

Debugging Rust with Visual Studio Code

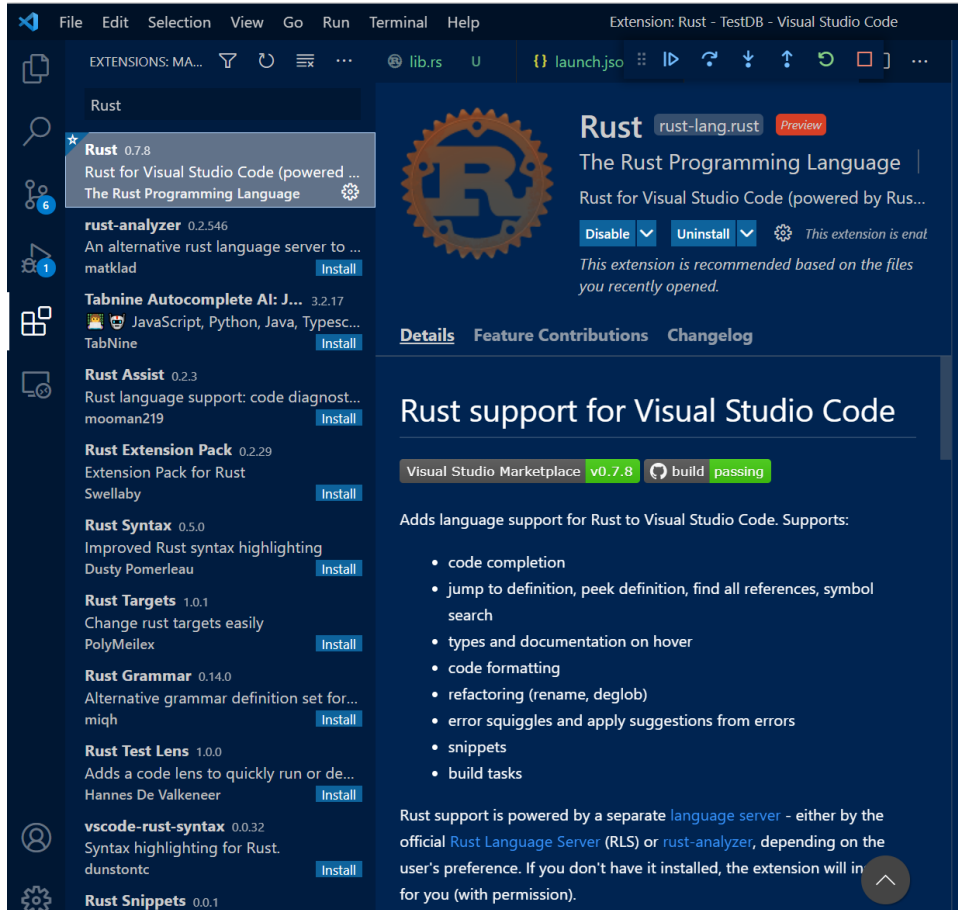
Jim Fawcett

<https://JimFawcett.github.io>

Contents

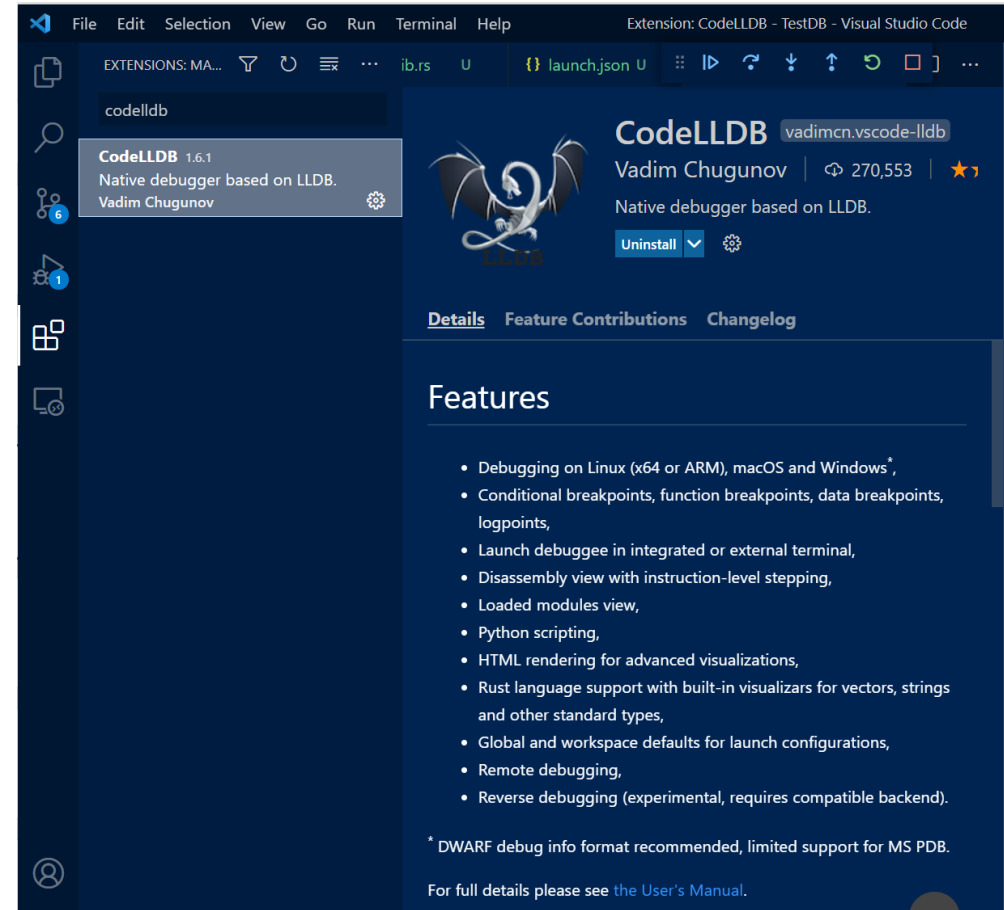
- Add Visual Studio Code Plugins: Rust, CodeLLDB
- Create new Rust library package
- Populate package with a small amount of Rust code
 - lib.rs, test1.rs, test2.rs
- Start debug session: Run -> start with debugging
- Dismiss no launch popup, Yes to create launch popup
- Look at launch.json content
- Start debugging

