JSON Output

This chapter documents the JSON structures emitted by `rustc`. JSON may be enabled with the `--error-format=json` flag. Additional options may be specified with the `--json` flag which can change which messages are generated, and the format of the messages.

JSON messages are emitted one per line to stderr.

If parsing the output with Rust, the `cargo_metadata` crate provides some support for parsing the messages.

When parsing, care should be taken to be forwards-compatible with future changes to the format. Optional values may be `null`. New fields may be added. Enumerated fields like "level" or "suggestion_applicability" may add new values.

Diagnostics

Diagnostic messages provide errors or possible concerns generated during compilation. `rustc` provides detailed information about where the diagnostic originates, along with hints and suggestions.

Diagnostics are arranged in a parent/child relationship where the parent diagnostic value is the core of the diagnostic, and the attached children provide additional context, help, and information.

Diagnostics have the following format:

```
{
    /* The primary message. */
    "message": "unused variable: `x`",
    /* The diagnostic code.
       Some messages may set this value to null.
    */
    "code": {
        /* A unique string identifying which diagnostic triggered. */
        "code": "unused_variables",
        /* An optional string explaining more detail about the diagnostic code. */
        "explanation": null
    },
    /* The severity of the diagnostic.
       Values may be:
       - "error": A fatal error that prevents compilation.
       - "warning": A possible error or concern.
       - "note": Additional information or context about the diagnostic.
       - "help": A suggestion on how to resolve the diagnostic.
       - "failure-note": A note attached to the message for further information.
       - "error: internal compiler error": Indicates a bug within the compiler.
    */
    "level": "warning",
    /* An array of source code locations to point out specific details about
       where the diagnostic originates from. This may be empty, for example
       for some global messages, or child messages attached to a parent.

       Character offsets are offsets of Unicode Scalar Values.
    */
```

```
"spans": [
    {
        /* The file where the span is located.
           Note that this path may not exist. For example, if the path
           points to the standard library, and the rust src is not
           available in the sysroot, then it may point to a non-existent
           file. Beware that this may also point to the source of an
           external crate.
        */
        "file_name": "lib.rs",
        /* The byte offset where the span starts (0-based, inclusive). */
        "byte_start": 21,
        /* The byte offset where the span ends (0-based, exclusive). */
        "byte_end": 22,
        /* The first line number of the span (1-based, inclusive). */
        "line_start": 2,
        /* The last line number of the span (1-based, inclusive). */
        "line_end": 2,
        /* The first character offset of the line_start (1-based, inclusive). */
        "column_start": 9,
        /* The last character offset of the line_end (1-based, exclusive). */
        "column_end": 10,
        /* Whether or not this is the "primary" span.

           This indicates that this span is the focal point of the
           diagnostic.

           There are rare cases where multiple spans may be marked as
           primary. For example, "immutable borrow occurs here" and
           "mutable borrow ends here" can be two separate primary spans.

           The top (parent) message should always have at least one
           primary span, unless it has zero spans. Child messages may have
           zero or more primary spans.
        */
        "is_primary": true,
        /* An array of objects showing the original source code for this
           span. This shows the entire lines of text where the span is
           located. A span across multiple lines will have a separate
           value for each line.
        */
        "text": [
            {
                /* The entire line of the original source code. */
                "text": "    let x = 123;",
                /* The first character offset of the line of
                   where the span covers this line (1-based, inclusive). */
                "highlight_start": 9,
                /* The last character offset of the line of
                   where the span covers this line (1-based, exclusive). */
                "highlight_end": 10
            }
```

```
        ],
        /* An optional message to display at this span location.
           This is typically null for primary spans.
        */
        "label": null,
        /* An optional string of a suggested replacement for this span to
           solve the issue. Tools may try to replace the contents of the
           span with this text.
        */
        "suggested_replacement": null,
        /* An optional string that indicates the confidence of the
           "suggested_replacement". Tools may use this value to determine
           whether or not suggestions should be automatically applied.

           Possible values may be:
           - "MachineApplicable": The suggestion is definitely what the
             user intended. This suggestion should be automatically
             applied.
           - "MaybeIncorrect": The suggestion may be what the user
             intended, but it is uncertain. The suggestion should result
             in valid Rust code if it is applied.
           - "HasPlaceholders": The suggestion contains placeholders like
             `(...)`. The suggestion cannot be applied automatically
             because it will not result in valid Rust code. The user will
             need to fill in the placeholders.
           - "Unspecified": The applicability of the suggestion is unknown.
        */
        "suggestion_applicability": null,
        /* An optional object indicating the expansion of a macro within
           this span.

           If a message occurs within a macro invocation, this object will
           provide details of where within the macro expansion the message
           is located.
        */
        "expansion": {
            /* The span of the macro invocation.
               Uses the same span definition as the "spans" array.
            */
            "span": {/*...*/}
            /* Name of the macro, such as "foo!" or "#[derive(Eq)]". */
            "macro_decl_name": "some_macro!",
            /* Optional span where the relevant part of the macro is
              defined. */
            "def_site_span": {/*...*/},
        }
    }
],
/* Array of attached diagnostic messages.
   This is an array of objects using the same format as the parent
   message. Children are not nested (children do not themselves
   contain "children" definitions).
```

```
    */
    "children": [
        {
            "message": "`#[warn(unused_variables)]` on by default",
            "code": null,
            "level": "note",
            "spans": [],
            "children": [],
            "rendered": null
        },
        {
            "message": "if this is intentional, prefix it with an underscore",
            "code": null,
            "level": "help",
            "spans": [
                {
                    "file_name": "lib.rs",
                    "byte_start": 21,
                    "byte_end": 22,
                    "line_start": 2,
                    "line_end": 2,
                    "column_start": 9,
                    "column_end": 10,
                    "is_primary": true,
                    "text": [
                        {
                            "text": "    let x = 123;",
                            "highlight_start": 9,
                            "highlight_end": 10
                        }
                    ],
                    "label": null,
                    "suggested_replacement": "_x",
                    "suggestion_applicability": "MachineApplicable",
                    "expansion": null
                }
            ],
            "children": [],
            "rendered": null
        }
    ],
    /* Optional string of the rendered version of the diagnostic as displayed
       by rustc. Note that this may be influenced by the `--json` flag.
    */
    "rendered": "warning: unused variable: `x`\n --> lib.rs:2:9\n  |\n2 |     let x
= 123;\n  |         ^ help: if this is intentional, prefix it with an underscore:
`_x`\n  |\n  = note: `#[warn(unused_variables)]` on by default\n\n"
}
```


Artifact notifications

Artifact notifications are emitted when the `--json=artifacts` flag is used. They indicate that a file artifact has been saved to disk. More information about emit kinds may be found in the `--emit` flag documentation.

```
{
    /* The filename that was generated. */
    "artifact": "libfoo.rlib",
    /* The kind of artifact that was generated. Possible values:
       - "link": The generated crate as specified by the crate-type.
       - "dep-info": The `.d` file with dependency information in a Makefile-like
syntax.
       - "metadata": The Rust `.rmeta` file containing metadata about the crate.
       - "save-analysis": A JSON file emitted by the `-Zsave-analysis` feature.
    */
    "emit": "link"
}
```