



# Java Magazine

Frameworks, GraalVM

## Use Python and R in your Java applications with GraalVM



[Tim Felgentreff](#) | January 21, 2022



[Stepan Sindelar](#) | January 21, 2022



[Lukas Stadler](#) | January 21, 2022



**GraalVM directly supports Python and R, and this opens the world of dynamic scripting and data science libraries to your Java projects.**

[GraalVM](#) is an open source project from Oracle that includes a modern just-in-time (JIT) compiler for Java bytecode, allows ahead-of-time (AOT) compilation of Java code into native binaries, provides a rich tooling ecosystem, and supports fast execution and integration of other languages.

GraalVM supports Python and R (among other languages), which opens the world of dynamic scripting and data science libraries to Java projects. Since scripting languages do not need a compilation step, applications written for GraalVM can allow users to easily extend them using any dynamic language supported by GraalVM. These user scripts are then JIT compiled and optimized with the Java code to provide optimal performance.

This article demonstrates how a Java application can be enhanced with the polyglot capabilities of

GraalVM, using a Micronaut chat application as an example.

## Get started with Micronaut

Micronaut is a microservice framework that works well with GraalVM. For the sake of clarity, this article will keep things simple by using one of the Micronaut examples, a [simple chat app using WebSockets](#). This app provides chat rooms with topics. A new user can join any chat room by just browsing at `http://localhost:8080/#{topicName}/{userName}`. Under the hood, the app is a WebSocket server.

Imagine you designed such an application for a customer and now this customer asks for rich, programmable customization options. Running on GraalVM, this can be very simple indeed. Look at the following message validation in the example chat app:

```
private Predicate<WebSocketSession> isValid(String topic) {  
    return s -> topic.equalsIgnoreCase(s.getUriVariables().get("topic", String  
}
```

[Copy code snippet](#)

Right now, the validation ensures that messages sent from one chat room are broadcast only to the users connected to the same chat room; that is, the topic that the sender is connected to matches that of the receiver.

The power of GraalVM's polyglot API makes this more flexible! Instead of a hardcoded predicate, write a Python function that will act as a predicate. Something like this will do exactly the same.

```
import java  
  
def is_valid(topic):  
    return lambda s: topic.lower() == s.getUriVariables().get("topic", java.lan
```

[Copy code snippet](#)

For testing it, put it in a `static String PYTHON_SOURCE` and then use it as follows:

```
private static final String PYTHON_SOURCE = /* ... */;  
private Context polyglotContext = Context.newBuilder("python").allowAllAccess
```

```
private Predicate<WebSocketSession> isValid(String topic) {
    polyglotContext.eval("python", PYTHON_SOURCE);
    Value isValidFunction = polyglotContext.getBindings("python").getMember("i
    return isValidFunction.execute(topic).as(Predicate.class);
}
```

[Copy code snippet](#)

That's it! So, what is happening here?

The Python code defines a new function, `is_valid`, which accepts an argument and defines a nested function that creates a closure over that argument. The nested function also accepts an argument, a `WebSocketSession`. Even though it is not typed, and that argument is a plain old Java object, you can call Java methods seamlessly from Python. Argument mapping from Python `None` to Java `null` is handled automatically by GraalVM.

Here, you'll see that the Java code has been extended with a private context field that represents the entry point into the polyglot world of GraalVM. You just create a new context with access to the Python language in which your script runs. However, because it's not wise to allow scripts to do whatever they want, GraalVM offers [sandboxing to restrict what a dynamic language can do at runtime](#). (More about that will come in a future article.)

The `isValid` function evaluates the Python code. Python code evaluated like this is run in the Python *global scope*, which you can access with the `getBindings` method. The GraalVM API uses the `Value` type to represent dynamic language values to Java, with various functions to read members, array elements, and hash entries; to test properties, such as whether a value is numeric or represents a string; or to execute functions and invoke methods.

The code above retrieves the `is_valid` member of the global scope. This is the Python function object you have defined, and you can execute it with the `topic` String. The result is again of type `Value`, but you need a `java.util.function.Predicate`. GraalVM has your back here, too, and allows casting dynamic values to Java interfaces. The appropriate methods are transparently forwarded to the underlying Python object.

