



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

PRACA DYPLOMOWA MAGISTERSKA

*Adaptive Web User Interfaces
for TANGO Control System*

*Adaptowalne Interfejsy Użytkownika
dla Systemu Kontroli TANGO*

Autor:	Michał Liszcz
Kierunek:	Informatyka
Opiekun pracy:	dr inż. Włodzimierz Funika

Kraków, 2016

OŚWIADCZENIE AUTORA PRACY

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): "Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) "Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej 'sądem koleżeńskim'", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

.....

PODPIS

Acknowledgements

I would like to express my gratitude to my supervisor, **Dr. Włodzimierz Funika** for his guidance and support over the last two years. This work would not have been possible without his help and engagement.

I would also like to thank the whole NSRC Solaris team, especially **Mr. Łukasz Żytniak** and **Mr. Piotr Goryl**. Łukasz was an invaluable source of field-related expertise while Piotr took the responsibility for the project management.

Abstract

Hardware installations in natural sciences and industry are becoming more and more complex. This raised a need for more sophisticated methods of control over growing number of devices. To address these needs, the **TANGO Control System** has been developed at ESRF as a generic framework for building distributed control systems. TANGO allows building graphical control panels composed of various widgets. As TANGO is a CORBA-based software, these client applications can run only on desktop.

Recently, thanks to their numerous benefits over conventional desktop applications, **web applications** are more and more widely used in different areas. Among a few attempts of moving TANGO applications to the web browsers, none gained popularity and wide adoption. Most projects have been abandoned reaching only a proof-of-concept stage.

In Thesis we present **TangoJS**, a modular, extensible software stack for building TANGO client applications. TangoJS addresses the faults and drawbacks of the existing solutions. Its architecture is layered for increased flexibility and easier maintenance. Multiple backend modules are supported, which allows accessing TANGO infrastructure using different technologies and protocols. TangoJS' Programmable API uses well-defined abstractions and interfaces from TANGO IDL specification. Web application developers are provided with a set of configurable widgets, which are building blocks for more complex user interfaces. TangoJS has been designed in compliance with the latest web standards and componentization in mind. The use of frontend technologies allows for building adaptive user interfaces, where widget's sizing and layout depends on context. Other than that, TangoJS provides an interactive application that allows to manually manipulate its interface at runtime.

Contents

1	Introduction	11
1.1	Research background	11
1.2	Goals of Thesis	13
2	Overview of TANGO Control System	15
2.1	Visualization and control functionality in TANGO Control System	15
2.2	TANGO GUI frameworks	17
3	State-of-the-art	19
3.1	Canone	19
3.2	Taurus Web	21
3.3	Tango REST	22
3.4	GoTan	22
3.5	mTango	22
3.6	Summary	24
4	Solution and Implementation	27
4.1	Introduction to TangoJS	27
4.2	Design Goals	28
4.3	Design Decisions	29
4.4	Architecture of TangoJS	30
4.5	TangoJS Core - the TANGO API for browsers	32
4.6	TangoJS Connector - pluggable backends	33
4.7	TangoJS WebComponents - HTML widget toolkit	34
4.8	Interworking between TangoJS layers	37
4.9	TangoJS Panel - synoptic panel application	39
5	Results	41
5.1	Goals review	41
5.2	Comparison with existing solutions	42
5.3	Usability evaluation	43
6	Conclusions	47
6.1	Role of web-based solutions in GUI development	47
6.2	Connecting TANGO and the Web	48

6.3	Future work	50
	References	53
	Glossary	59
A	Getting started with TangoJS	63
A.1	Configuring the backend	63
A.2	Managing a Node.js project	65
A.3	Using the <i>TangoJS WebApp Template</i>	66
A.4	Application code	66
B	Developing widgets for TangoJS	71
B.1	<i>TangoJS WebComponents</i> utilities	71
B.2	Widget's behavior	73
B.3	Building a basic widget	73
C	CORBA IDL to Javascript translation	79
D	Selected Aspects of Implementation	81
D.1	Web Components	81
D.2	ECMAScript 2015	84
D.3	CSS Level 3 modules	88
D.4	Limitations and browser support	89

List of Figures

1.1	A storage ring magnet in National Synchrotron Radiation Centre “Solaris” in Krakow. (source [1])	11
2.1	TANGO Control System architecture overview.	16
2.2	<i>JDrow</i> synoptic panel editor.	18
2.3	Taurus GUI.	18
3.1	Canone Web Interface. (source [24])	19
3.2	Canone architecture.	20
3.3	Taurus Web Interface. (source [25])	21
3.4	mTango architecture.	23
4.1	An overview of TangoJS high-level architecture.	31
4.2	TangoJS component-level architecture.	31
4.3	Class diagram of <i>TangoJS Core</i> package.	32
4.4	Dependency between the <i>TangoJS Core</i> and the <i>Connector</i>	33
4.5	Reading the value of an attribute using TangoJS Core API.	34
4.6	TangoJS widgets.	36
4.7	Widget’s lifecycle. Interactions with <i>TangoJS Core</i> has been simplified. . .	38
4.8	Example views from <i>TangoJS Panel</i> application.	40
6.1	Comparison of backend access methods.	49
6.2	Accessing TANGO from within web browser over HTIOP.	50
A.1	Steps required to create basic TangoJS application.	64
A.2	The <i>TangoJS WebApp Template</i> application’s view.	67
B.1	Widget states and transitions.	74
B.2	Interactions of a widget with other modules during its lifecycle. Message details omitted.	75

Chapter 1

Introduction

This chapter provides an introduction to the field of research, presents essential concepts and describes motivations behind Thesis. At the end, project goals are formulated.

1.1. Research background

Control of the expensive and sensitive hardware components in large installations like scientific facilities may be a challenging task. In order to conduct an experiment, multiple elements like motors, ion pumps, valves and power-supplies have to be orchestrated. An example part of such an installation, located at NSRC Solaris is presented in Fig. 1.1.

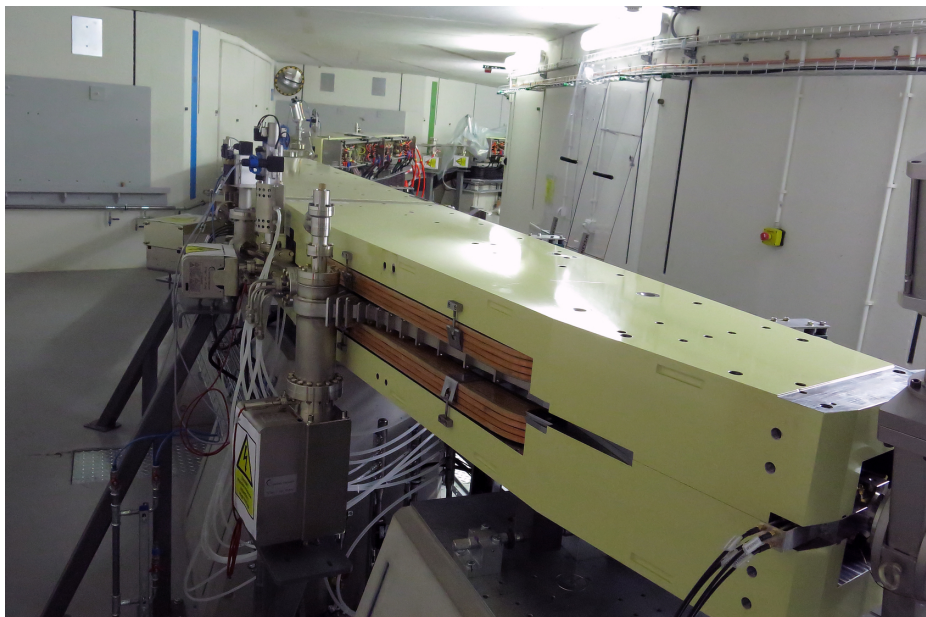


Figure 1.1: A storage ring magnet in National Synchrotron Radiation Centre “Solaris” in Krakow. (source [1])

This infrastructure has to be reliable and provide uninterruptible services to its users. It is the hardware operators and control systems engineers who are responsible for maintaining the hardware and ensuring it remains operational.

In recent days, computer software is used by operators to access and control the devices. Many systems aiming to facilitate hardware control and maintenance have been developed [2]. A common name for these systems is SCADA, which stands for *Supervisory Control And Data Acquisition* [3], [4].

1.1.1. TANGO Control System

Rapidly growing number of hardware to control, especially in natural sciences, raised a need for generic frameworks for building SCADA systems. ESRF synchrotron radiation facility addressed this need by creating the TANGO Control System [5].

During the years of development, it has been also widely adopted in the automation industry and gained popularity in the community, who contribute a lot of tools and utilities related to the TANGO project. The core TANGO is a free and open source software, released under GPLv3 license.

TANGO represents each piece of hardware as a *device server*, an abstract object that can be accessed from client applications over the network. These graphical client applications allow operators to control the hardware. TANGO supports creating client applications in **Java**, **Python** and **C++**. An overview of TANGO Control System architecture and discussion of TANGO GUI frameworks is provided later on in Chapter 2.

1.1.2. Web-based approach

Whereas it is easy to create a desktop client application, the ***web-based approach to TANGO GUI development is still an unexplored area***. This is mainly because TANGO, as a CORBA-based technology, requires access to the TCP/IP stack, which is not the case for the web browsers.

Trends in GUI development. The use of Web frontend technologies for GUI development for both mobile and desktop is gaining more and more popularity. Compared with the traditional approach of writing a native, dedicated application for each platform, it brings many benefits, including:

- the deployment process is simplified, especially in case of browser applications;
- multiple programming styles are supported, including functional and object-oriented;
- there is a wide choice of tools like transpilers, build tools, editors and other utilities;
- lots of open source frameworks, libraries and packages are available;
- portability, thanks to the projects like Electron [6] or Apache Cordova [7].

There are also disadvantages of using web technologies [8], like performance degradation and limitations in UI, but this are the costs of increased productivity and maintainability.

Server-side processing. In Thesis we focus only on frontend solutions that run solely inside a web browser, and any server side processing, like in e.g. PHP, is not necessary. The *old* approach, where web pages were generated on the server and returned in a HTTP response, is always paired with some Javascript whenever interactivity is required. With the emergence of *single-page web applications* [9], where each piece of UI is updated independently using AJAX calls or a similar technology, the number of use cases for the server-side approach decreases significantly.

1.1.3. Adaptive user interfaces

An Adaptive user interface, or AUI, is a user interface that both adapts, or changes its layout depending on the context where it is used and also allows the user to perform this adaptation manually. Providing the adaptive interface is a key part of maximizing the *user experience*.

Automatic adaptation. The interface can adapt its layout to the surrounding environment [10]. For instance, parts of a widget may be shown or hidden and the widgets may be reordered when displayed, e.g., on a mobile device. This is critical part behind the success of *mobile web* approach to web development [11]. Automatic adaptation may be easily achieved thanks to web technologies like *media queries* [12], but is often unsupported in desktop solutions. This also applies to the existing GUI frameworks for TANGO.

Manual adaptation. The second form of adaptive user interface is an interface that may be dynamically changed by the user according to their needs. This allows for creating personalized views for different users and different situations [13]. This approach is supported in TANGO world by the Taurus framework [14], which offers tools for manipulating the UI at runtime.

1.2. Goals of Thesis

The web-based approach to building user interfaces gains more and more popularity. However, it has not been widely adopted among the TANGO Control System client applications and GUI frameworks. Thesis goals may be formulated as follows:

- **evaluate the existing solutions (if any) for building TANGO clients for web browsers**; find out why these solutions have not been adopted by the community like their desktop counterparts; identify any pain points;
- consider the extending and updating of the existing solutions if possible, or **design and develop a new one**, reusing the good parts;
- if a new solution is delivered, implementation shall follow the principles:

- use modern, standardized web technologies; it should not be tied to any particular web framework and should have a limited number of dependencies;
- focus on *user experience* and adaptivity, both automatic and manual;
- be widget based, providing at least basic widgets known from other frameworks, which are recognized and appreciated by the users;
- be flexible and modular, allowing users to create their own widgets and extend the system functionality via plugins;
- offer an application, where end-users can build and adapt the GUI at runtime.

Chapter 2

Overview of TANGO Control System

This chapter provides an overview of the TANGO Control System architecture. Later, the existing frameworks for creating graphical TANGO applications are presented.

2.1. Visualization and control functionality in TANGO Control System

TANGO Controls is a distributed system built on top of CORBA [15]–[17] and ZeroMQ [18]. It introduces a concept of *device server* to represent a physical piece of hardware. This device server is available as a *remote object* that implements a well defined interface. TANGO is object-oriented middleware, and each device server is characterized by:

- *name* - a unique string in the form of a **group/family/member**;
- *attributes* - a set of data fields that may be writable or read only, e.g a current of a power supply;
- *commands* - a set of actions that the device can perform, e.g. a reset action;
- *properties* - a set of parameters that are not reflected in the hardware, but essential to the device server's implementation.

Device servers are distributed by vendors with the hardware or they are created by the community. The internal implementation, especially how the device server connects to the hardware, is out of TANGO's scope.

Another important part of TANGO is the database, where the TANGO schema is stored. This schema keeps information of all the available devices registered in the system. Client applications use the database for device server discovery. The database also stores the properties of each device. Device servers and clients do not access the database directly. Instead, the database is exposed as a device server, called *DataBaseds*, and interaction is performed using commands, e.g. there is a command that returns a list of all the registered device servers or a command that exports a new device. A MySQL [19] or MariaDB [20] server is required to run TANGO. It is also possible to run TANGO without a database

and with a limited functionality. In cases where high availability is required TANGO can run in a multi-database configuration ([21], p. 186).

The hardware is controlled by human operators. They use **graphical client applications**, that connect to the device servers. Typical tasks include manipulating device's attributes, observing the status of the hardware and collecting logs. Client applications are often dedicated to the hardware they operate on, but there are also applications where the operator may adapt the interface by selecting which attributes of which devices he/she wants to control. The applications that allow one to control multiple attributes and give an overview of a set of devices and attributes are called *synoptic panels*.

While relying heavily on CORBA, TANGO also uses ZeroMQ for **event-based communication**. There are events that the client registers for, and is notified whenever the event occurs, e.g. the value of an attribute changes significantly. This allows for efficient communication in scenarios where otherwise the client would constantly poll the device server. Device server developers may also fire events when server-initiated communication is required.

The concepts presented in this section are just an overview to give the reader a glimpse of what TANGO is. Providing an extensive description of TANGO is out of scope of this thesis and some simplifications have been made here. All these topics are discussed much deeper in the TANGO Control System Manual [21].

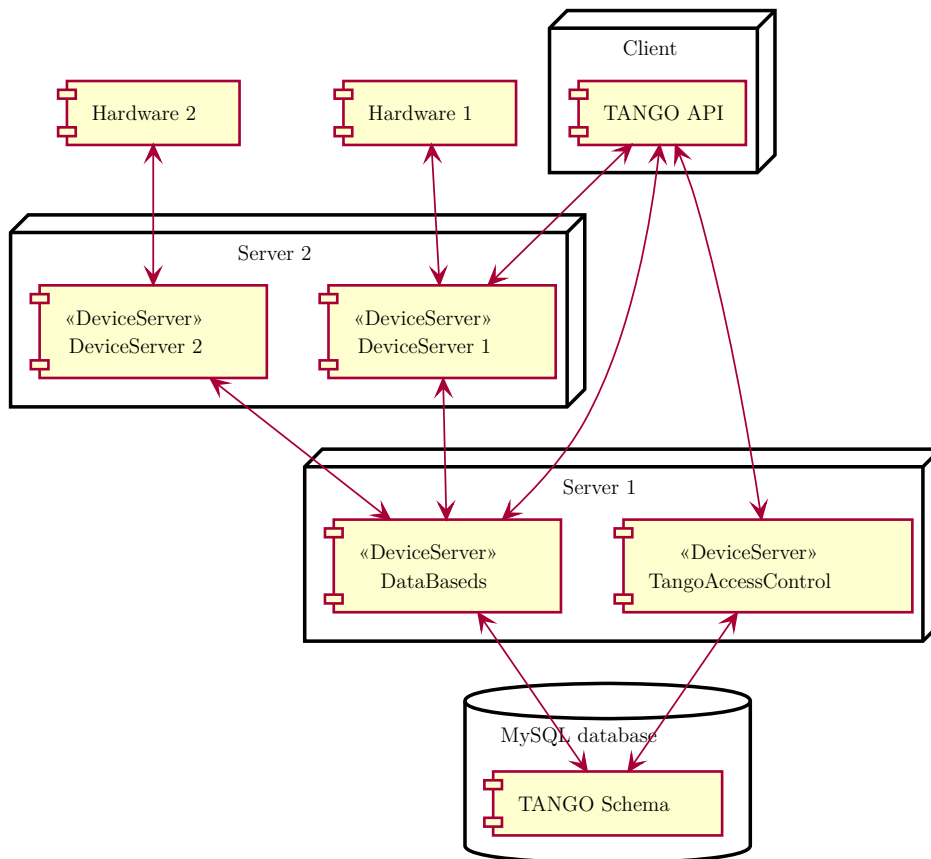


Figure 2.1: TANGO Control System architecture overview.

Architecture overview. TANGO deployment usually spans over several machines, connected in a network. There are clients and servers. *Clients* are just terminals that allow operators to interact with the hardware. *Server* machines are responsible for accessing the hardware, and they are the place where *device servers* run. A single machine can host any number of device server instances. The details of communication are hidden behind the CORBA and TANGO abstractions. Client application developer may treat the system just as a pool of device servers.

There are some special device servers that do not control any hardware. These are the above mentioned database device server, the *DataBaseds*, and the authorization service, the *TangoAccessControl* device server. This *TangoAccessControl* device server is responsible for user authorization and offers fine-grained permission control, at a single attribute level. TANGO's architecture is depicted in Fig. 2.1.

2.2. TANGO GUI frameworks

The TANGO has been designed to allow uniform access to the hardware resources. The end-users of TANGO-based software are hardware operators who are responsible for controlling the hardware during an experiment. They need reliable and convenient graphical client applications to do their job effectively.

The TANGO API offers abstractions like *DeviceProxy* or *AttributeProxy* that allow accessing devices programmatically. Using these proxies, a client application may be built using any technology and language where TANGO is available, including **C++**, **Java** and **Python**. Most of the client applications share common goals and requirements. They also use some common patterns to fulfill these goals. This raised the need for GUI standardization and development of universal frameworks that will speed up TANGO client applications development. The two leading solutions are ATK and Taurus, discussed below.

ATK. The *Tango Application Toolkit* [22], ATK, has been developed to address the need for consistent, easy to develop TANGO GUI applications. It provides universal *viewers*, for attributes and commands. These viewers are called widgets nowadays. ATK has been implemented with Java and Swing, which makes it portable in desktop environments.

ATK uses Model-View-Controller architecture, and is internally divided into two parts, the *ATKCore*, which offers APIs to access the *models* (e.g. devices, attributes or commands), and *ATKWidget*, which is a set of Swing-based viewers. There is also the third component, a graphical editor called *JDraw*. It provides a way to interactively build a schema of the system and then attach TANGO models, like commands or attributes, to it. This schema can be stored in a file and rendered at runtime by ATK as a *synoptic panel*. The *JDraw* interface is presented in Fig. 2.2.

ATK became a standard solution for building graphical TANGO clients and multiple applications, like *AtkPanel* or *AtkMoni*, has been built using this framework.

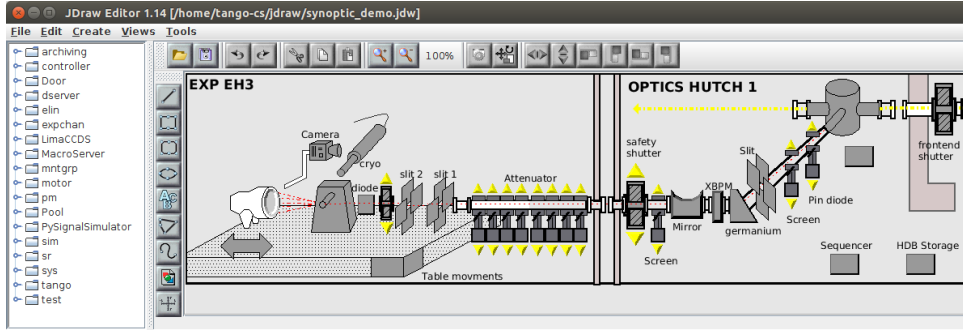


Figure 2.2: *JDraw* synoptic panel editor.

