Luplink: Link Budget calculation inside JSatOrb

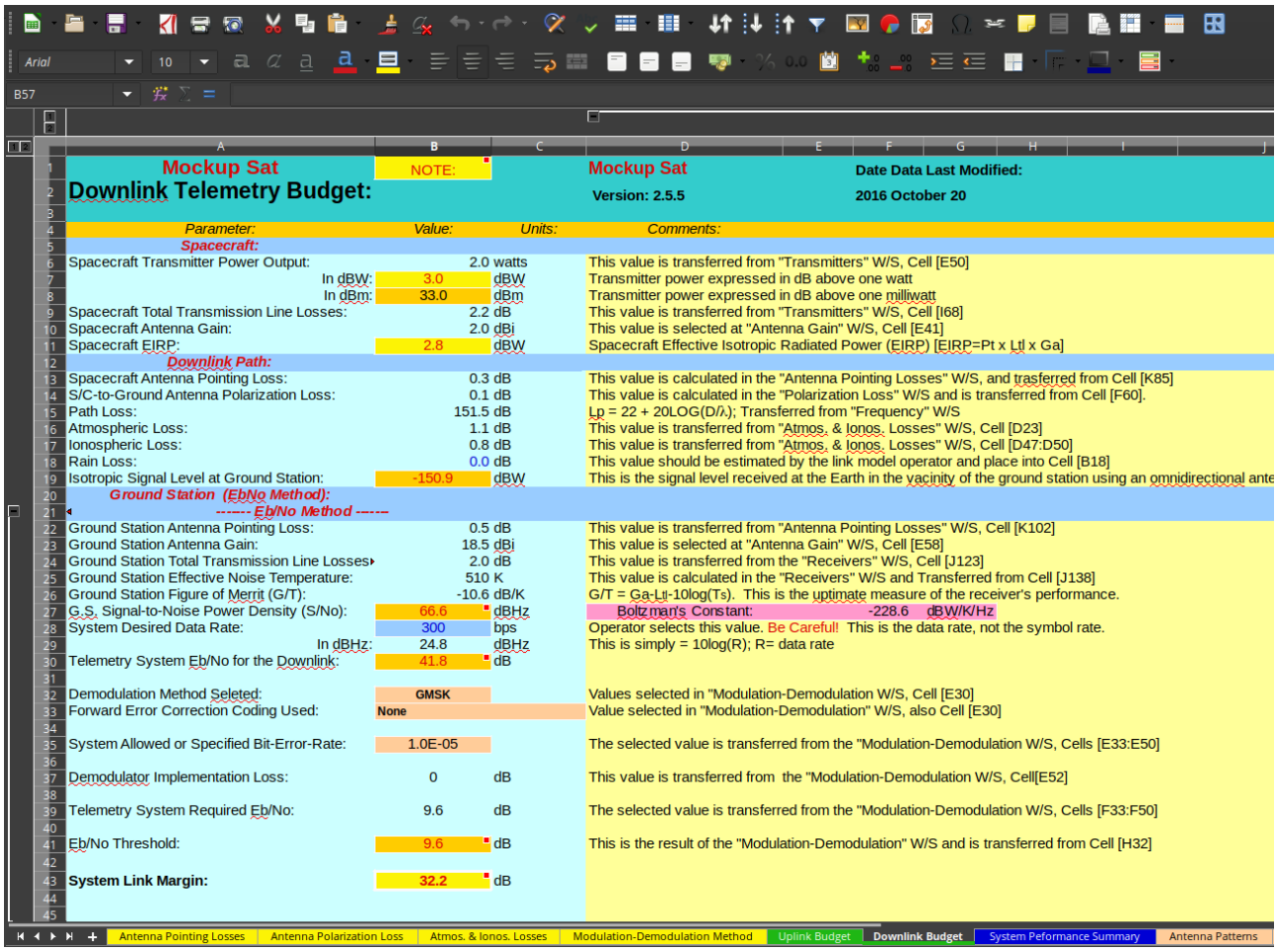
Introduction

The goal of this project is to build an interface for link budget calculation and integrate it inside JSatOrb. It should provide at least the same capabilities as the AMSAT sheet while making use of the added functionnalities provided by web applications.

