wavebinder™

Revolutionizing front-end data management

Index

Premise

Wavebinder™ is a library designed to help front-end developers facilitate data dependency management in front-end applications.

The growing demand for web applications requires increasingly complex front-ends, which are difficult to develop and maintain.

1.

The data flow problem

THE DATA FLOW PROBLEM

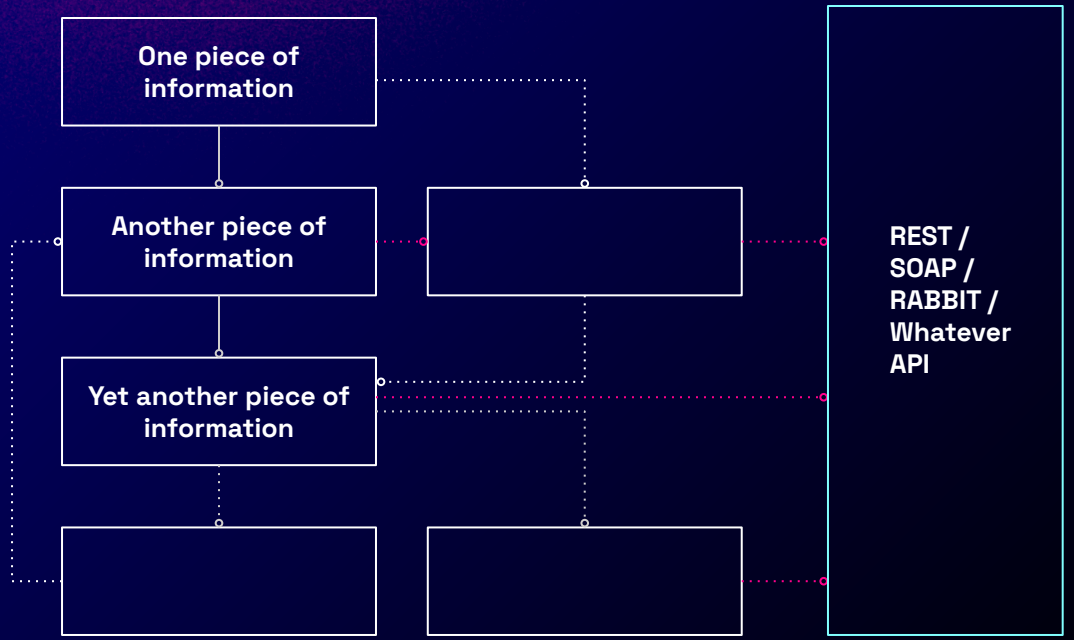
Why is data management so complex?

In modern applications, every field and component depends on other data.

Data doesn't update linearly.

Complexities increase when using slow and asynchronous APIs.

Interfaces are becoming increasingly complex and dynamic.



