValid BOOLEAN NOT NULL DEFAULT TRUE,
60
61     PRIMARY KEY(evEventId,evVariableId),
62     FOREIGN KEY(evEventId) REFERENCES Event(eventId) ON DELETE CASCADE ON UPDATE
           CASCADE,
63     FOREIGN KEY(evVariableId) REFERENCES Variable(variableId) ON DELETE CASCADE ON
           UPDATE CASCADE
```

```
64
65 );
66
67 CREATE TABLE ExternalDependency (
68
69     edId INT PRIMARY KEY AUTO_INCREMENT,
70     edName VARCHAR(50) NOT NULL
71
72 );
73
74 CREATE TABLE FunctionBlockExternalDependency(
75
76     fbedFbId INT NOT NULL,
77     fbedEdId INT NOT NULL,
78     fbEdVersion VARCHAR(20),
79
80     PRIMARY KEY(fbedFbId,fbedEdId),
81     FOREIGN KEY(fbedFbId) REFERENCES FunctionBlock(fbId) ON DELETE CASCADE ON
           UPDATE CASCADE,
82     FOREIGN KEY(fbedEdId) REFERENCES ExternalDependency(edId) ON DELETE CASCADE ON
           UPDATE CASCADE
83
84 );
85
86 CREATE FUNCTION checkEventVariable (inEventId INT, inVariableId INT) RETURNS
       BOOLEAN
87 DETERMINISTIC
88 BEGIN
89     DECLARE localEventFbId INT;
90     DECLARE localVariableFbId INT;
91     DECLARE localEventInoutType VARCHAR(5);
92     DECLARE localVariableInoutType VARCHAR(5);
93     DECLARE result BOOLEAN;
94     SELECT eventFbId,eventInoutType INTO localEventFbId,localEventInoutType FROM
           Event WHERE eventId=inEventId;
95     SELECT variableFbId,variableInoutType INTO localVariableFbId,
           localVariableInoutType FROM Variable WHERE variableId=inVariableId;
96
97     IF (localEventFbId = localVariableFbId AND localEventInoutType =
           localVariableInoutType ) THEN
98         SET result = TRUE;
99     ELSE
100        SET result = FALSE;
101
102    END IF;
103
104    RETURN result;
105 END
106 $$
```

```
107
108
109 CREATE TRIGGER ValidEventVariableInsert BEFORE INSERT ON EventVariable
110
111     FOR EACH ROW
112     BEGIN
113         IF NOT checkEventVariable(New.evEventId,New.evVariableId) THEN
114             SET New.evValid = FALSE;
115         END IF;
116     END
117 $$
118
119 CREATE TRIGGER ValidEventVariableUpdate BEFORE UPDATE ON EventVariable
120
121     FOR EACH ROW
122     BEGIN
123         IF NOT checkEventVariable(New.evEventId,New.evVariableId) THEN
124             SET New.evValid = FALSE;
125         END IF;
126     END
127 $$
```

Listing A.12: Database Schema

References

[1] App Store. https://www.apple.com/ios/app-store/. Online; accessed February 2020.

[2] The best front-end framework to learn in 2019. https://dev.to/duomly/the-best-front-end-framework-to-learn-in-2019-dn7. Online; accessed February 2020.

[3] Chrome Web Store. https://support.google.com/chrome_webstore/answer/1047776?hl=pt-BR&topic=1212379. Online; accessed December 2019.

[4] Comparing popular database management systems. https://www.altexsoft.com/blog/business/comparing-database-managemen-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/. Online; accessed February 2020.

[5] Composer. https://getcomposer.org/doc/. Online; accessed February 2020.

[6] Documentation | MQTT. http://mqtt.org/documentation. Online; accessed February 2020.

[7] Eclipse 4diac Documentation. https://www.eclipse.org/4diac/en_help.php?helppage=html/before4DIAC/4diacFramework.html.

[8] Eclipse SAM IoT 2020 | The Eclipse Foundation. https://events.eclipse.org/2020/sam-iot/. Online; accessed April 2020.

[9] Express - Node.js web application framework. https://expressjs.com/. Online; accessed April 2020.

[10] FBDK - The Function Block Development Kit. https://www.holobloc.com/doc/fbdk/index.htm. Online; accessed February 2020.

[11] FEUP - Research Center for Systems and Technologies. https://sigarra.up.pt/feup/en/uni_geral.unidade_view?pv_unidade=852. Online; accessed February 2020.

[12] Function block dtd. http://www.holobloc.com/xml/LibraryElement.dtd. Online; accessed May 2020.

[13] Getting Started – React. https://reactjs.org/docs/getting-started.html. Online; accessed February 2020.

[14] Guides - RubyGems Guides. https://guides.rubygems.org/. Online; accessed January 2020.

[15] HTTP. https://developer.mozilla.org/en-US/docs/Web/HTTP. Online; accessed January 2020.

[16] IEC61499 - International Standard for Distributed Systems. https://www.iec61499.de/. Online; accessed February 2020.

[17] npm Documentation. https://docs.npmjs.com/. Online; accessed January 2020.

