Using the FHIR Validator

This page documents the use of the FHIR Validator jar for validation. Additionally, you can use the web-based interface to this jar at https://validator.fhir.org/ or see [Validating Resources] for further information and other options for validating resources.

- Using the validator
- 2 Downloading the validator
- 3 Running the validator
  - 3.1 JDK Version
- 4 Privacy Issues
- 5 Choosing what to validate
- 6 Managing Output
- 7 Choosing the version
- 8 Validating against an implementation guide
  - 8.1 Loading an implementation Guide
  - 8.2 What to validate against
    - 8.2.1 Validating a single resource in a bundle
  - 8.3 Validating References
    - 8.3.1 assumeValidRestReferences
- 9 Other Validation Parameters
  - 9.1 Locale
  - 9.2 Jurisdiction
  - 9.3 Terminology Server
    - 9.3.1 SNOMED CT
  - 9.4 MustSupport
  - 9.5 Extensions
  - 9.6 Invariants
  - 9.7 Questionnaires
  - 9.8 Extensible binding warnings
  - 9.9 Level
  - 9.10 Best Practices
  - 9.11 Language / Display
  - 9.12 Native Validation
  - 9.13 Logging
  - 9.14 SecurityOnly
  - 9.15 Unicode Control Chars
  - 9.16 Verbose Mode
  - 9.17 Show Validator Internal Times
  - 9.18 Validating IPS documents
  - 9.19 Validating CDA Documents
  - 9.20 Example URLs
  - 9.21 Proxy Settings (v5.2.1 onward)
  - 9.22 Legacy FHIR Path Setting
- 10 Other functionality provided by the Validator
  - 10.1 Version Conversion
  - 10.2 Comparing Profiles
  - 10.3 Working with StructureMaps
  - 10.4 Running Automated Tests

Using the validator

The validator is a standalone java validator that can be run from the command line to check that a resource is conformant to the base FHIR specification, and to any other applicable FHIR implementation guides and profiles, and other terminology rules.

The FHIR Validator is provided as open source code (see https://github.com/hapifhir/org.hl7.fhir.core) by the FHIR project in association with HL7, SmileCDR, and the HAPI FHIR project. From a java code point of view, the validator is part of the HAPI core library and available as part the HAPI distribution. HL7 acknowledges the support of the ONC in providing the validator to the community.

While the contributors to the validate make every effort to ensure that the validation is as technically correct as possible, users should review all errors and warnings against the various specifications and check that the validator is behaving correctly. Further note that there are some ways for a resource to be invalid that cannot be checked by an automated tool. Further discussion about validation and the correct functioning of the validator is usually done here on the conformance channel at chat.fhir.org. Users who find issues are welcome to contribute validator test cases.

The validator is provided to the public in the form of a pre-built command line jar, and also hosted at https://validator.fhir.org/. Alternatively, the validator code may be repackaged for use by other servers as part of the HAPI infrastructure. One subject public repackaging is at https://inferno.healthit.gov.

The validator is only provided to assist developers in checking the correctness of their implementations. It is not supported for use in production systems; implementers intending to use the validator in production pipelines or servers should seek a commercial support relationship with one of the providing organisations above (irrespective of how the validation code is packaged up).

Downloading the validator

The validator is provided to the public in the form of a pre-built command line jar, and also hosted at https://validator.fhir.org/.
To download the validator: [https://github.com/hapifhir/org.hl7.fhir.core/releases/latest/download/validator_cli.jar](https://github.com/hapifhir/org.hl7.fhir.core/releases/latest/download/validator_cli.jar)

If you find security issues with the validator please report them directly to fhir-director@hl7.org. Else, if you wish to discuss additional validation steps that may be appropriate for security reasons, please raise them at https://chat.fhir.org/#narrow/stream/179247-Security-and.20Privacy. See "-security-only" and "-no_unicode_bidi_control_chars" below.

Running the validator

Note that you should always use the current validator (see above), irrespective of which FHIR Release you are validating. You need a current version of java to run the validator:

```
java -jar validator_cli.jar [params]
```

The params control how the validator works, and are documented here.

