**Slide 0 [00:05:00]**

*Wait 5 minutes for join all***Slide 1 [00:01:00]**

Before we start, I am planning to show 4 demos. I hope we have enough time. In case any questions – please ask me and do not wait for final slide.

Hi. My name is \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_. I’m L4 software engineer in \_\_\_\_\_.

Usually, I develop backends on .NET Core C# stack. But sometimes I’m working outside my primary skills and DevOps activities take big part in my tasks. Based on my experience, in the past 5 years, cloud configuration is a common task for many developers.

Very common task is creating database, backend and frontend, create host in cloud, deploy customer application into the cloud, setup configuration.

First problem is kept it in actual working state, fix issues in configuration.

Next problem is creating several similar environments like dev/test/pre-prod/prod for Continuous Delivery. And again, all environments must be in equals or similar state. Any change on dev environment must be applied for next environments.

All this problem can solve DevOps engineer. But customers don’t like to pay for DevOps engineer. For a small team, DevOps engineer capacity can be very small. For big projects, your DevOps team sprints are planned for the next 3 months, and difficult to do something fast. A common situation is when communication about change takes more time than a fix. Must plan more because have dependencies between teams or one teammate depends on another.

Let’s look at one of the possible ways to close all questions above, automate and simplify the process. Make it easier and interesting. We will look a little at the tasks of DevOps, one of the approaches and one of the possible utilities

First, let’s talk about ***Infrastructure as a Code*** approach.

**Slide 2 [00:02:00]**

Definition for ***Infrastructure as a Code*** you can see on slide.

A short and simple description is ability to store the definition and configuration for something as code.

Let’s talk about first 2 benefits:

**pros 1-2**: I should say the main and important point for me is the ability to define something in a cloud in several lines of code. Possible to share it with teammates, view history of changes in git.

There is no need to click the mouse or remember the pattern for naming services.

I don't need to remember differences in configuration for test environment and production. I don’t need to implement configuration for one environment and try to reproduce the same other.

Sound great, looks like silver bullet and can solve all problems. No negative moments, all ok. Let’s investigate it.

**Slide 3 [00:05:00]**

On the current project we could choose any tool. The main idea was to use the IaC approach and automate management of the infrastructure as much as possible. Prevent any manual configuration, have stable and repeatable environment. We do not have DevOps, only backend and frontend engineers.

I think we can skip the long story of the choice of tool. I can only say that Terraform is not the first chosen tool.

Minimal information you can see on the slide. All it taken from official Terraform web site.

HashiCorp provides a small CLI tool with the name terraform. All that requires to start use terraform CLI tool is to download it from the official site, unpack it to a directory and add the directory to environment variable ‘%PATH%’.

The image on slide explains a key feature of terraform a lot:

* we have several files with infrastructure. In out case it is Azure Storage Account, Logical App and Azure Function
* we use one small tool
  + logo on the image
* terraform create and configure cloud resources
  + on the image we are working with Azure, but it can be AWS, google cloud, GitHub, and many other
* terraform have something named state
  + state is a file with information about something that was created or changed by terraform.
  + for example, we would like to create Azure Storage Account. All that we have to do is create correct storage.tf file, run terraform and in the end of process terraform will add information about created Storage Account into terraform.state file. Terraform will know unique id of Azure Storage Account and other information that allow trach changes, update or delete resource in future
* resources can have dependencies and terraform can handle it
  + for example, Azure Function depends on Azure Storage Account. It is not a problem for Terraform. Terraform will create it in correct order: First is Azure Storage Account, second is Azure Function with valid configuration because we will know all information about Azure Storage Account and will use it.
* possible to change terraform files and re-run terraform CLI tool many times

Based on the image we can say that we are store information about our infrastructure 3 times:

* in terraform files (files with extensions \*.tf)
* in state file
* in cloud

Terraform know how to read that all, join together based on internal implementation and sync together.

I think there are already too many words, let's terraform something.

Before we start use terraform, let’s understand several moments in HashiCorp Configuration Language.

Terraform uses HashiCorp Configuration Language. It is a very simple and untestable language with minimal features, a system of types, have functions for strings, collections, numbers etc., have logical operations, local and input variables, have modules system, and more.**Slide 4 [ DEMO #0 ] [00:01:00]**

In first demo we will use 2 terraform features: providers and resources.

