ENSIME User Manual

Aemon Cannon

June 21, 2014

Last Updated: 06/21/2014

Contents

1 Introduction

1.1 What is ENSIME?

ENSIME is the ENhanced Scala Interaction Mode for Emacs. It provides many features that are commonly found only in IDEs, such as live error-checking, symbol inspection, package/type browsing, and basic refactorings. ENSIME's architecture is client/server: a thin Emacs client communicates with an analysis server written in Scala. The client component is based heavily on the SLIME environment for Common Lisp - using the same Swank RPC protocol. The server component services these calls using an instance of the official Scala compiler, so the results should always be consistent with the commandline compiler.

2 Installation

2.1 System Requirements

- Emacs 22 or later (23 is recommended)
- Linux, Mac OSX, Windows
- JVM Version 6
- Scala 2.8.x or 2.9.x compatible project

2.2 Getting Started

scala-mode:
Although it's not required, ENSIME is designed to compliment an existing scala major mode. scala-mode2 is an excellent scala mode, and can be found at https://github.com/hvesalai/scala-mode2

ensime-mode:
Download the latest ENSIME distribution from github.com at

`http://github.com/aemoncannon/ensime/downloads`

Make sure you get the version that is appropriate for your Scala version. Unpack the ENSIME distribution into a directory of your choosing, and add the following lines to your .emacs file:

```
;; Load the ensime lisp code...
(add-to-list 'load-path "ENSIME_ROOT/elisp/")
(require 'ensime)

;; This step causes the ensime-mode to be started whenever
;; scala-mode is started for a buffer. You may have to customize this step
;; if you're not using the standard scala mode.
```

```
(add-hook 'scala-mode-hook 'ensime-scala-mode-hook)
```

Create a .ensime config file:
See section 3.1.

Finally...

Execute *M-x ensime* from Emacs. Follow the instructions in the minibuffer to select your project. If you encounter any problems, see section 5 for troubleshooting.

3 Creating a Project

3.1 Generating a Config File

3.1.1 Using sbt to Generate a Config File

First, you need to install the ENSIME sbt plugin. Add the following lines to your project/plugins.sbt:

```
addSbtPlugin("org.ensime" % "ensime-sbt-cmd" % "VERSION")
```

Replace VERSION with the latest version of the plugin, available at https://github.com/aemoncannon/ensime-sbt-cmd. Then, from an sbt shell, generate your ENSIME project:

```
ensime generate
```

You should now have a .ensime file in the root of your project. Instead of editing this file directly, if you need to adjust ENSIME configuration settings, you can change the value of the sbt setting *ensime-config*. The value should be a lisp S-Expression, just like the .ensime configuration format. The ENSIME sbt plugin includes some helpers for building S-Expressions. For example, your Build.scala might include the following:

```
import org.ensime.sbt.Plugin.Settings.ensimeConfig
import org.ensime.sbt.util.SExp._

ensimeConfig := sexp(
  key(":compiler-args"), sexp("-Ywarn-dead-code", "-Ywarn-shadowing"),
  key(":formatting-prefs"), sexp(
    key(":alignParameters"), true
  )
)
```

If you have multiple sbt subprojects, they may each specify different ensime-config values.

3.1.2 Generating Configs for Other Project Types

ENSIME includes a wizard for automatically generating configuration files. In Emacs, execute *M-x ensime-config-gen*. Then simply follow the directions in the mini-buffer to create a .ensime file for your project. ENSIME will try to guess the type(mvn, custom, etc) of your project, based on the files and directory structure. If the config generator does a poor job for your project, please let us know so we can improve it. And of course you can still create the .ensime file for your project manually. See the section on the .ensime format below.

3.2 Notes on Specific Project Types

3.2.1 SBT

Inferior SBT:
The keystrokes *C-c C-v s* will launch (or switch to an existing) inferior sbt session.

Compile-on-Save:
If the value of the Emacs-Lisp variablt ensime-sbt-compile-on-save is non-nil ENSIME will invoke the 'compile' task in the inferior sbt process(presuming you have one running) whenever you save a Scala buffer. This option is enabled by default.

3.2.2 Other

See section 3.3 for how to specify dependency, source, and class-output locations.

3.3 Config File Format

Each project *must* have a .ensime file. The .ensime file contains the configuration for your project, and must be located in your project's root directory.The contents of the file must be a valid Emacs-Lisp S-Expression. Here's a quick primer on ELisp values.

"..."	A String
t	True
nil	False, null, or opposite of t.
(...)	A literal list.
:abcd123	A keyword
(:key1 val1 :key2 val2)	An indexed property-list.

What follows is a description of all available configuration options. Required

options are marked as 'Required'. Any filename or directory paths may be relative to the project root.

:root-dir

The root directory of your project. This option should be filled in by your editor.

Arguments:

```
String: a filename
```

:name

The short identifier for your project. Should be the same that you use when publishing. Will be displayed in the Emacs mode-line when connected to an ENSIME server.

Arguments:

```
String: name
```

:package

An optional 'primary' package for your project. Used by ENSIME to populate the project outline view.

Arguments:

```
String: package name
```

:module-name

The canonical module-name of this project.

Arguments:

```
String: name
```

:active-subproject

The module-name of the subproject which is currently selected.

Arguments:

```
String: module name
```

:depends-on-modules

A list of module-names on which this project depends.

Arguments:

```
List of Strings: module names
```

:version

The current, working version of your project.

Arguments:

```
String: version number
```

:compile-deps

A list of jar files and class directories to include on the compilation classpath. No recursive expansion will be done.

Arguments:

```
List of Strings: file and directory names
```

:compile-jars

A list of jar files and directories to search for jar files to include on the compilation classpath. Directories will be searched recursively.

Arguments:

```
List of Strings: file and directory names
```

:runtime-deps

A list of jar files and class directories to include on the runtime classpath. No recursive expansion will be done.

Arguments:

```
List of Strings: file and directory names
```

:runtime-jars

A list of jar files and directories to search for jar files to include on the runtime classpath. Directories will be searched recursively.

Arguments:

```
List of Strings: file and directory names
```

:test-deps

A list of jar files and class directories to include on the test classpath. No recursive expansion will be done.

Arguments:

```
List of Strings: file and directory names
```

:source-roots

A list of directories in which to start searching for source files.

Arguments:

```
List of Strings: directory names
```

:reference-source-roots

A list of files or directories in which to start searching for reference sources. Generally these are the sources that correspond to library dependencies.

Arguments:

```
List of Strings: a combination of directory names or .jar or .zip
  file names
```

:target

The root of the class output directory.

Arguments:

```
String: directory
```

:target

The root of the test class output directory.

Arguments:

```
String: directory
```

:disable-index-on-startup

Disable the classpath indexing process that happens at startup. This will speed up the loading process significantly, at the cost of breaking some functionality.

Arguments:

```
Boolean: t or nil
```

:disable-source-load-on-startup

Disable the parsing and reloading of all sources that normally occurs on startup.

Arguments:

```
Boolean: t or nil
```

:disable-scala-jars-on-classpath

Disable putting standard Scala jars (i.e. scala-library.jar, scala-reflect.jar and scala-compiler) on the classpath. Useful for compiling against custom Scala builds and for development of Scala compiler.

Arguments:

```
Boolean: t or nil
```

:only-include-in-index

Only classes that match one of the given regular expressions will be added to the index. If this is omitted, all classes will be added. This can be used to reduce memory usage and speed up loading. For example:

```
:only-include-in-index ("my\\.project\\.packages\\.\*" "important\\.dependency\\..\*"
```

This option can be used in conjunction with 'exclude-from-index' - the result when both are given is that the exclusion expressions are applied to the names that pass the inclusion filter.