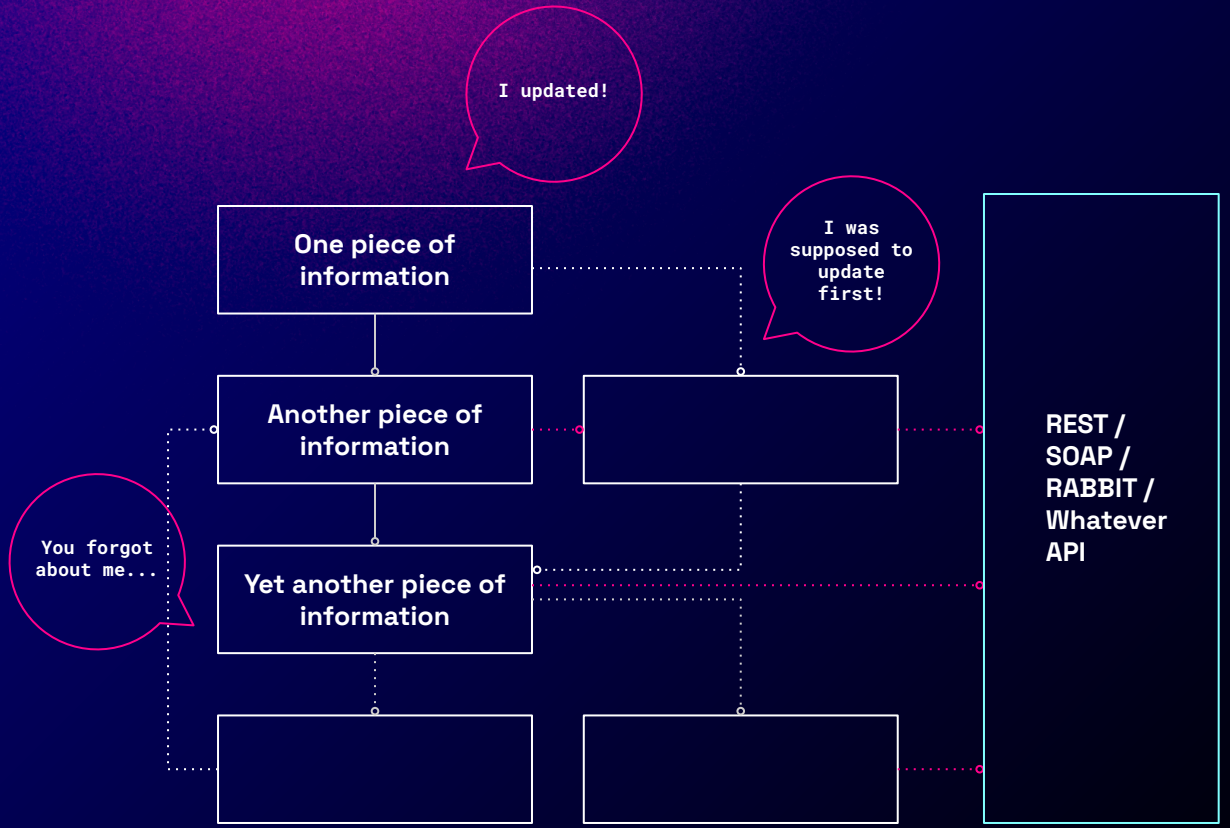
What are the consequences?

1. Long times and delays in interface development

2. Errors and unwanted behaviors

3. Suboptimal user experience

4. Difficulty in maintaining and updating logic



2.

The challenges for front-end developers

What are front-end developers looking for?

Front-end developers are looking for tools that simplify their lives and optimize the data flow between components.

Manual management of data dependencies becomes unsustainable in complex applications.

What is the answer to their needs?

WAVEBINDER™ was developed to respond to these needs, offering automation, flexibility, and control.

Wavebinder™

WAVEBINDER™ offers compatibility and ease of use that make it ideal for integration into your front-end projects.

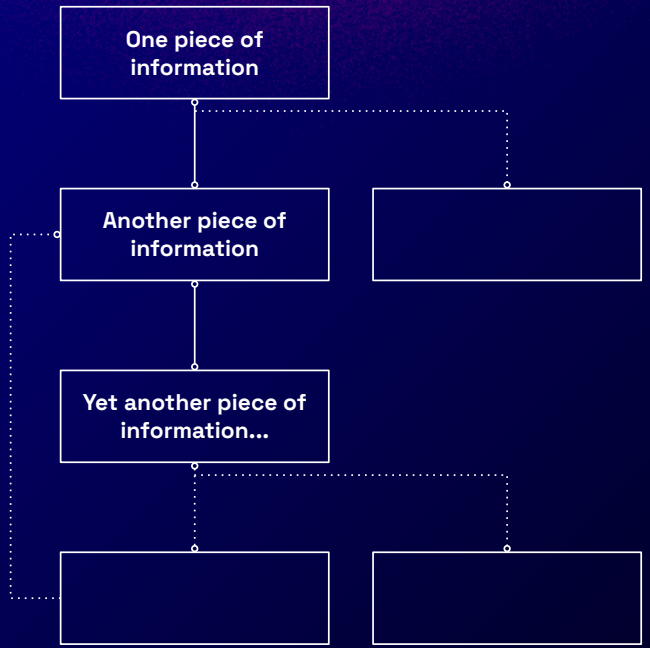
Here are some of its main features:

- Available as an NPM package
- Based on RXJS
- Works perfectly with major frameworks:
  - Vue.js
  - Angular
  - React

3.

How Wavebinder™ works

Modeling data as nodes in a graph.

