Last edited by **Ondřej Vašíček** 1 day ago

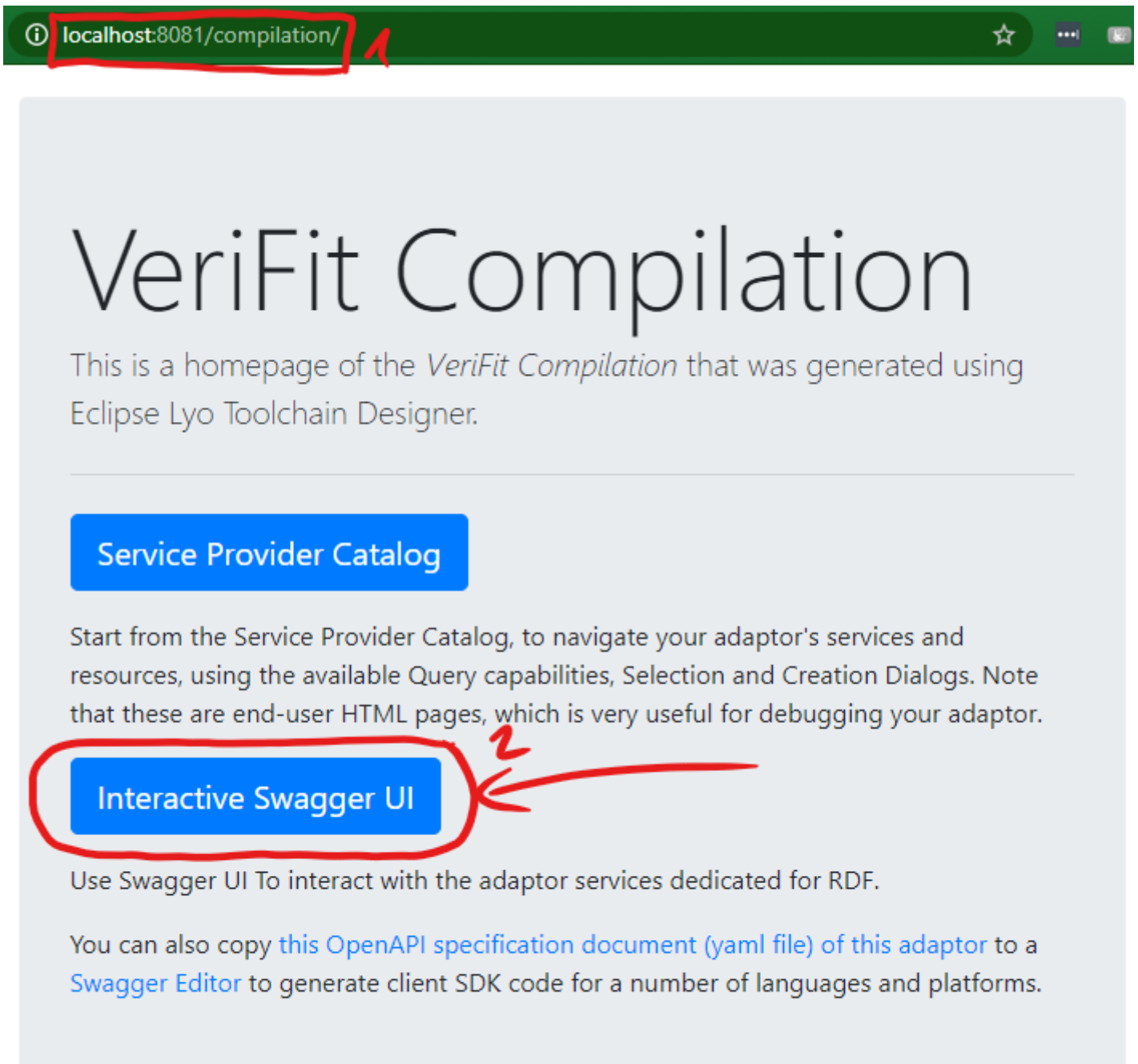
3. Analysis Using SwaggerUI

This guide includes links to the adapter's API. These links will only work if the adapter is running on localhost with default host:port configuration.

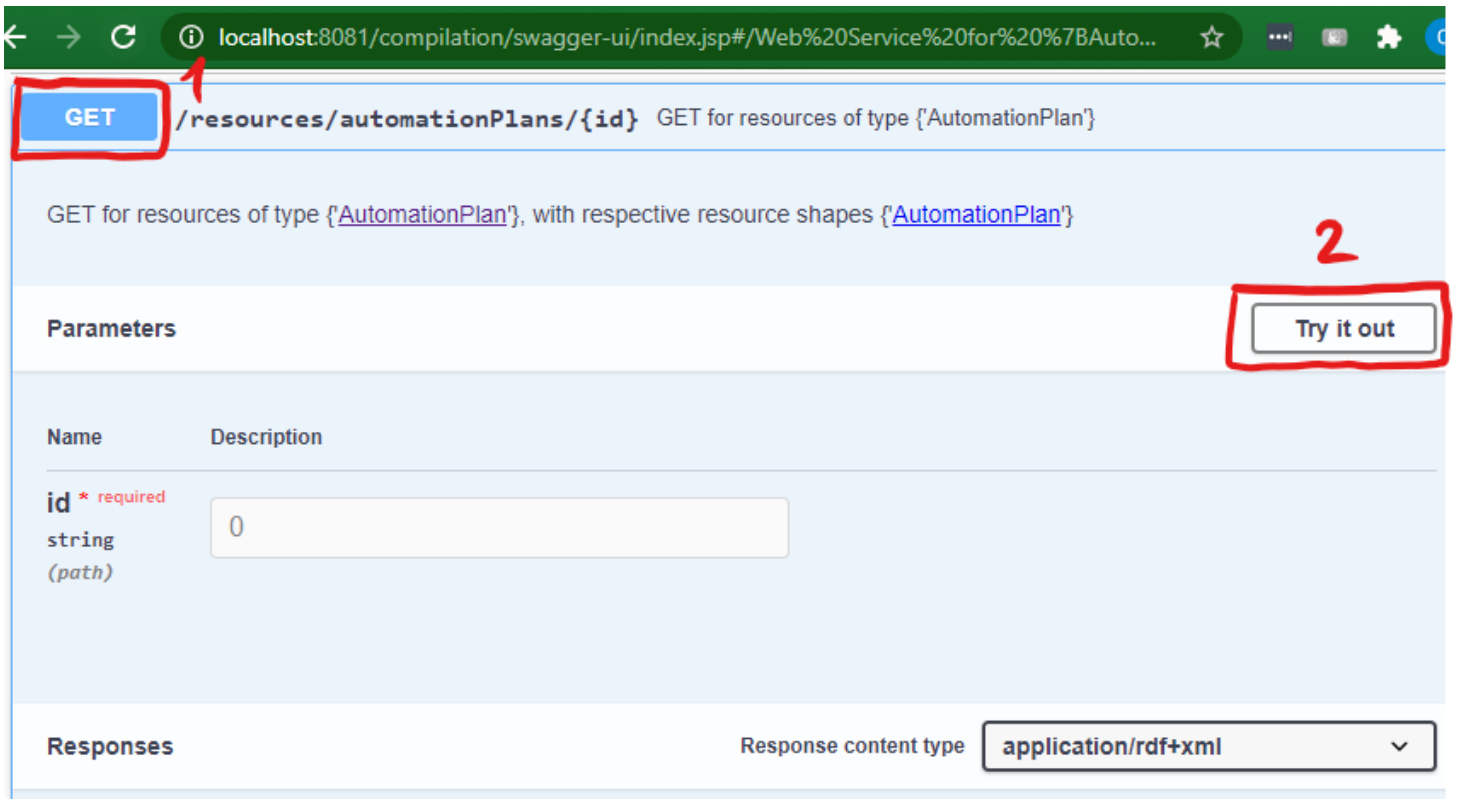
An alternative to this tutorial is the Postman Tutorial collection. That one, however, requires Postman to be installed.

1) Create an SUT

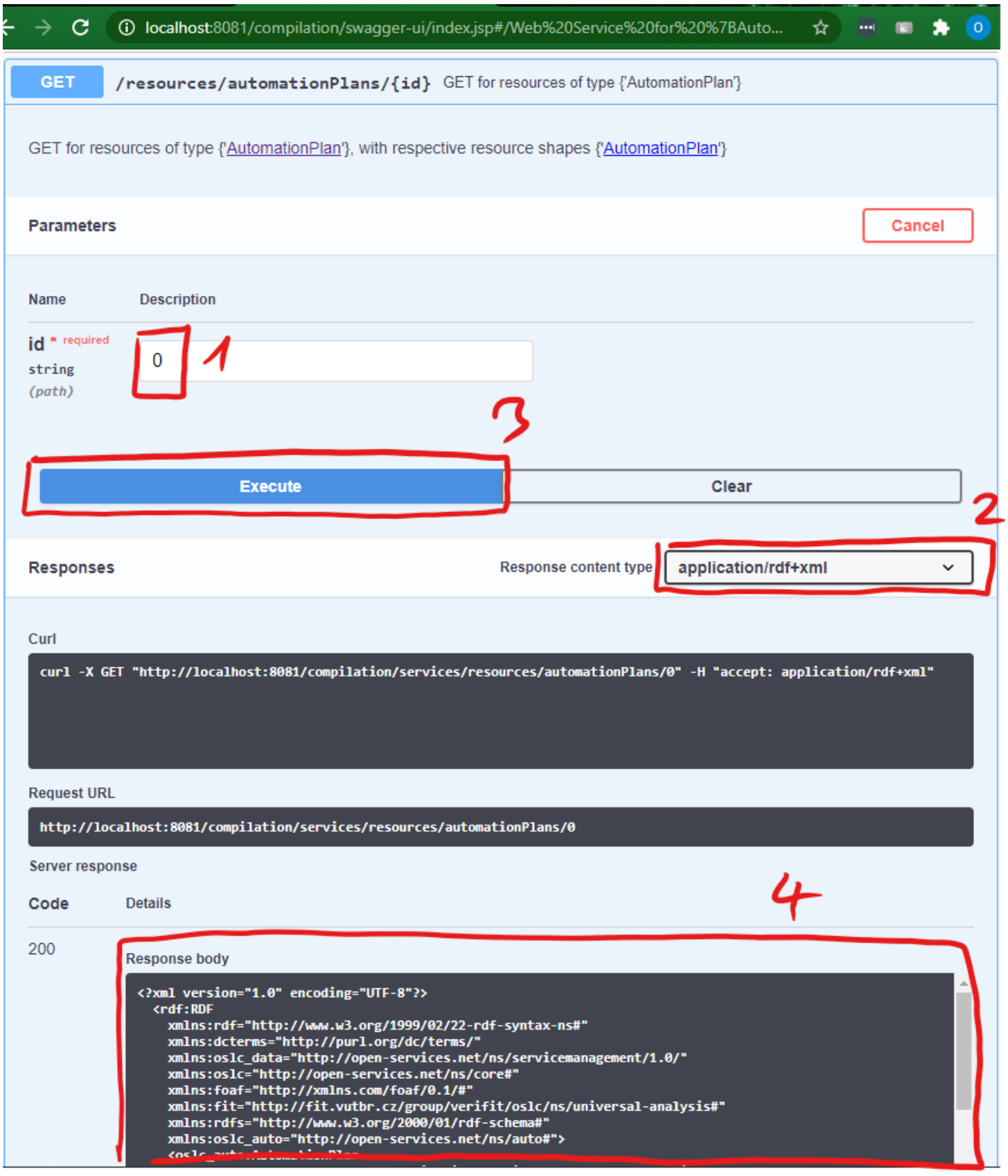
Open the Compilation adapter home page. And navigate to the swagger UI.