You can also use the validator for other things than validation: Using the FHIR Validator to transform content, todo...

Note if you get an error from java "Out of Memory Error" -- Increase your java heap size configuration

JDK Version

The validator is tested to run on all currently support LTS versions of Java (at the time of writing this documentation, JDK 11 and 17)

Privacy Issues

The validator does not keep a copy of the resources, or post a copy of them anywhere, except in any logs that configure it to keep. The validator can be used with protected private information.

By default, the validator does use the terminology server http://tx.fhir.org to validate concepts from terminologies such as Snomed CT, LOINC, RxNorm, ICD-10 and so forth. Only the concepts are passed to the server, and no contextual information about what is being validated. You can check what information is passed to the server using the -txLog parameter. The server tx.fhir.org does log the concept and the IP address it is submitted from, a rolling log that only lasts a few hours. If the fact that these concepts are posted to the terminology server is deemed to be a privacy concern, you can host your own terminology server, and direct the terminology calls there using the -tx parameter. In a production context, this is neccessary, since there is no SLA guaranteeing availability for tx.fhir.org.

Choosing what to validate

The validator takes a series of parameters that indicate the resources to validator. There must be at least one source param.

Each source parameter can contain either:

- a URL that returns the resource to validate (authentication is not supported)
- a filename (relative to the current directory, or absolute)
- a directory that contains resources to validate (all files are validated if they are recognised as resources)
- a pattern: a directory followed by a filename with an embedded asterisk. E.g. foo*-examples.xml or someresource.*, etc

All other parameters are 'named parameters' - e.g. -name value. Any parameter preceded by a recognised name is interpreted as a source parameter

```
java -jar validator_cli.jar /tmp/resource.json
```

or

```
java -jar validator_cli.jar c:\temp\patient.xml
```

The validator can validate resources in the following formats:

- JSON
- XML
- Turtle
- Smart health cards as JWT (signed JWTs), as downloadable files, or as the text representation of QR Codes. QR codes as images are on the todo list

If a one of the following file extensions is provided, the format is implied from the file extension: .json, .xml, .ttl, .jwt, .jws. Otherwise, the content is inspected to see whether it can be parsed using any of these formats

Managing Output

By default, the outcome of validating is simply printed to std out (the console / terminal / command window).

If the parameter -output is provided, a file will be created to contain the output. The file will contain an OperationOutcome, or if more than one resource is found, a Bundle of OperationOutcome resources
java -jar validator_cli.jar c:\temp\patient.xml -output c:\temp\validation.xml

If the parameter -html-output is provided, an HTML file will be created with a presentation of the outcomes of the validation. This presentation also includes additional information about slice matching, if this was performed.

java -jar validator_cli.jar c:\temp\patient.xml -html-output c:\temp\validation.html

The HTML output has been tested with Chrome, Firefox, Safari, and Edge.

You can specify an output style that changes the text that’s displayed to std out using the -output-style parameter, which can have one of the following values:

<table>
<thead>
<tr>
<th>value</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(not present)</td>
<td>The default output (defaults to this if no output specified)</td>
</tr>
<tr>
<td>eslint-</td>
<td>The compact ESLint format (see ESLint documentation)</td>
</tr>
<tr>
<td>compact</td>
<td></td>
</tr>
<tr>
<td>csv</td>
<td>A comma-separated output similar to eslint compact, but suitable for processing by a spreadsheet program such as Excel</td>
</tr>
<tr>
<td>xml</td>
<td>Render the output as an XML operation outcome (or bundle of operation outcomes) (default output style if an output is specified)</td>
</tr>
<tr>
<td>json</td>
<td>Render the output as a JSON operation outcome (or bundle of operation outcomes) (default output style if an output is specified and the filename ends with '.json')</td>
</tr>
</tbody>
</table>

The specified output style applies whether or not an output destination is specified.

Choosing the version

The validator always checks the resource against the base specification. You can specify the version, or you can leave the validator to figure it out.

