WebUI - rapid UI development for EGS-CC with modern web technologies.

Workshop on Simulation and EGSE for Space Programmes (SESP)
26 - 28 March 2019

ESA-ESTEC, Noordwijk, The Netherlands

Dariusz Walczak(1), Marek Młodkowski(1), Jakob Livschitz(2), Mateusz Głowacki(1), Łukasz Nojman(1), Aleksander Piotrowski(1)

(1) Bit by Bit Sp. z o.o. (Recoded)
ul. Za Bramka 9/10
61-842, Poznań, Poland
dariusz.walczak@thebitbybit.com

(2) ESTEC – European Space Research and Technology Centre
Keplerlaan 1
2201 AZ, Noordwijk, The Netherlands
Jakob.Livschitz@esa.int

INTRODUCTION

The paper presents a web-based user interface library, WebUI, supporting the rapid development of applications for the space sector. Possibilities of integration with existing ESA systems is discussed based on EGS-CC example. Finally, an end-to-end example will be described to demonstrate the whole concept.

The main motivation behind WebUI was to offer a lightweight and flexible alternative to the Eclipse-based EGS-CC thick client. To achieve this goal, the client was implemented using web technologies, which are characterized by very low cost in terms of development time.

We have prepared a set of React components useful for the space sector, i.e.: alphanumeric display, time series chart, out-of-limits display, etc. All of them are prepared to process fast-changing data and communicate with backend using the WebSocket. The selected protocol makes it possible to open a two-way interactive sessions between user's browser and a server. The web application can be easily notified if any change occurs on the backend service. The well known and widely used observer pattern can exist between server and client. The paper will describe the integration between OSGi based EGS-CC system and web-based client application. The idea of generic API, which is a thin backend service that enables web developers to call any OSGi method via web sockets will be presented. We are using a custom designed JSON language and Java Reflections to process API calls. The backend library keeps state objects, i.e.: session or subscription.

MOTIVATION

Recently, web applications became more advanced and related technologies had to adapt and mature. The number of frameworks and tools supporting web development is growing. With the advent of transpilation, developers can use modern language features introduced in new versions of JavaScript, add static type verification using Flow, while still producing apps compatible with all major web browsers on the market. Tools for building user interfaces also became more powerful and scalable with the introduction of HTML5 and CSS preprocessors like Sass. Today's browsers support multiple low-level APIs enabling developers to use local storage, camera, microphone etc. With all these technologies web developers can provide fully functional applications.

Despite technological advances, web technologies offers some organizational benefits as well. The obvious one is **easier maintenance**. Each update of the application can be immediately deployed to the users without their actions. No need to upgrade each PC in the organization. The same version of the applications run on multiple platforms, there is no need to prepare packages for multiple operating systems. With the responsive design, everyone can prepare an application that fits into different screen sizes and run in on every mobile devices.

Web applications provide **better security** than a desktop one. The usual app is deployed on dedicated servers, monitored and secured by experienced server administrators. It is more efficient than securing all workstations in the organizations. The data, which are processed by web apps, are usually sent over the network encrypted. Only authorized users have access to them. Short sessions allow to automatically log out the user if he/she is not in front of the computer. This is very important in case of a stolen or damaged hardware. With the traditional application, an organization can lose valuable data and software recovery can be a very costly and time-consuming situation requiring you to contact your software provider and request for the software to be re-installed on a new device. With a web application, a user can just login from a different machine and start working, data can be quickly restored from the cloud.

The web application can be **adaptable to increase workload**. In case of required additional processor time or storage, only server machine has to be adapted. With current clustering, virtualization and load balancing technologies this can be an automatic process, completely transparent for user. As workload increases, new servers can be added. If a server fails the web application can be easily moved to another one, also automatically. If a desktop application requires more power to perform tasks the hardware has to be upgraded with requires significant effort in terms of money and time.

Finally, web applications offer **lower development costs** due to environment, simplicity and labor cost. The uniform environment, lets the developer write the application once. Although the differences between browsers require testers to repeat the tests on each of them, the application itself needs only be developed for a single "operating system". Usually, the architecture of the application is much simpler, mainly due to the limited usage of multithreading. The trends on the labour market have to be taken under consideration as a very important part of software project budgets. The popularity of the web applications moves programmers to this segment. The number of job offers in web development is much higher than for desktop. On the other hand, due to the popularity of these technologies, the salaries for web programmers are lower than for desktop. Notice the charts below.