Taurus. The Taurus framework [14] has been developed with the same concepts in mind as the ATK, but targets Python environments and uses Qt library for GUI rendering. It shares with the ATK some common concepts and design principles. It is divided into two packages, *taurus.core* and *taurus.qt*. The former one brings *TaurusModel* object, which represents an entity from TANGO world, e.g. device or attribute. The latter one is a set of widgets that behave like standard Qt widgets, but each is bound to possibly multiple *TaurusModels*.

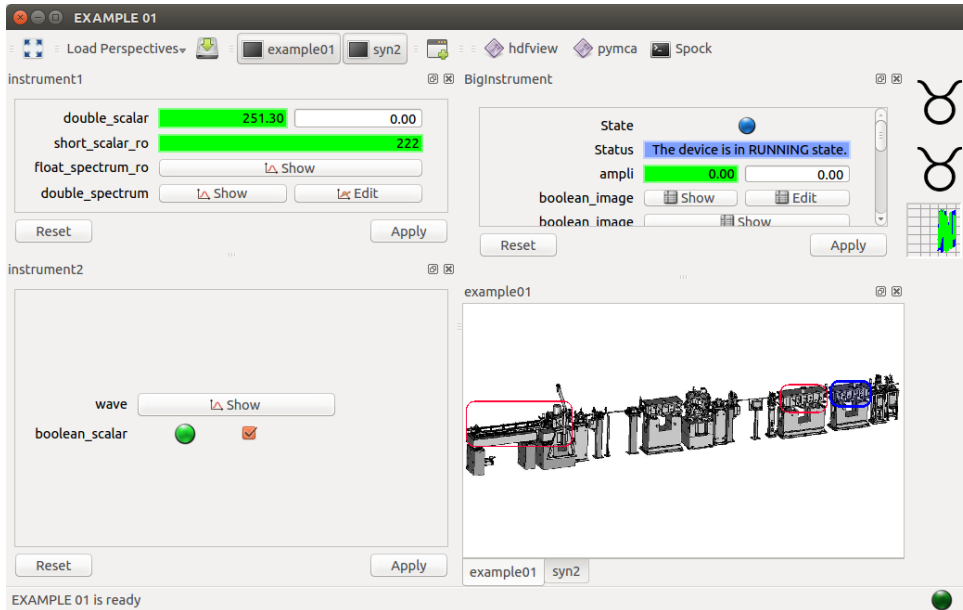


Figure 2.3: Taurus GUI.

Taurus applications can be created using any tools suitable for creating Python/Qt applications. However, Taurus ships with an interactive GUI builder, the *TaurusGUI* application. The users can put any widgets on a set of panels, without writing a single line of code. This GUI may be later adjusted at runtime. This application is shown in Fig. 2.3.

The Taurus has been widely adopted by the community and is currently more popular than ATK, since Python is the leading platform in science, where TANGO is mainly used.

Chapter 3

State-of-the-art

There have been a few projects aiming to enable development of web-based TANGO client applications. Most of them have been abandoned at early proof-of-concept stage. This chapter presents the existing solutions and evaluates different approaches to web-enabled TANGO clients.

3.1. Canone

Canone [23] allows for creating customizable, web-based TANGO panels, composed from various widgets. The widgets can be arranged on panels and configured according to operator's requirements. This is similar to the Taurus approach. Canone also allows for creating user accounts which are stored in a database, and provides fine-grained authorization for these accounts with permissions at a device level. An example application built with Canone is presented in Fig. 3.1.

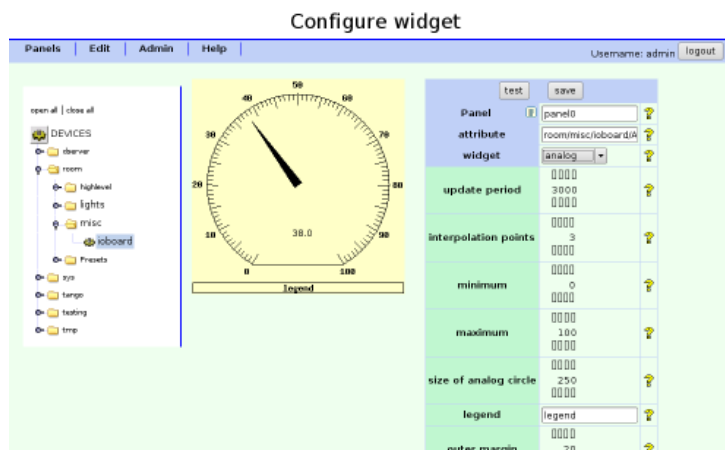


Figure 3.1: Canone Web Interface. (source [24])

Canone development started in 2005. The last release that brought new features was in May 2007. Since then, there was a bugfix release in 2011. The project is no longer maintained and has never been widely adopted by the TANGO community.

Architecture. Canone is divided into two parts. There is a server part called *socket server*, written in Python, and a frontend part, the Canone Web application, written in PHP. **The *socket server* acts like a proxy** between the existing TANGO infrastructure and the frontend part. The connection between these layers is handled over a standard TCP socket. Canone requires to be deployed on PHP server, e.g. Apache Httpd. Also, a separate database server is required for storing Canone configuration. The architecture is depicted in Fig. 3.2.

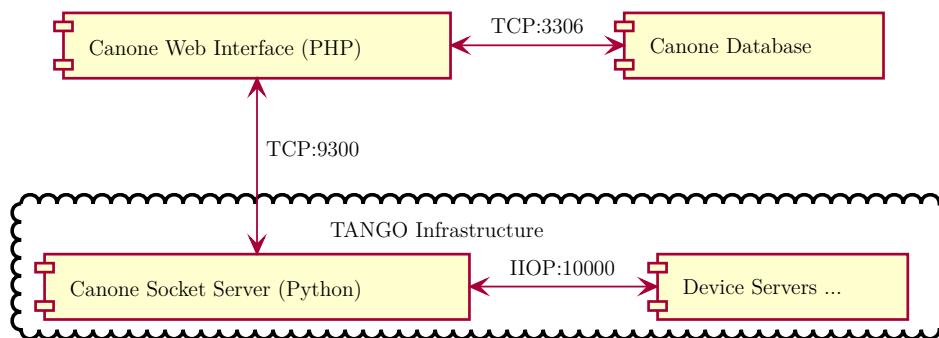


Figure 3.2: Canone architecture.

User experience. Being unmaintained for almost 10 years, Canone shows its age and may be considered a legacy software today. This is reflected in its limited interaction options and the poor overall user experience. The actions performed by the user are most of the time submitted to the PHP server, which next forwards the calls to the *socket server*. There is a limited support for AJAX in some widgets. These two layers of indirection and incorporation of old technologies affect user experience and makes Canone unsuitable for today's world of dynamic web applications.

Technological aspects. Canone generates UI on the server side, during HTTP request processing. However, there is no model-view- * framework or even templating engine used. All HTML code is printed to the response stream. This makes the code extremely unreadable and unmaintainable. When it comes to the frontend part, there is a minimal amount of CSS, but tables are used for maintaining the page layout. All these factors make it impossible to extend or reuse any parts of this 10-year-old project.

All Canone drawbacks described above were acceptable 10 years ago. This is a very important project and a first step in moving TANGO client applications to the browser. It is also the first project that introduced the concept of *proxy server* that connects CORBA-based world and that of the web browser.

3.2. Taurus Web

Taurus Web [25] is part of the Taurus project. Taurus Web provides access to TANGO devices from a web browser. The user interface is very limited and there are no customizable widgets. The source code is distributed with several demo applications, where just a few basic fields are displayed. The available interaction options are very limited. The example GUI panel is presented in Fig. 3.3.

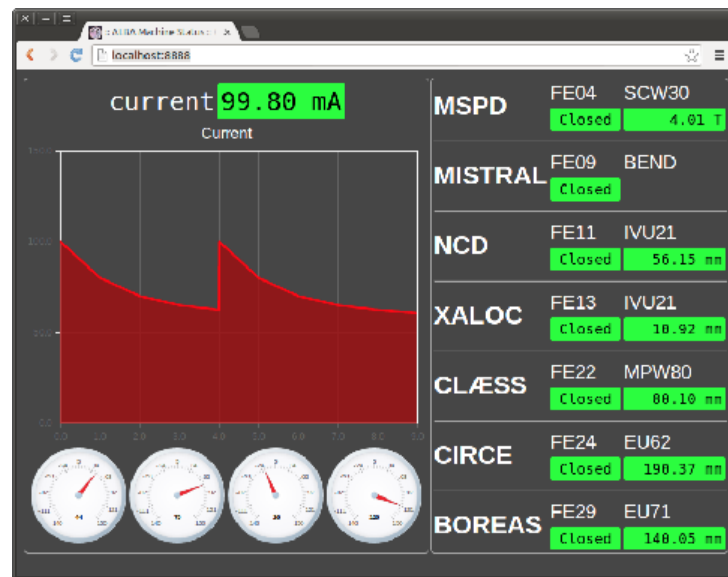


Figure 3.3: Taurus Web Interface. (source [25])

Architecture. The architecture of the Taurus Web is, like in Canone, divided into a server part and a frontend part. The server is written in Python and links the browser-based GUI with TANGO using WebSocket connection. This provides a full duplex link that may be used for robust integration of TANGO’s event system with browser. The frontend part requires a WebSocket-enabled browser, but this is not a problem today.

Technological aspects. According to its creators, Taurus Web is meant to be a framework that can be used to create a dedicated GUI applications rather than being a generic standalone application, like Canone. The backend part uses Tornado web server for handling WebSocket connection. The frontend part is written in Javascript, HTML5 and CSS3. The authors claim that it is ready for integration with modern web technologies like Angular, jQuery, Node.js and Dojo framework.

Taurus Web development has started in 2011 and the project was abandoned at the early proof of concept stage. There are just a few hundred lines of server code and **nearby no frontend code**.

3.3. Tango REST

Tango REST [26] is a RESTful interface for TANGO. The goal of this project was to expose a subset of TANGO APIs over a HTTP protocol. Tango REST acts as a proxy between *device servers* and a RESTful client, which may run in e.g. a web browser. The system can be configured to **authenticate users against an LDAP database**. The project also includes a mobile application for Android devices, which provides a Jive-like functionality using Tango REST server. Since this application is a native Java application and cannot run in a web browser, it is out of scope of this discussion.

Technological aspects. Tango REST is implemented as a Java EE servlet. It can be deployed on any application server or on a servlet container. It uses the standard JAX-RS API to provide a RESTful endpoint.

Tango REST, on its own, is not useful for building web user interfaces. However, being paired with a frontend client, **it can be used as a backend layer** for accessing TANGO API from a web browser.

3.4. GoTan

GoTan [27] is a complete ecosystem for accessing TANGO via RESTful interface over HTTP protocol. The project consists of multiple modules. However, it has no web frontend layer. GoTan has been written in Java and Groovy. The API is well documented, which makes it easy to write own client for GoTan server. Example minimal client applications are available for Android phones and iPhone. These are native clients, written in Java and Objective-C, respectively. GoTan does not support user accounts or permissions.

Technological aspects. GoTan server is a standalone application that exposes a RESTful API using the Restlet framework. It is written mostly in Groovy, with some parts written in Java. It is a generic solution, capable of accessing classes and servers defined in TANGO database as well as using the attributes, commands and properties of any device.

The project is not actively developed and has been discontinued in 2013, reaching only a proof-of-concept stage. The project can be integrated with a third party frontend layer to create a complete web-based TANGO client.

3.5. mTango

mTango [28] is a complete solution for building TANGO client applications for **web browsers** and **mobile devices**. It consists of multiple components, which can be used together to create flexible web-based TANGO GUIs. The key part of mTango is a REST server which allows to access TANGO *device servers* using HTTP protocol and a well

defined API. The UI part ships with a set of widgets useful in building client applications. mTango, as a Java-based web application, allows for **flexible security configuration, including authentication with LDAP**.

Architecture. The mTango architecture consists of a few loosely coupled components. There is a server part, called *mTangoREST server*, which interacts directly with TANGO and exposes its APIs via a RESTful gateway. The clients connect directly to this server via HTTP protocol. mTango offers two types of clients. The first one is called *mTangoREST client* and is written in Java. The second client, called *jsTangORB* is written in Javascript and runs in web browsers. On top of this browser-based client there is a UI layer, called *mTangoUI*. The architecture is depicted in Fig. 3.4.

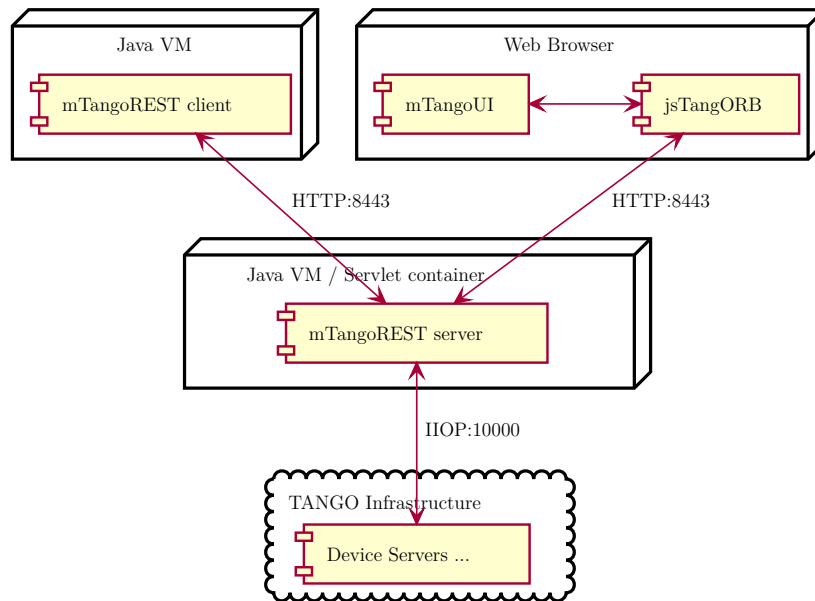


Figure 3.4: mTango architecture.

mTangoREST server. The server part is implemented in Java as a standard Java EE servlet. It uses JBoss RESTEasy to provide a RESTful web service. The server is distributed as a **web archive** package, or as a standalone application with embedded Apache Tomcat servlet container. User authentication can be performed in the filter chain during request processing. mTango integrates with TangoAccessControl to provide authorization. The server performs some optimizations, like caching. TANGO events are supported, thanks to the Comet model [29].

mTangoREST client. This client is a Java library that allows applications which run on JVM to communicate with *mTangoREST server*. With full access to TCP/IP stack, Java applications can use TANGO APIs directly, but in some cases using mTango may be required, e.g. in a network where all traffic but HTTP is blocked.

jsTangORB. The Javascript client allows for accessing *mTangoREST server* from a web browser. It uses **JavascriptMVC framework** to model abstractions like *Request* and *Response* or TANGO-specific proxies, e.g. *DeviceProxy*. It uses **JSONP** to overcome the same-origin policy restrictions which prevent clients from accessing servers on different

hosts. The API is callback-based which makes it difficult to write clean and maintainable code (problem known as the *callback hell*). mTango performs client-side caching and other optimizations.

mTangoUI. mTangoUI is a **JavascriptMVC application** built on top of *jsTangORB*. It provides abstractions like *Page*, and widgets like *DeviceAttribute*, *DeviceCommand* or *Plot*. One has to use the mTango CLI tools (which are wrappers for the JavascriptMVC CLI tools) to create a minimal application template. Then, widgets can be created declaratively, by placing an `mtango:attr` tag on a webpage. The `mtango:attr` tag can be customized by using a *view* attribute that controls widget's appearance. Two views are available: a text field and a list, both in read-only and writable variants. The user may also implement his/his own view. The `mtango:img` and `mtango:plot` tags can be used to create an image or a plot.

The mTango project is under active development since 2013, with the last release dating to August 2016. The frontend part has been updated in 2015. The server side part is very flexible and may be extended with custom filters when deployed on a standalone servlet container. In 2016 the standardization of the official TANGO REST API was started by the community, and mTango has been chosen as a reference implementation. **This is the most complete and feature rich RESTful interface to TANGO.**

On the other side, the frontend part, both *jsTangORB* and *mTangoUI* have some drawbacks. They are dependent on an old version of JavascriptMVC, a framework that never gained attention and has not been widely adopted by the web developers community. This is especially controversial in case of *jsTangORB*, as a non-UI library, to depend on a model-view-controller framework. Also, the **Rhino runtime** (and thus Java) is required for development. Today, the Node.js is a *de-facto-standard* in Javascript development and package management. Another drawback is that *jsTangORB* uses JSONP for handling cross origin requests. The canonical way for doing this is to implement the CORS protocol. The *jsTangORB* API has been designed to use callback functions for asynchronous code. This may lead to writing unmaintainable and unreadable code [30]. A solution for the *callback hell* problem is to use standard *Promise* API, or a polyfill library when older browsers have to be supported. Taking into consideration all these factors, **building lightweight, modern, standards-driven user interfaces for a web browser using jsTangoORB and mTangoUI may be a challenging task.**

Due to the incorporation of an unpopular, third party frontend framework and CLI tools, mTango has a steep learning curve that may slow developers down. It is hard to spin up a simple application if the user is not familiar with JavascriptMVC.

3.6. Summary

Tbl. 3.1 summarizes the solutions discussed above. The main focus is put on user experience, flexibility, extensibility. The technological aspects and software architecture are important

factors as well, since they affect the way the software can be extended (e.g. with new widgets).

Table 3.1: Comparison of existing solutions.

Feature	Canone	Taurus Web	TangoREST	GoTan	mTango
Has frontend layer	yes, (PHP)	yes, (Javascript)	no, (native Android)	no (native Android and iOS)	yes, (Javascript)
Has widgets	yes, (basic AJAX)	no	N/A	N/A	yes, (just a few views)
Has interactive <i>synoptic panel</i>	yes	N/A	N/A	N/A	no ¹
Supports user accounts	yes, (database)	no	yes, (multiple options)	no	yes, (multiple options)
Supports TANGO events	N/A	no	no	no	yes, (Comet model)
Backend server technology	PHP, Python	Python, Tornado, WebSocket	Java, JAX-RS, HTTP	Groovy, Restlet, HTTP	Java, RESTEasy, HTTP
Released	2005	2011	2015	2012	2013
Is actively developed	no	no	no	no	yes

The solutions discussed in this chapter use different techniques to access TANGO infrastructure from within a web browser. Most of them are no longer maintained and have been abandoned years ago. Only mTango project is actively developed and its user base is constantly growing making mTango the leading technology for integrating TANGO with the browser. The server part is flexible, configurable and well-performing. However, the evaluation has showed that mTango frontend layer (`jsTangORB` and `mTangoUI` modules)

¹In August 2016 the development of a mTango-based *Tango Webapp* has been announced.

has some drawbacks and issues that have to be addressed before it may become the ultimate solution for building web-based TANGO client applications.

Chapter 4

Solution and Implementation

Each of the existing solutions presented in Chapter 3 has some drawbacks. There is no *best choice* that will suit everyone's needs. **To address these issues and fulfill the goals set up in Chapter 1, we have developed a project called TangoJS.**

This chapter describes the implemented solution in terms of the formulated design goals and provides a high level overview of the system architecture, including all related software layers.

4.1. Introduction to TangoJS

TangoJS allows building TANGO client applications with standard web front-end technologies like HTML, CSS and Javascript. It gives TANGO developers a complete set of tools and APIs required for this task. There is a minimal set of dependencies required.

TangoJS has been designed to be a modular ecosystem - one includes only the modules one needs and configures everything according to the project requirements. There are three main separate layers, which are connected via well-defined interfaces.

- **TangoJS Core** - Javascript API for programmatic interactions with TANGO from a web browser, partly generated from TANGO IDL;
- **TangoJS Connector** - interface that abstracts-out communication with TANGO infrastructure via pluggable backend servers;
- **TangoJS WebComponents** - an extensible widget toolkit for rapid GUI applications development, inspired by Taurus.

All these components are described in details later on in this chapter.

4.2. Design Goals

Apart from the general goals formulated in Chapter 1, a set of design goals has been established before the development of TangoJS has started. These goals aim to meet the challenges in the areas where the existing solutions have failed. The goals are mostly related to the technological aspects of implementation.

