TRIGLAV OS · QUINZEE · DEVELOPER'S GUIDE

Volume 2: The Book of Quinzee

TOMAZ STIH

FIRST EDITION



Development edition · 16 November 2026. Commands using repository paths, Make targets or sample projects require access to the Quinzee source checkout. No public installer is available. Examples use disposable local environments.

Your path through Quinzee

Build a working application, learn each service, and provision its infrastructure with Bicep and ARM.

1. Meet Quinzee and its services
2. Run your first application
3. Prepare a C# development container
4. Sign in and create a resource group
5. Deploy your application's resources with Bicep
6. Store documents in Blob Storage
7. Process work with Queue Storage
8. Keep application secrets in Key Vault
9. Manage settings with App Configuration
10. Use PostgreSQL, MySQL and Redis
11. Observe your application
12. Deliver work with Service Bus
13. Run HTTP and Service Bus functions
14. Manage the environment with QHQ
15. Stop, restart and clean up safely
16. Troubleshoot a connection or workflow
17. Build Quinzee from source
18. Certificates, DNS and Azure URLs
19. Debug your application and QHQ
20. Run the emulator natively
21. Give applications and people the right identity
22. Host a web application
23. Service compatibility and limits
24. Turn infrastructure into repeatable deployments
25. Provision storage, secrets and configuration
26. Provision event delivery and durable messaging
27. Provision databases and a cache with real engines

28. Deploy application hosts and their identities
29. Deploy monitoring with the application

Commands marked **host** run from the repository root. Each exercise identifies its prerequisites, the place to run commands and the expected result. Use a disposable environment while learning.

DEVELOPER'S GUIDE

1. Meet Quinzee and its services

Quinzee Cloud is a local environment for applications that use Azure services. It accepts a supported subset of Azure's APIs, so you can create resources and use them through ordinary Microsoft client libraries. Your application can use URLs such as `https://management.azure.com` and `https://myaccount.blob.core.windows.net` inside a prepared development container. Quinzee's DNS sends those requests to your local emulator; its development certificates allow the client to verify the connection.

A resource group is a container for related resources. A **control-plane** call creates or configures a resource, such as a storage account. A **data-plane** call uses that resource, such as uploading a document. Azure Resource Manager handles the control plane; the individual service clients handle the data plane.

Services you can use

Service	What you can do locally	Main boundary
Microsoft Entra ID compatibility	Authenticate applications and local users; obtain service-specific bearer tokens	Client credentials, local browser login and Functions managed identities; bounded ARM/Service Bus roles, no MFA or federation
Azure Resource Manager	Work with subscriptions, resource groups, provider discovery and resource inventory	Existing local tenant; no Azure billing or subscription infrastructure
ARM deployments and Bicep	Deploy resource-group templates, inspect operations/outputs and redeploy incrementally	Supported expression subset; no Complete mode, nested deployments or resource loops
Blob Storage	Containers, block blobs, metadata, ranges, staged uploads and scoped service SAS	No page/append blobs, leases, snapshots or Azure Files
Queue Storage	Send, peek, receive, update and delete messages; visibility leases and redelivery	At-least-once delivery; bounded local capacity; no Queue SAS
Table Storage	Typed entities, filters, paging, conditional updates and entity group transactions	No Table SAS, ACLs or service properties
Event Grid	Custom/storage system topics, filtered WebHook and StorageQueue deliveries with retries	No advanced filters, dead-lettering, namespaces or MQTT