Add Plugins to Visual Studio Code



Visual Studio Code interface showing the Rust extension marketplace. The search results list several Rust-related extensions, with the main extension, "Rust" (rust-lang.rust), selected. The details for the "Rust" extension are displayed on the right, including its description, version (0.7.8), and installation options. The extension is described as "The Rust Programming Language" and "Rust for Visual Studio Code (powered by Rust Language Server)". It is marked as a "Preview" extension and is currently installed. The details section includes a "Rust support for Visual Studio Code" section with a "Visual Studio Marketplace" badge for version v0.7.8, which is marked as "build passing". Below this, it lists supported features: code completion, jump to definition, peek definition, find all references, symbol search, types and documentation on hover, code formatting, refactoring (rename, deglob), error squiggles and apply suggestions from errors, snippets, and build tasks. A note states: "Rust support is powered by a separate language server - either by the official Rust Language Server (RLS) or rust-analyzer, depending on the user's preference. If you don't have it installed, the extension will install it for you (with permission)."

Visual Studio Code interface showing the Rust extension marketplace. The search results list several Rust-related extensions, with the main extension, "Rust" (rust-lang.rust), selected. The details for the "Rust" extension are displayed on the right, including its description, version (0.7.8), and installation options. The extension is described as "The Rust Programming Language" and "Rust for Visual Studio Code (powered by Rust Language Server)". It is marked as a "Preview" extension and is currently installed. The details section includes a "Rust support for Visual Studio Code" section with a "Visual Studio Marketplace" badge for version v0.7.8, which is marked as "build passing". Below this, it lists supported features: code completion, jump to definition, peek definition, find all references, symbol search, types and documentation on hover, code formatting, refactoring (rename, deglob), error squiggles and apply suggestions from errors, snippets, and build tasks. A note states: "Rust support is powered by a separate language server - either by the official Rust Language Server (RLS) or rust-analyzer, depending on the user's preference. If you don't have it installed, the extension will install it for you (with permission)."



Visual Studio Code interface showing the CodeLLDB extension marketplace. The search results list the "CodeLLDB" extension (vadimcn.vscodellldb) by Vadim Chugunov. The details for the "CodeLLDB" extension are displayed on the right, including its description, version (1.6.1), and installation options. The extension is described as "Native debugger based on LLDB." and is currently installed. The details section includes a "Features" section with a list of capabilities: Debugging on Linux (x64 or ARM), macOS and Windows; Conditional breakpoints, function breakpoints, data breakpoints, logpoints; Launch debuggee in integrated or external terminal; Disassembly view with instruction-level stepping; Loaded modules view; Python scripting; HTML rendering for advanced visualizations; Rust language support with built-in visualizers for vectors, strings and other standard types; Global and workspace defaults for launch configurations; Remote debugging; Reverse debugging (experimental, requires compatible backend). A note states: "* DWARF debug info format recommended, limited support for MS PDB." Below the features list, it says "For full details please see the User's Manual."

