IMPLEMENTATION

The complete tool developed for this project is available on GitHub at: https://github.com/CSIRO-enviro-informatics/shacl-form

The software developed to fulfil the requirements of the project took the form of two Python applications. One generates the SHACL webform. It is responsible for parsing the SHACL Shapes file from RDF into memory, organising and interpreting the data, and filling Jinja2 templates to create the webform. The other is a Flask web server that demonstrates how this webform may be used. It displays the webform and accepts any data submitted, storing it in RDF Turtle format.

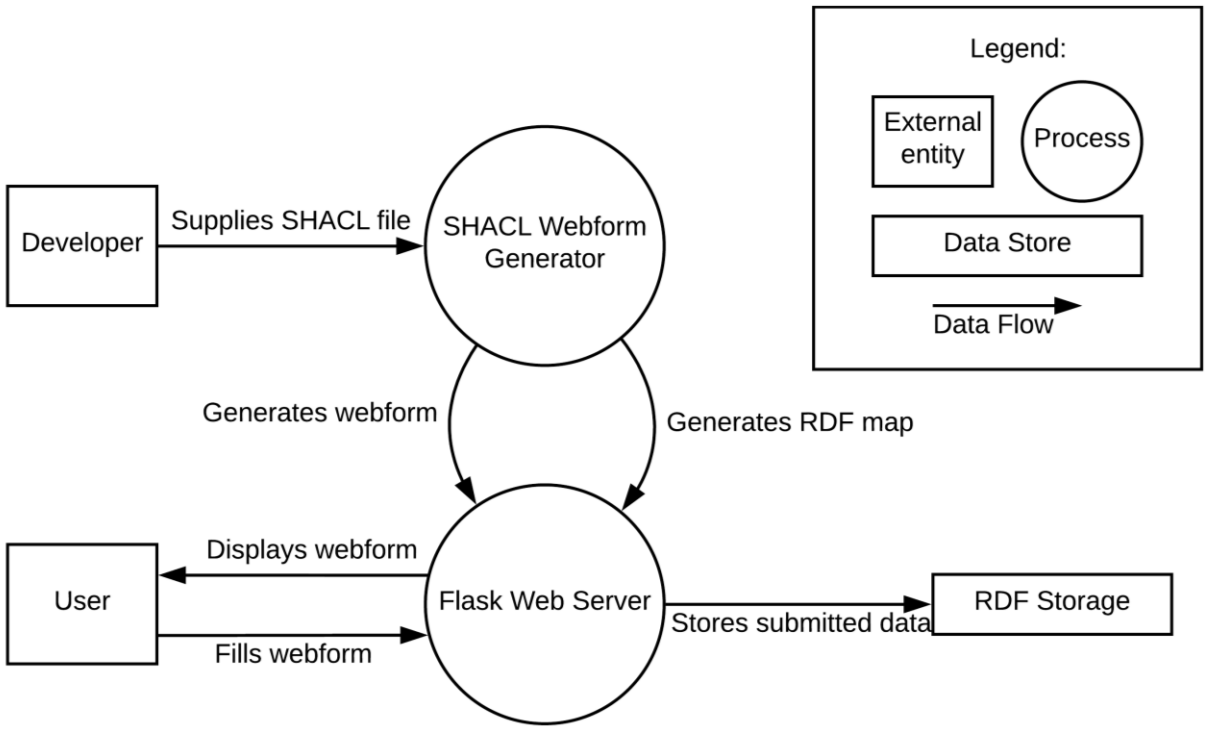

Figure 1. A data flow diagram displaying the inputs and outputs of the applications developed

Figure 1 expresses the flow of information between the SHACL webform generator and the Flask web server implemented to demonstrate it. A developer wishing to generate and host a webform will supply a SHACL Shapes file that defines the data model. Using this input, the SHACL webform generator creates a HTML file containing the webform, and an RDF map. These files are supplied to the Flask web server, which displays the form to a user and stores submitted data in RDF.

1 SHACL Webform Generator

The SHACL webform generator is responsible for interpreting the SHACL Shapes file to generate a HTML webform. It is divided into two packages: ‘rdfhandling’ and ‘rendering’. The ‘rdfhandling’ package uses the RDFLib Python module to read information from the supplied SHACL Shapes file, returning an in-memory representation of data model specified. The ‘rendering’ package uses Jinja2 to render a template using the data model provided by the ‘rdfhandling’ package.

