

Ollama

Since testcontainers-go  v0.29.0

Introduction

The Testcontainers module for Ollama.

Adding this module to your project dependencies

Please run the following command to add the Ollama module to your Go dependencies:

```
go get github.com/testcontainers/testcontainers-go/modules/ollama
```

Usage example

Creating a Ollama container

```
ctx := context.Background()

ollamaContainer, err := tcollama.RunContainer(ctx,
testcontainers.WithImage("ollama/ollama:0.1.25"))
if err != nil {
    log.Fatalf("failed to start container: %s", err)
}

// Clean up the container
defer func() {
    if err := ollamaContainer.Terminate(ctx); err != nil {
        log.Fatalf("failed to terminate container: %s", err) //
    nolint:gocritic
    }
}()
{}
```

Module reference

The Ollama module exposes one endpoint function to create the Ollama container, and this function receives two parameters:

```
func RunContainer(ctx context.Context, opts
...testcontainers.ContainerCustomizer) (*OllamaContainer, error)
```

- `context.Context`, the Go context.
- `testcontainers.ContainerCustomizer`, a variadic argument for passing options.

Container Options

When starting the Ollama container, you can pass options in a variadic way to configure it.


Image

If you need to set a different Ollama Docker image, you can use

`testcontainers.WithImage` with a valid Docker image for Ollama. E.g.

```
testcontainers.WithImage("ollama/ollama:0.1.25").
```

Image Substitutions

- Since testcontainers-go  [v0.26.0](#)

In more locked down / secured environments, it can be problematic to pull images from Docker Hub and run them without additional precautions.

An image name substitutor converts a Docker image name, as may be specified in code, to an alternative name. This is intended to provide a way to override image names, for example to enforce pulling of images from a private registry.

Testcontainers for Go exposes an interface to perform this operations:

`ImageSubstitutor`, and a No-operation implementation to be used as reference for custom implementations:

Image Substitutor Interface

```
// ImageSubstitutor represents a way to substitute container image names
type ImageSubstitutor interface {
    // Description returns the name of the type and a short description
    of how it modifies the image.
```

```
// Useful to be printed in logs
Description() string
Substitute(image string) (string, error)
}
```

Noop Image Substitutor

```
type NoopImageSubstitutor struct{}

// Description returns a description of what is expected from this
// Substitutor,
// which is used in logs.
func (s NoopImageSubstitutor) Description() string {
    return "NoopImageSubstitutor (noop)"
}

// Substitute returns the original image, without any change
func (s NoopImageSubstitutor) Substitute(image string) (string, error) {
    return image, nil
}
```

Using the `WithImageSubstitutors` options, you could define your own substitutions to the container images. E.g. adding a prefix to the images so that they can be pulled from a Docker registry other than Docker Hub. This is the usual mechanism for using Docker image proxies, caches, etc.

WithEnv

- Since testcontainers-go  [v0.29.0](#)

If you need to either pass additional environment variables to a container or override them, you can use `testcontainers.WithEnv` for example:

```
postgres, err = postgresModule.RunContainer(ctx,
testcontainers.WithEnv(map[string]string{"POSTGRES_INITDB_ARGS": "--no-
sync"}))
```

WithHostPortAccess

- Not available until the next release of testcontainers-go  [main](#)

If you need to access a port that is already running in the host, you can use `testcontainers.WithHostPortAccess` for example:

```
postgres, err = postgresModule.RunContainer(ctx,
testcontainers.WithHostPortAccess(8080))
```

To understand more about this feature, please read the [Exposing host ports to the container](#) documentation.

WithLogConsumers

- Since testcontainers-go  v0.28.0

If you need to consume the logs of the container, you can use

`testcontainers.WithLogConsumers` with a valid log consumer. An example of a log consumer is the following:

```
type TestLogConsumer struct {
    Msgs []string
}

func (g *TestLogConsumer) Accept(l Log) {
    g.Msgs = append(g.Msgs, string(l.Content))
}
```

WithLogger

- Since testcontainers-go  v0.29.0

If you need to either pass logger to a container, you can use

`testcontainers.WithLogger`.

Info

Consider calling this before other "With" functions as these may generate logs.