## Wrapping Python scripts into beans

Of course, hardcoding Python code strings and manually instantiating polyglot contexts is not the proper way to structure a Micronaut app. A better approach is to make the context and engine objects available as beans, so they can be injected to other beans managed by Micronaut. Since you cannot annotate third-party classes, go about this by implementing a [Factory](#), as follows:

```
@Factory
public class PolyglotContextFactories {
```

```

@Singleton
@Bean
Engine createEngine() {
    return Engine.newBuilder().build();
}

@Prototype
@Bean
Context createContext(Engine engine) {
    return Context.newBuilder("python")
        .allowAllAccess(true)
        .engine(engine)
        .build();
}
}

```

[Copy code snippet](#)

Each instance of the GraalVM polyglot context represents an independent global runtime state. Different contexts appear as if you had launched separate Python processes. The `Engine` object can be thought of as a cache for optimized and compiled Python code that can be shared between several contexts and thus warmup does not have to be repeated as new contexts are created.

Micronaut supports injection of configuration values from multiple sources. One of them is `application.yml`, which you can use to define where the Python code lives, as follows:

```

micronaut:
  application:
    name: websocket-chat
  scripts:
    python-script: "classpath:scripts/script.py"

```

[Copy code snippet](#)

You can then access this script and expose its objects in another bean, as follows:

```

@Factory
@ConfigurationProperties("scripts")
public class PolyglotMessageHandlerFactory {
    private Readable pythonScript;

    public Readable getPythonScript() { return pythonScript; }
    public void setPythonScript(Readable script) { pythonScript = script; }
}

```

```

@Prototype
@Bean
MessageHandler createPythonHandler(Context context) throws IOException {
    Source source = Source.newBuilder("python", pythonScript.asReader(),
        context.eval(source);
    return context.getBindings("python").getMember("ChatMessageHandler").
}
}

```

---

[Copy code snippet](#)

As before, you should expect a certain member to be exported in the Python script, but this time you are casting it to a `MessageHandler` interface. The following is how that is defined:

```

public interface MessageHandler {
    boolean isValid(String sender, String senderTopic, String message, String

    String createMessage(String sender, String message);
}

```

---

[Copy code snippet](#)

You can use this interface in the chat component.

```

@ServerWebSocket("/ws/chat/{topic}/{username}")
public class ChatWebSocket {
    private final MessageHandler messageHandler;
    // ...
    public ChatWebSocket(WebSocketBroadcaster broadcaster, MessageHandler mes
        // ...
    }
    // ...
    @OnMessage
    public Publisher<String> onMessage(String topic, String username, String
        String msg = messageHandler.createMessage(username, message);
        // ...
    }
}

```

---

[Copy code snippet](#)

Phew! That was a lot. But now you have the flexibility to provide different scripts in the application config, and you'll have the polyglot context only created and the script only run as needed by the app. The contract for the Python script is simply that it must export an object with an `isValid` and a `createMessage` function to match the Java `MessageHandler` interface. Besides that, you can use the full power of Python in message transformations.

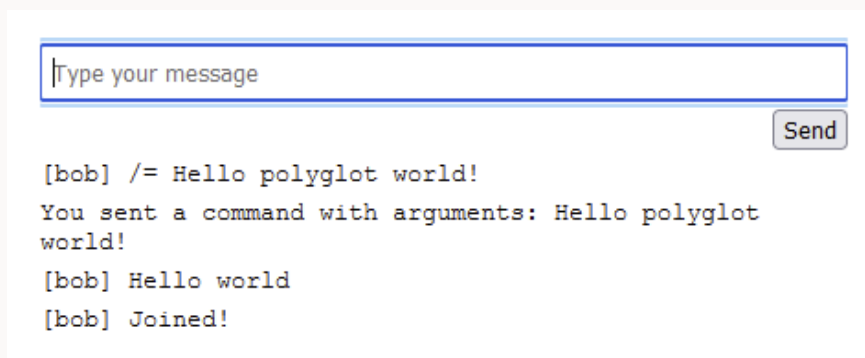
```
class ChatMessageHandler:
    def createMessage(sender, msg):
        match = re.match("/([^\ ]+)", msg)
        if match:
            try:
                if match.group(1) == "=":
                    return f"You sent a command with arguments: {msg.split()[1]}"
                else:
                    return "Unknown command!"
            except:
                return f"Error during command execution: {sys.exc_info()[0]}"
        return f"[{sender}] {msg}"

    def isValid(sender, senderTopic, message, receiver, receiverTopic):
        return senderTopic == receiverTopic
```