1.1 RDF Handling - SHACL file structure

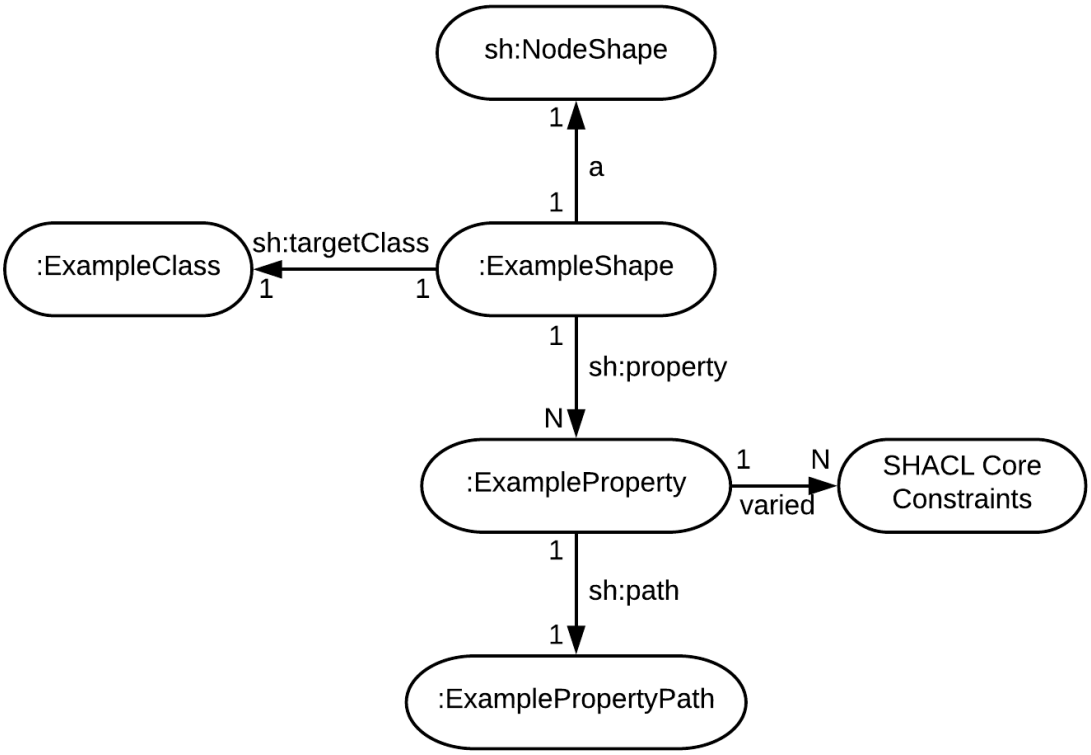

Figure 2. Node diagram showing the expected structure of a SHACL Shapes file expressing some constraints

Recalling that the SHACL Shapes file is provided in RDF format, demonstrates the expected structure of the file as a node diagram. The SHACL shape, shown as ‘:ExampleClass’, is used to identify all the linked nodes as part of a certain SHACL Shape. The target class, shown as ‘:ExampleClass’, represents the entity that the webform will collect information about (e.g. a person, a dog, or a model of car). A SHACL shape may have a number of properties which are linked to the shape using the ‘sh:property’ predicate, shown in this diagram as

‘:ExampleProperty’. This property must have a path, which defines what the property is (e.g. a name, a colour, etc.). Each property may have one or many constraints associated with it, such as ‘sh:minCount’ or ‘sh:datatype’.

1.2 RDF Handling - Implicit Target Class Declaration

There is an alternate structure that implicitly declares the target class, as shown in Figure 3. A shape that has both type ‘sh:NodeShape’ and type ‘rdfs:Class’ is a target class of itself, eliminating the need for an explicit target class node. This application can support either type of declaration.

