Semantic Web 14 (2023) 135–153
DOI 10.3233/SW-222945
IOS Press

Components.js: Semantic dependency injection

Ruben Taelman [a,*], Joachim Van Herwegen [a], Miel Vander Sande [b] and Ruben Verborgh [a]

[a] *IDLab, Department of Electronics and Information Systems, Ghent University – imec, Belgium*
[b] *meemoo, Flemish Institute for Archives, Belgium*

Editor: Aidan Hogan, Universidad de Chile, Chile
Solicited reviews: Pierre-Antoine Champin, Université de Lyon, France; Esko Ikkala, Aalto University, Finland; Daniel Garijo, Ontology Engineering Group, Spain

Abstract. A common practice within object-oriented software is using *composition* to realize complex object behavior in a reusable way. Such compositions can be managed by *Dependency Injection* (*DI*), a popular technique in which components only depend on minimal interfaces and have their concrete dependencies passed into them. Instead of requiring program code, this separation enables describing the desired instantiations in declarative configuration files, such that objects can be wired together automatically at runtime. Configurations for existing DI frameworks typically only have local semantics, which limits their usage in other contexts. Yet some cases require configurations outside of their local scope, such as for the reproducibility of experiments, static program analysis, and semantic workflows. As such, there is a need for globally interoperable, addressable, and discoverable configurations, which can be achieved by leveraging Linked Data. We created *Components.js* as an open-source semantic DI framework for TypeScript and JavaScript applications, providing global semantics via Linked Data-based configuration files. In this article, we report on the Components.js framework by explaining its architecture and configuration, and discuss its impact by mentioning where and how applications use it. We show that Components.js is a stable framework that has seen significant uptake during the last couple of years. We recommend it for software projects that require high flexibility, configuration without code changes, sharing configurations with others, or applying these configurations in other contexts such as experimentation or static program analysis. We anticipate that Components.js will continue driving concrete research and development projects that require high degrees of customization to facilitate experimentation and testing, including the Comunica query engine and the Community Solid Server for decentralized data publication.

Keywords: Dependency injection, inversion of control, RDF, linked data

1. Introduction

Object-oriented (OO) programming is a highly popular paradigm within the domain of software engineering. Considering *objects* containing data and logic as primary software elements makes it easy for developers to understand software, as it makes software resemble real-world mechanisms with interacting physical objects. Most OO languages enable object composition [16], a flexible pattern for managing object relationships, where objects can be contained within other objects.

*Corresponding author. E-mail: ruben.taelman@ugent.be.

A popular technique to manage the composition of objects is called *Dependency Injection* (DI) [15]. It enables objects to *ask* for the interfaces it requires, rather than *retrieving* or *instantiating* objects implementing these interfaces itself. A DI framework is then responsible for *instantiating* and *injecting* the necessary dependencies into these objects. This technique allows objects to be very loosely coupled, as they only depend on each other via a minimal and generic interface, without depending on concrete implementations of such interfaces. In order to link these interfaces to concrete implementations, a generic DI framework can provide specific implementations where needed based on some external configuration. Since objects only communicate by strict interfaces, and specific implementations are derived from an external configuration, the specific wiring of a software application is decoupled from the application's main implementation. This allows the wiring to be altered afterwards by only modifying this configuration, which makes the application more flexible.

Configurations for existing DI frameworks are either defined directly within a programming language, or are defined declaratively within text files with a domain-specific language using syntaxes such as JSON and XML. The latter type of configuration files is better suited for use cases where no changes can be made to existing code (e.g., in the case of pre-compiled languages), when the creators of these configuration files have no programming knowledge, or when configuration files are created automatically from an external tool (e.g., a visual drag-and-drop interface). Such declarative configuration files typically have only *local* semantics, which means that they are usually only usable within the DI framework for which they were created, and for the current application only. With the power of Linked Data [3] and the Semantic Web [4] in mind, these configurations could move *beyond* their local scope, and make them globally *interoperable*, *addressable*, and *discoverable*.

To this end, we present *Components.js*, a semantic DI framework for TypeScript and JavaScript applications that gives global semantics to software configurations, hence surpassing existing dependency injection frameworks. Components.js thereby enables highly modular applications to be built that are dynamically wired based on semantic configuration files. The framework is open-source [41], is available on npm [42], and has extensive documentation [35]. Furthermore, it is being actively used as core technology within popular tools such as the Community Solid Server [48] and Comunica [40]. Within Components.js, software configurations and modules are described as Linked Data using the *Object-Oriented Components vocabulary* [47] and the *Object Mapping vocabulary* [36]. By publishing such descriptions, the composition of software (and parts thereof) can be *unambiguously identified* by IRIs and retrieved through *dereferencing*. Components.js automatically *instantiates* such software configurations, including resolving the necessary dependencies. As such, this (de)referenceability of software configurations by IRI could be beneficial in use cases such as:

Experimental research Providing the full provenance trail of used software configurations to produce experimental results for improving reproducibility.

Static program analysis Discovering conflicts or compatibility issues of different classes within software using RDF tools such as SPARQL query engines and reasoners.

Semantic workflows Automatic wiring of software using RDF tools to optimally address a specific need.

We consider this article an extension of our previous work involving describing software as Linked Data [47]. Concretely, the contributions of this work are:

- the Components.js dependency injection framework and its architecture;
- the Components-Generator.js tool for generating component descriptions for TypeScript projects;
- the Object-Oriented Components and Object Mapping vocabularies; and
- the Linked Software Dependencies (LSD) service that makes npm packages dereferenceable.

While Components.js can aid in the reproduction of experiments as one possible use case, we consider full reproducibility of experiments out of scope for this work. Instead, to enable full replication of experiments, we refer to tools such as NixOS [26] that can describe full experimental environments, where Components.js can offer more granular software configuration descriptions.

In this article, we introduce the Components.js framework as follows. In the next section (Section 2), we discuss the related work. Next, in Section 3 we explain the declarative configuration files of Components.js, followed by an architectural overview of the framework itself in Section 4. Then, in Section 5, we mention some applications where Components.js is being used. Finally, we conclude in Section 6.

2. Related work

2.1. Dependency injection

Inversion of control Inversion of Control (IoC) [15] is a general principle within software engineering that inverts the usual flow of control within software architectures. This is mostly done to reduce coupling between software components, and make the overall architecture more modular and extensible. On the one hand, traditional procedural programming gives the developer direct control of the flow of logic, where code directly invokes other code. IoC on the other hand implies the use of a framework that manages this flow, and allows custom code – that is supplied by the developer – to be invoked when the frameworks deems it necessary. This concept is typically referred to as *"The Hollywood Principle: Don't call us, we'll call you"*.

A specific technique to achieve IoC is *Dependency Injection* (DI) [15], where the framework calls constructors and methods with the right parameters. As mentioned before, DI enables relationships between objects by allowing objects to *ask* for other objects, rather than actively getting them itself. These composed objects are tied to each other only by a lightweight interface, where different implementations may be possible for each interface. Using a DI framework, specific implementations for such interfaces can be configured, after which they can be instantiated into objects, and are injected into each other using the DI framework's *assembler* to complete the wiring of the software application.

The configuration of such a wiring of objects can either be done in code, or via external configuration files. The main motivations for configurations are the strict boundaries between configuration and logic, enabling non-developers to configure the code, and taking away the need to recompile the code for pre-compiled languages. However, when dependencies are defined based on some logic such as external conditions, configuration via code may be better suited, as this can become too complex to define in declarative configuration files.