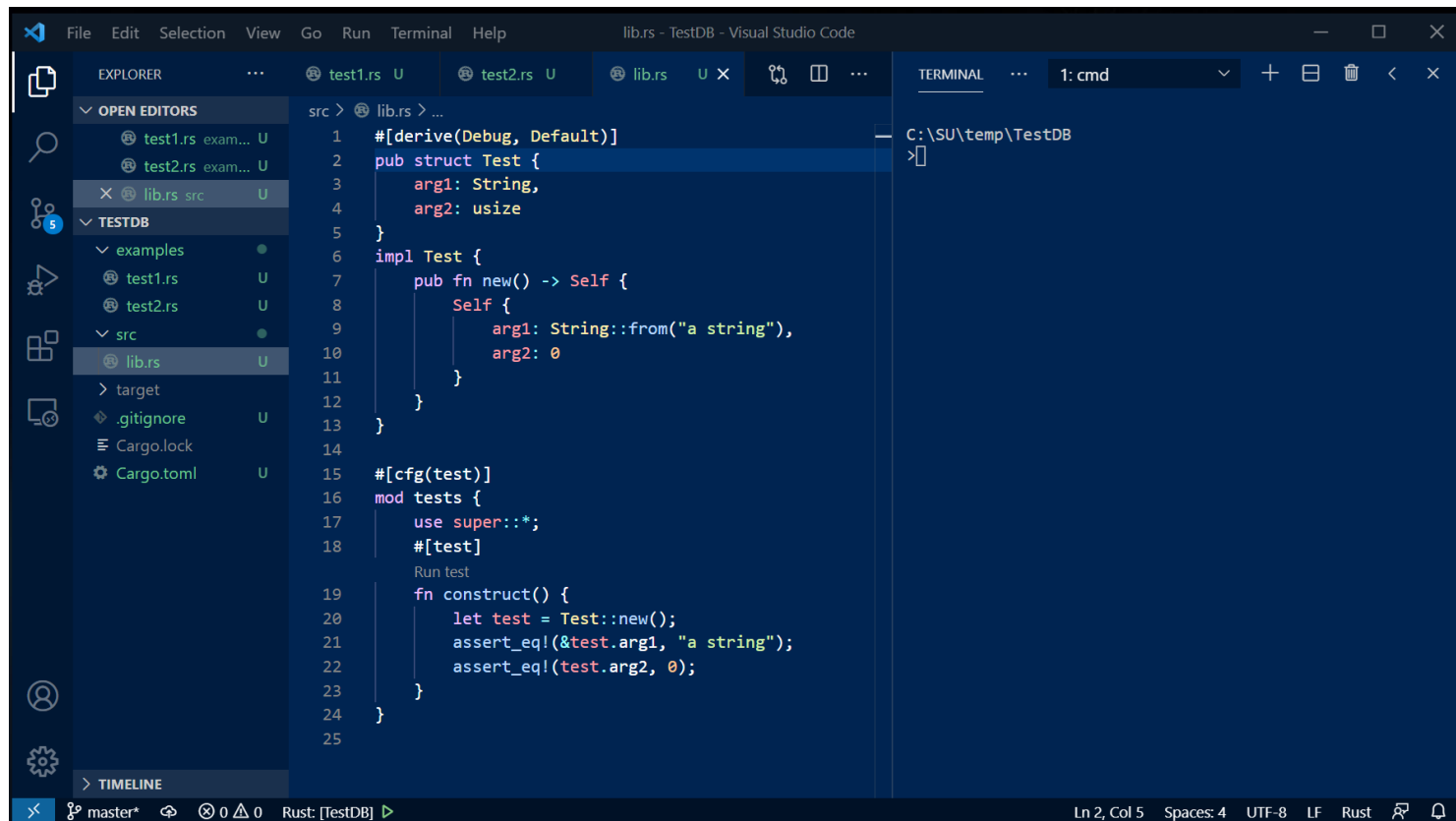
Visual Studio Code interface showing the CodeLLDB extension marketplace. The search results list the "CodeLLDB" extension (vadimcn.vscodellldb) by Vadim Chugunov. The details for the "CodeLLDB" extension are displayed on the right, including its description, version (1.6.1), and installation options. The extension is described as "Native debugger based on LLDB." and is currently installed. The details section includes a "Features" section with a list of capabilities: Debugging on Linux (x64 or ARM), macOS and Windows; Conditional breakpoints, function breakpoints, data breakpoints, logpoints; Launch debuggee in integrated or external terminal; Disassembly view with instruction-level stepping; Loaded modules view; Python scripting; HTML rendering for advanced visualizations; Rust language support with built-in visualizers for vectors, strings and other standard types; Global and workspace defaults for launch configurations; Remote debugging; Reverse debugging (experimental, requires compatible backend). A note states: "* DWARF debug info format recommended, limited support for MS PDB." Below the features list, it says "For full details please see the User's Manual."

Create new Rust package

```
Windows PowerShell x + v - □ x
> cargo new TestDB --lib --name test_db
Created library `test_db` package
C:\su\temp\
C:\su\temp\TestDB\
> mkdir examples
```

- Create lib
- Create examples subdirectory
- Add test1.rs and test2.rs
- Add a new type in the library
- Add a unit test
- There is no .vs directory since we have not opened in VS Code.