What is provider?

After installation, Terraform CLI know nothing about Azure, AWS, Google Cloud, GitHub and other. Terraform uses something named as ‘provider’. Based on my understanding and sources on GitHub, provider is intermediate layer or library for communication between terraform and infrastructure (some api, demon, tool and etc.).

We can use more that 1200 providers, <https://registry.terraform.io/browse/providers>.

Provider installation very similar to ‘using’ in c#, import in typescript, require in nodejs or import in Python.

Important moment is require\_provider directive, source and version. It allow define name and version of provider. After run command terraform init, terraform will automatically check downloaded provider and download in case provider missed.

*Is it clear? Any questions?*

**Slide 5 [ DEMO #0 ] [00:01:00]**

After installation of provider, we can work with resources in provider. If we are talking about Azure, we can use resource group.

Resources very similar for creating new object in c#, typescript, nodejs or python.

Keyword resource similar var/let/const and etc. Next part is a type of resource line int/string/object/number and etc, on the screen our type is azurerm\_resource\_group. Name of resource example.

*Is it clear? Any questions?*

**Slide 6 [ DEMO #0 ] [00:15:00]**

Let’s imagine situation when we need SQL database for some pet project or test. I can’t create it because my OS do not allow install server, or I don’t like install server. But I can create database in cloud. I can create it manually, but this process will take 10-15 minutes, I must fill several fields and I must remember password. In case any mistake, I have to repeat process and will take additional 10 minutes. Other possible approach is to use terraform.

First of all, let’s check my azure resources groups list. It has to be empty.

<https://portal.azure.com/>

I will run several commands in terminal without explanation. I will explain each of them in 5 minutes. I have to do that because SQL Server takes a lot of time.

*Command + K in terminal*

*Navigate to docs/demo#0/sql\_database/*

*Run commands from docs/ demo#0/sql\_database/README.md*

Let’s take a look into `main.tf` file:

In file we have `terraform` section with information about provider. In our demo it is `hashicorp/azurerm`. Terraform use `providers` to interact with different remote system. Each provider adds a set of resource types and data sources that Terraform can manage.

`hashicorp/azurerm` is official HashiCorp provider for Azure Cloud.

* Open [link](https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/managed_disk).
* Enter `azurerm\_` in search box

As you see we can manage hundreds of different resources and features.

Section `provider "azurerm"` with an empty list of features is provider configuration. In the current demo, we don't need additional features.

Next important part of file is:

resource "azurerm\_resource\_group" "example"

* first part of the line `resource` is a definition for some resource. As was discussed before it is like `var`, `let` or `const` in software development language.
* `azurerm\_resource\_group` is a type of resource. Like `string`, `number` or `boolean` type.
* `example` is a name, it is something like name of variable in software development language.
* In body we have 2 identifiers: `name` and `location`. There are Azure specific require identifiers for Resource Group. We can’t create resource group without these values.

We can add additional identifiers, like tags for Resource Group. Some identifiers impossible to set, for example Resource Group Id.

*Do you have any questions?*

Let’s look at the definition of SQL server. We see the same definition for resource, have password, login, name. And have a new important moment. It is dependencies between resources. I’m talking about `resource\_group\_name` and `location`. These two lines allow set Resource Group and location for server. In the background, SQL server will be created only after success creation of Resource Group.

Note: password in file is a wrong approach. I did it for simple and clear example. Do not repeat.

And last section defines SQL Database. Nothing new for us. Only information about SQL Server for database.

*Do you have any questions?*

Let’s check what we have in cloud.

*Show resource group, SQL Server and SQL Database*

Now let's go back to the terminal:

az login (missed because already logged in)

az account show

First of all, Terraform is not a magic. Terraform must use some approach for communicate with infrastructure. If we are talking about cloud in common it REST API, it can be demon or some console tool. It depends on what we need.

If we are talking about Azure cloud, Terraform works over az-cli tool. We have to login first in az-cli. Another cloud will use its own approach or tools.

terraform init

Will install dependencies. In our example it is `azurerm` provider.

terraform plan -out current.tfplan

this line checks all files with `\*.tf` extension in the directory, tries to find the state file, and builds a plan for changes.