Arguments:

```
List of Strings: regular expresions
```

:exclude-from-index

Classes that match one of the given regular expressions will not be added to the index. This can be used to reduce memory usage and speed up loading. For example:

```
:exclude-from-index ("com\\.sun\\..\*" "com\\.apple\\..\*")
```

This option can be used in conjunction with 'only-include-in-index' - the result when both are given is that the exclusion expressions are applied to the names that pass the inclusion filter.

Arguments:

```
List of Strings: regular expresions
```

:compiler-args

Specify arguments that should be passed to ENSIME's internal presentation compiler. Warning: the presentation compiler understands a subset of the batch compiler's arguments.

Arguments:

```
List of Strings: arguments
```

:builder-args

Specify arguments that should be passed to ENSIME's internal incremental compiler.

Arguments:

```
List of Strings: arguments
```

:java-compiler-args

Specify arguments that should be passed to ENSIME's internal JDT java compiler. Arguments are passed as a list of strings, with each pair being a key, value drawn from org.eclipse.jdt.internal.compiler.impl.CompilerOptions

Arguments:

```
List of Strings: arguments
```

:java-compiler-version

Specify version of java compiler to use (must be supported by internal JDT).

Arguments:

```
String: version
```

:formatting-prefs

Customize the behavior of the source formatter. All Scalariform preferences are supported:

:alignParameters	t or nil
:alignSingleLineCaseStatements	t or nil
:alignSingleLineCaseStatements_maxArrowIndent	1-100
:compactStringConcatenation	t or nil
:doubleIndentClassDeclaration	t or nil
:indentLocalDefs	t or nil
:indentPackageBlocks	t or nil
:indentSpaces	1-10
:indentWithTabs	t or nil
:multilineScaladocCommentsStartOnFirstLine	t or nil
:preserveDanglingCloseParenthesis	t or nil
:preserveSpaceBeforeArguments	t or nil
:rewriteArrowSymbols	t or nil
:spaceBeforeColon	t or nil
:spaceInsideBrackets	t or nil
:spaceInsideParentheses	t or nil
:spacesWithinPatternBinders	t or nil

Arguments:

`List of keyword, string pairs: preferences`

4 Usage

4.1 Startup

To start ensime type *M-x ensime*. You only need to do this once per project session. Follow the minibuffer instructions to specify the location of your .ensime project file. Bear in mind that the server may take several seconds to finish loading and analyzing your project's sources. To watch the progress of the ENSIME startup, switch to the **inferior-ensime-server** buffer.

4.2 Error Highlighting

Ensime will highlight errors and warnings in source files through the use of the Scala presentation compiler, a lightweight version of the Scala compiler. This is triggered in several ways:

- when you save a file
- when you type *C-c C-v c* or *C-c C-v a*

- after a short pause in typing. The frequency of these checks is controlled through the variables ensime-typecheck-idle-interval and ensime-typecheck-interval. This feature can be disabled by setting ensime-typecheck-when-idle to nil.

4.3 Symbol and Member Completion

ENSIME completion is initiated by pressing the *TAB* key. To complete a symbol, type the first couple characters, then press *TAB*. Currently this works for local variables, method parameters, unqualified method names, and type names. To complete a type member, type '.' or *SPACE* followed by *TAB*.

Completion menu key commands:
Candidates can be scrolled with *M-n* and *M-p* or *UP* and *DOWN*. Candidates can be searched by typing *C-s*. Press *TAB* again to complete a common prefix. To cancel completion, type *C-g*. Finally, if you've selected the completion you want, press *ENTER*. If the selected completion was a method name, the minibuffer will display help for the method parameters.

4.4 Type/Package Inspector

Invocation:
Control+Right-Clicking on a symbol in a Scala buffer, or typing *C-c C-v i* while the point is over a symbol will launch the type inspector. Typing *C-c C-v o* will open the inspector on the current project's main package. *C-c C-v p* will inspect the package of the current source file. Use the command *M-x ensime-inspect-by-path* to inspect an arbitrary type or package.

Package Inspector:
Displays a hierarchical view of a package, including all top-level types. Select a type to open the Type Inspector.

Type Inspector:
Lists the interfaces that contribute members to the inspected type. List each interface's methods, with full type signatures. If the type has a companion object/class, a link to the companion will appear under the heading.

Navigation:
Most things in the inspector are hyper-linked. You can click these links with the mouse or position your cursor over them and press ENTER. A history is kept of all the pages you view in the inspector. Go back in this history by typing ',' and forward by typing '.'.

4.5 The Scala REPL

First, ensure that you've set the :target directive in your config file. The REPL will load your project classes from the :target directory. Then, type *C-c C-v z* to launch the embedded Scala REPL. The REPL should be launched with all your project classes loaded and available. Please note that the Scala 2.8 REPL tab-completion does not currently work under ENSIME.

4.6 Incremental Building

Incremental building allows for fast turn-around in the running/testing of your application. The building/rebuilding support in ENSIME is intended for those who are not already using the sbt build system, as sbt users will probably wish to continue using the sbt shell's support for incremental building (it may still be worth a try though, as the build-manager included in ENSIME uses a finer grained check for modifications).

First, ensure that you've set the :target directive in your config file. The :target directory is where the classes will be written by the incremental builder. Then, type *C-c C-b b* to start building your project. When the build is finished, a window will appear listing the result of the build. After subsequent source changes, you may type *C-c C-b r* to rebuild only those parts of the project that depend on things you've changed.

4.7 Debugging

Debugging support in ENSIME is a work in progress. Feedback is welcome.
Break Points:
With your cursor on a line of Scala source, type *C-c C-d b* to set a breakpoint. Type *C-c C-d u* to remove the breakpoint. Note that breakpoints can be added and removed outside of any debug session. Breakpoints are not, however, persisted between runs of ENSIME.

Launching the Debugger:
Type *C-c C-d r* to launch the embedded Scala Debugger. ENSIME will prompt you for the class (with 'main' function) that you want to run (tab-completion works here), and then launch the debug VM. The first breakpoint your program hits will be highlighted and centered in Emacs.

Run Control:
Type *C-c C-d c* to continue after hitting a breakpoint, or *C-c C-d s* to step into the current line, or *C-c C-d n* to step to the next line, or *C-c C-d o* to step out of the current function.

Value Inspection:
When execution is paused, with your cursor over a local variable, type *C-c C-d i* to inspect the runtime value of a variable.

Show Backtrace:
When execution is paused, type *C-c C-d t* to display the current backtrace.

4.8 Refactoring

Rename:
Place your cursor over the symbol you'd like to rename. Type *M-x ensime-refactor-rename* and follow the minibuffer instructions.

Organize Imports:
Type *M-x ensime-refactor-organize-imports* in a Scala source buffer. Follow the minibuffer instructions.

Extract Method:
Select a region by setting the mark using *C-SPACE* and then placing the point at the end of the region. All selected code will be extracted into a helper method. Type *M-x ensime-refactor-extract-method* and follow the minibuffer instructions.

Inline Local:
Place your cursor over the local val whose value you'd like to inline. Type *M-x ensime-refactor-inline-local* and follow the minibuffer instructions.

4.9 Global Type and Method Search

Type *C-c C-v v* to start a global search. Type space separated keywords to filter the results of the search. For example, if I wanted to find java.util.Vector, I might start by typing 'vector', which would list all symbols containing the word 'vector'(case-insensitive), and then I would type 'java' to further refine the search. *C-p* and *C-n* move the selection up and down, respectively, and *ENTER* will jump to the source or definition of the selected symbol.

Note that typing a keyword with a capital letter will automatically enable case-sensitivity.

4.10 Source Formatting

ENSIME uses the Scalariform library to format Scala sources. Type *C-c C-v f* to format the current buffer. See section 3.3 for instructions on how to customize the formatting preferences.

