The SourceKit Protocol

This documents the request/response API as it is currently implemented. For specific details related to Swift, see `SwiftSupport.md`.

The protocol is documented in the following format:

```
{
    <KEY>: (type) // comments
}
```

- `"{ }"` indicates a dictionary
- `"[ ]"` indicates an array.
- `"[opt]"` indicates an optional key.
- Specific UIDs are written as `<UID string>`.

Table of Contents

Requests

Code Completion

SourceKit is capable of providing code completion suggestions. To do so, it must be given either the path to a file (`key.sourcefile`), or some text (`key.sourcetext`). `key.sourcefile` is ignored when `key.sourcetext` is also provided.

Request Name	Request Key	Description
`codecomplete`	`codecomplete`	Returns a list of completions.

Request Name	Request Key	Description
open	codecomplete.open	Given a file will open a code-completion session which can be filtered using codecomplete.update. Each session must be closed using codecomplete.close.

Request

```
{
    <key.request>:          (UID) <source.request.codecomplete>
    [opt] <key.sourcetext>: (string)   // Source contents.
    [opt] <key.sourcefile>: (string)   // Absolute path to the file.
    <key.offset>:           (int64)    // Byte offset of code-completion point inside the so
    [opt] <key.compilerargs> [string*] // Array of zero or more strings for the compiler arg
                                       // e.g ["-sdk", "/path/to/sdk"]. If key.sourcefile is
                                       // these must include the path to that file.
    [opt] <key.not_recommended> [bool] // True if this result is to be avoided, e.g. because
                                       // the declaration is unavailable.
}
```

codecomplete.open

```
{
    <key.request>:          (UID) <source.request.codecomplete.open>
    [opt] <key.sourcetext>: (string)   // Source contents.
    [opt] <key.sourcefile>: (string)   // Absolute path to the file.
    <key.offset>:           (int64)    // Byte offset of code-completion point inside the so
    [opt] <key.codecomplete.options>:    (dict)     // An options dictionary containing keys
    [opt] <key.compilerargs> [string*] // Array of zero or more strings for the compiler arg
                                       // e.g ["-sdk", "/path/to/sdk"]. If key.sourcefile is
                                       // these must include the path to that file.
    [opt] <key.not_recommended> [bool] // True if this result is to be avoided, e.g. because
                                       // the declaration is unavailable.
}
```

Response

```
{
    <key.results>: (array) [completion-result*]  // array of zero or more completion-result
}
```

completion-result ::=

```
{
  <key.description>:    (string)    // Text to be displayed in code-completion window.
  <key.kind>:           (UID)       // UID for the declaration kind (function, class, etc.)
  <key.sourcetext>:     (string)    // Text to be inserted in source.
  <key.typename>:       (string)    // Text describing the type of the result.
  <key.doc.brief>:      (string)    // Brief documentation comment attached to the entity.
  <key.context>:        (UID)       // Semantic context of the code completion result.
  <key.num_bytes_to_erase>: (int64) // Number of bytes to the left of the cursor that should
}

completion.open-result ::=
{
  <key.kind>:           (UID)         // UID for the declaration kind (function, class, etc.
  <key.name>:           (string)      // Name of the word being completed
  <key.sourcetext>:     (string)      // Text to be inserted in source.
  <key.description>:    (string)      // Text to be displayed in code-completion window.
  <key.typename>:       (string)      // Text describing the type of the result.
  <key.context>:        (UID)         // Semantic context of the code completion result.
  <key.num_bytes_to_erase>: (int64)   // Number of bytes to the left of the cursor that shou
  <key.substructure>:   (dictionary)  // Contains an array of dictionaries representing rang
      - <key.nameoffset>  (int64)     // The offset location of the given parameter
      - <key.namelength>  (int64)     // The length of the given parameter
      - <key.bodyoffset>  (int64)     // The `nameoffset` + the indentation inside the body
      - <key.bodylength>  (int64)     // The `namelength` + the indentation inside the body
}
```

Testing

```
$ sourcekitd-test -req=complete -offset=<offset> <file> [-- <compiler args>]
```