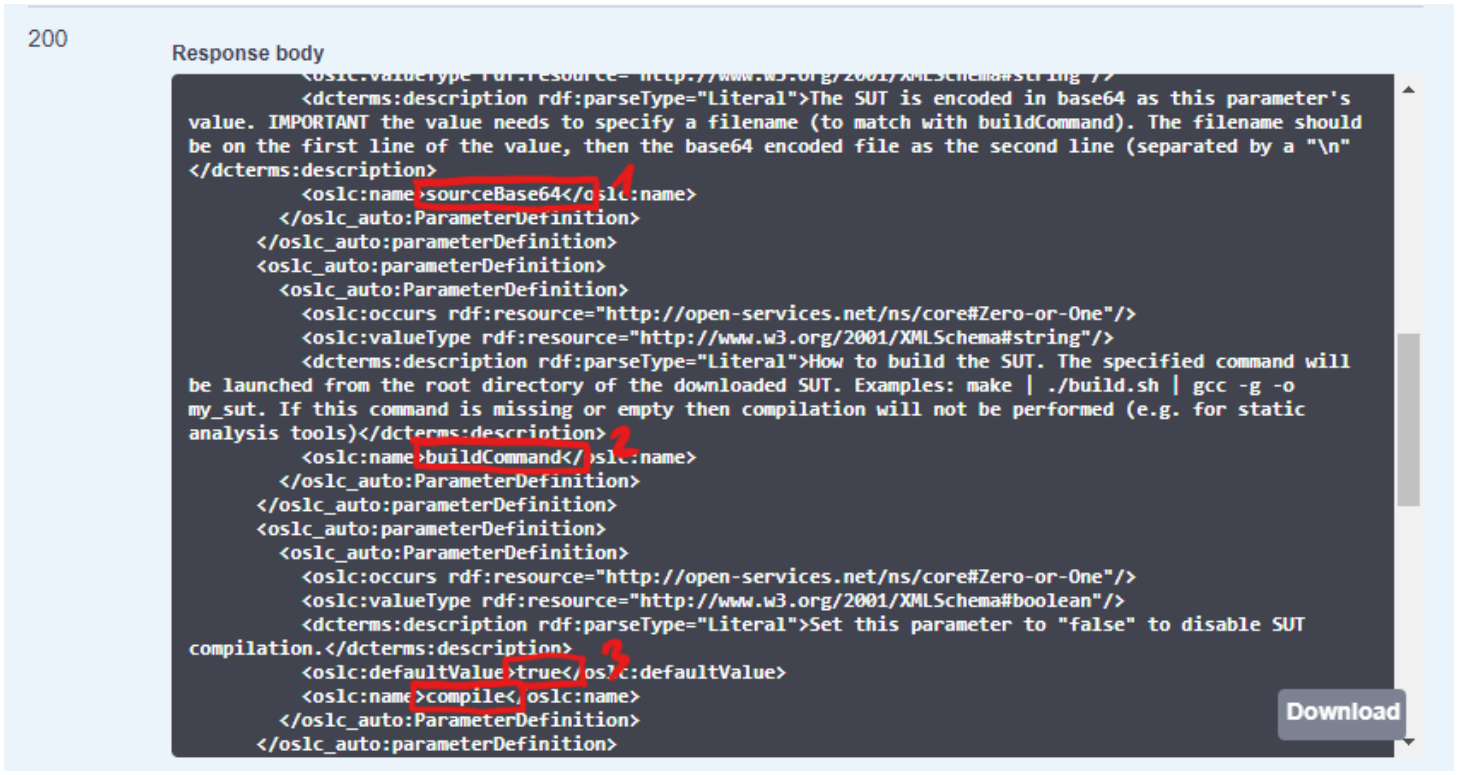
Find and expand the Automation Plan GET (or alt) capability (1) and click the "try it out" button (2).



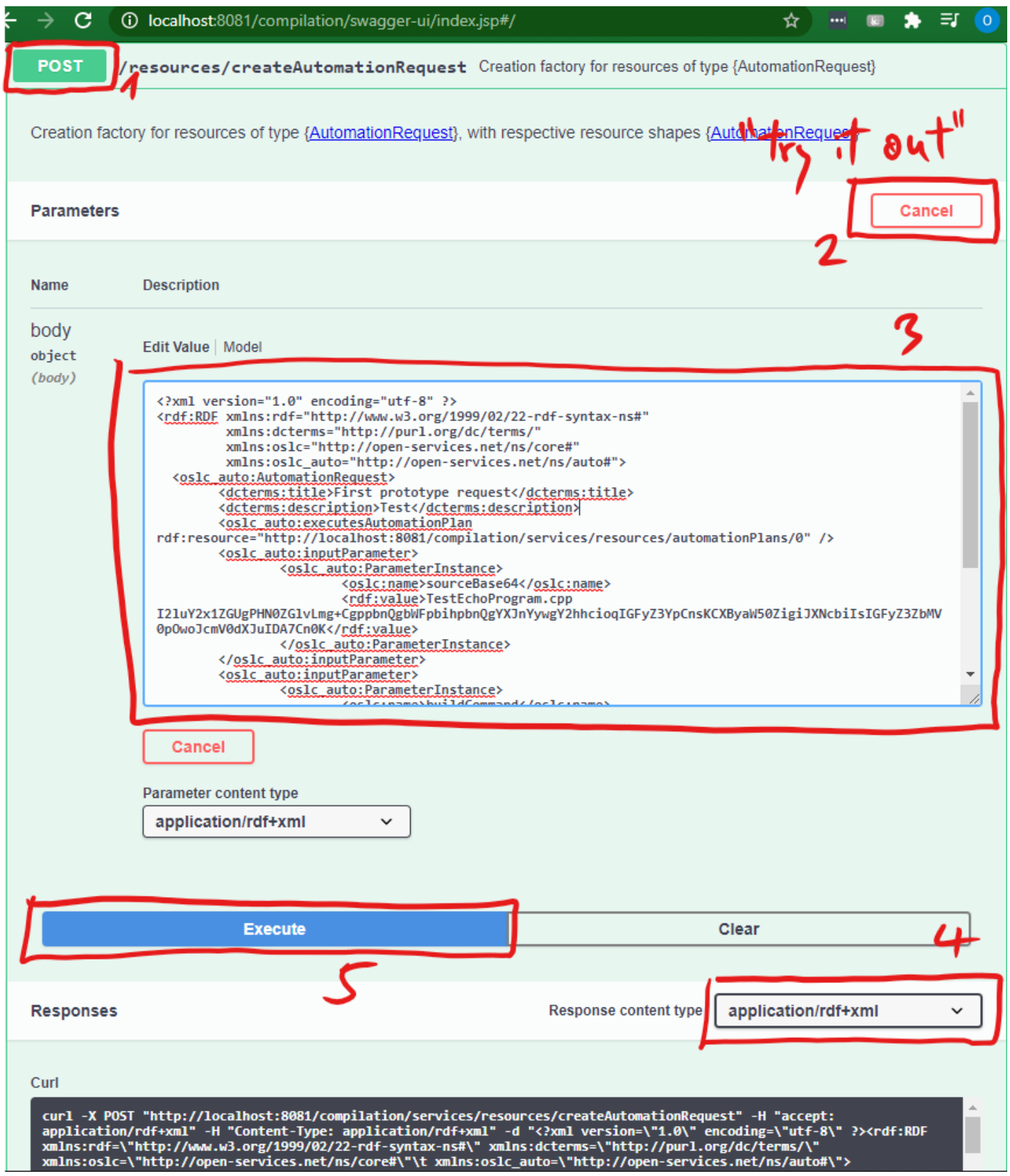
Set the ID to 0 (zero) (1) and pick your desired content type (2). Execute the request (3) and then look at the result (4) to learn about input parameters accepted by the Automation Plan which are defined using ParameterDefinition properties. The Automation Plan will look something like this.



