Go SDK

Up and Running with Azure

Get Ready to Learn

In this presentation, I will walk you through my own implementation of the Azure Go SDK that does the following:

1. Authenticates an Azure user;
2. Creates an instance of a http client for making requests;
3. Creates a resource group;
4. And deploys an Azure Resource Management, or ARM, Template

I do make a few assumptions about your previous knowledge and level of motivation:

1. You are willing to learn on the fly when an unknown concept is introduced.
2. You understand the basics of a programming language, ideally Go.
3. You understand the basics of Microsoft Azure.
4. You can use standard technical documentation to do things like install and use software.

Explore Resources

- Presentation transcript
- Go language spec
- SDK docs
- SDK reference
- ARM templates docs

Avoid Unnecessary Costs

If you execute the code from this presentation, there is the possibility that you may incur some costs from the Azure services you create.

The actual cost is largely dependent on how long you leave the services running, so make sure to keep a close eye on the bill for your subscription and delete any resources that you create as soon as you are done using them.

Understand the Azure SDK

The Azure SDK for Go provides several libraries to communicate with and programmatically manage Azure services.

The SDK is built on top of the Azure REST API. This hierarchy allows you to access the functionality of the Azure REST API directly from your Go programs.

So if you want to interact with Azure services through a server built in Go or an executable Go program running on a physical or virtual machine, the Azure Go SDK makes it possible.

Before I actually get into showing off the code to the SDK program, I want to mention some alternatives to using the Go SDK and provide general guidance for setting up a local development environment.

Understand Alternatives to the SDK

The Azure Resource Manager is an Azure service which receives all requests for managing a resource: these can be requests to read, create, delete, or update a *resource**.

When I send a request through my Go SDK program, the resource manager receives, authenticates and authorizes that request before forwarding it to the appropriate Azure service.

Because all requests are handled through the same Resource Manager API, the results and capabilities are consistent throughout all the available resource management tools, or interfaces as they are sometimes referred to in the docs.

While I have chosen to use the Go SDK as my tool for working with the Azure *Resource Manager*, I want to mention the alternative tools you could choose to use in place of the Go SDK:

- The Azure Portal;
- PowerShell;
- Azure CLI;
- The REST API;
- Terraform;
- And Client SDKs for Java, Dot Net, Javascript, Python, C, and C++

Each of these interfaces merit a bit of research to understand their advantages and disadvantages, but I will not talk more about them here. Just know that you could essentially use any of them in-place of the Go SDK.

Setup Local Environment

Running this project on one of the three major operating systems (that is, Mac, Windows, or Linux) requires the following:

- A command line processor;
- A text editor;

- An installation of Go;
- A Microsoft Azure account;
- And the Azure CLI.

Click on your operating system below for basic setup links to help you get setup on your system.

Mac

- Text Editor: https://code.visualstudio.com/docs/?dv=osx
- Command Line: https://support.apple.com/guide/terminal/welcome/mac
- Go: https://go.dev/doc/install
- Azure: https://azure.microsoft.com/en-ca/free/
- CLI: https://learn.microsoft.com/en-us/cli/azure/install-azure-cli-macos

Windows

- Command line: https://learn.microsoft.com/en-us/powershell/scripting/overview
- Text Editor: https://code.visualstudio.com/docs/?dv=win
- Go: https://go.dev/doc/install
- Azure: https://azure.microsoft.com/en-ca/free/
- CLI: https://learn.microsoft.com/en-us/cli/azure/install-azure-cli-windows?tabs=azure-cli

Linux

- Cross platform command line tool: https://tabby.sh/
- Text Editor: [Debian or Ubuntu] https://code.visualstudio.com/docs/?dv=linux64_deb
- Text Editor: [Red Hat, Fedora, or SUSE] https://code.visualstudio.com/docs/?dv=linux64_deb
- Go: https://go.dev/doc/install
- Azure: https://azure.microsoft.com/en-ca/free/
- CLI: https://learn.microsoft.com/en-us/cli/azure/install-azure-cli-linux

Use the Azure CLI for Authentication

As a developer working in a development only environment, authenticating through the CLI is the quickest way to get started.