Table of contents

Current tools

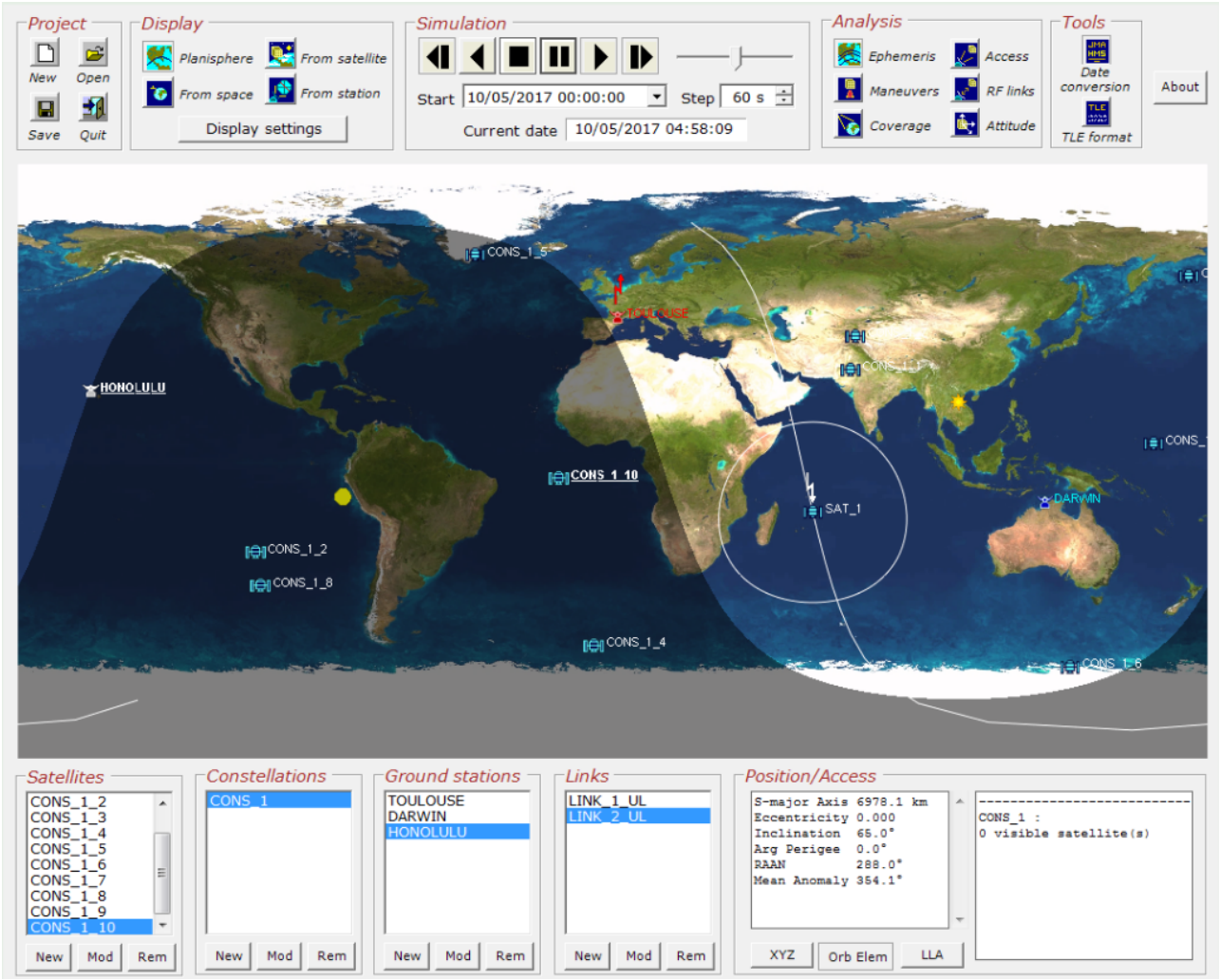
- AMSAT spreadsheet



	Mockup Sat	NOTE:		Mockup Sat	Date Data Last Modified:
	Downlink Telemetry Budget:			Version: 2.5.5	2016 October 20
	Parameter:	*Value:*	*Units:*	*Comments:*	
	Spacecraft:				
	Spacecraft Transmitter Power Output:	2.0	watts	This value is transferred from "Transmitters" W/S, Cell [E50]	
	In dBW:	3.0	dBW	Transmitter power expressed in dB above one watt	
	In dBm:	33.0	dBm	Transmitter power expressed in dB above one milliwatt	
	Spacecraft Total Transmission Line Losses:	2.2	dB	This value is transferred from "Transmitters" W/S, Cell [I68]	
	Spacecraft Antenna Gain:	2.0	dBi	This value is selected at "Antenna Gain" W/S, Cell [E41]	
	Spacecraft EIRP:	2.8	dBW	Spacecraft Effective Isotropic Radiated Power (EIRP) [EIRP=Pt x Ltl x Ga]	
	Downlink Path:				
	Spacecraft Antenna Pointing Loss:	0.3	dB	This value is calculated in the "Antenna Pointing Losses" W/S, and trasferred from Cell [K85]	
	S/C-to-Ground Antenna Polarization Loss:	0.1	dB	This value is calculated in the "Polarization Loss" W/S and is transferred from Cell [F60].	
	Path Loss:	151.5	dB	Lp = 22 + 20LOG(D/λ); Transferred from "Frequency" W/S	
	Atmospheric Loss:	1.1	dB	This value is transferred from "Atmos. & Ionos. Losses" W/S, Cell [D23]	
	Ionospheric Loss:	0.8	dB	This value is transferred from "Atmos. & Ionos. Losses" W/S, Cell [D47:D50]	
	Rain Loss:	0.0	dB	This value should be estimated by the link model operator and place into Cell [B18]	
	Isotropic Signal Level at Ground Station:	-150.9	dBW	This is the signal level received at the Earth in the vacinity of the ground station using an omnidirectional ante	
	Ground Station (Eb/No Method):				
	------ Eb/No Method ------				
	Ground Station Antenna Pointing Loss:	0.5	dB	This value is transferred from "Antenna Pointing Losses" W/S, Cell [K102]	
	Ground Station Antenna Gain:	18.5	dBi	This value is selected at "Antenna Gain" W/S, Cell [E58]	
	Ground Station Total Transmission Line Losses	2.0	dB	This value is transferred from the "Receivers" W/S, Cell [J123]	
	Ground Station Effective Noise Temperature:	510	K	This value is calculated in the "Receivers" W/S and Transferred from Cell [J138]	
	Ground Station Figure of Merrit (G/T):	-10.6	dB/K	G/T = Ga-Ltl-10log(Ts). This is the uptimate measure of the receiver's performance.	
	G.S. Signal-to-Noise Power Density (S/No):	66.6	dBHz	Boltzman's Constant: -228.6 dBW/K/Hz	
	System Desired Data Rate:	300	bps	Operator selects this value. Be Careful! This is the data rate, not the symbol rate.	
	In dBHz:	24.8	dBHz	This is simply = 10log(R); R= data rate	
	Telemetry System Eb/No for the Downlink:	41.8	dB		
	Demodulation Method Seleted:	GMSK		Values selected in "Modulation-Demodulation W/S, Cell [E30]	
	Forward Error Correction Coding Used:	None		Value selected in "Modulation-Demodulation" W/S, also Cell [E30]	
	System Allowed or Specified Bit-Error-Rate:	1.0E-05		The selected value is transferred from the "Modulation-Demodulation W/S, Cells [E33:E50]	
	Demodulator Implementation Loss:	0	dB	This value is transferred from the "Modulation-Demodulation W/S, Cell[E52]	
	Telemetry System Required Eb/No:	9.6	dB	The selected value is transferred from the "Modulation-Demodulation W/S, Cells [F33:F50]	
	Eb/No Threshold:	9.6	dB	This is the result of the "Modulation-Demodulation" W/S and is transferred from Cell [H32]	
	System Link Margin:	32.2	dB		

Antenna Pointing Losses | Antenna Polarization Loss | Atmos. & Ionos. Losses | Modulation-Demodulation Method | Uplink Budget | Downlink Budget | System Peformance Summary | Antenna Patterns

- SatOrb



- JSatOrb

General

Link budgets

A link budget is useful when designing a communication system. It accounts for the many losses that can happen during communication and returns a margin in dB. This margin will determine how easily it will be to communicate with our satellite.

Static vs dynamic

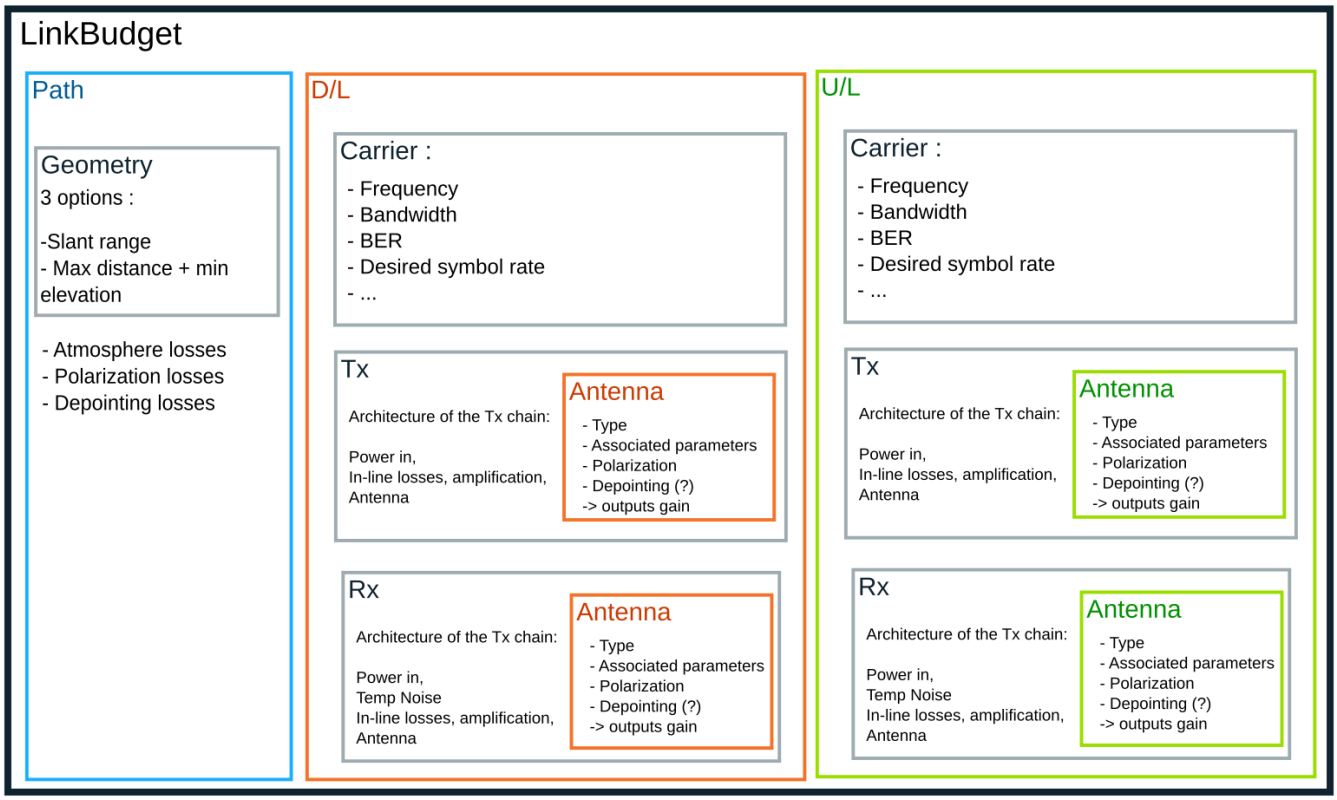
Often we use the worst-case scenario and calculate a link budget at a specific instant. While this help us know if the link will be possible or not, this doesn't take into account many things (can we have variable bitrate in order to maximize the amount of data transferred?, how to take into account pointing error over time?)

A dynamic link budget is similar to a static link budget. It has varying geometry (variable slant range) and time dependent inputs. This is already handled by linkpredict.

In the case of a static link budget, we are only interested in fixed output values while for a dynamic link budget we want to see their evolution (and also calculate the amount of data transferred for instance)

Required inputs and where do they belong?

Trying to hierarchize required inputs gives us this diagram:



Hierarchy of Inputs

This will be useful when designing our application layout.

Linking Luplink into JSatOrb

There are multiple levels of integration :

- data, we can import data from JSatOrb into Luplink
- UI, we can easily navigate between both apps
- API, we can import data through nanospace

UI:

How to navigate between both apps?

There are 2 main solutions :

- tight integration : we use nanospace-lb as a module inside JSatOrb. (+ only one app - standalone?, how to run JSatOrb without nanospace-lb? (different repos, ...))
- loose integration : both app keep existing separately and we have to link between them. This involves having two different adresses (+ keep repos separated, - need two separate adresses) but maybe it might be doable with a particular Apache configuration / website structures (see https://geekflare.com/fr/multiple-domains-on-one-server-with-apache-nginx/)

For instance: jso.localhost:80 & luplink.localhost:80

In the end the cleanest solution involves converting luplink to a library. This way, we can import its components both inside luplink-standalone & JSatOrb with the added perk of reformating the code in a cleaner way.

The chosen library architecture involves packing each feature (one or more components) inside a module and importing these modules to build the interface.

It appears that the development speed is not impaired thanks to the use of `npm link` which allows for quick rebuild of the library on changes. Explanation given in the README

The idea is to build a workspace with library/ and app/, this way it is possible to serve a standalone possible. But it would also be possible to publish the library, allowing to import its modules inside JSatOrb or any other application!

Data:

When exporting data from JSatOrb, we have particular informations such as:

- satellites
- constellations,
- ground stations We can let the user choose between all of these and import SMA+elevation in Luplink before doing the calculations This is quite easily done.

