# An OWL/RDF Ontology for Declarative User Interface Generation

**Daniel Theisges da Costa**
Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC),
Instituto Politécnico de Bragança, Portugal

**Júlio Costa**
Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC),
Instituto Politécnico de Bragança, Portugal

**Paulo Jorge Teixeira Matos**
Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC),
Instituto Politécnico de Bragança, Portugal

**Ramon Gallinad Corti**
Dept. de Telecomunicació i Enginyeria de Sistemes, Escola d'Enginyeria,
Universitat Autonoma de Barcelona, Spain

**Ramon Vilanova Arbos**
Dept. de Telecomunicació i Enginyeria de Sistemes, Escola d'Enginyeria,
Universitat Autonoma de Barcelona, Spain

**Maria João Varanda Pereira**
Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC),
Instituto Politécnico de Bragança, Portugal

## Abstract

Maintaining user interfaces for data-intensive applications is costly because interface code is tightly coupled to the underlying domain model: any change to an operation or data schema forces manual updates across multiple layers of the software stack. We present *OntologyToUI*, a framework that treats ontologies written in OWL (Web Ontology Language)/RDF (Resource Description Framework) as a Domain-Specific Language (DSL) for user-interface specification. Five modular base ontologies (schema-level definitions of abstract classes and properties, with no domain-specific instances) define the metamodel, covering communication technologies, external code references, data metadata, business operations, and User Interface (UI) element types. Domain-specific *view ontologies*, written in Turtle, instantiate the `OntologyToUI-UserInterface` ontology to declare UI elements. A generic React-based rendering engine interprets view ontologies at runtime through SPARQL (SPARQL Protocol and RDF Query Language) queries and a configuration-driven dispatch table, mapping element types to components without any domain-specific logic. We validate the framework through two case studies developed in the scope of the Smart_LEM project [11]: a Local Electricity Market management platform, where eleven views were created with zero application-code changes; and a real-time Internet of Things (IoT) dashboard that introduces a new Message Queuing Telemetry Transport (MQTT)-subscribed element type, demonstrating the framework's extensibility.

**2012 ACM Subject Classification** Software and its engineering → Domain specific languages; Software and its engineering → Model-driven software engineering; Information systems → Ontologies

**Keywords and phrases** OWL, RDF, domain-specific language, user interface generation, model-driven engineering, React, SPARQL, MQTT

**Digital Object Identifier** 10.4230/OASIcs.SLATE.2026.14

**Supplementary Material** *Software (NPM Package source code)*: `https://github.com/NewDaniC/OntologyToUI` [4]; archived at `swh:1:dir:840123e5e0c6e231e56bad1d41fe75ba1890f803`





**Funding** This work was supported by FCT - Fundação para a Ciência e Tecnologia, I.P. by projects: CeDRI, UID/05757/2025 (DOI: https://doi.org/10.54499/UID/05757/2025) and UID/-PRR/05757/2025 (DOI: https://doi.org/10.54499/UID/PRR/05757/2025); SusTEC, LA/P/0007/2020 (DOI: https://doi.org/10.54499/LA/P/0007/2020) and funded by CETP, the Clean Energy Transition Partnership under the 2022 CETP joint call for research proposals, co-funded by the European Commission (GAN° 101069750). Specifically, the financial assistance of the FCT grant CETP/0003/2022 (DOI https://doi.org/10.54499/CETP/0003/2022) towards this research is hereby acknowledged.

## 1 Introduction

In this section, the context, motivation and main goals of this work are presented, along with an overview of the article structure.

Modern software applications expose their domain through two tightly coupled artefacts: a backend Application Programming Interface (API) and a frontend user interface. Any evolution of the domain (a new operation, an additional parameter, or a changed response schema) requires coordinated updates in both artefacts. In domains such as energy management or IoT monitoring, where operational rules and data models evolve frequently, this coupling becomes a bottleneck that consumes developer resources and introduces the risk of inconsistency across the software stack [2].

Model-Driven Engineering (MDE) addresses this class of problem by raising the primary development artefact from code to models, from which executable software can be generated automatically [2]. Applied to user interfaces, Model-Based User Interface Development (MBUID) proposes that interfaces be specified at an abstract level and derived through systematic transformations [3]. However, existing MBUID approaches rely on proprietary modeling languages that require specialized tooling and do not exploit the formal semantics, standardized query mechanisms, and reasoning capabilities offered by Semantic Web ontologies.