Project State before starting debug session

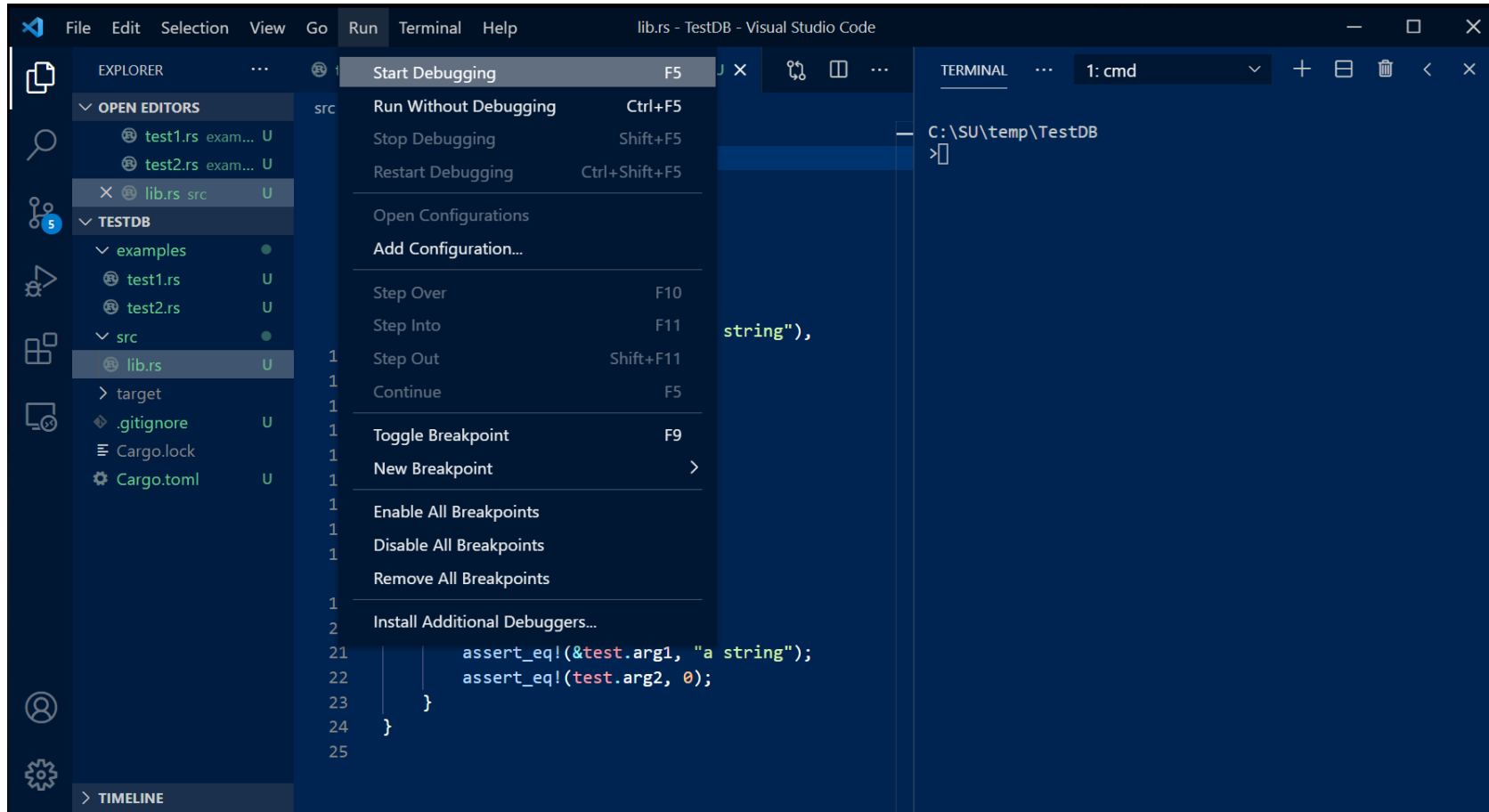


```
src > @ lib.rs > ...
1  #[derive(Debug, Default)]
2  pub struct Test {
3      arg1: String,
4      arg2: usize
5  }
6  impl Test {
7      pub fn new() -> Self {
8          Self {
9              arg1: String::from("a string"),
10             arg2: 0
11         }
12     }
13 }
14
15 #[cfg(test)]
16 mod tests {
17     use super::*;
18     #[test]
19     Run test
20     fn construct() {
21         let test = Test::new();
22         assert_eq!(&test.arg1, "a string");
23         assert_eq!(test.arg2, 0);
24     }
25 }
```