API

Some nanospace angular components are available, allowing to fetch data from database. Problem is, fetching all data at once would be easier for the user but the way the database is constructed makes it a bit complex. We should look for a solution to do so.

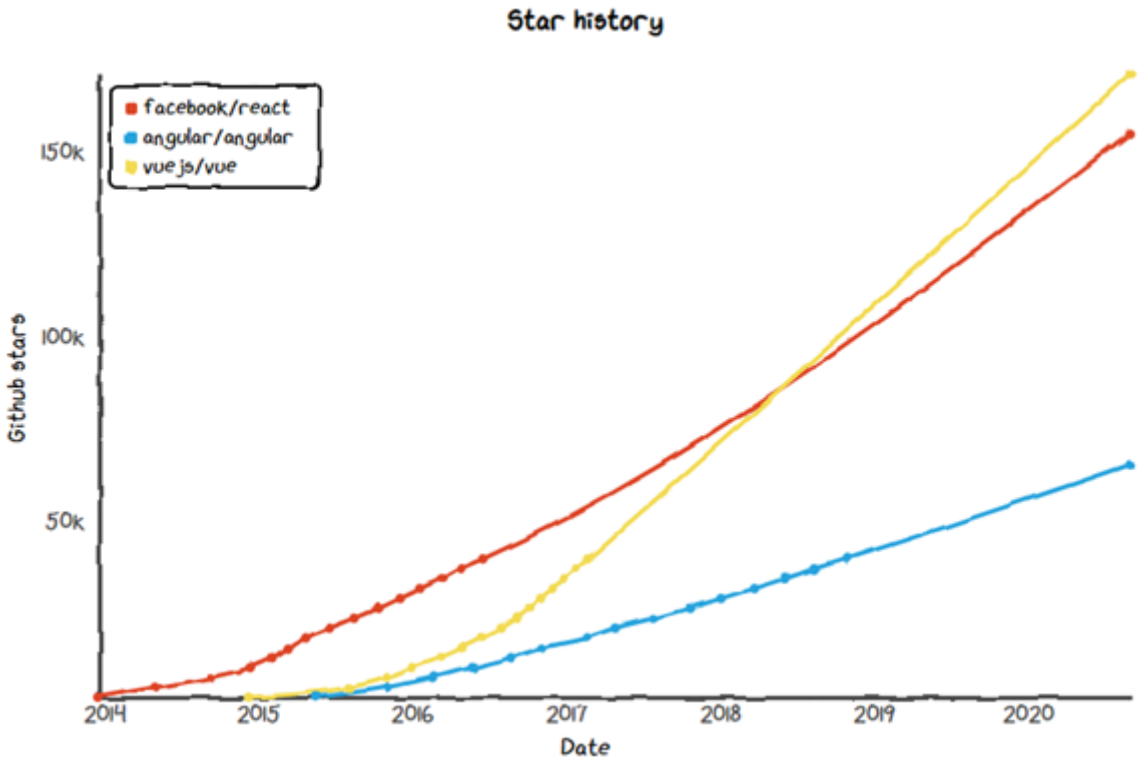
In the end, the library strategy seems to have payed off, we can now develop separately luplink from JSatOrb and still have a clean integration between the two. The only downside is that there are some more overhead when developping. The README.md will need to explain the development process thoroughly.

Tools

To build this application, we will be using a certain number of tools. The following section describes the different tools chosen and the reasons for choosing them

What frontend framework to use ?

Comparison of Vue, React and Angular in terms of GH stars :



Short descriptions:

- **Angular**, provides good abstraction, encourages good coding practices. Maintained by Google. Considered by somes a bit over-engineered. *See: Angular Coding Style (https://angular.io/guide/styleguide)*

Alternatives:

- **Vue.js**, A lighter and easier alternative to Angular. It also has better performances. However testing has not been as closely tied to the framework as it is the case for angular. *See: Comparison between Vue.js and other frameworks by the Vue.js team (https://fr.vuejs.org/v2/guide/comparison.html)*
- **React**, Maintained by Facebook.

Comparison

	Angular	**React**	**Vue.js**
Maintained by	Google	Facebook	
pros	Lots of functionnalities out of the box		Extensive built-in capabilities
	Convenient to refactor old code		Really Light
	Better reusability of components		Convenient to use

	Angular	React	Vue.js
cons	High threshold of entry	Not a complete framework but instead a libray, more time consuming. 3rd party tools might become outdated, adds cost to development	Runtime errors frequent and frustrating (linked to convenience of usage)
	Lot of conceptual overhead / Considered a bit over-engineered		

Opiniated sources : https://sloboda-studio.com/blog/the-ultimate-comparison-angular-vs-react-vs-vue/ https://itnext.io/dont-be-afraid-and-just-ng-update-1ad096147640

According to these articles, Angular updates happen regularly, at fixed times and aren't too much breaking. Maintaining clean code should not be too much of an hassle. While Google is known for closing projects, Angular has a big user base and is MIT-licensed. It also has maturity so not much to worry about.

In the end: There seems to be no real outcomer. All of these frameworks have been used to build successful web applications, are documented and have important user bases. The choice might end up being based on personal preferences.

Angular still has good testing capabilities out-of-the-box and uses typescript which enforces good programming practices. Using this framework sounds like a sensible choice.

It could be interesting to build another version in Vue.js for the sake of comparison.

Some Questions: *Does it break with each update ? Which version ? 11 released in november 2020, will be replaced in May 2021 but will be kept under LTS until May 2022. Would it be easier to use Vue.js or React ?*

Styling

SCSS or CSS: Angular handles SCSS natively. SCSS extends the capabilities of CSS. What can it do ? https://sass-lang.com/guide. It is under a MIT License. SCSS provides inheritance, variables & operators and is based on the DRY philosophy *(Don't Repeat Yourself)*. This is a good choice for our project. It doesn't require much knowledge. *Ressources : https://learnxinyminutes.com/docs/sass/*

Summary of advantages of SCSS : Increases readibility and intuitive to understand

What CSS framework ?

Lots of CSS frameworks exist. They help with UI consistency and faster prototyping. They often provide similar capabilities.

- **Bootstrap**, One of the biggest CSS frameworks. It is well documented and has extensive capabilities. It can however look a bit bland as it is used a lot. It's a bit heavy as it requires jQuery (unless we use

one of the two angular modules below. Their installation can be a bit complex however)

Alternative CSS frameworks :

- **Bulma**, is a pure CSS framework, gaining a lot of momentum. It is simple to use and has powerful tiling capabilities. It is also much lighter than bootstrap.
- **Picnic**, simple & opiniated CSS framework, open-source
- **Angular Material**, close integration to Angular, for a Material look
- **Semantic UI**, good alternative to Bootstrap

ngx-bootstrap / ng-bootstrap : Two different teams working on similar goals : Making Bootstrap usable without jQuery in Angular (however couldn't get ngx to work and ng isn't working with Angular 11)

Material Design This has been carefully designed. However, I feel like this isn't totally adapted to "data-heavy" UI, it takes up a lot of space. An other issue is the generic Google look of the application, unless we don't follow all the rules. Material Design Lite is another alternative

From the nanospace-client-lib README, bootstrap might be required for displaying () components

Linting

Angular 11 is at a cornerstone, TSLint while deprecated is still installed by default. We migrated from TSLint to ESLint so that it won't cause issues in the future.

This was the opportunity to set-up linting. We added the standard "AirBnB-base" linting rules since the code is intended to be shared.

Diagrams

AMSAT uses block diagrams which makes it clearer.

The use of SVGs is suitable : robust, easy to create & edit programmatically.

Adding annotations is a bit more problematic : text positionning in SVG can be tricky. A first solution is to use absolute positionning and DIV elements. This is a first solution.

The next question is "how to make it scalable?", this seems doable using JavaScript.

To test: Alternatively, we can use foreign HTML elements inside the SVG (https://stackoverflow.com/questions/6725288/svg-text-inside-rect)

It might seem that the chosen solution doesn't work as well as expected, a bit fragile.

The HTML canvas tag exists but isn't adapted to display changing informations and is resolution dependant.

The SVG tag can use relative width and positioning

Others

- **Popper.js**, handles tool-tips positioning and overflowing for us, open source (included in bootstrap.**bundle**.min.js), used by BS for dropdowns menus
- ngx-nanospace-client-lib depends on **'@fortawesome/free-solid-svg-icons'** et **'@fortawesome/angular-fontawesome'** these icons are open-source, we can use them for our project.

/!\ REALLY IMPORTANT /!\ : Don't ever forget to import FormsModule along with nanospace-client-libto avoid unrelated error messages. Refer to ngx-nanospace-client-lib README and strictly follow it!