We propose *OntologyToUI*: a framework in which ontologies written in OWL/RDF serve as the single source of truth for both API generation and UI specification. OntologyToUI is developed in the scope of Smart_LEM [11], a business model-oriented platform for Local Electricity Markets; the complete architecture of the system, spanning API generation and UI specification, is illustrated in Figure 1. From a DSL perspective [7], the base ontologies constitute the *grammar*, each view ontology is a *program* written in that language, and a generic rendering engine is the *interpreter* that executes it. The key design goal is the *Generic Rendering Principle*: the engine must accept any ontology conforming to the established schema and render only what is declared therein, without hardcoding element lists or making domain assumptions.

OntologyToUI has been designed with three objectives:

- **Declarative UI specification**: allow domain experts to author view interfaces in Turtle without programming knowledge, by instantiating a controlled vocabulary of UI element types.
- **Generic interpretation**: implement a rendering engine that processes any conformant view ontology dynamically, driven entirely by SPARQL queries and a dispatch table, with no hardcoded domain logic.
- **Extensibility**: support new UI element types by adding one entry to a registry and one React component, without modifying the rendering pipeline or any existing component.

This article is structured as follows. Section 2 reviews the state of the art in model-driven UI development and ontologies. Section 3 presents OntologyToUI as a Smart_LEM platform module that extracts information from ontologies and generates specifications of the UI

elements composing the interface, detailing the vocabulary of UI element types and the five-ontology architecture. Section 4 presents the rendering architecture. Section 5 validates the framework through two case studies. Section 6 concludes and discusses future work.

## 2 State of the Art

In this section, several works are contextualized and related with the proposed objectives, covering domain-specific languages and model-driven engineering, model-based user interface development, and the use of ontologies as a modelling substrate.

### 2.1 DSLs and Model-Driven Engineering

Mernik, Heering and Sloane [7] provide a systematic taxonomy of when and how DSLs should be developed, identifying ontology-based representations as a viable alternative to grammar-based external DSLs. A key advantage of DSLs is ease of use: a domain expert can describe system behaviour using domain vocabulary without needing to understand the underlying implementation. Brambilla, Cabot and Wimmer [2] argue that raising the abstraction level from code to models reduces redundancy and the risk of inconsistency across software layers, making MDE particularly valuable in domains with frequent domain evolution.

### 2.2 Model-Based User Interface Development

The Unifying Reference Framework of Calvary et al. [3] organises UI specification across four abstraction levels (task model, abstract UI, concrete UI, final UI) and remains the dominant conceptual model for multi-target UI generation. MARIA [9] and UsiXML [3] are concrete realisations: Extensible Markup Language (XML)-based DSLs with their own toolchains supporting multi-level transformation. Ruiz, Serral and Snoeck [10] evaluate model-driven UI generation approaches and note that most systems require proprietary notation, do not scale easily to evolving domains, and address the UI layer in isolation without co-designing the backend from the same model.

### 2.3 Ontologies as a Modelling Substrate

Gruber [5] defines an ontology as a formal, explicit specification of a shared conceptualisation. The OWL 2 specification [8] and SPARQL provide machine-processable semantics and a standardised query mechanism over RDF graphs. Prior work has used ontologies to annotate existing UI elements [1], but to our knowledge no published system uses a view-level OWL ontology as a directly executable UI specification interpreted by a generic runtime engine that hardcodes no element types, no domain operations, and no assumptions about which views exist, rendering only what the view ontology declares. OntologyToUI addresses this gap by treating the World Wide Web Consortium (W3C) OWL/RDF stack as the modelling substrate, obtaining formal semantics and standardised querying without custom parsers or compilers.

## 3 The OntologyToUI DSL

In this section, the OntologyToUI DSL is presented, covering its architecture, the DSL vocabulary of UI element types, and how a view ontology is authored.

