Journal of
open research software

SOFTWARE METAPAPER

]u[ubiquity press

auto{API} – A Web-Based Tool for Specification of an API Endpoint to Return JSON Data From an XML Source

ADRIAN MOORE

ABSTRACT

auto{API} is a web-based tool for the specification and management of API endpoints to online XML data sources. The APIs generated return the data in JSON format. The tool allows users to specify the field names to be used in the JSON data and to control which elements of the original data set are returned. Options are provided for the returned data to skip null values and to flatten the structure that is retrieved. **auto{API}** is written in Python/Flask and is available under the MIT licence. The software is available from GitHub (*https://github.com/aamoore/autoAPI*) and can be seen deployed at *https://autoapi-app.herokuapp.com/?http://apis.opendatani.gov.uk/translink/3042AA.xml*.

CORRESPONDING AUTHOR:
Adrian Moore
School of Computing, Ulster University, UK
aa.moore@ulster.ac.uk

KEYWORDS:
XML; JSON; API

TO CITE THIS ARTICLE:
Moore A 2021 auto{API} – A Web-Based Tool for Specification of an API Endpoint to Return JSON Data From an XML Source. *Journal of Open Research Software*, 9: 24. DOI: *https://doi.org/10.5334/jors.335*

(1) OVERVIEW

INTRODUCTION

XML (Extensible Markup Language) [1] and JSON (JavaScript Object Notation) [2] are competing notations for the representation and transfer of structured data between (often online) applications. Of these, XML is by far the longer established having been first formally specified by the World Wide Web Consortium in 1998, while JSON was eventually standardised as ECMA-404 in 2013, having been first specified in 2002. Although there are significant differences between the notations (for example XML is typeless while JSON is typed) they are essentially used for the same purpose, hence developers of software that is required to transfer data to or from collaborative services are often required to convert from one format to the other. Given the historical precedence of XML, the large number of legacy systems that use it, and the close integration of JSON with modern development languages such as Python; this conversion is most often from XML to JSON. There is no single accepted solution for conversion, but rather, the translation is performed in an application-dependent way. For example, depending on the requirements of the receiving software, the XML namespace definitions may be incorporated or ignored, while the treatment of XML node attributes may vary. This situation is represented in ***Figure 1***, where an application retrieves data from an XML data source via an API and converts it to JSON for further processing and/or presentation.

An example of a potential Data Consumer Application could be from the field of Data Mining – the research activity that refers to the analysis and processing of large data sets to identify patterns and rules that can be exploited to improve the performance, usefulness and functionality of a system, or to reduce costs. Many very large data sets are currently held in repositories and the human readability and tightly structured format of XML makes it a popular choice as a data representation notation. There have been many studies on the application of clustering and classification techniques to XML data, but such activities normally require that the XML is pre-processed into a more traditional form using a DOM or SAX builder. **auto{API}** addresses this by providing a means to return the entire XML dataset as a JSON object in a single operation. The JSON data can then be easily loaded to a database from where the data mining activity can take place. Where the database is arranged as a document store architecture (such as MongoDB), the entire dataset can be uploaded in a single operation.

The aim of **auto{API}** is to introduce a broker component implemented as a RESTful API that handles the retrieval of the XML and the conversion of the data to JSON format according to preferences that are established in a setup phase. The **auto{API}** preferences enable the consumer application to specify which XML elements are to be retrieved, the field names that they should be assigned and the structure of the JSON element to be returned. As well as greatly simplifying

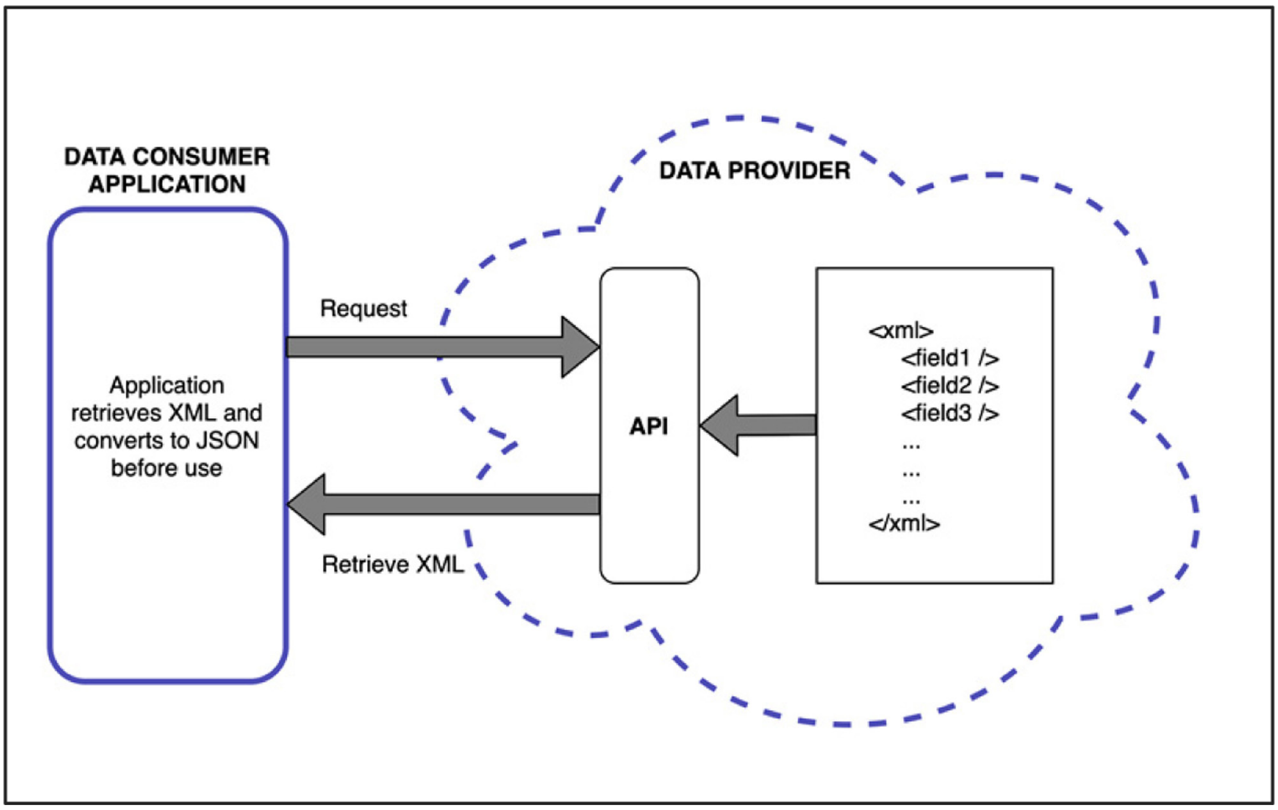


Figure 1 Direct access to external data source.

the processing task of the end-user application, this provides a degree of protection against future changes in the structure and content of the data source. If the data provider changes the format of the XML source, the **auto{API}** API preferences can be updated to reflect this – avoiding the need for the consumer application to be re-written. ***Figure 2*** illustrates the use of **auto{API}** and its role in data retrieval.