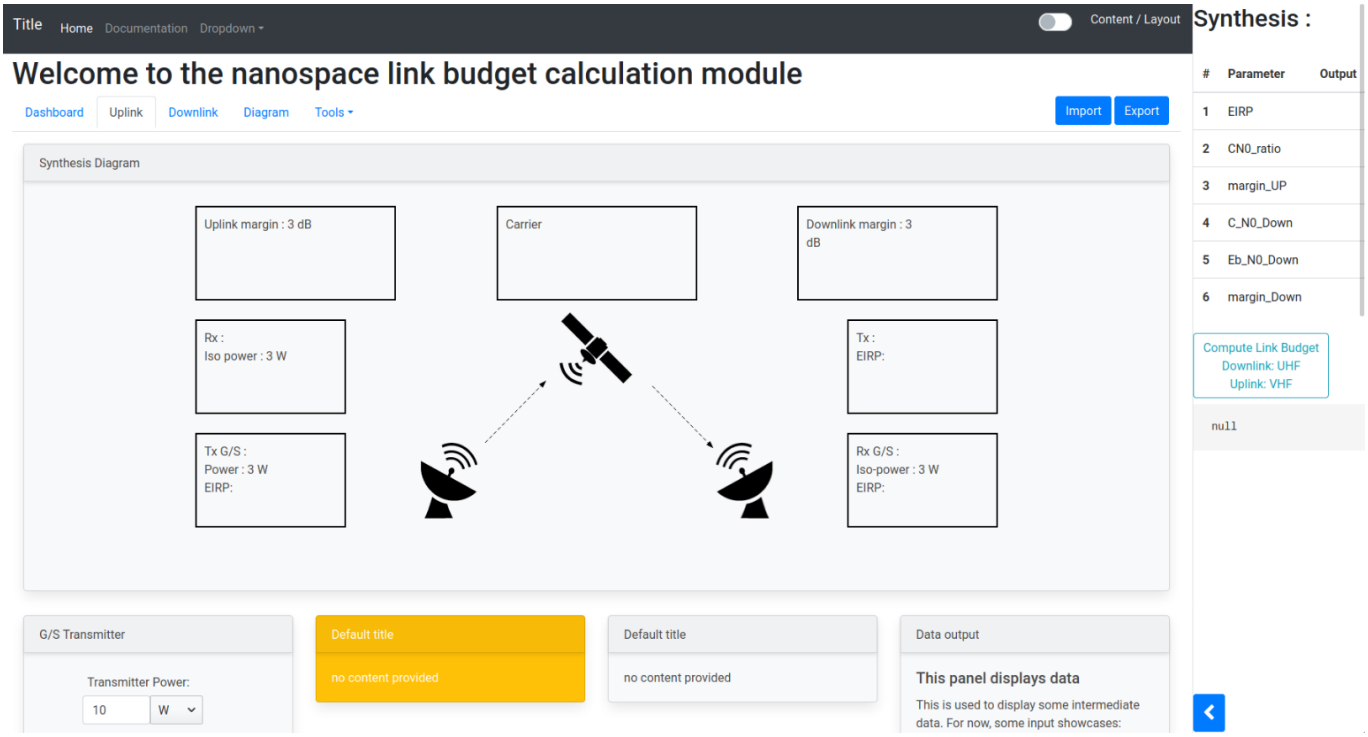
Building the interface

Layout system

To be able to quickly switch between totally different interface ideas, having a layout system is interesting. This is done using the RouterModule and nested routes : we can call a container module and children components through the path #TODO: add illustration

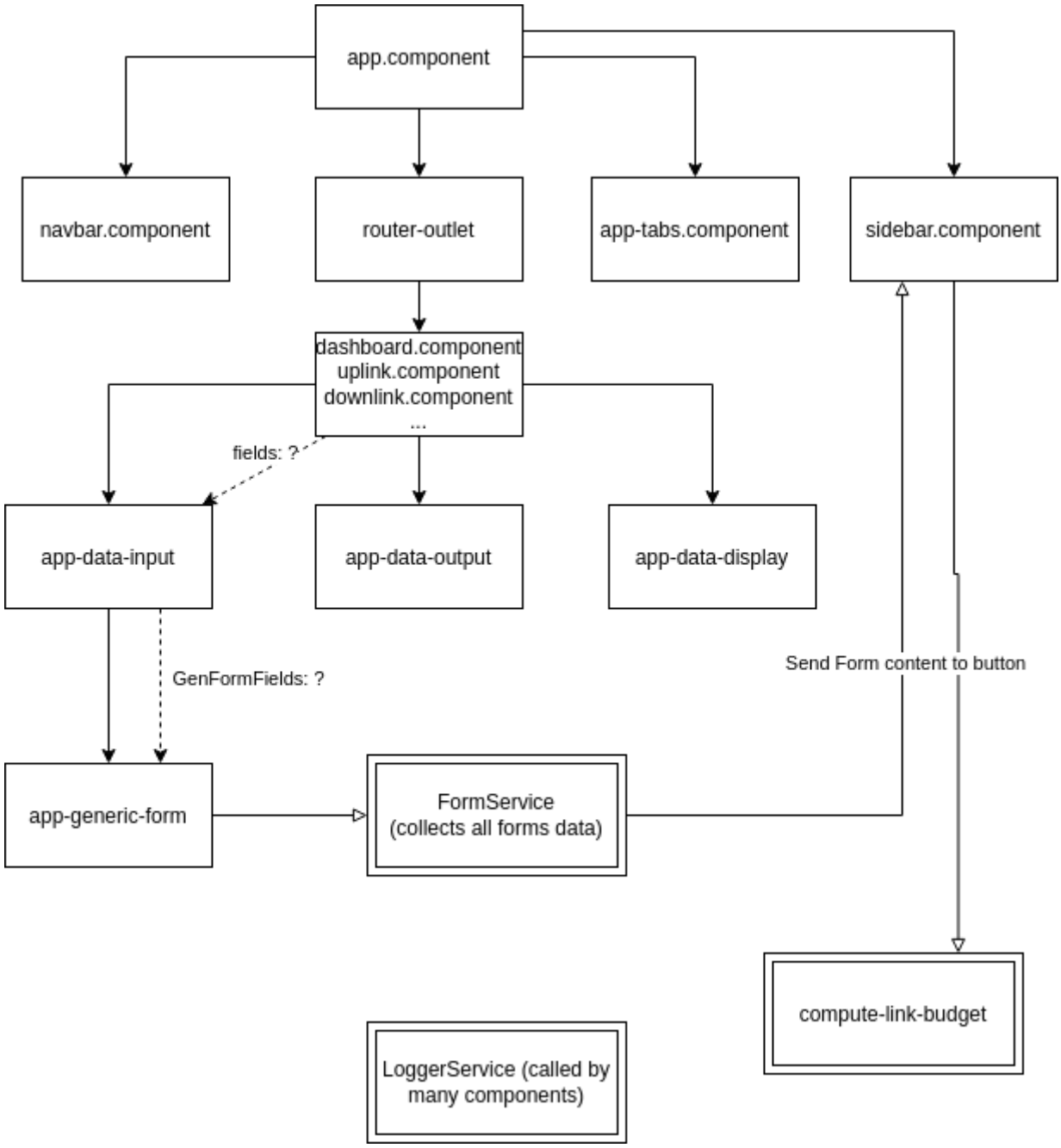
However it can be quite hard to reuse some of the components between the two layouts. For instance, in the case of a simple form component, while the logic stays exactly the same, the template uses entirely different classes and structure depending on the CSS framework.

V0



This first layout uses Bootstrap. Most of the work has been done on the underlying form logic, allowing communication with API, reusability of components and making it easy to add new parameters for calculations. This was also used to experiment with SVG drawings and bootstrap layout system.

Features: Procedurally generated form, unit handling



Card layout

A card layout is well suited to the component paradigm of angular. This allows us to separate information effectively and make the form more bearable for the user.

Cons : *We lose some hierarchy of information, too much cards is equally overwhelming.*

Some issues with positioning on ultra-wide monitors. But easy to add support for small screens

Sidebar

There was the idea of using a sidebar to show calculation at all times. At first, tried to add a column but too complex, not possible to use as a component. Tried implementing it with classic HTML, some result but not ideal I think.

ng-sidebar exists but not actively maintained.

angular-material has sidenav which seems to fit the use case.