4.11 Semantic Highlighting

Normally syntax highlighting is based on the *syntactic* aspects of the source code. Semantic Highlighting adds color-coding based on semantic properties of the source. For example: a syntax highlighter can't tell whether a given

identifier is a var or a val or a method call. Semantic Highlighting on the other hand can color vars differently to warn of their mutability.

Semantic Highlighting is *disabled* by default. People use a wide variety of color schemes in Emacs; it would have been difficult to arrive at coloring scheme that worked well for everyone.

Enabling Semantic Highlighting is as simple as changing the value of the variable ensime-sem-high-faces, which stores a list of (symbolType . face) associations. A 'face' can be a reference to an existing Emacs face, such as font-lock-keyword-face, or a list of the form (:foreground 'color'), where 'color' is either a standard Emacs color (such as 'slate gray') or a hex value like '#ff0000'. For example, you might add the following to your .emacs file:

```
(setq ensime-sem-high-faces
  '(
   (var . (:foreground "#ff2222"))
   (val . (:foreground "#dddddd"))
   (varField . (:foreground "#ff3333"))
   (valField . (:foreground "#dddddd"))
   (functionCall . (:foreground "#84BEE3"))
   (param . (:foreground "#ffffff"))
   (class . font-lock-type-face)
   (trait . (:foreground "#084EA8"))
   (object . (:foreground "#026DF7"))
   (package . font-lock-preprocessor-face)
   ))
```

Once the value of ensime-sem-high-faces has changed, the next time you save a file, the designated symbol types will be highlighted. By the way, the symbol types in the example above are all that are currently supported.

4.12 Scaladoc and Javadoc Browsing (customizing)

If ENSIME cannot find the source for a type or member, it will instead try to browse to the www documentation. Support is included for the java and scala standard libraries, as well as the android class library. To add your own doc library, you need to add a handler to the *ensime-doc-lookup-map*. This handler list is made up of (regex . handler) pairs, where regex is a regular expression string that will be matched against the fully qualified type name, and handler is a function that will be applied to the requested type and member and should return a url. Here's an example of how you might add new java docs for classes in com.example:

```
(defun make-example-doc-url (type &optional member)
  (ensime-make-java-doc-url-helper
    "http://developer.example.com/apidocs/" type member))
```

```
(add-to-list 'ensime-doc-lookup-map '("^com\\.example\\." . make-example-doc-url))
```

Note that *ensime-make-java-doc-url-helper*, and its Scala equivalent *ensime-make-scala-doc-url-helper*, are provided for doing the harder work of building the url paths.

4.13 Command Reference

TAB
Start completing a method/variable.

C-c C-v i or Control+Right-Click
Inspect the type of the expression under the cursor.

M-. or Control+Left-Click
Jump to definition of symbol under cursor.

M-,
Pop back to previously visited position.

C-c C-v .
Select the surrounding syntactic context. Subsequent taps of '.' and ',' will grow and shrink the selection, respectively.

C-c C-v v
Search globally for methods or types.

Control+Right-Click(on an imported package)
Inspect the package under cursor.

Mouse Hover
Echo the type of the expression under the cursor.

C-c C-v p
Inspect the package of the current source file.

C-c C-v o
Inspect the package specified in .ensime as :package.

C-c C-v r
List all references to the symbol under the cursor.

.
Forward one page in the inspector history.

,
Backward one page in the inspector history.

C-n or TAB
Forward one link in the inspector.

C-p
Backward one link in the inspector.

C-c C-v s
Switch to the sbt command-line (works for sbt projects only)

C-c C-v z
Switch to the scala interpreter, with project classes in the classpath.

C-c C-v c
Typecheck the current file.

C-c C-v a
Typecheck all files in the project.

C-c C-v e
Show all errors and warnings in the project.

C-c C-v f
Format the current Scala source file.

C-c C-v u
Undo a refactoring or formatting change.

M-n
Go to the next compilation note in the current buffer.

M-p
Go to the previous compilation note in the current buffer.

C-c C-d *x*
Where x is one of:

- **d** Start and run the debugger.

- **r** Start and run the debugger.
- **b** Set a breakpoint.
- **u** Clear a breakpoint.
- **s** Step.
- **n** Step over.
- **o** Step out.
- **c** Continue from a breakpoint.
- **q** Kill the debug session.
- **i** Inspect the local variable at cursor.
- **t** Show backtrace.

C-c C-r *x*
Where x is one of:

- **r** Rename the symbol at point.
- **o** Organize imports.
- **l** Extract local.
- **m** Extract method.
- **i** Inline local.
- **t** Add import for type at point.

C-c C-b *x*
Where x is one of:

- **b** Build the entire project.
- **r** Rebuild the project, incrementally.

M-x ensime-reload
Reload the .ensime file and recompile the project. Useful if you hit a server bug.

M-x ensime-config-get
Start the automatic configuration file generator.

5 Troubleshooting

5.1 Diagnosing Issues

You may want to examine the contents of the **inferior-ensime-server** buffer. This buffer collects the stdout and stderr of the server process, which is useful for debugging. If the compiler is in a broken state, you can restart it with M-x ensime-reload. Otherwise, if things are irreparably b0rked, you can always kill the **inferior-ensime-server** buffer (which kills the server process) and restart ensime with M-x ensime.

If you've hit a recurring bug, please post an issue to github.com/aemoncannon/ensime. Please include your OS, Emacs version, ENSIME version, and the contents of **inferior-ensime-server**.

5.2 Emacs Binary Search Path

When launching the embedded sbt shell, or the Scala repl, ENSIME uses the Emacs start-process command. Rather than using the value of the PATH environment variable, this command searches for binaries using the paths stored at the Emacs variable exec-path. On some Windows and OSX machines, exec-path will not by default contain the value of PATH. See

`http://xahlee.org/emacs/emacs_env_var_paths.html`

for more details. For example, the following Emacs Lisp could be used to manually add a Scala binary directory to the exec-path:

```
(setq exec-path (append exec-path (list "/home/aemon/scala/bin" )))
```

5.3 Custom JVM Options

If you're having problems with the default arguments (max heap, initial heap) that ENSIME uses in its startup script, you can modify the environment variable ENSIME_JVM_ARGS to override the arguments that are passed to the ENSIME Server JVM.

A Installation from Git Clone

Note: This section is for people who want to hack on ENSIME itself.

After cloning, run 'sbt update'. Then run 'sbt stage' to create the deployment directories underneath the root clone directory. Then follow the install instructions in section 2.2 above, substituting CLONE_DIR/dist as the root of your ENSIME distribution.

A common work-flow when hacking ENSIME:

- Edit source files
- 'sbt stage'
- Stop ENSIME server by killing **inferior-ensime-server** buffer
- Restart ENSIME with M-x ensime

B Running the End-to-End ENSIME Tests

- 'sbt stage'
- 'cd etc'
- 'bash run_emacs_tests.sh'
- Please be patient. These tests take a few mins to run. If all goes well you should see a buffer with a long list of 'ok's.

C Using the ENSIME Server with Other Editors

The ENSIME server is intentionally editor agnostic. It is our hope that it may be used to provide semantic information to Scala modes in other text editors. In order to interact with the ENSIME server, your editor's extension mechanism should ideally be able to open a persistent socket connection and respond to asynchronous events on that socket. Otherwise it may be difficult to interact with some of the long-running calls.