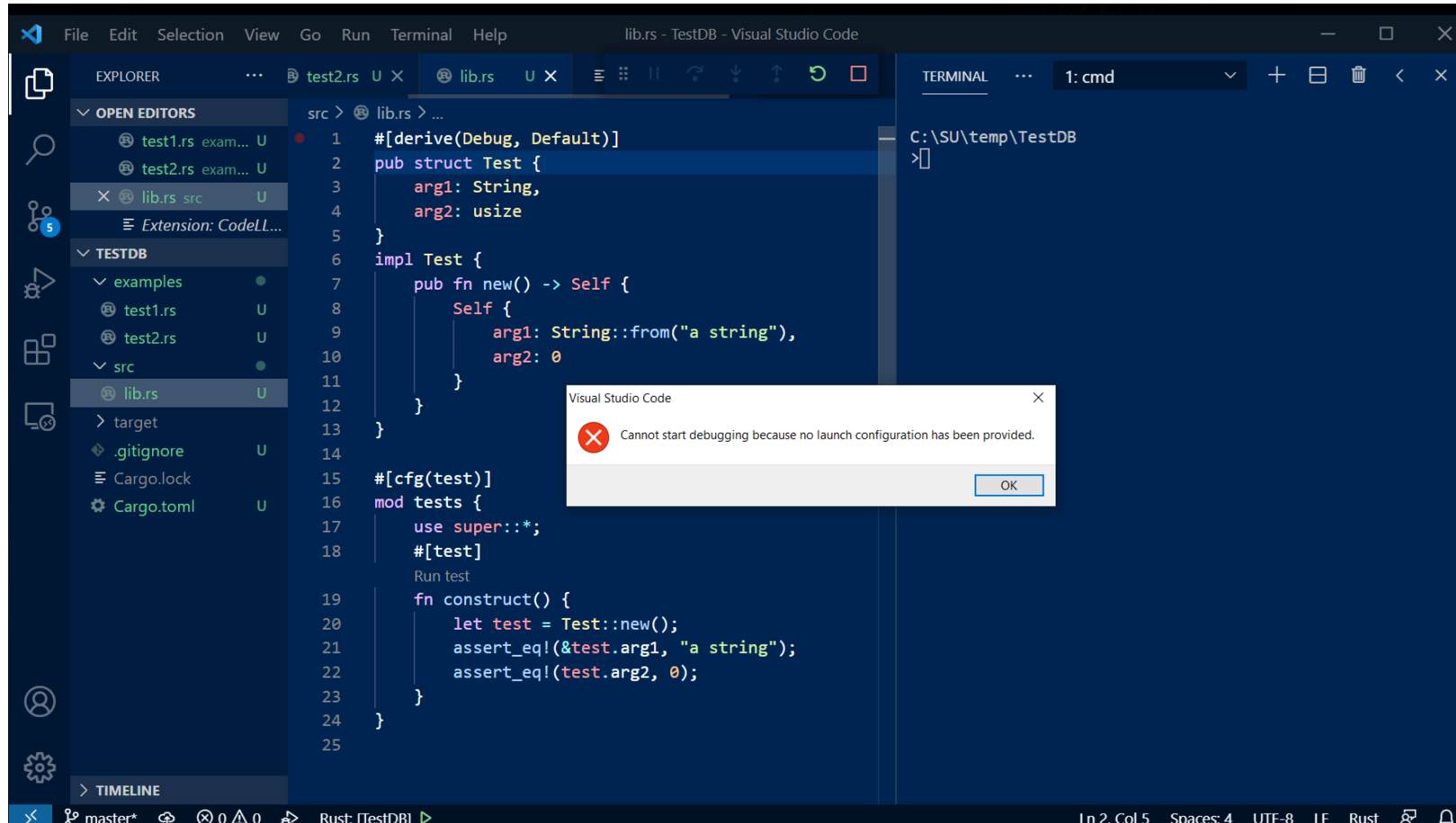
C:\SU\temp\TestDB
>

- Almost all of this code developer enters.
- Wizard only generates a basic test hook.