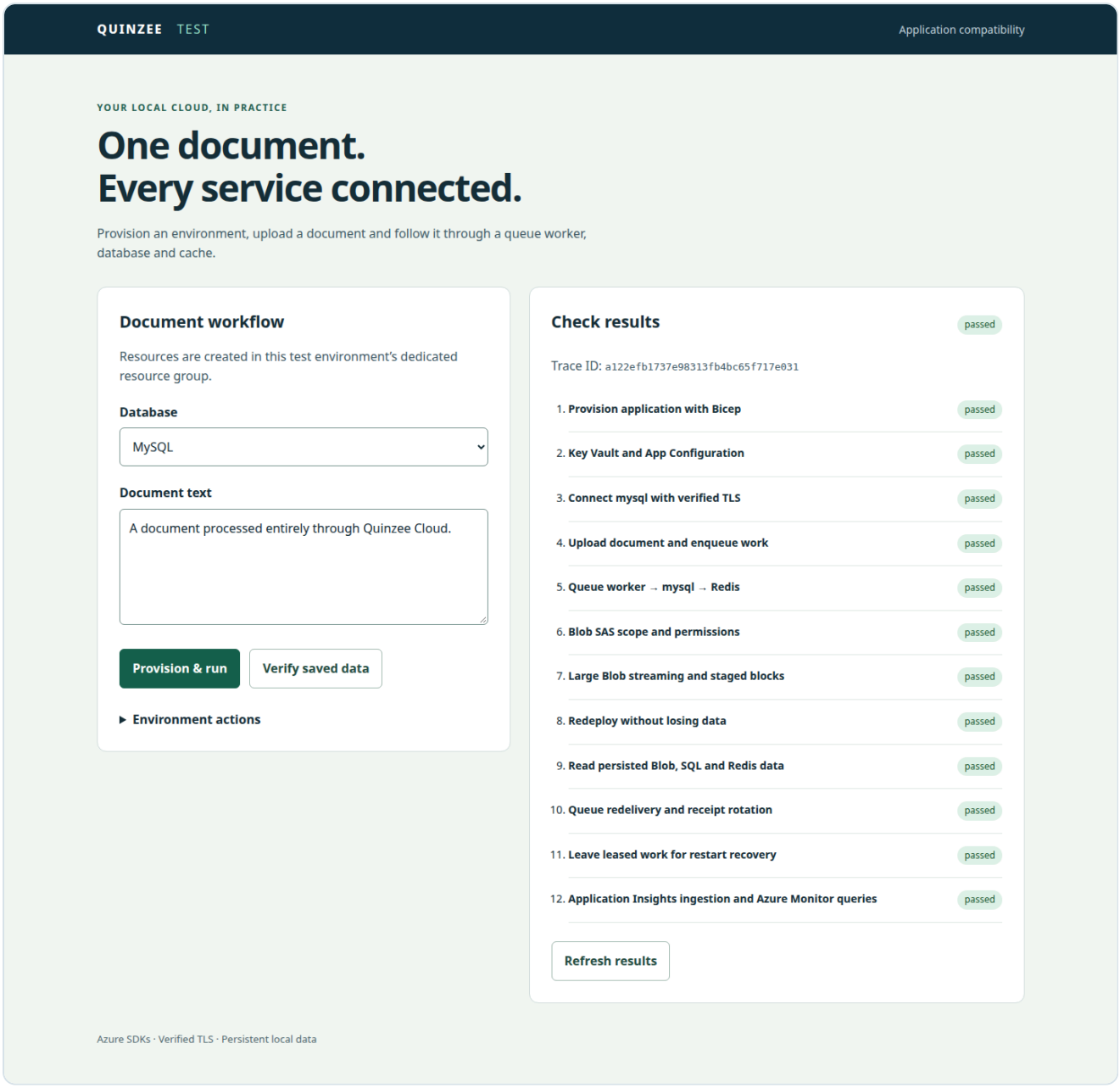
Service	What you can do locally	Main boundary
Service Bus	Queues, topics/subscriptions, AMQP/TLS and HTTPS messaging, locks and settlement	No sessions, batches, transactions, scheduling or WebSockets
Azure App Service	Real Linux web containers, plans, settings, lifecycle and managed identities	TLS 8443 container contract; no slots, autoscale, Windows hosting or cloud synchronization
Shared LDAP directory	Read/authenticate the same Entra users and groups over LDAPS	Optional PostgreSQL/OpenLDAP; no LDAP writes, Kerberos or domain join
Azure Functions	Real .NET isolated HTTP and Service Bus queue triggers in registered Linux images	Separate Docker helper; single instance, no autoscaling, slots or other workers
Key Vault	Versioned secrets, attributes, soft deletion, recovery and purge	Secrets only; no cryptographic keys/certificates or Azure RBAC enforcement
App Configuration	Keys, labels, revisions, conditional updates and locking	No snapshots, replicas or full feature-management service
PostgreSQL flexible servers	Provision a real PostgreSQL 18 server and databases; start, stop and restart	Local container, password login; no HA, backups or managed-identity database login
MySQL flexible servers	Provision a real MySQL 8.4 server and databases; start, stop and restart	Same local lifecycle limits as PostgreSQL
Managed Redis-compatible resources	Provision an encrypted cache and connect with a Redis client	Valkey engine, one nonclustered database; no key rotation or Redis cluster simulation
Log Analytics	Create workspaces and query ingested application telemetry	Small, explicit KQL subset and bounded retention/storage
Application Insights	Create workspace-backed components; ingest request, dependency, log, event, exception and metric envelopes	Supported official exporter configuration; no live metrics, profiler or alerting
Local DNS and platform observability	Resolve service names; inspect health, workers, queue depth, logs, metrics and traces	Development/operator facilities, not Azure DNS zones or the Azure Portal

Quinzee implements these services for application development. Real Azure's permissions, capacity, regional resilience and pricing are not simulated. The bootstrap identity has broad local access. New users and principals need supported scoped ARM/Service Bus permissions; other data planes have explicit limits. Storing an Azure-shaped policy property is not proof of enforcement. Use synthetic development data. Secrets and application data persist locally, and secret values are not encrypted at rest by the emulator.