Forms of injection In practice, three main forms of DI exist through which dependencies can be injected into an object:

Constructor injection Dependency objects are passed via a class constructor.
Setter injection Dependencies are passed to an object by invoking setter methods.
Interface injection The interface of dependencies expose a method that, when invoked, injects this dependency into an object that is passed to it. Such passed objects will typically be a setter method for this.

Constructor injection is the simplest and most popular form. It requires all dependencies to be wired at construction time, which usually leads to immutable wiring. Setter injection is more flexible as wiring can be changed afterwards, but could lead to problems where not all dependencies have been fully configured yet. Interface injection is more complex, and is mainly useful if bidirectional links between dependencies and dependents need to be configured.

Advantages and disadvantages To end this section, we summarize the main advantages and disadvantages of DI.

Advantages:

- Classes are loosely coupled, which leads to **lower maintenance** effort.
- Loose coupling also leads to better **testability**, as dependencies with lightweight interfaces can easily be mocked.
- Classes have a single responsibility, which leads to **better understandable** code.
- Applications are more **flexible**, as they can be wired differently by changing a configuration file.
- Applications are more **extensible**, as different interface implementations can be created, and swapped in or out easily.
- Since classes are coded against interfaces of dependencies, they lead to more **independent** code, which is beneficial in large teams that work in parallel.

Disadvantages:

- Defining the wiring of an application via **configurations can be complex**, so good defaults must be available.
- Logic **flow is harder to follow** when debugging, which leads to the need of good documentation.
- DI frameworks can lead to **overhead** in terms of understandability, execution time and software size.

2.2. Semantic software description

Configuration-based DI frameworks make use of some form of software description. Therefore, we introduce the related work around semantic software descriptions.

Software can be described on several levels of granularity, going from a high-level package overview to a low-level description of the actual code. The Software Ontology (SWO) [24] and Description of a Project (DOAP) [50] ontology focus on the high-level management of software development, enabling the description of tools, resources, contributors and tasks. At a slightly lower level, SWO includes interfaces, algorithms, versions, and the associated provenance data, but does not reach the level of detail to describe operational code.

Ontologies that describe software configuration from a research workflow perspective are LODFlow [31], Workflow-Centric Research Objects [2] with the *Wf4Ever Research Object Model* and the Ontologies for Describing the Context of Scientific Experiment Processes [25] with the *TIMBUS Context Model* to compliment the Research Objects model. From a more generic perspective, there exist the PROV Ontology [23], the OPMW-PROV Ontology [17], and the DDI-RDF Discovery Vocabulary [5]. However, these efforts can only cover (parts of) the connection between research and software, which is insufficient for dependency injection. Such descriptions are moreover interpretive in that any given tool is subject to having multiple descriptions by different users. In contrast to the human-driven descriptions, our work both enables and accelerates the generation of machine-driven Linked Data descriptions of software modules, their components, as well as their configurations to be uniformly created. Consequently, this makes it possible to accurately describe and instantiate software experiments that can be reused and compared with unambiguously.

Much more low-level and exact is the Core Software Ontology (CSO) [29], which provides a foundational vocabulary that is designed for extensibility. This includes the distinctive concepts to describe software as code, software as object to computational hardware, and software as a running computational activity, but also Interfaces, Classes, Methods, the relationships between them, and workflow information on their invocation. Its extension, the Core Ontology of Software Components (COSC), moves closer to the topic of this article by describing interfaces and protocols of objects. Similar in scope is the Software Engineering Ontology Network (SEON) [32], which consolidates multiple ontologies for the Software Engineering field. It includes a higher Core and Foundational layer, as well as multiple domain-specific ontologies. Of particular interest is their Software Ontology (SwO) that captures the different artifacts in software. More recently, the GraphGen4Code toolkit [1] has been introduced, which provides an ontology to capture code semantics to represent classes, functions and methods. In general, these ontologies (or suites) view software from a "network of communicating concepts" perspective. This allows for exhaustive descriptions of complex software systems, but is not suited for describing class instances or aspects of modular programming (e.g., package dependencies). As such, the vocabularies that we introduced do not make use of these existing ontologies, but they do make use of parts of them where possible.

2.3. Dependency injection frameworks

The large spectrum of existing dependency injection frameworks indicates a high demand for such systems. Java likely contains the largest collection of dependency injection frameworks. Much of this stems from the strict typing, which makes it difficult to create mock objects when required for testing if the dependencies are nested in the implementation.

One of the biggest Java frameworks is Spring [34], which amongst many things, also provides dependency injection. That is one of its advantages though: many projects already use Spring for other reasons, reducing the jump required to add the dependency injection framework. It supports two ways to do the injection. The first one is through an external XML configuration file which defines all the classes and how they are linked together. The other one is with annotations in the actual code that define how the interlinking of classes should work. Google's Guice [18] is a more lightweight alternative to Spring; Dagger [11] was created to be even more lightweight than Guice.

In JavaScript, dependency injection frameworks tend to be less common because of its flexible nature. However, with the increasing popularity of TypeScript – which provides strict typings for JavaScript –, the need for dependency injection is increasing. Still, multiple frameworks are available, such as BottleJS [6], Wire [52], and

Electrolyte [14], all backed by rather small communities. One of the biggest ones, InversifyJS [21], uses annotations similar to Java frameworks to define possible injections. Unlike standard JavaScript, it requires the developer to define interfaces and types via TypeScript, thereby allowing it to make use of this extra information to correctly handle the linking. Like Guice, it also has a bindings file to link classes to interfaces.

Components.js differs from the aforementioned frameworks on different aspects. First, Components.js decouples the dependency injection layer from the software implementation via **separate configuration files**, while the other JavaScript DI frameworks use code-based configuration and thereby have a stronger coupling of these layers. It is thereby more similar to the Java-based DI frameworks that tend to rely more on external configuration files. This is the primary reason why we opted to create a new framework, instead of extending an existing one. Second, Components.js is the only framework that makes use of **RDF-based configuration files**, which makes these configurations globally *interoperable*, *addressable*, and *discoverable*. Third, regarding the form of dependency injection, Components.js makes use of **constructor injection**, just like all other discussed frameworks. Only the Spring also provides the option to make use of the other forms of injection, but constructor injection is the most popular option.

2.4. JavaScript runtime environments

The most popular runtime environment for JavaScript is Node.js [27], which allows JavaScript code to be executed outside of a Web browser. Node.js is based on the highly performant V8 engine [45] that is also used within the Chrome browser. Even though Node.js can not directly execute TypeScript code, TypeScript code can be transpiled to JavaScript so that it can be executed in Node.js. Node.js makes use of the npm [28] package manager for distributing and installing third-party packages (over 1.3 million at the time of writing). Using a package.json file, all dependencies of a module can be defined together with their version range, which are resolved from npm at install time.

Deno [12] is a relatively new runtime environment for JavaScript that aims to become a modern replacement for Node.js. It is also based on the V8 engine, but it allows both JavaScript *and* TypeScript to be executed without prior transpilation. Furthermore, there is a significant difference in the way Deno handles dependencies compared to Node.js. Code written for Node.js can only refer to dependencies by a name, and requires a package manager such as npm to bind it to a concrete package and version. Deno avoids this decoupling by allowing code to directly refer to dependencies based on URLs that can include version ranges.