To interactively authenticate with the CLI, complete the following steps:
1. Create a basic account with a tenant and subscription.
2. Install the Azure CLI.
3. Use the CLI "az login" command to open a browser and sign-in.

These steps sign-in my user principle and provide a token that the NewAzureCLICredential function will use across the Go SDK.

Give the SDK Access to Azure

The first step in using the SDK is to provide an authentication token for creating and deploying resources.

[1] All the code in this file is part of the main package and so I must name it as such. All Go programs must begin execution in the main package.

[5-9] Here at the top of the file, I am importing some of the language's standard library packages that I will use and explain later.

[12-15] On the next few lines, I am importing the required Azure Go SDK packages. If you are using VS Code as your text editor, you should be able to get a link to these specific library's documentation by hovering over the line of code that imports that library. You can also manually search the go.dev site for the package.

[19] Next, I declare the required main function. I have seen Go programs that opt to put the main function at the bottom of the main package, but I personally like it to be at the top as it is the first function that gets called when the program is executed.

[20] Here, I am declaring two variables that will be assigned the returned values of the NewAzureCLICredential function. The signature of NewAzureCLICredential shows the returned values: The first value it returns, and the value that will be assigned to cred, is an instance of the AzureCLICredential struct. The second value it returns, that will be assigned to err (E. R. R.), is an instance of the built-in Go error type. Also according to the signature, NewAzureCLICredential accepts a parameter of options, but I pass nil as the parameter to accept the default options.

[21-23] The next few lines will log Fatal to print the error message and exit the program if err (E. R. R.) is not nil. In my case, the NewAzureCLICredential function would assign a string to err should something go wrong. This is the conventional interface for representing an error condition in Go, with the nil value representing no error.

Call the createResourceGroup Function

After obtaining an authentication token, the next step will be to create a resource group for the deployment. A resource group is a container that holds related resources that you want to manage as a group. That means that all the resources in your resource group should share the same lifecycle: being deployed, updated, and deleted together. When you create a resource group, you need to provide a location where that resource group's metadata will be stored.

The next few lines are simply assigning identifiers and context to variables that will need to be passed into function calls as parameters.

[25] On this line, I create an empty context by using the background function; I have to include the context only because it is a required parameter for the createResourceGroup function.

[26] Since the createResourceGroup function takes my subscriptionID string as a parameter and I consider that to be “sensitive” information, I have used the os.getenv method to retrieve subscriptionID value from my local environment variables. Not only is this the suggested way to handle this situation in the Go SDK docs, but I believe it is one of the fastest and easiest ways to store and retrieve sensitive information that your code needs. Whether you use local environment variables or not, keep in mind that when working with sensitive data like API keys, account credentials, or unique identifiers, you should not commit that sensitive information to version control (for example, a GitHub repository). The reason for this is that it can be complicated to remove traces of sensitive data once it has been recorded in version control history. I verify that subscriptionID is populating correctly by creating a breakpoint and running the program in debug mode. I then use the debug console to check the value of subscriptionID. If the variables are not populating correctly, you may find that restarting your terminal and code editor fixes the issue.

[27-28] The Resource Group name and location are additional important parameters required for the createResourceGroup function.

[30] I choose to encapsulate all the steps for creating a resource group into one function called createResourceGroup which will return a non-nil error if anything should go wrong. I could potentially also return a struct containing the newly created resource group. But since I do not need the response anywhere else, I am only returning an error.

[31-33] In the same way that I handled an error above, these lines of code will print the error message and exit the program if error is not nil. In other words, if something goes wrong while trying to create the resource group.

Define the createResourceGroup Function

[39-45] In the signature of createResourceGroup you can see that it requires five parameters and returns an error. The parameters each have a type assigned to them. This is great as I can quickly see exactly which parameters to include when I call this function.