IMPLEMENTATION AND ARCHITECTURE

auto{API} was developed between April and June 2020 using Python/Flask, SQLite, HTML, CSS and JavaScript/jQuery. It builds upon the existing **`xmltodict`** Python package [3] which parses an XML source (wither as a string or external file) and returns an equivalent JSON object. **auto{API}** adds value to **`xmltodict`** by providing a tool in which users can select specific fields of the XML source to be harvested while ignoring the others, and also by providing an option to collapse the resulting JSON structure to its simplest form. The architecture of **auto{API}** is illustrated in ***Figure 3*** which presents the relationship between 3 code modules and integrated database.

The first module ("*XML Field Selection*") is invoked by a **GET** request to the application root endpoint **`/?url`**, presenting the URL of the XML data source as the query

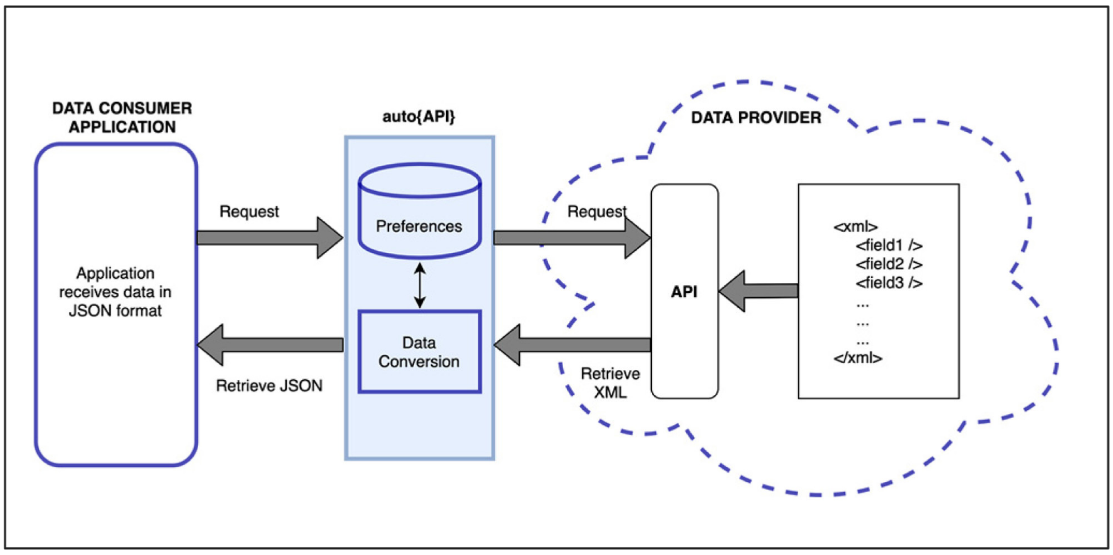


Figure 2 Using **auto{API}**.

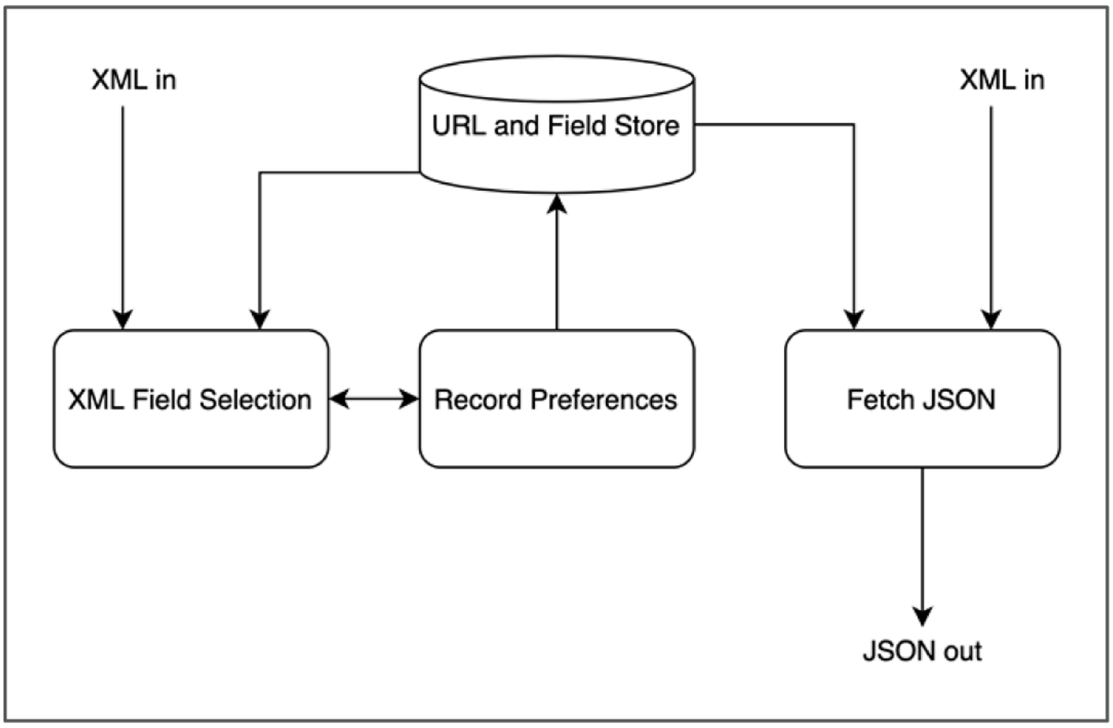


Figure 3 **auto{API}** structure.