For the remainder of this article, we will assume the usage of the Node.js runtime. This is because Components.js Node.js is still predominantly used at the time of writing. Nevertheless, since Deno's philosophy regarding dereferenceable modules is compatible with the dereferenceability of Components.js configurations, we will consider support for it in the future.

3. Declarative configurations

Components.js depends on two levels of configuration for enabling the wiring of software components. The first level is the creation of *components* files, which are the semantic representation of component (or class) constructors, and can usually be automatically generated. The second level is the creation of *configuration* files, which represent the actual instantiation of components based on the generated components files.

In this section, we discuss the two main vocabularies that are used within these component files, and show how configuration files can refer to them for instantiation. Next, we explain how URLs can be minted for software components, so that they become fully dereferenceable. Finally, we explain how these component files can be generated automatically from existing TypeScript code.

3.1. Object-oriented components vocabulary

Components.js distinguishes between three main concepts:

Module a software package containing zero or more components. For example, this is equivalent to a module within Node.js.

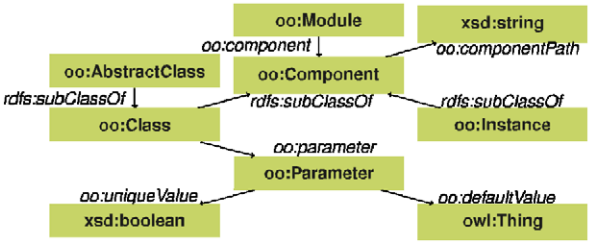

Fig. 1. Classes and properties in the *Object-Oriented Components vocabulary* (*OO*), with as prefix oo.

```
{
  "@context":[
    "https://linkedsoftwaredependencies.org/bundles/npm/componentsjs/
      ^4.0.0/components/context.jsonld",
    {"ex":"http://example.org/"}
  ],
  "@id":"ex:MyModule",
  "@type":"Module",
  "requireName":"my-module",
  "components":[
    {
      "@id":"ex:MyModule/MyComponent",
      "@type":"Class","requireElement":
      "MyComponent","parameters":[
        {
          "@id":
          "ex:MyModule/MyComponent#name",
          "unique":true,
          "range":"xsd:string"
        }
      ]
    }
  ]
}
```

Listing 1. A description of a module ex:MyModule with a single component using the JSON-LD serialization, compacted with the https://linkedsoftwaredependencies.org/bundles/npm/componentsjs/^4.0.0/components/context.jsonld context.

Component a class that can be instantiated by creating a new instance of that type with zero or more parameter values. Parameters are defined by the class constructor.

Configuration a semantic representation of an instantiation of a component into an object instance based on parameters.

These concepts are described in the programming language independent *Object-Oriented Components vocabulary* (*OO*) [47]. This vocabulary enables software components to be instantiated based on certain parameters, analog to constructor arguments in object-oriented programming. This is interpreted in the broad sense: only *classes*, *objects* and *constructor parameters* are considered. An overview is given in Fig. 1.

A module is considered a collection of components. Within object-oriented languages, this can correspond to for example a software library or an application. A component is typed as oo:Component, which is a *subclass* of rdfs:Class. The parameters to construct the component can therefore be defined as a property having that component as its domain.

Note that the vocabulary does not contain an *interface* class, because this notion does not exist in JavaScript, and it can exist in TypeScript code but, only before transpilation to JavaScript. Instead, we only define oo:AbstractClass, as both abstract classes and interfaces can be considered equivalent at the level of dependency injection.

```
{
  "@context":[
    "https://linkedsoftwaredependencies.org/bundles/npm/componentsjs/
      ^4.0.0/components/context.jsonld",
    {"ex":"http://example.org/"}
  ],
  "@type":"ex:MyModule/MyComponent",
  "ex:MyModule/MyComponent#name":"Some
  name"
}
```

Listing 2. Instantiation of ex:MyModule/MyComponent using a value for the parameter ex:MyModule/MyComponent#name.

We illustrate the usage of this vocabulary with an example in Listing 1 using the JSON-LD [22] serialization. This listing shows the definition of a new module (oo:Module) with compact IRI ex:MyModule. The name of the module is set with the compact IRI requireName, which expands to doap:name from the Description of a Project (DOAP) vocabulary [51].

Furthermore, our module contains a single class component (oo:Class) with compact IRI ex:MyModule/MyComponent. Since this is a class component (subclass of oo:Component), this means that this component is instantiatable based on parameters. Each component can refer to its path within a module using the oo:componentPath predicate (compacted as requireElement). Finally, our single component has a parameter (oo:Parameter) with compact IRI ex:MyModule/MyComponent#name that can be set when instantiating this component.

Since components and parameters are defined as RDFS vocabulary, we can instantiate components easily using the rdf:type predicate, and by using parameters as predicates on such new instances, as shown in Listing 2. Instead of passing literals as values to parameters, it is also possible to pass *other component instances* as values, thereby allowing nested component instantiations to be defined.

3.2. *Object mapping vocabulary*

As shown in the previous section, the OO vocabulary allows modules, components, and parameters to be defined, so that instances of components can be declared. However, this vocabulary only defines parameter values for component instances, but it does not define how these parameter values are used to invoke the constructor of this component. To enable this, we introduce the accompanying *Object Mapping vocabulary* (*OM*) [36]. Figure 2 shows an overview of all its classes and predicates.

The OM vocabulary makes use of the oo:constructorArguments predicate for the domain oo:Class, and thereby builds upon the OO vocabulary via the oo:constructorArguments extension point to define the class constructor's behaviour. Concretely, this new vocabulary defines a mapping between the component parameters as defined using the OO vocabulary, and the raw objects that are passed into the constructor during instantiation.

In essence, this vocabulary enables an (RDF) list of om:ObjectMapping's to be passed to the oo:constructorArguments of an oo:Class. An om:ObjectMapping represents an object containing zero or more key-value pairs, which are represented by om:ObjectMappingEntry. om:ArrayMapping is a special type of om:ObjectMapping that represents an array, where its elements can be other om:ObjectMapping's.

Building upon the OO example from Listing 1, we illustrate the usage of this vocabulary with an example in Listing 3, again using the JSON-LD serialization. The only difference with the previous example, is the addition of the constructorArguments block, which expands to oo:constructorArguments that is configured to always contain an RDF list. The constructor arguments contain a single om:ObjectMapping, which is implied by the presence of field, which expands to om:field. Since the field array contains just a single element (om:ObjectMappingEntry), it represents an object with a single key and value. The key is defined by keyRaw (expands to om:fieldName), which contains the constant name. The value is defined by value (expands to om:fieldValue), which refers to the ex:MyModule/MyComponent#name parameter.