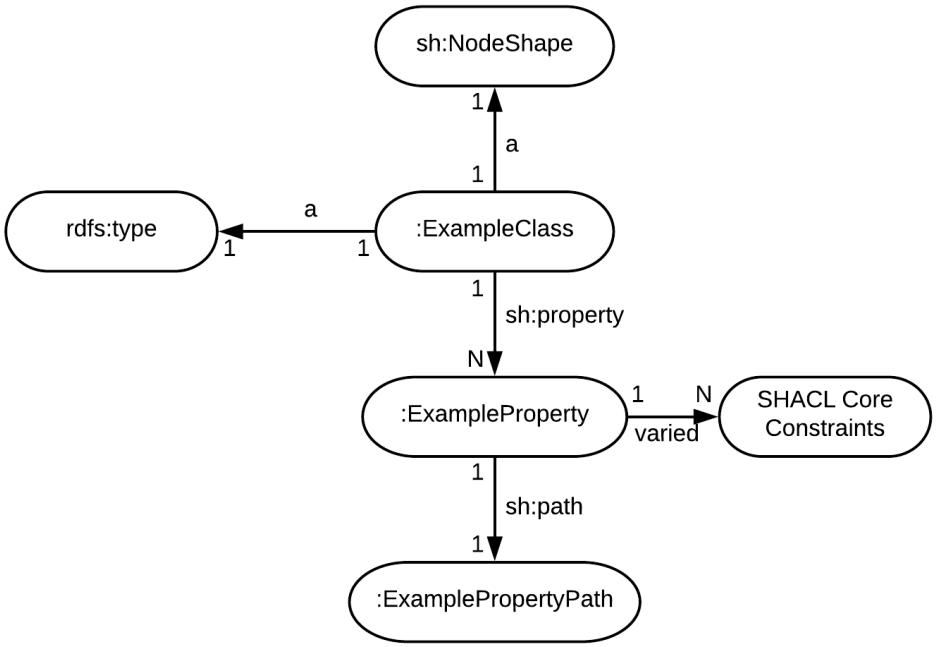

Figure 3. Node diagram of alternate structure for implicit target class declaration

1.3 RDF Handling - Importing other shapes

Additionally, SHACL shapes and property shapes may import other shapes using the ‘sh:node’ predicate. This application handles that feature by inserting the referenced shape into the root shape. Figure 4 demonstrates the structure of a SHACL shape importing another SHACL shape.

After being interpreted by the application, its structure is identical to Figure 5.

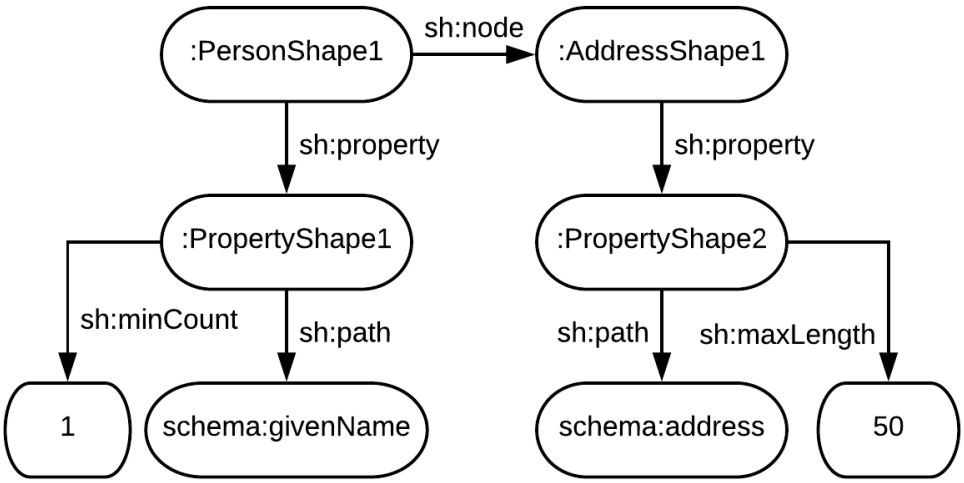

Figure 4. Node diagram with an example of importing other SHACL shapes

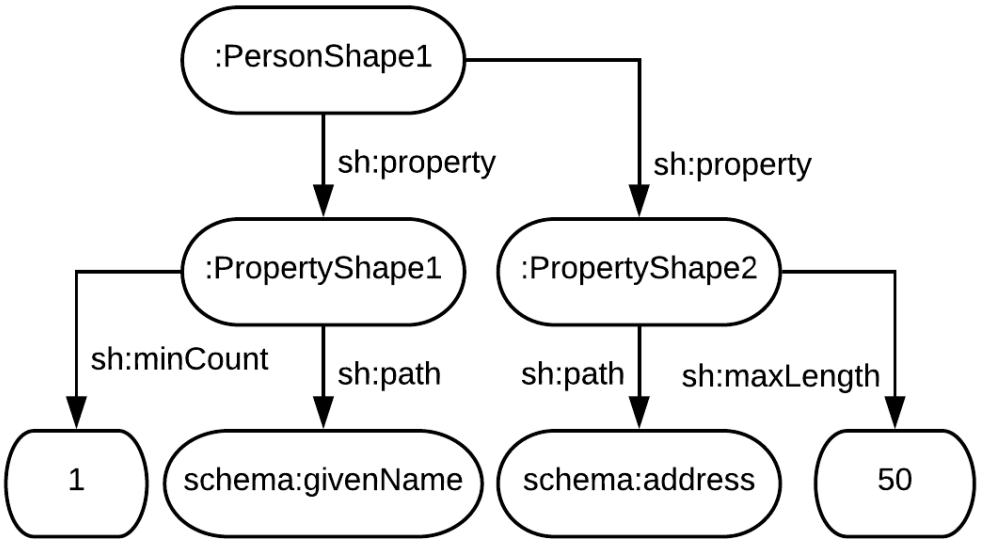

Figure 5. Node diagram without importing other shapes, which is functionally identical to Figure 7

1.4 Rendering

The ‘rendering’ package has several Jinja2 templates in use, each representing a component that allows the webform to be built in a modular way, as shown in Figure 6. ‘base.html’ sets out the structure of the document, and inserts a ‘property.html’ template for every property that must be present in the webform. The ‘property.html’ template contains the property label, description, add/remove buttons, and one or more entries per property, using ‘entry.html’. The

‘entry.html’ template either includes the ‘composite_property.html’ template or the ‘input_field.html’ template, depending on whether the property contains nested properties. The ‘composite_property.html’ template includes a ‘property.html’ template for every nested property, creating a loop to render every property that may be nested within another property. The ‘input_field.html’ template simply includes an input field with all the the constraints that apply to it.