What runs where

- **quinzee.cloud** serves ARM and HTTP data-plane APIs over HTTPS, and Service Bus AMQP over a separately enabled TLS listener.
- **quinzee.dns** resolves supported service names on a private Docker network.
- A separate **engine supervisor** starts and reconciles PostgreSQL, MySQL and Valkey containers. SQL and Redis clients speak their real native protocols.
- A separate **Functions helper** runs registered application images with the real Functions host. Its containers execute HTTP and Service Bus triggers.
- **quinzee.test** is a working .NET/React example application you can inspect in a browser. Separate locked examples exercise Service Bus and Functions in chapters 12–13.
- **QHQ** is the native management application for resources and diagnostics.
- Optional **Grafana, Prometheus, Loki, Tempo and an OpenTelemetry Collector** provide dashboards and trace/log exploration.

The sample environment prepares its own DNS, public trust certificates and credentials. It does not alter your host's DNS or trust store.



The supplied application brings the services together. This completed MySQL run includes Blob, Queue, Key Vault, App Configuration, Redis and monitoring checks.

Which Microsoft library?

The service libraries are the **Azure SDK for .NET**: for example, `Azure.Storage.Blobs`, `Azure.Security.KeyVault.Secrets` and `Azure.Monitor.Query.Logs`. They share Azure SDK conventions and accept a `TokenCredential` supplied by `Azure.Identity`.

You may be remembering **MSAL**, the Microsoft Authentication Library. MSAL is about authentication; it is not a Blob or Key Vault client. This book uses `ClientSecretCredential` from `Azure.Identity` so that the service libraries obtain the appropriate tokens for you. It deliberately avoids an interactive credential chain that might select a real Azure account on your workstation.

A `TokenCredential` obtains the token requested by a service client. The token's audience identifies that service; a token requested for Resource Manager cannot be reused as a directory or messaging token. Use the verified package versions in the workshop and check the returned HTTP status before assuming a newer SDK operation is supported.

Related reading

- Quinzee Administrator’s Guide: Start with a small, understood installation
- Quinzee Administrator’s Guide: Connect QHQ and make changes deliberately
- Quinzee Internals: Architecture and trust boundaries
- Quinzee Internals: The compatibility application
- Scenario overview: Developing for Azure Without Using Azure
- Scenario overview: Learning How the Cloud Works
- Scenario overview: Building a Complete Cloud Application from Scratch

2. Run your first application

The quickest introduction is the supplied document workflow. It provisions its own resources, uploads a document, sends queue work, writes SQL data, caches a result and queries its telemetry.

Step 1: check the prerequisites

Use a Linux x86-64 development machine with Docker Engine and Compose v2, GNU Make, Python 3, OpenSSL, .NET SDK 10, and Node.js 24 LTS or a compatible newer LTS. The first build needs access to package registries and container images. The runtime image is `docker-dotnet:latest` from the sibling Triglav-OS project. Build it there with `make -j8 docker-dotnet` if it is unavailable. Quinzee can load its exported image from `../os`; set `TRIGLAV_OS_DIR` for a different checkout.

Check the commands on the **host**:

```
docker version
docker compose version
dotnet --version
node --version
python3 --version
openssl version
```

The application helper builds the cloud image and the .NET/React application; you do not need to build the native QHQ application first. If you want QHQ or a native emulator binary, follow Building from source.

Step 2: start the environment

From the repository root on the **host**:

```
make application-observability-up
```

Leave this terminal running. Initial setup downloads/builds dependencies, prepares native engine images, creates private certificates and starts the cloud, DNS, application, engine supervisor and telemetry tools. Subsequent runs reuse the environment in `.local/application/development`.