Compliance with the latest web standards. The Web started to evolve faster and faster in recent years. A lot of applications has been migrated to the browser. This includes both desktop mobile applications. The developers are willing to write their software using web technologies because this approach allows them to be more productive and target a wider group of potential users. This trend forces standardization bodies, like W3C, and browser vendors to speed up their development cycle. The new standard of ECMAScript is released once a year and quickly becomes supported by major web browsers. The HTML standard and CSS modules are also constantly evolving, including more and more features. These technologies are attractive for developers because they have to write less code, which is also cleaner and more maintainable. They can put more focus on their business goals and deliver a high quality project on time. The developer will more likely select a framework that is being kept up-to-date with the latest web standards.

Extensibility. Today's libraries are easy to extend when necessary. Developers often choose a modular, pluggable architecture for their frameworks and libraries. This brings significant benefits for both library authors and users. A modular project is easier to maintain, especially when it becomes larger than a few files of a proof of concept. Dividing the library into modules can bring better separation of concerns, at a higher level of abstraction. Also, library users can benefit from a pluggable project. They can include only the required parts in their projects. This allows for reducing the dependencies, which are a crucial aspect of web development. When a single module does not meet user's requirements, he/she can develop their own one, and use it like a plugin. In case of TangoJS, this may be developing a *Connector* for a new backend server or creating a new widget that will integrate neatly with the rest of *TangoJS Web Components* modules.

Using well-proven concepts. Some of the existing web-based TANGO solutions provide configurable widgets that can be used as building blocks for larger graphical client applications, like *synoptic panels*. The leading solution for desktop TANGO clients, the Taurus framework, offers a broad collection of widgets. These widgets can be used programmatically or can interactively be put on panels via a simple drag-and-drop. One of the goals in TangoJS development was to deliver a set of widgets most commonly used in Taurus, like *label* (which visualizes single attribute), *line edit* (which is a writable label) or *trend* (which visualizes a set of attributes in time domain). The widgets shall be accessible programmatically from within Javascript, declaratively from an HTML page and interactively, using a dedicated application for rapid GUI development. In all ways, the user should be able to tweak the widget's appearance, e.g. hiding some optional parts. It should also be possible to change some parameters related to the internal details, e.g. the polling period.

Reuse of what works well. In the software development it is crucial to avoid *reinventing the wheel*. The evaluation of the existing solutions shows that there are usually at least two layers - the backend and the frontend. The frontend part directly impacts user experience. The backend is never exposed to the end user nor to the GUI application developer. Instead, it is the frontend layer that is responsible for interactions with the backend. The most complete solution for TANGO and browser integration is mTango. The mTango does its job very well on the server side. However, its frontend part has some drawbacks, which are, due to the design decisions, impossible to be overcome without rewriting it from a scratch. **TangoJS will use *mTango* as a default backend**, leaving developers the option to easily replace it with another backend of their choice.

Being future-proof. When starting a new software project, one should focus not only on the current requirements, but think about the project's future. It is important to ensure that the project ages slowly. When unmaintained or unpopular third party solutions are incorporated, it may soon turn out that the project uses deprecated software. It may be impossible to remove these dependencies later and the project will be destined to be abandoned. The goal in TangoJS development is to **minimize dependencies on third party code** and use the **latest web standards**, like HTML5, CSS Level 3 modules or ECMAScript 2016. The standard solutions rarely become deprecated. Instead, they evolve gradually, which makes easy to keep the project up-to-date.

Keeping it simple to start with. One of project goals was to minimize the learning curve. TangoJS Core API brings familiar TANGO abstractions, like *DeviceProxy* to the browser ecosystem. The widget collection has been inspired by the Taurus framework - the leading solution for building TANGO clients in Python/Qt. It was designed with the ease-of-use and ease-of-deployment in mind. Only basic knowledge of web-development and Node.js [31] is required to get started.

Considering of security aspects. Each application, especially one that involves network communication, has to secure sensible data against unwanted access. In case of TangoJS this is the job for the *Connector* and the corresponding backend server. For instance, mTango server may use the roles defined in the servlet container. It also integrates with TangoAccessControl. The authentication in the server is performed using HTTP Basic Auth, which is a simple and secure way of providing user credentials. The link between the *Connector* and the server may be secured by using **encrypted HTTPS protocol**.

4.3. Design Decisions

Also, some more technical design decisions have been made regarding the language of choice and development platform.

Choice of the best language. Since TangoJS aims to allow for building TANGO clients that run in web browsers, it immediately becomes obvious that the whole thing is going to be built with Javascript (and with HTML, when it comes to the presentation

layer). Javascript is the core language of the Web. Nowadays there are tons of languages that can be compiled to Javascript, but we have chosen pure Javascript to power TangoJS. Although each language offers its unique features, they come with the cost of introducing additional an buildstep for the compilation phase, maintaining sourcemaps, etc. Of course, every web project today requires a dedicated build process - at least for concatenating and minifying the code. However, this may not be true in the near future, due to the emerging support for HTTP/2, which is fully multiplexed and can handle parallel transfers over single a TCP connection.

TangoJS has been written using the latest standard of Javascript language, called ECMAScript 2015. It brings a lot of new features and goodness known from e.g. CoffeeScript. It also works in all modern browsers without a need for transcompilation to ECMAScript 5. The ECMAScript 2015 and its role in TangoJS is discussed in Chapter 5 and Appendix D.

Select a popular platform. We have chosen Node.js as both the development and target platform for TangoJS. In recent days it has become the most popular solution for building Javascript applications, both on the server and client side. It is not just an ordinary Javascript runtime, but a whole ecosystem, with dependency management, application packaging, support for complex build processes and development workflows. All components are available in the npm Registry [32] - the *de-facto standard* in distributing Javascript dependencies, not only for Node.js, but also for web browsers.

4.4. Architecture of TangoJS

The architecture of TangoJS is layered, where a next layer builds on the previous one. Application developer will interact mostly with the topmost layer, the *TangoJS WebComponents* module. When programmatic access to TANGO API is required, one can use *TangoJS Core*. *TangoJS Connector* implementations are used only by the core layer and are not exposed directly to the developer. The backend server is not part of TangoJS. Any backend can be used to access TANGO infrastructure, provided that a dedicated *Connector* is available. Currently there are two connectors to choose from: an in-memory *mock connector* and a **mTango connector** which allows to use mTango as a backend for TangoJS. The architecture of TangoJS is depicted in Fig. 4.1. Below we briefly discuss each layer.

TANGO Infrastructure. The whole TangoJS stack sits on top of the existing TANGO infrastructure. Since CORBA requires access to the complete TCP/IP stack, TANGO cannot be accessed directly from the web browser. Some sort of proxy software is required here. TangoJS addresses this issue by introducing the *Connector* concept.

TangoJS Connector. A Connector is a bridge between TANGO and TangoJS. There are two separate components in this layer: a server-side part and a client-side part. TangoJS specifies only the interface of the client-side part. How the client communicates with the server is an implementation detail and depends on the used backend.

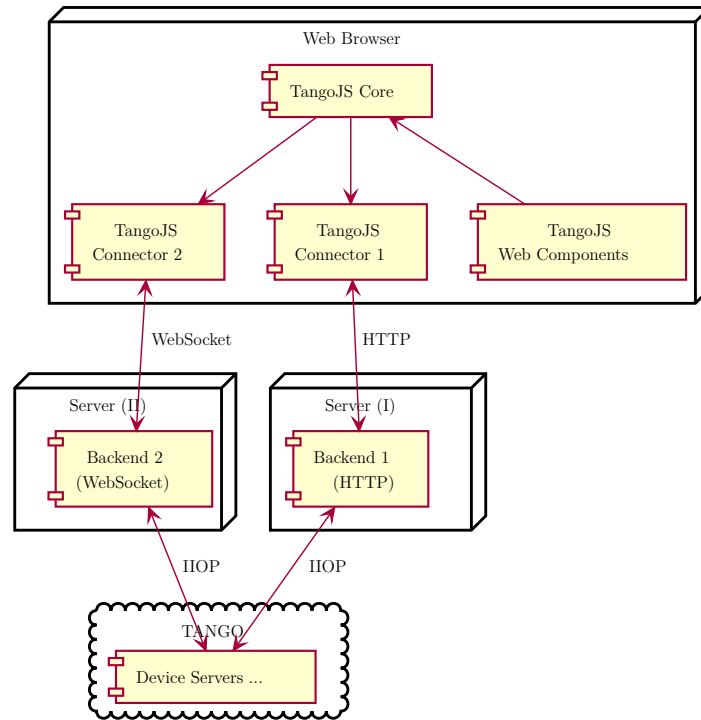


Figure 4.1: An overview of TangoJS high-level architecture.

TangoJS Core. This is a package that brings all the TANGO datatypes, structures, enums and interfaces to the Javascript world. It has been partly generated directly from TANGO IDL. The main goal was to maintain consistency with TANGO Java API, and provide the same set of abstractions with identical interfaces. It may be used from both Node.js and web browser.

TangoJS WebComponents. This is the largest part of the TangoJS stack. It is a collection of standalone widgets, which may be included in any web application. No third-party framework is required. The library offers widgets similar to Taurus core widgets, but provides means for developing new components.

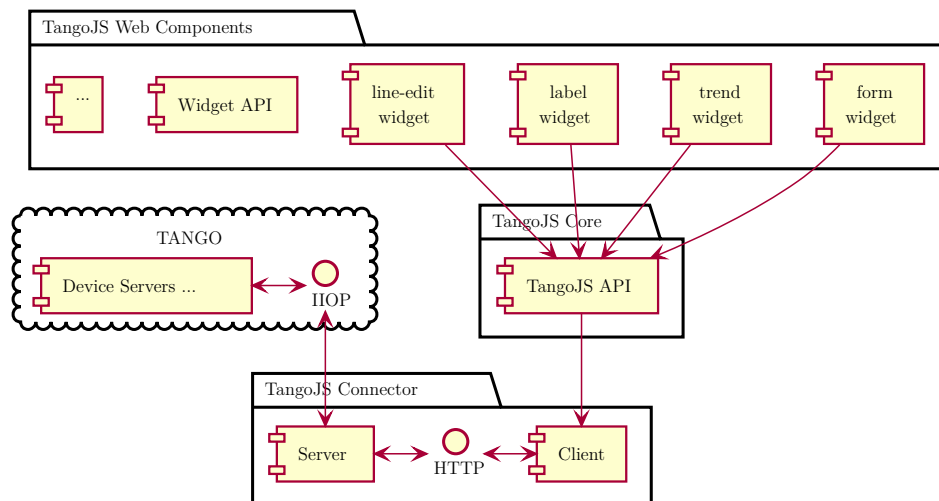


Figure 4.2: TangoJS component-level architecture.

A more detailed view on TangoJS architecture is shown in Fig. 4.2. The topmost module, *TangoJS WebComponents*, offers a set of widgets and an API for creating own widgets. Application developer also have access to *TangoJS Core API*, when direct access to raw TANGO proxies is required. This API forwards all calls to the *Connector* layer, which is then responsible for making the requests to the backend and collecting the responses.

Any kind of web application can be built upon the layers described above. As a proof of concept, **a synoptic panel application has been developed**. This application is described later in this chapter. The TangoJS layers are covered in details in the following sections.

4.5. TangoJS Core - the TANGO API for browsers

The main part of the TangoJS, the *TangoJS Core*, is a library for programmatic access to TANGO APIs. This library **gives the user access to concepts like *DeviceProxy***, a client side representation of a *device server*, typically used in client applications in TANGO world. This module also **contains all the structures, enums, typedefs and interfaces defined by the TANGO IDL**.

General assumptions. The goal of this layer is to provide a well-defined, convenient set of classes that hide the lower layers of the TangoJS stack, like *Connector* or the backend. This is an object-oriented API, similar to jTango, the standard TANGO Java API. This makes the TANGO developers familiar with it. The API is nonblocking, due to the asynchronous nature of Javascript. It extensively uses *Promises*, which means that each method call returns immediately a standard *Promise* object. This promise may be later resolved to a value or may be rejected in error case. The use of promises can save the user from the situations like *callback hell*^[^03-callback-hell].

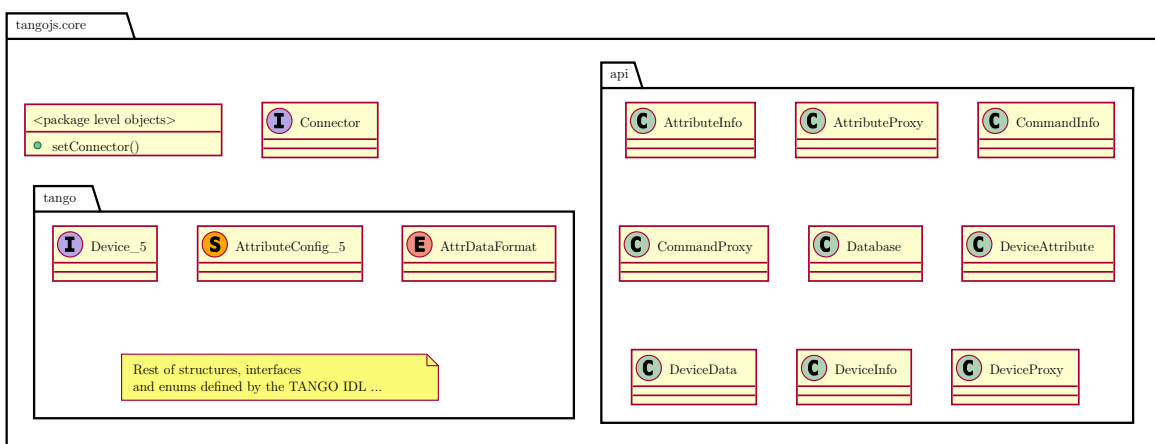


Figure 4.3: Class diagram of *TangoJS Core* package.

Module structure. The *TangoJS Core* is distributed as an UMD module [33]. This makes it easy to load it in different environments, like Node.js or web browser. When loaded in the browser, it is attached to the global object `tangojs.core`. The module is

internally divided into two packages: **api** and **tango**. The **api** package contains TANGO proxies as well as information classes for *Device*, *Attribute* and *Command* entities. The **tango** package has been generated from the TANGO IDL and provides the common TANGO data types. Event-related APIs are unavailable, since **events are not currently supported**. A class diagram of the Core module is presented in Fig. 4.3.

4.6. TangoJS Connector - pluggable backends

The *Connector* concept allows the TangoJS to support multiple backends. A *Connector* is an implementation of the *Connector* interface from the *TangoJS Core* module. The dependencies between the core and the connector are depicted in Fig. 4.4.

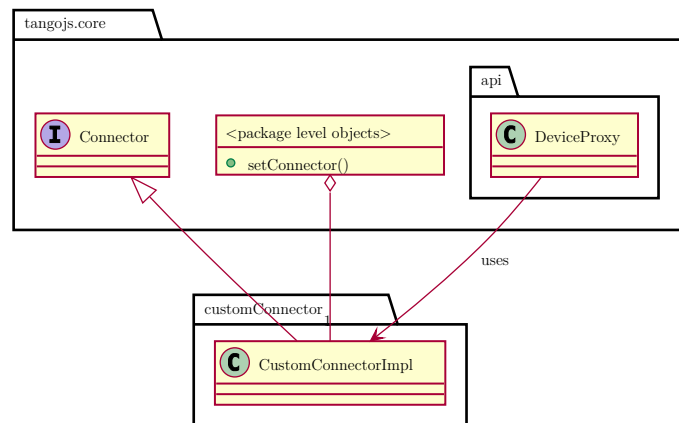


Figure 4.4: Dependency between the *TangoJS Core* and the *Connector*.

A concrete **connector implementation** has to be plugged into **TangoJS**, before it may be used. This process is shown in Lst. 4.1. There is always only one connector active. The upper layer, which is *TangoJS Core*, forwards most calls to this connector and awaits for the results. Since the core knows nothing about the backend used, it cannot perform any caching or optimizations. This has to be handled at the connector level.

Listing 4.1 Connector setup process.

```
// following to be done before using the TangoJS:
const connector = new CustomConnectorImpl( /* configuration */ )
tangojs.core.setConnector(connector)

// now TangoJS will work using this connector
const deviceProxy = new tangojs.core.api.DeviceProxy('sys/tg_test/1')
```

mTango connector. The mTango has been chosen as a default backend for TangoJS. The connector for the *mTangoREST* server is a simple client that consumes a RESTful API exposed by the mTango server. In most cases TangoJS will be deployed on a different server than the mTango. *TangoJS mTango Connector* supports this setup, including **implementation of CORS protocol**. The mTango's servlet container has to be configured

for CORS support. Most containers provide a suitable servlet filter that can be included in the filter chain. mTango also supports user authorization. Since this process is handled completely by the server, there is no concept of user in TangoJS APIs. The only part aware of user identity is the connector. User credentials have to be passed upon the connector instantiation. It's the application developer responsibility to secure these credentials and re-instantiate the connector when the user's identity changes.

In-memory mock connector. TangoJS also offers a mocked connector implementation, which mimics a real TANGO infrastructure with in-memory hierarchy of objects. There is no network communication performed. This approach is useful e.g, for automated testing the upper layers. This connector is flexible and can be configured with a list of *device* objects, where each device exposes some attributes and commands.

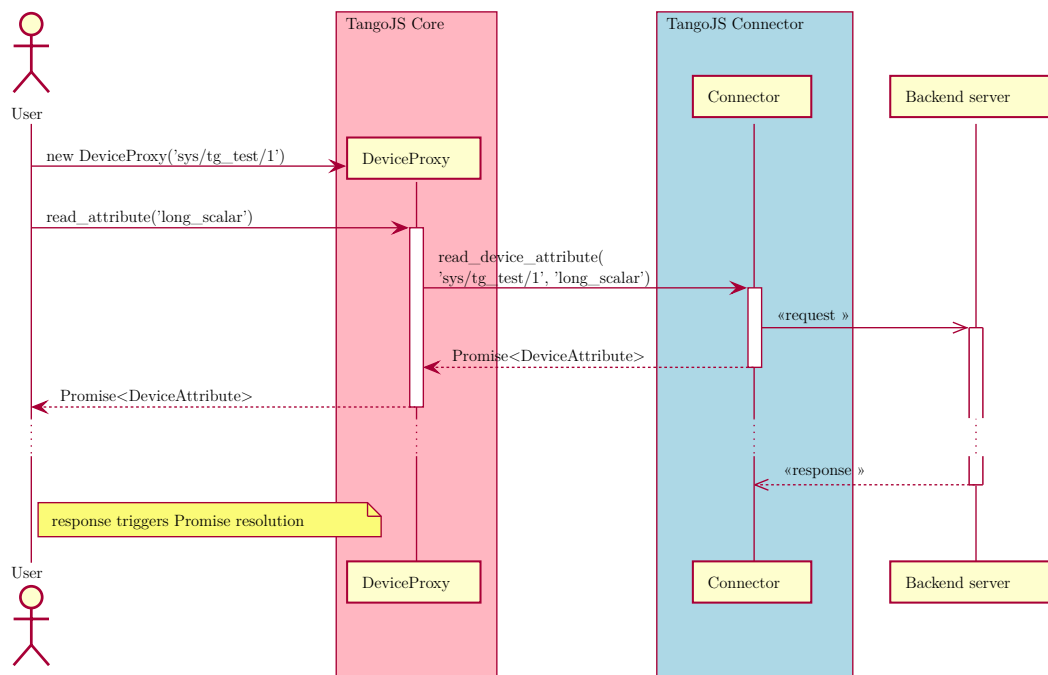


Figure 4.5: Reading the value of an attribute using TangoJS Core API.

4.7. TangoJS WebComponents - HTML widget toolkit

The most important part of TangoJS, from end user's perspective, is the widget toolkit. This module sits on top of the core layer in TangoJS stack. It contains a set of customizable widgets and provides utilities to easily build own widgets.

Module structure. This module is designed for in-browser use only. The widgets are packed as separate HTML files and can be included on demand. The module also exports the `tangojs.web` package, which contains mixins and utility functions useful for creating non-standard widgets. These tools are covered in details in Appendix B.

Widget concept. A widget is a self-contained piece of UI, that may be used on its own and requires zero configuration. The widget is a graphical representation of a *model*.

A model is an abstraction that may be a *device*, a device's *attribute* or a *command*. Two examples are a device status indicator widget and an edit field for a value of an attribute. Some widgets, e.g. a trend widget, can be bound to multiple models. A widget may only interact with its corresponding models. It is isolated from the rest of the TANGO environment and independent from the application that it is embedded in. To achieve these goals, TangoJS widget toolkit module has been developed using the latest set of W3C standards, including *Custom Elements*, *HTML Imports*, *HTML Templates* and *Shadow DOM*. These standards together are called *Web Components*. From the developer point of view, these widgets behave like native web controls, e.g. *input* or *button*.