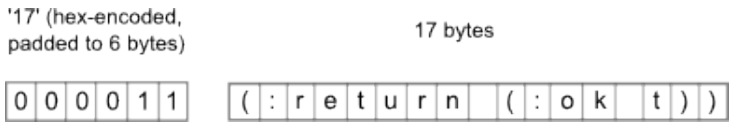
C.1 Starting the Server

Emacs starts the ENSIME server using the server.sh script in the *bin* folder of the ENSIME distribution. Rather than tell the server explicitely what tcp port it should bind to,we instead pass the filename of a temporary file to the script. The first thing the server does on startup is choose a random, open port, and write the number to the given file. Emacs then reads this file and connects to the server.

C.2 The Swank Protocol

The Emacs ENSIME client communicates with the server using the Swank protocol. This protocol was originally developed for the SLIME lisp environment for Emacs. A socket connection is maintained for the duration of the session. The client and server exchange s-expressions. At the wire level, these messages are encoded as sequences of bytes. Each message is prepended with a fixed-size header denoting its length.

To send an s-expression, first encode the s-expression as a UTF-8 string. Determine the string's length in bytes and encode that length as a padded six-digit hexadecimal string. Write this value (which will always be six bytes) to the output socket first, then write the UTF-8 encoded s-expression. On the receiving side, the reader loop should read six bytes into a buffer and convert that into an integer, then read that number of bytes from the socket. The result is a UTF-8 string representation of the s-expression. This s-expression should then be parsed using a suitable lisp reader.



See the E-Lisp function *ensime-net-send* in ensime.el for details on how messages are sent from Emacs, and the function *ensime-net-encode-length* for the implementation of the header encoding. See the functions readMessage and writeMessage in org.ensime.protocol.SwankProtocol to see how the messaging is handled in Scala.

At the application level, the s-expressions encode RPC calls, RPC responses, and events. RPC calls are used to make requests of the server. Events are generally used for the server to communicate un-prompted, asynchronous activity to the client – such as an error or warning generated during compilation. What follows is a commented excerpt from the initialization of an ENSIME session. These s-expressions were copied from the **ensime-events** buffer in Emacs which logs all protocol events (useful for learning the application protocol!). Server messages are indented. Comments prefixed with #.

```
# The client requests information about the server
(:swank-rpc
 (swank:connection-info) 1)

# Server responds. Note the numbering scheme for RPC calls: call & response
# have the same call id (1 in this case).
(:return
```

```
 (:ok
  (:pid nil :server-implementation
(:name "ENSIMEserver")
:machine nil :features nil :version "0.0.1")) 1)

# The client initializes the session by sending the configuration.
# This is read from the .ensime file.
(:swank-rpc
 (swank:init-project
  (:package "org.ensime" :root-dir "/home/aemon/src/misc/ensime/")) 2)

# The acknowledges the message and sends some useful project info
# back to the client.
(:return
 (:ok
  (:name "ensime" :source-roots
 ("/home/aemon/src/misc/ensime/src/main/scala"
 "/home/aemon/src/misc/ensime/src/main/java"
 "/home/aemon/src/misc/ensime/src/test/scala"))) 2)


# Server asks client to display a user message.
# Note: this is not part of any RPC call - it's just an event from the server.
(:background-message 105 "Initializing Analyzer. Please wait...")

# Server alerts client that the analyzer is now ready for RPC requests.
(:compiler-ready t)

# Server tells client the result of the last full compilation
# (nil means 0 notes (errors or warnings))
(:typecheck-result
 (:lang :scala :is-full t :notes nil))
```

C.3 ENSIME Swank RPC: Version 0.8.7

C.3.1 Protocol Change Log

```
0.8.8
  Add optional :archive member to Position and RangePosition
0.8.7
  Add optional file contents parameter to typecheck-file
0.8.6
  Add support for ranges to type-at-point, inspect-type-at-point,
    type-by-name-at-point
0.8.5
```

```
  DebugLocation of type 'field now gets field name from :field, not from :name
  The debug-to-string call now requires thread id
0.8.4
  Add local-name to SymbolInfo
0.8.3
  Add debug-to-string call.
  Refactor debug-value-for-name to debug-locate-name + debug-value
  Adds typecheck-files
  Unifies debug locations under DebugLocation
0.8.2
  Debug attachment to remote VM
  CompletionInfo type-sig now has structure see CompletionSignature
0.8.1
  Add patch-source.
  Completions now takes a 'reload' argument.
0.8
  Add RPC calls for debugging
  Protocol is now explicitely UTF-8
0.7.4
  Add optional 'owner-type-id' key to SymbolInfo
  Add optional 'case-sens' option to swank:completions call
0.7.3
  Add optional 'to-insert' key to CompletionInfo
  Add optional a max results argument to swank:completions call
0.7.2
  Get rid of scope and type completion in favor of unified
  swank:completions call.
  Wrap completion result in CompletionInfoList.
0.7.1
  Remove superfluous status values for events such as :compiler-ready,
  :clear-scala-notes, etc.
0.7
  Rename swank:perform-refactor to swank:prepare-refactor.
  Include status flag in return of swank:exec-refactor.
```

C.3.2 Important Datastructures

Certain datastructures, such as the *position* structure used to describe a source position, are re-used in many RPC calls. Implementors may wish to factor these structures out as classes or utility functions.

Position

A source position.

```
(
```

```
  :file    //String:A filename. If :archive is set, :file is the entry within the arc
  :archive //String(optional): If set, a jar or zip archive that contains :file
  :offset  //Int:The zero-indexed character offset of this position.
)
```

RangePosition

A source position that describes a range of characters in a file.

```
(
  :file   //String:A filename. If :archive is set, :file is the entry within the arch
  :archive //String(optional): If set, a jar or zip archive that contains :file
  :start  //Int:The character offset of the start of the range.
  :end    //Int:The character offset of the end of the range.
)
```

Change

Describes a change to a source code file.

```
(
:file //String:Filename of source  to be changed
:text //String:Text to be inserted
:from //Int:Character offset of start of text to replace.
:to //Int:Character offset of end of text to replace.
)
```

SymbolSearchResult

Describes a symbol found in a search operation.

```
(
:name //String:Qualified name of symbol.
:local-name //String:Unqualified name of symbol
:decl-as //Symbol:What kind of symbol this is.
:owner-name //String:If symbol is a method, gives the qualified owner type.
:pos //Position:Where is this symbol declared?
)
```

ParamSectionInfo

Description of one of a method's parameter sections.

```
(
:params //List of (String TypeInfo) pairs:Describes params in section
:is-implicit //Bool:Is this an implicit parameter section.
)
```

CallCompletionInfo

Description of a Scala method's type

```
(
:result-type //TypeInfo
:param-sections //List of ParamSectionInfo:
)
```

TypeMemberInfo

Description of a type member

```
(
:name //String:The name of this member.
:type-sig //String:The type signature of this member
:type-id //Int:The type id for this member's type.
:is-callable //Bool:Is this a function or method type.
)
```

TypeInfo

Description of a Scala type.

```
(
:name //String:The short name of this type.
:type-id //Int:Type Id of this type (for fast lookups)
:full-name //String:The qualified name of this type
:decl-as //Symbol:What kind of type is this? (class,trait,object, etc)
```

```
:type-args //List of TypeInfo:Type args this type has been applied to.
:members //List of TypeMemberInfo
:arrow-type //Bool:Is this a function or method type?
:result-type //TypeInfo:
:param-sections //List of ParamSectionInfo:
:pos //Position:Position where this type was declared
:outer-type-id //Int:If this is a nested type, type id of owning type
)
```

InterfaceInfo

Describes an inteface that a type supports

```
(
:type //TypeInfo:The type of the interface.
:via-view //Bool:Is this type suppoted via an implicit conversion?
)
```

TypeInspectInfo

Detailed description of a Scala type.

```
(
:type //TypeInfo
:companion-id //Int:Type Id of this type's companion type.
:interfaces //List of InterfaceInfo:Interfaces this type supports
)
```