The addition of an object mapping to a component requires no changes as to how a component is instantiated, which means that our component from Listing 3 can still be instantiated in the exact same way as the one from

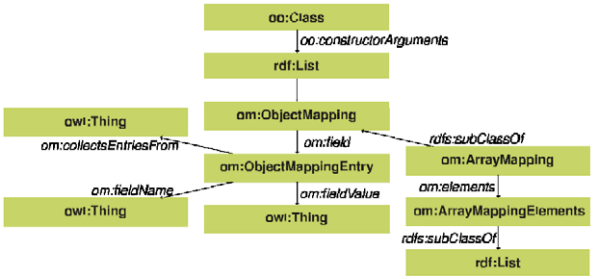

Fig. 2. Classes and properties in the *Object Mapping* vocabulary [36], with as prefix om.

```
{
  "@context":[
  "https://linkedsoftwaredependencies.org/bundles/npm/componentsjs/
    ^4.0.0/components/context.jsonld",
  {"ex":"http://example.org/"}
  ],
  "@id":"ex:MyModule",
  "@type":"Module",
  "requireName":"my-module",
  "components":[
    {
      "@id":"ex:MyModule/MyComponent",
      "@type":"Class","requireElement":
      "MyComponent","parameters":[
        {
          "@id":
          "ex:MyModule/MyComponent#name",
          "unique":true,
          "range":"xsd:string"
        }
      ]
    }
  ],
  "constructorArguments":[
    {
      "fields":[
        {
          "keyRaw":"name",
          "value":"ex:MyModule/MyComponent#name"
        }
      ]
    }
  ]
}
```

Listing 3. A description of a module ex:MyModule with a single component having constructor arguments using the JSON-LD serialization, compacted with the https://linkedsoftwaredependencies.org/bundles/npm/componentsjs/^4.0.0/components/context.jsonld context.

Listing 1. The only difference now, is that we are able to determine how exactly the parameter values are to be used for invoking the component constructor. For example, the instantiation of Listing 2 corresponds to the following code in JavaScript:

```
new
MyComponent({name:'Some name'})
```

A real-world example of the combined usage of the OO and OM vocabularies can be found at https://linkedsoftwaredependencies.org/bundles/npm/@comunica/core/1.21.1/components/Actor.jsonld.

3.3. Dereferenceability

In previous work [47] we introduced the Linked Software Dependencies (LSD) service [46], which makes all resource URLs within components files fully dereferenceable.

Since our current focus is on enabling dependency injection for JavaScript, this LSD service provides Linked Data subject pages for *all* packages within the npm package manager [28] for JavaScript. For example, the URL https://linkedsoftwaredependencies.org/bundles/npm/@comunica/core/1.21.1 is an identifier for the @comunica/core package at version 1.21.1. Listing 4 shows a snippet of the JSON-LD contents when dereferencing this URL.

This LSD service allows creators of components files to automatically mint LSD-based URLs for their packages, which will automatically become dereferenceable as soon as these packages are published to npm. The LSD service thereby removes the dereferenceability responsibility from package developers that want to use dependency injection via Components.js, but do not have the will or ability to make their component files dereferenceable themselves. The LSD service is not required for the functioning of the Components.js framework, so developers are not obligated to publish their package to npm or mint their own URLs if they do not have this desire. But since publishing packages to npm in a common practise within the JavaScript community, we consider this a low barrier to entry.

This dereferenceability is beneficial for enabling querying execution within and across component files. For example, it enables using the follow-your-nose principle to analyze class inheritance chains of certain modules. Another example in the domain of reproducibility is the ability to analyze which config parameters had the largest influence on the performance of a system, assuming that the experimental results have also been linked to the semantic configuration.

```
{
  "@context":[
    "https://linkedsoftwaredependencies.org/contexts/npm.jsonld",
    {"lsd":"https://linkedsoftwaredependencies.org/"}
  ],
  "@type":"doap:Version",
  "@id":"lsd:bundles/npm/%40comunica%2Fcore/1.21.1",
  "name":"@comunica/core",
  "version":"1.21.1",
  "description":"Lightweight,semantic and modular actor framework",
  "dependencies":{
    "@comunica/types":"lsd:bundles/npm/%40comunica%2Ftypes/%5E1.21.1",
    "immutable":"lsd:bundles/npm/immutable/%5E3.8.2"
  },
  "maintainers":[
    {
      "email":"mailto:rubensworks@gmail.com",
      "@id":"lsd:users/npm/rubensworks"
    }
  ],
  "dcterms:license":{
      "@id":"https://spdx.org/licenses/MIT.html",
      "rdfs:label":"MIT"
  },
  "lsd:scripts/npm/test":{
      "@id":"lsd:bundles/npm/%40comunica%2Fcore/1.21.1/scripts/test"
  }
}
```

Listing 4. Part of the JSON-LD contents of https://linkedsoftwaredependencies.org/bundles/npm/@comunica/core/1.21.1.

```
/**
 * This is a great class!
 */
export class MyClass extends OtherClass{
  /**
   * @param paramA - My parameter
   */
   constructor(paramA: boolean, paramB: number){
  }
}
```

Listing 5. TypeScript class that is used as input to Components-Generator.js.

The long-term sustainability of the LSD service and its minted URLs is guaranteed by Ghent University, which places a strong emphasis on ensuring that data is preserved in the long term. In the unlikely event that the LSD service would experience downtime, all applications that make use of Components.js will still remain functional, because the Components.js framework does not rely directly on the dereferenceability of these URLs.

3.4. Generation from TypeScript

For larger projects, the manual creation of components files for all classes in the project can require significant manual effort, and can therefore become error-prone. For projects that make use of a strongly-typed language, such as TypeScript, all required information to create such components files is in fact already available implicitly via the source code files. In order to minimize manual effort for such projects, we provide the open-source tool *Components-Generator.js* [37] (Zenodo [39]) for TypeScript projects.

Concretely, this tool can be installed into any TypeScript project. When its command-line script is invoked, it scans all exported TypeScript classes within this project, and generates corresponding components files for them. In doing so, it preserves information that is important for dependency injection, such as component extensions via class inheritance relationships and parameter types with constructor arguments mapping via class constructors.

For example, assuming an npm package named my-package containing the single TypeScript class from Listing 5, Components-Generator.js will generate the components file in Listing 6.

A real-world example of such conversion can be seen in the Community Solid Server [48] project. For example, the CorsHandler TypeScript class (https://github.com/solid/community-server/blob/9b6eab27bc4e5ee25d1d3c6ce5972e83db90c650/src/server/middleware/CorsHandler.ts#L31) is converted to the components file at https://linkedsoftwaredependencies.org/bundles/npm/@solid/community-server/2.0.0/dist/server/middleware/CorsHandler.jsonld.

4. Dependency injection framework

Building on top of the declarative configurations that were explained in previous section, we now discuss Components.js, which is a system that can interpret these configurations for enabling dependency injection within JavaScript/TypeScript projects. In this section, we first explain the main architecture, followed by the most relevant implementation details.

4.1. Architecture

The primary functional requirement of our architecture is the ability to perform dependency injection based on the configuration files from previous section. Concretely, this involves *parsing* the configuration files, *interpreting* them, and *instantiating* the necessary components. Next to these functional needs, we took the following non-functional requirements into account when designing the architecture:

- Usability: Developers using the framework should only be required to interact with a single entry point.
- Extensibility/Maintainability: The system should be robust against different future functional requirements.

```
{
  "@context":["https://linkedsoftwaredependencies.org/bundles/npm/my-
    package/
      ^1.0.0/components/context.jsonld"
  ],
  "@id":"npmd:my-package",
  "components":[
    {
      "@id":"ex:MyFile#MyClass",
      "@type":"Class",
      "requireElement":"MyClass",
      "extends":"ex:OtherFile#OtherClass",
      "comment":"This is a great class!",
      "parameters":[
        {
          "@id":"ex:MyFile#MyClass_paramA",
          "range":"xsd:boolean",
          "comment":"My parameter",
          "unique":true,
          "required":true
        },
        {
          "@id":"ex:MyFile#MyClass_paramB",
          "range":"xsd:integer",
          "unique":true,
          "required":true
        }
      ],
        "constructorArguments":[
         {"@id":"ex:MyFile#MyClass_paramA"},
         {"@id":"ex:MyFile#MyClass_paramB"}
      ]
    }
  ]
}
```

Listing 6. Components file that is generated by Components-Generator.js from the TypeScript file from Listing 5.