[Copy code snippet](#)

What's going on? This defines two functions on a class, `ChatMessageHandler`, to match the Java interface. The code exports the class *object* (not an instance!) to the polyglot bindings, so that the Java code can pick it up. The validation code is essentially the same just simplified due to different arguments.

The `createMessage` transformation, however, is more interesting. Many chat systems treat a leading slash / specially, allowing bots or extended chat commands to be run like in a command-line interface. You can provide the same here, by matching such commands. Right now, the chat supports exactly one command, `/=`, which simply echoes its argument, as seen in **Figure 1**.



The screenshot shows a chat application interface. At the top is a text input field with the placeholder text "Type your message". To the right of the input field is a "Send" button. Below the input field is a chat log displaying the following messages:

```
[bob] /= Hello polyglot world!
You sent a command with arguments: Hello polyglot world!
[bob] Hello world
[bob] Joined!
```

**Figure 1.** The simple chat application simply echoes an input.

Next, you'll see how to use this flexibility to enhance the chat application with plotting for serious work chats—and with funny GIF files pulled straight from an online service for a less serious chat experience.

## Finding cat pictures in Python and plotting image metadata with R

Nothing supports an argument in an online discussion more than a nice graph. You can extend the script with a command that draws a simple scatterplot from provided numbers. The syntax will be `/plot` followed by a whitespace-separated list of numbers.

There are numerous Python packages that provide plotting functionality. Unfortunately, none are in the standard Python library. You'll learn how to install and use third-party Python packages shortly, but for now, you can use another superpower of GraalVM: language interoperability.

Among the languages supported by GraalVM is R, the language often used for statistical computing and graphics. With R you do not need to install any additional third-party packages to create the plot. Start with altering the way you build the context instances by adding another permitted language, as follows:

```
@Prototype
@Bean
Context createContext(Engine engine) {
    return Context.newBuilder("python", "R")
        // ...
        build();
}
```

Next, define an R function, `toPlot`, that takes the list of numbers as a string and returns the desired plot, again as a string containing a scalable vector graphics (SVG) markup.

```
library(lattice)

toPlot <- function(data) {
    y = scan(textConnection(data))
    x <- seq_along(y)
    svg('/dev/null', width=5, height=3)
    print(xyplot(y~x))
    return (svg.off())
}

export('toPlot', toPlot)
```

The last line adds the R function to the set of symbols shared between all the languages running within one context. Call this set the *polyglot bindings*. Here is how to access and use this R function in your Python code that is called from Java.

```
from polyglot import import_value
r_plot = import_value('toPlot')

def transform_message(sender, msg):
```

```

match = SLASH_COMMAND_RE.match(msg)
if match:
    try:
        command = match.group(1)
        args = msg.split(maxsplit=1)[1]
        if command == "img":
            # ...
        elif command == "plot":
            return r_plot(args)

```

This uses the `import_value` function provided by the built-in `polyglot` package. This function imports values that have been exposed to the `polyglot` bindings. The object that the `import_value` function returns is still the R function, but in most scenarios, you can work with it as if it were a regular Python function.