If you don't specify a version, the validator will look through the source material - both the instances to validate, and any reference source in the -ig parameter (see below) and determine which version is in scope. If the version isn't found, it will default to the current released version of the specification (4.0.1).

You may not want to validate against that version, so the first thing to do is to specify which version of the spec to use.

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0

Valid values for version are

<table>
<thead>
<tr>
<th>R2</th>
<th>R2B</th>
<th>R3</th>
<th>R4</th>
<th>R4B</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.4</td>
<td>3.0</td>
<td>4.0</td>
<td>4.3</td>
<td>5.0</td>
</tr>
<tr>
<td>1.0.2</td>
<td>1.4.0</td>
<td>3.0.2</td>
<td>4.0.1</td>
<td>4.3.0</td>
<td>5.0.0</td>
</tr>
</tbody>
</table>

Note: alternatively, you can specify the version -defn or -ig parameters:

java -jar validator_cli.jar c:\temp\patient.xml -defn hl7.fhir.r3.core#3.0.2
java -jar validator_cli.jar c:\temp\patient.xml -ig hl7.fhir.r3.core

These two are synonymous with specifying the version, and maintained for backwards compatibility. It's simpler to use the version parameter.

Note: validating against the base FHIR specification requires that the package for the specific version be installed in your FHIR Package Cache. If it's not, the validator will download it and install it.

Validating against an implementation guide

The validator can validate against an implementation guide. Do this involves 2 steps:

- loading the package for the implementation guide
- telling the validator what to validate against

Loading an implementation Guide

Tell the validator to load the package for an implementation guide using the -ig parameter:

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -ig hl7.fhir.us.core#1.0.1

The package validator loads the relevant implementation guide. Rather than nominating the the package id, you can also provide the canonical url of the implementation guide:
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -ig http://hl7.org/fhir/us/core%7C1.0.1

You don't have to specify the version:

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -ig hl7.fhir.us.core
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -ig http://hl7.org/fhir/us/core

In this case, the current published version of the IG is used.

Advanced note: If the IG refers to a package, like this, the validator understands several special versions: current, current$(branch) and dev. Current means the current build, from the github master/main branch. If you want a variant branch, you can use current$branch and name the branch. dev means the latest build you did locally

Alternatively, the -ig parameter can contain:

- a URL that returns a relevant resource (profile, extension definition, code system or value set) to load
- the name of a file that contains relevant resource to load, i.e. this can be a single profile file
- the name of a directory that contains relevant resources (scan and load *.xml, *.json, and *.ttl - any that can be parsed)
- the name of a gzipped tarball (e.g. package.tgz) that contains the relevant resources to load

Notes:

- Packages will be installed in your FHIR Package Cache as required
- Packages are assumed to have the same underlying fhir version as that specified in the -version parameter. If you want the implementation guide to be loaded for a different version of FHIR, you can prefix the IG with the appropriate version in square brackets: (-ig [[fhirVer]][id] -[igVer]).
- if you want to validate against the current build version (pre-publication) of an implementation guide auto-published through build.fhir.org, use 'current' as the version.
- if you want to validate against an implementation that you built yourself using the IG publisher on your own machine, use 'dev' as the version.
- if you have problems with an implementation guide, please ask on [the conformance stream on chat.fhir.org]
- you can load more than one version of an implementation guide, though this is not usually very useful

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -ig hl7.fhir.us.core#1.0.1 -ig hl7.fhir.us.core#1.1.0

What to validate against

Use the -profile parameter to tell the validator what to validate against:

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -ig hl7.fhir.us.core#1.0.1 -profile http://hl7.org/fhir/us/core/StructureDefinition/us-core-patient

The -profile parameter is the canonical URL for the profile you wish to validate against. This is usually clearly specified on the page where the profile is published. If the profile you specify has not been loaded through one of the implementation guides, the validator will try and load it directly from the canonical url, but it's better to load it with an -ig parameter first.