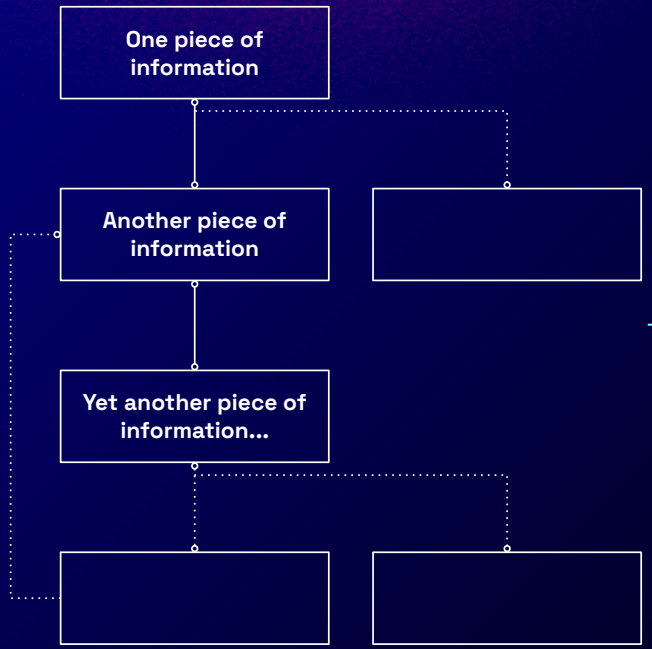


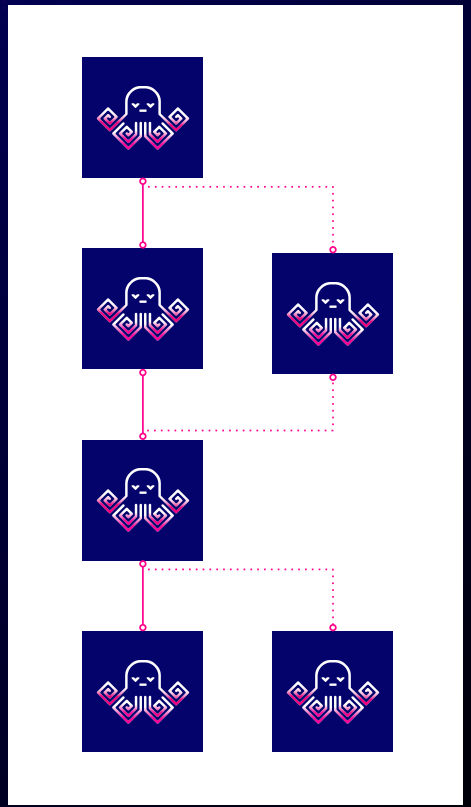
Defining nodes in a JSON object

```
const config: =
[
  {
    "name": "region",
    "type": "MULTI",
    "path": "/region",
    "la": {
      "type": "GET",
      "addr": "/region/list",
      "serviceName": "RETRIEVE_DATA"
    },
    "dep": []
  },
  {
    "name": "province",
    "type": "MULTI",
    "path": "/province",
    "la": {
      "type": "GET",
      "addr": "/{region}/province/list",
      "serviceName": "RETRIEVE_DATA"
    }
  },
  ...
]
```

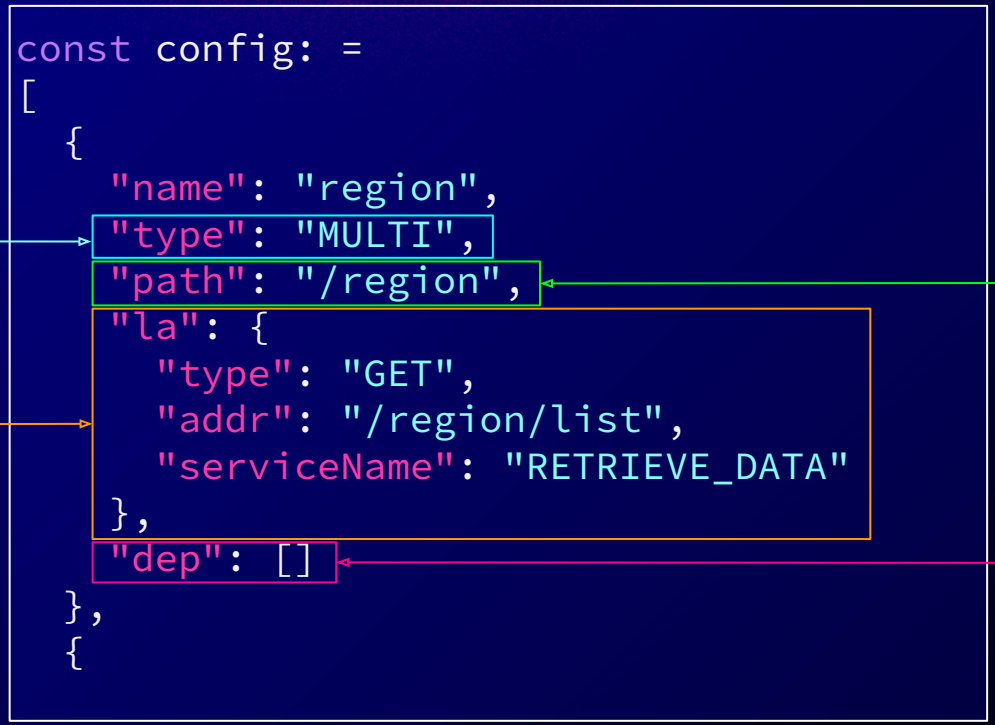
Dependencies between nodes in the JSON structure

```
{
   "name": "city",
   "type": "MULTI",
   "path": "/city",
   "la": {
     "type": "GET",
     "addr": "/{region}/{province}/city/list",
      "serviceName": "RETRIEVE_DATA"
    },
   "dep": [
     {
       "nodeName": "region",
       "isOptional": false,
       "onUpdate": true,
       "type": "PATH_VARIABLE"
     },
     {
       "nodeName": "province",
       "isOptional": false,
       "onUpdate": true,
       "type": "PATH_VARIABLE"
     }
   ]
}
```