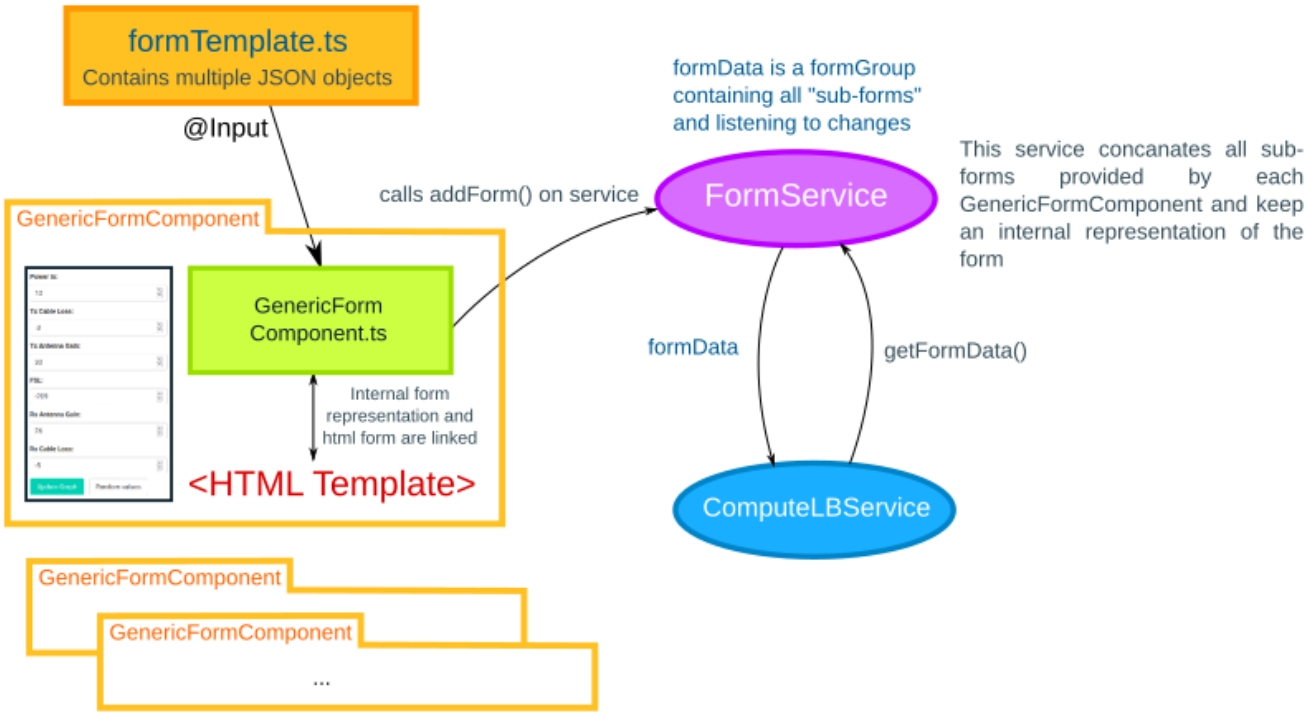
Next : Clarify the need, is there another way to show it ? more adapted than a navbar

Also this sidebar isn't responsive. #FIXME: add responsiveness to sidebar

Forms

Forms are at the basis of this application. The user has to input a lot of data. We want to avoid making this too tiresome, complex or inefficient.

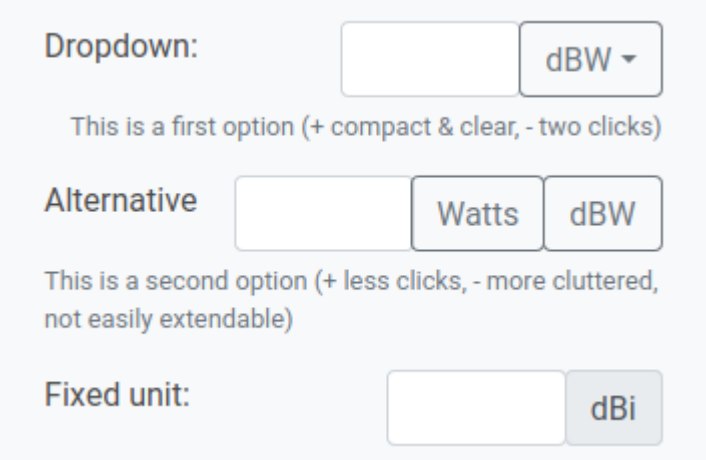
Architecture:



Forms Architecture

#TODO: Description

Units: Two ways to represent units, with each its perks. Having both at the same time lacks consistency. I slightly prefer the dropdown as it is more compact, easily extendable, and unambiguous.



Also should an input field be a component ? (https://dev.to/tsuzukayama/angular-5---how-many-components-is-too-many-components--2l9c).

These inputs require setting a unit and some logic so it might be a good idea to make it into a component.

Reusability: The forms are generated from json objects. This adds reusability and result in probably cleaner components for a reasonable added complexity.

Implementation: In angular there are two techniques to create forms : template-driven and reactive.

Template-Driven	Reactive
Asynchronous	Mostly Synchronous
Logic driving by template	Logic in component
Legacy	Modern
Limited to e2e tests	Form validation can be unit tested
adapted to small/medium forms	A bit overkill for small forms

(reference: https://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/)

From this comparison, it appears clear that Reactive forms are better suited in our case (complex form splitted in multiple parts)

#TODO: Simplify next section, lot of history but might not be needed!

The data is split into many sub-forms. I started to implement it before stumbling upon this example which corresponds to our goal: https://itnext.io/partial-reactive-form-with-angular-components-443ca06d8419 https://coryrylan.com/blog/building-reusable-forms-in-angular https://medium.com/angular-in-depth/angular-nested-reactive-forms-using-cvas-b394ba2e5d0d (composite Control Value Accessors)

This will require a bit more thought than initially expected. The main solutions for this would be:

- Parent to child communication with @Input and @Ouputs
- Communication using RxJs and services
- Making sure every form has a same common parent

The child-parent solutions would have some impact on the layout. It would also become really confusing real quick with these chains of @Inputs/@Outputs.

Using services seems like the obvious solution after some comparison. However using RxJs is not simple.

Merging Observables : https://github.com/ReactiveX/rxjs/issues/1308

https://angular.io/guide/dynamic-form

-> ArrayForms handles multiple FormGroup thus allowing to get notified from each change in forms.

Next problem : Routing destroys components. We have to make form data persistent. Ideas :

- Recover the state on tab change
- Don't use routing (make a full-page form and hide sections as required)
- RouteReuseStrategy -> This works but this doesn't feel totally right. #FIXME: Go back here, clean up and find if it's still a good idea

https://github.com/angular/angular/issues/5275 -> see RouteReuseStrategy / Sticky routes
https://samerabdelkafi.wordpress.com/2020/12/14/angular-rout-with-sticky-state/
https://medium.com/@disane1987/angular-router-and-tab-based-navigation-within-a-spa-c0a8ca2b3bc4
This can be handled more easily by UI-Router library

Questions: -Where to handle units ? (For now in BE but would also make sense to handle them in FE) **Finally have something working for the form, maybe a bit convoluted, should compare with other frameworks**

How to handle data not entered ?

We cannot guess, in the future persistance of data, possibility to import. Right now maybe some generic data automatically inputed to save time when using the interface.

- import inputs from a file (will be useful later)
- construct the JSON request from this

In the end, all data will be known, whether the components are created or not

Save/Load capabilities: The user can upload or download the json state of the form. #FIXME: Still lacks a bit of robusteness and security.

What about units ?

Getting values + units will probably require a custom FormControl (https://indepth.dev/posts/1055/never-again-be-confused-when-implementing-controlvalueaccessor-in-angular-forms) Really interesting article about how to build custom ControlValueAccessor

https://blog.thoughtram.io/angular/2016/07/27/custom-form-controls-in-angular-2.html How to build a custom component -> use native components if possible

Backend

Available libraries:

- Django (a bit overkill as it has similarities with angular)
- Bottle -> Good option according to second link (lightweight but comprehensive)
- EVE (for small to medium-sized projects)
- Flask-RESTful
- ...

https://www.fullstackpython.com/api-creation.html

Restless is a light framework that allows using the same API code for any WSGI framework

See also : API testing tools

Nice comparison about OSS projects : https://nordicapis.com/8-open-source-frameworks-for-building-apis-in-python/

Clear Documentation is paramount to API creation.

RAML : REST API Modelling Language https://raml.org/about-raml Swagger (OAM)

Testing

After researching unit testing in Angular, there is a lot of boilerplate code to write. Handling dependencies between components can quickly become cumbersome. One of the proposed solutions is to use the NO_ERRORS_SCHEMA. However this can have unexpected side effects : unknown properties bindings are ignored, tests pass and we only get an error message when building without indication of the origin of the issue.

This leaves us with three correct solutions on how to do unit testing : - Writing mocks for 1st level dependencies (calling them component_name.component.mock.ts, and making it implement the real component) - Can avoid mocking if doing integration tests - Using external tools like the npm module **Spectator** that drastically reduces boilerplate code and provides solutions for quickly mocking components, services, ...

This last solution seems to be a good compromise for keeping a clean project and clean testing files while doing everything as correctly as possible.

===

After some time away from testing, getting back to spectator implies learning how to use it again. The documentation was sometime a bit outdated. I think I will instead stick to classic unit-testing.

Found an interesting Medium article explaining how to handle dependencies between components while testing :

- Adding child component to declarations (least preferable)
- Adding NO_ERRORS_SCHEMA
- Manually mockup components
- use ngMock, MockComponent() (most preferable)

In the end, we use the default Jasmine TestBed for all our tests since Spectator's documentation can be a bit limimted at times. This makes our test a bit more verbose but easier to fix if something goes wrong

What to test ?