For example, to get a code completion suggestion for the 58th character in an ASCII file at `/path/to/file.swift`:

```
$ sourcekitd-test -req=complete -offset=58 /path/to/file.swift -- /path/to/file.swift
```

You could also issue the following request in the `sourcekitd-repl`:

```
$ sourcekitd-repl
Welcome to SourceKit.  Type ':help' for assistance.
(SourceKit) {
    key.request: source.request.codecomplete,
    key.sourcefile: "/path/to/file.swift",
    key.offset: 57,
    key.compilerargs: ["/path/to/file.swift"]
}
```

Indexing

SourceKit is capable of "indexing" source code, responding with which ranges of text contain what kinds of source code. For example, SourceKit is capable of telling you that "the source code on line 2, column 9, is a reference to a struct".

To index source code, SourceKit must be given either the path to a file (`key.sourcefile`), or some text (`key.sourcetext`). `key.sourcefile` is ignored when `key.sourcetext` is also provided.

A hash (`key.hash`) may be provided in order to determine whether the source code has changed since the last time it was indexed. If the provided hash matches the one generated from the source code, the response will omit entries that have already been returned.

Request

```
{
    <key.request>:           (UID) <source.request.indexsource>
    [opt] <key.sourcetext>: (string)   // Source contents.
    [opt] <key.sourcefile>: (string)   // Absolute path to the file.
    [opt] <key.compilerargs> [string*] // Array of zero or more strings for the compiler arg
                                       // e.g ["-sdk", "/path/to/sdk"]. If key.sourcefile is
                                       // these must include the path to that file.
    [opt] <key.hash>: (string)         // Known hash for the indexed file, used to determine
                                       // the file has changed since the last time it was in
}
```

Response

```
{
    <key.dependencies>: (array) [dependency*] // Array of zero or more dependencies.
    <key.hash>:     (string)                  // Hash associated with the indexed file.
    [opt] <key.entities>: (array) [entity*]   // Array of zero or more top-level indexed ent
                                              // If the key.hash provided in the request mat
                                              // one in the response, this key will not be i
                                              // the response.
}

entity ::=
{
    <key.kind>:                   (UID)            // UID for the declaration or reference
    <key.name>:                   (string)         // Displayed name for the entity.
    <key.usr>:                    (string)         // USR string for the entity.
    <key.line>:                   (int64)          // Line of the position of the entity in
    <key.column>:                 (int64)          // Column of the position of the entity
    [opt] <key.is_test_candidate> (bool)           // Whether the instance method matches w
                                                   // to be a viable test: a class instance
```

```
                                                        // parameters, returns void, and begins
                                                        // is only present if the value is true.
    [opt] <key.entities>:         (array) [entity+] // One or more entities contained in the
    [opt] <key.related>:          (array) [entity+] // One or more entities related with the
}

dependency ::=
{
    <key.kind>:        (UID)    // UID for the kind (import of a swift module, etc.).
    <key.name>:        (string) // Displayed name for dependency.
    <key.filepath>:    (string) // Path to the file.
    [opt] <key.hash>:  (string) // Hash associated with this dependency.
}
```

Testing

```
$ sourcekitd-test -req=index <file> [-- <compiler args>]
```

For example, to index a file at `/path/to/file.swift`:

```
$ sourcekitd-test -req=index /path/to/file.swift -- /path/to/file.swift
```

You could also issue the following request in the `sourcekitd-repl`:

```
$ sourcekitd-repl
Welcome to SourceKit.  Type ':help' for assistance.
(SourceKit) {
    key.request: source.request.index,
    key.sourcefile: "/path/to/file.swift",
    key.compilerargs: ["/path/to/file.swift"]
}
```

Documentation

SourceKit is capable of gathering symbols and their documentation, either from Swift source code or from a Swift module. SourceKit returns a list of symbols and, if they are documented, the documentation for those symbols.

To gather documentation, SourceKit must be given either the name of a module (`key.modulename`), the path to a file (`key.sourcefile`), or some text (`key.sourcetext`). `key.sourcefile` is ignored when `key.sourcetext` is also provided, and both of those keys are ignored if `key.modulename` is provided.