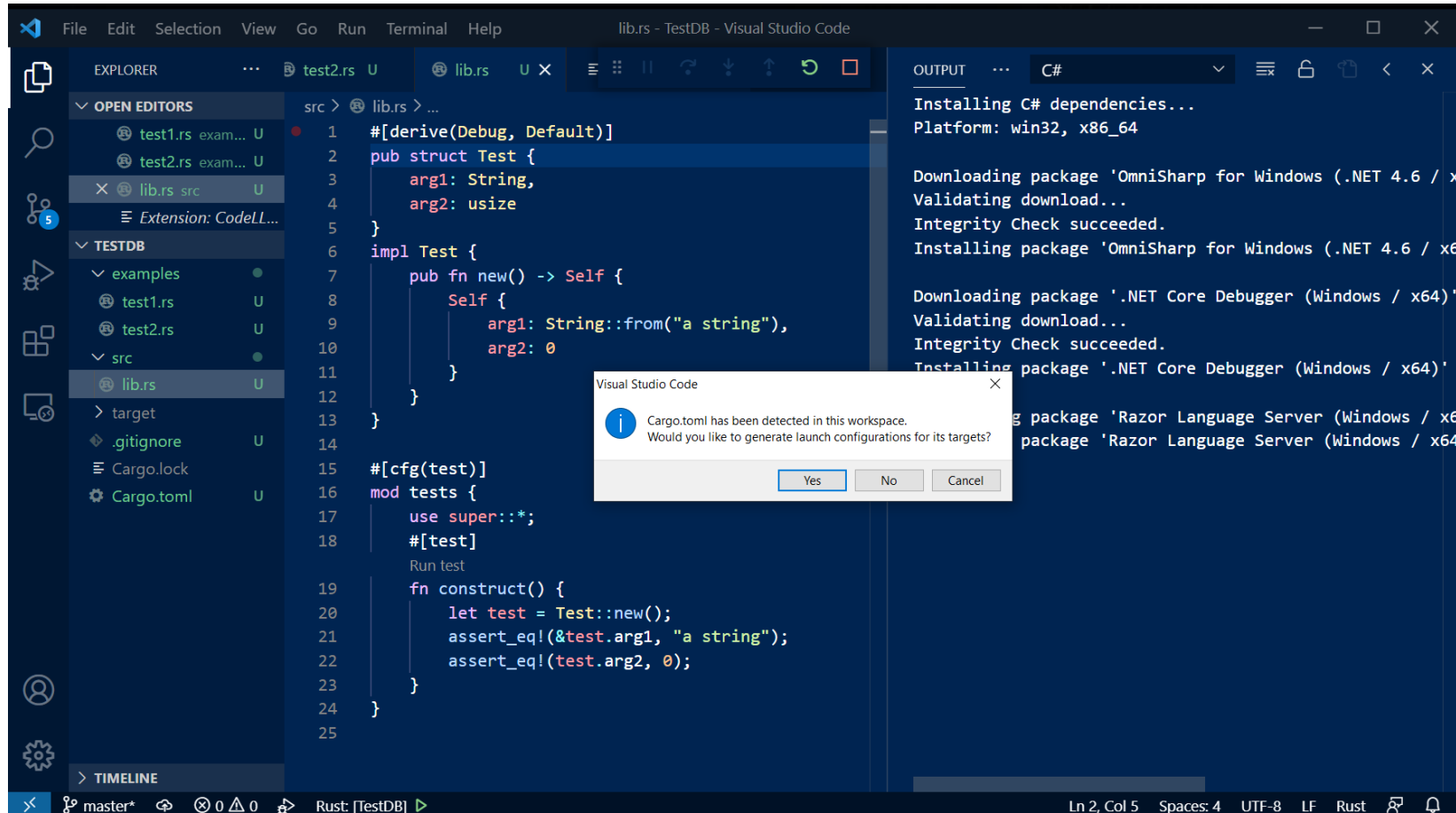
Start First Debug Session



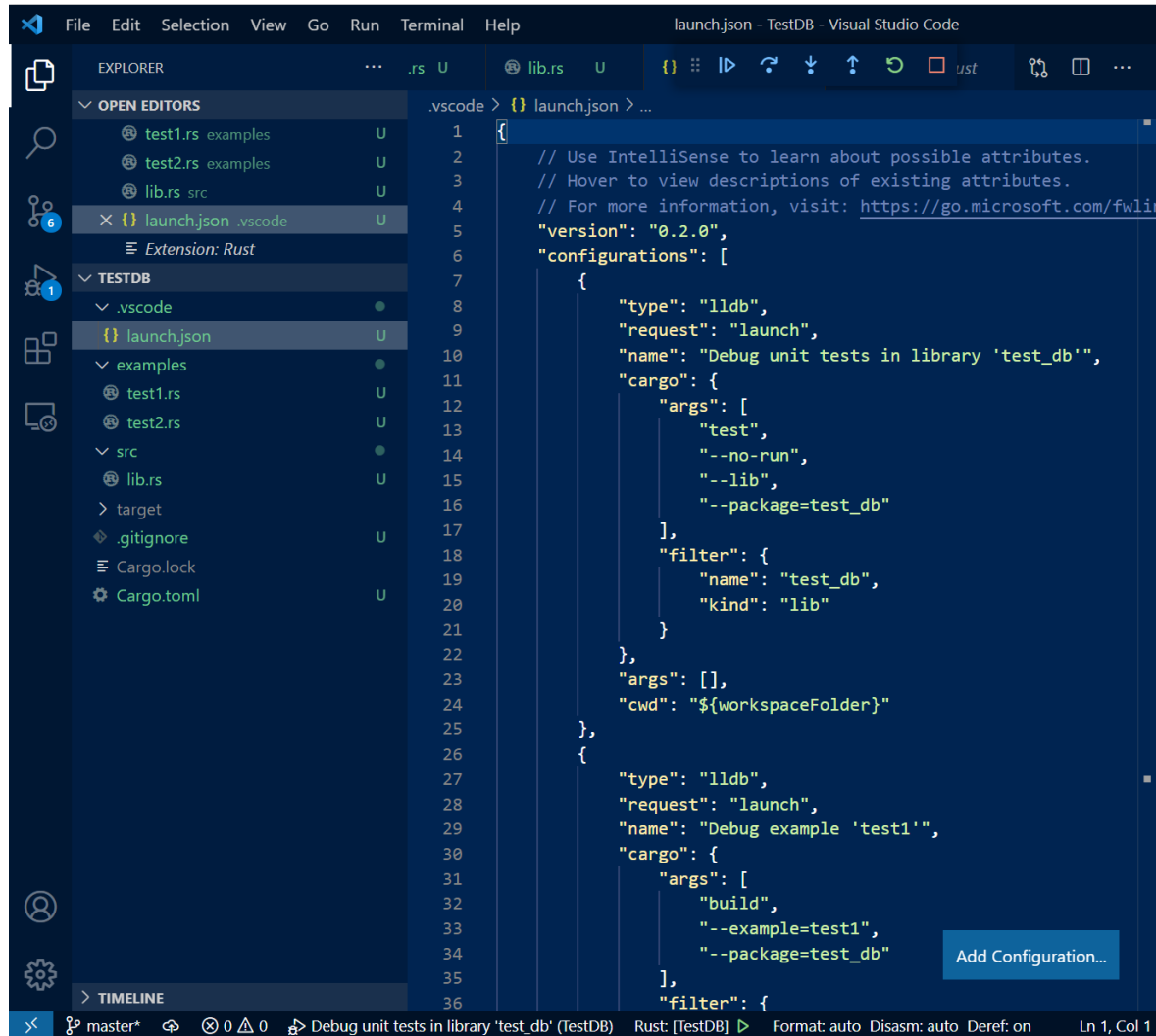
There is no launch.json so can't start



After clicking ok a new prompt appears



Clicking yes creates a launch.json



The screenshot shows the Visual Studio Code interface with the Explorer view on the left and the Editor view on the right. The