## 3.1 Five-Ontology Architecture

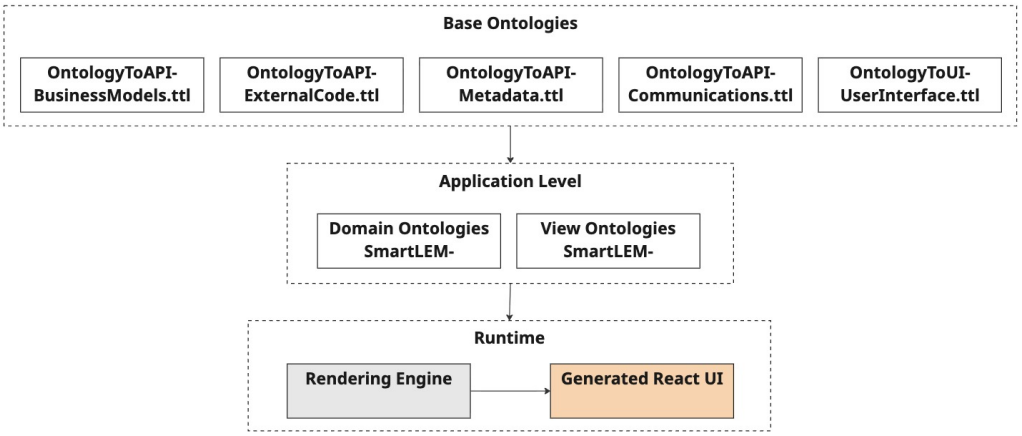


**Figure 1** Five-ontology architecture of the Smart_LEM framework. Base ontologies define the metamodel vocabulary; application-level ontologies instantiate it for a specific domain; the rendering engine interprets view ontologies at runtime.

The framework is organised in two levels. At the *base level*, five modular OWL 2 ontologies define the abstract vocabulary (Figure 1):

- **OntologyToAPI-Communication**: models seven communication modalities (Representational State Transfer (REST) API, database, file system, message queue, WebSocket, Remote Procedure Call (RPC), service mesh).
- **OntologyToAPI-ExternalCode**: models references to Python functions and their library dependencies.
- **OntologyToAPI-Metadata**: models data fields, types, and source bindings.
- **OntologyToAPI-BusinessModel**: ties operations to input parameters, required metadata, and external code.
- **OntologyToUI-UserInterface**: defines UI element types and their configurable properties.

At the *application level*, *domain ontologies* instantiate the base vocabulary for a specific system, and *view ontologies* compose UI element instances that reference those domain objects.

This paper focuses on the last ontology, `OntologyToUI-UserInterface`, which is dedicated to interface construction. The remaining ontologies are described in detail on the Smart_LEM project webpage [11].

## 3.2 UI Element Vocabulary

`OntologyToUI-UserInterface` defines eight concrete UI element types, shown in Table 1. Every element individual must carry a `ui:order` property that controls its rendering position within the view. The rendering engine queries for individuals of each type at runtime; the set of elements present in a view is therefore determined entirely by the ontology.

**Table 1** UI element types defined by `OntologyToUI-UserInterface`.

| Element type | Key properties | React component |
|---|---|---|
| `ui:Label` | `hasText`, `hasType`, `hasLevel` | `LabelComponent` |
| `ui:Form` | `hasTitle`, `hasBusinessModel` | `FormComponent` |
| `ui:Table` | `hasBusinessModel`, `hasColumnMetadata` | `TableComponent` |
| `ui:LinePlot` | `hasTitle`, `hasIndependentMetadata`, | `GraphComponent` |
| `ui:BarPlot` | `hasDependentMetadata`, `hasSeriesAlias` | `GraphComponent` |
| `ui:LiveTable` | `hasTitle`, `hasWebSocketUrl`, `hasTopicFilter` | `LiveTableComponent` |
| `ui:Image` | `hasMetadata`, `hasBusinessModel` | `ImageComponent` |

## 3.3 Writing a View Ontology

Following Mernik et al.'s taxonomy [7], OntologyToUI is a Smart_LEM platform module that uses an *ontology-based external DSL* to define the UI elements to be generated: the base ontologies form the grammar, each view ontology is a program, and the rendering engine is the interpreter. Domain experts author programs in Turtle without programming knowledge.

Listing 1 shows a minimal program: a three-element view that generates a form for creating an energy community (a local group of prosumers that collectively share and trade energy). Each `ui:Label` declares `hasType` (heading style: `"title"` or `"paragraph"`), `hasText` (the displayed text), and `ui:order` (render position). The `ui:Form` element references a `BusinessModel` individual defined in the domain ontology; the rendering engine resolves the reference at runtime to discover the operation's parameters, Hypertext Transfer Protocol (HTTP) endpoint, and response schema, without any of that information being present in the view ontology.