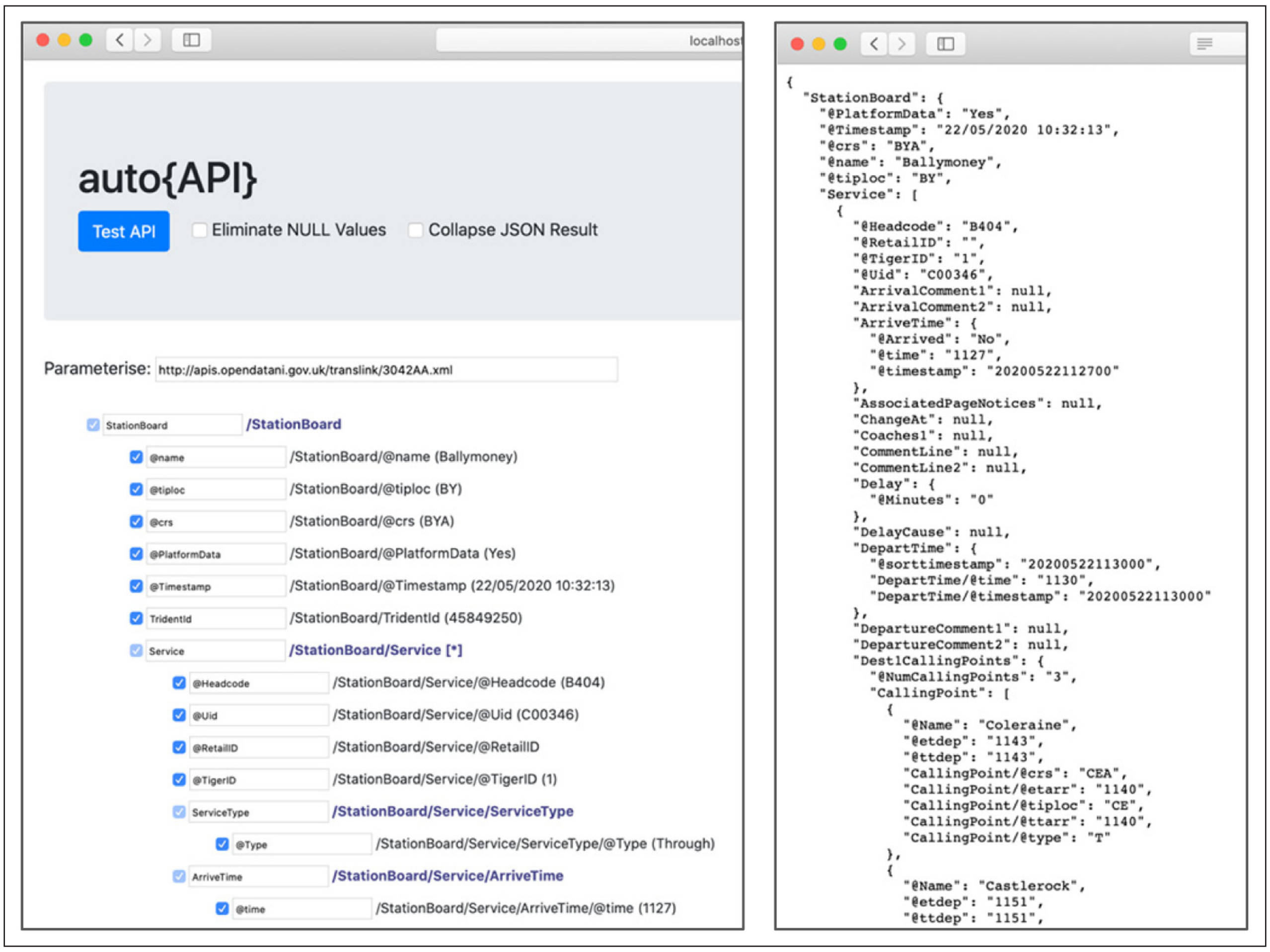
string. The module retrieves the XML data from the specified address, converts it to JSON by the **`xmltodict.parse()`** method and displays the equivalent JSON tree structure within an HTML form in the browser. The JSON tree includes a checkbox for each element, which enables the user to select the fields that they wish to be included in the resulting data structure. When the user submits the form, a **`POST`** request to the application root endpoint **`/?url`** invokes the "*Record Preferences*" module which accepts the user's field selections and updates the database accordingly. Finally, a POST request to the endpoint **`/api?url`** invokes the "*Fetch JSON*" module which retrieves the XML source and generates the JSON output according to the user's field selections.

The basic operation of the application is illustrated by ***Figure 4*** which presents the clickable node tree and the equivalent JSON output for the real-time passenger information for Ballymoney Train Station made available in XML form by Northern Ireland Railways as part of the OpenDataNI initiative [4] at *http://apis.opendatani.gov.uk/translink/3042AA.xml*.

The power and flexibility of **auto{API}** is enabled in part by the generation of the clickable node tree which allows the user to specify the fields to be retrieved. This is implemented by structuring the tree as an HTML form containing a set of nested groups of checkboxes. Each leaf node in the structure corresponds to a raw data value, while the non-leaf nodes correspond to collections in the data source. Clicking a checkbox for a non-leaf node sets or clears the check boxes for all children of that node, hence the entire dataset can be selected or deselected by clicking the root "*StationBoard*" element in the example above, while clicking the "*Service*" node selects or deselects that node plus all nodes which are descendants of "*Service*".

Each node is assigned a unique name based on the path to that node in the original XML structure. The name is assigned by selecting the minimum selection from the end of the path that preserves uniqueness, hence the node with path **`/StationBoard/Service/ArriveTime/@time`** is assigned the name **`@time`**, while the subsequent node with path **`/StationBoard/Service/DepartTime/@time`** is assigned the name **`DepartTime/@time`**. In addition, the user can specify their own name for any node by modifying the value in the corresponding text box. Error trapping ensures that all names are unique before the user is permitted to submit their selections.

Additional features of the tree include display of a sample value for leaf nodes (e.g. "*Ballymoney*" for **`/StationBoard/@name`** in the example above and the use of ***** and **+** qualifiers following the path to denote that the element is a collection or optional respectively



Parameterise: http://apis.opendatani.gov.uk/translink/3042AA.xml

```
✓ StationBoard   /StationBoard
    ✓ @name            /StationBoard/@name (Ballymoney)
    ✓ @tiploc          /StationBoard/@tiploc (BY)
    ✓ @crs             /StationBoard/@crs (BYA)
    ✓ @PlatformData    /StationBoard/@PlatformData (Yes)
    ✓ @Timestamp       /StationBoard/@Timestamp (22/05/2020 10:32:13)
    ✓ TridentId        /StationBoard/TridentId (45849250)
    ✓ Service          /StationBoard/Service [*]
        ✓ @Headcode        /StationBoard/Service/@Headcode (B404)
        ✓ @Uid             /StationBoard/Service/@Uid (C00346)
        ✓ @RetailID        /StationBoard/Service/@RetailID
        ✓ @TigerID         /StationBoard/Service/@TigerID (1)
        ✓ ServiceType      /StationBoard/Service/ServiceType
            ✓ @Type            /StationBoard/Service/ServiceType/@Type (Through)
        ✓ ArriveTime       /StationBoard/Service/ArriveTime
            ✓ @time            /StationBoard/Service/ArriveTime/@time (1127)
```

```
{
  "StationBoard": {
    "@PlatformData": "Yes",
    "@Timestamp": "22/05/2020 10:32:13",
    "@crs": "BYA",
    "@name": "Ballymoney",
    "@tiploc": "BY",
    "Service": [
      {
        "@Headcode": "B404",
        "@RetailID": "",
        "@TigerID": "1",
        "@Uid": "C00346",
        "ArrivalComment1": null,
        "ArrivalComment2": null,
        "ArriveTime": {
          "@Arrived": "No",
          "@time": "1127",
          "@timestamp": "20200522112700"
        },
        "AssociatedPageNotices": null,
        "ChangeAt": null,
        "Coaches1": null,
        "CommentLine": null,
        "CommentLine2": null,
        "Delay": {
          "@Minutes": "0"
        },
        "DelayCause": null,
        "DepartTime": {
          "@sorttimestamp": "20200522113000",
          "DepartTime/@time": "1130",
          "DepartTime/@timestamp": "20200522113000"
        },
        "DepartureComment1": null,
        "DepartureComment2": null,
        "Dest1CallingPoints": {
          "@NumCallingPoints": "3",
          "CallingPoint": [
            {
              "@Name": "Coleraine",
              "@etdep": "1143",
              "@ttdep": "1143",
              "CallingPoint/@crs": "CEA",
              "CallingPoint/@etarr": "1140",
              "CallingPoint/@tiploc": "CE",
              "CallingPoint/@ttarr": "1140",
              "CallingPoint/@type": "T"
            },
            {
              "@Name": "Castlerock",
              "@etdep": "1151",
              "@ttdep": "1151",
```

Figure 4 Clickable node tree and resulting JSON output.