- Performance: Parts of the architecture that are prone to performance issues should be cacheable.

To meet these requirements, the Components.js dependency injection tool goes through three main phases:

- Loading: Initialization of DI components, discovery of modules, and loading of configuration files.
- Preprocessing: Handling of constructor arguments before construction.
- Construction: Instantiation of JavaScript classes based on configuration files.

These three phases are handled by the ComponentsManager, which acts as the main entrypoint of the framework as can be seen in Fig. 3 in the Appendix. This manager class is constructed via a static build method, via which custom options can be passed, such as a callback for loading modules and configuration files. To meet the *usability* requirement, this is the only part that most users of the framework will interact with.

For the sake of clarity, all UML architecture diagrams that we include in this article only contain simplified representations of the actual classes. So there may be minor differences when comparing the diagrams with the actual source code.

Hereafter, we explain these three phases in more detail.

4.1.1. Loading

When the ComponentsManager is being built, the loading phase will be initiated, which will make use of the classes within the load package. The most important classes within this package are shown in Fig. 4 in the Appendix.

This phase aims to contain all major I/O operations, which could be expensive on slow disks and/or in large projects. This allows later phases to purely work on memory. Furthermore, the loaded information is designed to be cacheable, which means that software that require repeated invocations may optimize the loading phase by caching certain parts, which thereby meets the *performance* requirement.

The ModuleStateBuilder is a class that is responsible for scanning the current JavaScript project and its dependencies. The main objective of this class is to build an IModuleState, that contains information such as the paths to available components and dependencies.

ComponentRegistry and ConfigRegistry are classes that are exposed via a callback to invokers of Components Manager.build(). These classes respectively enable modules and configurations to be registered, after those modules and configurations will be loaded.

4.1.2. Preprocessing

Before a configuration is instantiated during the construction phase, it always goes through a preprocessing phase. Concretely, this involves processing all parameters and constructor arguments, for which the most relevant classes and interfaces are shown in Fig. 5 in the Appendix. To meet the *extensibility* and *maintainability* requirements, the architecture allows different parameters and constructor arguments handlers to be injected. This makes the architecture more robust against currently unforeseen functional requirements regarding the handling of parameters and constructor arguments.

IConfigPreprocessor is an interface that represents a preprocessing algorithm for a configuration, and can have multiple implementations.

ConfigPreprocessorComponent is a preprocessor that is able to determine what component is being instantiated within a configuration. It will check if the linked component exists, and it will validate all passed parameters. For this parameter validation, the ParameterHandler class is used, which works based on a list of IParameterProperty Handler's. For instance, parameter property handlers exist for validating the range of parameters, checking the uniqueness, handling default values, and more.

ConfigPreprocessorComponentMapped is another preprocessor that builds upon ConfigPreprocessorComponent, so that it *additionally* handles constructor arguments as defined by the Object Mapping vocabulary. Concretely, after validating parameters, it will handle the constructor arguments recursively using a list of IConstructorArguments ElementMappingHandler's. These handlers can handle specific types of constructor arguments and parameters, such as the conversion of om:ObjectMapping to an object, and the conversion of om:ArrayMapping to an array.

The end-result of the preprocessing phase is a configuration that represents the raw constructor call of a class, together with the required arguments.

4.1.3. Construction

The construction phase is responsible for instantiating a configuration. The main classes for this are shown in Fig. 6 in the Appendix. Like before, the *extensibility* and *maintainability* requirements also apply here regarding the way in which things are constructed, for which we also provide the ability to inject different handlers.

ConfigConstructorPool is the main entrypoint that is used when a user instantiates a configuration via ComponentsManager.instantiate(). Before actually instantiating a config, it will first check if it had been instantiated before, in which case it returns it from a cache. This may occur for nested configurations that reuse the same component in different places. If the config has not been instantiated before, it will first go through the preprocessing phase as explained in the previous section, and then the processed config will be passed on to the ConfigConstructor.

The ConfigConstructor is able to convert the representation of a class constructor call into an actual constructor call to obtain an object. For this, the arguments of the constructor are first converted into actual objects, which is done via a list of IArgumentConstructorHandler's. For example, handlers exist to handle primitive values such as strings and numbers, arrays, and references to other components (which requires a recursive call to ConfigConstructorPool). Once the arguments have been resolved, the constructor can be applied to obtain the final instantiated object.

By default, the ConfigConstructor assumes that configurations are instantiated via the CommonJS JavaScript standard, which is primarily used by the Node.js runtime environment. However, Components.js has been designed to handle different kinds of instantiation, which can be done via different IConstructionStrategy's. For instance, this allows the framework to be compatible with other upcoming JavaScript standards such as JavaScript modules.

4.2. Implementation

Components.js has been implemented in TypeScript, and is available on GitHub [41] and Zenodo [43] under the MIT license. At the time of writing, the latest release is at version 4.4.1, which is published via the npm package manager [42].

Due to the critical nature of this framework, it is being tested thoroughly. At the time of writing, it consists of 538 unit tests, which reach a test coverage of 100%.

Components.js is being maintained by IDLab via software projects that make use of this framework. Furthermore, Components.js is part of the Comunica Association [9], which is a non-profit organization that aims to ensure the long-term sustainability of certain open-source projects. A sustainability of this project is available on GitHub [38].

Finally, in-depth documentation [35] is available, which explains how to create component and configuration files, and how to invoke the DI tool.

5. Usage

A measure of the usage of an open-source project without the use of any tracking software is a picture that is always incomplete. Nevertheless, we analyze the usage of Components.js in this section on two aspects: empirical usage via available metrics, and in-use analysis of specific projects. We discuss these two aspects hereafter.

5.1. Usage metrics

As the source code of Components.js is hosted on GitHub [41], it is possible to inspect the usage of this project within other projects hosted on GitHub. As of August 2 2021, there are 9 GitHub projects that depend on Components.js directly, and 268 that depend on it indirectly via transitive dependencies. This shows that Components.js is primarily used as a core library to support larger projects that have a broad usage.

The npm package manager [42] from which Components.js can be installed offers us additional insights. For the week of July 26 2021 until August 1 2021 (the last completed week before writing this section) there were 5.351 downloads, which is an average number when comparing it to previous weeks. However, there are outliers for which Components.js has weekly downloads peak up to around 200.000 downloads.

While these GitHub and npm metrics give us *some* insight into the usage of Components.js, they are incomplete, as projects may be hosted on other source code platforms such as GitLab, Bitbucket, or even private instances. Furthermore, direct downloads from npm are also incomplete, as downstream users may use bundling tools such as Webpack [49] to incorporate the Components.js source code directly within their library, which makes downloads of that library not go via the Components.js npm package anymore. On the other hand, automated downloads by bots (e.g. for mirror services) may artificially increase the download number, without actually representing real usage. Therefore, we conclude that the metrics reported here are merely an estimate.

5.2. In-use analysis

In the previous section, we provided an informed estimate as to *how much* Components.js is being used. In this section, we provide an analysis of *in what way* Components.js is being used in four real-world projects: Community Solid Server, Handlers.js, Digita Identity Proxy, and Comunica.