*Explain output*

*Show logs after plan command*

*Part of data will know after applying*

*Part of data sensitive*

*Show applies after plan command*

*One way for execute, impossible do it in parallel*

In log we see set of planned changes. In our example we will create several resources.

Other possible actions are destroy and change, sometimes destroy and create new instance of resource, as a result re-create.

terraform apply current.tfplan

Run plan file or apply all planned changes from the file.

*Do you have any questions?*

What about state? As I said before Terraform have state file.

*Show state file.*

As you see in terraform file, we have information about the last run (date), have some unique number for a run, have a list of all resources and properties-values for them.

*Do you have any questions?*

I must run one additional long running operation.

*Run: terraform destroy*

In my experience, I used state file several times. For example, to search for a password after launching a terraform as part of a CI/CD pipeline. HasiCorp is confident that the security of this file is not their problem, it is the developer's problem. Please be careful. Terraform can store this file in cloud for you, but can’t hide it, can’t encrypt it for you. This is developer responsibility.

*Do you have any questions?***Slide 7 [ DEMO#1 ] [00:00:00]**

Next important moment is iterative approach when we have to develop some kind of infrastructure some period of time. I suppose it is common situation for any developer or DevOps engineer.

Let's simulate real changes in infrastructure in an iterative approach.

In this example I’ll use prepared files without any changes, I’ll pull different git commits. I will explain the differences in files in each commit.

First, authentication require.

<https://dev.azure.com/AHEpam/_usersSettings/tokens>

*Generate token and add into .env file*

*Show list of projects (have to me empty)*

*Command + K in terminal*

*Navigate to docs/examples/azure\_dev\_ops/main/*

In this example we have other provider `microsoft/azuredevops`. And in that case, it is verified provider for Azure DevOps (Microsoft developed this provider).

On iteration#0 will add 2 resources:

* new Azure DevOps project with name `DotNet\_Community\_How\_to\_Terraform`
* new git repository with name `How\_to\_Terraform\_Git`

*Run terminal commands*

*Explain planned changes*

*Check project and repository*

As you see we have new project with expected name and 2 repositories. Why do we see 2 when we define only one?

It is an Azure DevOps implementation feature. When we create a new project, a new repository with the same name will be created in any case.

On iteration#1 will change project resource:

* rename project
* add several optional properties
* change name of project in terraform file
* change name of repository in terraform file

*Run terminal commands*

*Explain planned changes*

*Check project and repository*

On iteration#2 will change repositories:

* remove existing repository
* create new repository `How\_to\_Terraform\_Git\_I` with initial type `Clean`
* create new repository `How\_to\_Terraform\_Git\_II` with initial type `Clean`

*Run terminal commands*

*Explain planned changes*

*Check project and repository*

As we saw, the repository was removed. Possibly we lose code in the repository.

We have to be careful and be attentive when executing Terraform changes.

Terraform is a very powerful tool. But at the same time, very easy to destroy something and lose data at all. Please be careful. Terraform try to help us, but in any case, it is the engineer's responsibility.

*Do you have any questions?*

Let’s keep it as is and complete first part of current demo.

**Slide 8 [ DEMO #2 ] [00:01:00]**

Other important feature is data source.

On the slide you see resource of type git repository. Name of it How\_to\_terraform\_Git\_I.

On the other image you can see import of already created resource as data source. Name of imported resource How\_to\_terraform\_Git\_I.

The difference in images is type of words: resource and data, type of both the same, property name for each the same.

**Slide 9 [ DEMO #2 ] [00:15:00]**

let’s try to understand what to do in case we like to use terraform, but a lot of resources already created, and impossible to re-implement it in terraform.

Let's extend existing resources from previous demo with Terraform.

Let’s think that we already have project and repositories in Azure DevOps. All things were created manually many days ago by the team lead at the project start.

I will use resources from previous demo.

*Command + K in terminal*

*Navigate to docs/demo#2/azure\_dev\_ops/extensions/*

*Show branch policies*

The provider is `microsoft/azuredevops`.

First, authentication require.

*Copy token and add into .env file*

First of all, we have to import data about the already existing project.

Syntax `data "type" "name"` the same as `resource "type" "name"`. But it will not try to create resources, it will try to find existing resources and use them as data sources.

Next step is import repositories as data sources. We need to know repository `name` and `project\_id`. We will import 2 repositories from previous example.

I would like to set 2 or more reviewers to each pull request.