Using TangoJS widgets. The developer should be able to include the desired widgets in his/her application. He/she may then optionally configure these widgets to match his/her requirements. The widgets can be instantiated and configured in two ways. The first way is via standard DOM manipulation APIs available in Javascript, like the `document.createElement` function. This is shown in Lst. 4.2. Apart from the imperative access, there is also a way to create widgets declaratively, by simply putting a desired tag in HTML markup. This is shown in Lst. 4.3.

Listing 4.2 Imperative widget instantiation from Javascript code.

```
const lineEdit = document.createElement('tangojs-line-edit')


lineEdit.setAttribute('model', 'sys/tg_test/1/long_scalar_w')
lineEdit.pollPeriod = 1000 // attributes have reflected properties
lineEdit.showName = true
lineEdit.showQuality = true
```

Listing 4.3 Declarative widget instantiation from HTML markup.

```
<tangojs-trend
  model="sys/tg_test/1/long_scalar_w,sys/tg_test/1/double_scalar"
  poll-period="1000"
  data-limit="20">
</tangojs-trend>
```

Available widgets. The *TangoJS Web Components* is highly influenced by the Taurus library. The initial goal was to bring the most commonly used widgets to the browser. The widgets offer a layout and appearance similar to their counterparts from Taurus. The available widgets are depicted in Fig. 4.6. Below, each widget is provided with a short description:


- `tangojs-label` - displays the name, value, unit and status of a *read-only* attribute;
- `tangojs-line-edit` - displays the name, value, unit, status and edit box of a *writable* attribute. Depending on an attribute type, the edit box may be a text field, a spinner or a checkbox;
- `tangojs-state-led` - displays the name, state and status of a device;
- `tangojs-command-button` - when pressed, executes a command. Fires a DOM event when result becomes available;

sine_trend	-0.4042164694784...	u	
------------	---------------------	---	---

(a) tangojs-label bound to a scalar attribute.

boolean	false	<input type="checkbox"/>	
---------	-------	--------------------------	---

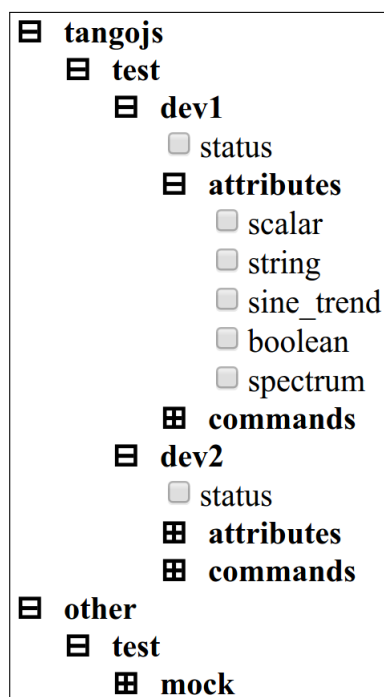
(b) tangojs-line-edit bound to a boolean attribute.

tangojs/test/dev1	ALARM	
-------------------	-------	---

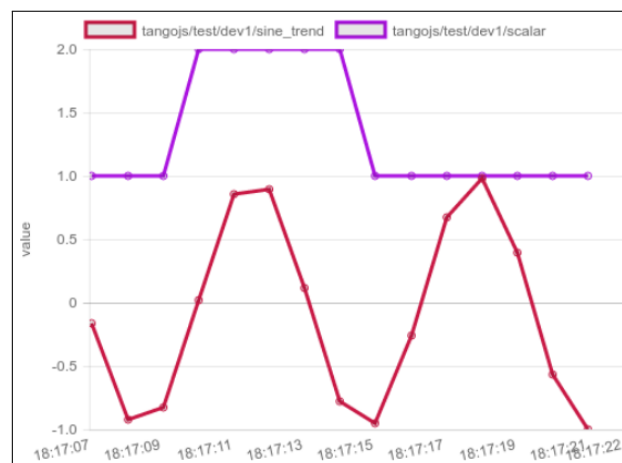
(c) tangojs-state-led bound to a device instance.

state: ON	state: OFF	state: FAULT	state: ALARM
-----------	------------	--------------	--------------




(d) A set of tangojs-command-buttons.



(e) tangojs-device-tree with all devices from the database.



(f) tangojs-trend bound to two scalar attributes

tangojs/test/dev1	ALARM	
sine_trend	-0.9816482085847568	u 
scalar	1	1 

(g) tangojs-form bound to three models of different kinds

Figure 4.6: TangoJS widgets.

- `tangojs-device-tree` - displays all devices defined in TANGO database. This is the only widget that is not bound to any model;
- `tangojs-trend` - displays the values of multiple attributes in the time domain;
- `tangojs-form` - displays a group of widgets for multiple models. The best matching widget is chosen for each model depending on its type and read/read-write mode.

Widget’s lifecycle. The behavior of most widgets is similar. Each one represents the status of a *device server* or a value of *device server*’s attribute. Upon initialization the widget is configured according to the HTML attributes specified, e.g. the `model` attribute. Then, with help of lower TangoJS layers, the widget starts polling the *device server*. The widget updates its interface periodically, using the latest data received from the *device server*, e.g. `tangojs-trend` draws a new point on plot. Most attributes of a widget may be changed dynamically, after the widget has been created. A change to some attributes, like mentioned `model` or `poll-period` leads to a widget’s reinitialization. During reinitialization, the polling loop is usually restarted.

Apart from constantly updating its layout, a widget also handles user’s input. `tangojs-line-edit`, for instance, reads the value entered by user, converts it to a TANGO data type and sends it to the *device server*. This is performed independently of layout updates.

All widgets are custom HTML elements. As such, their lifecycle is controlled by callbacks defined in *Custom Elements* specification. Widget initialization is handled in `createdCallback`. Reinitialization due to an attribute change is performed in `attributeChangedCallback`. A sequence diagram with widget’s lifecycle is presented in Fig. 4.7. Widgets and *Custom Elements* are covered more deeply in Appendix B and Appendix D.

4.8. Interworking between TangoJS layers

TangoJS modules are separated from each other and have different responsibilities. However, even simple use cases like reading a device’s state or setting an attribute, require interactions between all the layers.

When it comes to the programmable access, application developer typically interacts with the **TangoJS Core** layer. All actions are passed down through the TangoJS stack until they reach the *Connector* layer, where any network communication occurs. In the connector, a promise of result is always returned. It is later resolved to a concrete value. A scenario with reading a single attribute is presented in Fig. 4.5. Other use cases, like writing value to an attribute, reading a device’s state or invoking a command follow a similar pattern.

More complex scenarios are often implemented in widgets. Widgets are usually bound to TANGO models. Upon initialization or reinitialization the widget has to fetch its configuration from the device and then periodically poll the device to update its value.

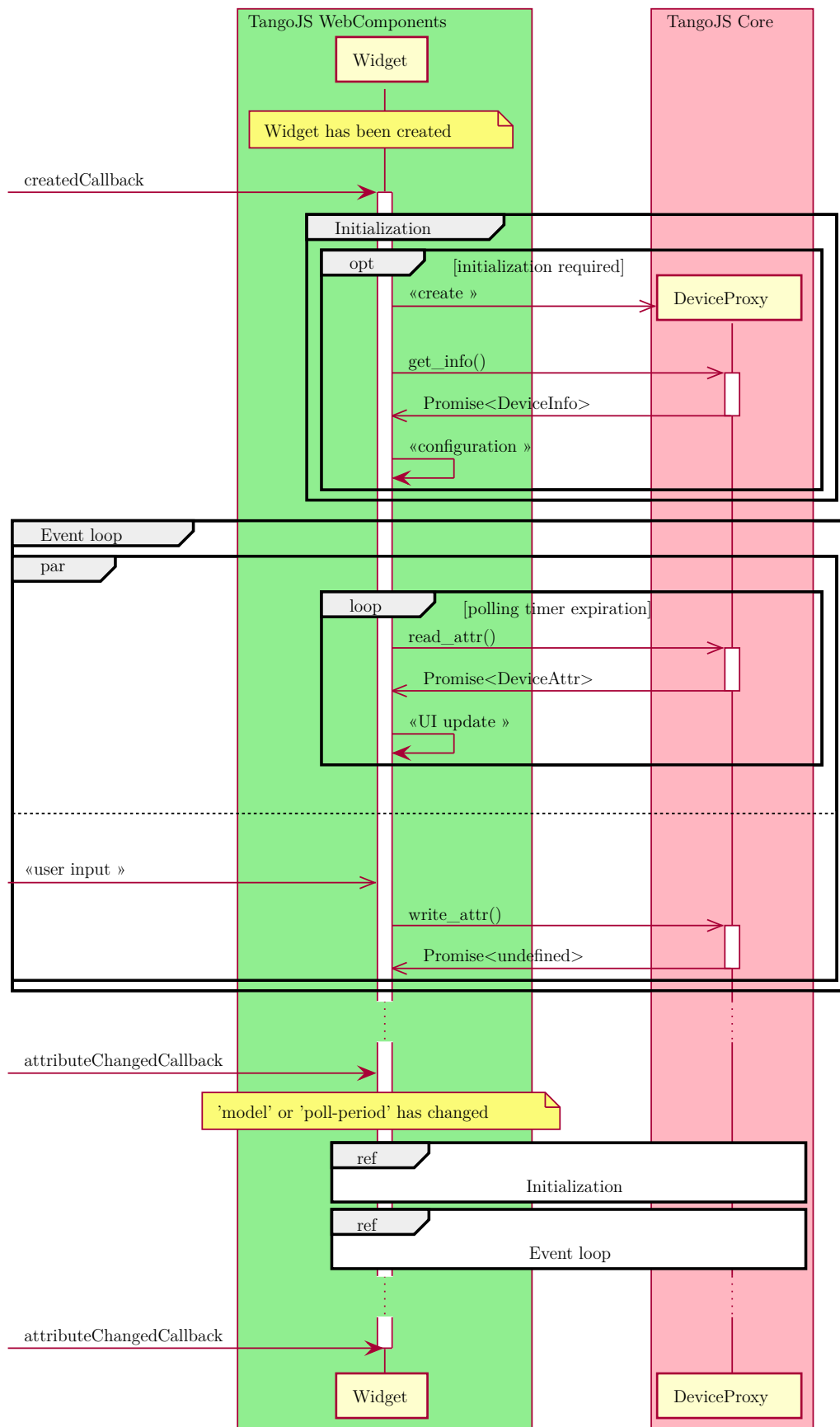


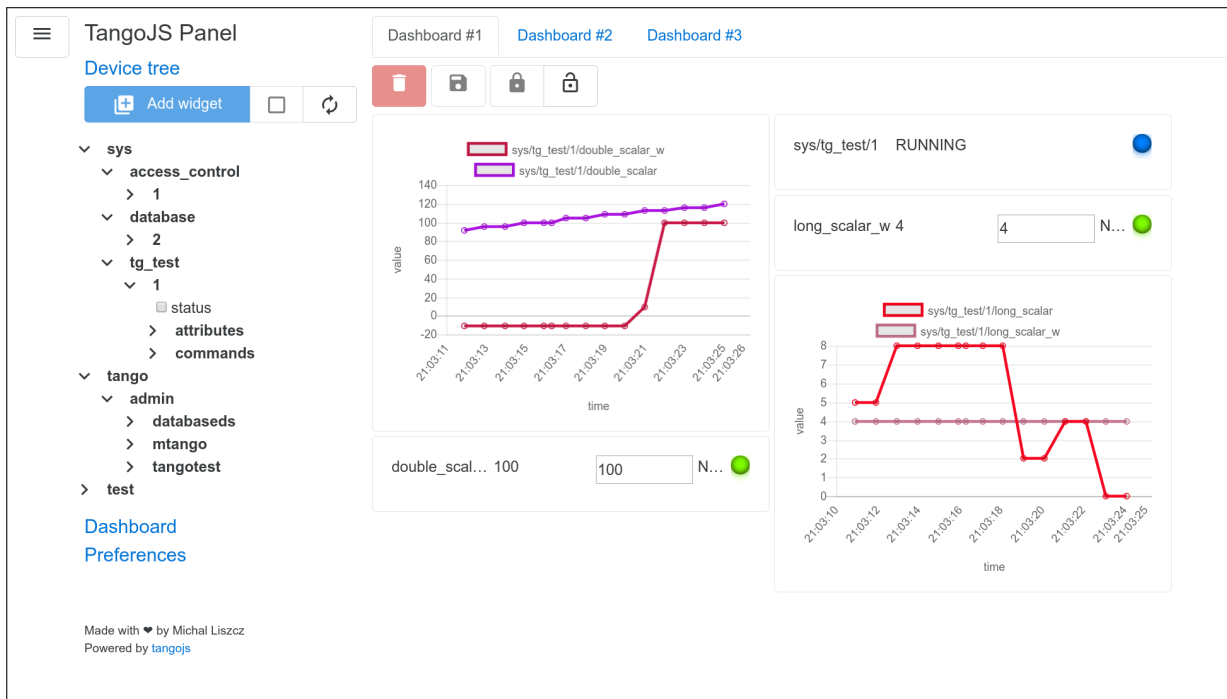
Figure 4.7: Widget's lifecycle. Interactions with *TangoJS Core* has been simplified.

Some widgets, like `tangojs-trend` or `tangojs-form` offers support for multiple models. Such scenarios require polling multiple devices and monitoring multiple attributes.

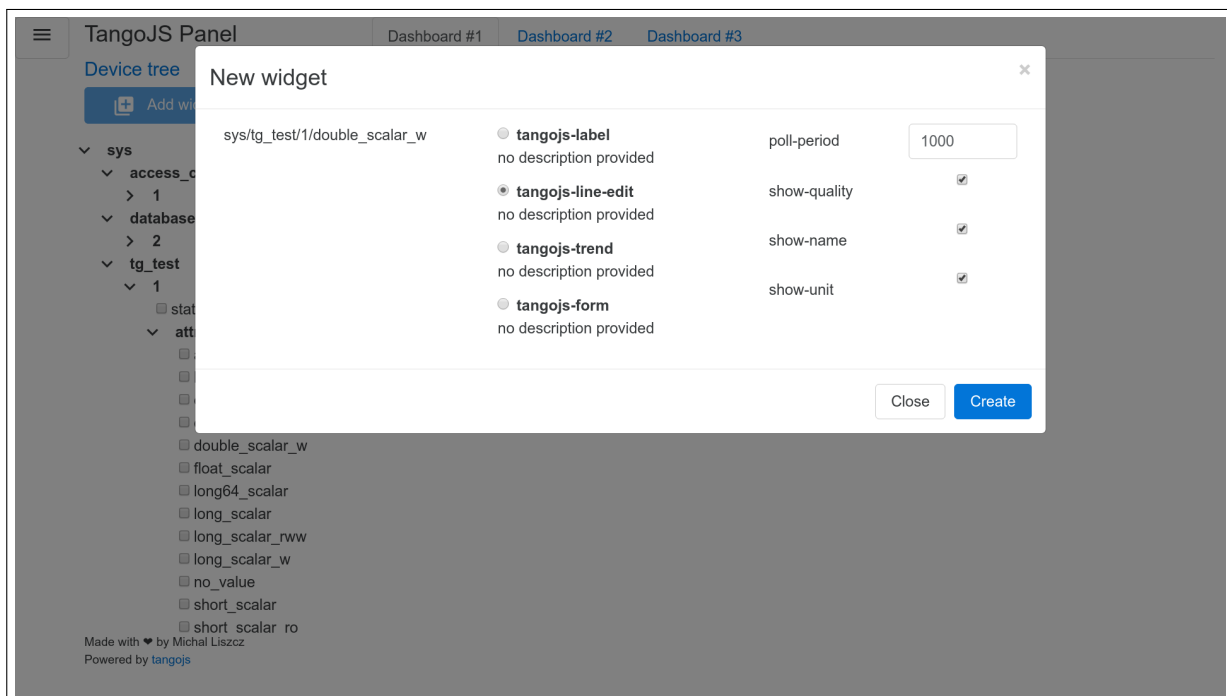
4.9. TangoJS Panel - synoptic panel application

It is not always necessary to build a dedicated GUI application for a given set of hardware components. There are tools like Taurus, which support creating GUIs interactively, by placing widgets on a panel. This process can be performed by the hardware operator, to **adapt the GUI to the current requirements** and **display only relevant information**. TangoJS also supports this scenario, through a web application called *TangoJS Panel*. Because of its complexity, it proves that TangoJS can be successfully used for building complete and non-trivial web applications.

TangoJS Panel is a framework-less solution that also uses *Web Components* technology to separate its UI parts. This approach gives the developer full control over the DOM and allows for some optimizations related UI updating. The application is presented in Fig. 4.8.



(a) Dashboard created by placing desired widgets on a panel.

(b) *New widget* dialog window.Figure 4.8: Example views from *TangoJS Panel* application.

Chapter 5

Results

In this chapter we discuss the outcomes of the presented work, review the fulfillment of the initial goals and summarize the achievements. TangoJS is compared to the existing solutions and its usability and usefulness in web development is evaluated.

5.1. Goals review

The goals of this work formulated initially has not been in a form of strict, measurable requirements. It was rather a set of design principles and guidelines to follow during the project development. **All the goals have been achieved.**

The main reason behind the development of TangoJS was to provide a **simple, yet extensible and flexible** solution for building web-based client applications for TANGO Control System. This has been reflected in design decisions, software architecture, platform choice and even the distribution model. There are highlighted the areas where most focus has been put into.

Adaptability. An adaptive user interface changes its layout depending on the current context. This process can be performed automatically by the UI layer, or the UI can provide end-user with an option to change the layout manually. In TangoJS' case both options are supported. This is true at a widget level and also for the whole *TangoJS Panel* application, built with these widgets.

TangoJS widgets benefit from the latest CSS features, like Flexbox [34] and Grid Layout [35]. This allows for flexible layout adaptation to the available space and other sizing constraints. As a web technology, TangoJS layout can be configured via CSS rules. The application developer may then use media queries [36] together with appropriate selectors to adapt the widgets automatically to the device they are displayed on.

Apart from all the adaptability aspects offered by separate widgets, the *TangoJS Panel* application allows users to configure the interface manually, by choosing *what widgets* are displayed, *where* the widgets are displayed and *what fields* are included in each widget.

This allows the user for building personalized *synpotic panels*, adapted for the current requirements and changed dynamically.

Simplicity. TangoJS is simple to start with and has a minimal learning curve. Only general knowledge of web development is required to create own applications. There are no third-party frameworks involved and the set of dependencies is kept minimal. All components from the TangoJS stack have been deployed to the npm. Thus, including TangoJS is as simple as adding a new dependency to the project. Novice developers to TangoJS may choose to start with a blank web application template available in TangoJS repositories.

Extensibility. TangoJS has been designed to be extensible at many levels. The whole backend part, which connects TangoJS to the TANGO infrastructure, may be replaced, by writing a dedicated *Connector*. This pluggable backends allow TangoJS to be adapted to certain deployment requirements and network configuration. It is also possible to extend the behavior of TangoJS by implementing new widgets. Developers are supported in this task by the various utility functions available in the *TangoJS Web Components* package.

Standards-driven. By using only standard web technologies, a wider audience of developers can be targeted. Web standards form the core of browser-based development, and every other framework is built upon them. It is safe to assume that most web developers are familiar with raw DOM APIs. This also makes TangoJS a future-proof technology and does not expose it to the risk of being tied to old, unpopular or deprecated third party libraries.

User experience. Any piece of software exposed for direct interactions with end-users that has to care about user experience. This is especially crucial in case of graphical applications. The overall user experience consists of various factors like UI's usability, accessibility and design. Since TangoJS is a library, not an application, thus, it makes no sense to evaluate TangoJS against these factors. Instead, a more detailed analysis of *TangoJS Panel* application is provided in Sec. 5.3.

5.2. Comparison with existing solutions

TangoJS can be used as a frontend for any of the existing backend solutions, including *Tango REST*, *Taurus Web* or *mTango*. In case of *mTango* it may be used as a replacement for its native frontend layer. Tbl. 5.1 compares TangoJS to the existing web-based TANGO frontend libraries.

Table 5.1: Comparison of TangoJS with existing web-based GUI libraries.