5.2.1. Community solid server

The Community Solid Server [48] is a server-side implementation of the Solid specifications [33], which provides a basis for the Solid decentralization effort. When such a server is hosted, it allows users to create their own personal storage space (pod) and identity, so that this data can be used within any external Solid application. This server is written in TypeScript, and is being developed by Inrupt [20] and imec [19], which includes authors of this article.

This server makes use of dependency injection because a primary goal of the server is to be as flexible as possible, so that developers can easily modify the capabilities of the server, or even add additional capabilities. This is

especially useful in the context of research, where new components can be added to the server for experimentation, before they are standardized and become a part of the Solid specifications. To enable this level of flexibility, all components within this server are loosely coupled, and are wired via customizable Components.js configuration files.

Since the Community Solid Server makes use of TypeScript, it is able to make use of the Components-Generator.js tool as explained before in Section 3, which avoids the need to manually create components files, and thereby significantly simplifies the usage of Components.js within this project. At the time of writing, this server contains 246 components that can be customized via specific parameters, and wired together to form a server instance with specific capabilities.

5.2.2. Handlers.js

Handlers.js [44] aims to provide a comprehensive collection of generic logic classes, that can be wired together via the composition pattern. While this project is still under development, it already provides numerous handlers and services pertaining to data flows, storage, logging, error handling, as well as logic about serving data over HTTP (routing, CORS, content negotiation, ...). This project is written in TypeScript, and is being developed by Digita [13].

In contrast to the Community Solid Server, Handlers.js is not meant to be usable by itself as stand-alone tool. Instead, it is an accompanying library that can be used by other tools. The components within Handlers.js are meant to capture common patterns within projects that depend on composition-based components, so that they can be reused by other projects that make use of DI frameworks such as Components.js. While Components.js is the primary DI framework this library was designed for, it does not strictly depend on it thanks to the loosely coupling of the Components.js DI layer and software implementations.

Handlers.js also make use of the Components-Generator.js tool to convert TypeScript classes into components files. At the time of writing, this project exposes 40 components that range from abstract logic flows to specific ones for setting up a simple HTTP server. Since components within Components.js have global semantics, these components can be easily reused across projects.

5.2.3. Digita identity proxy

The Digita Identity Proxy (not public at the time of writing) is a Solid-OIDC [8]-compliant proxy server that acts as a modular, and easily configurable compatibility layer for classic OIDC [30] Identity Providers. It enables Solid apps to authenticate at Solid pod servers with these existing identity services, without any necessary modification. This project is also written in TypeScript, and is under development by Digita [13].

Several components exists that enable additional functionality of Solid-OIDC, which can be plugged into the proxy when the need exists. With Components.js, these components can be easily configured and plugged in via a configuration file.

5.2.4. Comunica

Comunica [40] is another project that makes use of Components.js at its core. Comunica is a highly modular SPARQL query engine that has been designed to be a flexible research platform for SPARQL query execution. It has been written in TypeScript, and is developed by Ghent University, by authors of this article.

The modular nature of Comunica calls for a dependency injection framework due to its actor-mediator-bus paradigm. All logic within Comunica is placed within small actors, which are registered on task-specific buses following the publish-subscribe pattern. In order to select a certain actor on a bus for achieving a certain task, the mediator pattern is applied, which allows different actors to be selected based on different actions. These actors, buses, and mediators are loosely coupled with each other, and are wired together via Components.js configuration files. For example, this allows users of Comunica to create and plug in a different algorithm for resolving a certain SPARQL query operator.

At the time of writing, Comunica does not yet make use of the Components-Generator.js tool, as it was developed before Components-Generator.js was created. Therefore, all components files within Comunica are created manually, which shows that Components.js is flexible in this regard.

As Comunica is a research platform for research around query execution, the ability to *reproduce* experiments is crucial. This is where the benefit of Components.js becomes especially apparent. It is often the case that research

articles with experimental results only report on the used software, without mentioning the exact version and configuration that was used. When using a Components.js configuration file, the necessary semantics for accurately replicating such experiments are available as Linked Data. The reproducibility of experimental results is often considered to be even more important than the research article itself [7], as the article can be considered to be merely *advertising of the scholarship*. For example, the Comunica research article [40] contains an experiment workflow (*https://comunica.github.io/Article-ISWC2018-Resource/#evaluation-workflow*) that is backed by the used Components.js configuration files.

6. Conclusions

After more than four years of development, Components.js has become a stable Dependency Injection framework for TypeScript and JavaScript projects, and has seen a significant uptake by popular tools that make use of it as core technology. It enables the primary tasks of a DI framework, but thanks to its semantic configuration files, it also brings with it the power of Linked Data and the Semantic Web for enabling globally interoperable and discoverable configurations. Using the Linked Software Dependencies service, components and configurations become dereferenceable and citable, which allows software configurations to be shared easily with others, which is for example beneficial for improving the reproducibility of software experiments.

The previous section has shown that Components.js provides significant value in real-world applications. On the one hand, tools such as the Community Solid Server and Comunica allow developers and researchers to rewire these applications based on their specific needs. On the other hand, applications by companies such as Digita depend on this flexibility for making logic changes via configuration files, as they want to enable their clients to make changes by only modifying the configuration files, since their clients are sometimes non-technical people that have limited programming knowledge.

We can recommend Components.js for TypeScript/JavaScript projects that have at least a subset of the following characteristics:

- Architectures that require **high modularity and flexibility**;
- Need to modify wiring of components **without changing code**;
- Need for ability to **share wiring configurations** with others;
- Managing and including **configurations across different projects**;
- Using **configurations in other contexts**.

As with all DI frameworks, Components.js comes with the downside that for large applications, configurations can become complex and logic flow may be harder to follow. In order to mitigate these risks, we recommend a structured management of configuration files, which may involve splitting up configuration files based on an architecture's primary subsystems, which is the approach followed by large projects such as Community Solid Server and Comunica.

The dereferenceability of software configurations by IRI is also an important benefit of the Components.js framework. In the introduction, we mentioned that this dereferenceability could be beneficial for experimental research, static program analysis, and semantic workflows. So far, we only have concrete proof of the experimental research use case as shown in Section 5.2.4. We hope to see examples of the other use cases making use of this functionality in future work.

In future work, we do not foresee the need for any major changes or additions within the Components.js framework itself, aside from keeping up with new language features from JavaScript and TypeScript. However, all large projects that make use of Components.js have identified the need for better tooling to create and manage configuration files. For example, the Comunica project is developing a graphical user interface [10] to visually customize the wiring of the engine, which can then be exported into a reusable configuration file. Since Components.js configurations make use of the Linked Data principles, it is possible to create a generic user interface to create such configuration files for any project that makes use of Components.js. Furthermore, since components and configuration files are largely programming language-independent, it is possible to create equivalent implementations of Components.js for other OO languages such as Java and C#. Another venue that deserves investigation is the

task of automatically letting the Linked Software Dependencies service execute the Components-Generator.js on all TypeScript projects that do not provide component files yet, which could open up a huge domain on injectable components.

In general, Components.js gives us the necessary foundation for building next-level applications that depend on high flexibility, such as smart agents. These applications are crucial for environments such as Linked Data and the Semantic Web, which require and benefit from this level of flexibility. Therefore, DI frameworks such as Components.js pave the road towards a world with more flexible applications.

Acknowledgements

We thank Wouter Termont for sharing his insights into the usage of Components.js within the products of Digita. The described research activities were funded by Ghent University and imec. Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1274521N).