In this example we also use `TestLogger` which writes to the passed in `testing.TB` using `Logf`. The result is that we capture all logging from the container into the test context meaning its hidden behind `go test -v` and is associated with the relevant test, providing the user with useful context instead of appearing out of band.

```
func TestHandler(t *testing.T) {
    logger := TestLogger(t)
    _, err := postgresModule.RunContainer(ctx,
    testcontainers.WithLogger(logger))
    require.NoError(t, err)
    // Do something with container.
}
```

Please read the [Following Container Logs](#) documentation for more information about creating log consumers.

Wait Strategies

If you need to set a different wait strategy for the container, you can use `testcontainers.WithWaitStrategy` with a valid wait strategy.

Info

The default deadline for the wait strategy is 60 seconds.

At the same time, it's possible to set a wait strategy and a custom deadline with `testcontainers.WithWaitStrategyAndDeadline`.

Startup Commands

- Since testcontainers-go  [v0.25.0](#)

Testcontainers exposes the `WithStartupCommand(e ...Executable)` option to run arbitrary commands in the container right after it's started.

Info


To better understand how this feature works, please read the [Create containers: Lifecycle Hooks](#) documentation.

It also exports an `Executable` interface, defining the following methods:

- `AsCommand()`, which returns a slice of strings to represent the command and positional arguments to be executed in the container;
- `Options()`, which returns the slice of functional options with the Docker's `ExecConfigs` used to create the command in the container (the working directory, environment variables, user executing the command, etc) and the possible output format (Multiplexed).

You could use this feature to run a custom script, or to run a command that is not supported by the module right after the container is started.

Ready Commands

- Since testcontainers-go  v0.28.0

Testcontainers exposes the `WithAfterReadyCommand(e ...Executable)` option to run arbitrary commands in the container right after it's ready, which happens when the defined wait strategies have finished with success.

Info

To better understand how this feature works, please read the [Create containers: Lifecycle Hooks](#) documentation.

It leverages the `Executable` interface to represent the command and positional arguments to be executed in the container.

You could use this feature to run a custom script, or to run a command that is not supported by the module right after the container is ready.

WithNetwork

- Since testcontainers-go  v0.27.0

By default, the container is started in the default Docker network. If you want to use an already existing Docker network you created in your code, you can use the `network.WithNetwork(aliaes []string, nw *testcontainers.DockerNetwork)` option, which receives an alias as parameter and your network, attaching the container to it, and setting the network alias for that network.

In the case you need to retrieve the network name, you can simply read it from the struct's `Name` field. E.g. `nw.Name`.

Warning

This option is not checking whether the network exists or not. If you use a network that doesn't exist, the container will start in the default Docker network, as in the default behavior.

WithNewNetwork

- Since testcontainers-go  v0.27.0

If you want to attach your containers to a throw-away network, you can use the `network.WithNewNetwork(ctx context.Context, aliases []string, opts ...network.NetworkCustomizer)` option, which receives an alias as parameter, creating the new network with a random name, attaching the container to it, and setting the network alias for that network.

In the case you need to retrieve the network name, you can use the `Networks(ctx)` method of the `Container` interface, right after it's running, which returns a slice of strings with the names of the networks where the container is attached.

Docker type modifiers

If you need an advanced configuration for the container, you can leverage the following Docker type modifiers:

- `testcontainers.WithConfigModifier`
- `testcontainers.WithHostConfigModifier`
- `testcontainers.WithEndpointSettingsModifier`

Please read the [Create containers: Advanced Settings](#) documentation for more information.

Customising the ContainerRequest

This option will merge the customized request into the module's own `ContainerRequest`.

```
container, err := RunContainer(ctx,
    /* Other module options */

    testcontainers.CustomizeRequest(testcontainers.GenericContainerRequest{
        ContainerRequest: testcontainers.ContainerRequest{
            Cmd: []string{"-c", "log_statement=all"},
        },
    })),
)
```

The above example is updating the predefined command of the image, **appending** them to the module's command.

Info

This can't be used to replace the command, only to append options.

Container Methods

The Ollama container exposes the following methods:

ConnectionString

This method returns the connection string to connect to the Ollama container, using the default `11434` port.

Get connection string

```
connectionStr, err := container.ConnectionString(ctx)
```

Commit

This method commits the container to a new image, returning the new image ID. It should be used after a model has been pulled and loaded into the container in order to create a new image with the model, and eventually use it as the base image for a new container. That will speed up the execution of the following containers.