In this case, you can invoke the function, passing in a Python string again; R can process Python strings just fine. To stitch this together with your Java code, you will only need to evaluate the R script before evaluating the Python script. For this you will add one more configuration value—the R script—to be used in the factory that creates the `polyglot MessageHandler` service.

```

# application.yml
# ...
scripts:
  python-script: "classpath:scripts/script.py"
  r-script: "classpath:scripts/plot.R"

// Java code:
@Factory
@ConfigurationProperties("scripts")
public class PolyglotMessageHandlerFactory {
    Readable pythonScript;
    Readable rScript; // new field for the configuration value

    @Bean
    @ThreadLocal
    MessageHandler createPythonHandler(Context context) throws IOException {
        // Evaluate the R script and continue as before
        Source rSource = Source.newBuilder("R", rScript.asReader(), rScript.g
context.eval(rSource);

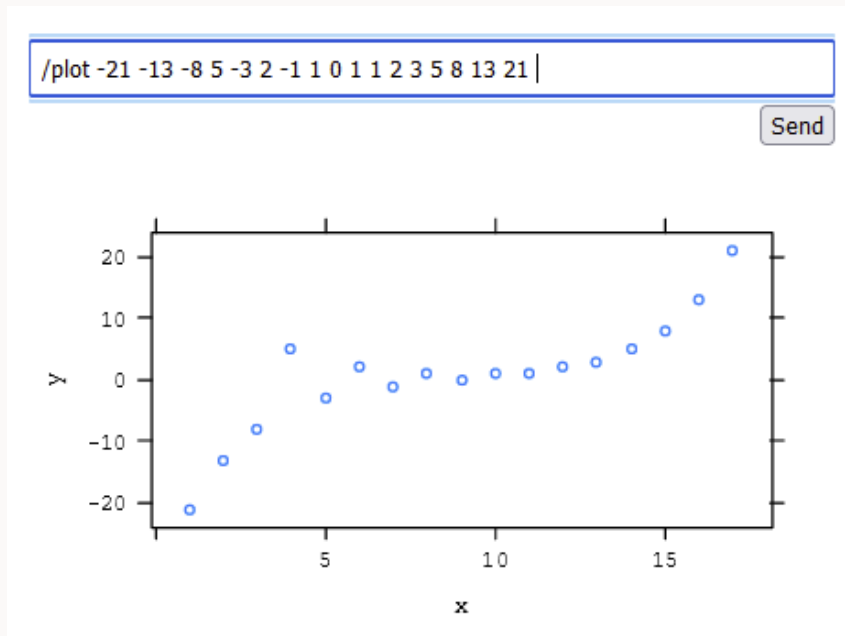
        Source pySource = ...
        context.eval(pySource);
        // ...
    }
}

```



Also note that this adds the `ThreadLocal` scope, because unlike Python, R is a single-threaded language and does not support concurrent access from multiple threads. Because you already decoupled your application to individual Micronaut beans, changing the application so that the Python code is executed in a separate context for each Java thread is this simple!

Now, you can use a proper command to plot the 17 bidirectional Fibonacci numbers around 0, as shown in **Figure 2**.



**Figure 2.** R is used to plot a Fibonacci sequence.

## Adding external packages to GraalVM

Python's packages are on the [Python Package Index \(PyPI\) site](#), where you can find a nice package to interface with [Giphy](#), the humorous GIF search service. But how do you bundle packages with the application? You certainly do not want to have to install them systemwide, or even globally, for the GraalVM installation.

Python developers recommend using *virtual environments*, or *venv*, to bundle project dependencies. This works on GraalPython (GraalVM's Python runtime) as well. To install the package, run the following code:

```
$GRAALVM_HOME/bin/graalpython -m venv src/main/resources/chat_venv
source src/main/resources/chat_venv/bin/activate
pip install giphypop
deactivate
```

[Copy code snippet](#)

This will first create a venv called `chat_venv`, which is the recommended way to isolate Python package sets. You then activate this `chat_venv` in the current shell, making any subsequent Python commands work in that environment. Thus, when you install the `giphypop` package on the third line, it gets installed into the `chat_venv` folder rather than messing with installationwide packages.