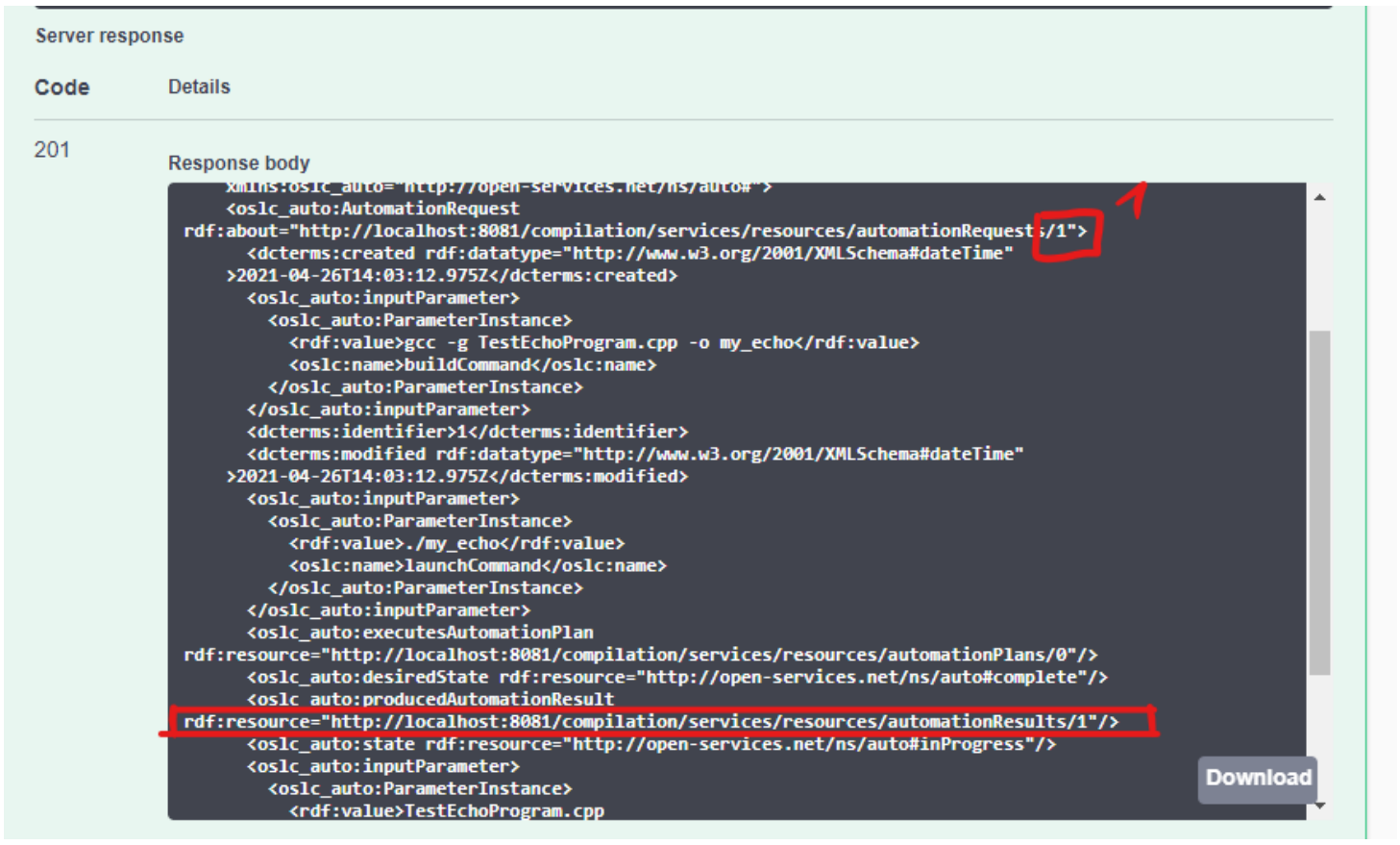
Input parameters will include a number of source* parameters (1), a build command and a launch command for the SUT (2), and other functional parameters like the compile parameter (3) which has a default value of true.



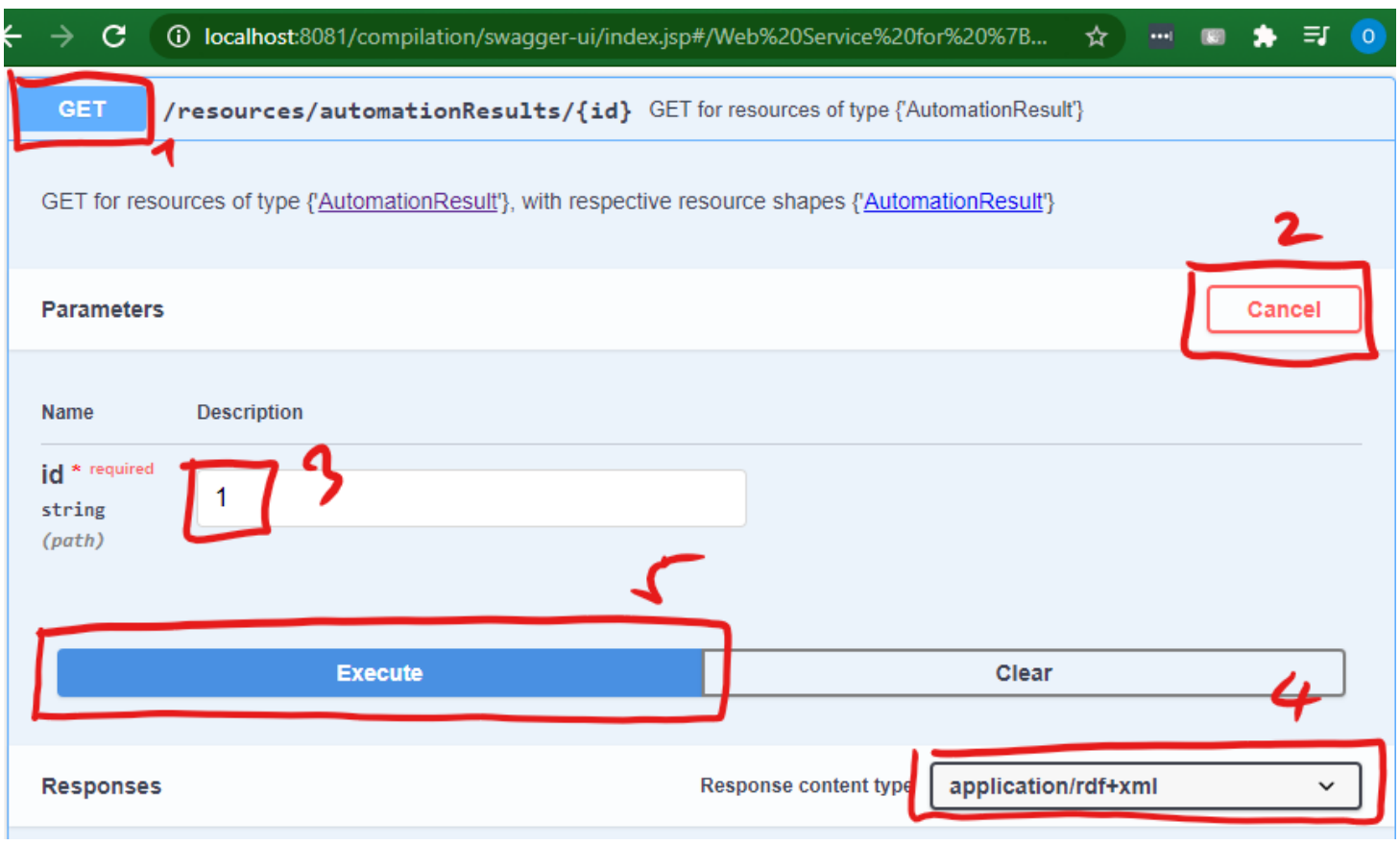
Next find and expand the Automation Request POST capability (1) and again click "try it out" (2). Paste your Automation Request into the input box (3), select your desired content type (4), and execute the request (5). The Automation Request needs to contain input parameters base on the executed Automation Plan and have a link to the executedAutomationPlan. Input parameters in our case are an SUT source (sourceBase64), an SUT launchCommand and an SUT buildCommand. Try this example request



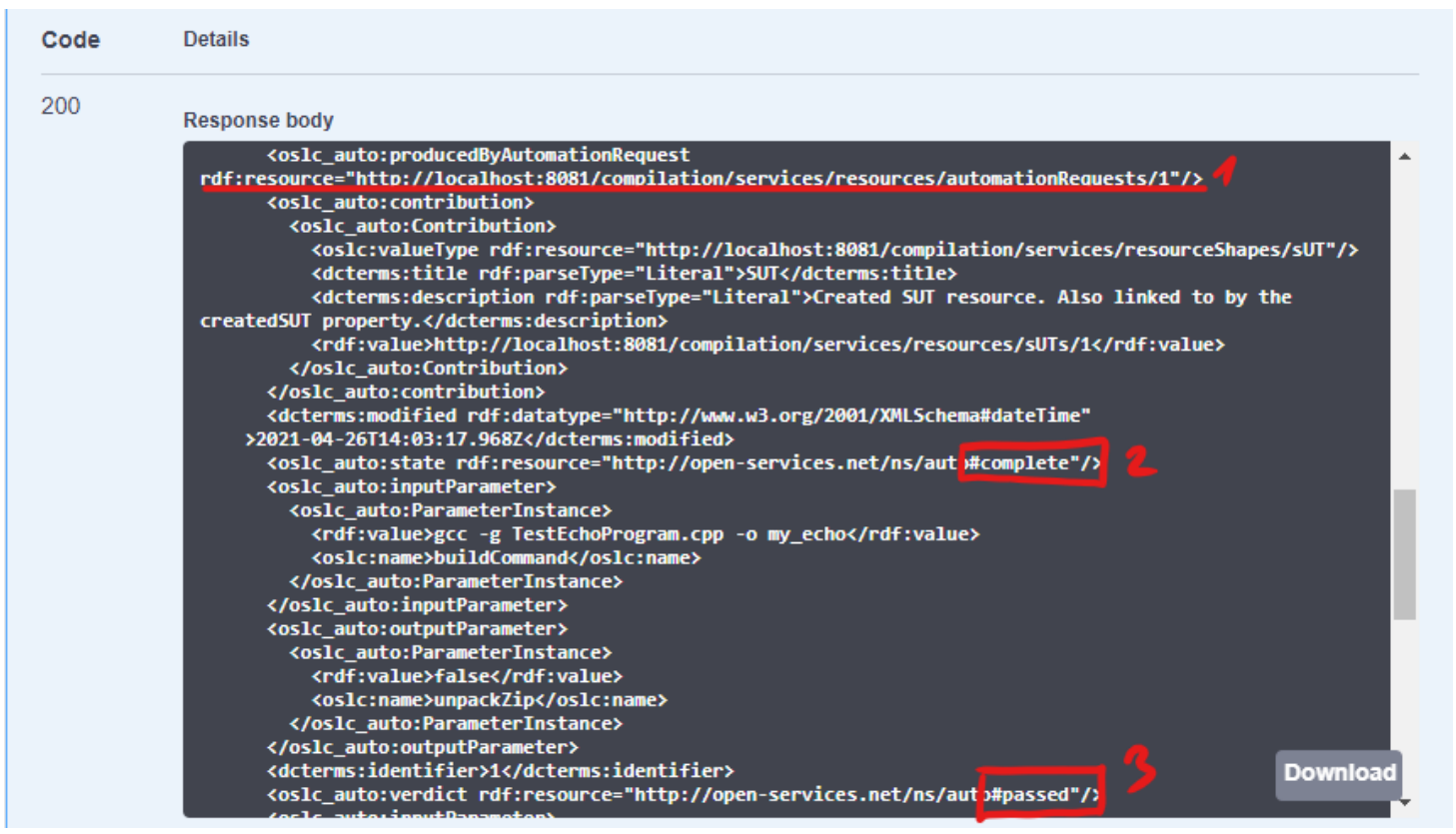
The response to your request will look something like this. The created Automation Request will have a URI with a numerical ID at the end (1). And its most important property for us now is a link to the producedAutomationResult (2). Notice that the numberical ID of the A.Result is the same as of the A.Request this can be used as a shorcut for finding A.Results.



