
# Using Bazel with Rust to Build and Deploy an Application

28 minute read  Updated: July 11, 2023

Enoch Chejieh

In this Series

Table of Contents

**The article discusses integrating Bazel with Rust for improved build speeds. Earthly provides caching mechanisms that can accelerate build times for Rust developers. Check it out.**

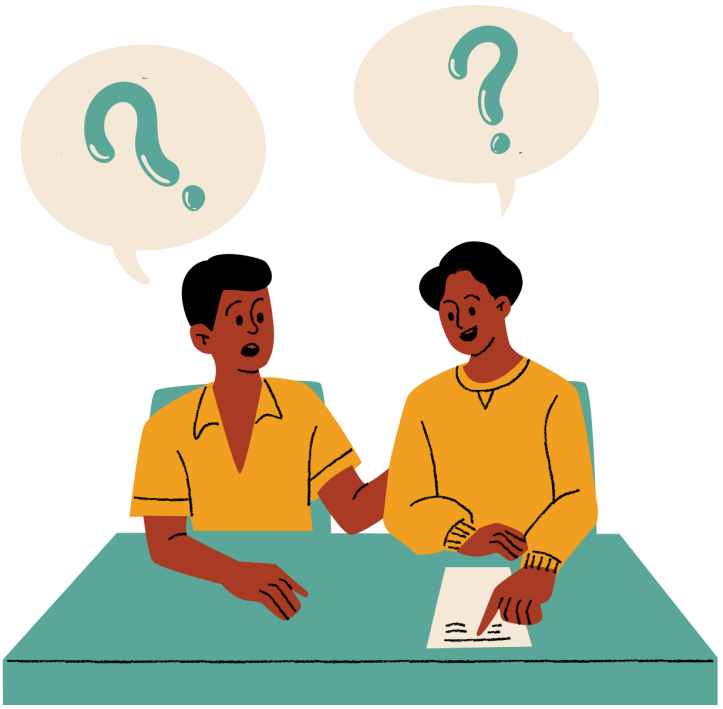
Building and deploying software can be a complicated and time-consuming process, especially as applications grow in size and complexity. One tool that can help simplify this process is Bazel, an open-source build tool developed by Google. Bazel is designed to make it easy to build and test large and complex codebases and is particularly well-suited for monorepos, which are codebases that contain multiple projects or components.

One of the key features of Bazel is its ability to speed up builds and tests. Bazel's caching and dependency analysis features facilitate fast, incremental builds. This makes it possible to quickly iterate on code changes, which can be especially usef for large teams working on a codebase. Additionally, Bazel supports multiple

languages and platforms, including Rust, and can be extended to support new languages.

In this article, you'll learn how to prepare your workspace, run, and test your code, and develop a basic application using Rust with Bazel. By the end of this article, you'll know how to use Bazel to streamline your development workflow and improve the efficiency of your builds and tests.