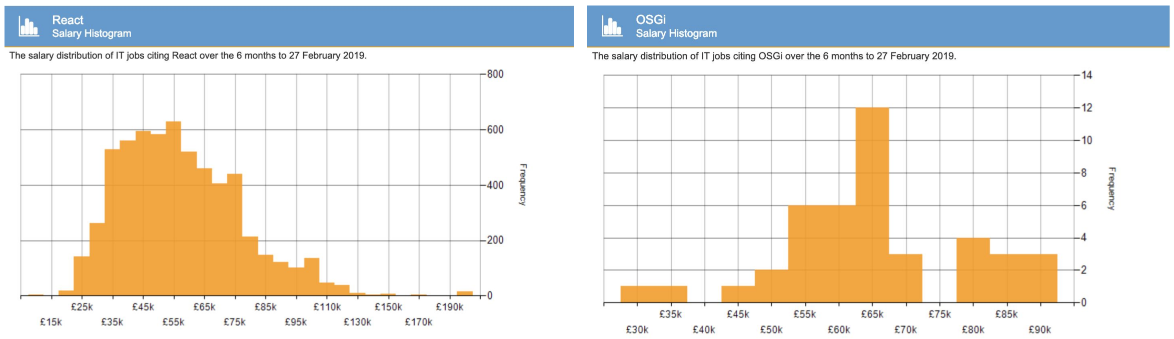


Fig 1. Salary distribution for React and OSGi technologies according to itjobswatch.co.uk, 2019-03-10.

Despite of all benefits, there are some issues as well. Web application requires network availability. It is possible to prepare a web application that works offline but still, most of the application requires network connectivity to access data on the server. Support for different browsers and versions are required although each year the compatibility is increasing. Some frameworks try to handle those differences and offer users a consistent API for each environment.

WEB FRAMEWORKS

Among available web frameworks, we decided to use React. It is designed by Facebook for creating rich and fast web applications with minimal coding. Two main key features make it suitable for the space sector:

- the best possible rendering performance with mature architecture,
- focus on building reusable, encapsulated components.

The components can create hierarchical structures and pass properties among each other, thus providing an easy way of composing them to make complex applications. To help manage the state of the whole application, an additional state container can be used to provide a single source of truth when it comes to the application state. In our application, we used Redux, a predictable state container for JavaScript apps. Unidirectional data flow found in Redux combined with the declarative nature of React components makes applications more predictable and easier to debug.

We chose React among other competitors like Angular2 and VueJS because of maturity, stability and popularity. React is a framework with a focus on stability. Breaking changes appear only between major releases, which rarely happen. Two last major releases took place in September 2017 and April 2016. Moreover, creators of the framework always provide an easy to follow guide on how to update product code easily between major versions, many times they provide automated scripts to upgrade component syntax. Furthermore, React creators believe in Gradual Update via Warnings:

before adding any breaking changes, they provide helpful warnings for developers with informations about planned depreciation of features. These warnings will not affect production build and give us time to update our code.

Maturity and stability of the Javascript framework are important aspects. Table below list number of releases of each of the most popular framework. React on average was updated almost 19 times a year. For a long term project this is a large number and forces developers for regular updates. For VueJS and Angular2 we have 49 and 60 releases a year. Significant amount of time is required to keep the framework or application up to date. This is why we consider the smaller number of releases as an indication of maturity.

Table 1. Framework statistics, downloads between 2019-02-01 and 2019-02-03

Framework	Initial release	Releases	Contributors	Stars*	Forks	Downloads from npm registry
React	May 29, 2013	**111**	**1285**	123,394	**22,393**	**22,922,719**
VueJS	February 2014	245	263	**129,228**	18,431	3,448,079
Angular2	September 14, 2016	301	858	45,717	11,936	8,589,675

* Staring methodology available here: https://help.github.com/en/articles/about-stars

React has the **biggest number of contributors** out of these three frameworks. There are a lot of **production-tested** packages to work with React such as *react-redux* which helps to integrate with previously mentioned Redux. The framework is also the undisputed leader in the number of downloads.