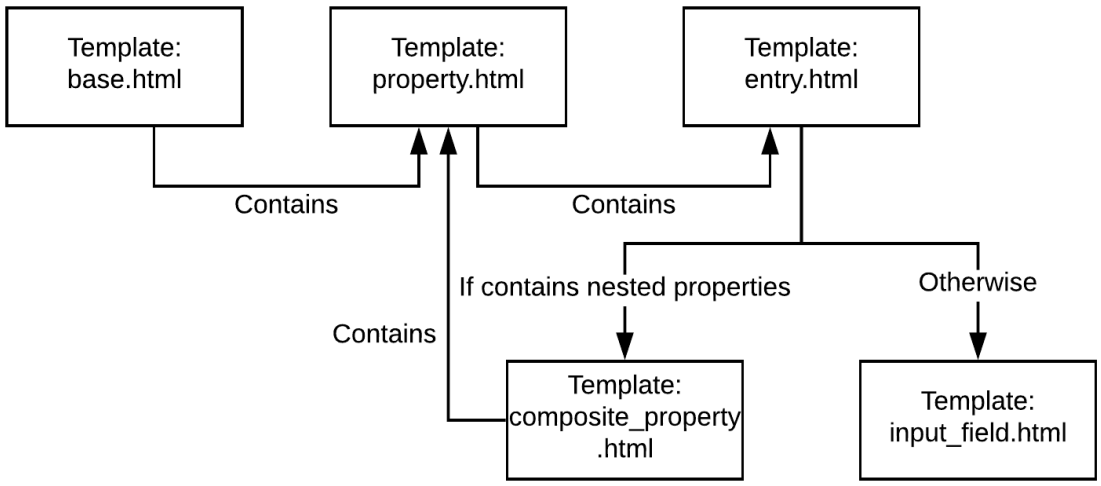

Figure 6. A diagram showing the hierarchy of templates used to generate the webform

While in early iterations this application generated a JavaScript script to complement the webform, it was later altered so that the script did not need to be customised for each webform. This allowed for all JavaScript to be removed from templates and relocated as a standalone script, which improved its maintainability.

2 Flask Web Server and Data Storage

The Flask web server was developed to demonstrate how this webform may be used. It displays the webform and accepts any data submitted, storing it in RDF Turtle format. While the application is simple and part of a minimalist framework, it implements MVC architecture, which keeps the application logic separate from the user interface and the data storage as shown in Figure 7. The Python file ‘routes.py’ defines the routes in the application, as well as all of the application logic. The ‘view’ directory contains everything related to the user interface, such as the CSS files, webform scripts and Jinja2 templates. The ‘results’ directory contains the RDF files stored by the application.

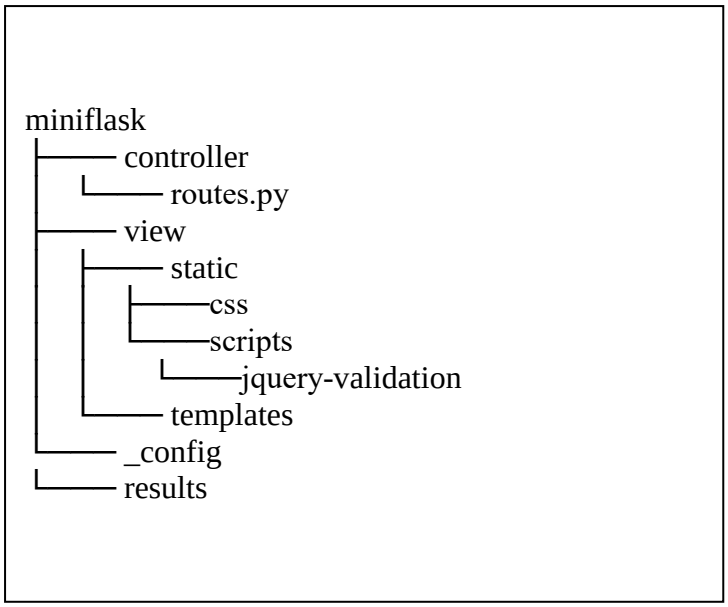

Figure 7. The file hierarchy of the Flask web server