**Listing 1** View ontology for an energy community creation page (SmartLEM-CreateEC_View.ttl).

```
@prefix ui:   <http://www.cedri.com/OntologyToUI-UserInterface#> .
@prefix EC:   <http://www.cedri.com/SmartLEM-EC-Operations#> .
@prefix view: <http://www.cedri.com/SmartLEM-CreateEC-View#> .

view:PageTitle   rdf:type ui:Label ;
    ui:hasType   "title" ;
    ui:hasText   "Create Energy Community" ;
    ui:hasLevel   1 ;
    ui:order      1 .

view:Description rdf:type ui:Label ;
    ui:hasType   "paragraph" ;
    ui:hasText   "Fill the form to create a new LEM community." ;
    ui:order     2 .

view:CreateECForm rdf:type ui:Form ;
    ui:hasTitle         "Energy Community Details" ;
    ui:hasBusinessModel EC:CreateEnergyCommunity ;
    ui:order            3 .
```

This separation means that the grammar (base ontologies) can be extended with new element types and the interpreter (rendering engine) can be upgraded independently of any existing program (view ontology). These abstractions eliminate the need to write React component code manually: a domain expert authors a Turtle file and the rendering engine generates the interface automatically.

# 4 Rendering Architecture

This section describes the UI generation process based on the view ontology, covering the three-layer architecture and the Generic Rendering Principle that ensures domain-independent interpretation.

## 4.1 Three-Layer Design

The system is organised into three layers (Figure 2).

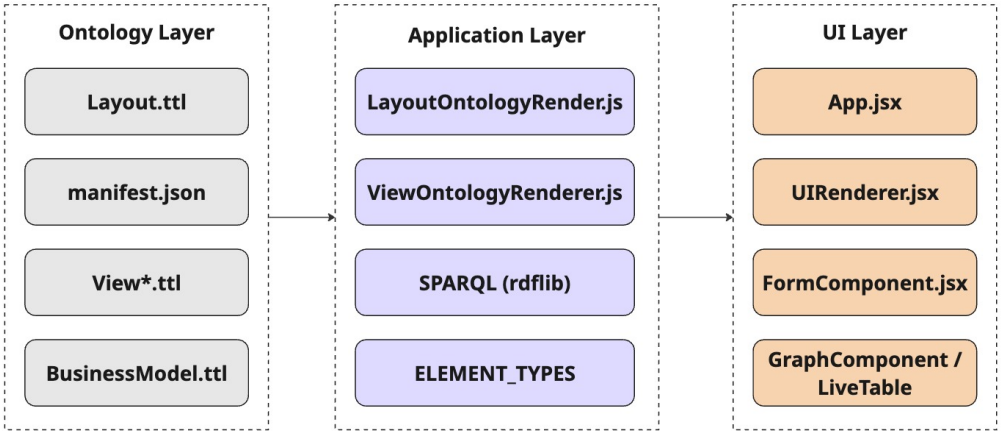


**Figure 2** Three-layer architecture of OntologyToUI. Turtle files in the Ontology Layer are parsed into an `rdflib` graph store; SPARQL queries extract element definitions, which are dispatched to React components.

The **Ontology Layer** contains all domain knowledge as Turtle files: a layout ontology defines the page shell; a `manifest.json` lists available views; view ontologies declare UI elements; and domain ontologies define business operations and data sources.

The **Application Layer** is entirely generic. The `rdflib` library (v2.3.5) parses ontology files into an in-memory RDF graph store. `LayoutOntologyRenderer.js` queries for layout configuration by type without hardcoding any individual Uniform Resource Identifier (URI). `ViewOntologyRenderer.js` runs one SPARQL SELECT query per element family and assembles a flat list of element descriptors sorted by `ui:order`. Using SPARQL instead of manual triple traversal keeps the extractor declarative and domain-independent: the same queries work over any conformant ontology without custom parsing code.

The **UI Layer** is built on React, Ant Design (v6.0), and Plotly.js. `UIRenderer.jsx` maps each element descriptor to a React component through a `COMPONENT_MAP` dispatch table and renders the list in order. No component contains domain knowledge; they depend only on the props passed from the extractor.