Request

```
{
    <key.request>:          (UID) <source.request.docinfo>
    [opt] <key.modulename>: (string)   // The name of the Swift module.
    [opt] <key.sourcetext>: (string)   // Source contents.
```

```
    [opt] <key.sourcefile>: (string)   // Absolute path to the file.
    [opt] <key.compilerargs> [string*] // Array of zero or more strings for the compiler arg
                                       // e.g ["-sdk", "/path/to/sdk"]. If key.sourcefile is
                                       // these must include the path to that file.
}
```

Response

```
{
    <key.sourcetext>:        (string)              // Source contents.
    <key.annotations>:       (array) [annotation*] // An array of annotations for the tokens
                                                   // source text, they refer to the text vi
                                                   // entries. This includes syntactic annot
                                                   // keywords) and semantic ones. The semar
                                                   // the name and USR of the referenced sym
    [opt] <key.entities>:    (array) [entity*]     // A structure of the symbols, similar to
                                                   // request returns (a class has its metho
                                                   // etc.). This includes the function para
                                                   // types as entities. Each entity refers
                                                   // original text via offset + length entr
    [opt] <key.diagnostics>: (array) [diagnostic*] // Compiler diagnostics emitted during pa
                                                   // This key is only present if a diagnost
                                                   // the length of the array is non-zero).
}

annotation ::=
{
    <key.kind>:   (UID)   // UID for the declaration kind (function, class, etc.).
    <key.offset>: (int64) // Location of the annotated token.
    <key.length>: (int64) // Length of the annotated token.
}

entity ::=
{
    <key.kind>:                   (UID)             // UID for the declaration or reference k
    <key.name>:                   (string)          // Displayed name for the entity.
    <key.usr>:                    (string)          // USR string for the entity.
    <key.offset>:                 (int64)           // Location of the entity.
    <key.length>:                 (int64)           // Length of the entity.
    <key.fully_annotated_decl>:   (string)          // XML representing the entity, its USR,
    [opt] <key.doc.full_as_xml>: (string)           // XML representing the entity and its do
                                                    // when the entity is documented.
    [opt] <key.entities>:         (array) [entity+] // One or more entities contained in the
}

diagnostic ::=
{
```

```
    <key.id>:               (string)       // The internal ID of the diagnostic.
    <key.line>:             (int64)        // The line upon which the diagnostic was emitte
    <key.column>:           (int64)        // The column upon which the diagnostic was emitt
    <key.filepath>:         (string)       // The absolute path to the file that was being p
                                           // when the diagnostic was emitted.
    <key.severity>:         (UID)          // The severity of the diagnostic. Can be one of
                                           //   - source.diagnostic.severity.note
                                           //   - source.diagnostic.severity.warning
                                           //   - source.diagnostic.severity.error
    <key.description>:      (string)       // A description of the diagnostic.
    [opt] <key.categories>: (array) [UID*] // The categories of the diagnostic. Can be:
                                           //   - source.diagnostic.category.deprecation
                                           //   - source.diagnostic.category.no_usage
}
```

Testing

```
$ sourcekitd-test -req=doc-info <file> [-- <compiler args>]
```

For example, to gather documentation info for a file at `/path/to/file.swift`:

```
$ sourcekitd-test -req=doc-info /path/to/file.swift -- /path/to/file.swift
```

You could also issue the following request in the `sourcekitd-repl` to gather all the documentation info for Foundation (careful, it's a lot!):

```
$ sourcekitd-repl
Welcome to SourceKit.  Type ':help' for assistance.
(SourceKit) {
    key.request: source.request.docinfo,
    key.modulename: "Foundation",
    key.compilerargs: ["-sdk", "/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.
}
```

Module interface generation

Request

```
{
    <key.request>:          (UID) <source.request.editor.open.interface>
    <key.name>:             (string) // virtual name/path to associate with the interface do
    <key.modulename>:       (string) // Full module name, e.g. "Foundation.NSArray"
    [opt] <key.compilerargs> [string*] // array of zero or more strings for the compiler arg
                                       // e.g ["-sdk", "/path/to/sdk"]
}
```

Response

This will return the Swift interface of the specified module.