– demonstrated by the **[*]** flag for the `/StationBoard/Service` node shown above.

The node tree is generated by the parsing algorithm illustrated in ***Figure 5*** below.

The algorithm recursively processes the children of each node, beginning with the root of the tree structure. As each node is processed, if it is being encountered for the first time, then a new record is created in the database with the path to the element and its unique name. If the node is one that has been found previously (for example the second `/StationBoard/Service` element in the example presented above), then the number of times that this node has been found is updated. If the element is a collection (i.e. a non-leaf node), then the process is repeated for each member of the collection; otherwise it is a leaf node and the database entry can be updated with the sample data value.

When all nodes have been processed, a unique sequence code is generated for each element such that all nodes within a subtree share a common prefix, as

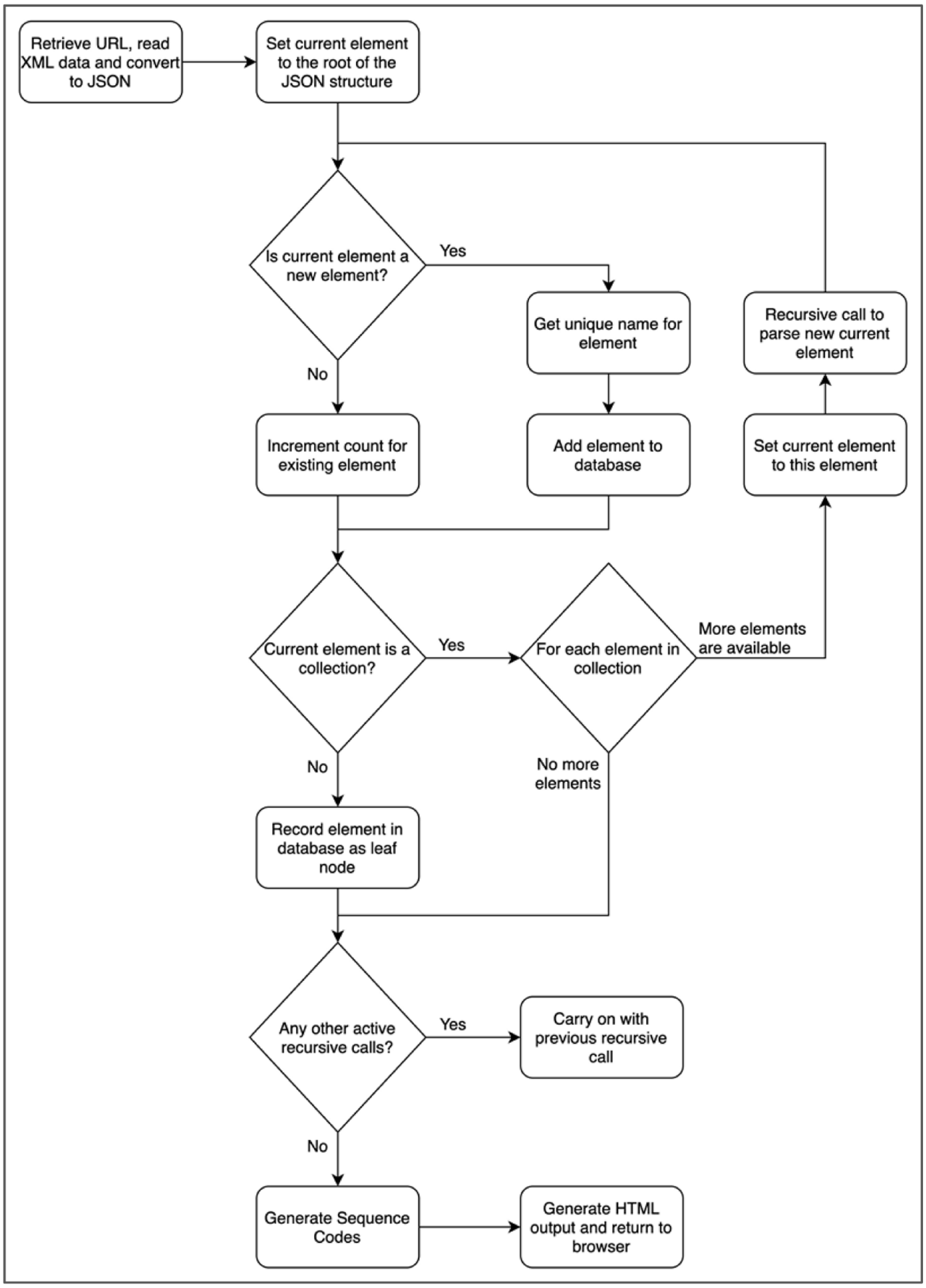

Retrieve URL, read XML data and convert to JSON
Set current element to the root of the JSON structure
Is current element a new element?
Yes
No
Get unique name for element
Recursive call to parse new current element
Increment count for existing element
Add element to database
Set current element to this element
Current element is a collection?
Yes
For each element in collection
More elements are available
No
No more elements
Record element in database as leaf node
Any other active recursive calls?
Yes
Carry on with previous recursive call
No
Generate Sequence Codes
Generate HTML output and return to browser

Figure 5 Parsing the source XML.

illustrated in *Figure 6*, which presents a sample 3-level XML structure and the sequence codes that would be generated for each node. The root node is allocated the sequence number 1., while direct children of the root have sequence numbers 1.0001, 1.0002, 1.0003 and so on. Likewise, children of node 1.0002 are labelled 1.0002.0001, 1.0002.0002, etc. The structure of the sequence nodes and the uniqueness of each enables the implementation of the jQuery function that allows selection/deselection of entire subtrees with a single click.

The effect of the clickable node tree is illustrated by *Figure 7* which demonstrates the selection of a subset of nodes and the JSON that is generated as a result. Here, we use the checkboxes to specify that only a small selection of node values is required and that the nodes `/StationBoard/Service/ArriveTime/@time` and `/StationBoard/Service/DepartTime/@time` should be re-named as *ArrivingAt* and *DepartingAt*, respectively.

It can be seen from the JSON returned in *Figure 7* that the entire JSON structure is generated, regardless of whether a particular collection or sub-collection contains one of the selected nodes. For example, the `StationBoard/Service/Delay` collection is returned as an empty object, as it's only child `@Minutes` was not one of the selected values. In addition, since all node names are unique there is no danger of conflicts between identically named nodes in different sub-trees, and so it may be appropriate (or more useful to the user) to present the resulting JSON data as a flatter structure.

This is enabled by the "*Collapse JSON Result*" flag that can be submitted with the Test API button, illustrated by *Figure 8*. Using this option generates a much more compact data set, with empty collections removed and all leaf nodes promoted to the highest level possible, while maintaining the integrity of the data. In this example, the data is presented as a list of `Service` nodes, with each node value represented as a top-level element of the `Service`.