Feature	Canone	Taurus Web	mTango	TangoJS
Has widgets	yes, (limited AJAX support)	no	yes, (5 views, image and plot)	yes, (7 base widgets and variants)

Feature	Canone	Taurus Web	mTango	TangoJS
Has interactive <i>synoptic panel</i>	yes	N/A	no	yes, (<i>TangoJS Panel</i>)
Supports user accounts	yes (database)	no	yes (multiple options)	N/A, (depends on backend)
Supports TANGO events	no	no	yes	no
Backend server technology	PHP, Python	Python, Tornado, WebSocket	Java, RESTEasy, HTTP	N/A, (depends on backend)

Only the web-related aspects have been chosen as comparison criteria. The comparison indicates that the TangoJS is the leading choice in all categories but TANGO events support. **Events are currently not implemented and shall be addressed in future releases. Use of events can significantly reduce network traffic, but it requires support on both the frontend and backend side.**

5.3. Usability evaluation

A quantitative evaluation of user interfaces is a complex task and the results are often different from real user experiences [37]. This is due to the involvement of rather unpredictable human factors, like mind, perception or personal preferences. To address these issues, a method called *heuristic evaluation* has emerged [38]. This is one of the *usability engineering* methods.

The heuristic evaluation helps to detect problems with the software usability, by examining the user interface in the context of a set of well-defined principles, called *heuristics*. The most widely used heuristics have been proposed by the Jakob Nielsen in 1994 [39]. They are however still applicable to today's software.

The heuristic evaluation is not the only method of *usability inspection*. There have been other, more formal methods proposed [40], like *cognitive walkthroughs*, *formal inspections*, *feature inspections* or *standards inspections*. These methods usually require involving domain experts who provide a usability feedback.

The heuristic evaluation is quite a simple method and produces a reasonable outcome. It has been chosen to evaluate TangoJS' usability and user experience. The 10 heuristics, as formulated by Nielsen [40]:

1. **Visibility of system status:** *«The system should always keep users informed about what is going on, through appropriate feedback within reasonable time».*
2. **Match between system and the real world:** *«The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order».*
3. **User control and freedom:** *«Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo».*
4. **Consistency and standards:** *«Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions».*
5. **Error prevention:** *«Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action».*
6. **Recognition rather than recall:** *«Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate».*
7. **Flexibility and efficiency of use:** *«Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions».*
8. **Aesthetic and minimalist design:** *«Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility».*
9. **Help users recognize, diagnose, and recover from errors:** *«Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution».*
10. **Help and documentation:** *«Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large».*

The above heuristics have been applied to *TangoJS Panel* application as well as to separate widgets to detect potential usability issues.

Visibility of system status. Most widgets in TangoJS have a *quality/status* LED bulb indicator. In the normal mode of operation this bulb shows status received from device server. Whenever a communication error occurs, the indicator changes its color. This directly translates to the visibility of system in the case of *Panel* application.

Match between system and the real world. TangoJS widgets map directly to the corresponding entities in TANGO's object model, like *devices* or *attributes*. Each widget has a strict set of responsibilities and usually performs a limited number of operations, like reading or writing a value. The dashboards in the *Panel* application correspond to physical panels and switches in real-world control rooms.

User control and freedom. The use cases for TangoJS widgets are simple and often consist of a single action. In case of the *Panel* application, users can re-arrange the widgets and remove unwanted ones.

Consistency and standards. The widgets are consistent one with another and they share the same set of design principles related to the functionality and layout. The widgets have been inspired by the Taurus framework, which is the leading and widely used library for creating widget-based GUI clients for TANGO on desktops.

Error prevention. In case of web-based or distributed applications it is impossible to prevent errors from occurring. This is due to the fact that whenever network communication is involved, the errors may happen. In such a case, the widgets are inoperable, until the communication is back, e.g. the backend server becomes reachable again. These errors are indicated to the user. Other kinds of errors are handled and corrected by the widget code, e.g. a bad input.

Recognition rather than recall. The widgets in TangoJS are simple pieces of UI. There are no dialog boxes or nested views and user always interacts with a single view that maps directly to a TANGO entity. The *Panel* application uses dialogs, e.g. for initial widget configuration, but the dialog view includes only entries related to a single widget, like HTML attributes configuration.

Flexibility and efficiency of use. From the end-user's perspective, the *accelerators*, as defined by Nielsen, are not applicable to TangoJS at the current stage. The UI is flat, with almost no nested dialogs. Most of the time use cases are just single actions.

Aesthetic and minimalist design. The TangoJS GUI is kept clean and minimal. It's up to the application developer to decide what information will be included in the widget's layout, e.g. a unit field or a quality bulb. In the case of the *Panel* application, the user can decide which widgets are present on the dashboard and what fields are included.

Help users recognize, diagnose, and recover from errors. If an error occurs within a widget, the user is only informed by the change of indicator's color. No message is displayed to the user. The same applies to the *Panel* application.

Help and documentation. TangoJS provides an extensive set of resources, for both developers and end-users, including webpage, widget specifications, API documentation and template project to get started.

From the analysis carried out, we see that the **rule related to error the diagnosis is violated by TangoJS**, as the user does not get sufficient information in case of an error. An explanation of error cause may help him/her to diagnose the problem and find a solution, or at least submit a descriptive ticket in the TangoJS issue tracker. This problem is going to be addressed in future TangoJS releases. Since a widget should not pop with alerts on its own, a reasonable way of error indication has to be proposed.

The analysis performed according to Nielsen's heuristics may quickly provide GUI developers with a useful feedback, without a need to involve other people, like UI experts or targeted users. The outcome of heuristic evaluation are those usability problems that may affect user experience. By performing the analysis during software development, one may address these issues at an early stage.

Chapter 6

Conclusions

It's there to conclude the thesis, the research performed and the solution delivered.

The work presented in this thesis had two main goals: to evaluate the existing solutions for building web-based TANGO applications and to bring up to date web development techniques to the TANGO web clients. The evaluation has showed that there is only one solution which achieved success and is recognized by the TANGO community. However, due to some disputable assumptions and design decisions, it may be unsuitable for everyone. The TangoJS presented in Thesis tries to address these issues. The initial goals have been fulfilled and an extensible, adaptive and modular solution has been proposed.

6.1. Role of web-based solutions in GUI development

The web-based approach is the trending solution in building graphical user interfaces for web browsers, desktops and mobile phones. There are strong reasons behind that - the applications are portable, may easily be adjusted for different devices and screen sizes, lots of libraries and frameworks are available and applications may be developed using different styles, like object-oriented or functional programming. The web applications are fast to prototype due to the use of declarative HTML and styling with CSS.

This has been verified through TangoJS development. The TangoJS applications may be ported to desktop or mobile, using tools like Electron or Cordova. The developers may use any framework they like, because TangoJS widgets behave like native UI controls, derived from `HTMLElement`. The commonly agreed semantics applies here - the widgets can work without any configuration, and to write a simple web application using TangoJS, one has to write nearly no Javascript code.

Framework-less solutions. From many available frontend frameworks, like Angular [41] or React [42], TangoJS chooses none. Developing even complex web applications using plain DOM APIs is possible nowadays. Features once available only in libraries like jQuery have been incorporated into the DOM. A one popular argument against this framework-less

approach was that the *vanilla* JS application are unstructured, and thus unmaintainable. This is not true anymore, thanks to the technologies like *Web Components*.

Componentized approach. The recent emergence of *Web Components* standard is going to put a definitive end to monolithic web applications. The componentized approach is already used in frameworks like Angular 2, React or Polymer [43], but *Web Components* allow to achieve the same effects, when not being tied to any particular framework. The component-based web applications are more maintainable than a classic MVC applications, especially when they grow larger. This approach has been applied successfully during *TangoJS Panel* development, where each piece of UI, e.g. a tabs bar or a modal window, is a separate component that talks to other components via DOM events.

6.2. Connecting TANGO and the Web

There were many attempts to access TANGO infrastructure from within web browsers. All of them however have a common concept of a *proxy server* that connects both worlds. This server has to run on a platform where the TANGO is already supported. On one side of this proxy, there is CORBA, IIOP and ZeroMQ in some cases. On the opposite side, the proxy exposes an interface accessible from the browser. This uses different technologies for data transport, including HTTP and WebSockets. Each approach have advantages and disadvantages.

HTTP. The most obvious choice for the transport technology is HTTP, because it is the native communication protocol of the Web. It is a high-level application protocol, and the HTTP methods may be mapped on to operations on TANGO resources, like GET maps to reading an attribute value, and PUT sets the value. A RESTful API can be built using HTTP protocol. The HTTP is supported by all, even old browsers. Proxies and firewalls can handle HTTP traffic. It may be cached, both on the server and browser side. HTTP scales horizontally, since it is a stateless protocol. The security can be provided with SSL for encryption.

However, in control systems, a near real-time transmission is required. The HTTP introduces large overhead which affects performance. This overhead is even larger when SSL handshake is involved. As a text based protocol and use of headers, HTTP requests and responses exhibit poor efficiency when it comes to the data-to-header ratio in case of small messages, like reading or writing a single attribute. HTTP is also not suitable for delivery of asynchronous events triggered on the server side. Applications has to constantly poll the resource to detect any changes. This limitation is commonly bypassed by using the Comet model, like long-polling. There is support for server-sent events in HTML5 standard, but it has not been widely adopted yet in web applications.

HTTP is used by Canone, Tango REST, GoTan and mTango projects. Thus, it is also used in TangoJS for accessing its default backend, mTango.

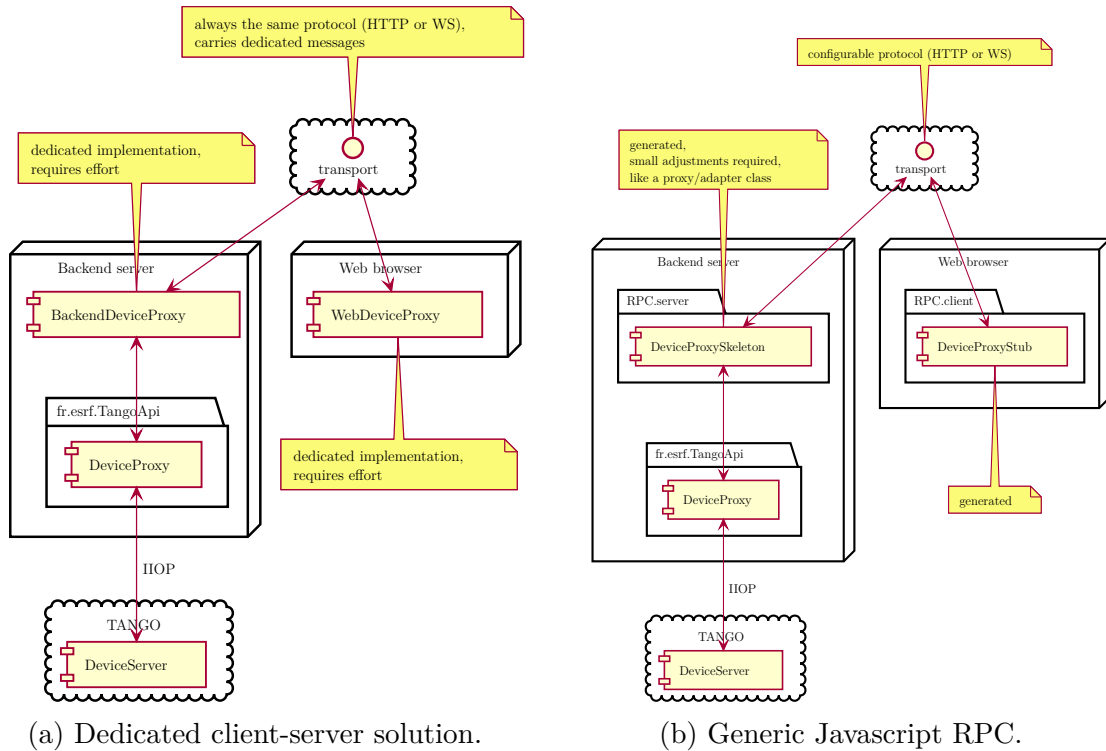


Figure 6.1: Comparison of backend access methods.

WebSocket. The goal behind WebSocket development was to provide full-duplex, bidirectional communication between the client and the server. This protocol is currently supported in all modern browsers. It is TCP-based, but uses HTTP for initial handshakes. The servers may then handle HTTP and WebSocket traffic on the same port. The WebSocket protocol have been developed to fill the gap where communication has to be initiated from the server side, and obsoletes the non-standard solutions like Comet. It is the best choice, when event-driven communication is required, like in TANGO applications. Again, SSL can be used for encryption of WebSocket traffic.

When compared with the HTTP, the WebSocket is a low-level one. The abstractions like request, response or methods have to be implemented at application level. This requires some effort but also gives the developer flexibility and field for optimizations.

The only web-based TANGO solution that uses WebSocket is the Taurus Web.

Generic RPC. To communicate with TANGO *device servers*, the abstractions like *DeviceProxy* or *DeviceAttribute* have to be available in the browser. In the currently discussed model with a middleman proxy server, these abstractions are already present on the server side. Instead of creating a dedicated RESTful or WebSocket-based API, to access the proxies from a web browser, it may be possible to use some generic middleware to perform this task. This approach, compared to the dedicated backend implementation, is depicted in Fig. 6.1

One example of such a middleware is JSON-WS [44] based on JSON-RPC specification. This project aims to support creating of RPC-based application that may be accessed

from a web-browser. It can automatically generate code for in-browser client proxies. Both HTTP and WebSocket are supported as transport mechanisms.

Currently, no project uses the above mentioned approach. **Future work aims to investigate the RPC-based solutions more deeply and try to implement a new backend for TangoJS using the JSON-WS to access jTango library on the server side.** Such solution should be easy to integrate with TangoJS, thanks to its support for pluggable backends.

HTIOP. The GIOP, *General Inter-ORB Protocol*, is the specification of a protocol that CORBA uses for communication. The IIOP, *Internet Inter-ORB Protocol* is the default implementation of GIOP used by the request brokers. It requires full access to the TCP/IP stack, which is not possible in web browsers. There is also an implementation called HTIOP [45], *HyperText Inter-ORB protocol*, which is basically an IIOP over HTTP. This protocol has been developed as part of ACE [46], the Adaptive Communication Environment, which offers TAO, a CORBA-compliant ORB. However, this project has been abandoned by the authors.

Using TAO broker in TANGO implementation may put into question the need of using a proxy server for TANGO-browser communication. This significantly simplifies the architecture, as depicted in Fig. 6.2. **The future research is going to investigate this possibility and try to implement support for HTIOP in TangoJS.**

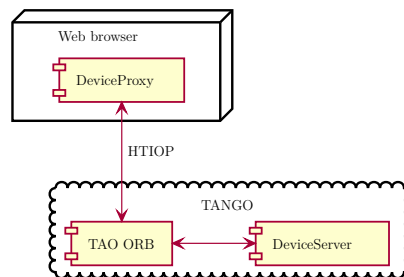


Figure 6.2: Accessing TANGO from within web browser over HTIOP.

Removing the CORBA. There have been several attempts to remove the heavyweight and complex CORBA from TANGO and replace it by another technology, e.g. ZeroMQ[47]. All these attempts have failed. With the recent emergence of lightweight RPC frameworks that work in a web browser like gRPC or Apache Thrift, it should be possible to replace the middleware layer without any modifications to the TANGO code, just by implementing proxy wrappers for new stubs and skeletons, to expose CORBA-like interface to the TANGO code.

6.3. Future work

The Evaluation of the proposed solution has shown that there is still place for improvements and a few issues have to be addressed in TangoJS.

Widgets. The set of the currently implemented widgets allows for developing even complex web clients for TANGO, but there are some use cases which are currently not covered, e.g. a scatter X-Y plots. TangoJS widget toolkit has been designed to be extensible, and such a widget can be added at any time without any impact on the rest of the project.

Event support. At the current stage, TANGO events are not supported in TangoJS. The constant polling can affect performance and generate unnecessary traffic. To implement event support in TangoJS, a backend with event support is required. The mTango supports events, but uses JSONP and Comet model. As a design decision, TangoJS should use only standardized technologies, and it would be possible to handle events with WebSocket connection or using server-sent events from HTML5. This should be investigated to propose the best solution.

Error indication. This issue has arisen during the usability evaluation. The user is not sufficiently informed about the cause when error happened. Some way to indicate such problems has to be developed, since changing the indicator bulb is not enough and does not help user to find a solution for the problem's root cause.

Backends. Even that mTango does its job as a default backend reasonably well, it would be better to have such a backend generated automatically by some third-party RPC framework. This shall probably introduce some overhead and optimizations will not be possible, but the maintenance of backend and *Connector* will be simplified. Other scenarios, e.g. backend-less one, as described above, should also be investigated.

References

- [1] NSRC Solaris, “Resources to download,” 20015. [Online]. Available: http://www.synchrotron.uj.edu.pl/en_GB/materialy-do-pobrania. [Accessed: 01-Oct-2016].
- [2] A. Daneels and W. Salter, “Selection and evaluation of commercial scada systems for the controls of the cern lhc experiments,” in *Proceedings of the 1999 international conference on accelerator and large experimental physics control systems, trieste*, 1999, p. 353.
- [3] A. Daneels and W. Salter, “What is scada?” 1999.
- [4] S. A. Boyer, *Scada: Supervisory control and data acquisition*, 4th ed. USA: International Society of Automation, 2009.
- [5] A. Götz and others, “TANGO—An object oriented control system based on corba,” 1999.
- [6] GitHub, Inc., “Electron,” 2013. [Online]. Available: <http://electron.atom.io/>. [Accessed: 26-Sep-2016].
- [7] The Apache Software Foundation, “Apache cordova,” 2012. [Online]. Available: <https://cordova.apache.org/>. [Accessed: 26-Sep-2016].
- [8] A. Charland and B. Leroux, “Mobile application development: Web vs. native,” *Communications of the ACM*, vol. 54, no. 5, pp. 49–53, 2011.
- [9] A. Mesbah and A. van Deursen, “Migrating multi-page web applications to single-page ajax interfaces,” in *11th european conference on software maintenance and reengineering (csmr’07)*, 2007, pp. 181–190.
- [10] K. Z. Gajos, M. Czerwinski, D. S. Tan, and D. S. Weld, “Exploring the design space for adaptive graphical user interfaces,” in *Proceedings of the working conference on advanced visual interfaces*, 2006, pp. 201–208.
- [11] A. Charland and B. Leroux, “Mobile application development: Web vs. native,” *Commun. ACM*, vol. 54, no. 5, pp. 49–53, May 2011.
- [12] F. Rivoal, “Media Queries,” W3C, W3C Recommendation, Jun. 2012.
- [13] C. A. Miller, H. Funk, R. Goldman, J. Meisner, and P. Wu, “Implications of adaptive vs. adaptable uis on decision making: Why ‘automated adaptiveness’ is not always the

right answer,” in *Proceedings of the 1st international conference on augmented cognition*, 2005, pp. 22–27.

[14] C. Pascual-Izarra, G. Cuní, C. Falcón-Torres, D. Fernández-Carreiras, Z. Reszela, and M. Rosanes, “Effortless creation of control & data acquisition graphical user interfaces with taurus,” *THHC3O03, ICALEPCS2015, Melbourne, Australia*, 2015.

[15] Object Management Group, Inc., “CORBA,” 1997. [Online]. Available: <http://www.corba.org/>. [Accessed: 26-Sep-2016].

[16] Object Management Group, Inc., “Common object request broker architecture spec,” 2012–Nov-01AD. [Online]. Available: <http://www.omg.org/spec/CORBA/>. [Accessed: 26-Sep-2016].

[17] R. Ben-Natan, Ed., *Corba: A guide to common object request broker architecture*. Hightstown, NJ, USA: McGraw-Hill, Inc., 1995.

[18] iMatix Corporation, “ZeroMQ,” 2007. [Online]. Available: <http://zeromq.org/>. [Accessed: 26-Sep-2016].

[19] Oracle Corporation, “MySQL,” 1995. [Online]. Available: <https://www.mysql.com/>. [Accessed: 28-Nov-2016].

[20] MariaDB Corporation Ab, MariaDB Foundation, “MariaDB,” 2009. [Online]. Available: <https://mariadb.org/>. [Accessed: 28-Nov-2016].

[21] The TANGO Team, “The TANGO Control System Manual Version 9.1,” 2015–Aug-18AD. [Online]. Available: http://ftp.esrf.eu/pub/cs/tango/tango_91.pdf. [Accessed: 26-Jun-2016].

[22] F. Poncet and J. L. Pons, “TANGO application toolkit (atk),” in *10th ical eps int. conf. on accelerator & large expt. physics control systems. geneva*, 2005.