In `application.yml`, add a key-value, `python-venv: "classpath:venv/pyvenv.cfg"`, under the `scripts` section. This file was automatically created when you created the venv and is a useful marker file for the application to find its environment.

```
micronaut:
  application:
    name: websocket-chat
  scripts:
    python-script: "classpath:scripts/script.py"
    python-venv: "classpath:chat_venv/pyvenv.cfg"
```

[Copy code snippet](#)

Next, modify the context creation to tell Python to use this venv by passing two new Python options:

```
@Factory
@ConfigurationProperties("scripts")
public class PolyglotContextFactories {
    String pythonVenv;

    // ...

    @Bean
    Context createContext(Engine engine, ResourceResolver r) throws URISyntaxException {
        Path exe = Paths.get(r.getResource(pythonVenv).get().toURI()).resolve("python")
        return Context.newBuilder("python", "R")
            .option("python.ForceImportSite", "true")
            .option("python.Executable", exe.toString())
    }

    // ...
}
```

[Copy code snippet](#)

These two options are important to make the embedded Python code behave as if it had been launched from a shell where the venv was activated. The standard library Python `site` package is automatically imported when you launch any Python on the command line but not when you embed Python.

The first option, `python.ForceImportSite`, enables importing the `site` package even while you are

embedding Python. The package code queries the current executable path and determines if that was launched from a venv folder. That is why the second option, `python.Executable`, is there: to pretend to the Python code that the executable is actually inside the venv—even though really you are running a Java process. This is all possible because Python's package system is set up to use the file system for the implicit discovery of package paths.

To use the service, simply update `script.py` as follows:

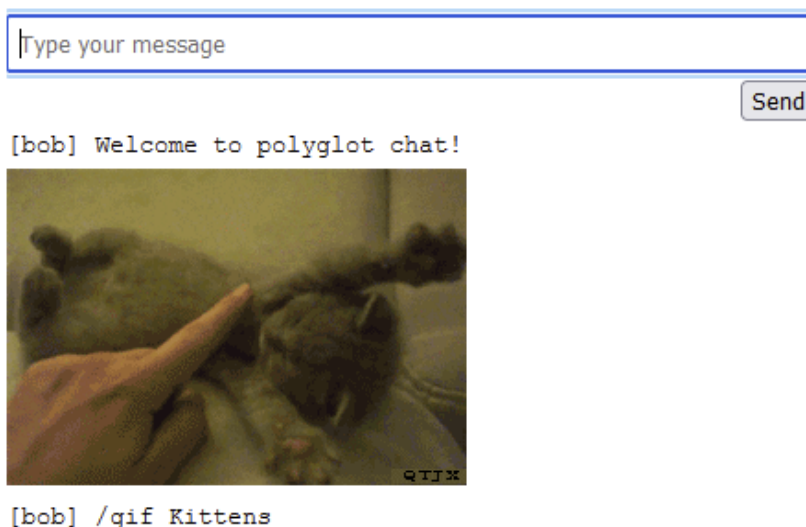
```
# ...

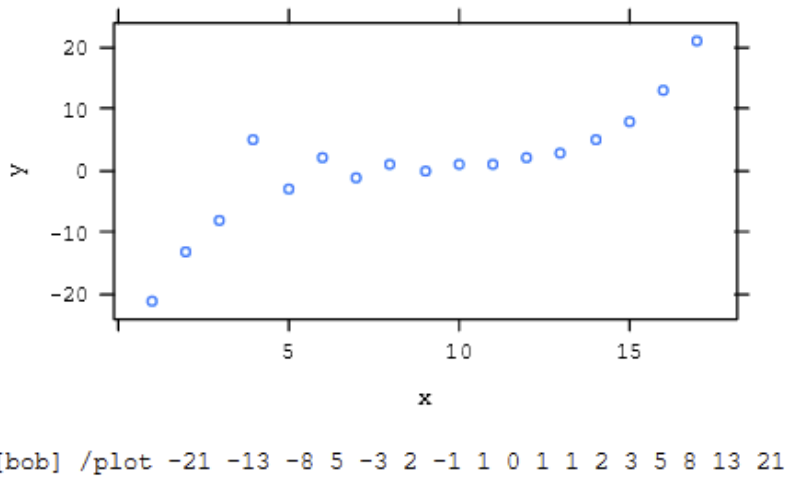
def transform_message(sender, msg):
    match = SLASH_COMMAND_RE.match(msg)
    if match:
        try:
            command = match.group(1)
            args = msg.split(maxsplit=1)[1]
            # ...
        elif command == "gif":
            import giphy
            gif = giphy.Giphy().translate(phrase=args)
            return f"<img src='{gif.media_url}'/>" if gif else "No GIF found"
    # ...
```