[46-49] The first order of business is to create a client for the resource group. The SDK is built on-top-of the Azure REST API. So it needs a way to tap into the underlying HTTP “pipeline” used to make requests and handle responses. Azure uses the term *client* to refer to a specific instance of an HTTP pipeline for a specific Azure resource. As the developer, it is my responsibility to call the SDK function that creates a client instance for the resource I want to make requests for. The SDK function for creating an instance of the client is

NewResourceGroupsClient. If there is an error while creating the client, I am returning that error to the calling function instead of simply logging it, as I did above.

[51] The CheckExistence function takes a resource group name and checks if it already exists in the Azure subscription. This is important because calling the CreateOrUpdate function when it is not necessary is a waste of network resources. The first value this function returns is a struct with a field name of Success and a field value of boolean. If Success is equal to true, the resource group *does* already exist. In accordance with the SDK's convention, I have used nil as the last parameter to accept the default options.

[52-59] After the CheckExistance function is evaluated, there are three possible outcomes: One, there is an error and I return it to the calling function. Two, the response boolean evaluates to true, meaning the resource group exists. In that case, I will log a message and return to the calling function an error value of nil representing no error. Three, I log that the resource does not yet exist and continue execution. Remember, if there is no return statement, execution inside the current function will continue as normal.

[61-72] If execution of my program reaches this line, a new resource group will be created (or updated). The CreateOrUpdate function returns the newly created or updated resource group. I am not doing much with the returned value other than printing it out, but I have the option to return it to the calling function. One other notable thing about the CreateOrUpdate function is that the parameter containing the resource location is a struct. When creating the location environment variable, make sure that you format the location string correctly; all lowercase letters and no spaces.

[74] If none of the previous operations have returned an error, all has gone well and the resource group should be logged in my terminal. I return a nil value to the calling function.

Define the deployTemplate function

The deployTemplate function follows much of the same logic as the createResourceGroup function:

[84-87] A client is created or the program is stopped in the case where an error occurs

[89-97] In the CheckExistence function, I check if the deployment already exists. As with the resource group check, I return a struct with a Success boolean and an error to determine if the program should continue creating the deployment.

[99-107] Here is where the logic diverges from the previous createResourceGroup function. The readJSON function returns a Map type containing the template and parameters for the deployment.

[140] When provided with a valid JSON file, this returns a string of the JSON content.

[146] This creates a Map that will store the data from the JSON string. Copy the link in the comment above to learn more.

[150] This stores the JSON string inside the Map and checks for an error. Copy the link in the comment above to learn more.

[155] If there have not been any errors, the Map containing the JSON data and an error with the value nil are returned.

[112-122] Once the Map of both the template and parameter are in scope, I am ready to execute the deployment for the indicated resource group. By convention, methods that start a long running operation or “LRO” are prefixed with "Begin" and return a *Poller*. The Poller is used to periodically poll the service until the operation finishes. The BeginCreateOrUpdate function takes a number of parameters, but the armresources.Deployment.Properties is of particular interest as it accepts the template and parameters Maps and requires that a Mode be defined as either “incremental” or “complete”. In Incremental mode, resources are deployed without deleting existing resources that are not included in the template. In Complete mode, resources are deployed and existing resources in the resource group that are not included in the template are deleted. Be careful when using Complete mode as you may unintentionally delete resources. Also keep in mind that, in addition to template parameters being included in a Map, parameters could also be hard coded directly in the template file or linked from an external source.

[129-133] The PollUntilDone function is called to eventually either log out that the deployment succeeded or return a non-nil error. Notice how it depends on the poller that the BeginCreateOrUpdate function returned?

Wrap-up

If you have followed along with me, you should be able to login to your Azure portal and see that the resource group and resources from the template have been created in your subscription. If something did not work correctly, check and debug your code carefully. If all your troubleshooting attempts do not yield very good results, you can get a working example running by executing the code I used (after changing the subscriptionID value).

Finally, if you have successfully created any Azure resources throughout this presentation, make sure to delete the entire resource group to avoid incurring any undesired costs in your subscription.