PROJECT ARCHITECTURE

Our main goal was to develop a library of React widgets which can be reused by an EGS-CC AIT system to build up a user interface. The widgets can create hierarchical structures and pass properties among each other, thus providing an easy way of composing them to make complex applications or advanced dashboards.

In order to make the development easier, we have provided the Javascript SDK, called later the library. It adds a state container, based on Redux, to manage the behaviour of the application and communication with the server. It provides a single source of truth when it comes to application. Unidirectional data flow found in Redux combined with the declarative nature of React components make applications more predictable and easier to debug. The SDK speeds up the development as the user does not have to take care of the communication with the EGS-CC Kernel. In order to achieve this we had to build OSGi bundle that expose proper API for our SDK – OSGi-JS bridge.

Finally, the solution consists of 5 main parts:

- WebUI - the components library - UI controls that can be easy to import and use in any React application.
- SDK, the library - Javascript application based on Redux. The application can communicate with EGS-CC via HTTP protocol.
- OSGi-JS Bridge - osgi bundle with HTTP and WebSocket APIs for EGS-CC.
- Demo application - demo application that shows the full potential of the library.
- Usage examples - series of simple applications that shows the possibility of integration with every component.

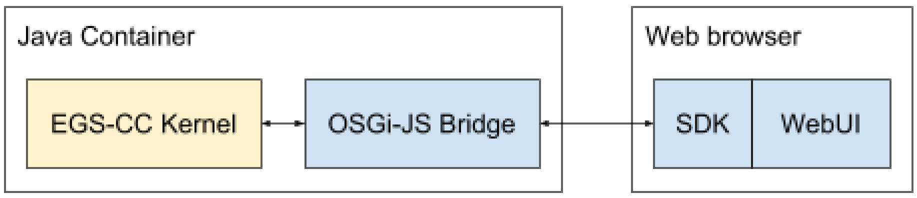


Fig. 2. Solution architecture.

COMPONENTS - WebUI

The WebUI is a set of React components grouped into displays, which provides set of functionalities, they presents time series, values in tables etc. Each display can be configured via user interface or a developer in the source code of final applications. Adjusting colors, fonts, background requires updating CSS files only. We have provided two versions of the controls styling, one based on ESA Corporate Visual Identity Manual and an alternative, dark version, inspired by The Martian movie.