A similar flag "*Eliminate NULL Values*" can be used to specify that only selected leaf nodes that contain data should be returned.

The final element of **auto{API}** functionality recognises that data providers sometimes employ URLs with embedded parameters that identify the specific data to serve. In the case of the example presented here, part of the URL is a code denoting the station for which passenger information is delivered. Hence the URL *http://apis.opendatani.gov.uk/translink/3042AA.xml* returns real-time information for Ballymoney Train Station, while the URL *http://apis.opendatani.gov.uk/translink/3045CE.xml* returns similar information for Portadown Train Station. Using **auto{API}**, the user can identify the variable element

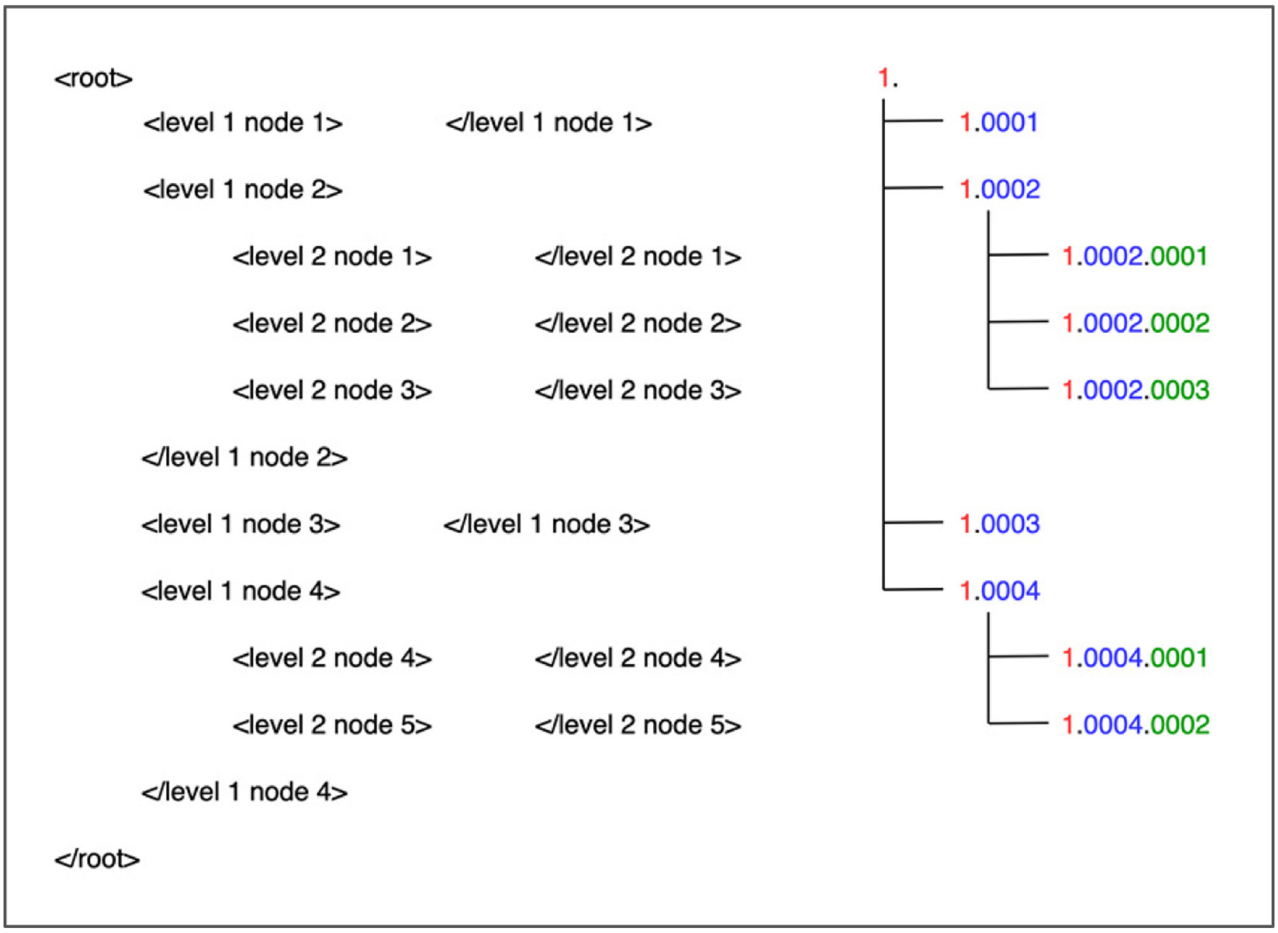
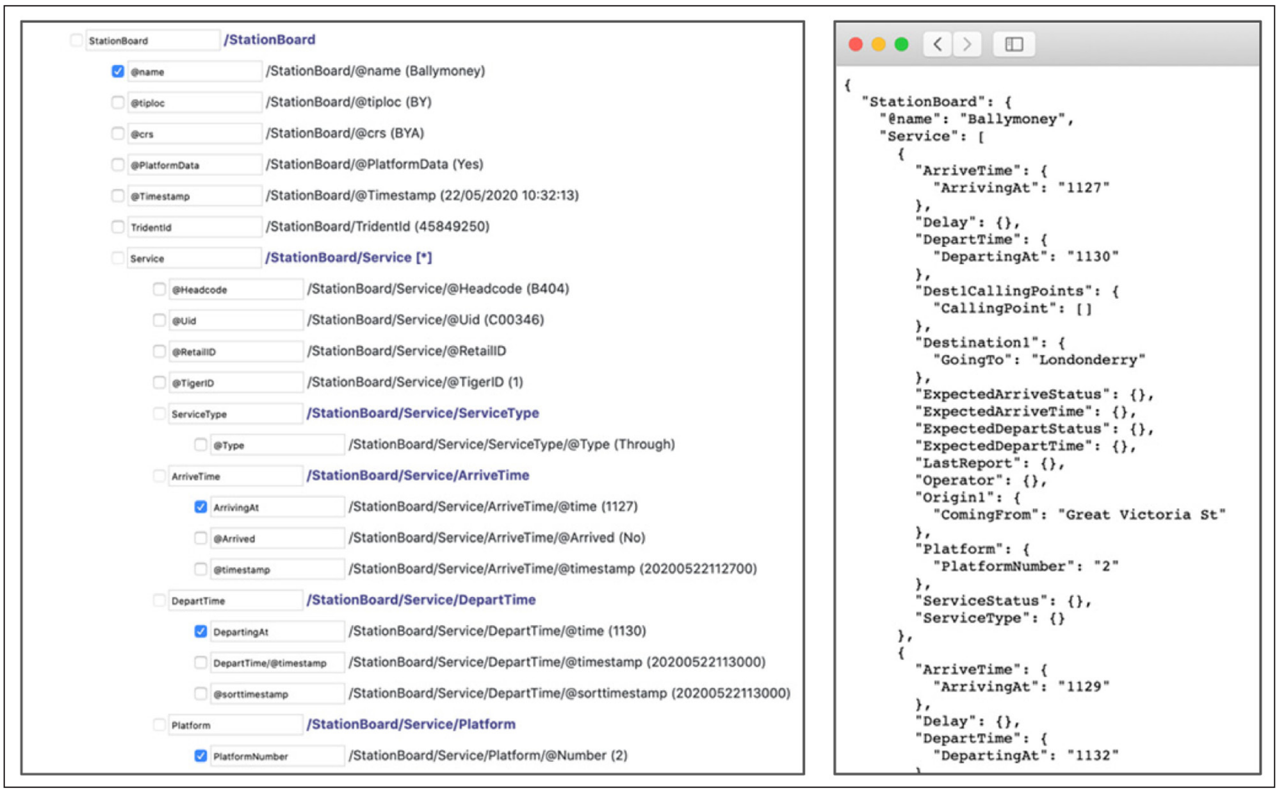