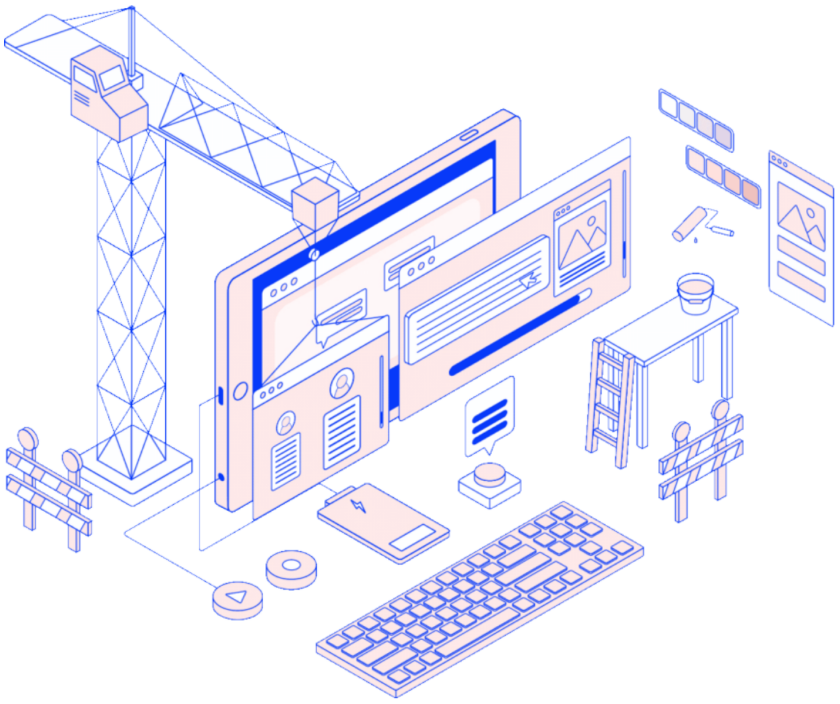
## How Rust and Bazel Work Together



Bazel supports Rust through its built-in rules, which are sets of instructions that tell Bazel how to build and test code written in specific languages. These rules allow you to build and test Rust code using Bazel's powerful execution capabilities.

In addition, Bazel's Rust rules provide a set of common macros, which make it easy to perform common tasks, such as building libraries, running tests, and creating binaries. These macros can be used to simplify the development process and reduce the amount of boilerplate code that you have to write.

# Build and Develop a Rust Application with Bazel



Now that you know how Rust and Bazel work together, let's create a basic Rust application that makes use of a custom substring Rust library that you'll build and deploy using Bazel.

All the source code for this tutorial is available in this GitHub repository.

## Install Rust

To install and set up Rust, you need to start by installing rustup, the official Rust installer. You can do so with the following command:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

This will download and run the rustup installer, which will guide you through the installation process.

Once rustup is installed, you can use it to install the latest stable version of Rust by running the following command:

```
rustup install stable
```

## Install Bazel

Next, you need to install and set up Bazel by following the instructions on the Bazel website for your operating system. If you're using Windows, Linux, or macOS, Bazel recommends installing it using Bazelisk, which is useful for switching between different versions of Bazel and for keeping it updated to the latest release.

## Write a Basic Rust App

Once Bazel is installed, you need to create a new Rust project called `rs-bazel` by running the following command:

```
cargo new rs-bazel
```

Then change the directory to your newly created project:

```
cd rs-bazel
```

For this demonstration, you need to create a Rust crate inside your project. To do so, run the following command in your current directory:

```
cargo new --lib substring-library
```

Here, you create the `substring-library` crate.

After following these steps, your file structure should look like this:

```
[rs-bazel]
   src/
      - main.rs
   substring-library/
      src/
         - lib.rs
      Cargo.toml
```

Cargo.toml

Open your **lib.rs** file and add the following code:

lib.rs Copy

```
// lib.rs

pub fn find_substring<'a>(s: &'a str, substring: &str) -> Option<&'a str> {
    s.find(substring).map(|i| &s[i..i + substring.len()])
}

pub fn replace_substring(s: &str, from: &str, to: &str) -> String {
    s.replace(from, to)
}
```

The **find_substring** function takes in two parameters: **s**, a string slice with a lifetime **'a**, and **substring**, a reference to a string slice. It returns an **Option<&'a str>** type if the first occurrence of the **substring** was found within the **s** string; otherwise, it returns **None**.

The **replace_substring** function takes in three parameters: references to string slices **s**, **from**, and **to**. It replaces all occurrences of the **from** string with the **to** string and returns a **String** type.