We have:

* `project\_id`
* `reviewer\_count` equals 2
* several additional options
* in sections `scope` reference repositories

Additionally, lets change projects features.

*Check branches policies in repositories*

When we have any branch policy, we see special icon to branch. No icons at all.

*Run terminal commands*

*Explain planned changes*

*Check project and repository*

*Check icons*

*Do you have any questions?*

What next? Set possible merge types in repositories.

*Try to find actual values for main branch*

What to do? All the same as for reviewer’s policy:

* `project\_id`
* several additional options
* policy settings
* in sections `scope` reference repositories

*Run terminal commands*

*Explain planned changes*

*Check project and repository*

*Refresh project settings page*

*Do you have any questions?*

**Slide 10 [ DEMO #3 ] [00:15:00]**

Next important feature is input variables.

On scree you see 2 variables. Both have type string and define validating rules. First validation check stage name, possible only dev and prod, but list can be extended. Second validation rule more interesting. We are expecting version in semver format. We use regular expression for checking match with template and after that check number of matches. We can pass values for variables as command line arguments, create special file with variables or use Environment variables as values for terraform.

*Is it clear? Any questions?*

**Slide 11 [ DEMO #3 ] [00:15:00]**

Next important feature is local values. Local values similar to local variables in some method. We can initialize local values in runtime, create based on input values, use functions etc.

*Is it clear? Any questions?*

**Slide 12 [ DEMO #3 ] [00:01:00]**

And final feature for the next demo is a output values. We can define this values and after success terraform execution we can get all require information. On the screen we put in output host name. But it can be any important value. For example, app service name, SQL server name, name of resource group etc.

It very useful in DevOps pipelines. My actual project release pipeline does not have hardcoded values app services and database, we parse all require for deploy information from terraform output.

*Is it clear? Any questions?*

**Slide 13 [ DEMO #3 ] [00:15:00]**

All that we see in previous demo looks good, but how to use it in real world?

*Command + K in terminal*

*Navigate to docs/demos#3*

We will create several resources:

- 2 Azure App Services

- 1 SQL Server

- 1 SQL database

- 1 KeyVault

- 1 KeyVault secret for storing connection string

- Access policy for KeyVault secrets access

- 1 Managed identity for secure connection between backend and KeyVault

- Password generation for SQL Administrator account

- Link all resources together

Additionally, we will use input variables, local variables, collections, naming pattern, set configuration for provider, outputs and more.

Note: This Terraform file correct, we can validate/plan changes/deploy all resources. Terraform will create in Azure all resources without any problems. But part of important settings missed. For example, SKUs for several resources or security settings.

You can use this file, but please be careful. Think and read documentation for any resource/identifier.

And again, I'll run several commands because it takes a lot of time.

*Navigate to docs/demos#3*

*Command + K in terminal*

*Run commands from readme file*

In case of any questions, please interrupt me and ask.

Let’s review main.tf file:

* provider
  + We use azure provider
  + We have provider configuration for KeyVault in our provider. It is global and applicable for any KeyVault in any terraform file in current directory
  + Provider configuration disable soft delete and restore soft deleted keys
* random password generator
  + Any SQL Server require Administrator account
  + I generate password. Important moment that password will be generated once when we create SQL Server. Any next run will restore password value from state
  + We set information for password generator. Minimal number of numbers/upper/lower case characters and etc.
  + For special characters default list exits, but we will override this list
* read data about current account
  + Any creation of resources has to be executed based on Azure account. We read this information and will use it little later

*Do you have any questions?*

* stage variable, type of it is a string
  + Stage can be only dev or prod, any other value will fire exception
  + Validation error for variable. It is required. If we have validation, we should have validation message
* ver variable, type of it is a string
  + I can’t define variable with name version. I suppose it is reserved keyword.
  + Validation for version use regex. In pattern we have:
    - It is a simple and wrong example of semantic versioning
  + Important moment that regex return matches, not true or false
  + We are checking number of matches
  + Validation error for variable.
* tags variable, type of it is a map of strings (Dictionary<string,string> in c#)
  + Type and validation as above
  + Prevent ability to pass empty list of tags
  + Default values defined. We can skip it parameter when run terraform
* dev.tfvars files was create for skip variables enter
  + Syntax for variable very simple: key-value and enter in the end of the string
  + I have never use variables file approach, but suppose it will be very simple to generate the same file as a part of CI/CD pipeline