SymbolInfo

Description of a Scala symbol.

```
(
:name //String:Name of this symbol.
:local-name //String:Unqualified name of this symbol.
:type //TypeInfo:The type of this symbol.
:decl-pos //Position:Source location of this symbol's declaration.
:is-callable //Bool:Is this symbol a method or function?
:owner-type-id //Int: (optional) Type id of owner type.
)
```

CompletionSignature

An abbreviated signature for a type member

```
(
//List of List of Pairs of String: Parameter sections
//String: Result type
)
```

CompletionInfo

An abbreviated symbol description.

```
(
:name //String:Name of this symbol.
:type-sig //A CompletionSignature
:type-id //Int:A type id.
:is-callable //Bool:Is this symbol a method or function?
:to-insert //String|Nil:The representation that should be
  written to the buffer.
)
```

CompletionInfoList

An annotated collection of CompletionInfo structures.

```
(
:prefix //String:The common prefix for all selections,
  modulo case.
:completions //List of CompletionInfo:In order of descending
  relevance.
)
```

PackageInfo

A description of a package and all its members.

```
(
:name //String:Name of this package.
:full-name //String:Qualified name of this package.
:members //List of PackageInfo | TypeInfo:The members of this package.
:info-type //Symbol:Literally 'package
)
```

SymbolDesignations

Describe the symbol classes in a given textual range.

```
(
:file //String:Filename of file to be annotated.
:syms //List of (Symbol Integer Integer):Denoting the symbol class and start and end
)
```

PackageMemberInfoLight

An abbreviated package member description.

```
(
:name //String:Name of this symbol.
)
```

RefactorFailure

Notification that a refactoring has failed in some way.

```
(
:procedure-id //Int:The id for this refactoring.
:message //String:A text description of the error.
:status //Symbol:'failure.
)
```

RefactorEffect

A description of the effects a proposed refactoring would have.

```
(
:procedure-id //Int:The id for this refactoring.
:refactor-type //Symbol:The type of refactoring.
:status //Symbol:'success
:changes //List of Change:The textual effects.
)
```

RefactorResult

A description of the effects a refactoring has effected.

```
(
:procedure-id //Int:The id for this refactoring.
:refactor-type //Symbol:The type of refactoring.
:touched //List of String:Names of files touched by the refactoring.
:status //Symbol:'success.
)
```

Note

Describes a note generated by the compiler.

```
(
:severity //Symbol: One of 'error, 'warn, 'info.
:msg //String: Text of the compiler message.
:beg //Int: Zero-based offset of beginning of region
:end //Int: Zero-based offset of end of region
:line //Int: Line number of region
:col //Int: Column offset of region
:file //String: Filename of source file
)
```

Notes

Describes a set of notes generated by the compiler.

```
(
:is-full //Bool: Is the note the result of a full compilation?
:notes //List of Note: The descriptions of the notes themselves.
)
```

FilePatch

Describes a patch to be applied to a single file.

```
(
// '+' is an insert of text before the existing text starting at i
// '-' is a deletion of text in interval [i,j)
// '*' is a replacement of text in interval [i,j)
[("+" i "some text") | ("-" i j) | ("*" i j "some text")]*
)
```

DebugLocation

A unique location in the VM under debug. Note: this datastructure
is a union of several location types.

```
(
  :type   //Symbol:One of 'reference, 'field, 'element, 'slot
  [ // if type is 'reference
  :object-id  //String:The unique id of the object.
  ]
  [ // if type is 'field
  :object-id  //String:The unique id of the object.
  :field  //String:Name of the field of the object.
  ]
  [ // if type is 'element
  :object-id  //String:The unique id of the array.
  :index  //Int:A zero-indexed offset within array.
  ]
  [ // if type is 'slot
  :thread-id  //String:The unique id of the thread.
  :frame  //Int:Select the zero-indexed frame in the thread's call stack.
  :offset  //Int:A zero-indexed offset within frame.
  ]
)
```

C.3.3 RPC Calls

The ENSIME server understands all of the following RPC calls:

swank:connection-info

Request connection information.

Arguments: None

Return:

```
(
:pid  //Int:The integer process id of the server (or nil if unnavailable)
:implementation
  (
  :name  //String:An identifying name for this server implementation.
  )
:version //String:The version of the protocol this server supports.
)
```

Example Call:

```
(:swank-rpc (swank:connection-info) 42)
```

Example Return:

```
(:return (:ok (:pid nil :implementation (:name "ENSIME - Reference Server")
:version "0.7")) 42)
```

swank:init-project

Initialize the server with a project configuration. The server returns it's own knowledge about the project, including source roots which can be used by clients to determine whether a given source file belongs to this project.

Arguments:

```
A complete ENSIME configuration property list. See manual.
```

Return:

```
(
:project-name //String:The name of the project.
:source-roots //List of Strings:The source code directory roots..
)
```

Example Call:

```
(:swank-rpc (swank:init-project (:use-sbt t :compiler-args
(-Ywarn-dead-code -Ywarn-catches -Xstrict-warnings)
:root-dir /Users/aemon/projects/ensime/)) 42)
```

Example Return:

```
(:return (:ok (:project-name "ensime" :source-roots
("/Users/aemon/projects/ensime/src/main/scala"
"/Users/aemon/projects/ensime/src/test/scala"
"/Users/aemon/projects/ensime/src/main/java"))) 42)
```

swank:peek-undo

The intention of this call is to preview the effect of an undo before executing it.

Arguments: None

Return:

```
(
:id //Int:Id of this undo
:changes //List of Changes:Describes changes this undo would effect.
:summary //String:Summary of action this undo would revert.
)
```

Example Call:

```
(:swank-rpc (swank:peek-undo) 42)
```

Example Return:

```
(:return (:ok (:id 1 :changes ((:file
"/ensime/src/main/scala/org/ensime/server/RPCTarget.scala"
:text "rpcInitProject" :from 2280 :to 2284))
:summary "Refactoring of type: 'rename") 42)
```

swank:exec-undo

Execute a specific, server-side undo operation.

Arguments:

```
An integer undo id. See swank:peek-undo for how to learn this id.
```

Return:

```
(
:id //Int:Id of this undo
:touched-files //List of Strings:Filenames of touched files,
)
```

Example Call:

```
(:swank-rpc (swank:exec-undo 1) 42)
```

Example Return:

```
(:return (:ok (:id 1 :touched-files
("/src/main/scala/org/ensime/server/RPCTarget.scala"))) 42)
```

swank:repl-config

Get information necessary to launch a scala repl for this project.

Arguments: None

Return:

```
(
:classpath //String:Classpath string formatted for passing to Scala.
)
```

Example Call:

```
(:swank-rpc (swank:repl-config) 42)
```

Example Return:

```
(:return (:ok (:classpath "lib1.jar:lib2.jar:lib3.jar")) 42)
```

swank:builder-init

Initialize the incremental builder and kick off a full rebuild.

Arguments: None

Return: None

Example Call:

```
(:swank-rpc (swank:builder-init) 42)
```

Example Return:

```
(:return (:ok ()) 42)
```

swank:builder-update-files

Signal to the incremental builder that the given files have changed and must be rebuilt. Triggers rebuild.

Arguments:

```
List of Strings:Filenames, absolute or relative to the project root.
```

Return: None

Example Call:

```
(:swank-rpc (swank:builder-update-files
("/ensime/src/main/scala/org/ensime/server/Analyzer.scala")) 42)
```

Example Return:

```
(:return (:ok ()) 42)
```

swank:builder-add-files

Signal to the incremental builder that the given files should be added to the build. Triggers rebuild.