Now find and expand the Automation Result GET (or alt) capability (1) and again click "try it out" (2). Set the ID to match the ID of your Automation Result (3) (from the producedAutomationResult property in the previous screenshot), select your desired content type (4), and execute the request (5).



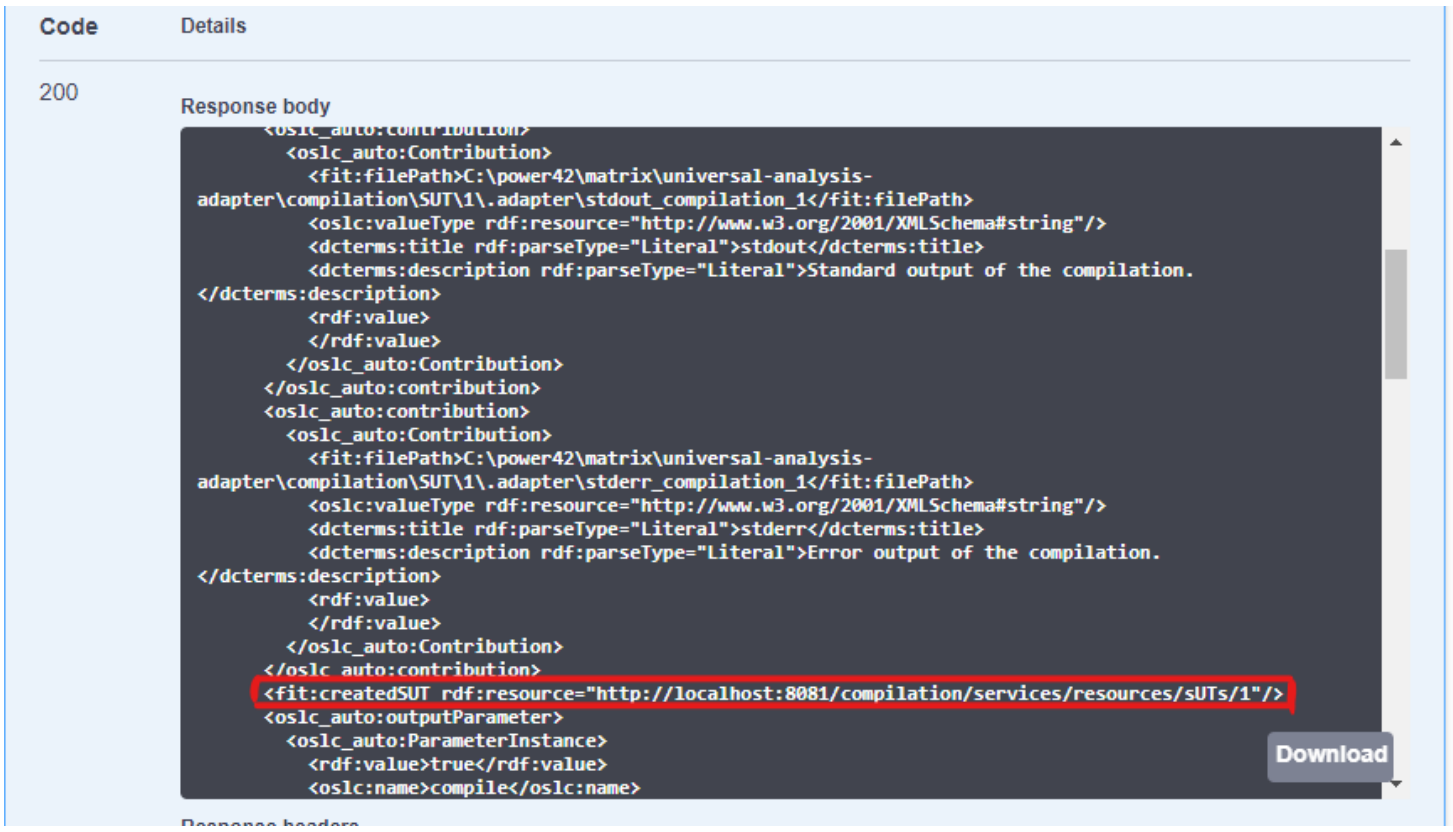
The most important properties right now are the state (2) and verdict (3) properties which say if the SUT creation was finished and if it was successful. The A.Result also contains a backward link to its associated A.Request (1). The A.Result will look something like this.



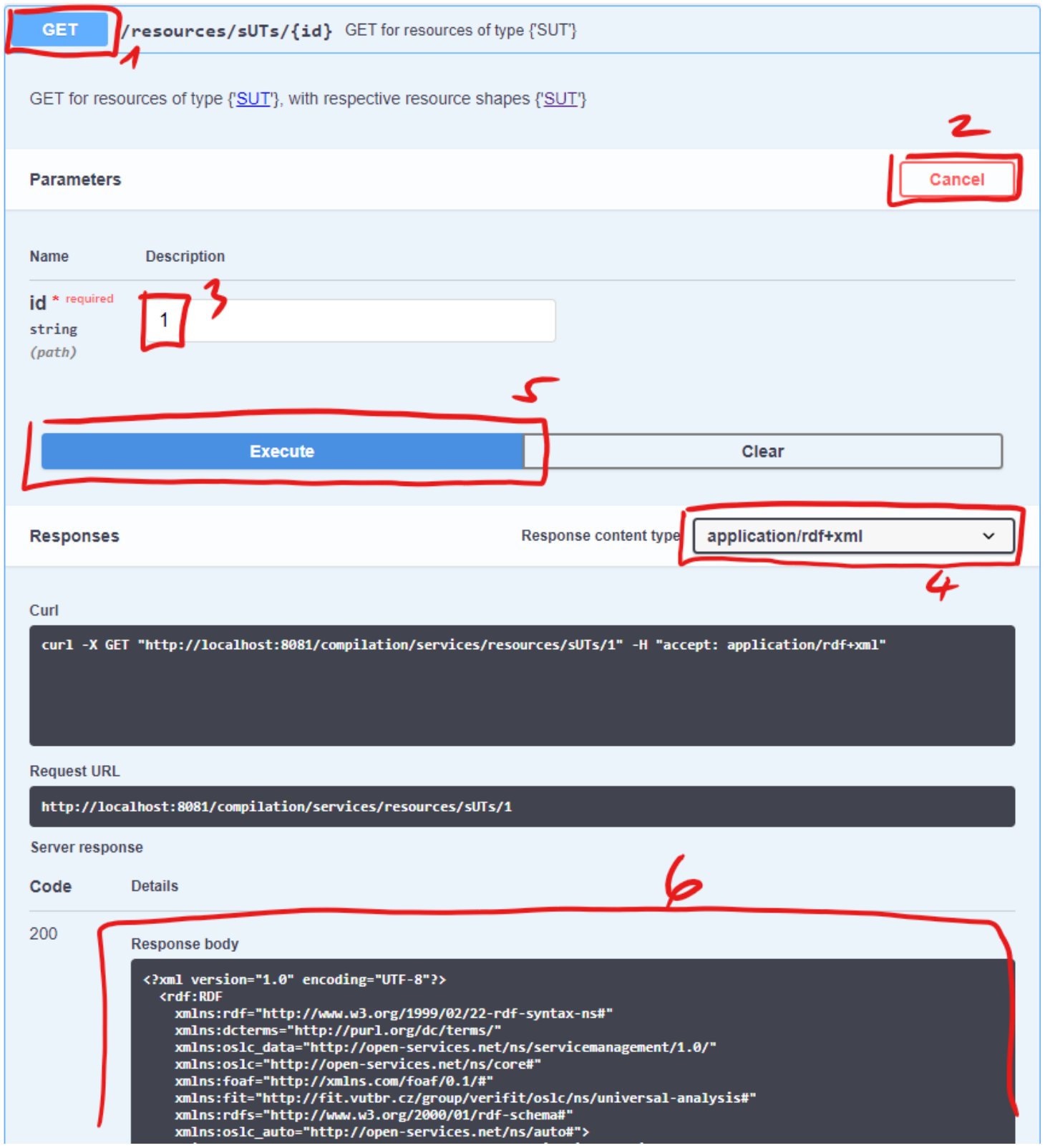
The A.Result will contain logs of the SUT creation process.

```
      <oslc_auto:contribution>
        <oslc_auto:Contribution>
          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
          <dcterms:title rdf:parseType="Literal">statusMessage</dcterms:title>
          <dcterms:description rdf:parseType="Literal">Status messages from the adapter about the execution.
</dcterms:description>
          <rdf:value>SUT fetch successful
Executing: gcc -g TestEchoProgram.cpp -o my_echo
   as: powershell.exe ./.adapter/exec_compilation_1.ps1 ""
   In dir: SUT\1
Compilation completed successfully
SUT resource created
</rdf:value>
        </oslc_auto:Contribution>
      </oslc_auto:contribution>
```