Specify the different properties for each node



Specify the different properties for each node

The type of data it contains (e.g., a single value, a list, an object...)

The action required to load its value (e.g., an API call, a user selection...)

The path where to store it on a model

Its dependencies on other nodes

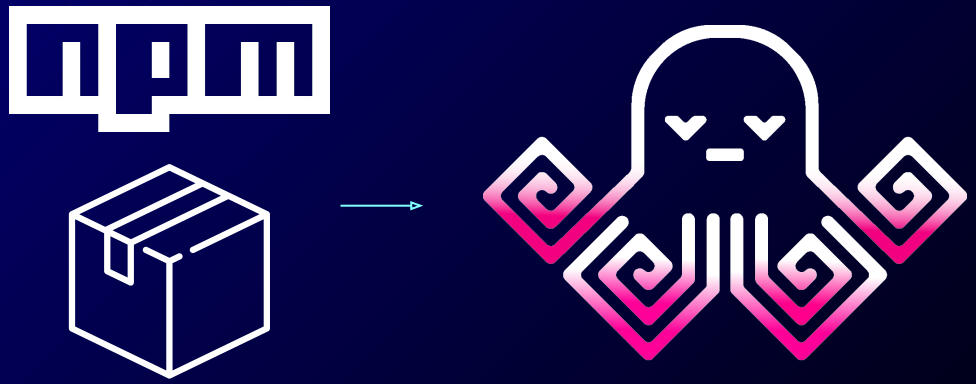
Easy installation and configuration

WAVEBINDER™ integrates easily into existing projects.

Just run npm install wave-binder to start.

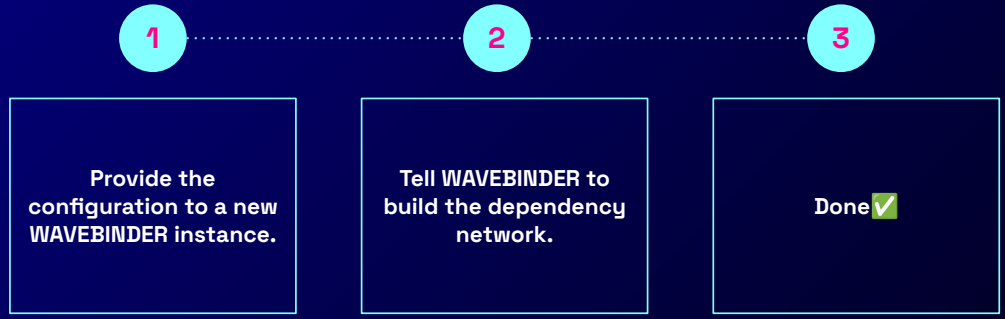
With a few steps, you can configure nodes and their dependencies using JSON.

Even for developers new to using nodes, the CLI tool simplifies the creation and management of data dependencies.