Next, you need to add tests for your crate. To do so, add the following lines of code to the **lib.rs** file:

lib.rs Copy

```
// lib.rs

#[cfg(test)]
mod tests {
    use crate::{find_substring, replace_substring};

    #[test]
    fn find() {
        let s = "Dragons fly!";
        let substring = find_substring(s, "fly");
```

```
        assert_eq!(substring, Some("fly"));
    }

    #[test]
    fn replace() {
        let s = "Hello, World!";
        let new_string = replace_substring(s, "World", "Rust");
        assert_eq!(new_string, "Hello, Rust!");
    }
}
```

This test code checks the expected output of the **`find_substring()`** and **`replace_substring()`** functions, which are expected to return the first occurrence of a substring in a string and replace the substring, respectively.

Next, navigate to your **`main.rs`** file, which is the entry point of your Rust project, and add the following code:

main.rs Copy

```
// main.rs

extern crate substring_library;

use substring_library::{find_substring, replace_substring};

fn main() {
    let s = "Hello, World!";

    let substring = find_substring(s, "World");
    println!("Found substring: {:?}", substring);

    let new_string = replace_substring(s, "World", "Rust");
    println!("New string: {}", new_string);
}
```

This code uses the **`substring_library`** crate you just created, which contains the **`find_substring`** and **`replace_substring`** functions to find a substring within a string and replace a substring with another one, respectively. It uses the **`println!`** macro to output the results.

At this point, you've created your Rust program. Now, you need to set up your Bazel environment to be able to test, build, and deploy your program.

## Build and Test With Bazel

To build and test with Bazel, create a **`WORKSPACE`** file in your root directory. Your file structure should look like this:

WORKSPACE Copy

```
[rs-bazel]
   src/
      - main.rs
   substring-library/
      src/
         - lib.rs
      Cargo.toml
   Cargo.toml
   WORKSPACE
```

The **`WORKSPACE`** file in Bazel serves as the root of your project. It defines the root and specifies the external dependencies that your project relies on. It's like a map that guides Bazel in finding all the necessary files and dependencies for your project. It helps Bazel know where to start building your project and ensures that all the dependencies are in place.

Now, create a **`WORKSPACE`** file in your current directory and add this code to it:

WORKSPACE Copy

```
# ./WORKSPACE

load("@bazel_tools//tools/build_defs/repo:http.bzl", "http_archive")

# Downloads and extracts a compressed archived file from the \
specified URL and creates a new repository rule with the given name.
http_archive(
    name = "rules_rust",

    # Specifies the SHA-256 hash of the tar file. This is used to \
```

```
    verify the integrity of the downloaded tar file.
    sha256 = "aaaa4b9591a5dad8d8907ae2dbe6e0eb49e6314946ce4c7149241648e56a12

    # Specifies the URL of a compressed archived file to download.
    urls = ["https://github.com/bazelbuild/rules_rust/releases/download/0.16
)
```

The **load** function is responsible for importing the necessary custom functions, macros, and logic needed in your **BUILD** file and **.bazelrc**, which you will learn more about later in this tutorial.

The **http_archive** function is used for downloading a Bazel repository as a compressed archive file, decompresses it, and makes its targets available for binding, which in this case is the rules for Rust that you'll be using in your Bazel environment.

Next, in the same **WORKSPACE** file, add the following code:

WORKSPACE Copy

```
# ./WORKSPACE

# Loads the `rules_rust_dependencies` and `rust_register_toolchains` \
function definitions.
load("@rules_rust//rust:repositories.bzl", "rules_rust_dependencies", \
"rust_register_toolchains")
```

This loads the **rule_rust_dependencies** and **rust_register_toolchains** functions. Bazel uses the **rule_rust_dependencies** function to know what dependencies your project needs to successfully run, build, or test your application.

In the same **WORKSPACE** file, add the following line of code:

WORKSPACE Copy

```
# WORKSPACE

# Adds the necessary dependencies for the Rust rules.
rules_rust_dependencies()
```

The **rust_register_toolchains** function registers the Rust toolchains with the given

versions and editions you will need to use within your project.