## 4.2 Generic Rendering Principle

The `ELEMENT_TYPES` configuration object in `ViewOntologyRenderer.js` is the sole coupling point between the grammar (ontology schema) and the interpreter (rendering engine). Each entry specifies the `rdf:type` URI to query for and the ontology properties to extract:

```
LiveTable: {
  type:       UI('LiveTable'),
  properties: {
    title:       UI('hasTitle'),
    wsUrl:       UI('hasWebSocketUrl'),
    topicFilter: UI('hasTopicFilter'),
  },
},
```

Each entry declares the `rdf:type` URI to match and the ontology properties to extract from each individual; the rendering engine uses this configuration to build component descriptors dynamically, with no knowledge of any specific domain. Adding a new element type requires adding one entry to `ELEMENT_TYPES`, one entry to `COMPONENT_MAP`, and creating the corresponding React component; no other file is modified. The current layout model renders elements sequentially, stacked vertically by `ui:order`; this reflects the characteristics of the energy simulation systems targeted by the framework, where forms, tables, and charts are naturally presented as a vertical stack.

## 5 Case Studies

We validate the framework through two applications from qualitatively different domains and integration paradigms.

### 5.1 SmartLEM: Local Electricity Market Management

SmartLEM [11] is a business model-oriented platform for developing Local Electricity Markets (LEMs) [6] and accelerating clean energy transition. It provides a digital solution with business models targeting three actor classes: prosumers (participants who both produce and consume energy), aggregators (entities that coordinate groups of prosumers in energy markets), and LEM market participants. The platform supports market-based trading mechanisms, auctions, and settlement procedures, enabling prosumers and communities to trade energy surplus and flexibility (the capacity to adjust energy production or consumption on demand) while increasing the use of renewable energy sources. Its frontend generates all views from ontologies; its backend REST API is likewise generated from the same business model ontologies.

#### Domain ontologies

`SmartLEM-ECModels.ttl` defines eleven `bm:BusinessModel` individuals covering community creation, member and asset enrolment, energy simulation, and data retrieval. Each individual is bound to input parameters (types: `str`, `int`, `float`, `UploadFile`), a Python wrapper function, and output metadata individuals that describe the response schema.

#### View ontologies

Eleven view ontologies were authored, one per application page. Table 2 summarises their distribution across functional categories and element types.

### Separation of concerns

All eleven views were created by authoring Turtle files and adding entries to `manifest.json`. The following source files were not modified during the entire development of the SmartLEM views: `App.jsx`, `ViewOntologyRenderer.js`, `UIRenderer.jsx`, `FormComponent.jsx`, `GraphComponent.jsx`, `TableComponent.jsx`. This confirms the Generic Rendering Principle: domain experts can author new views in Turtle without touching application code.

**Table 2** SmartLEM view ontologies grouped by functional category.

| Category | Views | UI Elements Used |
|---|---|---|
| Community Management | CreateEC, GetEC | Label, Form |
| Member & Asset Mgmt | AddMember, AddAsset, GetMember | Label, Form |
| Simulation | SimulateEC | Label, Form |
| Data Visualisation | EC ConsProd, EC Market, Member ConsProd, Member Asset, Weather Graph | LinePlot, Table |

Figure 3 shows the *Create Energy Community* view, one of the eleven views generated entirely from Turtle ontologies. The interface supports the full lifecycle of a Local Electricity Market: managing communities, members and assets; running energy simulations; and visualising consumption, production, market balance, and weather data.