You can nominate more than one profile to validate against


If the Implementation Guide specifies a global profile that applies to all uses of a conformance resource, then you can nominate the canonical URL of the implementation guide resource itself:

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -ig hl7.fhir.us.core#1.0.1 -profile http://hl7.org/fhir/us/core

If the implementation guide doesn't specify a global profile for the relevant type, you'll get an error.

Validating a single resource in a bundle

You might want to validate a single resource in a bundle without validating the entire bundle. You can't use the -profile parameter because that's the profile for the bundle itself. It's possible to write a bundle profile that invokes the right profile for the right resource, but this is very tedious. Instead, you can just ask the validator to validate a particular resource in the bundle against a given profile:

-bundle (entry rule) (profile url)

This invokes the nominated profile (by canonical URL) on any entry in any bundle validated that meets the entry rule. The entry rule is either a Resource name, a integer index, or both:

- Patient - validate any patient against the nominated profile
- 1 - validate the 1th resource (actually the second - index is 0 based) against the nominated profile
- Patient:0 - validate the first patient resource against the nominated profile

E.g:
Validating References

In many cases, profiles make rules about the content of references. E.g. the subject of an Observation must be a Patient, and it must be a patient that conforms to a particular patient. In order to test rules like this, the validator must be able to fetch the target of a reference, and test whether the target conforms to the profile.

Fetching .. todo...

**assumeValidRestReferences**

If the validator can't fetch target resources, it can at least be instructed validate that the target resource is correct. For instance, if the allowed target types are (Patient or RelatedPerson), and the reference is Group/12345, then this would be usually be an error. However this is not actually explicitly stated in the FHIR specification - a reference could be to http://some.server/somewhere - it doesn't need to look like a valid FHIR RESTful reference. In fact, it's not an error even if it happens to. So by default, the validator can assume nothing about the type of resource from the content of the reference itself. However most implementers do follow those rules - a reference to either Group/12345 or http://some-server/somewhere/Group/12345 is definitely to a Group resource. (and it's definitely recommended to follow these rules).

The flag assumeValidRestReferences instructs the validator to use the type found in references that look like valid RESTful URLs when validating the type of the reference.

Other Validation Parameters

There are other validation parameters that affect validation:

**Locale**

Specifies the locale/language of the validation result messages

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -locale de
```

Locales can only be used if translations have been provided. See [https://github.com/hapifhir/org.hl7.fhir.core/tree/master/org.hl7.fhir.validation/src/main/resources](https://github.com/hapifhir/org.hl7.fhir.core/tree/master/org.hl7.fhir.validation/src/main/resources) for which translations are available. See also [How to add translations to the FHIR Validator](http://some-server/somewhere/Group/12345).

**Jurisdiction**

Specifies the jurisdiction to validate in

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -jurisdiction de
```

The jurisdiction is used internally in some profiles where country specific bindings are defined. The default jurisdiction is derived from the default Locale settings for the computer. If you want to specify No jurisdiction, this is functionally equivalent to a jurisdiction of the 'the whole world', which is -jurisdiction uv or -jurisdiction global.

**Terminology Server**

The validation engine uses a terminology server to validate codes from large external terminologies such as SNOMED CT, LOINC, RxNorm, etc. By default, the terminology server used is tx.fhir.org, which supports most of these terminologies. If you want to use another terminology server, you can specify one using the -tx parameter:

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -tx http://myserver/r3
```

If you nominate another terminology server, the following rules apply:

- it must support the same version as the -version parameter
- it must implement the $validate-code operation in the FHIR specification
- there must be no authentication
- There are multiple open source servers that support these requirements. The software that runs tx.fhir.org is available from [Health Intersections](http://some-server/somewhere/Group/12345).

Alternatively, you can run without any terminology support:

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -tx n/a
```

External codes are not validated when run like this.