And, in the same **`WORKSPACE`** file, add this line of code:

WORKSPACE Copy

```
# ./WORKSPACE

# Registers Rust toolchains with the given versions and editions.
rust_register_toolchains(
    versions=["1.66.0"],

    # Specifies the Rust edition to use for the registered toolchains
    edition = "2021",
)
```

When another developer clones the project and runs the **`bazel build`** command, Bazel will check for the Rust version specified in the **`rust_register_toolchains`** , and if the correct version of Rust isn't already installed on the local system, Bazel will download and install it before building the project.

> It's important to keep the Rust version in sync across all the developers working on the project, as different versions of Rust can cause compatibility issues and build errors.

### Using Labels to Identify and Build Specific Targets

In addition to specifying the correct version of Rust, Bazel also uses **`labels`** to identify and build specific targets within the project. A **`label`** is a unique identifier that points to a specific target, such as a binary or a library within the project.

**`Labels`** have the following format: **`//package:target`** , where **`package`** refers to the package or directory containing the target and **`target`** refers to the specific target within that package.

Bazel uses **`labels`** to determine which targets to build or run as well as to resolve dependencies between targets. By specifying the correct **`labels`** , developers can easily build and run specific parts of their projects without having to navigate through the entire codebase.

When building a Rust project with Bazel, the following command is used for building and running build targets. Try running this command in the terminal inside your current directory:

```
bazel run //:rs_bazel
```

Here, the `//` signifies the root directory of the project. **`rs_bazel`** signifies the Rust binary target defined in the **`BUILD`** file within your root directory.

Alternatively, you could also run the **`bazel build //:rs_bazel`** command, but this only compiles the target. The main difference between the **`bazel build`** and **`bazel run`** commands is that **`bazel build`** compiles the code, while **`bazel run`** runs the compiled binary. The **`bazel run`** command also builds the code if it hasn't been built yet.

Now, notice the error output from your terminal:

```
Starting local Bazel server and connecting to it...
ERROR: Skipping '//:rs-bazel': no such package '': BUILD file not \
found in any of the following directories. Add a BUILD file to a \
directory to mark it as a package.
 - /Users/apple/Documents/rs-bazel
WARNING: Target pattern parsing failed.
ERROR: no such package '': BUILD file not found in any of the \
following directories. Add a BUILD file to a directory to \
mark it as a package.
 - /Users/apple/Documents/rs-bazel
INFO: Elapsed time: 4.276s
INFO: 0 processes.
FAILED: Build did NOT complete successfully (0 packages loaded)
ERROR: Build failed. Not running target
```

Output

As you can see, your Rust application **`BUILD`** file is missing.

The **`BUILD`** is a critical part of the Bazel build system because the **`BUILD`** file is used to describe the components of your application and how they should be built by Bazel and, in this case, your Rust application.

### Set Up Bazel to Build and Deploy Your Rust Application

And now comes the exciting part! You're going to set up Bazel to build and deploy your basic Rust application with the custom substring Rust library you previously created.

To start, create a **`BUILD`** file in the root of your project and paste the following code in it:

BUILD.bazel Copy

```
# ./BUILD

# Loads the Rust rules and the `rust_binary` function definition.
load("@rules_rust//rust:defs.bzl", "rust_binary")
```

This line of code loads the **`defs.bzl`** file from the **`@rules_rust//rust`** package, specifically the **`rust_binary`** function definition. The **`rust_binary`** function is used to define a Rust binary target that can be built by Bazel. This function can be used to specify the dependencies, settings, and other information needed for building the Rust binary.