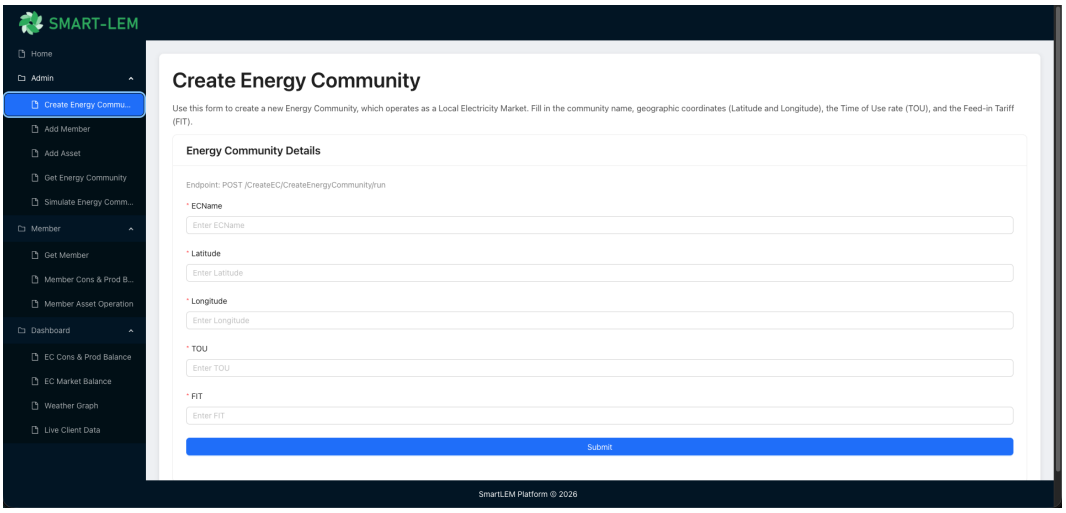


**Figure 3** Screenshot of the SmartLEM *Create Energy Community* view generated by OntologyToUI.

## 5.2 Smart_LEM: Real-Time IoT Dashboard

This case study demonstrates that the framework handles event-driven, push-based data sources with no REST API layer. The data producer is a Python agent-based smart home simulator that publishes timestamped sensor readings (energy, water, solar, battery, climate) to an MQTT broker (Mosquitto) via two topic patterns:

`clients/<house_id>/data` and `clients/<house_id>/status`

The broker exposes a WebSocket listener on port 9001, which is the connection point for the browser-side component.

### Extending the framework

None of the existing element types fits the real-time subscription scenario. A new type, `ui:LiveTable`, was introduced by modifying exactly two files (`ELEMENT_TYPES` and `COMPONENT_MAP`) and adding one React component (`LiveTableComponent.jsx`). No other file in the codebase was changed.

### View ontology

The complete dashboard specification is given in Listing 2. It declares four individuals: two `ui:Label` elements and two `ui:LiveTable` elements targeting different MQTT topic filters on the same broker endpoint. The view contains no `ui:BusinessModel` reference and no link to any REST ontology; the specification is self-contained in 29 lines of Turtle.

**Listing 2** Complete IoT dashboard view ontology (SmartLEM-Dashboard_View.ttl).

```
@prefix ui:   <http://www.cedri.com/OntologyToUI-UserInterface#> .
@prefix view: <http://www.cedri.com/SmartLEM-Dashboard-View#> .

view:PageTitle  rdf:type ui:Label ;
    ui:hasText  "IoT Client Monitor" ;
    ui:hasType  "title" ; ui:hasLevel 1 ; ui:order 1 .

view:SubTitle   rdf:type ui:Label ;
    ui:hasText  "Real-time sensor data via MQTT" ;
    ui:hasType  "paragraph" ; ui:order 2 .

view:ClientDataTable  rdf:type ui:LiveTable ;
    ui:hasTitle        "Live Sensor Data" ;
    ui:hasWebSocketUrl "ws://127.0.0.1:9001" ;
    ui:hasTopicFilter  "clients/+/data" ;
    ui:order           3 .

view:ClientStatusTable  rdf:type ui:LiveTable ;
    ui:hasTitle        "Client Status" ;
    ui:hasWebSocketUrl "ws://127.0.0.1:9001" ;
    ui:hasTopicFilter  "clients/+/status" ;
    ui:order           4 .
```

### LiveTableComponent

The React component connects to the broker using the MQTT protocol over WebSocket via the `mqtt` npm package, subscribes to the configured topic filter, and for each incoming message appends the parsed JavaScript Object Notation (JSON) payload, augmented with a client identifier and a timestamp, to a sliding window of at most 20 rows. *Schema self-discovery*: columns are computed at render time by collecting all keys present across received payloads, so the table columns are inferred automatically from the received data, with no schema declaration required in the view ontology.

### Integration

This view is registered in `manifest.json` alongside the eleven SmartLEM views. From the application's perspective the view is indistinguishable from any other: it is discovered, loaded, parsed, and rendered by the same pipeline.

Figure 4 shows the resulting dashboard. Two live tables update automatically as sensor data arrives: one displays real-time climate and energy readings per client, and the other shows the connection status of each smart home.