- `key.sourcetext`: The pretty-printed module interface in swift source code
- `key.syntaxmap`: An array of syntactic annotations, same as the one returned for the source.request.editor.open request.
- `key.annotations`: An array of semantic annotations, same as the one returned for the source.request.editor.open request.

All SourceKit requests that don't modify the source buffer should work on the opened document, by passing the associated 'name' for the document.

If pointing at a symbol which came from a clang module or the stdlib, then the response for the cursor-info request will have an entry for the module name:

```
key.modulename: "<module-name>"
```

Also if there is already a generated-interface document for this module previously opened, there will be an entry with the "virtual name" associated with this document (from the previous 'editor.open.interface' request):

```
key.module_interface_name: "<virtual name for interface document>"
```

After 'opening' the module interface, to 'jump' to the location of a declaration with a particular USR, use the 'find_usr' request:

```
{
    <key.request>:       (UID) <source.request.editor.find_usr>
    <key.usr>:           (string) // USR to look for.
    <key.sourcefile>:    (string) // virtual name/path associated with the interface documen
}
```

This returns the byte offset if the USR is found, or an empty response otherwise:

```
key.offset: <byte offset in the interface source>
```

Diagnostics

Diagnostic entries occur as part of the responses for editor requests. If there is a diagnostic, `<key.diagnostics>` is present and contains an array of diagnostic entries. A diagnostic entry has this format:

```
{
    <key.severity>:          (UID)   // severity of error
    <key.offset>:            (int64) // error location
    <key.description>:       (string) // error description
    [opts] <key.fixits>:     (array) [fixit+] // one or more entries for fixits
    [opts] <key.ranges>:     (array) [range+] // one or more entries for ranges
    [opts] <key.diagnostics>: (array) [diagnostic+] // one or more sub-diagnostic entries
    [opts] <key.educational_note_paths>: (array) [string+] // one or more absolute paths of
}
```

Where `key.severity` can be one of:

- source.diagnostic.severity.note
- source.diagnostic.severity.warning
- source.diagnostic.severity.error

```
fixit ::=
{
    <key.offset>:        (int64) // location of the fixit range
    <key.length>:        (int64) // length of the fixit range
    <key.sourcetext>:    (string) // text to replace the range with
}
```

```
range ::=
{
    <key.offset>:        (int64) // location of the range
    <key.length>:        (int64) // length of the range
}
```

Sub-diagnostics are only diagnostic notes currently.

Demangling

SourceKit is capable of "demangling" mangled Swift symbols. In other words, it's capable of taking the symbol `_TF13MyCoolPackageg6raichuVS_7Pokemon` as input, and returning the human-readable `MyCoolPackage.raichu.getter : MyCoolPackage.Pokemon`.

Request

```
{
    <key.request>: (UID) <source.request.demangle>,
    <key.names>:   [string*] // An array of names to demangle.
}
```

Response

```
{
    <key.results>: (array) [demangle-result+] // The results for each
                                              // demangling, in the order in
                                              // which they were requested.
}
```

```
demangle-result ::=
{
    <key.name>: (string) // The demangled name.
}
```

Testing

```
$ sourcekitd-test -req=demangle [<names>]
```

For example, to demangle the symbol `_TF13MyCoolPackageg6raichuVS_7Pokemon`:

```
$ sourcekitd-test -req=demangle _TF13MyCoolPackageg6raichuVS_7Pokemon
```

Note that when using `sourcekitd-test`, the output is output in an ad hoc text format, not JSON.

You could also issue the following request in the `sourcekitd-repl`, which produces JSON:

```
$ sourcekitd-repl
Welcome to SourceKit.  Type ':help' for assistance.
(SourceKit) {
    key.request: source.request.demangle,
    key.names: [
      "_TF13MyCoolPackageg6raichuVS_7Pokemon"
    ]
}
```

Simple Class Mangling

SourceKit is capable of "mangling" Swift class names. In other words, it's capable of taking the human-readable `UIKit.ViewController` as input and returning the symbol `_TtC5UIKit14ViewController`.