Now, copy, and paste the following code into the **`BUILD`** file:

BUILD.bazel Copy

```
# ./BUILD

# Declares a Rust binary target with the given name.
rust_binary(
    name = "rs_bazel",

    # Specifies the source file for the binary.
    srcs = ["src/main.rs"],

    # Specifies dependencies for the binary.
    deps = [
        # Depend on the `substring_library` target, which is \
        the crate you created.
        '//substring-library:substring_library'
    ],
```

```
    # Specifies the Rust edition to use for this binary.
    edition = "2021"
)
```

Before running Bazel again, create another **`BUILD`** file within your **`substring-library`** crate directory. Your file structure should look similar to this:

```
[rs-bazel]
   src/
      - main.rs
   substring-library/
      src/
         - lib.rs
      BUILD
      Cargo.toml
   BUILD
   Cargo.toml
   WORKSPACE
```

Then add the following code in the **`BUILD`** file you just created within your **`substring-library`** crate directory:

BUILD.bazel Copy

```
# ./substring-library/BUILD

load("@rules_rust//rust:defs.bzl", "rust_library", "rust_test")
```

This is similar to what you did earlier when you imported the **`rust_binary`** function, but this time, you're defining your crate and its tests using the **`rust_library`** and **`rust_test`** functions. The **`rust_library`** and **`rust_test`** functions are used for defining a Rust library and Rust test target that can be built by Bazel.

### Verify Your Bazel Build Is Working Correctly

To ensure your **`substring-library`** crate and it's tests are included in your Bazel build, copy and paste the following code in the **`BUILD`** file in that same crate directory:

BUILD.bazel Copy

```
# ./substring-library/BUILD

# Declares a Rust library target with the given name.
rust_library(
    name = "substring_library",

    # Specifies the source files for the library.
    srcs = ["src/lib.rs"],

    # Specifies the Rust edition to use for this library.
    edition = "2021"
)

# Declares a Rust test target with the given name.
rust_test(
    name = "substring_library_test",

    # Specifies the source file for the test.
    srcs = ["src/lib.rs"],

    # Specifies dependencies for the test.
    deps = [
        # Depend on the library we just declared.
        ":substring_library",
    ],

    # Specifies the Rust edition to use for this test.
    edition = "2021"
)
```

In your terminal, try running Bazel again with the following command:

```
bazel run //:rs_bazel
```

>_

You should get the following error:

Outp

```
ERROR: /Users/apple/Documents/rs-bazel/BUILD:8:12: in rust_binary \
```

```
rule //:rs_bazel: target '//substring-library:substring_library' is \
not visible from target '//:rs_bazel'. Check the visibility declaration \
of the former target if you think the dependency is legitimate
ERROR: /Users/apple/Documents/rs-bazel/BUILD:8:12: Analysis of target \
'//:rs_bazel' failed
ERROR: Analysis of target '//:rs_bazel' failed; build aborted:
INFO: Elapsed time: 0.264s
INFO: 0 processes.
FAILED: Build did NOT complete successfully (1 packages loaded, \
2 targets configured)
ERROR: Build failed. Not running target
```

By default, all targets have their visibility set to private, meaning that only rules within the same package can depend on them. So when you declared your Rust binary target dependencies to depend on the **`substring-library`** crate you created, Bazel returned errors because it doesn't have access to the crate and is private by default.

Now, copy, and paste the following code directly below your load function within the **`substring-library`** crate **`BUILD`** file:

BUILD.bazel Copy

```
# Set the default visibility for the package to be public.
package(default_visibility = ["//visibility:public"])
```

This code sets the default visibility for the package to public. In Bazel, visibility controls which rules and targets can depend on a given target. By setting the default visibility to public, it allows rules in other packages to depend on the targets defined in this package, as long as they are not explicitly marked as private. This can be useful if you want to make the targets in this package available to other parts of your codebase.

Now, try running the **`bazel run //:rs_bazel`** command again. You should see the following outputs:

Output

```
INFO: Analyzed target //:rs_bazel (0 packages loaded, 0 targets configured).
INFO: Found 1 target...
Target //:rs_bazel up-to-date:
```

```
  bazel-bin/rs_bazel
INFO: Elapsed time: 0.149s, Critical Path: 0.00s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Running command line: bazel-bin/rs_bazel
Found substring: Some("World")
New string: Hello, Rust!
```

And your application has been built successfully!