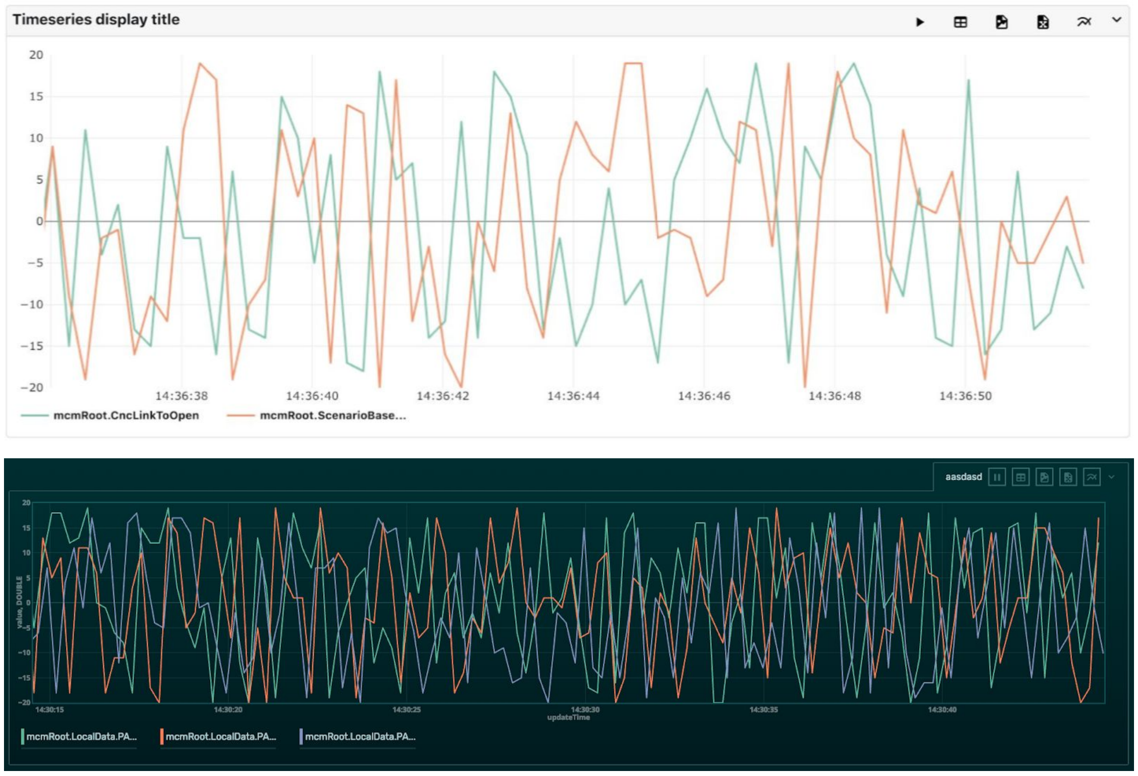


Fig. 3. Time Series display with two different styling files.

At the moment of writing the article the library consists of following displays:
* MCM Tree - shows EGS-CC M&C Tree, its' filters, state of the nodes etc.
* Time Series - as shown above, a display for presenting time series data
* Alphanumeric - generic component for displaying tabular data
* Log - shows list of log information
* State Transition History - shows different states in time
* Raw Data - displays data packets in different formats
* Activities Log - shows state of the activity invocation
* Terminal Window - enables users to use Karaf webconsole feature

The library provides also a dialog which enables users to provide parameters values for activity invocations. Finally we have developed a Synoptic Display which aggregates multiple displays on the common background image.

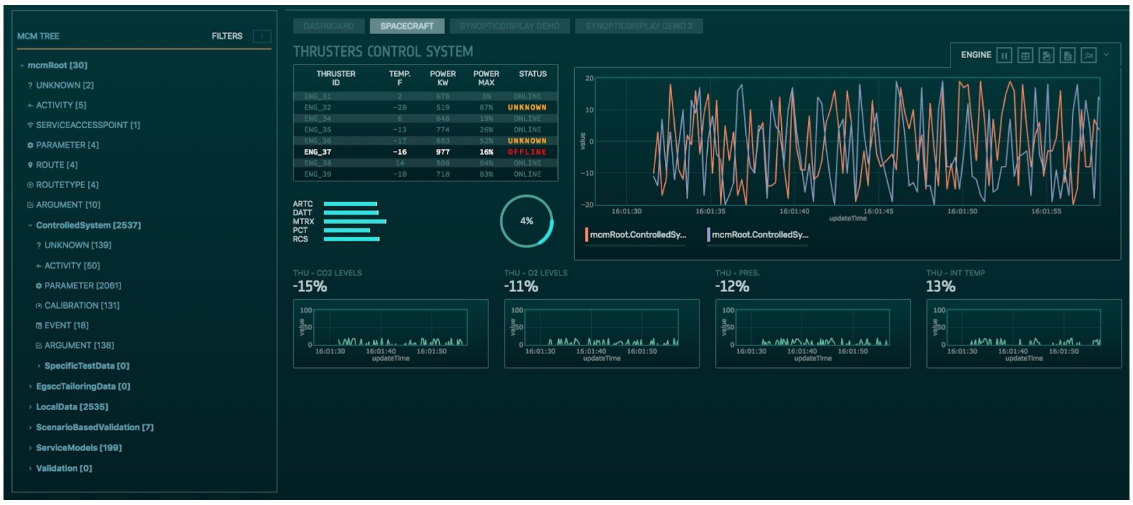
In order to evaluate the library we have built the demo application that enable user to use all the provided displays. The user can login to the system, browse M&C tree, subscribe for parameter updates or invoke activities. Parameters can be visualized on any of the available displays.

We are working on the evaluation of the framework. We are conducting stress test and long term running tests. Below we show first results from tests conducted on MacBook Pro (Retina, 13-inch, Late 2013), with 2,4 GHz Intel Core i5 CPU and 8 GB 1600 MHz DDR3. We have used Chrome Web Browser, version: 72.0.3626.119 (official, 64-bit).