Request

```
{
    <key.request>: (UID) <source.request.mangle_simple_class>,
    <key.names>:   [mangle-request*] // An array of requests to mangle.
}

mangle-request ::=
{
    <key.modulename>: (string)  // The Swift module name
    <key.name>: (string)        // The class name
}
```

Response

```
{
    <key.results>: (array) [mangle-result+]   // The results for each
                                              // mangling, in the order in
                                              // which they were requested.
}

mangle-result ::=
{
    <key.name>: (string) // The mangled name.
```

```
}
```

Testing

```
$ sourcekitd-test -req=mangle [<names>]
```

For example, to mangle the name `UIKit.ViewController`:

```
$ sourcekitd-test -req=mangle UIKit.ViewController
```

Note that when using `sourcekitd-test`, the output is output in an ad hoc text format, not JSON.

You could also issue the following request in the `sourcekitd-repl`, which produces JSON:

```
$ sourcekitd-repl
Welcome to SourceKit.  Type ':help' for assistance.
(SourceKit) {
    key.request: source.request.mangle_simple_class,
    key.names: [
      {
          key.modulename: "UIKit",
          key.name: "ViewController"
      }
    ]
}
```

Protocol Version

SourceKit can provide information about the version of the protocol that is being used.

Request

```
{
    <key.request>: (UID) <source.request.protocol_version>
}
```

Response

```
{
    <key.version_major>: (int64) // The major version number in a version string
    <key.version_minor>: (int64) // The minor version number in a version string
}
```

Testing

```
$ sourcekitd-test -req=version
```

or

```
$ sourcekitd-repl
Welcome to SourceKit.  Type ':help' for assistance.
(SourceKit) {
    key.request: source.request.protocol_version
}
```

Compiler Version

SourceKit can provide information about the version of the compiler version that is being used.

Request

```
{
    <key.request>: (UID) <source.request.compiler_version>
}
```

Response

```
{
    <key.version_major>: (int64) // The major version number in a version string
    <key.version_minor>: (int64) // The minor version number in a version string
    <key.version_patch>: (int64) // The patch version number in a version string
}
```

Testing

```
$ sourcekitd-test -req=compiler-version
```

or

```
$ sourcekitd-repl
Welcome to SourceKit.  Type ':help' for assistance.
(SourceKit) {
    key.request: source.request.compiler_version
}
```

Cursor Info

SourceKit is capable of providing information about a specific symbol at a specific cursor, or offset, position in a document.

To gather documentation, SourceKit must be given either the name of a module (key.modulename), the path to a file (key.sourcefile), or some text (key.sourcetext). key.sourcefile is ignored when key.sourcetext is also provided, and both of those keys are ignored if key.modulename is provided.

Request

```
{
    <key.request>:            (UID)     <source.request.cursorinfo>,
    [opt] <key.sourcetext>:   (string)  // Source contents.
    [opt] <key.sourcefile>:   (string)  // Absolute path to the file.
                                        // **Require**: key.sourcetext or key.sourcefile
    [opt] <key.offset>:       (int64)   // Byte offset of code point inside the source conte
    [opt] <key.usr>:          (string)  // USR string for the entity.
                                        // **Require**: key.offset or key.usr
    [opt] <key.compilerargs>: [string*] // Array of zero or more strings for the compiler ar
                                        // e.g ["-sdk", "/path/to/sdk"]. If key.sourcefile i
                                        // these must include the path to that file.
    [opt] <key.cancel_on_subsequent_request>: (int64) // Whether this request should be canc
                                        // new cursor-info request is made that uses the sam
                                        // This behavior is a workaround for not having firs
                                        // cancelation. For backwards compatibility, the def
}
```

Response

```
{
    <key.kind>:                  (UID)    // UID for the declaration or reference kind (func
    <key.name>:                  (string) // Displayed name for the token.
    <key.usr>:                   (string) // USR string for the token.
    <key.filepath>:              (string) // Path to the file.
    <key.offset>:                (int64)  // Byte offset of the token inside the source cont
    <key.length>:                (ist64)  // Length of the token.
    <key.typename>:              (string) // Text describing the type of the result.
    <key.annotated_decl>:        (string) // XML representing how the token was declared.
    <key.fully_annotated_decl>:  (string) // XML representing the token.
    [opt] <key.doc.full_as_xml>: (string) // XML representing the token and its documentatio
    <key.typeusr>:               (string) // USR string for the type.
}
```