Arguments:

```
List of Strings:Filenames, absolute or relative to the project root.
```

Return: None

Example Call:

```
(:swank-rpc (swank:builder-add-files
("/ensime/src/main/scala/org/ensime/server/Analyzer.scala")) 42)
```

Example Return:

```
(:return (:ok ()) 42)
```

swank:builder-remove-files

Signal to the incremental builder that the given files should be removed from the build. Triggers rebuild.

Arguments:

```
List of Strings:Filenames, absolute or relative to the project root.
```

Return: None

Example Call:

```
(:swank-rpc (swank:builder-remove-files
("/ensime/src/main/scala/org/ensime/server/Analyzer.scala")) 42)
```

Example Return:

```
(:return (:ok ()) 42)
```

swank:remove-file

Remove a file from consideration by the ENSIME analyzer.

Arguments:

```
String:A filename, absolute or relative to the project.
```

Return: None

Example Call:

```
(:swank-rpc (swank:remove-file "Analyzer.scala") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:typecheck-file

Request immediate load and check the given source file.

Arguments:

```
String:A filename, absolute or relative to the project.
String(optional): if set, it is substituted for the file's contents
```

Return: None

Example Call:

```
(:swank-rpc (swank:typecheck-file "Analyzer.scala") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:typecheck-files

Request immediate load and check the given source files.

Arguments:

```
List of String:Filenames, absolute or relative to the project.
```

Return: None

Example Call:

```
(:swank-rpc (swank:typecheck-files ("Analyzer.scala")) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:patch-source

Request immediate load and check the given source file.

Arguments:

```
String:A filename
A FilePatch
```

Return: None

Example Call:

```
(swank:patch-source "Analyzer.scala" (("+" 6461 "Inc")
  ("-" 7127 7128)))
```

Example Return:

```
(:return (:ok t) 42)
```

swank:typecheck-all

Request immediate load and typecheck of all known sources.

Arguments: None

Return: None

Example Call:

```
(:swank-rpc (swank:typecheck-all) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:format-source

Run the source formatter the given source files. Writes the formatted sources to the disk. Note: the client is responsible for reloading the files from disk to display to user.

Arguments:

```
List of String:Filenames, absolute or relative to the project.
```

Return: None

Example Call:

```
(:swank-rpc (swank:format-source ("/ensime/src/Test.scala")) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:public-symbol-search

Search top-level symbols (types and methods) for names that contain ALL the given search keywords.

Arguments:

```
List of Strings:Keywords that will be ANDed to form the query.
Int:Maximum number of results to return.
```

Return:

```
List of SymbolSearchResults
```

Example Call:

```
(:swank-rpc (swank:public-symbol-search ("java" "io" "File") 50) 42)
```

Example Return:

```
(:return (:ok ((:name "java.io.File" :local-name "File" :decl-as class
:pos (:file "/Classes/classes.jar" :offset -1))) 42)
```

swank:import-suggestions

Search top-level types for qualified names similar to the given type names. This call can service requests for many typenames at once, but this isn't currently used in ENSIME.

Arguments:

```
String:Source filename, absolute or relative to the project.
Int:Character offset within that file where type name is referenced.
List of String:Type names (possibly partial) for which to suggest.
Int:The maximum number of results to return.
```

Return:

```
List of Lists of SymbolSearchResults:Each list corresponds to one of the
  type name arguments.
```

Example Call:

```
(:swank-rpc (swank:import-suggestions
"/ensime/src/main/scala/org/ensime/server/Analyzer.scala"
2300 (Actor) 10) 42)
```

Example Return:

```
(:return (:ok (((:name "scala.actors.Actor" :local-name "Actor"
:decl-as trait :pos (:file "/lib/scala-library.jar" :offset -1)))))
42)
```

swank:completions

Find viable completions at given point.

Arguments:

```
String:Source filename, absolute or relative to the project.
Int:Character offset within that file.
Int:Max number of completions to return. Value of zero denotes
  no limit.
Bool:If non-nil, only return prefixes that match the case of the
  prefix.
Bool:If non-nil, reload source from disk before computing completions.
```

Return:

```
CompletionInfoList: The list of completions
```

Example Call:

```
(:swank-rpc (swank:completions
"/ensime/src/main/scala/org/ensime/protocol/SwankProtocol.scala
22626 0 t) 42)
```

Example Return:

```
(:return (:ok (:prefix "form" :completions
((:name "form" :type-sig "SExp" :type-id 10)
(:name "format" :type-sig "(String, <repeated>[Any]) => String"
:type-id 11 :is-callable t))) 42))
```

swank:package-member-completion

Find possible completions for a given package path.

Arguments:

```
String:A package path: such as "org.ensime" or "com".
String:The prefix of the package member name we are looking for.
```

Return:

```
List of PackageMemberInfoLight: List of possible completions.
```

Example Call:

```
(:swank-rpc (swank:package-member-completion org.ensime.server Server)
42)
```

Example Return:

```
(:return (:ok ((:name "Server$") (:name "Server"))) 42)
```

swank:call-completion

Lookup the type information of a specific method or function type. This is used by ENSIME to retrieve detailed parameter and return type information after the user has selected a method or function completion.

Arguments:

```
Int:A type id, as returned by swank:scope-completion or
  swank:type-completion.
```

Return:

```
A CallCompletionInfo
```

Example Call:

```
(:swank-rpc (swank:call-completion 1)) 42)
```

Example Return:

```
(:return (:ok (:result-type (:name "Unit" :type-id 7 :full-name
"scala.Unit" :decl-as class) :param-sections ((:params (("id"
(:name "Int" :type-id 74 :full-name "scala.Int" :decl-as class))
("callId" (:name "Int" :type-id 74 :full-name "scala.Int"
:decl-as class)))))) 42)
```

swank:uses-of-symbol-at-point

Request all source locations where indicated symbol is used in this project.

Arguments:

```
String:A Scala source filename, absolute or relative to the project.
Int:Character offset of the desired symbol within that file.
```

Return:

```
List of RangePosition:Locations where the symbol is reference.
```

Example Call:

```
(:swank-rpc (swank:uses-of-symbol-at-point "Test.scala" 11334) 42)
```

Example Return:

```
(:return (:ok ((:file "RichPresentationCompiler.scala" :offset 11442
:start 11428 :end 11849) (:file "RichPresentationCompiler.scala"
:offset 11319 :start 11319 :end 11339))) 42)
```

swank:type-by-id

Request description of the type with given type id.

Arguments:

```
Int:A type id.
```

Return:

```
A TypeIfo
```

Example Call:

```
(:swank-rpc (swank:type-by-id 1381) 42)
```

Example Return:

```
(:return (:ok (:name "Option" :type-id 1381 :full-name "scala.Option"
:decl-as class :type-args ((:name "Int" :type-id 1129 :full-name "scala.Int"
:decl-as class)))) 42)
```

swank:type-by-name

Lookup a type description by name.

Arguments:

```
String:The fully qualified name of a type.
```

Return:

```
A TypeIfo
```

Example Call:

```
(:swank-rpc (swank:type-by-name "java.lang.String") 42)
```

Example Return:

```
(:return (:ok (:name "String" :type-id 1188 :full-name
"java.lang.String" :decl-as class)) 42)
```

swank:type-by-name-at-point

Lookup a type by name, in a specific source context.

Arguments:

```
String:The local or qualified name of the type.
String:A source filename.
Int or (Int, Int):A character offset (or range) in the file.a
```

Return:

```
A TypeInfo
```

Example Call:

```
(:swank-rpc (swank:type-by-name-at-point "String"
"SwankProtocol.scala" 31680) 42)
```

Example Return:

```
(:return (:ok (:name "String" :type-id 1188 :full-name
"java.lang.String" :decl-as class)) 42)
```

swank:type-at-point

Lookup type of thing at given position.

Arguments:

```
String:A source filename.
Int or (Int, Int):A character offset (or range) in the file.
```

Return:

```
A TypeInfo
```

Example Call:

```
(:swank-rpc (swank:type-at-point "SwankProtocol.scala"
 32736) 42)
```

Example Return:

```
(:return (:ok (:name "String" :type-id 1188 :full-name
"java.lang.String" :decl-as class)) 42)
```

swank:inspect-type-at-point

Lookup detailed type of thing at given position.

Arguments:

```
String:A source filename.
Int or (Int, Int):A character offset (or range) in the file.
```

Return:

```
A TypeInspectInfo
```

Example Call:

```
(:swank-rpc (swank:inspect-type-at-point "SwankProtocol.scala"
 32736) 42)
```

Example Return:

```
(:return (:ok (:type (:name "SExpList$" :type-id 1469 :full-name
"org.ensime.util.SExpList$" :decl-as object :pos
(:file "SExp.scala" :offset 1877))......)) 42)
```

swank:inspect-type-by-id

Lookup detailed type description by id

Arguments:

```
Int:A type id.
```

Return:

```
A TypeInspectInfo
```

Example Call:

```
(:swank-rpc (swank:inspect-type-by-id 232) 42)
```

Example Return:

```
(:return (:ok (:type (:name "SExpList$" :type-id 1469 :full-name
"org.ensime.util.SExpList$" :decl-as object :pos
(:file "SExp.scala" :offset 1877))......)) 42)
```

swank:symbol-at-point

Get a description of the symbol at given location.

Arguments:

```
String:A source filename.
Int:A character offset in the file.
```

Return:

```
A SymbolInfo
```

Example Call:

```
(:swank-rpc (swank:symbol-at-point "SwankProtocol.scala" 36483) 42)
```

Example Return:

```
(:return (:ok (:name "file" :type (:name "String" :type-id 25
:full-name "java.lang.String" :decl-as class) :decl-pos
(:file "SwankProtocol.scala" :offset 36404))) 42)
```

swank:inspect-package-by-path

Get a detailed description of the given package.

Arguments:

```
String:A qualified package name.
```

Return:

```
A PackageInfo
```

Example Call:

```
(:swank-rpc (swank:inspect-package-by-path "org.ensime.util" 36483) 42)
```

Example Return:

```
(:return (:ok (:name "util" :info-type package :full-name "org.ensime.util"
:members ((:name "BooleanAtom" :type-id 278 :full-name
"org.ensime.util.BooleanAtom" :decl-as class :pos
(:file "SExp.scala" :offset 2848)).....))) 42)
```

swank:prepare-refactor

Initiate a refactoring. The server will respond with a summary of what the refactoring *would* do, were it executed.This call does not effect any changes unless the 4th argument is nil.

Arguments:

```
Int:A procedure id for this refactoring, uniquely generated by client.
Symbol:The manner of refacoring we want to prepare. Currently, one of
  rename, extractMethod, extractLocal, organizeImports, or addImport.
An association list of params of the form (sym1 val1 sym2 val2).
  Contents of the params varies with the refactoring type:
  rename: (newName String file String start Int end Int)
  extractMethod: (methodName String file String start Int end Int)
  extractLocal: (name String file String start Int end Int)
  inlineLocal: (file String start Int end Int)
```

```
  organizeImports: (file String)
  addImport: (qualifiedName String file String start Int end Int)
Bool:Should the refactoring require confirmation? If nil, the refactoring
  will be executed immediately.
```

Return:

```
RefactorEffect | RefactorFailure
```

Example Call:

```
(:swank-rpc (swank:prepare-refactor 6 rename (file "SwankProtocol.scala"
start 39504 end 39508 newName "dude") t) 42)
```

Example Return:

```
(:return (:ok (:procedure-id 6 :refactor-type rename :status success
:changes ((:file "SwankProtocol.scala" :text "dude" :from 39504 :to 39508))
)) 42)
```

swank:exec-refactor

Execute a refactoring, usually after user confirmation.

Arguments:

```
Int:A procedure id for this refactoring, uniquely generated by client.
Symbol:The manner of refacoring we want to prepare. Currently, one of
  rename, extractMethod, extractLocal, organizeImports, or addImport.
```

Return:

```
RefactorResult | RefactorFailure
```

Example Call:

```
(:swank-rpc (swank:exec-refactor 7 rename) 42)
```

Example Return:

```
(:return (:ok (:procedure-id 7 :refactor-type rename
:touched-files ("SwankProtocol.scala"))) 42)
```

swank:cancel-refactor

Cancel a refactor that's been performed but not executed.

Arguments:

```
Int:Procedure Id of the refactoring.
```

Return: None

Example Call:

```
(:swank-rpc (swank:cancel-refactor 1) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:symbol-designations

Request the semantic classes of symbols in the given range. These classes are intended to be used for semantic highlighting.

Arguments:

```
String:A source filename.
Int:The character offset of the start of the input range.
Int:The character offset of the end of the input range.
List of Symbol:The symbol classes in which we are interested.
  Available classes are: var,val,varField,valField,functionCall,
  operator,param,class,trait,object.
```

Return:

```
SymbolDesignations
```

Example Call:

```
(:swank-rpc (swank:symbol-designations "SwankProtocol.scala" 0 46857
(var val varField valField)) 42)
```

Example Return:

```
(:return (:ok (:file "SwankProtocol.scala" :syms
((varField 33625 33634) (val 33657 33661) (val 33663 33668)
(varField 34369 34378) (val 34398 34400)))) 42)
```

swank:expand-selection

Given a start and end point in a file, expand the selection so that it spans the smallest syntactic scope that contains start and end.

Arguments:

```
String:A source filename.
Int:The character offset of the start of the input range.
Int:The character offset of the end of the input range.
```

Return:

```
A RangePosition:The expanded range.
```

Example Call:

```
(:swank-rpc (swank:expand-selection "Model.scala" 4648 4721) 42)
```

Example Return:

```
(:return (:ok (:file "Model.scala" :start 4374 :end 14085)) 42)
```

swank:method-bytecode

Get bytecode for method at file and line.

Arguments:

```
String:The file in which the method is defined.
Int:A line within the method's code.
```

Return:

```
A MethodBytecode
```

Example Call:

```
(:swank-rpc (swank:method-bytecode "hello.scala" 12) 42)
```

Example Return:

```
(:return
(:ok (
:class-name "SomeClassName"
:name "SomeMethodName"
:signature ??
:bytcode ("opName" "opDescription" ...
)
42)
```

swank:debug-active-vm

Is a there an active vm? if so return a description.

Arguments: None

Return:

```
Nil | A short description of the current vm.
```

Example Call:

```
(:swank-rpc (swank:debug-active-vm) 42)
```

Example Return:

```
(:return (:ok nil) 42)
```

swank:debug-start

Start a new debug session.

Arguments:

```
String:The commandline to pass to the debugger. Of the form:
  "package.ClassName arg1 arg2....."
```

Return: None

Example Call:

```
(:swank-rpc (swank:debug-start "org.hello.HelloWorld arg") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-attach

Start a new debug session on a target vm.

Arguments:

```
String: The hostname of the vm
String: The debug port of the vm
```

Return: None

Example Call:

```
(:swank-rpc (swank:debug-attach "localhost" "9000") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-stop

Stop the debug session

Arguments: None

Return: None

Example Call:

```
(:swank-rpc (swank:debug-stop) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-set-break

Add a breakpoint

Arguments:

```
String:The file in which to set the breakpoint.
Int:The breakpoint line.
```

Return: None

Example Call:

```
(:swank-rpc (swank:debug-set-break "hello.scala" 12) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-clear-break

Clear a breakpoint

Arguments:

```
String:The file from which to clear the breakpoint.
Int:The breakpoint line.
```

Return: None

Example Call:

```
(:swank-rpc (swank:debug-clear "hello.scala" 12) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-clear-all-breaks

Clear all breakpoints

Arguments: None

Return: None

Example Call:

```
(:swank-rpc (swank:debug-clear-all-breaks) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-list-breakpoints

Get a list of all breakpoints set so far.