[18] pip - The Python Package Installer — pip 20.0.2 documentation. https://pip.pypa.io/en/stable/. Online; accessed February 2020.

[19] Prosys OPC UA Client. https://downloads.prosysopc.com/opcua/apps/JavaClient/dist/3.2.0-328/Prosys_OPC_UA_Client_UserManual.pdf. Online; accessed April 2020.

[20] Socket.IO — Docs. https://socket.io/docs/index.html. Online; accessed April 2020.

[21] vsftpd - Secure, fast FTP server for UNIX-like systems. https://security.appspot.com/vsftpd.html. Online; accessed February 2020.

[22] What is OPC? https://opcfoundation.org/about/what-is-opc/. Online; accessed May 2020, Library Catalog: opcfoundation.org.

[23] DIGI2-FEUP/dinasore. https://github.com/DIGI2-FEUP/dinasore, June 2020. Online; accessed February 2020.

[24] Travelling salesman problem. https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=964564555, June 2020. Online; accessed May 2020.

[25] Z. M. Bi, S. Y. T. Lang, W. Shen, and L. Wang. Reconfigurable manufacturing systems: the state of the art. *International Journal of Production Research*, 46(4):967–992, February 2008.

[26] Alessandro Brusaferi, Andrea Ballarino, and Emanuele Carpanzano. Reconfigurable Knowledge-based Control Solutions for Responsive Manufacturing Systems. *Studies in Informatics and Control*, 20(1), March 2011.

[27] James H Christensen, Thomas Strasser, and Antonio Valentini. The IEC 61499 Function Block Standard: Software Tools and Runtime Platforms. page 12, March 2009.

[28] Tyler Crawford and Tauqeer Hussain. A Comparison of Server Side Scripting Technologies. page 8.

[29] Michael Crosby. BlockChain Technology: Beyond Bitcoin. (2):16, 2016.

[30] Wenbin Dai and Valeriy Vyatkin. Redesign Distributed PLC Control Systems Using IEC 61499 Function Blocks. *IEEE Transactions on Automation Science and Engineering*, 9(2):390–401, April 2012.

[31] Dale Fisk. Programming with Punched Cards. http://www.columbia.edu/cu/computinghistory/fisk.pdf, 2005. Online; accessed February 2020.

[32] Google. Google Play. https://play.google.com/store?hl=en. Online; accessed January 2020.

[33] Shah Ahsanul Haque, Syed Mahfuzul Aziz, and Mustafizur Rahman. Review of Cyber-Physical System in Healthcare. *International Journal of Distributed Sensor Networks*, 10(4):217415, April 2014.

[34] Bhaskar Krishnamachari, Jerry Power, Seon Ho Kim, and Cyrus Shahabi. I3: An IoT Marketplace for Smart Communities. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '18*, pages 498–499, Munich, Germany, 2018. ACM Press.

[35] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, Orlando, FL, USA, May 2008. IEEE.

[36] Jay Lee, Behrad Bagheri, and Hung-An Kao. A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18–23, January 2015.

[37] Wolfgang Mahnke and Stefan-Helmut Leitner. The future standard for communication and information modeling in automation. *OPC Unified Architecture*, page 6, March 2009.

[38] Kresimir Misura and Mario Zagar. Data marketplace for Internet of Things. In *2016 International Conference on Smart Systems and Technologies (SST)*, pages 255–260, Osijek, Croatia, October 2016. IEEE.

[39] Dimitris Mourtzis and Ekaterini Vlachou. A cloud-based cyber-physical system for adaptive shop-floor scheduling and condition-based maintenance. *Journal of Manufacturing Systems*, 47:179–198, April 2018.

[40] Cheng Pang, Sandeep Patil, Chen-Wei Yang, Valeriy Vyatkin, and Anatoly Shalyto. A portability study of IEC 61499: Semantics and tools. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 440–445, Porto Alegre RS, Brazil, July 2014. IEEE.

[41] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a service model for smart cities supported by Internet of Things. *Transactions on Emerging Telecommunications Technologies*, 25(1):81–93, January 2014.

[42] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference on - DAC '10*, page 731, Anaheim, California, 2010. ACM Press.

[43] Alejandro Seif, Carlos Toro, and Humza Akhtar. Implementing Industry 4.0 Asset Administrative Shells in Mini Factories. *Procedia Computer Science*, 159:495–504, 2019.

[44] Steven Spielberg. Jurassic Park, June 1993. event-location: USA.

[45] Stéphane Surowka. A Brief History of User Interfaces 1/2, July 2019. Online; accessed February 2020.

[46] Tarik Terzimehic, Andreas Bayha, and Kirill Dorofeev. Function Blocks for the Interaction with the Asset Administration Shell. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1235–1238, Zaragoza, Spain, September 2019. IEEE.

[47] Valeriy Vyatkin. IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review. *IEEE Transactions on Industrial Informatics*, 7(4):768–781, November 2011.

[48] Xinghuo Yu and Yusheng Xue. Smart Grids: A Cyber–Physical Systems Perspective. *Proceedings of the IEEE*, 104(5):1058–1070, May 2016.