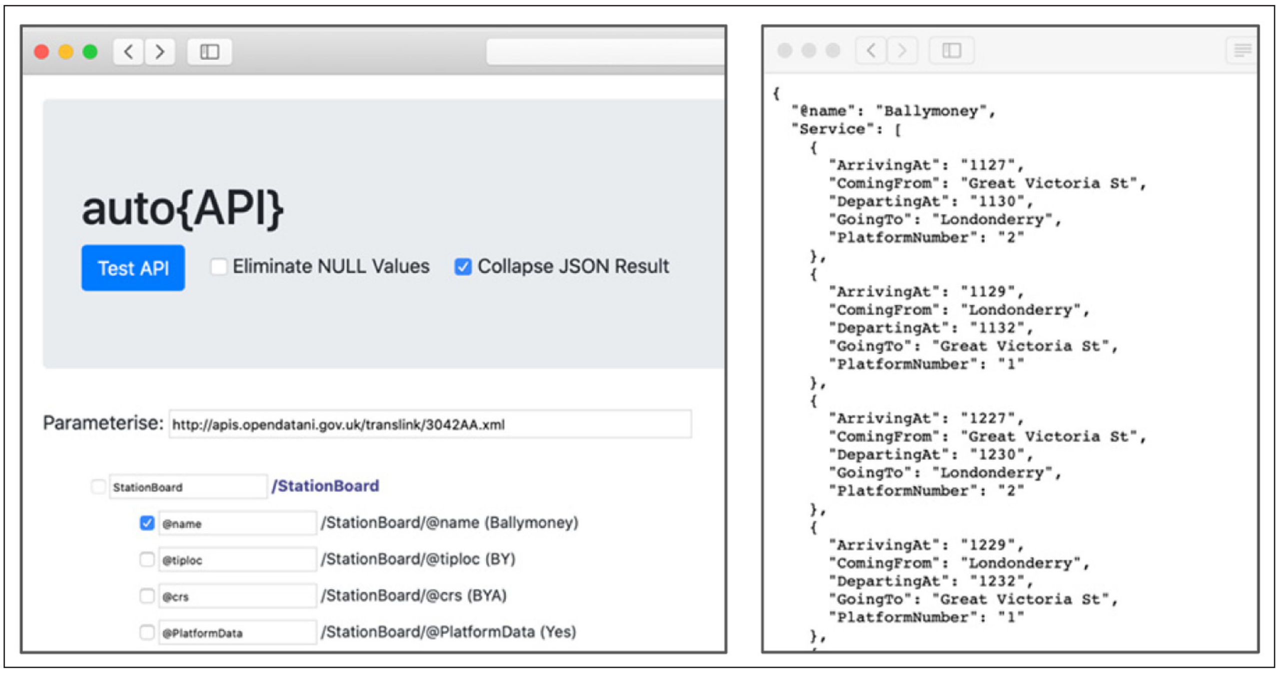
Figure 6 Assigning unique sequence numbers to elements.



```
☐ StationBoard            /StationBoard
  ☑ @name                 /StationBoard/@name (Ballymoney)
  ☐ @tiploc               /StationBoard/@tiploc (BY)
  ☐ @crs                  /StationBoard/@crs (BYA)
  ☐ @PlatformData         /StationBoard/@PlatformData (Yes)
  ☐ @Timestamp            /StationBoard/@Timestamp (22/05/2020 10:32:13)
  ☐ TridentId             /StationBoard/TridentId (45849250)
  ☐ Service               /StationBoard/Service [*]
    ☐ @Headcode           /StationBoard/Service/@Headcode (B404)
    ☐ @Uid                /StationBoard/Service/@Uid (C00346)
    ☐ @RetailID           /StationBoard/Service/@RetailID
    ☐ @TigerID            /StationBoard/Service/@TigerID (1)
    ☐ ServiceType         /StationBoard/Service/ServiceType
      ☐ @Type             /StationBoard/Service/ServiceType/@Type (Through)
    ☐ ArriveTime          /StationBoard/Service/ArriveTime
      ☑ ArrivingAt        /StationBoard/Service/ArriveTime/@time (1127)
      ☐ @Arrived          /StationBoard/Service/ArriveTime/@Arrived (No)
      ☐ @timestamp        /StationBoard/Service/ArriveTime/@timestamp (20200522112700)
    ☐ DepartTime          /StationBoard/Service/DepartTime
      ☑ DepartingAt       /StationBoard/Service/DepartTime/@time (1130)
      ☐ DepartTime/@timestamp  /StationBoard/Service/DepartTime/@timestamp (20200522113000)
      ☐ @sorttimestamp    /StationBoard/Service/DepartTime/@sorttimestamp (20200522113000)
    ☐ Platform            /StationBoard/Service/Platform
      ☑ PlatformNumber    /StationBoard/Service/Platform/@Number (2)
```

```
{
  "StationBoard": {
    "@name": "Ballymoney",
    "Service": [
      {
        "ArriveTime": {
          "ArrivingAt": "1127"
        },
        "Delay": {},
        "DepartTime": {
          "DepartingAt": "1130"
        },
        "Dest1CallingPoints": {
          "CallingPoint": []
        },
        "Destination1": {
          "GoingTo": "Londonderry"
        },
        "ExpectedArriveStatus": {},
        "ExpectedArriveTime": {},
        "ExpectedDepartStatus": {},
        "ExpectedDepartTime": {},
        "LastReport": {},
        "Operator": {},
        "Origin1": {
          "ComingFrom": "Great Victoria St"
        },
        "Platform": {
          "PlatformNumber": "2"
        },
        "ServiceStatus": {},
        "ServiceType": {}
      },
      {
        "ArriveTime": {
          "ArrivingAt": "1129"
        },
        "Delay": {},
        "DepartTime": {
          "DepartingAt": "1132"
```

Figure 7 Selected fields and resulting JSON output.