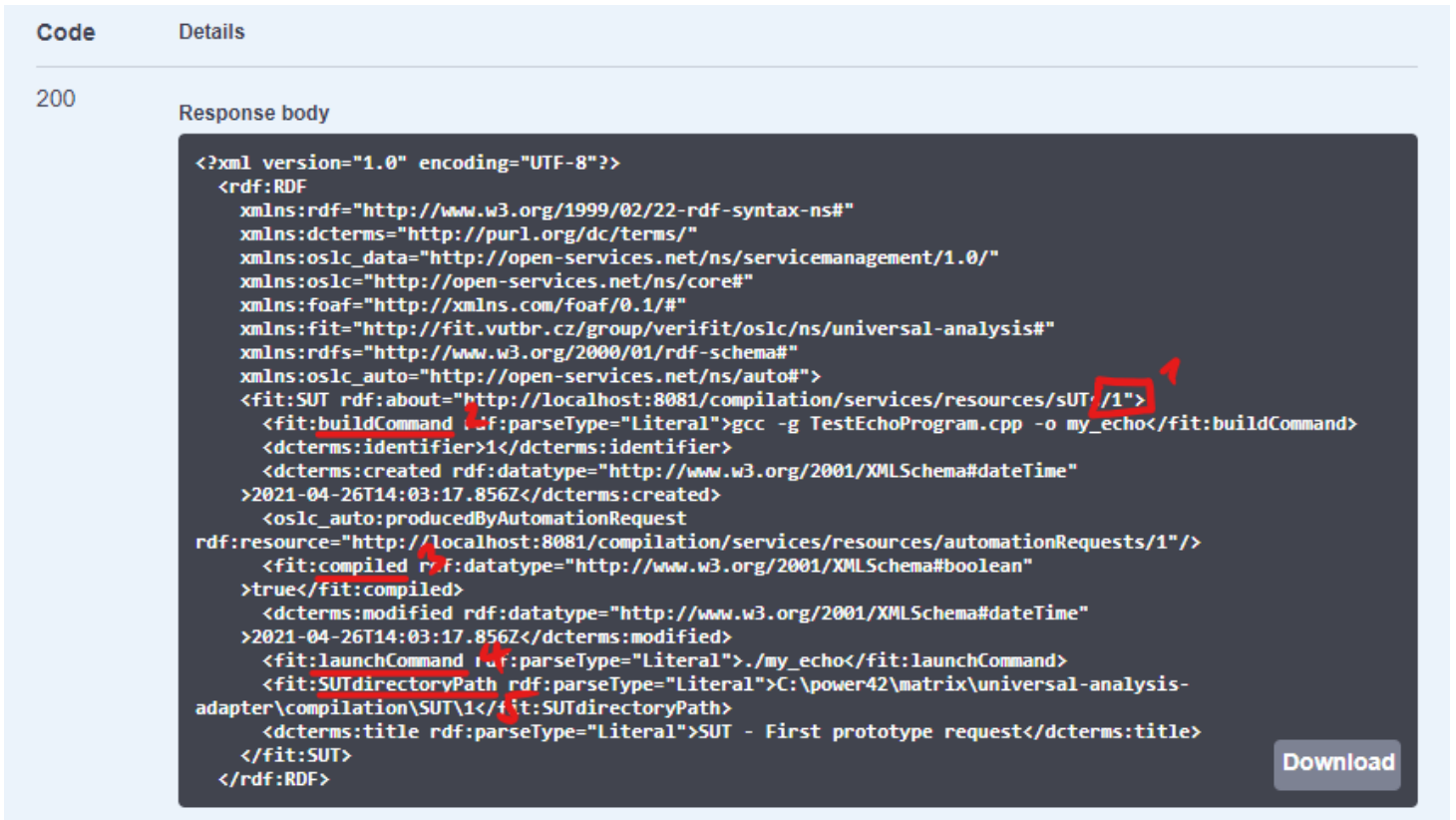
And most importantly the A.Result will contain a link to the createdSUT



To retrieve the created SUT resource navigate to the SUT GET capability (or alt) (1), click (2), set the ID based on the cretedSUT property from the last screenshot (3), pick your content type (4), execute the request (5), and look at the SUT resource in the received response (6).

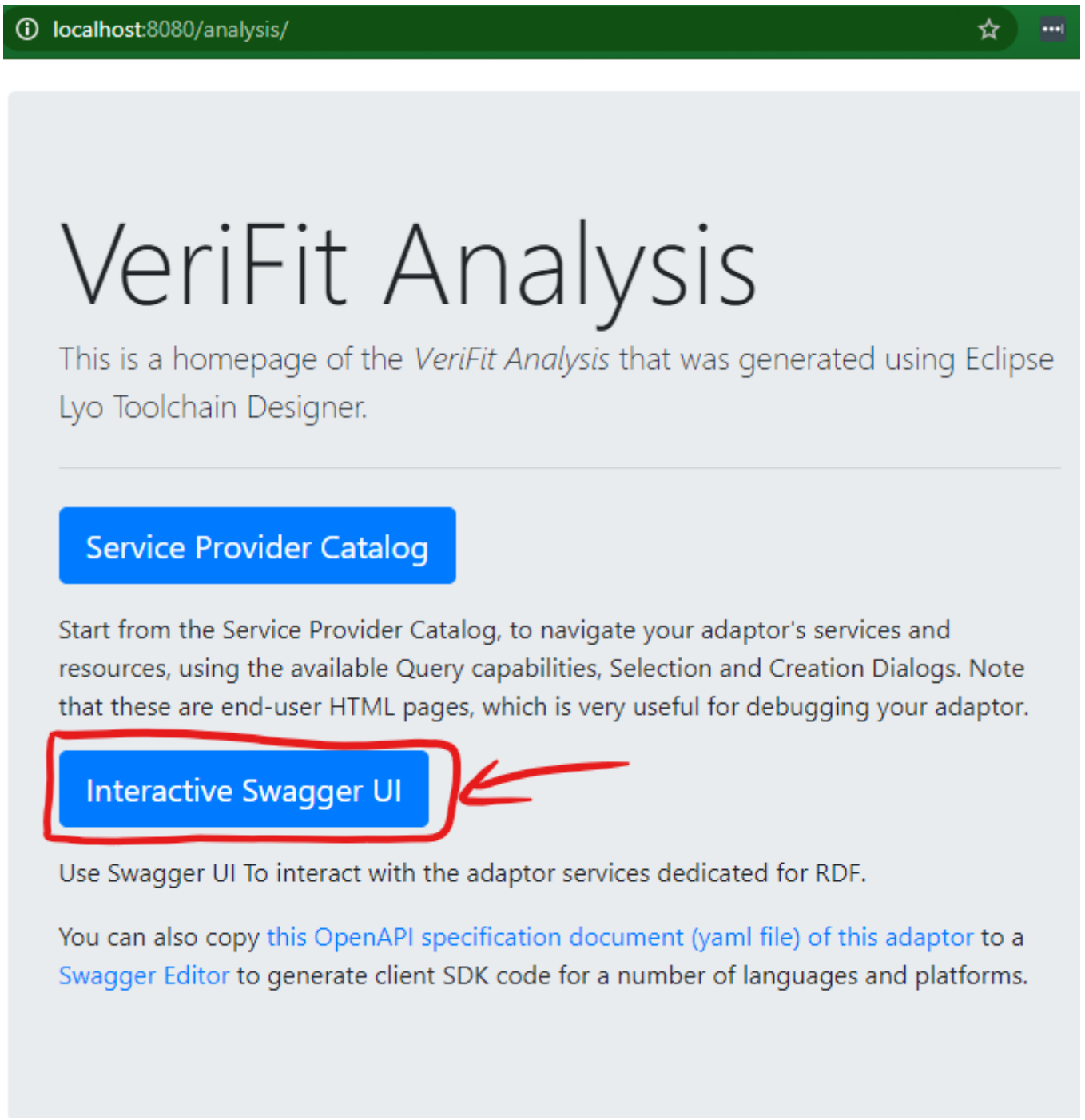


The SUT resource will have a URI (1) which needs to be remembered by the client in order to be used as an input parameter when requesting analysis. It will have properties which are then used by the analysis adapter such as the build command (2) and the launch command (4), a compilation flag (3), and a directory (5). The SUT resource will look something like this.