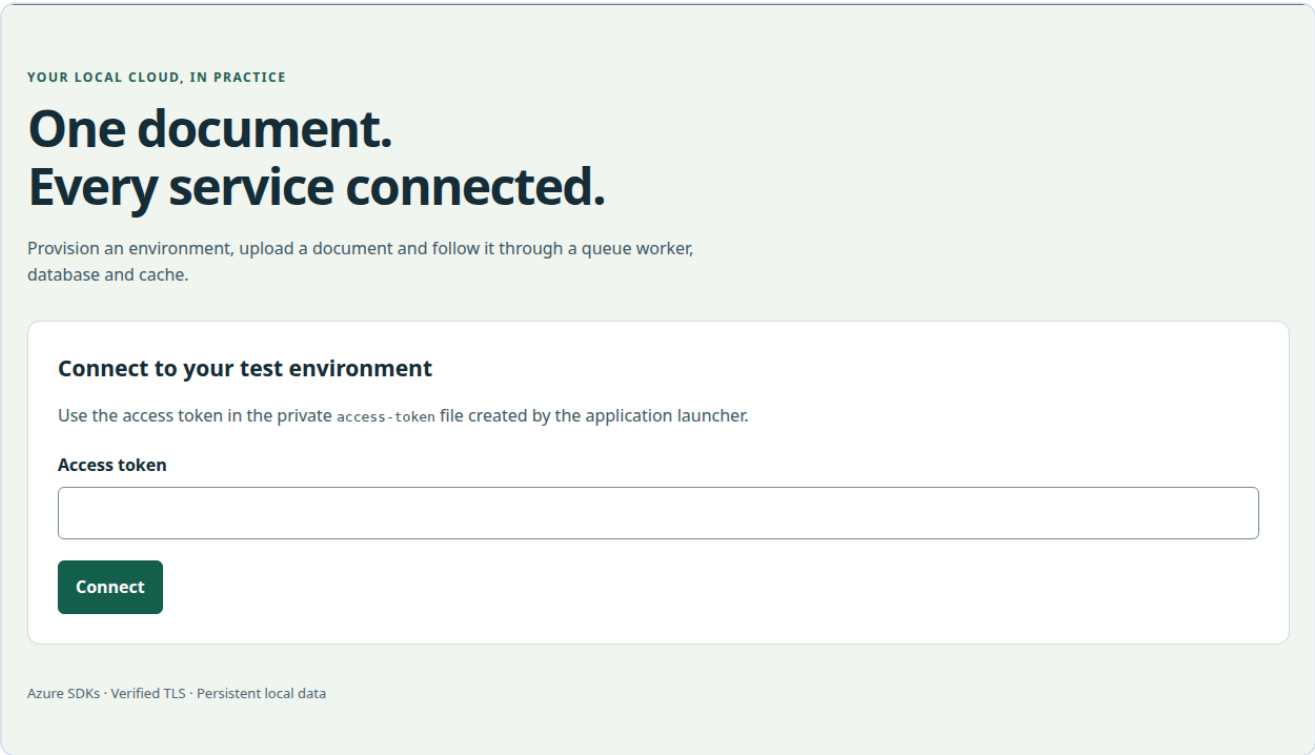
The command prints the application URL (normally **https://localhost:3000**), the public browser CA path and the path to an application access-token file. Wait for those messages before opening the browser. Build and environment logs are in `.local/application/development/environment.log`.

For the same environment without the external dashboards, use `make application-up` instead. Application Insights and Log Analytics APIs still work; those services run inside Quinzee itself.

Step 3: open and sign in

1. Trust the **public development CA certificate printed by the command** in your development browser. This environment has its own CA, separate from the one used by `make cloud-up`. Do not import a private `.key` file.
2. Open the printed HTTPS URL. The browser should accept the certificate for `localhost`; do not bypass a certificate warning.
3. Read `.local/application/development/access-token` privately and paste the token into the application's sign-in form.

This browser token unlocks the example application's UI. It is not an Azure bearer token or the application's client secret. Azure credentials remain in the backend's private settings; the browser does not receive them.



Connect with the private token printed by your launcher. The image intentionally shows the form before a token is entered.

Step 4: process a document

Select **PostgreSQL** and leave the initial document text in place. Start the workflow. Each step shows its progress, including provisioning, Key Vault and App Configuration access, Blob and Queue operations, SQL/Redis writes and Application Insights queries. A successful run shows a trace ID.

Repeat with **MySQL**. The template provisions both SQL engines; the selection controls which one processes the document. This uses real database servers, not in-memory mock SQL responses.