**SNOMED CT**
You can specify which edition of SNOMED CT for the terminology server to use when doing SNOMED CT Validation using the -sct parameter

- `sct intl`

The valid choices are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>intl</td>
<td>International edition (900000000000207008)</td>
</tr>
<tr>
<td>us</td>
<td>US edition (731000124108)</td>
</tr>
<tr>
<td>uk</td>
<td>United Kingdom Edition (999000041000000102)</td>
</tr>
<tr>
<td>es</td>
<td>Spanish Language Edition (449081005)</td>
</tr>
<tr>
<td>nl</td>
<td>Netherlands Edition (11000146104)</td>
</tr>
<tr>
<td>ca</td>
<td>Canadian Edition (20611000087101)</td>
</tr>
<tr>
<td>dk</td>
<td>Danish Edition (554471000005108)</td>
</tr>
<tr>
<td>se</td>
<td>Swedish Edition (45991000052106)</td>
</tr>
<tr>
<td>au</td>
<td>Australian Edition (32506021000036107)</td>
</tr>
<tr>
<td>be</td>
<td>Belgium Edition (11000172109)</td>
</tr>
</tbody>
</table>

Notes:

- editions can only be supported if they are loaded / configured on the terminology server. (To add to this list, or ask for additional editions to be loaded on tx.fhir.org, ask the terminology stream on chat.fhir.org.

- If you’re loading implementation guides, and validating against them, and they specify value sets that bind to particular editions of SNOMED CT, the edition specified in this parameter will be ignored for those value sets.

### MustSupport

In some cases (e.g. when creating examples for implementation guides or when checking for potential interoperability issues with a new communication partner), it can be useful to know when data elements are present in an instance when those elements are not “mustSupport” in the profile(s) the instance is being validated against. Identifying situations where this occurs might drive a change to the profile or cause a designer to drop an element from the instance. In other cases, the presence of the element can be fine and the information message ignored.

To get the validator to flag such issues, invoke it with the parameter `-hintAboutNonMustSupport`

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -hintAboutNonMustSupport
```

### Extensions

The `-extension` parameter controls how extensions are validated by the validator. By default, unknown extensions are prohibited.

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -extension any
```

This allows all unknown extensions (the magic word 'any')

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -extension http://example.org/
```

This allows extensions from the specified domain (by matching the URL for the extension). This parameter can repeat any number of times

### Invariants

The `-want-invariants-in-messages` parameter controls whether the FHIRPath for invariants is included in the message. By default, FHIR Path is omitted - this is easier for end-users but sometimes developers wish for the full source of the invariant to be in the message.

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -want-invariants-in-messages
```

It’s also possible to tell the validator to ignore all invariants using the `-no-invariants` parameter

```java
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -no-invariants
```

By default, invariants are always run. The only reason to turn this off is for performance, if there’s no need to run invariants (e.g. when you are are in production, and weirdly confident)

### Questionnaires

By default, the validator will validate a QuestionnaireResponse resources against the specified questionnaire resource, if one is specified. The behavior of this can be controlled by the `-questionnaire` parameter:
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -questionnaire none

Possible values for this parameter:
- none: do not validate questionnaire responses against the matching questionnaire
- check: validate questionnaire responses against the matching questionnaire, if one is provided
- required: validate questionnaire responses against the matching questionnaire, and report it as an error if none is specified

Support added v5.1.16

Extensible binding warnings

When the validator encounters a code that is not part of an extensible binding, it adds a warning to suggest that the code be reviewed

java -jar validator_cli.jar [src] -no-extensible-binding-warnings

The validator can’t determine whether the meaning of the code makes it an inappropriate extension, or not; this requires human review. Hence, the warning. But the code may be valid - that’s why extensible is defined - so in some operational uses of the validator, it is appropriate to turn these warnings off

Level

Set the minimum level for validation messages

java -jar validator_cli.jar [src] -level warnings

Possible values:
- hints - report all hints, warnings and errors. (same as if not present)
- warnings - report all warnings and errors, but not hints
- errors - report all errors, but not warnings and hints

Best Practices

note: this parameter is not presently supported

There are a few contraints in the specification that are warnings but marked as 'best practice'. These are typically rules that the committees believe that should be followed, but cannot be due to legacy data constraints. by specifying the parameter -best-practice, these will be treated as errors not warnings. This parameter has no value

Language / Display

note: these parameters re not presently supported

The -lang parameter tells the vlaidator what language to default to if content has no specified language. The value is the same as for xml:lang. This is most useful when checking displays on coded values.