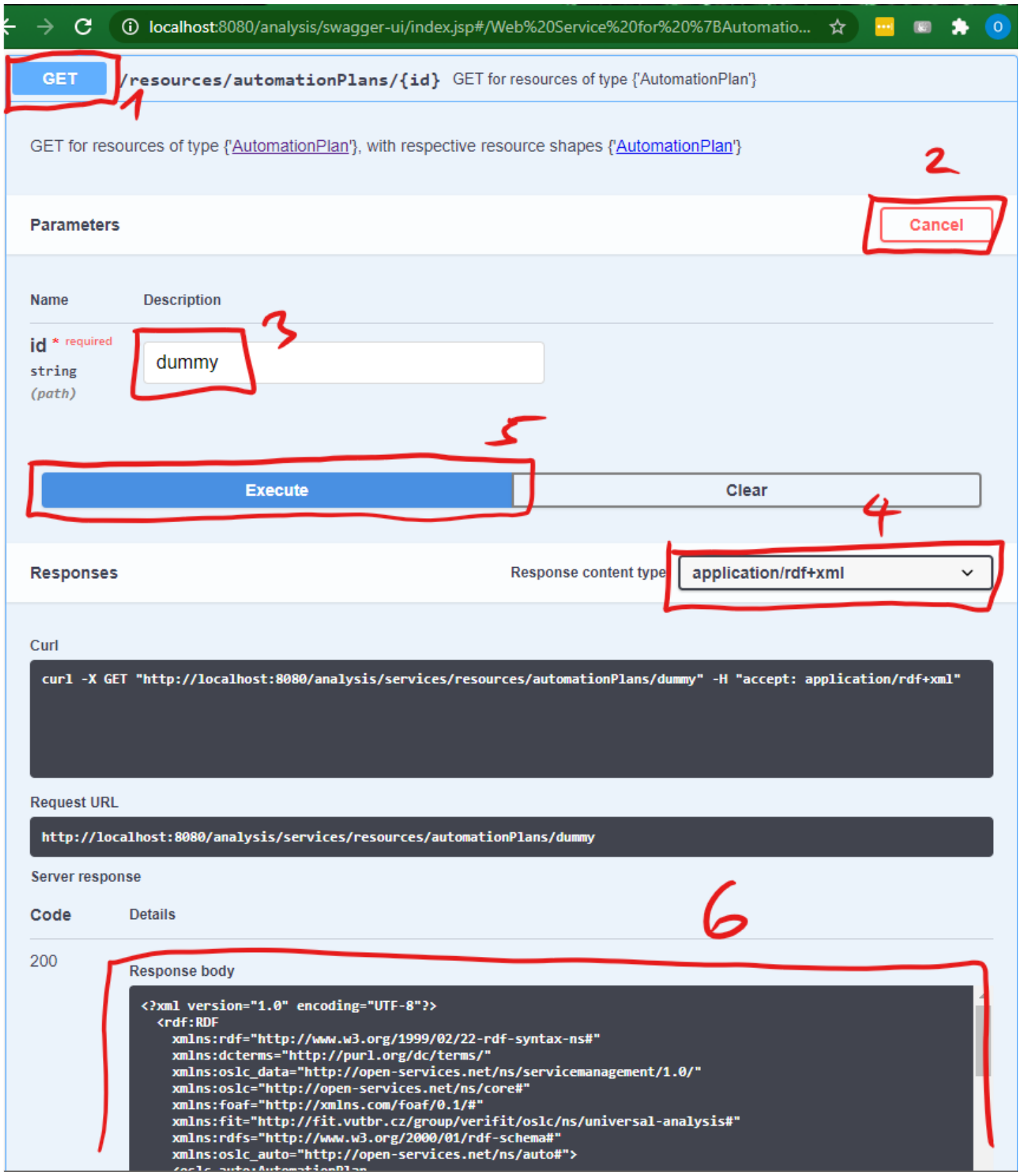


2) Execute Analysis

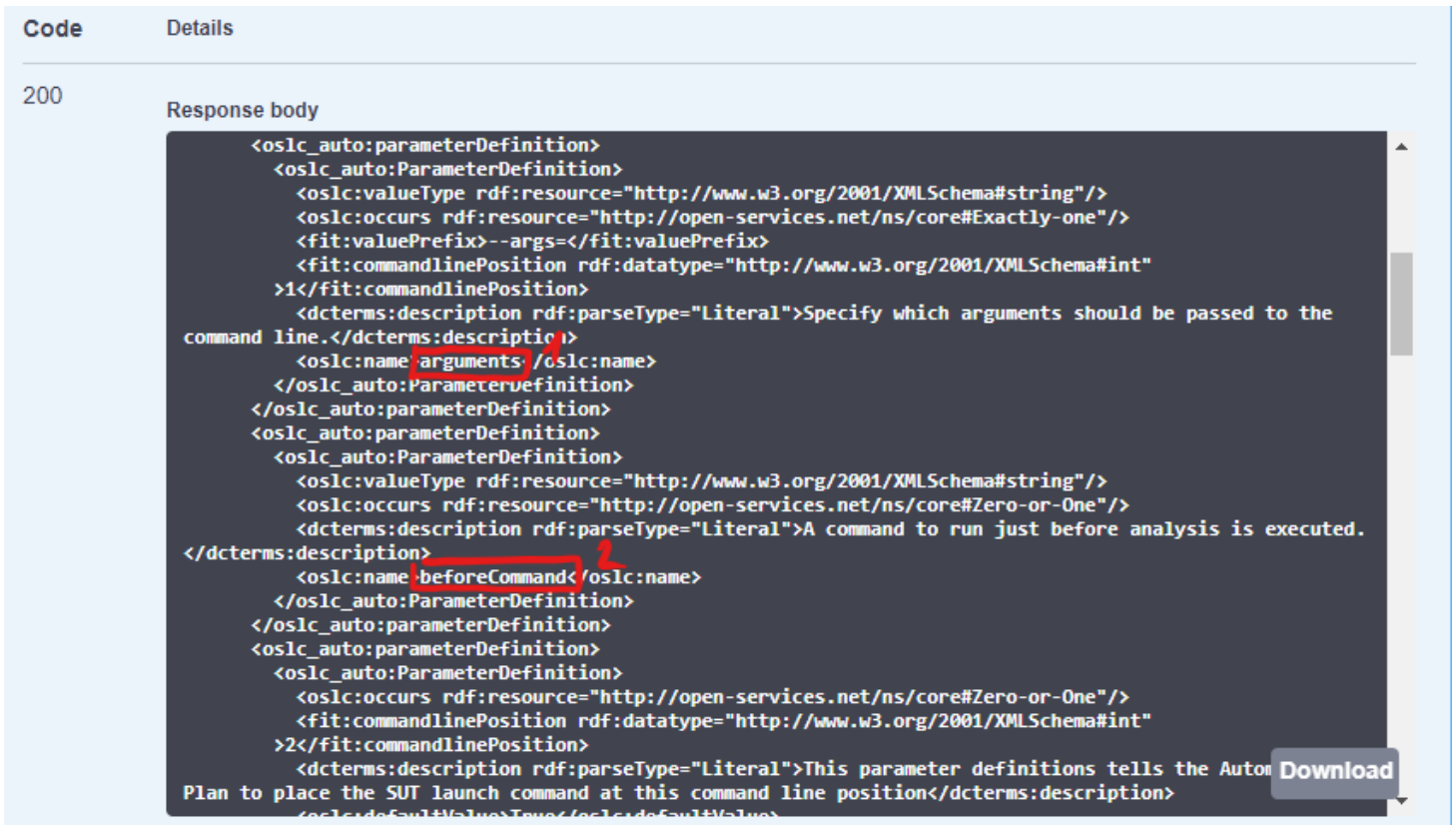
Open the Analysis adapter home page. And navigate to the swagger UI.



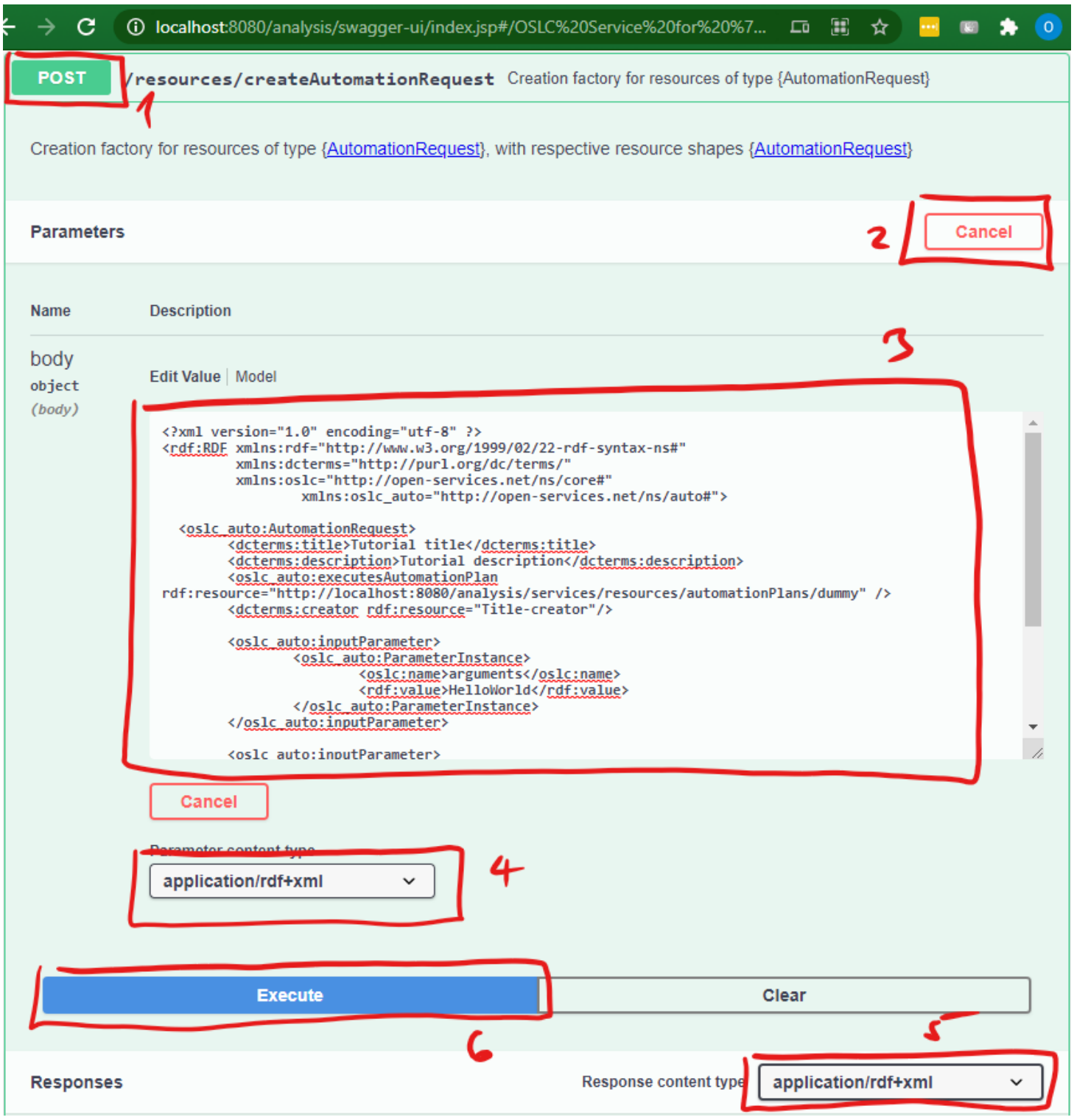
Find and expand the Automation Plan GET (or alt) capability (1) and click the "try it out" button (2). Set the ID to the ID of your analysis tool (3) and pick your desired content type (4). Execute the request (5) and then look at the result (6) to learn about input parameters accepted by the Automation Plan which are defined using ParameterDefinition properties. The Automation Plan will look something like this