☐ Eliminate NULL Values ☑ Collapse JSON Result

Parameterise: http://apis.opendatani.gov.uk/translink/3042AA.xml

```
☐ StationBoard          /StationBoard
  ☑ @name               /StationBoard/@name (Ballymoney)
  ☐ @tiploc             /StationBoard/@tiploc (BY)
  ☐ @crs                /StationBoard/@crs (BYA)
  ☐ @PlatformData       /StationBoard/@PlatformData (Yes)
```

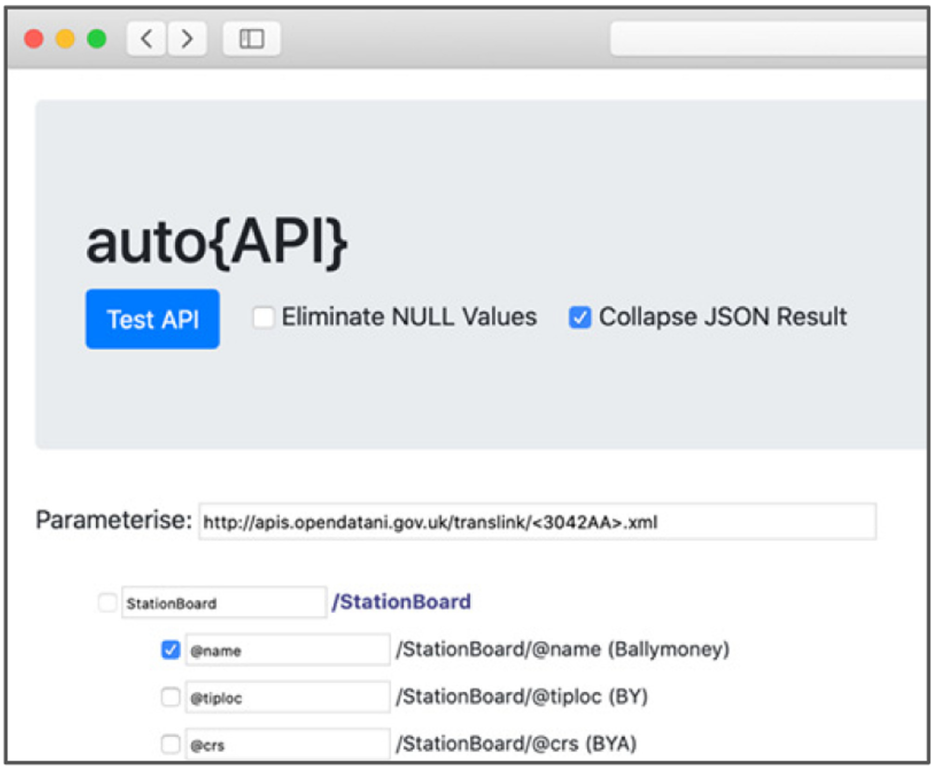
```
{
  "@name": "Ballymoney",
  "Service": [
    {
      "ArrivingAt": "1127",
      "ComingFrom": "Great Victoria St",
      "DepartingAt": "1130",
      "GoingTo": "Londonderry",
      "PlatformNumber": "2"
    },
    {
      "ArrivingAt": "1129",
      "ComingFrom": "Londonderry",
      "DepartingAt": "1132",
      "GoingTo": "Great Victoria St",
      "PlatformNumber": "1"
    },
    {
      "ArrivingAt": "1227",
      "ComingFrom": "Great Victoria St",
      "DepartingAt": "1230",
      "GoingTo": "Londonderry",
      "PlatformNumber": "2"
    },
    {
      "ArrivingAt": "1229",
      "ComingFrom": "Londonderry",
      "DepartingAt": "1232",
      "GoingTo": "Great Victoria St",
      "PlatformNumber": "1"
    },
```

Figure 8 Collapsing the JSON result.

of the URL by enclosing it within `<` and `>` characters in the *Parameterise* text box and then provide the actual station code to be used by **POST**ing it as a field with the name `parameter`. A database table stores details of any variable URLs specified so that the parameterisation can be automatically restored the next time that data set is requested. ***Figure 9*** illustrates the specification of the variable element of the URL in the *Parameterise* text box, while ***Figure 10*** shows how this causes the *Parameter value* text box to be revealed so that the user can provide the value to be substituted into the URL, as well as the JSON result that is returned.

QUALITY CONTROL

The software has been extensively tested for functionality and usability. Each of the endpoints has been tested by

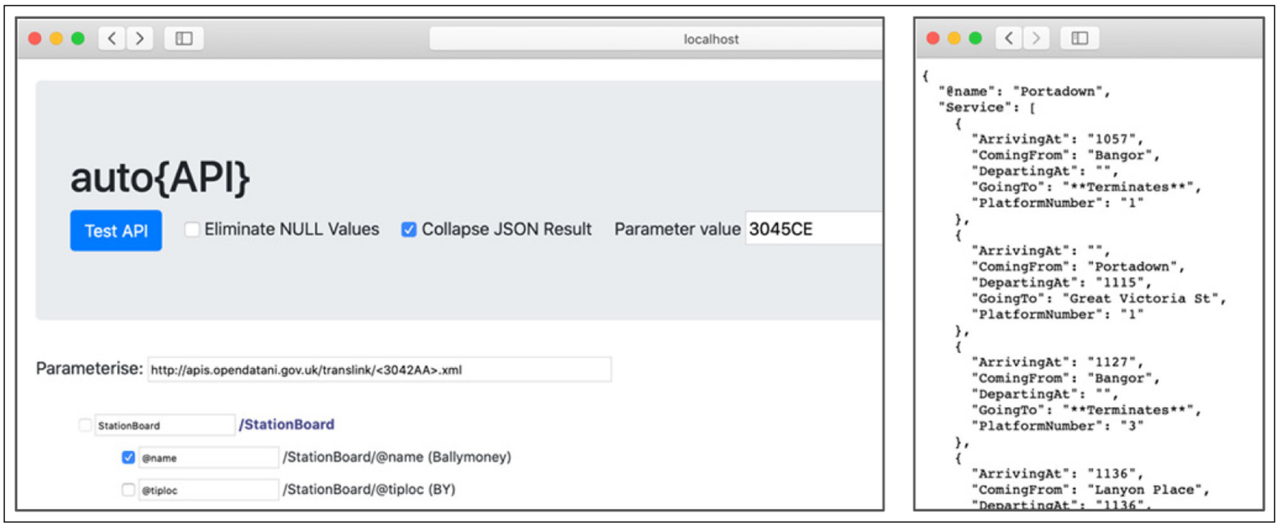

auto{API}

Test API ☐ Eliminate NULL Values ☑ Collapse JSON Result

Parameterise: http://apis.opendatani.gov.uk/translink/<3042AA>.xml

☐ StationBoard /StationBoard
☑ @name /StationBoard/@name (Ballymoney)
☐ @tiploc /StationBoard/@tiploc (BY)
☐ @crs /StationBoard/@crs (BYA)

Figure 9 Parameterise the URL.


localhost

auto{API}

Test API ☐ Eliminate NULL Values ☑ Collapse JSON Result Parameter value 3045CE

Parameterise: http://apis.opendatani.gov.uk/translink/<3042AA>.xml

☐ StationBoard /StationBoard
☑ @name /StationBoard/@name (Ballymoney)
☐ @tiploc /StationBoard/@tiploc (BY)

```
{
  "@name": "Portadown",
  "Service": [
    {
      "ArrivingAt": "1057",
      "ComingFrom": "Bangor",
      "DepartingAt": "",
      "GoingTo": "**Terminates**",
      "PlatformNumber": "1"
    },
    {
      "ArrivingAt": "",
      "ComingFrom": "Portadown",
      "DepartingAt": "1115",
      "GoingTo": "Great Victoria St",
      "PlatformNumber": "1"
    },
    {
      "ArrivingAt": "1127",
      "ComingFrom": "Bangor",
      "DepartingAt": "",
      "GoingTo": "**Terminates**",
      "PlatformNumber": "3"
    },
    {
      "ArrivingAt": "1136",
      "ComingFrom": "Lanyon Place",
      "DepartingAt": "1136",
```

Figure 10 Supply a parameter value to request a specific data set.

providing a range of valid and invalid input and examining the results obtained. In addition, the test suite capabilities of the Postman API development platform have been used to ensure that run-time errors are trapped and appropriate JSON error messages returned. The software has been tested on Windows and MacOS environments, with a wide range of modern web browsers.