The final application package size was 28,6 MB but client has to download:
- 73.1 KB of images,
- 1.3 MB of Javascript code,
- 48 KB of CSS files.

Total Page Transfer Size reported by developer tools was: 1,5 MB.

First test was conducted with dedicated data generator. The parameter updates were generated every 250 ms. Subscription has been done in an iterative manner with random pauses between each parameter. The CPU usage depends on the type of the display. Alphanumeric display is significantly faster than the time series. The table updates only rows containing the updated values. For the chart, each point on the screen has to be redrawn. For our test setup 50 subscriptions generated more than 80% CPU usage. This is definitely not the limit for the browser - the chart is a SVG file that is updated after receiving new value from the server. Modern browsers also offer drawing charts with hardware acceleration via OpenGL, this approach will be investigated in the future.



THRUSTER ID	TEMP. F	POWER KW	POWER MAX	STATUS
ENG_31	2	678	3%	ONLINE
ENG_32	-20	519	87%	UNKNOWN
ENG_34	6	648	19%	ONLINE
ENG_35	-13	774	26%	ONLINE
ENG_36	-17	683	52%	UNKNOWN
ENG_37	-16	977	16%	OFFLINE
ENG_38	14	508	84%	ONLINE
ENG_39	-10	718	83%	ONLINE

Fig. 4. Demo application.

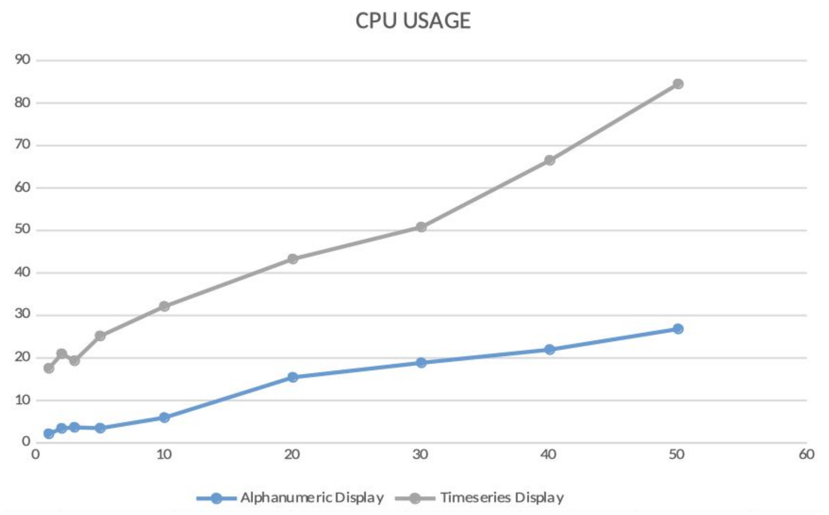


Fig. 5. CPU Usage based on the number of subscribed parameters.

LIBRARY - OSGi-JS bridge

The goal of OSGi-JS bridge is to enable connection between EGS-CC Kernel and React application. We had to build a communication layer that is acceptable by the browser and supports two ways of communication: stateless and stateful. For the first on we choose REST and for the second WebSockets.

REST, Representational State Transfer, is a software architectural style that provides interoperability between computer systems on the Internet. RESTful web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations. In a RESTful web service, requests made to a resource's URI will elicit a response with a payload formatted in either HTML, XML or JSON, in this case the last one. By using a stateless protocol and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow, by re-using components that can be managed and updated without affecting the system as a whole, even while it is running.

However, EGS-CC use also stateful communication and not all operations can be transformed to stateless REST commands. Especially asynchronous operations like sending state updates are not compatible with traditional web communication style. To avoid long polling or constant requests to the server to check the state of the operations or availability of new update we chose to use WebSocket technology.

WebSocket is a transport protocol defined by a persistent, bi-directional communication channel between server and client that takes place over a single TCP socket connection. Because a WebSocket connection is held open for the duration of a session, messages can flow back and forth between participating endpoints with little overhead and low latency. To initiate a WebSocket connection, the client and server must negotiate a handshake via HTTP, and from that point onwards, asynchronous WebSocket rules apply. Specifically, bulky HTTP headers are replaced with message frames only a few bytes in size, and both the server and client can simultaneously send new data without the other asking for it.