Appendix. Architectural diagrams

This appendix section contains the architectural diagrams that were discussed in Section 4.1. Figure 3 contains the main entrypoint of the framework, Fig. 4 represents the loading phase, Fig. 5 represents the preprocessing phase, and Fig. 6 represents the construction phase.

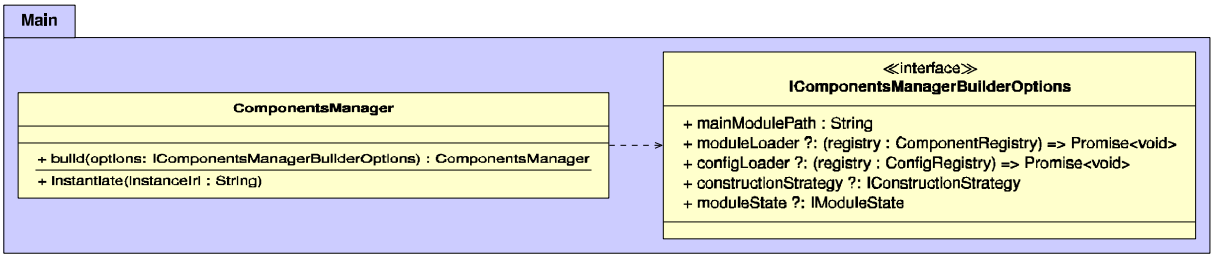

Fig. 3. UML diagram of the classes within the main package, which contains the main entrypoint of the framework.

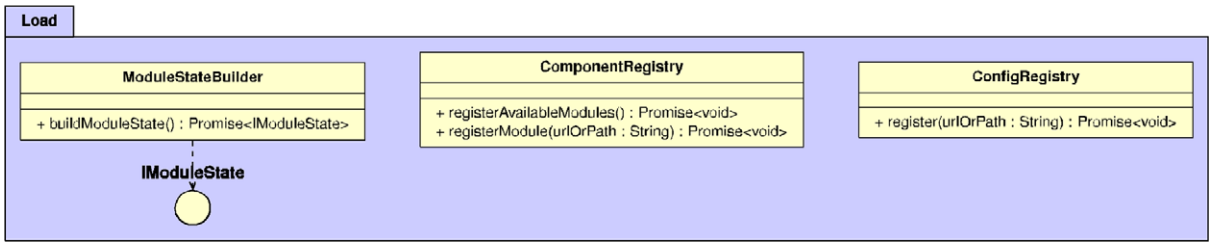

Fig. 4. UML diagram of the classes within the load package, which are responsible for loading components and configurations.

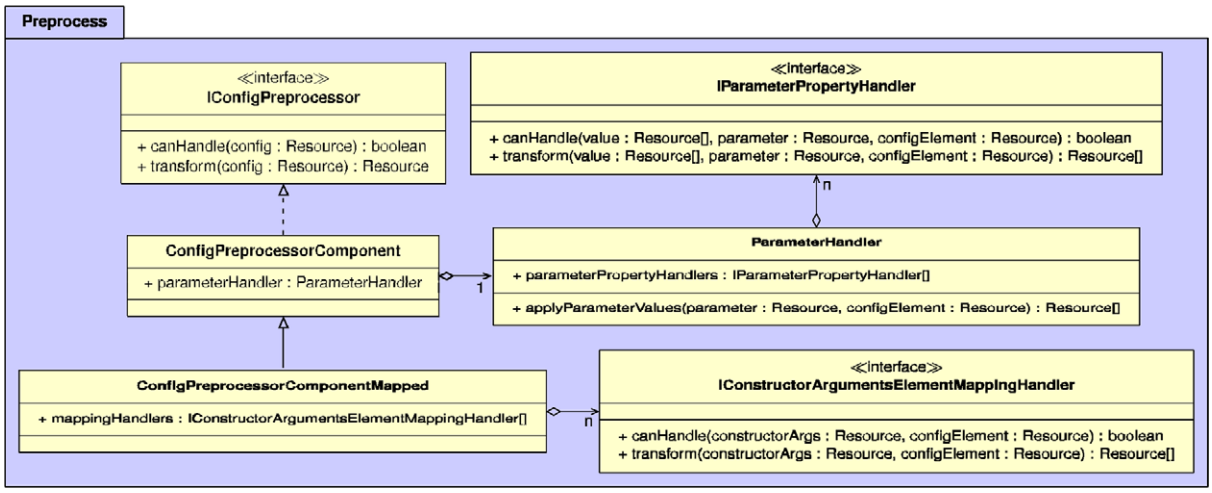

Fig. 5. UML diagram of the classes within the preprocess package, which are responsible for preprocessing config parameters and constructor arguments.

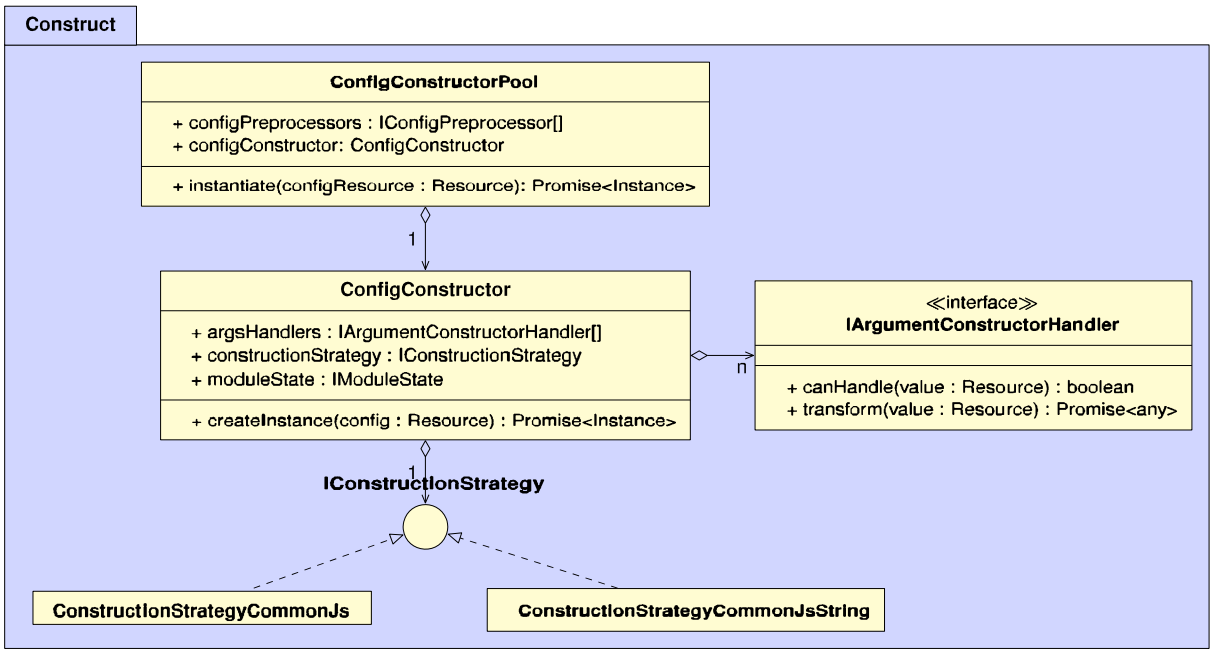

Fig. 6. UML diagram of the classes within the construct package, which are responsible for instantiating configs according to a certain strategy.