The -coding-display parameter controls to what degree code displays are checked. Possible value are Ignore, Check, CheckCaseAndSpace, CheckCase, CheckSpace

Native Validation

note: this parameter is not presently supported

By default, the validation engine only validates using the FHIR structures and profiles. The publication processes also generate a variety of xml, json and RDF schemas. You can ask the validator to validate against these as well using the native parameter:

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -native

Note that there is nothing in these schemas that is not validated directly by the engine itself anyway, so the main use for this is to see the kind of errors that would be reported from these schemas by other software.

Engines:
- xml: Xerces
- json: tba
- rdf: tba

Logging

To help with debugging validation problems, you can ask the validator to produce a log of terminology queries. You may be asked to turn this log on and share it if you report issues with the terminology validation
SecurityOnly

(this is not yet supported)

Unicode Control Chars

Unicode Control Characters can be a problem in any parsed text - see CVE-2021-42574. By default, the validator will produce a warning anytime it sees an unterminated control character in a string in XML or JSON. But you can make it produce an error for any bidi control character at all with this parameter:

- no_unicode_bidi_control_chars

Verbose Mode

When crumb-trails is set, the validator will create hints against the resources to explain which profiles it has validate the resource against, and why, and reasons for slice matching decisions

- verbose

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -verbose

This has an alis for legacy reasons:

- crumb-trails

Note that the validator knows why a particular resource fails to meet the definition of a slice. What it can't know is whether you expected it to or not, so verbose mode has a low signal to noise ratio, providing obvious reasons for resources don't match slices.

Show Validator Internal Times

When show-times is set, the validator will produce a line in the output summarising how long some internal processes took

- show-times

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -show-times

Validating IPS documents

FHIR IPS documents are normal bundles, and there's nothing special to validate them, but there's a syntactical

Validating CDA Documents

The FHIR Validator can validate CDA documents.

java -jar validator_cli.jar c:\temp\cda.xml -ig hl7.fhir.cda

This is not yet supported

Example URLs

Some of the examples in the FHIR specification have URLs in them that refer to example.org. By default, the validator will always mark any such references as an error, but this can be overridden with the parameter `-allow-example-urls`

Example:

java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -allow-example-urls true

Proxy Settings *(v5.2.1 onward)*

The FHIR Validator supports use with a proxy. For basic, no authentication, add the parameter `-proxy` followed by the `<url>:<port>`.

Example:

java -jar validator Cli.jar c:\temp\patient.xmL -version 3.0 -proxy 192.168.0.1:8080

If the proxy you are using requires basic authentication, also add the `-auth` parameter followed by the `<username>`:<password>`.
Example:

```
java -jar validator_cli.jar c:\temp\patient.xml -version 3.0 -proxy 192.168.0.1:8080 -auth username:password
```

**Legacy FHIR Path Setting**

As of v5.6.48, a bug was fixed in the FHIRPath engine used by the validator. This parameter allows to restore the behaviour that existed prior to the bug.

-implicit-fhirpath-string-conversions

For the FHIRPath functions `length()`, `lower()`, `upper()`, `contains()`, `indexOf`, `startsWith()`, `endsWith()`, `matches()`, `matchesFull()`, `replace()`, `replaceMatches()`, `toChars()`, the engine would automatically convert the value to a string. e.g. `Claim.created.length()=25` would return true or false. However the FHIRPath specification is clear that `.length()` should only work on a string, so that should return `{}` instead. This was corrected in 5.6.48 but that make break some existing usages, so this parameter can be used to restore the prior functionality.

**Other functionality provided by the Validator**

The validator also provides additional functionality beyond just validation. This section documents those various functions.