- *generic-form*
    - should create each field
    - should display the correct labels
    - should not be valid if wrong type of input / too long / short ?
    - should provide all units
    - should have the correct default unit
    - should have the correct default values
- *dashboard*

    - should display the cards
    - cards should have the correct title
    - cards should have the correct color
- *sidebar*
    - should display the data at the right place
    - should provide a button
- *services*
    - how to test services ? Most of the logic there
- Other components don't implement much logic yet, writing correct tests for generic-form will be a good start

And then continue with integration testing : https://dev.to/cjcoops/how-to-write-simple-angular-integration-tests-with-spectator-1i1b

Testing a component by mocking services (3 ways): https://blog.danieleghidoli.it/2016/11/06/testing-angular-component-mock-services/

API Testing

- Test adimensional values
-

Performance Tests

A first way to do performance tests is by using Apache Bench `ab -n <number_of_requests> -c <concurrency> <url>`

Form Validation

We need to run some checks on the form before sending it to the API.

Objectives :

- Clear explanation of how to fix the error
- Avoid user frustration (eg: validating fields too late or too soon, not helping much if something is wrong)

Rules :

- Fields:
    - Empty field -> default value used (for educational purposes) -> let the user decide if he wants some strict mode ?
    - Non-numerical value -> error (later we'll also have dropdowns to validate, but no text ?)
    - Number of digits limitation ? -> 1MW ?
- Units:
    - One is selected by default for convenience
    - Expect user to validate it himself
    - Support for custom units ? -> implies parsing them, having too many of them in the dropdowns would be frustrating. Maybe giving the possibility to use pre-defined custom units to users in a settings menu ?

Realisation :

- if default values are used, give some gentle 'important' *orange* notification (in the tab, near the fields and near the synthesis)
- if incorrect values are given, show a pop-up warning and clearly indicate the faulty tab and fields with a *red* marker
- When a tab is correct, some subtle confirmation everything is fine -> giving a sense of completion, progress (but avoid making everything cluttered because of this)

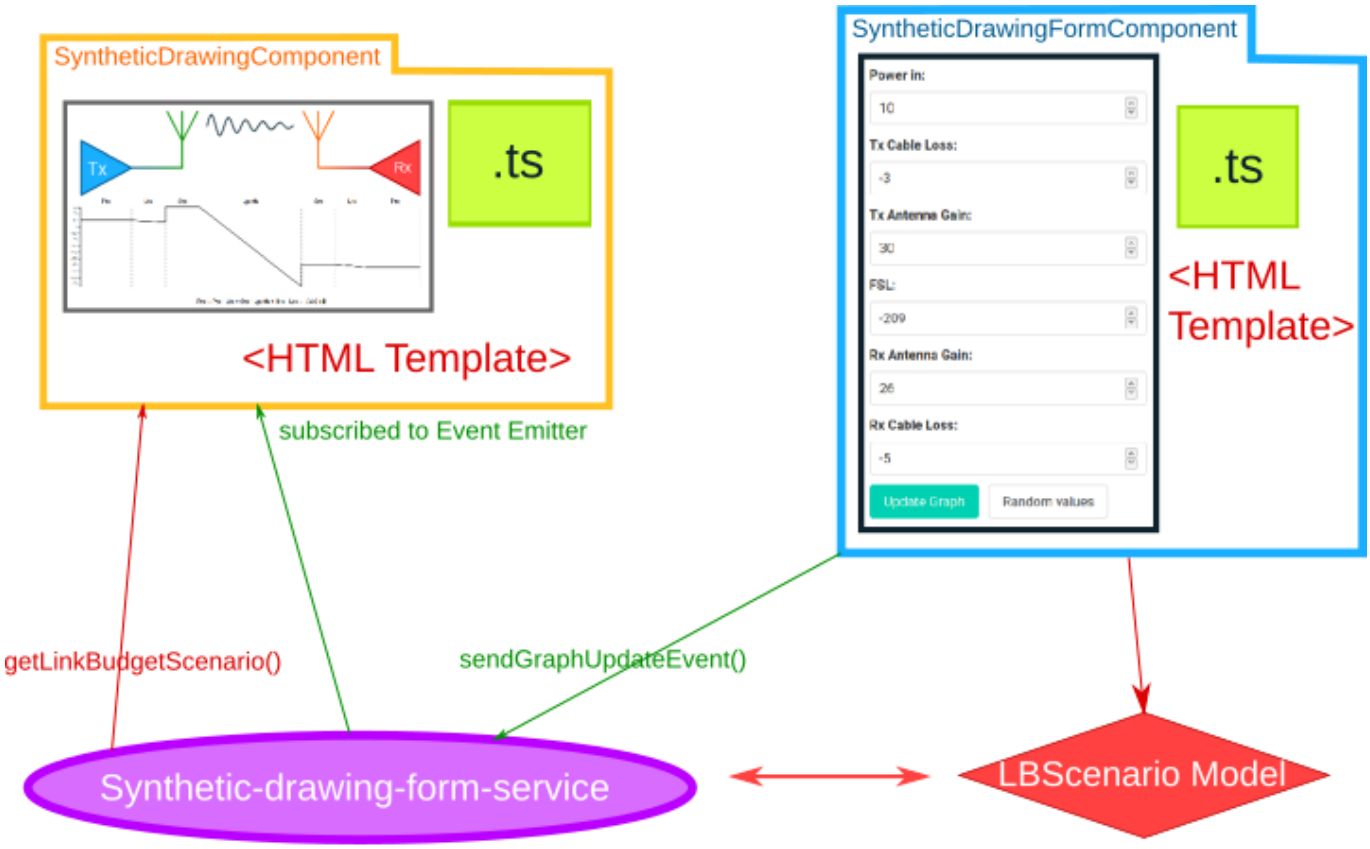
Also :

- Add custom validation rules : frequency should be >0 while gain can be negative
- Clearly define whether the losses are positive or negative (losses -> , gain ->)
- Defining a form as a table just like below would be a good idea

Input Name	Input Type	Input Validations	Validation Type
User Name	Text	1. not empty 2. at least 5 characters long 3. can't be more than 25 characters long 4. must contain only numbers and letters 5. unique in our system	1. required 2. minlength 5 3. maxlength 25 4. pattern validation 5. custom validation
Email	email	1. not empty 2. valid email	1. required 2. pattern validation
Password	password	1. not empty 2. at least 5 characters long 3. must contain at least one uppercase, one lowercase, and one number	1. required 2. minlength 5 3. pattern validation
Confirm password	password	1. not empty 2. Equal to password	1. required 2. custom validation
Terms	checkbox	1. accepted	1. pattern validation

Diagram (drawing + graph)

We want to provide some visual feedback to our user. Building a simple graph was a first step. #TODO: more details