WebSocket is fully HTML5-compliant. It eliminates the need for client-server polling and is especially useful for any JavaScript-based application that processes data frequently, asynchronously, and in real time.

GENERIC API

The Generic API is a part of OSGi-JS bridge. The idea behind it is to use Java Reflections and provide simple but generic operations to the user via REST/WebSocket API. Such an interface gives more flexibility and can be used in theory throughout the EGS-CC to interact with any component. Unfortunately it is significantly more complex on the client side. The developer has to know both, the OSGi-JS Bridge API and EGS-CC API to make any call.

We have designed a set of instructions for calling the EGS-CC service. One single API call can consist of several steps related to getting the factory, parameters preparation, listeners creation and at the end calling object methods. Each step can store results in the context of the user session within the plugin so objects created in one step can be easily used in the next step.

In order to call the Generic API one has to send "execute" request via WebSocket:
```
{
	"type": "execute",
	"instructions": []
}
```

Field called "instructions" holds the list of JSON objects that represents each step of the call. There are multiple type of instructions available: osgi, create, set, proxy, invoke. Below we describe some of them:

- Accessing OSGi factories [osgi] - EGS-CC uses Factory design pattern. Through factories user gets access to the services, that is why there is a dedicated instruction type for getting the factory:

  ```
  {
  	"type": "osgi",
  	"className": "...",
  	"filter": "..."
  	"result": "..."
  }
  ```
  where:
  - className - fully qualified name of the factory class,
  - filter - additional filter that can be applied,
  - result - variable name under which the factory object will be stored.

- Command for object creation [create] - the parameters required by EGS-CC API are usually defined by interfaces or abstract classes. This makes it impossible to deserialize objects from the browser as serializers do not know which implementation to use. We have provided a dedicated instruction that lets user create an object before invoking the method:

  ```
  {
      "type": "create",
      "className": "...",
  ```

```
    "result": "...",
    "value": {
        ...
    }
}
```
where:

- className - is fully qualified name of class to be created
- result - the variable name under which new object will be created
- value - serialized object fields values

- Creating a list of objects [set] - for some objects/methods a list of parameters is required. In that case one can use "set" instruction.

  ```
  {
  "type": "Set",
  "className": "...",
  "result": "...",
  "value": [...]

  }
  ```
  The only difference to the "create" instruction is the type and value field. Type is "set" and value is a list of objects, not a single object.

Critical feature that lets us develop the Generic API is java.lang.reflect.Proxy class. This is a dynamic proxy class that implements a list of interfaces specified at runtime when the class is created. What is even more important, each proxy instance has an associated invocation handler object which enable us to receive updates from the kernel and send it back to the browser. User can create this type of listener with "proxy" type of message:

```
{
        "type": "proxy",
        "className": "...."
        "name": "....",
        "messageType": "....",
}
```
where:

- className - is a fully qualified name of the interface
- name - is a variable name that will hold created interface for further processing
- messageType - is a label that will be used to annotated messages sent to the browser after calling interface method

With proxy listener one can invoke a method with "invoke" type:

```
{
        "type": "invoke",
        "variable": "...",
        "method": "...",
        "args": [...],
        "result": "..."
}
```
where:

- variable - is the name of the variable on which the method will be called
- method - a method that will be called on the variable object
- args - list of parameters
- results - the name of the variable for storing the result

For the most common operations we have provided the dedicated interfaces. This API is easier to use but the simplification requires a lot of effort during the implementation of each endpoint. Each API endpoint is developed with http service available in Karaf and Jersey library.

SUMMARY

Current state of HTML5/CSS3 and Javascript based technologies enables developers to provide advanced and robust web applications. Great popularity of those tools among developers and organizational benefits makes this technology very attractive in most of the industries including space. New approach gives possibilities to provide fast, light and easy to use applications that are easy to develop and maintain.

In this project we have demonstrated that it is possible to add an alternative user interface to the standard EGS-CC UI, smoothly integrated into the system due to the flexibility of the OSGi architecture despite using a completely different front-end technology. This opens enormous possibilities to customizations and tailoring, especially considering how much simpler the WebUI is, having at the same time much higher flexibility.