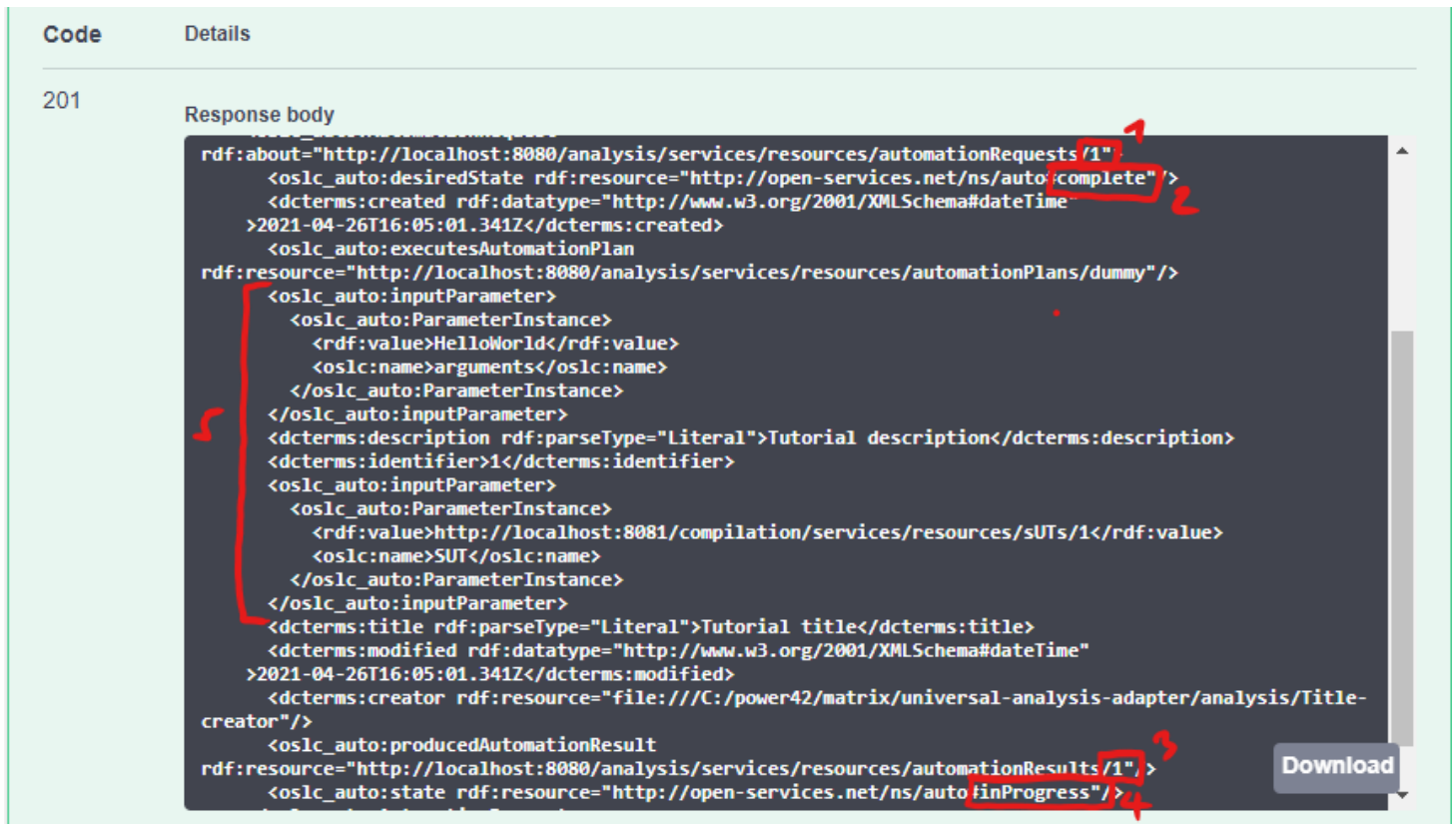
Parameter Definitions will match the ones defined by your for the analysis tool (e.g. 1) and there will be common adapter input parameters as well (e.g. 2).



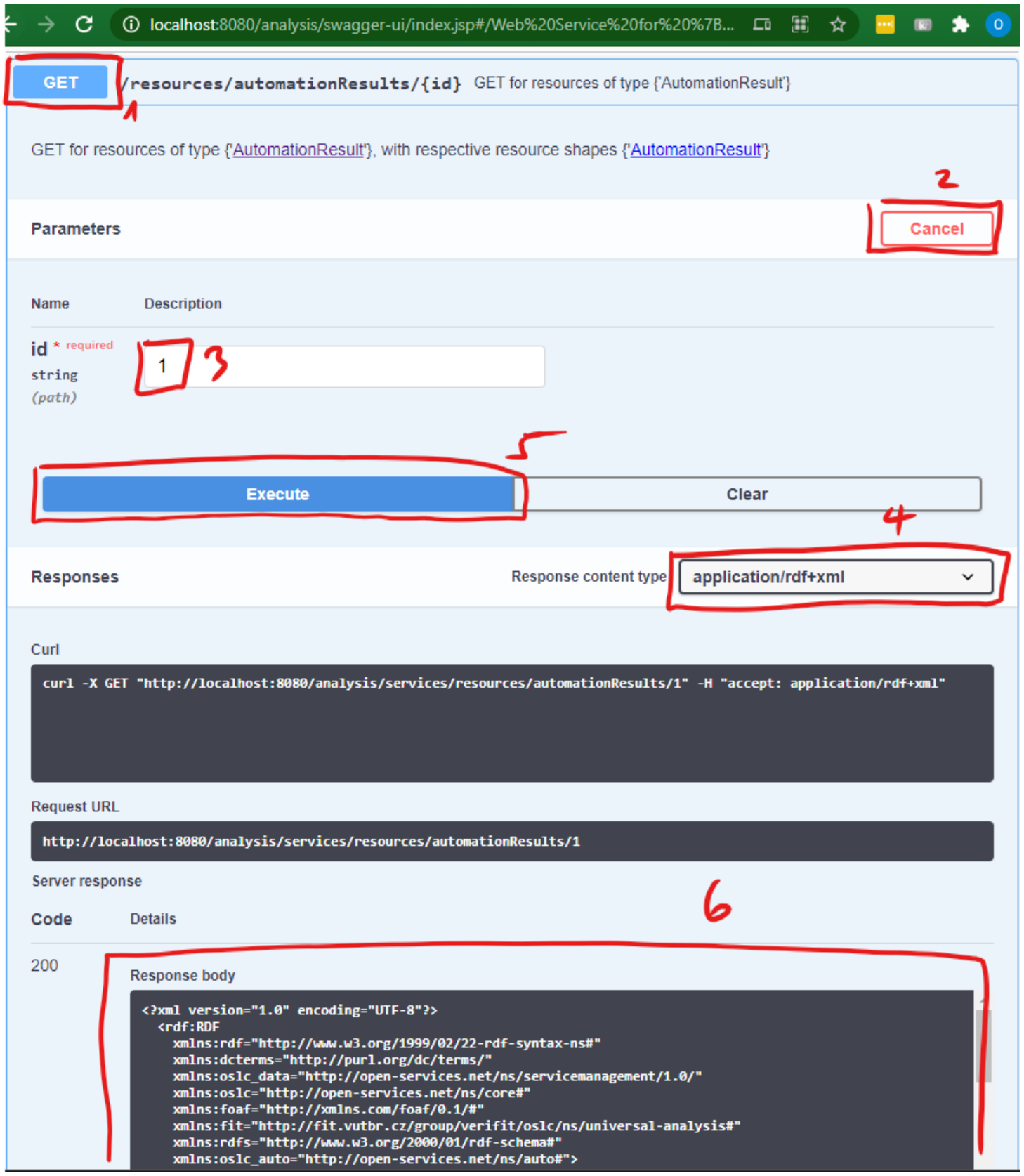
Next find and expand the Automation Request POST capability (1) and again click "try it out" (2). Paste your Automation Request into the input box (3), set the content type of the request body (4), select your desired content type (5), and execute the request (6). The Automation Request needs to contain input parameters base on the executed Automation Plan and have a link to the executedAutomationPlan. Input parameters in our case are a link to an SUT resource to be analysed and arguments for the analysis tool. Try this 📎 example request