[23] M. Pelko, K. Zagar, L. Zambon, and A. Green, “Canone—A highly-interactive web-based control system interface,” in *Proceedings of ical eps07*, 2007.

[24] T. Controls, “Canone: The animated graphical web interface of tango,” 2005. [Online]. Available: <http://plone.tango-controls.org/download/canone/canone>. [Accessed: 28-Sep-2016].

[25] CELLS / ALBA Synchrotron, “Taurus in the cloud,” 2013. [Online]. Available:

[http://plone.tango-controls.org/Events/meetings/may_2013/Taurus Web - Tango Collaboration Meeting ALBA 2013.pdf](http://plone.tango-controls.org/Events/meetings/may_2013/Taurus%20Web%20-%20Tango%20Collaboration%20Meeting%20ALBA%202013.pdf). [Accessed: 16-Jun-2016].

[26] Ł. Mitka, “Mobile application for tango control system and testing its performance.” Master’s thesis, AGH University of Science and Technology, 2015.

[27] V. Hardion, “GoTan project website,” 2012. [Online]. Available: <https://github.com/hardion/GoTan>. [Accessed: 18-Jun-2016].

[28] I. Khokhriakov, “mTango project website,” 2013. [Online]. Available: <https://bitbucket.org/hzgwpn/mtango/wiki/Home>. [Accessed: 19-Jun-2016].

[29] D. Crane and P. McCarthy, “What are comet and reverse ajax?” in *Comet and reverse ajax: The next-generation ajax 2.0*, Berkeley, CA: Apress, 2009, pp. 1–9.

[30] K. Kambona, E. G. Boix, and W. De Meuter, “An evaluation of reactive programming and promises for structuring collaborative web applications,” in *Proceedings of the 7th workshop on dynamic languages and applications*, 2013, pp. 3:1–3:9.

[31] Node.js Foundation, “Node.js - a JavaScript runtime,” 2009. [Online]. Available: <https://nodejs.org/>. [Accessed: 05-Apr-2016].

[32] npm, Inc., “npm - a package manager for JavaScript,” 2009. [Online]. Available: <https://www.npmjs.com/>. [Accessed: 05-Apr-2016].

[33] The UMD Contributors, “Universal module definition,” 2011. [Online]. Available: <https://github.com/umdjs/umd>. [Accessed: 26-Sep-2016].

[34] T. Atkins Jr., E. J. Etemad, and R. Atanassov, “CSS Flexible Box Layout Module Level 1,” W3C, W3C Candidate Recommendation, May 2016.

[35] T. Atkins Jr., E. J. Etemad, and R. Atanassov, “CSS Grid Layout Module Level 1,” W3C, W3C Working Draft, May 2016.

[36] B. S. Gardner, “Responsive web design: Enriching the user experience,” *Sigma Journal: Inside the Digital Ecosystem*, vol. 11, no. 1, pp. 13–19, 2011.

[37] K. Väänänen-Vainio-Mattila, V. Roto, and M. Hassenzahl, “Towards practical user experience evaluation methods,” *EL-C. Law, N. Bevan, G. Christou, M. Springett & M. Lárusdóttir (eds.) Meaningful Measures: Valid Useful User Experience Measurement*

(*VUUM*), pp. 19–22, 2008.

[38] J. Nielsen and R. Molich, “Heuristic evaluation of user interfaces,” in *Proceedings of the sigchi conference on human factors in computing systems*, 1990, pp. 249–256.

[39] J. Nielsen, “Heuristic evaluation,” *Usability inspection methods*, vol. 17, no. 1, pp. 25–62, 1994.

[40] J. Nielsen, “Usability inspection methods,” in *Conference companion on human factors in computing systems*, 1994, pp. 413–414.

[41] Google, “AngularJS,” 2010. [Online]. Available: <https://www.angularjs.org/>. [Accessed: 18-Sep-2016].

[42] Facebook Inc., “React,” 2013. [Online]. Available: <https://facebook.github.io/react/>. [Accessed: 18-Sep-2016].

[43] Google, “Polymer Project,” 2012. [Online]. Available: <https://www.polymer-project.org/>. [Accessed: 18-Sep-2016].

[44] M. Stanchev and P. Stoev, “JSON-WS,” 2015. [Online]. Available: <https://github.com/ChaosGroup/json-ws>. [Accessed: 28-Sep-2016].

[45] D. C. Schmidt, “HTTP tunneling inter-orb protocol,” 2011. [Online]. Available: <https://github.com/cflowe/ACE/tree/master/TAO/orbsvcs/orbsvcs/HTIOP>. [Accessed: 28-Sep-2016].

[46] D. C. Schmidt, “The adaptive communication environment (ace),” 2011. [Online]. Available: <http://www.cs.wustl.edu/~schmidt/ACE.html>. [Accessed: 28-Sep-2016].

[47] E. Taurel, “tango_zmq,” 2013. [Online]. Available: https://github.com/taurel/tango_zmq. [Accessed: 28-Sep-2016].

[48] The Apache Software Foundation, “Apache tomcat,” 1999. [Online]. Available: <http://axis.apache.org/axis2/java/core/>. [Accessed: 27-Sep-2016].

[49] A. Croll, “A fresh look at javascript mixins.” [Online]. Available: <https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins>. [Accessed:

26-Sep-2016].

[50] The Apache Software Foundation, “Apache axis 2,” 2004. [Online]. Available: <http://axis.apache.org/axis2/java/core/>. [Accessed: 27-Sep-2016].

[51] W3C, “Web components current status,” 2014. [Online]. Available: <https://www.w3.org/standards/techs/components>. [Accessed: 28-Sep-2016].

[52] Z. Rocha and others, “WebComponents.org,” 2014. [Online]. Available: <http://webcomponents.org/polyfills/>. [Accessed: 28-Sep-2016].

[53] D. Glazkov, R. Weinstein, and T. Ross, “HTML Templates,” W3C, W3C Working Group Note, 2016–Feb. 25AD.

[54] D. Denicola, “Custom Elements,” W3C, W3C Working Draft, Aug. 2016.

[55] D. Glazkov and H. Morrita, “HTML Imports,” W3C, W3C Working Draft, Feb. 2016.

[56] H. Ito, “Shadow DOM,” W3C, W3C Working Draft, Aug. 2016.

[57] Mozilla Foundation, “Brick,” 2013. [Online]. Available: <http://brick.mozilla.io/>. [Accessed: 26-Sep-2016].

[58] Microsoft Corporation, “X-Tag Web Components,” 2013. [Online]. Available: <https://x-tag.github.io/>. [Accessed: 26-Sep-2016].

[59] R. Harris, O. Segersvard, and others, “rollup.js,” 2015. [Online]. Available: <http://rollupjs.org/>. [Accessed: 26-Sep-2016].

[60] T. A. Jr., “CSS Custom Properties for Cascading Variables Module Level 1,” W3C, W3C Candidate Recommendation, Dec. 2015.

[61] T. Atkins Jr. and E. J. Etemad, “CSS Values and Units Module Level 3,” W3C, W3C Candidate Recommendation, Jun. 2015.

Glossary

AJAX (Asynchronous Javascript and XML) is a set of technologies used by the web browsers to make asynchronous requests to the web servers. Often XML is replaced by JSON.

API (Application Programming Interface) is a set of procedures and interfaces exposed by a library or a framework to its users.

ATK (Tango Application ToolKit) is a framework for creating Swing-based graphical TANGO applications in Java.

backend is the part of a web application that runs on a web server. It is usually responsible for data processing.

CLI (Command-line Interface) is a textual UI where any interactions are performed by typing commands.

Comet is a model of connection handling between a browser and a server where the server side can initialize communication.

CORBA (Common Object Request Broker Architecture) is an object-oriented middleware based on GIOP protocol.

CORS (Cross Origin Resource Sharing) is a technique that allows accessing resources from another *origin* (domain) using OPTIONS HTTP method.

CSS (Cascading Style Sheets) is a styling language for HTML.

DataBases is a purely software device server that allows for interactions with TANGO database.

device server is an abstract entity in TANGO system. It often controls a single piece of hardware.

DOM (Document Object Model) is an API for manipulating HTML from within Javascript.

ECMAScript is a dynamic scripting language standardized by ECMA International. Javascript is one of ECMAScript implementations.

end user is a person for whom the software has been designed.

framework is a software used by another software to facilitate certain tasks.

frontend is the part of a web application that runs in a browser. It is usually the presentation layer.

GUI (Graphical User Interface) is an UI where interactions may be performed using graphical icons and dialogs.

HTML (HyperText Markup Language) is a markup language for defining web page layout.

IDL (Interface Definition Language) is a language for defining language-independent interfaces.

Javascript is a scripting language for web browsers.

JSDoc is a tool that generates API documentation from Javascript source comments.

JSON (Javascript Object Notation) is a textual data exchange format which uses Javascript syntax to represent objects.

JSONP is a technique that allows to bypass the same origin policy restrictions without implementing the CORS protocol.

LDAP (Lightweight Directory Access Protocol) is a protocol for accessing objects in remote directory. It is often used for authentication purposes.

middleware is a software that allows connecting systems created using different technologies. It often works in distributed environments.

model is, in Taurus framework, an abstract entity that represents a physical device or its attribute.

Model-View-* is a common name for *Model-View-Controller*, *Model-View-Presenter*, *Model-View-Adapter* and *Model-View-ViewModel* architectural patterns.

Node.js is a cross-platform Javascript runtime based on Google's V8.

npm is a package manager and a package repository for Node.js.

operator is a person who controls the hardware during an experiment. Also end user of TANGO GUI tools.

polyfill is a patch that provides functionality from latest language releases to the older browsers.

promise is a class introduced in ECMAScript 2015 that encapsulates a result of an asynchronous computation.

REST (Representational State Transfer) is an architectural pattern used by web applications which navigate between resource representations, usually JSON objects.

RPC (Remote Procedure Call) is a concept of executing a procedure on a remote machine.

same origin policy is a security policy that prevents from accessing resources located on another domain (*origin*).

SCADA (Supervisory Control And Data Acquisition) is a type of computer system that allows for hardware control and supervision.

synoptic panel is a common name for TANGO applications that provide control over multiple devices.

TANGO database is a database where configuration of all device servers is stored.

TANGO IDL is an IDL document that defines all TANGO objects and data types.

TangoAccessControl is a purely software device server that allows for fine-grained access control over other device servers.

Taurus is a Qt-based framework for creating TANGO GUI applications in Python.

UI (User Interface) is a general term that describes interactions between the software and the user. Most common UIs are GUIs and CLIs.

UMD (Universal Module Definition) is a module specification for Javascript which allows creating modules for both Node.js and browsers.

Vanilla JS is a term for a framework-less Javascript application that uses only standard APIs.

W3C (World Wide Web Consortium) is a standardization body for multiple web technologies like HTML or CSS.

WebComponents is a set of web standards that allows building componentized web applications.

WebSocket is a full-duplex protocol for communication between a browser and a server.

widget is a small, reusable, isolated, self contained piece of GUI.

ZeroMQ is a middleware for RPC and message-based communication.

Appendix A

Getting started with TangoJS

TangoJS has been designed to be easy to start with. Only a basic experience in web development is required to build a simple TangoJS application. The one whose building is shown in this Appendix visualizes two scalar attributes using the *trend* widget. Additionally, value of the writable one can be changed by the user with the *line-edit* widget.

This process is visualized in Fig. A.1. The description of each step is provided in the following sections.

A.1. Configuring the backend

First of all, a backend server has to be configured. One may skip this step if he/she wants to use the simple in-memory *Connector*. However, for accessing a real TANGO infrastructure, a working instance of mTango is required.

A preconfigured mTango REST server is available in a Docker container.¹ This container has been created during the TangoJS development. It may be used with the existing TANGO deployment. The rest of this section shows how to configure mTango manually from scratch.

Deploying mTango on Tomcat. A servlet container, like Apache Tomcat [48], is required to make mTango work with TangoJS. The standalone `.jar` version of mTango won't work, since additional configuration is required. A standard `.war` archive may be downloaded from the mTango webpage². This archive should be deployed in the container as usually, e.g. by placing the archive in the `webapps` directory. Then, a device with the following parameters should be registered in the TANGO database using a tool like Jive:

- class/instance: `TangoRestServer/development`;
- device `test/rest/0`;

¹<https://hub.docker.com/r/mliszczy/mtango/>

²<https://bitbucket.org/hzgwpn/mtango/downloads/mtango.server-rc2-0.3.zip>

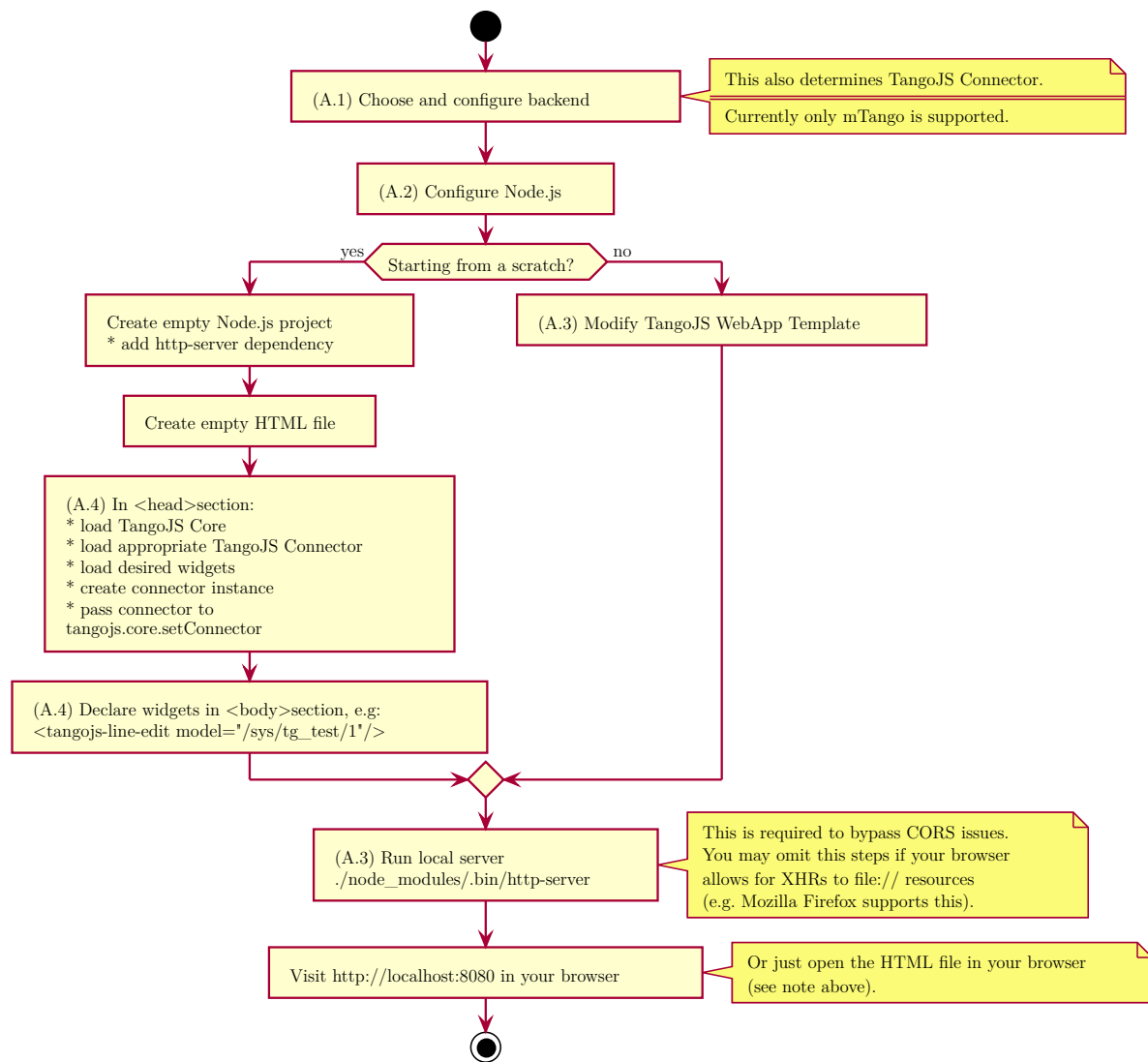


Figure A.1: Steps required to create basic TangoJS application.

mTango uses the role-based authorization offered by Tomcat. One has to add a new user to the `config/tomcat-users.xml` file, like shown in Lst. A.1.

Listing A.1 `tomcat-users.xml` - defining a user for mTango.

```
1 <role rolename="mtango-rest"/>
2 <user username="tango" password="secret" roles="mtango-rest"/>
```

At this point the container may be started to check if mTango has been set up properly. After starting Tomcat, the following URL, when typed in the browser, should return a list of all TANGO devices: `http://localhost:8080/mtango/rest/rc2/devices`.

Enabling CORS. Cross Origin Resource Sharing has to be enabled in the container to allow TangoJS access the mTango RESTful API. With the container shut down, one should edit the `web.xml` file located where the mTango archive has been extracted. A new filter has to be appended to the filter chain. Tomcat, like most web containers, offers a configurable filter that can handle the CORS preflights. Apart from configuring the filter, the OPTIONS request should be allowed to pass the security constraints. This requires modifications to the default security configuration, also in the `web.xml` file. The changes are shown in Lst. A.2.

A.2. Managing a Node.js project

The Node.js [31] is the most commonly used runtime for Javascript. It also comes with the npm [32], a package manager. The TangoJS packages are available in npm registry under the following names:

- *TangoJS Core* - `tangojs-core`;
- *TangoJS WebComponents* - `tangojs-web-components`;
- in-memory mock connector - `tangojs-connector-local`;
- mTango connector - `tangojs-connector-mtango`;

The rest of this section assumes that Node.js has been installed and configured. To create an empty project, one has to type `npm init` in an empty directory and then respond to a few questions. The TangoJS packages may be then installed like: `npm install tangojs-core`. The required steps are as follows:

```
1 ~ $ mkdir webapp
2 ~ $ cd webapp
3 ~/webapp $ npm init
4 ~/webapp $ npm install tangojs-core tangojs-web-components
```

During the above process, the TangoJS packages have been pulled to the `node_modules` directory. Also, a `package.json` file has been created in the current directory. This file contains the project definition, including all the installed dependencies. Changes to this file may be made manually, but when a version of a dependency is changed or a new

dependency is added, one should type `npm install`, with no package names, to update the dependencies.

A.3. Using the *TangoJS WebApp Template*

The steps described in the previous section may be omitted, and the *TangoJS WebApp Template*³ may be used instead. It is a starter TangoJS project that has all the required dependencies declared and provides an `index.html` file skeleton.

To use the project, one has to just clone the repository, select a branch with a desired backend and pull the dependencies. The steps are as follows:

```
1 ~ $ git clone https://github.com/tangojs/tangojs-webapp-template
2 ~ $ cd tangojs-webapp-template
3 ~/tangojs-webapp-template $ git checkout mtango
4 ~/tangojs-webapp-template $ npm install
5 ~/tangojs-webapp-template $ npm run server
```

A simple HTTP server has been started. This server binds to the 8080 port by default and serves the files from the project's repository. To see the `index.html` rendered in the browser, one should type the `http://localhost:8080` in the address bar.

NOTE: TangoJS requires the *Web Components* and *CSS Grid Layout* to run properly. These features may be disabled in some browsers. For Mozilla Firefox, the flags `dom.webcomponents.enabled` and `layout.css.grid.enabled` should be set to `true`. In Chromium and derivatives *Experimental Web Platform features* should be enabled.

A.4. Application code

With the *WebApp Template* up and running, one may start to customize it, in order to create an own web-based TANGO client application. Any libraries and frameworks may be included at this stage.

Loading TangoJS. The `index.html` file, mentioned earlier, is a standard HTML file. In the `head` section, after the usual entries like `title` and `meta` tags, the TangoJS is loaded and configured. This template application uses the *trend* widget, thus *Chart.js* and *Moment.js* dependencies are loaded first. Then the TangoJS packages like *core* and *connector* are loaded. These packages are standard Javascript files loaded with the `script` tag. After the packages, the developer may choose which widgets to load. Each widget is packaged in a separate file, which should be loaded as an *HTML Import*, using the `link` element.

³<https://github.com/tangojs/tangojs-webapp-template>

Once all the packages have been loaded, the connector has to be configured. This is also usually done in the `head` section, inside a `script` tag. One may instantiate a desired connector, e.g. the *TangoJS mTango Connector* and then pass this connector to the `tangojs.core.setConnector(connector)` function. The process described here is shown in Lst. A.3.