Explorer view shows the project structure with the following folders and files:

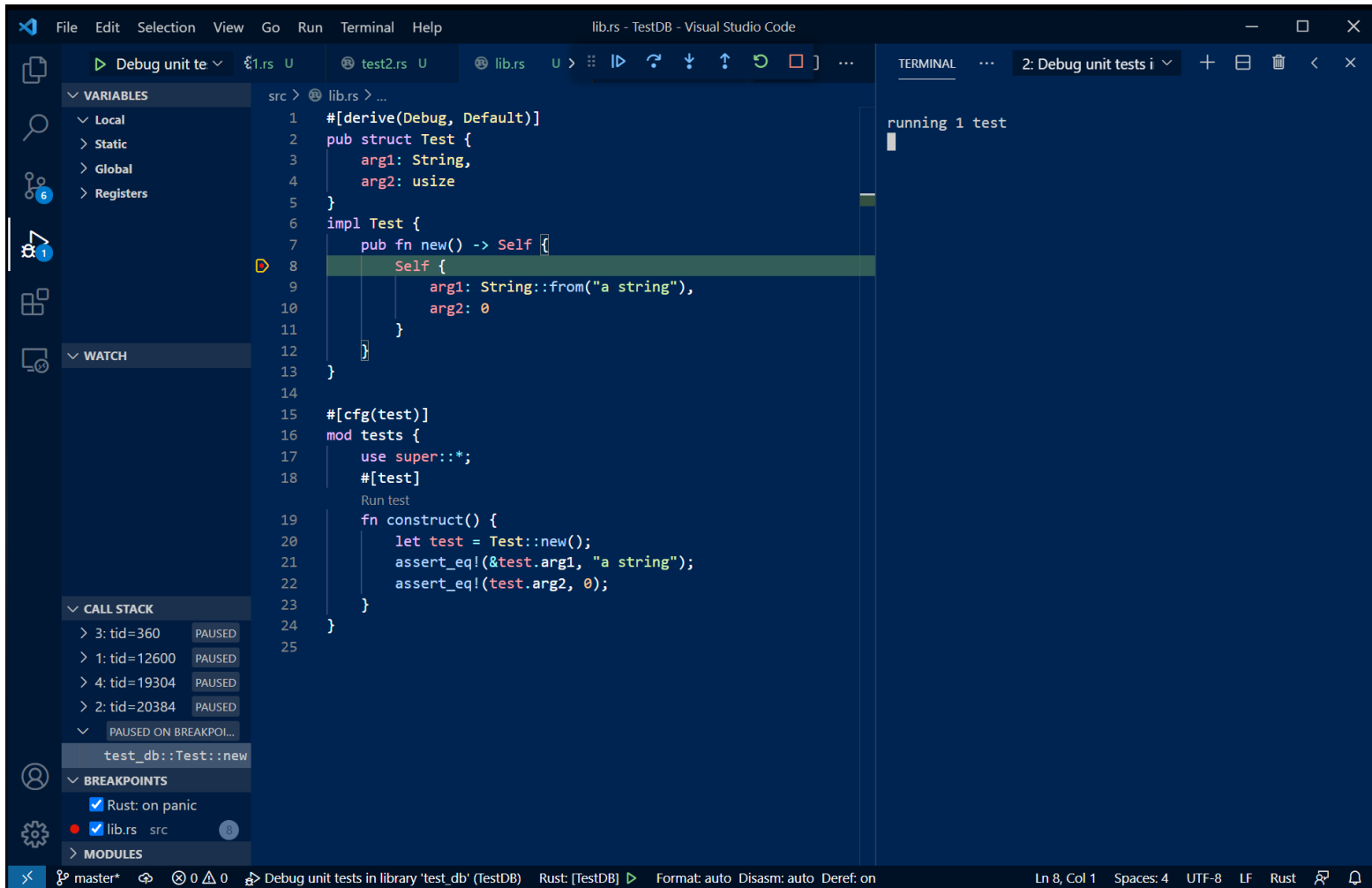
- test1.rs examples
- test2.rs examples
- lib.rs src
- launch.json .vscode
- Extension: Rust
- TESTDB
 - .vscode
 - launch.json
 - examples
 - test1.rs
 - test2.rs
 - src
 - lib.rs
 - target
 - .gitignore
 - Cargo.lock
 - Cargo.toml