One of the key features of Bazel is its ability to cache build results so that it can avoid rebuilding parts of the project that haven't changed.

As Bazel builds a project, it keeps track of the inputs and outputs of each build step. When it encounters a build step that it has previously performed, it checks the inputs and outputs against the cached results to see if they are the same. If they are the same, Bazel can reuse the cached output rather than perform the build step again. This can save a significant amount of time, especially for large projects with many dependencies.

Overall, Bazel caching helps to improve the build performance and the repeatability of your project.

## Test Your Application

Now, for testing, run the following command in your terminal to run the tests present within your crate:

```
bazel test //substring-library:substring_library_test
```

>_

When using Bazel to test a Rust project, the command **`bazel test //package:target`** is used to run tests for a specific crate; in this case, **`bazel test //substring-library:substring_library_test`**. Here, the **`//`** signifies the root directory of the project. **`substring-library`** is the name of the crate folder, and **`substring_library_test`** is the specific test target defined in the **`BUILD`** file within that crate.

After you've built your application, Bazel creates the following files within your root directory:

- The **`bazel-bin`** directory contains the compiled binary files of your targets, including the stand-alone binary.
- The **`bazel-out`** directory contains the output files and dependency information of your build process.
- The **`bazel-testlogs`** directory contains the logs generated during test runs. This includes the test output and the results of the test run.
- The **`bazel-rs-bazel`** directory contains the files that are specific to the **`rs_bazel`** target. This is where the final binary of your target will be located.

The build executable generated after running **`bazel run`** can be found in the **`bazel-bin`** directory. This executable, named **`rs_bazel`**, can be run just like a normal Rust program. Additionally, it can also be distributed as a stand-alone binary, eliminating the need for dependencies or Bazel installation on other machines.

In your terminal, you can run the executable by typing **`./bazel-bin/rs_bazel`**. This will execute the binary created by Bazel, and you should see the desired output:

```
Found substring: Some("World")
```

Output

```
New string: Hello, Rust!
```

## Conclusion

In this article, you learned how Bazel can be used to speed up the build and deployment process of a Rust application while still leveraging the features of the Rust language. You also saw how to set up a workspace and then run and test a Rust application using Bazel's rules for Rust.

Yet, while Bazel is a fantastic tool for building Rust apps, it can also be complex and intricate. It may be overkill for smaller projects or for teams that aren't familiar with its intricacies. That's where Earthly comes into the picture.

Earthly offers a simpler approach to building monorepos and containerization, focusing on streamlining the build process, maintaining a minimal setup, and promoting the use of best practices. It aims to simplify the build system and make it accessible for more developers, offering a potentially lower learning curve compared to Bazel. Earthly can handle both small and large projects, offering you scalability without the additional complexity.

Remember, the ultimate goal is to choose a tool that not only suits your current needs but also has the capacity to grow with you and your project, all the while ensuring a simpler, faster, and more efficient software development process. Be it Bazel, Docker, Earthly, or any other tool, the choice should make your build process a breeze, not a hurdle.




**Enoch Chejieh**
Enoch Chejieh is a software engineer who enjoys teaching and sharing his knowledge with others.

*Writers at Earthly work closely with our talented editors to help them create high quality content. This article was edited by:*

## Bala Priya C

Bala is a technical writer who enjoys creating long-form content. Her areas of interest include math and programming. She shares her learning with the developer community by authoring tutorials, how-to guides, and more.

**Updated:** July 11, 2023

**Published:** May 23, 2023



## You May Also Enjoy

### Using Bazel with TypeScript

16 minute read

Learn how to use Bazel with TypeScript to build and test your projects faster and more efficiently. Discover the benefits of Bazel's advanced caching and par...

## How to Build Node.js Application with Bazel

6 minute read

Learn how to build a Node.js application with Bazel, an open-source build tool that speeds up builds and tests. This tutorial guides you through setting up t...