Commit Ollama image

```
// Defining the target image name based on the default image and a
// random string.
// Users can change the way this is generated, but it should be unique.
targetImage := fmt.Sprintf("%s-%s", ollama.DefaultOllamaImage,
strings.ToLower(uuid.New().String()[4]))

err := container.Commit(context.Background(), targetImage)
```

Examples

Loading Models

It's possible to initialise the Ollama container with a specific model passed as parameter. The supported models are described in the Ollama project: <https://github.com/ollama/ollama?tab=readme-ov-file> and <https://ollama.com/library>.

Warning

At the moment you use one of those models, the Ollama image will load the model and could take longer to start because of that.

The following examples use the `llama2` model to connect to the Ollama container using HTTP and Langchain.

Using HTTP

```
ctx := context.Background()

ollamaContainer, err := tcollama.RunContainer(
    ctx,
    testcontainers.WithImage("ollama/ollama:0.1.25"),
)
if err != nil {
    log.Fatalf("failed to start container: %s", err)
}
defer func() {
    if err := ollamaContainer.Terminate(ctx); err != nil {
        log.Fatalf("failed to terminate container: %s", err) //
    nolint:gocritic
    }
}()

model := "llama2"

_, _, err = ollamaContainer.Exec(ctx, []string{"ollama", "pull",
model})
if err != nil {
    log.Fatalf("failed to pull model %s: %s", model, err)
}

_, _, err = ollamaContainer.Exec(ctx, []string{"ollama", "run",
model})
if err != nil {
    log.Fatalf("failed to run model %s: %s", model, err)
}

connectionStr, err := ollamaContainer.ConnectionString(ctx)
if err != nil {
    log.Fatalf("failed to get connection string: %s", err) //
    nolint:gocritic
}

httpClient := &http.Client{}

// generate a response
payload := `{
    "model": "llama2",
    "prompt": "Why is the sky blue?"
}`

req, err := http.NewRequest("POST", fmt.Sprintf("%s/api/generate",
connectionStr), strings.NewReader(payload))
if err != nil {
```

```

        log.Fatalf("failed to create request: %s", err) //
nolint:gocritic
    }

    resp, err := httpClient.Do(req)
    if err != nil {
        log.Fatalf("failed to get response: %s", err) // nolint:gocritic
    }

```

Using Langchaingo

```

ctx := context.Background()

ollamaContainer, err := tcollama.RunContainer(
    ctx,
    testcontainers.WithImage("ollama/ollama:0.1.25"),
)
if err != nil {
    log.Fatalf("failed to start container: %s", err)
}
defer func() {
    if err := ollamaContainer.Terminate(ctx); err != nil {
        log.Fatalf("failed to terminate container: %s", err) //
nolint:gocritic
    }
}()

model := "llama2"

_, _, err = ollamaContainer.Exec(ctx, []string{"ollama", "pull", model})
if err != nil {
    log.Fatalf("failed to pull model %s: %s", model, err)
}

_, _, err = ollamaContainer.Exec(ctx, []string{"ollama", "run", model})
if err != nil {
    log.Fatalf("failed to run model %s: %s", model, err)
}

connectionStr, err := ollamaContainer.ConnectionString(ctx)
if err != nil {
    log.Fatalf("failed to get connection string: %s", err) //
nolint:gocritic
}

var llm *langchainollama.LLM
if llm, err = langchainollama.New(
    langchainollama.WithModel(model),
    langchainollama.WithServerURL(connectionStr),
); err != nil {
    log.Fatalf("failed to create langchain ollama: %s", err) //
nolint:gocritic
}

```

```

completion, err := llm.Call(
    context.Background(),
    "how can Testcontainers help with testing?",
    llms.WithSeed(42), // the lower the seed, the more
deterministic the completion
    llms.WithTemperature(0.0), // the lower the temperature, the more
creative the completion
)
if err != nil {
    log.Fatalf("failed to create langchain ollama: %s", err) //
nolint:gocritic
}

words := []string{
    "easy", "isolation", "consistency",
}
lwCompletion := strings.ToLower(completion)

for _, word := range words {
    if strings.Contains(lwCompletion, word) {
        fmt.Println(true)
    }
}

```