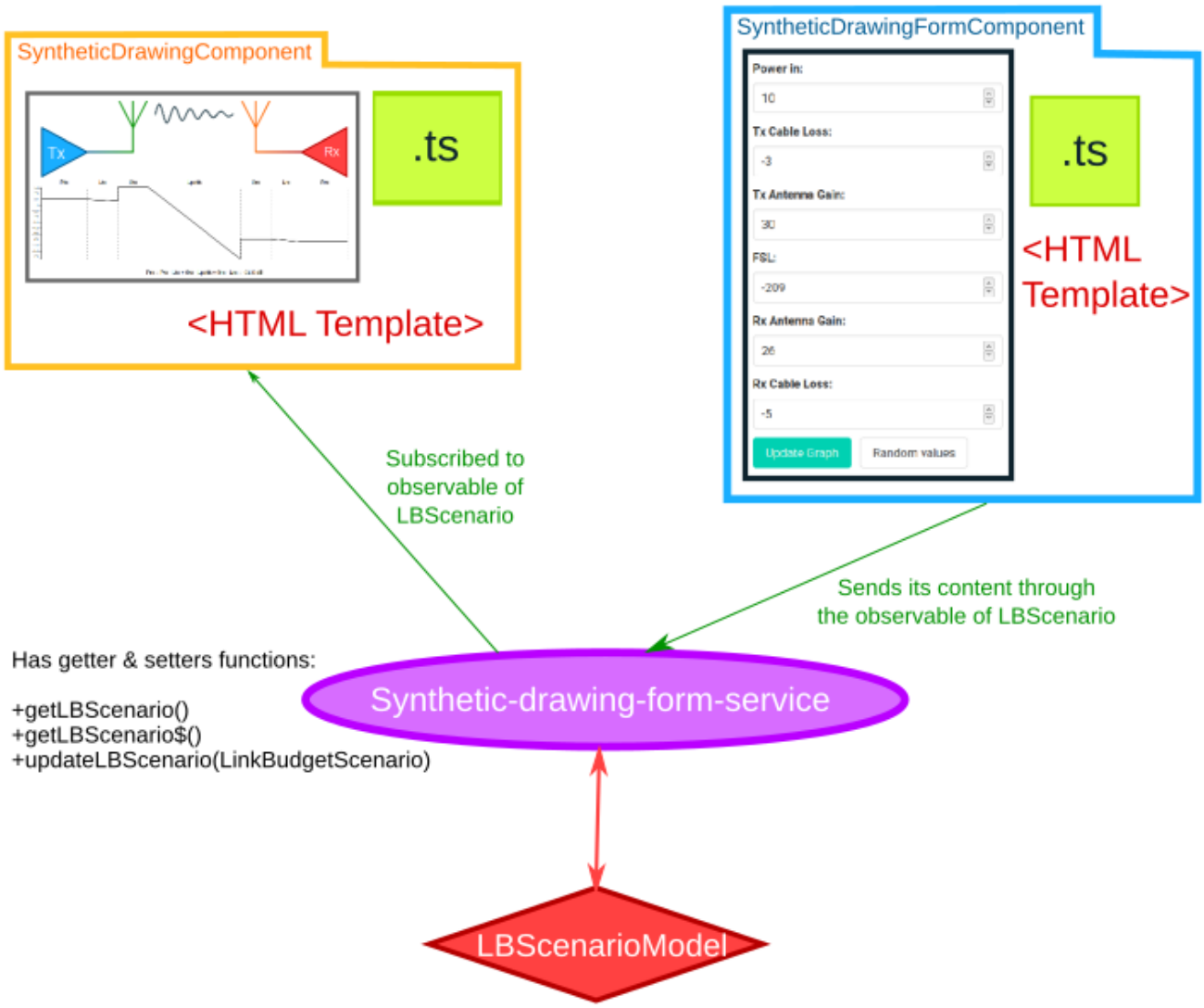
Synthetic Drawing Architecture

This architecture looks like it could use the Observer Design Pattern. In Angular, this is commonly used and RxJS already implements it through Observables. (https://angular.io/guide/observables).

The RxJS BehaviorSubject allows multiple observer to observe the same observable instance. By making the LBScenario as such we can subscribe to it as many time as we want

We need a singleton service (only one instance) #TODO: What if we need multiple instances ? -> We could replace our singleton with a structure containing multiple scenarios and add a pointer to our components

Updated architecture :



SyntheticDrawingModule Architecture

This takes advantages of RxJS capabilities and makes a much cleaner & robust structure.

Tried to implement an example of a simplified link-budget diagram

The goal was also to familiarize with D3.js since it will probably come in handy later in the project.

At first I tried to do the visual part of the drawing using D3js but it clearly was inefficient. Positionning was really complex. After spending too much time on this, I realized this part of the diagram didn't depend on any data and instead chose to do it in Inkscape.

Then for the graph, the syntax is complex and it took me a really long time to get something working. (Lack of documentation, many outdated examples). Now we have a working graph, and after the initial set-up it becomes actually easy to use.

Now that we have almost solved the scaling issues, I can't help but wonder if we are not doing it wrong. D3 seems really complex to use. It doesn't seems straigthforward. There exist alternatives : using HTML5 Canvas, since we won't be redrawing it too often could be interesting. Other SVG libraries such as svg.js

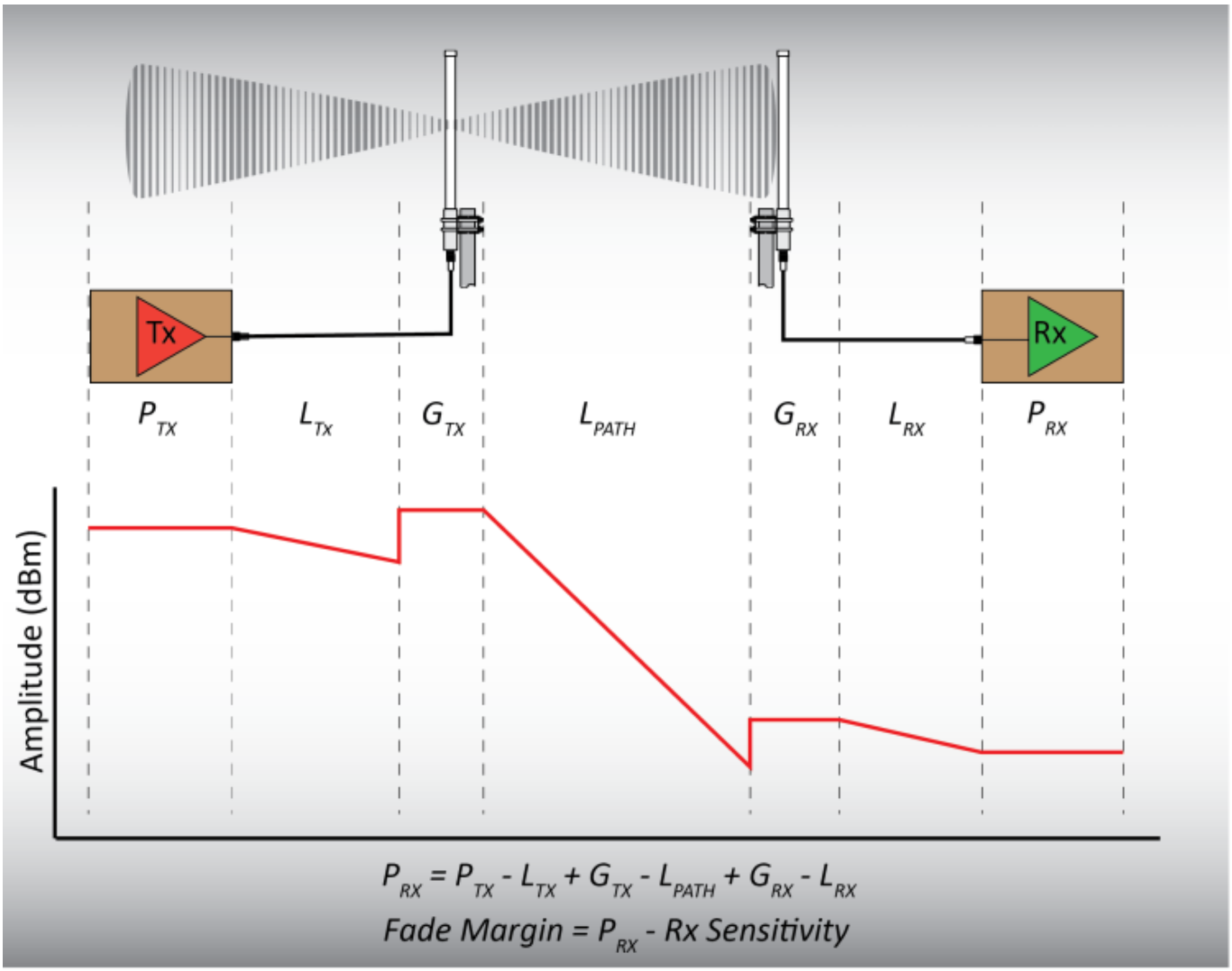


FIGURE 1-1. System Gain-Loss Profile for a Link Budget

Default form

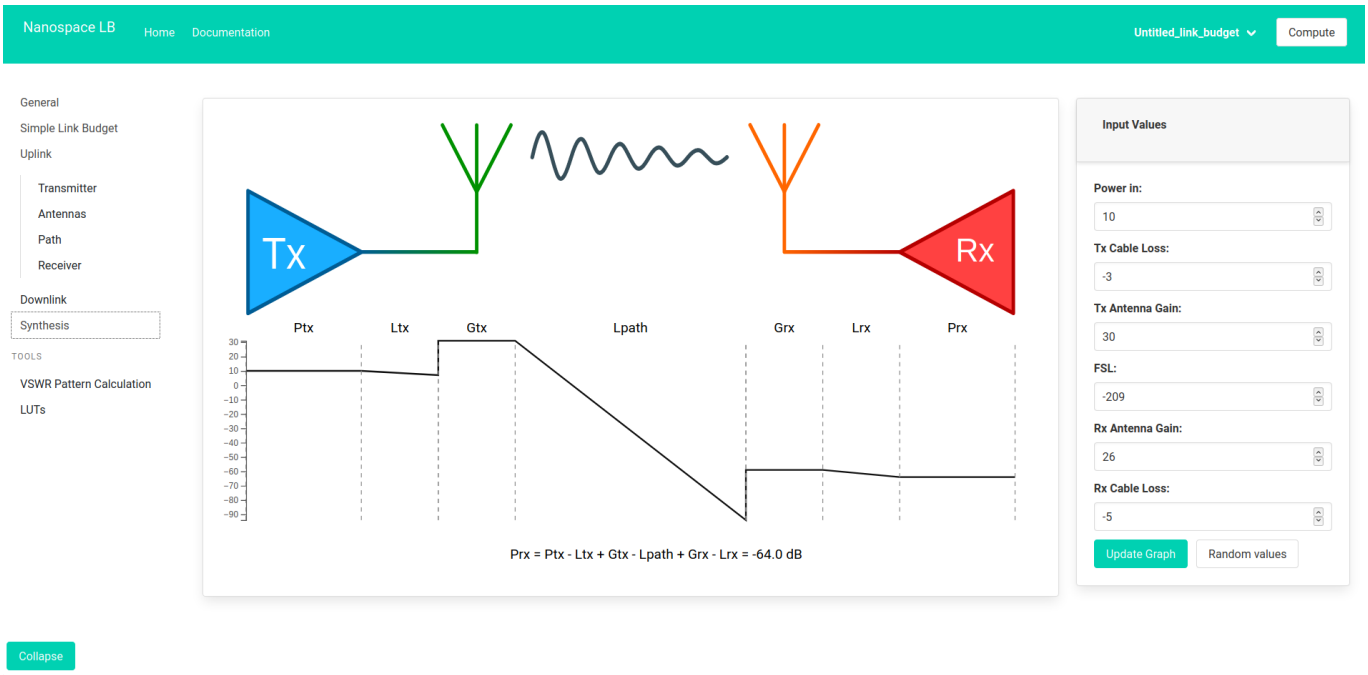
We want to provide a default example to user. One way to do it would be using placeholders values and filling the gaps. The issue is that the user might forget to fill one of the fields and end up with the wrong result. To fix this, we could add a visual indicator that some default values are used but this might be visually too distracting or too easy to miss. A solution is to NOT use placeholders, fill the values with a 'default' config and let the user change this config or even create his own.