(2) AVAILABILITY

OPERATING SYSTEM

A platform-independent application, running in any modern web browser (Safari, Chrome, Edge, Firefox, etc.)

PROGRAMMING LANGUAGE

Python v3.x

ADDITIONAL SYSTEM REQUIREMENTS

None.

DEPENDENCIES

Uses the following Python packages

- Flask (*https://flask.palletsprojects.com/en/1.1.x/*)
- xmltodict (*https://pypi.org/project/xmltodict/*)

INSTALLATION

1. Install the packages `xmltodict` and `flask` to a Python 3 environment. This is most easily done using the pip package manager with the commands `pip install xmltodict` and `pip install flask`.
2. Copy the source file **autoAPI.py** and the database file **autoAPI.db** to the selected working directory.
3. Create a sub-folder "templates" and copy the file **index.html** to this location
4. In the working directory, invoke the application by the command `python autoAPI.py`
5. With the python application running, use a web browser to visit the location `http://localhost:5000/?url` where `url` is the address of the XML source that you want to use. For example, the URL *http://localhost:5000/?http://apis.opendatani.gov.uk/translink/3042AA.xml* will launch the application with the XML passenger information data described in this paper.

SOFTWARE LOCATION

Archive (e.g. institutional repository, general repository) (required – please see instructions on journal website for depositing archive copy of software in a suitable repository)

Name: Zenodo
Persistent identifier: DOI *10.5281/zenodo.3889859*
Licence: MIT
Publisher: Moore, Adrian
Version published: v1.1
Date published: 25/05/2020

Code repository

Name: GitHub
Identifier: *https://github.com/aamoore/autoAPI*
Licence: MIT
Date published: 19/05/2020

LANGUAGE

English

(3) REUSE POTENTIAL

The software can be used in any situation where an XML data source is required to be converted to JSON for use in an online application.

As an example, the author has incorporated **auto{API}** into a voice interface research project as the data retrieval module for an Amazon Alexa Skill that implements a voice interface to the Northern Ireland Railways live passenger information discussed in this paper. In this application, a user can query Alexa with a question such as "when is the next train from Coleraine to Belfast?" so that the Skill logic calls the appropriate endpoint via **auto{API}**, parses the JSON result and returns the output to be voiced by the Alexa-enabled device. Additional logic adds context to the conversation by maintaining state information to facilitate follow-up queries such as "is there a later train?" or "when is the first train this afternoon?". This research is ongoing, but **auto{API}** has been a very useful tool in making the native XML data available in a format much better suited to query and analysis.

Apart from the Data Mining scenario outlined earlier, there are many other research areas where **auto{API}** has the potential to make an enabling contribution. These include (but are by no means limited to):

- Natural Language Processing (NLP): One active area of NLP research is in the provision of a natural language interface to an XML dataset. This is most often accomplished by converting the natural language query to an XQuery statement that can be applied to an XML database [5]. However, the structure and flexibility of XQuery has been subject to criticism [6] and conversion of the XML data to JSON by a tool such as **auto{API}** makes it more accessible to modern languages such as Python that provide native support for JSON structures.
- Geographic Information Systems (GIS): The Geography Markup Language (GML) is an XML-based notation used to represent geographic information. It is commonly used as a data transfer syntax between applications that model and process such data, but the lack of complex query support and relatively high storage requirement of GML has been identified in the literature as a significant drawback [7]. Using **auto{API}** to retrieve the GML data as JSON would enable easy integration with modern database systems that provide greatly enhanced query and analysis facilities.
- Medical Data Analysis: Current medical practice generates an enormous volume of patient data which can be analysed to identify trends and other important features. XML-based tools such as SOMA have been used to parse and visualise the results to facilitate expert interpretation [8]. Integration of a tool such as **auto{API}** would allow the results to be easily made available in JSON so that more complex computation-based analysis can also be performed.

There is no explicit official support for **auto{API}**, but those with queries or expressions of interest in collaboration should contact the author by email at *aa.moore@ulster.ac.uk*.

COMPETING INTERESTS

The author has no competing interests to declare.

AUTHOR AFFILIATION

Adrian Moore *orcid.org/0000-0001-6545-3228*
School of Computing, Ulster University, UK

REFERENCES

1. **Quin L.** *Extensible Markup Language (XML)* [online]; 2016. Available at: *https://www.w3.org/XML/* [Accessed 10 June 2020].
2. **ECMA International.** *The JSON Data interchange Syntax,* 2nd Ed. [online]; 2017. Available at: *http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf* [Accessed 10 June 2020].
3. **Blech M.** xmltodict. *Python module that makes working with XML feel like you are working with JSON*. [online]; 2014. Available at: *https://github.com/martinblech/xmltodict* [Accessed 10 June 2020].
4. **OpenDataNI.** *Open Data Northern Ireland*. [online]; 2015. Available at *https://www.opendatani.gov.uk* [Accessed 10 June 2020].
5. **Jiffy J, Panicker J, Meera M.** An efficient natural language interface to XML database. *2016 International Conference on Information Science (ICIS)*, Kochi, 2016; 207–212. DOI: *https://doi.org/10.1109/INFOSCI.2016.7845328*
6. **David M.** *Ten Problems with XQuery and the SQL/XML Standard*. [online]; 2010. Available at *https://www.databasejournal.com/sqletc/article.php/3865201/Ten-Problems-with-XQuery-and-the-SQLXML-Standard.htm* [Accessed 07 July 2020].
7. **Lu C, Dos Santos RF, Sripada LN, Kou Y.** Advances in GML for Geospatial Applications. *Geoinformatica*. 2007; 11: 131–157. DOI: *https://doi.org/10.1007/s10707-006-0013-9*
8. **Somaraki V, Xu Z.** Knowledge representation of large medical data using XML. *22nd International Conference on Automation and Computing (ICAC)*, Colchester. 2016; 423–428. DOI: *https://doi.org/10.1109/IConAC.2016.7604956*

TO CITE THIS ARTICLE:
Moore A 2021 auto{API} – A Web-Based Tool for Specification of an API Endpoint to Return JSON Data From an XML Source. *Journal of Open Research Software,* 9: 24. DOI: *https://doi.org/10.5334/jors.335*

Submitted: 15 June 2020 Accepted: 05 August 2021 Published: 20 August 2021