Using the widgets. TangoJS widgets may be used declaratively, by placing the desired tags in the document `body`. Widgets may be also created using the imperative DOM manipulation APIs. The available widgets have been discussed in Sec. 4.7 of Thesis. The TangoJS project webpage⁴ also lists the widgets, as well as provides the description of the attributes that may alter the widgets's layout or behavior.

In this example application, two widgets are going to be added to the page, using two possible approaches. The first widget is a `tangojs-trend`, which displays the values of two attributes over time. These attributes are `sys/tg_test/1/long_scalar_w` and `sys/tg_test/1/double_scalar`. The `long_scalar_w` is a writable attribute. Thus, another widget, `tangojs-line-edit` will be added to allow one change to the value of this attribute. The code shown in Lst. A.4 should be added to the `body` tag.

The layout of the application built in this section is shown in Fig. A.2.

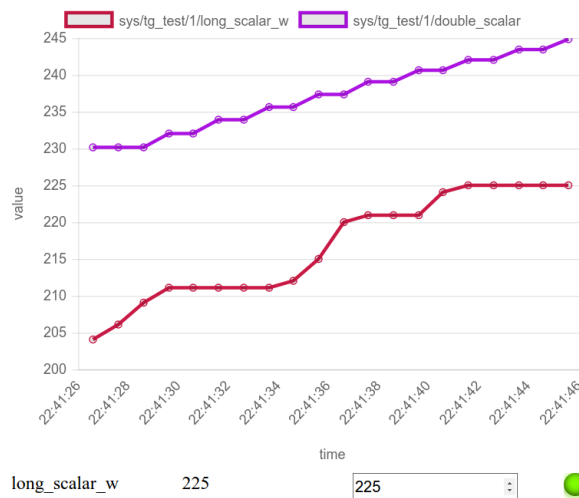


Figure A.2: The *TangoJS WebApp Template* application's view.

⁴<http://tangojs.github.io/>

Listing A.2 web.xml - mTango CORS configuration.

```

1  <?xml version="1.0"?>
2  <web-app>
3
4      <!-- rest of the file omitted ... -->
5
6      <filter>
7          <filter-name>CorsFilter</filter-name>
8          <filter-class>org.apache.catalina.filters.CorsFilter</filter-class>
9
10         <init-param>
11             <param-name>cors.allowed.methods</param-name>
12             <param-value>GET,HEAD,POST,PUT,DELETE,OPTIONS</param-value>
13         </init-param>
14
15         <init-param>
16             <param-name>cors.allowed.headers</param-name>
17             <param-value>
18 Accept,Accept-Encoding,Accept-Language,Access-Control-Request-Method,
19 Access-Control-Request-Headers,Authorization,Cache-Control,Connection,
20 Content-Type,Host,Origin,Referer,User-Agent,X-Requested-With
21             </param-value>
22         </init-param>
23     </filter>
24
25     <filter-mapping>
26         <filter-name>CorsFilter</filter-name>
27         <url-pattern>/rest/*</url-pattern>
28     </filter-mapping>
29
30     <security-constraint>
31         <web-resource-collection>
32             <web-resource-name>Tango RESTful gateway</web-resource-name>
33             <url-pattern>/rest/*</url-pattern>
34             <http-method>OPTIONS</http-method>
35         </web-resource-collection>
36     </security-constraint>
37
38     <security-constraint>
39         <web-resource-collection>
40             <web-resource-name>Tango RESTful gateway</web-resource-name>
41             <url-pattern>/rest/*</url-pattern>
42         </web-resource-collection>
43         <auth-constraint>
44             <role-name>mtango-rest</role-name>
45         </auth-constraint>
46     </security-constraint>
47
48 </web-app>

```

Listing A.3 TangoJS configuration.

```

1 <head>
2
3   <!-- ... -->
4
5   <!-- Dependencies required if you plan to use the tangojs-trend. -->
6   <script src="node_modules/moment/min/moment.min.js"></script>
7   <script src="node_modules/chart.js/dist/Chart.min.js"></script>
8
9   <!-- TangoJS stack. -->
10  <script src="node_modules/tangojs-core/lib/tangojs-core.js"></script>
11  <script src="node_modules/[...]/tangojs-connector-mtango.js"></script>
12  <script src="node_modules/[...]/tangojs-web-components.js"></script>
13
14  <!-- Include desired widgets. -->
15  <link rel="import" href="node_modules/[...]/tangojs-line-edit.html">
16  <link rel="import" href="node_modules/[...]/tangojs-trend.html">
17
18  <!-- Connector setup. -->
19  <script type="text/javascript">
20    (function (tangojs) {
21      'use strict'
22
23      const connector = new tangojs.connector.mtango.MTangoConnector(
24        'http://172.18.0.5:8080/rest/rc2', // endpoint
25        'tango',                        // username
26        'secret')                      // password
27
28      tangojs.core.setConnector(connector)
29
30    })(window.tangojs)
31  </script>
32
33  <!-- The widgets may be styled with CSS. -->
34  <style>
35    tangojs-trend {
36      width: 500px;
37      height: 400px;
38      display: block;
39      padding-bottom: 10px;
40    }
41
42    tangojs-line-edit {
43      display: block;
44      width: 500px;
45    }
46  </style>
47
48 </head>

```

Listing A.4 Creating TangoJS widgets from application's code.

```
1 <body>
2
3   <tangojs-trend
4     model="sys/tg_test/1/long_scalar_w,sys/tg_test/1/double_scalar"
5     poll-period="1000"
6     data-limit="20">
7   </tangojs-trend>
8
9   <script>
10  (function (window) {
11    'use strict'
12
13    const document = window.document
14
15    function initApplication () {
16
17      const lineEdit = document.createElement('tangojs-line-edit')
18
19      lineEdit.setAttribute('model', 'sys/tg_test/1/long_scalar_w')
20      lineEdit.pollPeriod = 1000
21      lineEdit.showName = true
22      lineEdit.showQuality = true
23
24      document.querySelector('body').appendChild(lineEdit)
25    }
26
27    if (HTMLImports && !HTMLImports.useNative) {
28      // If HTMLImports polyfill is used, custom elements are not yet
29      // upgraded when the DOMContentLoaded is fired. The polyfill
30      // fires a~special event to indicate that all imports have been
31      // loaded. This is required to work in Firefox.
32      window.addEventListener('HTMLImportsLoaded', initApplication, true)
33    } else {
34      window.addEventListener('DOMContentLoaded', initApplication, true)
35    }
36  })(window)
37 </script>
38
39 </body>
```

Appendix B

Developing widgets for TangoJS

TangoJS is an extensible project and aims to simplify the process of adding a new widget. This section shows how to create a simple widget. Only a general knowledge of web technologies like DOM and Web Components is required to proceed.

B.1. *TangoJS WebComponents* utilities

TangoJS provides utilities that facilitate development of new widgets. The APIs described here are completely optional to use and are provided just for convenience. One may start with a blank HTML file as well.

The mentioned utilities are part of the `tangojs-web-components` package. In a TangoJS application this package is available at runtime as a `tangojs.web` object. The structure of this package is described below.

The `tangojs.web.components` is an object where constructor functions for all widgets should be attached. Attaching constructors to this object is not necessary, but this is the standard place where tools like `TangoJS Panel` look for widgets. This object is discussed later in this section.

The `tangojs.web.util` package contains a set of generic utility functions and two subpackages, namely: `converters` and `mixins`. The `converters` package contains functions for conversion between Javascript properties and DOM attributes. This is useful for implementing, e.g the reflecting properties. The `mixins` package contains mixins, or traits that provide common behaviors for widgets. The most important utilities from this package are presented below.

Functions from the `tangojs.web.util` package.

- export `function registerComponent` (`tagName`, `constructor`, `descriptor`)

Registers a custom element in the current document. The `tagName` is a `string` that will be used as a HTML tag for the new widget. The `constructor` is a constructor

function. The third argument, `descriptor` is optional and is not required for simple widgets.

- `export function getCurrentDocument ()`

Returns a `document` object that contains the currently executed script. This is useful for loading templates from imported HTML files.

- `export function hyphenatedForm (s)`

Converts the `s` from a camelCasedString to a hyphenated-string; this is the preferable naming convention in the DOM.

Functions from the `tangojs.web.util.mixins` package. There are many ways how Javascript mixins can be implemented. TangoJS uses the convention proposed in [49]. The mixins are just functions that append various methods to a given class prototype. This prototype is passed as `this`, thus mixing-in has to be performed using, e.g. a `call` method, like `mixin.call(prototype, args...)`.

- `export function withPolledModel ()`

Adds a polling behavior to the widget. The underlying attribute model is constantly polled to keep the widget up-to-date. The prototype that mixes this in should have the following methods defined:

- `onModelRead` which takes a map from model `string` to read result `Object`. This method is invoked whenever a new value is obtained from TANGO;
- `onModelError` which takes an `Error` object. This method is called whenever an error occurs;
- `createProxy` that takes a `string` model and returns a proxy, either `DeviceProxy` or `AttributeProxy`;
- `readProxy` that takes the proxy and returns a promise of `DevState` or `DeviceAttribute`.

The following methods are then added to the prototype:

- `onModelChange` which should be invoked by the widget's author whenever the model should be changed;
- `onPollPeriodChange` which should be invoked by the widget's author whenever the poll period should be changed;

This mixins supports multiple attributes, when a string array is passed to the `onModelChange` method. The polling is started when this method is called for the first time. An example of how to use this mixin is presented later.

- `export function withReflectedAttribute (descriptor)`

Adds a reflected attribute to the widget. It is a widely adopted convention that it should be possible to configure HTML elements using only DOM attributes and pure HTML. However, it may be inconvenient to use the `getAttribute/setAttribute` functions and manually converting their arguments in application's code. Instead, attributes may be reflected as Javascript properties. This mixin adds a property that is backed by an attribute. The binding is bidirectional. Any conversions are performed automatically on the fly.

The mixin is configured via a `descriptor` argument that should have following the properties:

- `attributeName` - a `string` with the name of the attribute;
- `reflectedName` - a `string` with the name of the reflected property;
- `type` - a type of the attribute; `string` or constructor function;
- `defaultValue` - optional default value, returned when the attribute is not present;
- `onChange` - a callback function, called whenever the attribute value is changed.

B.2. Widget's behavior

There are two types of external triggers that alter widget's state: lifecycle callbacks and user's input. User actions are specific for each widget, e.g. user may click on a *button* or enter value in a *text-box*. Lifecycle callbacks are part of the *Custom Elements* standard. Their detailed description is provided in Appendix D. Fig. B.1 shows possible widget's states and transitions between them as a result of the above mentioned triggers.

Since TangoJS widgets are built with Web Components, they behave like ordinary HTML elements. During initialization widget prepares its layout, attaches event handlers and (usually) starts the polling timer to periodically update its state. Update of widget's attributes may require reinitialization, recreation of the underlying `DeviceProxy` or restart of the polling timer. The sequence chart in Fig. B.2 shows these procedures and their interactions with other parts of TangoJS stack.

B.3. Building a basic widget

With the utilities described in first section, it is relatively easy to build a simple widget for TangoJS.

Widget structure. One may start with an empty HTML file and populate it with the contents shown on Lst. B.1. It is a standard HTML file with two elements, `template` and `script`. The `template` element is part of *HTML Templates* standard and allows for defining the reusable HTML blocks. This template is the widget's view, which will be rendered on the webpage. The `script` element will contain the widget's logic. It is

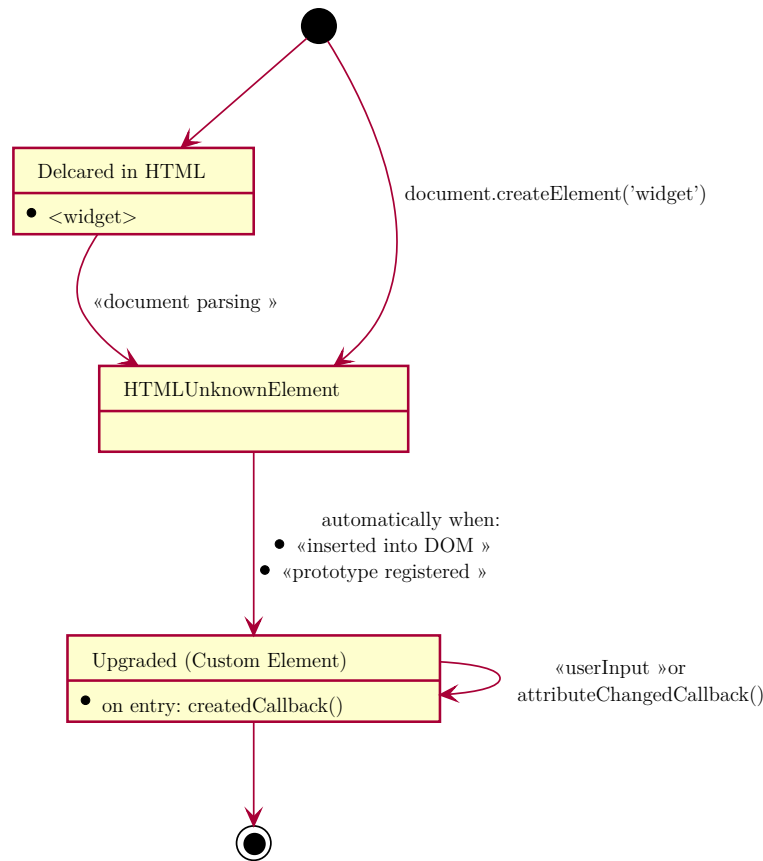


Figure B.1: Widget states and transitions.

wrapped in an IIFE and the `window`, `document` and `tangojs` global variables are injected into this function.

Populating the view. Any valid HTML can be placed in the `template` tag. Then, the template can be accessed from the Javascript code by querying the document, like `tangojs.web.util.getCurrentDocument().querySelector('template')`. Each widget instance clones this template and attaches it to its DOM tree. Alternatively, widgets may create layouts imperatively, using APIs like `document.createElement`.

The widget described here will be a simple label that displays the value of an attribute. A single `span` element will be enough to achieve this. This has been already appended to the `template` element. An `id` has been assigned to that element, but still there may be multiple instances of this widget in a single document, due to the encapsulation provided by the *Shadow DOM*.

The widget class. A widget is a class, or a prototype that extends `HTMLElement` and is registered in the browser as a custom element. The definition of this simple widget is shown on Lst. B.2. It is discussed in the following paragraphs. This code should be placed inside the `script` tag.

The widget usually sets its layout in the `createdCallback`. This method is invoked

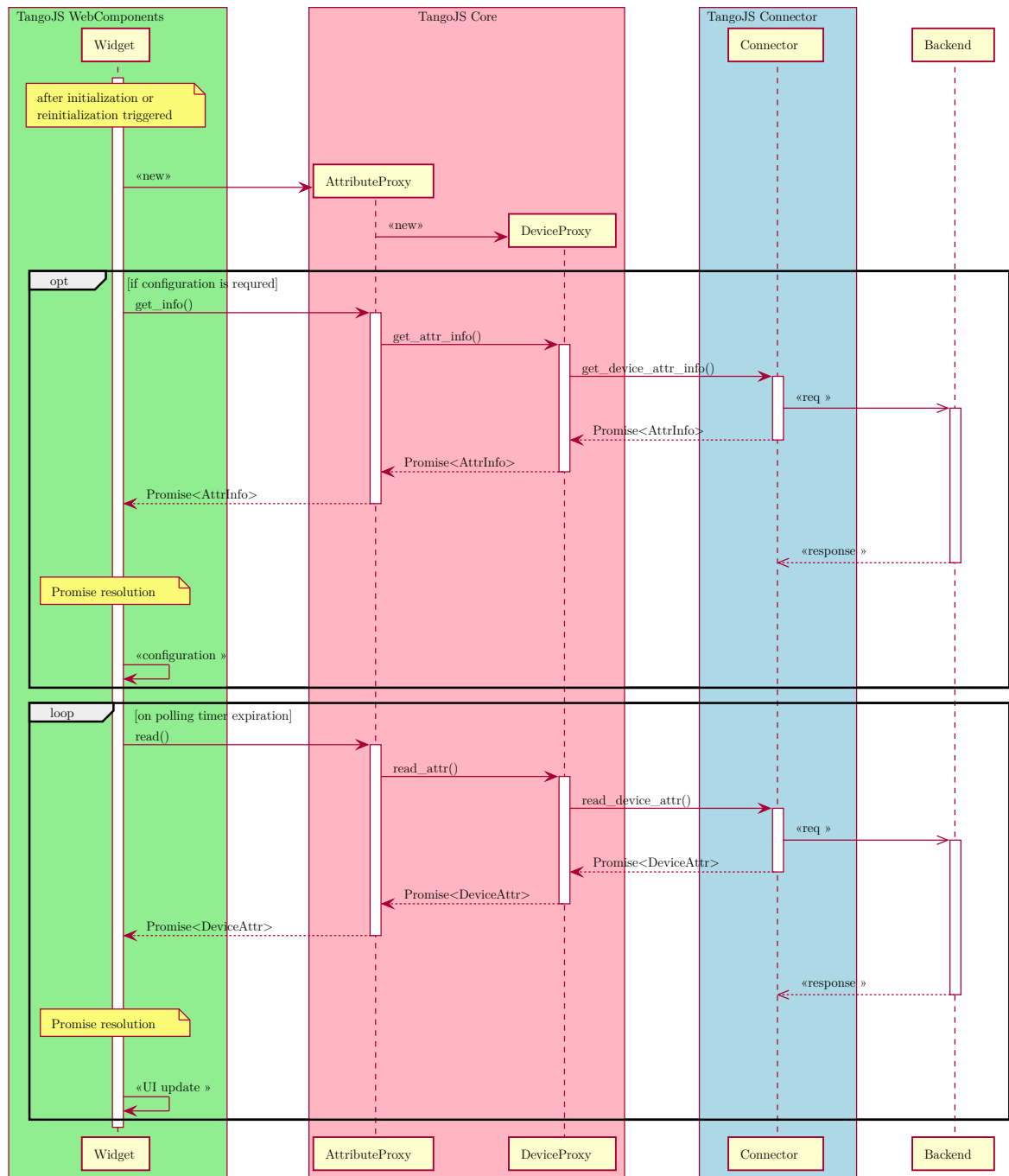


Figure B.2: Interactions of a widget with other modules during its lifecycle. Message details omitted.

Listing B.1 Empty widget file.

```
1 <!DOCTYPE html>
2 <html>
3
4   <template>
5     <span id="label"></span>
6   </template>
7
8   <script type="text/javascript">
9     (function (window, document, tangojs) {
10       'use strict'
11     })(window, window.document, window.tangojs)
12   </script>
13
14 </html>
```

whenever the widget is *upgraded* from `HTMLUnknownElement`. In the example implementation, the template is cloned and attached to the shadow root¹.

Polling the model. To poll the device and be notified whenever a new value is obtained, the `withPolledModel` mixin may be used. This requires implementing the four methods mentioned below. The `createProxy` method parses the model and instantiates an `AttributeProxy` from the *TangoJS Core* package. The `onModelRead` method is called with a map of models, but this widget expects only one model, stored in the `model` property. The `onModelRead` and `onErrorRead` methods are responsible for updating the widget's view periodically. In these methods, the label's text is set to the obtained value. This mixin is invoked on the widget's prototype, immediately after class definition.

Reflecting the attributes. The widget has two attributes, a `model` and a `poll-period`. The first one is a string that represents TangoJS model. The second one is a number that denotes how frequently the model is polled. These attributes should be reflected into the `model` and `pollPeriod` properties of the widget class. The `withReflectedAttribute` mixin can be used to create such properties. The `onChange` callbacks are used to invoke methods added by the `withPolledModel` mixin.

Registering the widget. The last step is to register the widget. The `registerComponent` utility function can be used. This function registers the widget in the current document and attaches its constructor to the `tangojs.web.components` object. It should be called after all mixins have been applied.

Using the widget. The widget is now ready. It may be included in any web application that uses TangoJS. Putting the widget in an HTML file allows it to be loaded using the *HTML imports*. The widget behaves like an ordinary HTML element.

¹This example uses the *Shadow DOM V0* API, which has been deprecated, but no browser implements the V1 API yet.