Timeline:

- User opens app, default scenario pre-filled
    - User changes the scenario, the form updates
    - User changes a value, the scenario switches to 'custom'
    - User chooses to create a new scenario
    - Possibility of saving / loading

V0.1

Added a layout system and started working on a second layout using Bulma CSS framework instead of Bootstrap



The idea was to :

- Find shortcomings of v0 and fix them
- Check reusability of components
- Test new design and UX ideas, experiment with other frameworks to find one that fits our needs.

This layout frees some vertical space, keeps the cards and add a sidebar navigation.

Reusing some of the components required small changes but nothing too time-consuming. This will be even easier next time. However, I realized there are many components and reusing them requires setting up services and such so it might be the time to document them.

About Bulma :

- Documentation sometimes a bit sparse or incorrect but not too much of a hassle.
- Much nicer handling of forms with "add-ons" (for units)
- Forms can be a bit heavy with too much "controls" divs.

UI Experimnents

Quick way to test some components in context

New hierarchy:

The left sidebar was confusing. Different concepts at the same level. Wanted to improve it & find a cleaner base layout for the future.

Some observations:

- Should keep editing values consistent, always at the same place
- Clear and concise navigation, user should know exactly where to click to edit a certain parameter
- There's a compromise between compacity & easy to locate information : either we have a lot of unused space with lots of clicks to get to a precise value or we have many parts together minimizing the amount of effort to get there but this is less readable.

static, dynamic, basic or full?: in order to choose efficiently between these, we can tie them to the current LB being edited as a parameter. When we switch from full -> basic, we keep all informations but don't use it for calculations.

hierarchy: Based on the Input Hierarchy diagram, we'll have main categories for differents LinkParts and subcategories for sub-LinkParts. We will have a 2-panel interface with an edition zone & a synthesis zone. In the synthesis zone, we will be able to switch between local & global and in the editing zone, we will find our sub-forms organized by sub-LinkPart allowing user to get to them directly either by using HTML anchors or scrolling.

pros: Solves many of our previous issues and it is easy enough to make it responsive. A good compromise between too much white space and clearly organized inputs.

cons: Not many for now, maybe too much scrolling.

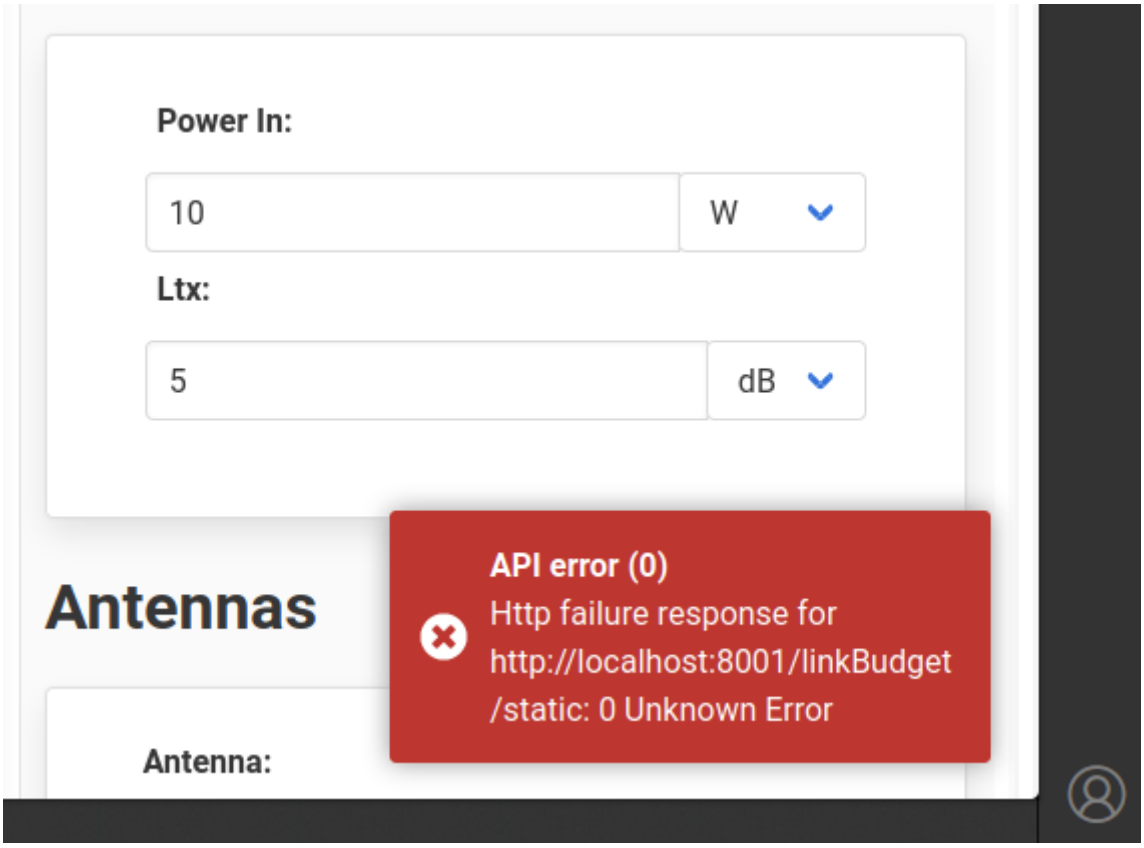
Antenna selection

Had to redefine the way we handle user inputs in order to incorporate antenna-selection, ... #TODO: Add diagram here

Displaying results

Logging

Already had a basic logger service in v0. This logger allows to keep all logs at the same place. For instance, by using the Toastr angular module, we can get errors directly visible by the user (& developper)



This made me think about giving a visual cue when frontend is connected to API.

#TODO: One way to do it would be to ping regularly the back-end. This would require persistent HTTP connection in order to minimize handshakes. Is it worth it ? Might be simpler to just give loading, ...

Components inventory

- Layout 1:
    - card
    - navbar
    - sidebar
    - tabs
    - views : *These are typically not reused because they are tightly coupled to the CSS framework and how components are to be positionned*
        - dashboard
        - diagram
        - downlink
        - uplink
- Layout 2
    - card
    - simple-lb
    - synthesis
    - uplink
- Shared
    - generic-form: *A simple generic form that can be passed to a card through ng-content*. **These are linked by the form service and should be provided with a Form object as @Input**
    - synthetic-drawing: *A D3.js graph + SVG drawing representing a radio link*
    - synthetic-drawing-form: *Can be used along with synthetic-drawing to directly input data and update graph*

Naming

- **Component** : using this term for the different parts of the comm is confusing with the angular concept of component. Alternatives : part, subsystem, **LinkPart** #TODO: find alternative
-

Detailed explanation

Detecting changes

The ngOnChanges() function in angular is considered to have some problems : is called for each input change so it requires filtering which property actually need to be changed. This reduces readability. Using getters and setters is a first alternative, but it introduces a new ghost variable which can be changed anywhere in the code. It is also quite verbose. The typescript way of doing it would be with decorators. However it is still considered an experimental feature so we will stick with getters & setters (https://angular.io/guide/component-interaction#intercept-input-property-changes-with-a-setter). NgOnChanges() might still be used for multiple interacting input properties.

When & how to use each of the first two approaches: https://stackoverflow.com/questions/38571812/how-to-detect-when-an-input-value-changes-in-angular

Card system

Comprehensive architecture diagram

#TODO: Explanations on how to add new cards, ...

Documentation

Users should have access to documentation about what the different values are, ...

Math formulas

We will probably need to display easily some math formula.

"MathJax is an open source JavaScript display engine for mathematics that works in all modern browsers"

It is used by mathoverflow and other math blogs. It seems to be a reliable, widely used solution.

However this solution is poorly handled by Angular, because of issues with the use of {} in the templates.

So currently, we can display MathJax Formula contained in variables containing HTML but we cannot use them directly in the template because Angular won't allow it. The issue of keeping the HTML in variable is that we do not have access to tools to properly format it. A solution could be to put this documentation in external HTML files and load it

A choice between interpolation and math equations. Maybe put equations inside variables

In the end, the installation of MathJax on angular was too much of a hassle and we ended up choosing an angular module called ng-katex. It is able to display LaTeX code contained inside a variable.