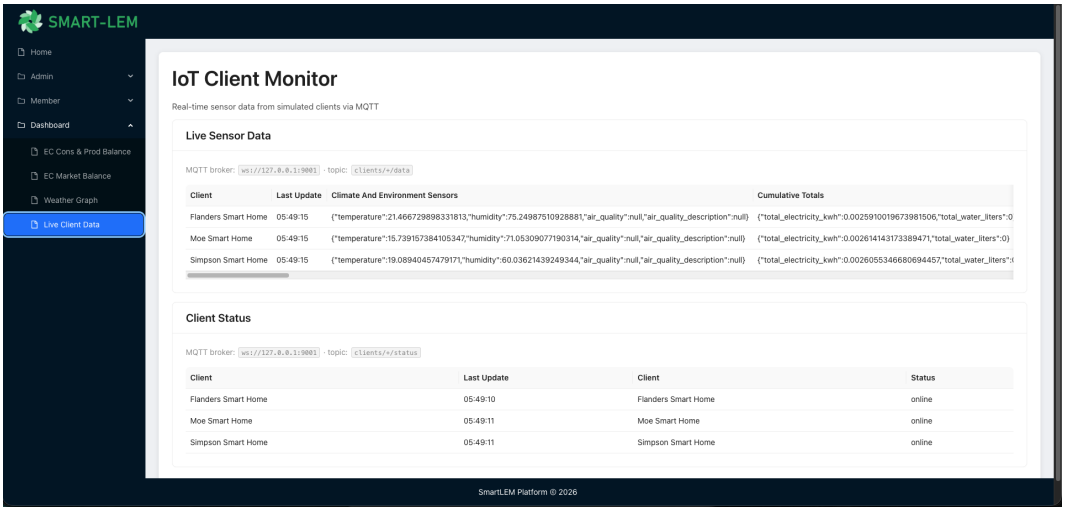


**Figure 4** Screenshot of the IoT dashboard generated by OntologyToUI, showing real-time sensor data received via MQTT.

## 6 Conclusion

We presented OntologyToUI, an ontology-based DSL framework for automatic user interface generation. The Smart_LEM five base ontologies define a reusable metamodel; view ontologies written in Turtle serve as declarative UI programs; and a generic React rendering engine interprets them through SPARQL without domain-specific logic.

Two case studies validate complementary properties of the framework. SmartLEM validates the Generic Rendering Principle: eleven views of four element-type families were created without modifying any application source file, confirming clean separation between domain knowledge, rendering logic, and presentation. The IoT dashboard case study validates extensibility: a new real-time MQTT-subscribed element type was added by changing two files and adding one component, while the rest of the codebase remained intact. Together, the case studies confirm that the framework is domain-independent, spanning REST API-connected UIs and event-driven push-based UIs.

### Future work

Shapes Constraint Language (SHACL) validation shapes would allow structural errors in view ontologies to be reported before rendering. A visual ontology editor, where domain experts compose views by dragging and dropping element types onto a canvas, would generate the corresponding Turtle automatically, removing the need for RDF syntax knowledge. A quantitative evaluation comparing lines of Turtle against lines of equivalent handwritten React code would provide stronger empirical evidence of authoring effort reduction.

## References


1. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001. doi:10.1038/scientificamerican0501-34.
2. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2 edition, 2017.
3. Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003. doi:10.1016/S0953-5438(03)00010-9.
4. Daniel Theisges da Costa, Júlio Costa, Paulo Jorge Teixeira Matos, and Maria João Tinoco Varanda Pereira. SmartLEM / OntologyToUI. Software, version v1.0., swhId: swh:1:dir:840123e5e0c6e231e56bad1d41fe75ba1890f803 (visited on 2026-07-17). URL: https://github.com/NewDaniC/OntologyToUI, doi:10.4230/artifacts.27314.
5. Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. doi:10.1006/knac.1993.1008.
6. Esther Mengelkamp, Johannes Gärttner, Kerstin Rock, Scott Kessler, Lawrence Orsini, and Christof Weinhardt. Designing microgrid energy markets: A case study: The Brooklyn Microgrid. *Applied Energy*, 210:870–880, 2018. doi:10.1016/j.apenergy.2017.06.054.
7. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005. doi:10.1145/1118890.1118892.
8. Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. OWL 2 web ontology language: Structural specification and functional-style syntax. W3c recommendation, W3C, 2012. URL: https://www.w3.org/TR/owl2-syntax/.
9. Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction*, 16(4), 2009. doi:10.1145/1614390.1614394.
10. Jenny Ruiz, Estefanía Serral, and Monique Snoeck. Evaluating user interface generation approaches: Model-based versus model-driven development. *Software & Systems Modeling*, 18:2753–2776, 2019. doi:10.1007/s10270-018-0698-x.
11. Smart_LEM Consortium. Smart local energy market (Smart_LEM) project, 2022. Accessed: April 2026. URL: https://smartlem.ase.ro/.