References

[1] I. Abdelaziz, J. Dolby, J. McCusker and K. Srinivas, A toolkit for generating code knowledge graphs, 2021. doi:10.1145/3460210.

[2] K. Belhajjame, J. Zhao, D. Garijo, M. Gamble, K. Hettne, R. Palma, E. Mina, O. Corcho, J.M. Gómez-Pérez, S. Bechhofer, G. Klyne and C. Goble, Using a suite of ontologies for preserving workflow-centric research objects, *Web Semantics: Science, Services and Agents on the World Wide Web* **32** (2015), 16–42. doi:10.1016/j.websem.2015.01.003.

[3] T. Berners-Lee, Linked data, 2006, https://www.w3.org/DesignIssues/LinkedData.html.

[4] T. Berners-Lee, J. Hendler and O. Lassila, The semantic web, *Scientific American* **284** (2001), 28–37.

[5] T. Bosch, R. Cyganiak, J. Wackerow and B. Zapilko, DDI-RDF discovery vocabulary, 2015, http://rdf-vocabulary.ddialliance.org/discovery.html.

[6] BottleJS, 2021, https://github.com/young-steveo/bottlejs.
[7] J.B. Buckheit and D.L. Donoho, WaveLab and reproducible research, Stanford University, 1995, https://git.its.aau.dk/CLAAUDIA/teach_reproducibility/raw/commit/dbea465c0d10bca50b0cca23fd93afd0ffea08dc/litt/.
[8] A. Coburn, elf Pavlik and D. Zagidulin (eds), Solid-OIDC, 2021, https://solid.github.io/authentication-panel/solid-oidc/.
[9] Comunica Association, 2021, https://comunica.dev/association/.
[10] Comunica packager source code, 2020, https://github.com/comunica/comunica-packager.
[11] Dagger, 2021, https://github.com/google/dagger.
[12] Deno, 2021, https://deno.land/.
[13] Digita – Solid for enterprises, 2021, https://www.digita.ai/.
[14] Electrolyte, 2021, https://github.com/jaredhanson/electrolyte.
[15] M. Fowler, Inversion of control containers and the dependency injection pattern, 2004, https://martinfowler.com/articles/injection.html.
[16] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Elements of Reusable Object-Oriented Software. Design Patterns*, Addison-Wesley Publishing Company, Massachusetts, 1995.
[17] D. Garijo and Y. Gil, A new approach for publishing workflows: Abstractions, standards, and linked data, in: *Proceedings of the 6th Workshop on Workflows in Support of Large-Scale Science*, ACM, 2011, pp. 47–56. doi:10.1145/2110497.2110504.
[18] Guice, 2021, https://github.com/google/guice.
[19] imec R&D, 2021, https://www.imec-int.com/en.
[20] Inrupt, 2021, https://inrupt.com/.
[21] InversifyJS, 2021, https://github.com/inversify/InversifyJS.
[22] JSON-LD 1.1: A JSON-based serialization for linked data, https://www.w3.org/TR/json-ld/.
[23] T. Lebo, S. Sahoo and D. McGuinness, Prov-O: The PROV ontology. W3C, 2013, https://www.w3.org/TR/prov-o/.
[24] J. Malone, A. Brown, A.L. Lister, J. Ison, D. Hull, H. Parkinson and R. Stevens, The software ontology (SWO): A resource for reproducibility in biomedical data analysis, curation and digital preservation, *Journal of biomedical semantics.* **5** (2014), 25. doi:10.1186/2041-1480-5-25.
[25] R. Mayer, T. Miksa and A. Rauber, Ontologies for describing the context of scientific experiment processes, in: *10th International Conference on e-Science*, IEEE, 2014. doi:10.1109/escience.2014.47.
[26] NixOS – NixOS Linux, 2021, https://nixos.org/.
[27] Node.js, 2021, https://nodejs.org/en/.
[28] npm, 2021, https://www.npmjs.com/.
[29] D. Oberle, S. Grimm and S. Staab, An ontology for software, in: *Handbook on Ontologies*, Springer, 2009, pp. 383–402. doi:10.1007/978-3-540-92673-3_17.
[30] OpenID Connect, 2020, https://openid.net/connect/.
[31] S. Rautenberg, I. Ermilov, E. Marx, S. Auer and A.-C. Ngonga Ngomo, LODFlow: A workflow management system for linked data processing, in: *Proceedings of the 11th International Conference on Semantic Systems*, ACM, 2015, pp. 137–144. doi:10.1145/2814864.2814882.
[32] F.B. Ruy, R. de Almeida Falbo, M.P. Barcellos, S.D. Costa and G. Guizzardi, SEON: A software engineering ontology network, in: *European Knowledge Acquisition Workshop*, Springer, 2016, pp. 527–542. doi:10.1007/978-3-319-49004-5_34.
[33] Solid technical reports, 2021, https://solid.github.io/specification/.
[34] Spring, 2021, https://spring.io/.
[35] R. Taelman, Components.js documentation, 2021, https://componentsjs.readthedocs.io/.
[36] R. Taelman, Object mapping vocabulary, 2017, https://linkedsoftwaredependencies.org/vocabularies/object-mapping.
[37] R. Taelman, Components.js generator source code, 2021, https://github.com/LinkedSoftwareDependencies/Components-Generator.js/.
[38] R. Taelman, Components.js sustainability plan, 2017, https://github.com/LinkedSoftwareDependencies/Components.js/wiki/Sustainability-Plan.
[39] R. Taelman, J.V. Herwegen and P. Heyvaert, LinkedSoftwareDependencies/Components-Generator.js: 2.6.1, 2021. doi:10.5281/zenodo.5644902.
[40] R. Taelman, J. Van Herwegen, M. Vander Sande and R. Verborgh, Comunica: A modular SPARQL query engine for the web, in: *Proceedings of the 17th International Semantic Web Conference*, 2018. doi:10.1007/978-3-030-00668-6_15.
[41] R. Taelman, J. Van Herwegen and R. Verborgh, Components.js source code, 2021, https://github.com/LinkedSoftwareDependencies/Components.js.
[42] R. Taelman, J. Van Herwegen and R. Verborgh, Components.js package, 2021, https://www.npmjs.com/package/componentsjs.
[43] R. Taelman, R. Verborgh, J.V. Herwegen and L. Noterman, LinkedSoftwareDependencies/Components.js 2.0.0, 2018. doi:10.5281/zenodo.1243988.
[44] W. Termont, Handlers.js, 2021, https://github.com/digita-ai/handlersjs.
[45] V8 JavaScript engine, 2021, https://v8.dev/.
[46] J. Van Herwegen and R. Taelman, Linked software dependencies, 2021, https://linkedsoftwaredependencies.org/.
[47] J. Van Herwegen, R. Taelman, S. Capadisli and R. Verborgh, Describing configurations of software experiments as linked data, in: *Proceedings of the 1st SemSci Workshop*, 2017.
[48] J. Van Herwegen, R. Verborgh and R. Taelman, Community solid server source code, 2021, https://github.com/solid/community-server/.
[49] webpack, 2021, https://webpack.js.org/.
[50] E. Wilder-James, Description of a project, 2017, http://use-fulinc.com/ns/doap.

[51] E. Wilder-James, Description of a project, 2017, https://github.com/ewilderj/doap/wiki.
[52] Wire, 2021, https://github.com/cujojs/wire.