*Do you have any questions?*

* locals allow define local values
  + Local variables very similar to static internal properties with getter only defined in global scope. In any part of application, we can read this value, but can’t change it.
  + Location local value it is simple string. I will use it like Azure resource location.
  + Prefix will be used for resources names; we are using pattern with stage inside. dev/prod possible values for stage, as you remember.
  + Resource Group name and all next names created based on postfix and abbreviation for resource
  + For KeyVault we can’t use some chars, only alphabetical characters allowed, we replace hyphens to empty string
  + SQL Server values just a string, nothing interesting
  + Front-end URL we are build based on values from resource app service front-end
    - I use information known after deploying to create this local variable
    - It will be used for back-end CORS configuration
  + Back-end URL we are build based on local value for resource name
    - It will be used for API URL configuration in our front-end application
    - We can’t repeat the same formatting approach for back-end URL that we use for front-end URL. We will catch cycle dependency error.
    - Back-end and front-end URLs should use equal approach, but for demo only we generate similar values in two different ways
  + SQL Server password saved to local value
  + SQL Database connection string
    - We create value based on template and interpolation terraform syntax
    - And we use information from several other resources to create it
  + We create tags local variable based on input variable tags
    - The main idea is extending list of values in variable
    - We can skip duplicated values. It is not required to enter stage/version to times
    - Other benefit is predefined list of tags, for example team or author. Both tags will be the same for any environment.

*Do you have any questions?*

* Resources. Approach for create resources we already saw several times. I think we can skip part of it and highlight only interesting moments:
  + For resource group we have minimal set of parameters and use local values for define each of it
  + For service plan we have little more configuration:
    - For location and resource group name we use values from resource group resource. We will repeat it for each resource.
    - we have to set SKU because it is require parameter
  + In front-end application only one interesting moment.
    - It is API\_URL app setting. I’m strongly sure that this value valid and will not have problems 404 errors because app service configuration missed, or back-end name an as a result URL was changed.
  + In back-end app service resource we have two interesting moments:
    - Application setting with name AllowedOrigins. Can be used for CORS policy configuration
    - System Assigned Managed identity enabled for App Service. We will use it in KeyVault configuration.
  + For SQL Server we set Administrator account login and password
  + Firewall rule with start and end IP address equals 0.0.0.0 allow any Azure resource connect to SQL Server
  + SQL Database have reference to SQL Server and this is all that require.
  + KeyVault configuration little more difficult
    - We have to set TenantId. It is require parameter. We use datasource.
    - From my side I would like to add that KeyVault is more difficult resource that I saw. KeyVault has soft delete for key, Tenant who control access to it. Sometimes I catch errors when try to deploy or remove it.
  + CLI Access policy for KeyVault
    - As you see, we use the same dataasorce that for key vault
    - We need that because CLI tool have to introduce changes in KeyVault secrets and should have permissions for that
  + Back-end access policy for KeyVault
    - Only one Get permission for secrets
    - Object that will have this permission is a back-end managed identity that we set above for the back-end app service
    - And dependency to CLI access policy. This section allows define priority for access policies. Back-end access policy will be created after CLI access policy. And back-end access policy will be removed before CLI access policy. We did that because CLI access policy allow manage secrets and we need that for create/remove secret.
  + Last resource is KeyVault secret
    - It will store connection string

*Do you have any questions?*

* Outputs, back-end and front-end host names.
  + All that we need is add prefixes `https://` and we have URLs for our app services.
* Outputs, connection string
  + Problem with connection string is password. Sensitive data.
  + I tried to use several different options for this output parameter, but without success result. All that we have in terminal output is a sensitive flag.
  + We can use `terraform output -json` command and in output we will have connection string with password
* Outputs, database SKU
  + We do not have configuration for database SKU
  + Azure provider or azure CLI has default value for it
  + Default value will be used in case identifier missed in resource
* Outputs, database price
  + I suppose more than 300$ is a lot for dev database
  + Be careful with default values for identifiers. It can cost a lot

I think we can skip review for terraform commands and logs. Let’s check Azure resources.

As you see we have list of resources, expected names, all created, and all exists. Possible to see list of KeyVault secrets. App services have expected app settings.