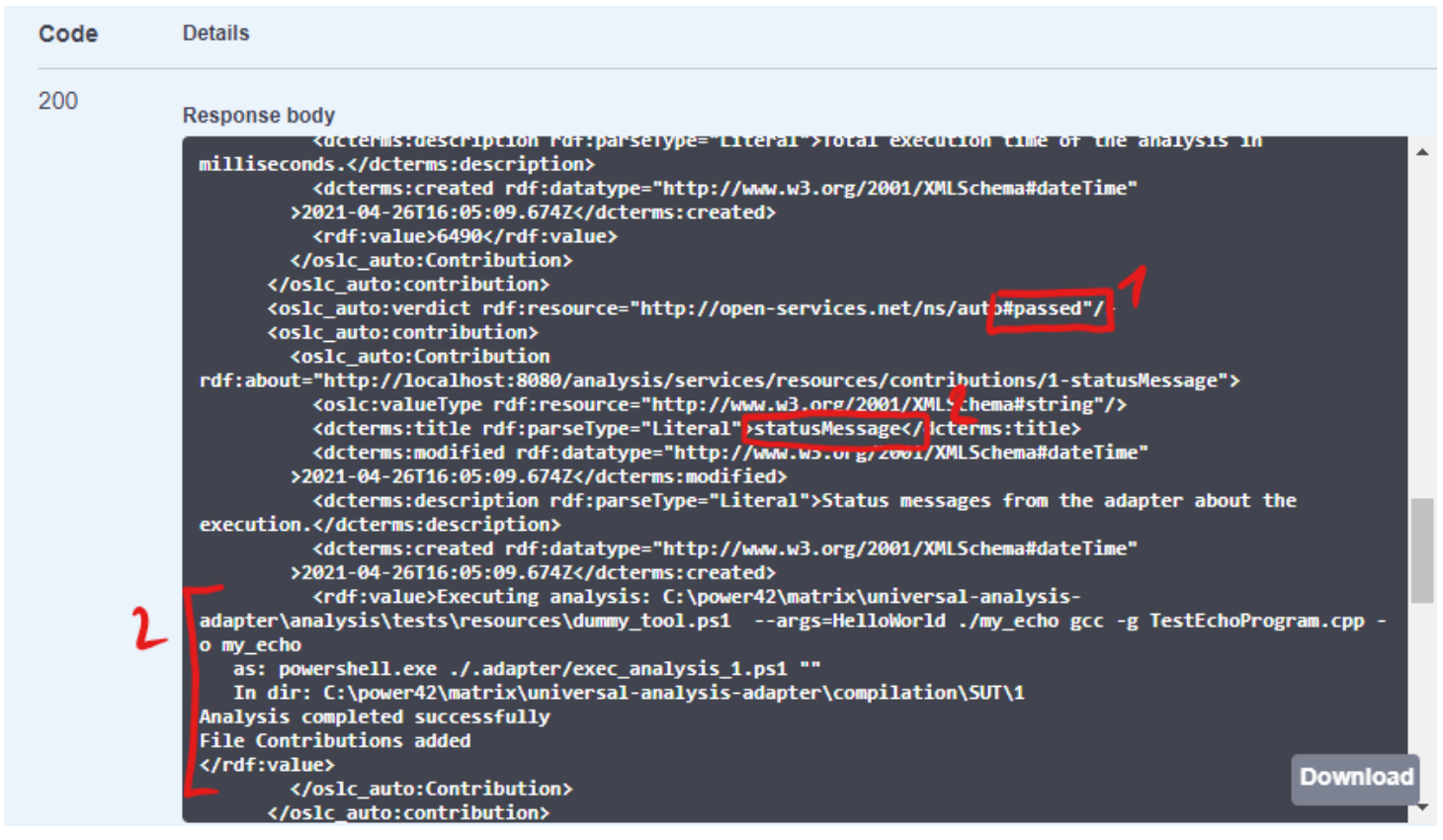
The response to your request will look something like this. The created Automation Request will have a URI with a numerical ID at the end (1). And its most important property for us now is a link to the producedAutomationResult (3). Other important properties include the desired state (2), state (4), and your input parameters (5). Notice that the numberical ID of the A.Result is the same as of the A.Request this can be used as a shorcut for finding A.Results.



Now find and expand the Automation Result GET (or alt) capability (1) and again click "try it out" (2). Set the ID to match the ID of your Automation Result (3) (from the producedAutomationResult property in the previous screenshot), select your desired content type (4), and execute the request (5). The result will be shown in (6).



The A.Result will look something like this. The most important properties right now are the state (5) and verdict (1) properties which say if the SUT creation was finished and if it was successful. Other important properties are output parameters which inform the client about parameters the adapter used on its own with default values (4). There will be status messages from the adapter about the execution process (2), standard output logs of the analysis (6, 7), return code of the analysis (3), and potential other contributions depending on the tool and the input parameters used.



200

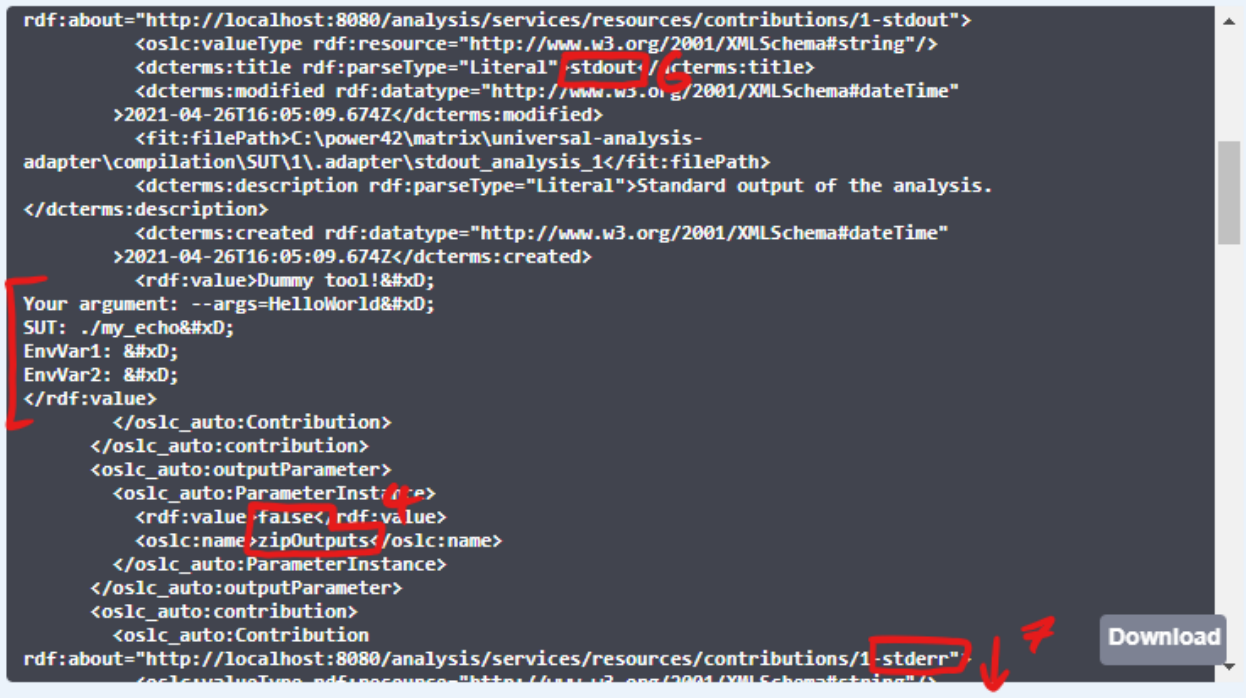
Response body


```
      <oslc_auto:contribution>
        <oslc_auto:Contribution
rdf:about="http://localhost:8080/analysis/services/resources/contributions/1-returnCode">
          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
          <dcterms:title rdf:parseType="Literal">returnCode</dcterms:title>
          <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"
        >2021-04-26T16:05:09.674Z</dcterms:modified>
          <dcterms:description rdf:parseType="Literal">Return code of the execution. If non-zero, then
the verdict will be #failed.</dcterms:description>
          <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"
        >2021-04-26T16:05:09.674Z</dcterms:created>
          <rdf:value>0</rdf:value>
        </oslc_auto:Contribution>
      </oslc_auto:contribution>
      <oslc_auto:inputParameter>
        <oslc_auto:ParameterInstance>
          <rdf:value>http://localhost:8081/compilation/services/resources/sUTs/1</rdf:value>
          <oslc:name>SUT</oslc:name>
        </oslc_auto:ParameterInstance>
      </oslc_auto:inputParameter>
      <oslc_auto:outputParameter>
        <oslc_auto:ParameterInstance>
          <rdf:value>True</rdf:value>
          <oslc:name>SUTbuildCommand</oslc:name>
        </oslc_auto:ParameterInstance>
      </oslc_auto:outputParameter>
      <dcterms:creator rdf:resource="file:///C:/power42/matrix/universal-analysis-adapter/analysi
creator"/>
      <oslc_auto:state rdf:resource="http://open-services.net/ns/aut#complete"/>
```

Code Details

200

Response body




```
rdf:about="http://localhost:8080/analysis/services/resources/contributions/1-stdout">
          <oslc:valueType rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
          <dcterms:title rdf:parseType="Literal">stdout</dcterms:title>
          <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"
        >2021-04-26T16:05:09.674Z</dcterms:modified>
          <fit:filePath>C:\power42\matrix\universal-analysis-
adapter\compilation\SUT\1\.adapter\stdout_analysis_1</fit:filePath>
          <dcterms:description rdf:parseType="Literal">Standard output of the analysis.
</dcterms:description>
          <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime"
        >2021-04-26T16:05:09.674Z</dcterms:created>
          <rdf:value>Dummy tool!&#xD;
Your argument: --args=HelloWorld&#xD;
SUT: ./my_echo&#xD;
EnvVar1: &#xD;
EnvVar2: &#xD;
</rdf:value>
        </oslc_auto:Contribution>
      </oslc_auto:contribution>
      <oslc_auto:outputParameter>
        <oslc_auto:ParameterInstance>
          <rdf:value>false</rdf:value>
          <oslc:name>zipOutputs</oslc:name>
        </oslc_auto:ParameterInstance>
      </oslc_auto:outputParameter>
      <oslc_auto:contribution>
        <oslc_auto:Contribution
rdf:about="http://localhost:8080/analysis/services/resources/contributions/1-stderr"
```

3) Links to example files for quick access

Compilation / SUT creation

- 📎 Automation Plan GET response
- 📎 Automation Request POST
- 📎 Automation Request POST response
- 📎 Automation Result GET response
- 📎 SUT GET response.

Analysis

- 📎 Automation Plan GET response
- 📎 Automation Request POST
- 📎 Automation Request POST response
- 📎 Automation Result GET response