Functionality

Once WAVEBINDER™ is installed and your problem is modeled in the configuration structure, you are almost ready to go.




```
const wb: WaveBinder = new WaveBinder(config);
```

```
wb.tangleNodes();
```

Functionality

WAVEBINDER™ exposes:

- The methods necessary to retrieve nodes

```
const listNode = wb.getNodeByName('myListNode');
listNode.next(3);
```

```
const regionNode = wb.getNodeByName('region');
regionNode.setSelection(1);
```

```
const regionListNode = wb.getNodeByNameAndType('regionList', 'LIST');
regionListNode.children[1].setSelection(1);
```

```
public constructor(protoNodes: ProtoNode[],
                   extApis: Map<string, HttpServiceSetting>,
                   customFunctions: FunctionInstance[]) {
}

public tangleNodes(): void {
}
```

Functionality

WAVEBINDER™ exposes:

- The methods necessary to retrieve nodes
- Various ways to interact with them.

```
const regionListNode = wb.getNodeByNameAndType('regionList', 'LIST');
regionListNode.children[1].setSelection(1);
```

```
public constructor(protoNodes: ProtoNode[],
                   extApis: Map<string, HttpServiceSetting>,
                   customFunctions: FunctionInstance[]) {
}

public tangleNodes(): void {
}

public getNodes(): WaveBinderNode[] {
}

public getDataPool(): void {
}

public getNodesInfo(logDepth?: number): WaveBinderNodeView[] {
}

public getNodeByName(name: string): WaveBinderNode {
}

public getNodeByNameAndType(name: string,
                            type: string): WaveBinderNode {
```

Functionality

WAVEBINDER™ exposes:

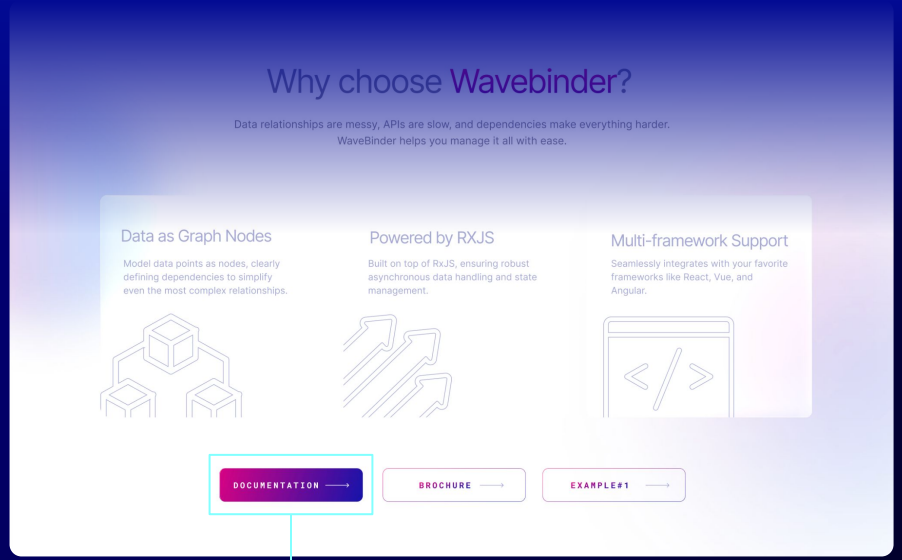
- The methods necessary to retrieve nodes
- Various ways to interact with them.

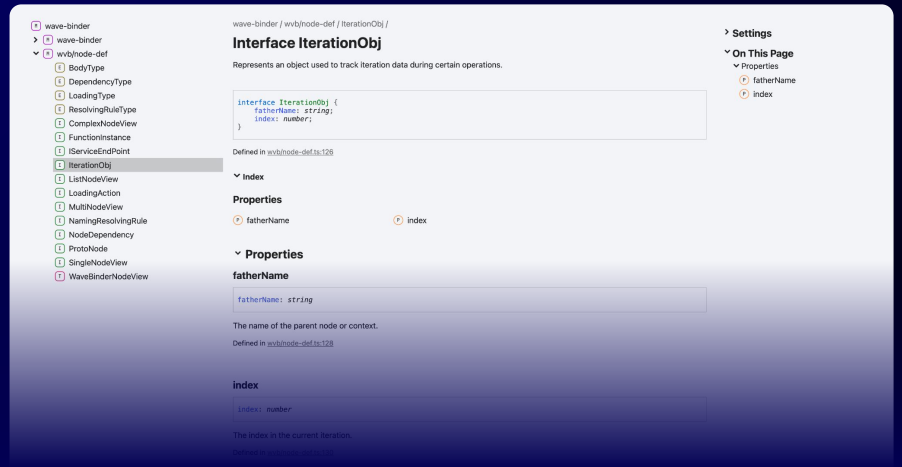
```
public getDataPool(): void {
}

public getNodesInfo(logDepth?: number): WaveBinderNodeView[] {
}

public getNodeByName(name: string): WaveBinderNode {
}

public getNodeByNameAndType(name: string,
                            type: string): WaveBinderNode {
}

public getNodeByNameAmongNodes(name: string,
                               nodes: WaveBinderNode[])
                               : WaveBinderNode {
}

public getNodeByNameAndTypeAmongNodes(name: string,
                                      type: string,
                                      nodes: WaveBinderNode[])
                                      : WaveBinderNode {
}
```