*Do you have any questions? Possibly would like to check something in resource group.*

**Slide 14 [00:05:00]**

After several examples let’s go back to Infrastructure as a Code slide and check all pros and cons.

Let’s talk about Pros:

pros 1-2 already discussed:

pros 4: In case of any problems, it's hard to imagine how you give admin access to your cloud to some stranger from the Internet, who promised to help. But possible to check <https://stackoverflow.com>, or <https://github.com>. As we saw before, infrastructure is a text file with all possible benefits.

prop 5: I can create GitHub repository/define build pipelines/create Amazon and Azure services/create mailbox/manage Jira in one way with one approach based on one coding language.

prop 6: I know that all environment fully equals and have to work in the same way without any exclusions. In most cases, there are no problems in different operations of one application in different environments.

prop 7: Minimal changes in infrastructure files will allow create copy of some environment. I’m talking about dev/test/aqa/staging/pre-prod/prod and etc. Reconfigure all together.

prop 8: And last point about coding language. Most often the programming language is understandable and simple for engineers, has good documentation and is easy to understand. But it depends on the specific implementation.

Cons is very common and applicable for any new language/approach/framework:

cons 1: Complexity increases because we have to use a new approach/tool with all its features and problems.

cons 2: Dependencies to versions and implementation of providers. Infrastructure as a code is not magic. It is using some kind of intermediate mechanisms like HTTP API calls, CLI tools command, communication with demons/agents and etc. It is a possible situation when you have to use some feature in a cloud, but your Infrastructure as a Code tool does not support it yet. Or you did update for provider version and part of your file triggers a warning about deprecation or already removed from the driver at all.

\*Changes in language syntax possible too.

cons 3: More long process for changes. You have to review the code, pass gates and etc. But who knows how long it will take to find the issue after manual configuration?

cons 4: Infrastructure language have to be simple with minimal functionality and files can grow very quickly and be very difficult to understand/refactor/fix/extend/update.

\*Depends on implementation.

cons 5: If execution fails somewhere, it may not be as easy to restart from the exact same point, and re-executing from scratch may take a long time. The mechanism of communication with infrastructure is a black box and who knows how exactly does it work? In any case It has bugs, not clear error messages, unexpected fails and etc. Internet problems can cause a 30-minute process to fail.

cons 6: If we talk about me personally, I have not fully learned F#, React, Benchmark, Cypress, GraphQL, SAFe, and tens of other unknown for Microsoft Word words. And work with something new. Infrastructure as a Code is a complex and difficult in some cases approach. It will take time to study. It will take time to actualize knowledge.**Slide 9 [00:01:00]**

When we try googling terraform alternatives, first result page contains information about tool on the slide.

I know and have experience only with part of the list.

Based on experience with terraform, I would like to say that:

- tools from the list have different functionality

- part of features missed

- other tools have extra features

- several tools can’t be used like Terraform because cover only small part of features

- several tools can do more and have extra features

- part of it open-source, other not

- several can work only with specific cloud

**Slide 16 [00:04:00]**

Copy-paste from terraform official site you can see on screen.

Why I like Terraform:

* Terraform opensource and I can check how it works. I used it once and it was helpful.
* Terraform is provisioning tool. We can do a lot with different providers. But can use terraform like configuration tool too.
* Terraform cloud agnostic. It doesn’t mean that app services can be created in Azure/Google/Amazon clouds with one piece of code. It means that approach the same, names and types of resources cloud specific.
* Terraform use state files and do not have dependency to some server/agents/demons. Not require update it, manage etc.
* Difficult to compare Terraform community with another community for Terraform alternatives. But looks like Terraform has the biggest community.
* Declarative syntax. Simple and clear. From another site terraform files can be huge, and modules try to solve it.
* Terraform modern tool but cover a lot and can do a lot.
* In case any manual changes terraform will try to roll it back. But it depends on provide and implementation. I saw how it works sometimes and sometimes not.

Terraform not an ideal tool that can solve all possible problems but cover almost all of them. And of course, add new. But the quantity and quality of positive points covers all negative points.

In case you are thinking about using it or not. My advice is to try to use it.

**Slide 11 [00:00:00]**

Do you have any questions?**Slide 12 [00:01:00]**

From my side this is all.

Please scan QR code in case you like to review examples from demos.