[Copy code snippet](#)

The convenience of many Python packages is now right at your fingertips. You can also strip a few megabytes off the venv by removing some folders, if size becomes important; however, a few megabytes of Python source files are to be expected in any case.

**Figure 3** shows the finished chat application, with all the message handlers in use.





**Figure 3.** The completed chat application using Python and R

## Conclusion

GraalVM puts the world of scripting languages such as Python at your fingertips. Any JVM-based application can be extended with any additional languages that GraalVM supports. GraalVM comes with a state-of-the-art just-in-time compiler that optimizes and compiles your Java code and the scripting languages together to provide excellent performance.

GraalVM also allows you to sandbox these additional languages at a fine-grained level. How you can restrict memory usage, execution time, resource access, and even native code execution will be the topic of another article. With GraalVM you can also build native binaries out of your application that include the capability to dynamically load and run scripting languages such as Python and R.

## Dig deeper

- [GraalVM home page](#)
- [GraalVM Python runtime](#)
- [GraalVM R runtime](#)
- [Pedal to the metal: High-performance Java with GraalVM Native Image](#)
- [Java is a top language for AI and ML developers, says research study](#)
- [How to program machine learning in Java with the Tribuo library](#)





### Tim Felgentreff

Tim Felgentreff (@timfelgentreff) is the language development lead for Python on the GraalVM at Oracle. He has been involved with the PyPy/RPython project and various other dynamic language VMs since 2007. Felgentreff also keeps in touch with the Software Architecture Group in the Hasso Plattner Institute at the University of Potsdam, where he received his doctoral degree in 2017.



### Stepan Sindelar

Stepan Sindelar is member of the FastR and GraalPython teams at Oracle Labs. Before joining, he was an intern at Oracle Labs Australia, where he joined the Java Vulnerability Detection project. His interests include static and dynamic program analysis and compiler optimizations. Sindelar received a master's degree from Charles University in Prague.



### Lukas Stadler

Lukas Stadler (@lstad\_ler) is a researcher at Oracle Labs Austria and manages the Python (GraalPython), LLVM (Sulong), and R (FastR) runtimes of the GraalVM polyglot language ecosystem. He works on all aspects of the Graal and Truffle projects, with a special focus on novel compiler optimizations and dynamic language implementations. Stadler's research interests include novel techniques for programming language execution, virtual machines, dynamic compiler optimizations, and feedback-directed optimizations.

[◀ Previous Post](#)

[Next Post ▶](#)

[Resources](#)

[Why Oracle](#)

[Learn](#)

[What's New](#)

[Contact Us](#)

Try Oracle

**for**

About

Careers

Developers

Investors

Partners

Startups

Analyst  
Reports

Best CRM

Cloud  
Economics

Corporate  
Responsibility

Diversity and  
Inclusion

Security  
Practices

What is  
Customer  
Service?

What is ERP?

What is  
Marketing  
Automation?

What is  
Procurement?

What is Talent  
Management?

What is VM?

Try Oracle  
Cloud Free Tier

Oracle  
Sustainability

Oracle COVID-  
19 Response

Oracle and  
SailGP

Oracle and  
Premier  
League

Oracle and Red  
Bull Racing  
Honda

US Sales  
1.800.633.0738

How can we help?

Subscribe to  
Oracle Content

Try Oracle Cloud  
Free Tier

Events

News

© 2022 Oracle

| [Site Map](#)

[Privacy / Do Not Sell My Info](#)

[Cookie Preferences](#)

[Ad Choices](#)

[Careers](#)