Documentation

Every aspect of **WAVEBINDER™** configuration and management is documented at **www.wavebinder.it.**

This ensures developers have a constant guide for advanced configurations and edge cases, making adoption even simpler.





4.

Why Wavebinder™ works

A powerful library, with years of experience, with the vision of abstracting complexity once and for all.

It allows anyone to focus only on what the client truly desires: business logic.

Order and precision: dependencies and actions that organize your code

For each node, you are asked to specify two very important things:

- Dependencies
- Loading action

```
"dep": [
    {
      "nodeName": "region",
      "isOptional": false,
      "onUpdate": true,
      "type": "PATH_VARIABLE"
    },
    {
      "nodeName": "province",
      "isOptional": false,
      "onUpdate": true,
      "type": "PATH_VARIABLE"
    }
 ]
```

Dependencies are simply references to other nodes.

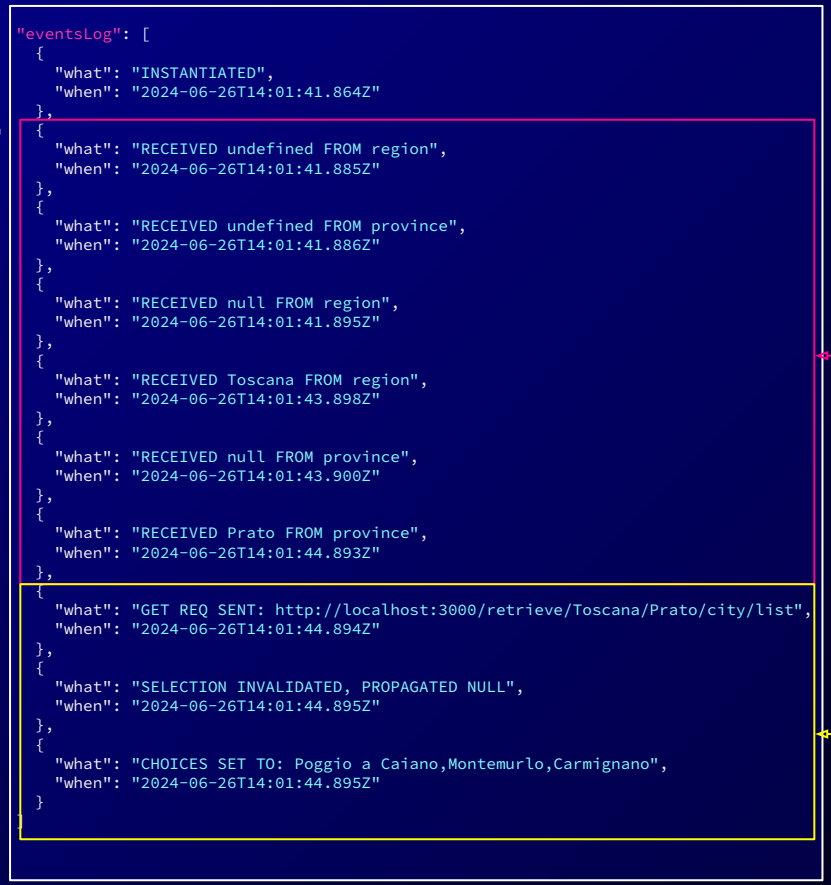
```
"la": {
  "type": "GET",
  "adr": "/region/list",
  "serviceName":"RETRIEVE_DATA"
}
```

If at any moment those reference nodes are completely and correctly loaded, the loading action is triggered.

This, of course, can trigger the loading actions of other nodes.

Constant control over every action of your interface

Every node maintains a log of what happened to it.



Updates from other nodes set as its dependencies.

Loading action executed when dependencies are satisfied.

wavebinder™

Thank you