Listing B.2 Simple *label*-like widget prototype implementation.

```

1  const template = tangojs.web.util.getCurrentDocument()
2      .querySelector('template')
3
4  class SimpleTangoLabel extends window.HTMLElement {
5      createdCallback () {
6          const clone = document.importNode(template.content, true)
7          const root = this.createShadowRoot()
8          root.appendChild(clone)
9          this.appendChild(root)
10         this._label = clone.querySelector('#label')
11     }
12     createProxy (model) {
13         const [_ , devname, name] = model.match(/^(.+)\s/([A-Za-z0-9_]+)$/ )
14         return new tangojs.core.api.AttributeProxy(devname, name)
15     }
16     readProxy (proxy) {
17         return proxy.read()
18     }
19     onModelRead (deviceAttributes) {
20         const attribute = deviceAttributes[this.model]
21         this._label.textContent = attribute.value
22     }
23     onModelError (error) {
24         this._label.textContent = '-error-'
25     }
26 }
27
28 tangojs.web.util.mixins.withPolledModel.call(SimpleTangoLabel.prototype)
29
30 tangojs.web.util.mixins.withReflectedAttribute.call(
31     SimpleTangoLabel.prototype, {
32         attributeName: 'model',
33         reflectedName: 'model',
34         type: 'string',
35         onChange: SimpleTangoLabelElement.prototype.onModelChange
36     })
37
38 tangojs.web.util.mixins.withReflectedAttribute.call(
39     SimpleTangoLabel.prototype, {
40         attributeName: 'poll-period',
41         reflectedName: 'pollPeriod',
42         type: 'string',
43         onChange: SimpleTangoLabelElement.prototype.onPollPeriodChange
44     })
45
46 tangojs.web.util.registerComponent('simple-tango-label', SimpleTangoLabel)

```

Appendix C

CORBA IDL to Javascript translation

One of TangoJS's goals was to bring the standard TANGO data types to the browser. These types represent abstractions that TANGO developers are already familiar with. The use of standard structures and enumerations will allow for easier integration with other TANGO technologies in the future.

TANGO IDL. All data types in TANGO are defined in the *IDL* file. The *IDL*, *Interface Description Language*, is a specification language describes interfaces and data types of a system. The purpose of using IDL for interface description is to provide the *language independence*. The components of the system can be written in different programming languages, but still they are able to communicate. The IDL is used to *generate interfaces in concrete languages*, and then the *middleware* carries the messages between components. This middleware is meant to take care of message serialization and deserialization.

TANGO uses the *OMG IDL*, which is the standard IDL in CORBA. The middleware that connects the components in the system is CORBA itself.

Code generation. There are tools that parse the IDLs and generate *stubs* and *skeletons* for both the client and the server. This can be done e.g. for C++ or Java. Since Javascript is not supported by CORBA, there are no tools that can generate Javascript from IDL. To address this, we have developed the *idl2js*¹ tool.

idl2js. The *idl2js* does not aim to be a Javascript equivalent of the existing tools that generate *stubs* and the marshalling code. This is not possible, since CORBA cannot run in a web browser. All what TangoJS needs is to have the TANGO structures, enumerations and interfaces translated to the Javascript. The mapping is relatively simple:

- a **struct** becomes a **class** with properties reflecting the fields;
- an **enum** becomes a *frozen* object with properties reflecting the enumerated values;
- an **interface** becomes a **class** with empty methods;

¹<https://github.com/mliszczyk/idl2js>

- a `typedef` becomes an ESDoc `@typedef` comment;

For each generated entity a docstring comment is also generated. The comments use the syntax supported by the JSDoc and ESDoc tools. These comment strings can be used to generate the API documentation. Also, static typecheckers or code intelligence tools can use such comments as a source of information about data types. The example output of generated Javascript code is shown in Lst. C.1.

Listing C.1 Example structure translated from IDL to Javascript.

```

1  /**
2   * @public
3   */
4  export class DevAttrHistory {
5
6      /** @param {Object} data */
7      constructor(data = {}) {
8          /** @private */
9          this._data = Object.assign({}, data)
10     }
11
12     /** @type {boolean} */
13     get attr_failed() {
14         return this._data.attr_failed
15     }
16
17     /** @type {AttributeValue} */
18     get value() {
19         return this._data.value
20     }
21
22     /** @type {DevError[]} */
23     get errors() {
24         return this._data.errors
25     }
26 }
```

Translation. The translator has been written in Scala language. It uses the `IDLLexer`, `IDLParser` and `IDLVisitor` classes from the Apache Axis2 [50] project to generate the internal representation of the parsed IDL. Then, each type is transformed to its Javascript equivalent, as discussed above.

Using the `idl2js`. The `idl2js` is a simple, command line utility, that takes a single argument, the path to the *OMG IDL* file and prints generated Javascript to standard output. Assuming that SBT is used to build the project, it can be used this way:

```

1  $ sbt compile pack
2  $ ./target/pack/bin/main ./idl/tango.idl > ./generated.js
```


Appendix D

Selected Aspects of Implementation

This chapter discusses the web technologies used by TangoJS. The focus is put on the frontend-related aspects, like layout, styling and componentization. The benefits of each technology are highlighted together with an explanation why it has been chosen over the competing solutions.

D.1. Web Components

The *Web Components* [51] standard is the most important technology used by TangoJS. It is a set of four independent standards that emerged recently to satisfy the growing need for componentization and reusability in today's web. The specification is open and maintained by W3C. Although the standardization process of Web Components is currently in progress, the vendors have already implemented these features in their browsers. For older browsers the community have developed a set of polyfills [52]. The mentioned four standards are discussed next.

HTML Templates. The specification for HTML Templates [53] has been already finished and it has been included in the HTML5 standard. This feature adds the new `template` element to the set of HTML tags. The purpose of the `template` element is to hold a reusable fragment of HTML that can be later cloned and inserted into the document. To achieve the same goal without use of the *HTML Templates* one have to invoke `document.createElement` and `element.setAttribute` multiple times, which may be inconvenient, especially for creating more complex views consisting of multiple elements. The `template` is never rendered by the browser, but it can be accessed programmatically, using APIs like `document.querySelector`. The `content` property of a `template` is a `DocumentFragment` that represents all its children. An example of cloning the template and inserting the clone into a document as shown on Lst. D.1.

Custom Elements. The *Custom Elements* [54] standard provides a way to extend the HTML with user-defined custom tags. These tags behave like ordinary HTML elements, e.g. `input`, and can be created both declaratively in HTML and using the

Listing D.1 Using the HTML Template.

```

1 <body>
2
3   <template>
4     <span>Hello</span>
5     <span>World</span>
6   </template>
7
8   <script>
9     (function (document) {
10       const template = document.querySelector('template')
11       const body = document.querySelector('body')
12
13       body.appendChild(template.cloneNode(true))
14       body.appendChild(template.cloneNode(true))
15     })(window.document)
16   </script>
17
18 </body>

```

`document.createElement` in Javascript. Any prototype that extends `HTMLElement`, directly or not, may be *registered* in the browser, like:

```
1 document.registerElement('tag-name', { prototype: tagPrototype })
```

There is a restriction that the name of a custom element must contain a dash. The prototype passed in the second argument may define four lifecycle callbacks that are invoked by the browser on certain conditions:

- `createdCallback()` is invoked when the element is *upgraded*, that means the HTML parser has encountered the tag and the element has been already registered. Otherwise, the tag is rendered as a `HTMLUnknownElement`;
- `attachedCallback()` is invoked whenever the element is appended to the DOM;
- `detachedCallback()` is invoked whenever the element is removed from the DOM;
- `attributeChangedCallback(name, oldVal, newVal)` is invoked whenever an attribute on this element is added, changed or removed.

The custom element may also extend any of the existing HTML elements. The base prototype should be in an element's prototype chain and the `extends` property has to be added to the second argument of the `document.registerElement`, like `{ ..., extends: 'button' }`.

The above callbacks cover the whole lifecycle of an element. This allows implementing even complex use cases. A custom element may be used as a widget, that is seen by the users as a single entity. A simple widget shown in Lst. D.2 creates its layout programmatically. The real widgets often use *HTML Templates* for this task.

Listing D.2 Creating a Custom Element.

```

1 class ReverseWidget extends HTMLElement {
2   createdCallback () {
3     const input = document.createElement('input')
4     const button = document.createElement('button')
5
6     this.appendChild(input)
7     this.appendChild(button)
8
9     button.innerHTML('reverse!')
10    button.addEventListener('click', () => {
11      input.value = input.value.split('').reverse().join('')
12    })
13  }
14 }
15
16 document.registerElement('reverse-widget', {
17   prototype: ReverseWidget.prototype
18 })

```

HTML Imports. The *HTML Imports* [55] specification describes how HTML documents (*imported documents*) can be included from another documents (*import referrers*). The link element has been extended to support *HTML Imports*. Its `rel` attribute can be also `import`:

```

1 <link rel="import" href="imported-document.html">

```

The external document is asynchronously fetched and parsed when such a `link` is encountered. When accessed from Javascript, the `import` property of a `link` element returns the imported document, of type `Document`.

The *HTML Imports* is a natural way of distributing the self-contained widgets that consist of HTML, CSS and Javascript code, packed into a single file. The scripts inside the imported document are evaluated when the document is being parsed. This allows custom elements, like widgets, to register themselves in the browser.

Shadow DOM. The *Shadow DOM* [56] is the most complex of the *Web Components* standards. It brings a concept of *shadow tree* to the DOM. The existing DOM trees are called *light DOM* from now on. With each HTML element there may be multiple `ShadowRoot` elements associated. A *shadow tree* may be attached to each `ShadowRoot`. A `ShadowRoot` node is the root of all nodes in the corresponding *shadow tree*, like `document` is the root of all nodes in the DOM.

During rendering, the shadow trees are merged with the element they are attached to. However, these trees are not part of the DOM. Nodes in a shadow tree cannot be accessed by querying the `document`. Also, they are not affected by the stylesheets other

than `style` elements inside the shadow tree¹. This encapsulation may be desired in some cases, e.g. the contents of a custom element may be put inside a shadow tree to create a widget that appears to the user as a single entity that has no internal structure. The widget will always look the same, no matter what stylesheets are loaded. External code cannot modify widget's behavior accidentally. An example that creates a shadow root is shown in Lst. D.3.

Listing D.3 Creating a shadow tree.

```
1  const outer = document.createElement('div')
2  const inner = document.createElement('div')
3
4  // using the Shadow DOM V0 API
5  const root = outer.createShadowRoot()
6  root.appendChild(inner)
7
8  // using the Shadow DOM V1 API
9  const root = outer.attachShadow({ mode: 'open' })
10 root.appendChild(inner)
```

The *Shadow DOM* also allows for creating *slots*, i.e the elements in a shadow tree where another elements may be inserted. This feature is however not used by TangoJS.

Component-based development. The *Web Components* standard brings in a new set of possibilities to web development. The *components* may be self-contained, reusable and highly encapsulated, thus writing componentized applications becomes more and more popular. These applications are created from basic building blocks, the components, which have a small set of responsibilities, are easy to create, easy to test, and easy to maintain. This is the opposite to building a large, unstructured application or using the MVC pattern where *view* is the whole page, managed by a single *controller*. In the component-based approach, the MVC pattern can be applied at single component level.

Alternatives. *Web Components* allow for building componentized applications without using third-party frameworks. This is often desirable, e.g. when building a widget library like TangoJS. The user can integrate such widgets with any framework, just like the standard `inputs` or `buttons`. If one wants to sacrifice the flexibility for nicer APIs, there are a few alternatives available, the frameworks like Angular 2 [41] and built on top of the *Web Components*: Google's Polymer [43], Mozilla's Brick [57] and Microsoft's X-Tag [58].

D.2. ECMAScript 2015

The Javascript development has been stalled for years. The 5th version of the standard has been published by the Ecma International, the standardization body behind Javascript, in 2009. Since that time, new features have not been added to the language until 2015.

¹The *shadow-piercing selectors* have been deprecated and no alternative has been proposed yet.

The rapid growth of web development popularity has forced the Ecma to speed up the standardization process. In June 2015, the ECMAScript 2015 specification has been published. It contained numerous improvements over the previous version and a set of new features. After the ECMAScript 2015 release, the development schedule has been changed. A new version of ECMAScript shall be expected every year.

This section discusses the new features introduced in the latest releases and extensively used by TangoJS.

Standard library. The list of standard prototypes has been extended, by adding new a collection classes, like `Map`, `Set`, `WeakMap`, `WeakSet` and utilities like `Promise`, `Symbol` and `Proxy`.

The `Map` and the `Set` are the standard collections known from other languages. Until now, an `Object` has been a common replacement for `Map`. However, `Object`'s keys are always converted to `strings`. The `Map` can be indexed with any type. In the `WeakMap` and `WeakSet` the objects can be garbage-collected when there are no references to the keys or entries.

The `Promise` is an abstract object that represents an asynchronous computation, which may succeed and resolve the promise, or may fail, rejecting the promise. This allows asynchronous functions to return a value, instead of accepting a callback as an argument. The promises are chainable which means that the result of a previous computation may be passed to the next one. The errors may safely flow through the chain and should be caught at the end. The example code is shown in Lst. D.4.

Listing D.4 An example use of ECMAScript 2015 Promises.

```

1 (new Promise((resolve, reject) => {
2   setTimeout(() => {
3     resolve(1)
4   }, 1000)
5 })))
6 .then(x => Promise.resolve(x + 1)) // promises may be returned
7 .then(x => x / 0)
8 .then(x => x * 2)
9 .then(x => console.log(`result: ${x}`))
10 .catch(e => console.error(`error: ${e}`))

```

Classes. The Javascript is an object-oriented language with a prototypical inheritance. This means that for each object there is also another object, its prototype. Each prototype has its own prototype. These relations form a prototype chain. There are multiple ways to define a *class* in Javascript. There are also multiple ways to make this class extend another class. The ECMAScript 2015 introduced one more method, with the `class` keyword. This method is the most readable one because it resembles constructs from class-based OOP languages like Java or C++. Class can define methods, properties and static attributes. Different ways of defining a class are shown in Lst. D.5 and Lst. D.6.

Listing D.5 Classes and inheritance in ES5.

```

1  const Base = function (x) {
2      this._x = x
3  }
4  Base.prototype.add = function (y) {
5      return this._x + y
6  }
7  const Derived = function (x) {
8      Base.call(this, x)
9  }
10 Derived.prototype = Object.create(Base.prototype);
11 Derived.prototype.constructor = Derived
12 Derived.prototype.mul = function (y) {
13     return this._x * y
14 }

```

Listing D.6 Classes and inheritance in ES2015.

```

1  class Base {
2      constructor (x) {
3          this._x = x
4      }
5      add (y) {
6          return this._x + y
7      }
8  }
9  class Derived extends Base {
10     constructor (x) {
11         super(x)
12     }
13     mul (y) {
14         return this._x * y
15     }
16 }

```

Arrow functions. The ECMAScript 2015 introduced a new kind of functions, called *arrow functions*. In a normal function, `this` may point to different things, depending on how the function has been called. In the *arrow functions* `this` is lexically scoped, which means that `this` from within a surrounding scope may be accessed without creating a variable for it. This is useful in, e.g. anonymous functions. An example of such a function is shown in Lst. D.7.

Modules. In Javascript a module is usually an object that holds another objects or classes. When a browser environment is considered, the modules are often attached to the global `window` object. This is different for, e.g. a server-side runtime, Node.js, where modules are loaded synchronously with the `require` function.

An important part of ECMAScript 2015 is the module specification. This is the first step to a truly modular Javascript applications. The standard specifies only the syntax

Listing D.7 Arrow functions in ES2015.

```

1  class Calculator {
2      constructor (x) {
3          this._x = x
4      }
5      addLater (y) {
6          return new Promise((resolve) => {
7              setTimeout(() => {
8                  resolve(this._x + y)
9              }, 1000)
10         })
11     }
12 }

```

used to export and import objects from modules. The *module loader* specification is still under development, but the new module syntax can be used already thanks to the tools like Rollup [59]. Rollup preprocesses Javascript sources and changes the **exports** and **imports** to the global variables or **require** calls, depending on the targeted environment.

A module in ECMAScript 2015 is a *file* that contains **import** and **export** statements. Any variable or constant may be exported. An example of such a module is shown in Lst. D.8.

Listing D.8 Modules in ES2015.

```

1  import functionA from './util/functions'
2  import * as helpers from './util/helpers'
3
4  export function functionB (x) {
5      return functionA(helpers.mul(x))
6  }

```

Other features. Apart from the major features described above, a few less important but useful improvements have been introduced in ECMAScript 2015. This includes *de-structuring* which allows to assign an object or array to an expected pattern, the default parameters, the template literals where variables can be interpolated, the block-scoped declarations like **let** and **const** and the short object initializers.

Alternative languages. Javascript is a dynamic and flexible language, which makes it easy to create an *x-to-javascript* transcompilers. A lot of new languages has been developed in recent years. The two most popular are *CoffeeScript* and *TypeScript*. Both are compiled to Javascript and there is no runtime overhead. The CoffeeScript aims to simplify Javascript's syntax with classes, arrow functions and template literals. These features are however available in pure Javascript since ECMAScript 2015. The TypeScript is a Javascript with optional typechecking.

D.3. CSS Level 3 modules

After the release of CSS2, the CSS specification has been divided into modules. Each module evolves independently, reaching new *levels*. Each builds upon a previous level. TangoJS uses modules described here to handle its layout.

CSS Custom Properties. The *CSS Custom Properties for Cascading Variables Module Level 1* [60] adds support for variables to the stylesheets. Variables may be defined on any element using the syntax `--name: value;`. Variables are cascading and are inherited by all child nodes. An example of a variable is shown in Lst. D.9.

Listing D.9 Using the CSS variables.

```
1 :root {
2   --main-color: #06c;
3   --accent-color: #006;
4 }
5 #foo h1 {
6   color: var(--main-color);
7 }
```

CSS Values. The *CSS Values and Units Module Level 3* [61] introduces two new functions, namely `calc` and `attr`. The first one allows calculating values for attributes dynamically. The second one allows referencing a value of a certain attribute. An example using this functions is shown on Lst. D.10.

Listing D.10 Using the CSS values.

```
1 section {
2   float: left;
3   margin: 1em; border: solid 1px;
4   width: calc(100%/3 - 2*1em - 2*1px);
5   height: attr(width em);
6 }
```

CSS Flexible Box. The *CSS Flexible Box Layout Module Level 1* [34] is a new `display` option for building containers filled with items that occupy all available space in horizontal or vertical direction. Each item gets space proportional to its `flex` attribute. On the orthogonal axis the elements may be aligned, centered or stretched. This gives the flexibility in positioning the elements and building complex user interfaces without a need for *floats*. The elements within a flexbox may be reordered using CSS attributes. A vertical flexbox where each item gets the same space is shown in Lst. D.11.

CSS Grid Layout. The *CSS Grid Layout Module Level 1* [35] allows specifying `grid` as a `display` option. An element which is displayed as a grid is divided into a number of rows and columns. Each of them may be assigned a `grid-column` or a `grid-row` attribute to position the item on a certain grid cell. Items spanning over multiple grid cells are allowed as well. This gives the developer an ultimate flexibility on how the items can be

Listing D.11 Using the CSS Flexbox.

```

1 .root {
2     display: flex;
3     flex-direction: column;
4 }
5 .root > * {
6     flex: 1;
7 }

```

aligned on the grid. Multiple areas can be defined on the grid. An area is a named set of adjacent cells. Child nodes may be assigned to these areas without a need for setting rows and columns. This is presented in Lst. D.12.

Listing D.12 Using the CSS Grid Layout.

```

1 #grid {
2     display: grid;
3     grid-template-areas: "header header"
4                          "menu    tabs"
5                          "data    data"
6                          "data    data";
7     grid-template-columns:
8         auto minmax(min-content, 1fr);
9     grid-template-rows:
10         auto auto minmax(min-content, 1fr) auto
11 }
12
13 #title { grid-area: header }
14
15 #board { grid-column: 1 / span 2; grid-row: 3 / span 1;}

```

D.4. Limitations and browser support

Most technologies described in this Appendix are undergoing the standardization process. They are often implemented in the major browsers, but are disabled and hidden behind various flags. This is because these features have not been fully tested yet and APIs may slightly change during standardization. All the features from this Appendix have been verified to work on Mozilla Firefox 46 and Google Chrome 50. For Mozilla Firefox, the flags `dom.webcomponents.enabled` and `layout.css.grid.enabled` should be set to `true` in *about:config* tab. In Google Chrome, the *Experimental Web Platform features* should be enabled in the *chrome://flags* tab.