**Version Conversion**

The validator can convert between versions of FHIR (r2, r3, and r4). To use the validator to convert versions, provide the following parameters:

```
java -jar validator_cli.jar c:\temp\observation.xml -version 3.0 -to-version 4.0 -output c:\temp\observation4.json
```

The key parameter is "-to-version" which causes the validator to invoke the version conversion routine.

Technical notes:

- the validator can use either it's own internal routines, or the structure maps found at [https://github.com/FHIR/packages/tree/master/interversion](https://github.com/FHIR/packages/tree/master/interversion).
- By default, the internal routines will be used for resources with a canonical URL (e.g. code system etc) and the structure maps will be used otherwise.
- If the internal routines fail, the structure maps will be used anyway.
- you can use the parameter `-do-native` to get the validator to try the internal routines first for any resource, and the parameter `-no-native` to tell it not to try them at all
- Issues with the structure maps should be discussed on the chat.fhir.org implementers channel, or submitted as PRs against the github repo above.

**Comparing Profiles**

The validator can compare profiles. To compare profiles, use the following parameters:

```
java -jar validator_cli.jar -compare -dest /home/user/work/ig-comparison -version 4.0 -ig hl7.fhir.us.carin-bb#1.1.0 -ig hl7.fhir.us.davinci-crd#1.0.0

```

Lots of parameters! Explanation:

- `-compare`: tell the validator to run the comparison logic
- `-dest`: folder to produce the output. This must exist, and the validator will overwrite existing content if it needs to. The output isn't simple - see below
- `-version Maj.Min`: the version to use. You can leave this out and let the validator infer this, but it's there so that you can compare profiles across versions. E.g. if you specify `-version 4.0`, the profiles will both be treated as R4 profiles, even if they aren't.
- `-ig`: a repeating parameter that specifies the packages to load, that contain the profiles you want to compare
- `-left` and `-right`: the two profiles to compare. There's no functional difference between left and right, except that the comparison will keep to left and right consistently

The output starts at index.html. The output isn't simple (and may take some time to generate)

For each pair of profiles, the comparison generates a union and an intersections. The union is that total set of things that are allowed by either profile - that's what you could expect to read as a consumer of resources conforming to both profiles. The intersection is the set of things that both implementation guides allow - this is what you are required/allowed to write into a resource if you are creating one that must conform to both profiles. It's possible that the intersection will be empty - there's no valid instance that can conform to both profiles. In that case, consult the IG authors.

The comparison will (must) compare the sub-profiles that the profiles refer to. E.g. if both profiles on a resource such as MedicationAdministration refer to a profile of patient, those profiles will also be compared.

**Working with StructureMaps**
Running Automated Tests

The validator can be used to run all the JUnit tests for the FHIR core library against a set of test cases in a local directory. To do this, you must provide a specific parameter with a directory value:

```
java -Djava.locale.providers=COMPAT -jar validator_cli.jar -tests ./my/path/to/fhir-test-cases
```

The directory must follow the same conventions as the reference test cases found at [https://github.com/FHIR/fhir-test-cases](https://github.com/FHIR/fhir-test-cases). The tests which use these test cases are JUnit tests which can be found in the [https://github.com/hapifhir/org.hl7.fhir.core](https://github.com/hapifhir/org.hl7.fhir.core) project. Note that these tests must be executed with the Java property `java.locale.providers` set to `COMPAT`, which ensures that the Java consistently formats dates regardless of the Java version installed.

This parameter is compatible with `-txCache`, `-test-modules`, and `-test-classname-filter` parameters.

The following test-specific parameters can be used to limit which tests are run:

- `-test-modules [module-names]` A comma delimited list of Java module names for which to run JUnit tests. By default, all modules are used to run tests. Example: `-test-modules org.hl7.fhir.dstu2,org.hl7.fhir.dstu2016may`
- `-test-classname-filter [regex]` A regex filter applied to test Java class names or selecting which JUnit tests to run. By default, all tests are run. Example: `-test-classname-filter .*ShexGeneratorTests`