Testing

```
$ sourcekitd-test -req=cursor -offset=<offset> <file> [-- <compiler args>]
$ sourcekitd-test -req=cursor -pos=<line>:<column> <file> [-- <compiler args>]
```

For example, using a document containing:

```
struct Foo {
    let bar: String
}
```

To get the information about the type `Foo` you would make one of the following requests:

```
$ sourcekitd-test -req=cursor -offset=7 /path/to/file.swift -- /path/to/file.swift
$ sourcekitd-test -req=cursor -pos=1:8 /path/to/file.swift -- /path/to/file.swift
```

Note that when using `sourcekitd-test`, the output is output in an ad hoc text format, not JSON.

You could also issue the following request in the `sourcekitd-repl`, which produces JSON:

```
$ sourcekitd-repl
Welcome to SourceKit.  Type ':help' for assistance.
(SourceKit) {
  key.request: source.request.cursorinfo,
  key.sourcefile: "/path/to/file.swift",
  key.offset: 7,
  key.compilerargs: ["/path/to/file.swift"]
}
```

Expression Type

This request collects the types of all expressions in a source file after type checking. To fulfill this task, the client must provide the path to the Swift source file under type checking and the necessary compiler arguments to help resolve all dependencies.

Request

```
{
    <key.request>:            (UID)     <source.request.expression.type>,
    <key.sourcefile>:         (string)  // Absolute path to the file.
    <key.compilerargs>:       [string*] // Array of zero or more strings for the compiler ar
                                        // e.g ["-sdk", "/path/to/sdk"]. If key.sourcefile i
                                        // these must include the path to that file.
    <key.expectedtypes>:      [string*] // A list of interested protocol USRs.
                                        // When empty, we report all expressions in the file
                                        // When non-empty, we report expressions whose types
}
```

Response

```
{
    <key.expression_type_list>:       (array) [expr-type-info*]   // A list of expression ar
}

expr-type-info ::=
{
  <key.expression_offset>:    (int64)    // Offset of an expression in the source file
  <key.expression_length>:    (int64)    // Length of an expression in the source file
```

```
  <key.expression_type>:      (string)   // Printed type of this expression
  <key.expectedtypes>:        [string*]  // A list of interested protocol USRs this expressi
}
```

Testing

```
$ sourcekitd-test -req=collect-type /path/to/file.swift -- /path/to/file.swift
```

Variable Type

This request collects the types of all variable declarations in a source file after type checking. To fulfill this task, the client must provide the path to the Swift source file under type checking and the necessary compiler arguments to help resolve all dependencies.

Request

```
{
    <key.request>:            (UID)     <source.request.variable.type>,
    <key.sourcefile>:         (string)  // Absolute path to the file.
    <key.compilerargs>:       [string*] // Array of zero or more strings for the compiler ar
                                        // e.g ["-sdk", "/path/to/sdk"]. If key.sourcefile i
                                        // these must include the path to that file.
    [opt] <key.offset>:       (int64)   // Offset of the requested range. Defaults to zero.
    [opt] <key.length>:       (int64)   // Length of the requested range. Defaults to the en
}
```

Response

```
{
    <key.variable_type_list>: (array) [var-type-info*]   // A list of variable declarations
}

var-type-info ::=
{
    <key.variable_offset>:       (int64)    // Offset of a variable identifier in the source
    <key.variable_length>:       (int64)    // Length of a variable identifier an expression
    <key.variable_type>:         (string)   // Printed type of the variable declaration
    <key.variable_type_explicit> (bool)     // Whether the declaration has an explicit type
}
```

Testing

```
$ sourcekitd-test -req=collect-var-type /path/to/file.swift -- /path/to/file.swift
```

UIDs

Keys

- key.column
- key.compilerargs
- key.description
- key.kind
- key.line
- key.name
- key.offset
- key.results
- key.request
- key.sourcefile
- key.sourcetext
- key.typename
- key.usr
- key.version_major
- key.version_minor
- key.annotated_decl
- key.fully_annotated_decl
- key.doc.full_as_xml
- key.typeusr