Arguments: None

Return:

```
List of Position:A list of positions
```

Example Call:

```
(:swank-rpc (swank:debug-list-breakpoints) 42)
```

Example Return:

```
(:return (:ok (:file "hello.scala" :line 1)
(:file "hello.scala" :line 23)) 42)
```

swank:debug-run

Resume execution of the VM.

Arguments: None

Return: None

Example Call:

```
(:swank-rpc (swank:debug-run) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-continue

Resume execution of the VM.

Arguments:

```
String:The thread-id to continue.
```

Return: None

Example Call:

```
(:swank-rpc (swank:debug-continue "1") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-step

Step the given thread to the next line. Step into function calls.

Arguments:

```
String:The thread-id to step.
```

Return: None

Example Call:

```
(:swank-rpc (swank:debug-step "982398123") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-next

Step the given thread to the next line. Do not step into function calls.

Arguments:

```
String:The thread-id to step.
```

Return: None

Example Call:

```
(:swank-rpc (swank:debug-next "982398123") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-step-out

Step the given thread to the next line. Step out of the current function to the calling frame if necessary.

Arguments:

```
String:The thread-id to step.
```

Return: None

Example Call:

```
(:swank-rpc (swank:debug-step-out "982398123") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-locate-name

Get the binding location for the given name at this point in the program's execution.

Arguments:

```
String: The thread-id in which to search.
String: The name to search for.
```

Return:

```
A DebugLocation
```

Example Call:

```
(:swank-rpc (swank:debug-locate-name "thread-2" "apple") 42)
```

Example Return:

```
(:return (:ok (:slot :thread-id "thread-2" :frame 2 :offset 0)) 42)
```

swank:debug-value

Get the value at the given location.

Arguments:

```
DebugLocation: The location from which to load the value.
```

Return:

```
A DebugValue
```

Example Call:

```
(:swank-rpc (swank:debug-value (:type element
 :object-id "23" :index 2)) 42)
```

Example Return:

```
(:return (:ok (:val-type prim :summary "23"
 :type-name "Integer")) 42)
```

swank:debug-to-string

Returns the result of calling toString on the value at the given location

Arguments:

```
String: The thread-id in which to call toString.
DebugLocation: The location from which to load the value.
```

Return:

```
A DebugValue
```

Example Call:

```
(:swank-rpc (swank:debug-to-string "thread-2"
 (:type element :object-id "23" :index 2)) 42)
```

Example Return:

```
(:return (:ok "A little lamb") 42)
```

swank:debug-set-value

Set the value at the given location.

Arguments:

```
DebugLocation: Location to set value.
String: A string encoded value.
```

Return:

```
Boolean: t on success, nil otherwise
```

Example Call:

```
(:swank-rpc (swank:debug-set-stack-var (:type element
 :object-id "23" :index 2) "1") 42)
```

Example Return:

```
(:return (:ok t) 42)
```

swank:debug-backtrace

Get a detailed backtrace for the given thread

Arguments:

```
String: The unique id of the thread.
Int: The index of the first frame to list. The 0th frame is the
  currently executing frame.
Int: The number of frames to return. -1 denotes _all_ frames.
```

Return:

```
A DebugBacktrace
```

Example Call:

```
(:swank-rpc (swank:debug-backtrace "23" 0 2) 42)
```

Example Return:

```
(:return (:ok (:frames () :thread-id "23" :thread-name "main")) 42)
```

swank:shutdown-server

Politely ask the server to shutdown.

Arguments: None

Return: None

Example Call:

```
(:swank-rpc (swank:shutdown-server) 42)
```

Example Return:

```
(:return (:ok t) 42)
```

C.3.4 Events

The ENSIME server will dispatch the following types of events:

:compiler-ready

Signal that the compiler has finished its initial compilation and the server is ready to accept RPC calls.

```
(:compiler-ready)
```

:full-typecheck-finished

Signal that the compiler has finished compilation of the entire project.

```
(:full-typecheck-finished)
```

:indexer-ready

Signal that the indexer has finished indexing the classpath.

```
(:indexer-ready)
```

:scala-notes

Notify client when Scala compiler generates errors,warnings or other notes.

```
(:scala-notes
notes //List of Note
)
```

:java-notes

Notify client when Java compiler generates errors,warnings or other notes.

```
(:java-notes
notes //List of Note
)
```

:clear-all-scala-notes

Notify client when Scala notes have become invalidated. Editor should consider all Scala related notes to be stale at this point.

```
(:clear-all-scala-notes)
```

:clear-all-java-notes

Notify client when Java notes have become invalidated. Editor should consider all Java related notes to be stale at this point.

```
(:clear-all-java-notes)
```

:debug-event (:type output)

Communicates stdout/stderr of debugged VM to client.

```
(:debug-event
  (:type //Symbol: output
   :body //String: A chunk of output text
))
```

:debug-event (:type step)

Signals that the debugged VM has stepped to a new location and is now paused awaiting control.

```
(:debug-event
  (:type //Symbol: step
   :thread-id //String: The unique thread id of the paused thread.
   :thread-name //String: The informal name of the paused thread.
   :file //String: The source file the VM stepped into.
   :line //Int: The source line the VM stepped to.
))
```

:debug-event (:type breakpoint)

Signals that the debugged VM has stopped at a breakpoint.

```
(:debug-event
  (:type //Symbol: breakpoint
   :thread-id //String: The unique thread id of the paused thread.
   :thread-name //String: The informal name of the paused thread.
   :file //String: The source file the VM stepped into.
   :line //Int: The source line the VM stepped to.
))
```

:debug-event (:type death)

Signals that the debugged VM has exited.

```
(:debug-event
  (:type //Symbol: death
))
```

:debug-event (:type start)

Signals that the debugged VM has started.

```
(:debug-event
  (:type //Symbol: start
))
```

:debug-event (:type disconnect)

Signals that the debugger has disconnected form the debugged VM.

```
(:debug-event
  (:type //Symbol: disconnect
))
```

:debug-event (:type exception)

Signals that the debugged VM has thrown an exception and is now paused waiting for control.

```
(:debug-event
  (:type //Symbol: exception
   :exception //String: The unique object id of the exception.
   :thread-id //String: The unique thread id of the paused thread.
   :thread-name //String: The informal name of the paused thread.
   :file //String: The source file where the exception was caught,
      or nil if no location is known.
   :line //Int: The source line where the exception was thrown,
      or nil if no location is known.
))
```

:debug-event (:type threadStart)

Signals that a new thread has started.

```
(:debug-event
  (:type //Symbol: threadStart
   :thread-id //String: The unique thread id of the new thread.
))
```

:debug-event (:type threadDeath)

Signals that a new thread has died.

```
(:debug-event
  (:type //Symbol: threadDeath
   :thread-id //String: The unique thread id of the new thread.
))
```

C.4 Other Protocols

The ENSIME server is designed to support pluggable protocols. org.ensime.protocol.SwankProtocol is just one implementation of the org.ensime.protocol.Protocol interface. Adding a new protocol (JSON-based, or binary or Java marshalled objects...) should only require adding a new implementation of org.ensime.protocol.Protocol. Please contact the ENSIME maintainer if this is your plan, however, since we still need to add a command-line switch to control the protocol that ENSIME uses.