The Editor view shows the content of the launch.json file:

```
1 {
2     // Use IntelliSense to learn about possible attributes.
3     // Hover to view descriptions of existing attributes.
4     // For more information, visit: https://go.microsoft.com/fwlink/?linkid=829996
5     "version": "0.2.0",
6     "configurations": [
7         {
8             "type": "lldb",
9             "request": "launch",
10            "name": "Debug unit tests in library 'test_db'",
11            "cargo": {
12                "args": [
13                    "test",
14                    "--no-run",
15                    "--lib",
16                    "--package=test_db"
17                ],
18                "filter": {
19                    "name": "test_db",
20                    "kind": "lib"
21                }
22            },
23            "args": [],
24            "cwd": "${workspaceFolder}"
25        },
26        {
27            "type": "lldb",
28            "request": "launch",
29            "name": "Debug example 'test1'",
30            "cargo": {
31                "args": [
32                    "build",
33                    "--example=test1",
34                    "--package=test_db"
35                ],
36                "filter": {
```

- The Rust and CodeLLDB plugins conspire to populate the launch configurations from contents of the project directories.
- Now, debugging works “out of the box”.
 - You may need to add command line arguments.

Now Debugging Works!



```
src > lib.rs > ...
1  #[derive(Debug, Default)]
2  pub struct Test {
3      arg1: String,
4      arg2: usize
5  }
6  impl Test {
7      pub fn new() -> Self {
8          Self {
9              arg1: String::from("a string"),
10             arg2: 0
11         }
12     }
13 }
14
15 #[cfg(test)]
16 mod tests {
17     use super::*;
18     #[test]
19     Run test
20     fn construct() {
21         let test = Test::new();
22         assert_eq!(&test.arg1, "a string");
23         assert_eq!(test.arg2, 0);
24     }
25 }
```

running 1 test

test_db::Test::new

Rust: on panic

lib.rs src

master* 0 0 0 Debug unit tests in library 'test_db' (TestDB) Rust: [TestDB] Format: auto Disasm: auto Deref: on Ln 8, Col 1 Spaces: 4 UTF-8 LF Rust

- You can set breakpoints
- Step: over, into, out of
- Restart
- Run to completion

Data Visualizers – you will need to experiment

The screenshot displays the Visual Studio Code IDE with a Rust project named 'lib.rs - TestDB'. The editor shows the following code:

```
1  #[derive(Debug, Default)]
2  pub struct Test {
3      arg1: String,
4      arg2: usize
5  }
6  impl Test {
7      pub fn new() -> Self {
8          Self {
9              arg1: String::from("a string"),
10             arg2: 0
11         }
12     }
13 }
14
15 #[cfg(test)]
16 mod tests {
17     use super::*;
18     #[test]
19     Run test
20     fn construct() {
21         let test = Test::new();
22         assert_eq!(&test.arg1, "a string");
23         assert_eq!(test.arg2, 0);
24     }
25 }
```

The left sidebar contains several panels:

- VARIABLES:** Shows a local variable `test: {arg2:0}` with values `arg1: "a string"` and `arg2: 0`. It also shows `left_val`, `right_val`, `arg0`, and `arg1`.
- CALL STACK:** Shows the current call stack with the top frame being `test_db::tests::construct` at line 19, which is paused on step 21.
- BREAKPOINTS:** Shows a breakpoint set at `lib.rs src` on line 8.

The right sidebar shows the **TERMINAL** panel with the output: `running 1 test` and a cursor on the first line.

The status bar at the bottom indicates the current position: `Ln 21, Col 1`, `Spaces: 4`, `UTF-8`, `LF`, `Rust`, and `Format: auto`.

That's all Folks!