Choose **Verify** with the same database and document to check persisted data and recover the leased queue item left by the run. A verification consumes that recovery item: run the full workflow again before repeating that recovery check. **Cancel** stops client work, but does not roll back resources already created. **Cleanup** deletes this sample application's resource group and its data.

Step 5: find the trace

Grafana is normally **https://localhost:13001**. Trust `.local/observability/ca.crt`, sign in as `admin` with the password from `.local/observability/grafana-password`, and open the **Quinzee observability** dashboard. In Explore, select Tempo and look up the workflow trace ID. The trace can include application, cloud and engine spans.

Use the monitoring chapter to understand the separate platform dashboards and Azure-compatible application telemetry APIs.

Step 6: continue or stop

Keep the environment running for the next chapter. To finish a session, first exit any development containers you joined to its network, then press Ctrl+C in the application terminal. Its state and native data volumes are retained. Stop the separately operated dashboard tools with:

```
make observability-down
```

Restart with the same application command to reuse your environment. Use application/resource-group cleanup only when you intend to delete data.

Related reading

- Quinzee Administrator’s Guide: Start with a small, understood installation
- Quinzee Administrator’s Guide: Connect QHQ and make changes deliberately
- Quinzee Internals: Architecture and trust boundaries

- Quinzee Internals: The compatibility application
- Scenario overview: Developing for Azure Without Using Azure
- Scenario overview: Learning How the Cloud Works
- Scenario overview: Building a Complete Cloud Application from Scratch

3. Prepare a C# development container

Keep the chapter 2 environment running. You will attach a separate .NET SDK container to its private network. Edit C# files with your host editor and run `dotnet` inside that container. The source files remain on your host.

The supplied `quinzee.test` container already has working DNS and CA trust, but contains the .NET **runtime**, not an interactive SDK toolchain. Do not install development tools into it. The separate SDK container below shares the network and public trust bundle without receiving CA signing keys or a Docker socket. The SDK image includes the compiler and build tools. A runtime image contains what an already-built application needs to execute, so `dotnet build` belongs in the SDK container below.

Step 1: prepare a private tutorial directory

Run this on the **host**, from the repository root, after the application prints its URL:

```
umask 077
mkdir -p .local/book/project
python3 - <<'PYSETUP'
from pathlib import Path
import copy, json, secrets, shutil
import xml.etree.ElementTree as ET
base = Path('.local/application/development')
source = json.loads((base / 'client/settings.json').read_text())
book = Path('.local/book')
settings = book / 'settings.json'
if not settings.exists():
    prefix = 'bk' + secrets.token_hex(4)
    settings.write_text(json.dumps({
        **{key: source[key] for key in ('tenantId', 'clientId', 'clientSecret',
'subscriptionId')},
        'prefix': prefix, 'group': 'quinzee-book-' + prefix,
        'databasePassword': secrets.token_urlsafe(24)
    }))
    settings.chmod(0o600)
# Reuse the repository's verified package versions and complete lock file.
project = ET.Element('Project', Sdk='Microsoft.NET.Sdk')
properties = ET.SubElement(project, 'PropertyGroup')
for name, value in {'OutputType':'Exe', 'TargetFramework':'net10.0',
                    'ImplicitUsings':'enable', 'Nullable':'enable',
                    'RestorePackagesWithLockFile':'true'}.items():
    ET.SubElement(properties, name).text = value
items = ET.SubElement(project, 'ItemGroup')
ET.SubElement(items, 'FrameworkReference', Include='Microsoft.AspNetCore.App')
for package in ET.parse('test/backend/Quinzee.Test.csproj').findall('.//PackageReference'):
```

```
    items.append(copy.deepcopy(package))
ET.indent(project)
ET.ElementTree(project).write(book / 'project/BookClient.csproj', encoding='unicode')
shutil.copyfile('test/backend/packages.lock.json', book / 'project/packages.lock.json')
PYSETUP
```

This creates independent tutorial resource names and a database password while using the local tenant's application identity. It never prints secrets. Reusing this directory reuses the same tutorial resources. If you recreate the entire application environment, prepare a new tutorial directory for its new identity.

The packages include `Azure.Identity`, the ARM/deployment clients, Blob and Queue clients, Key Vault, App Configuration, Azure Monitor exporters/query clients, and Npgsql/MySqlConnector/StackExchange.Redis. The extra ASP.NET Core framework reference makes the logging examples available to this console app. The generated project and caches stay under Git-ignored `.local/`.

Step 2: attach the SDK container

On the **host**:

```
book_env="$PWD/.local/application/development"
book_network=$(python3 -c 'import json,sys; print(json.load(open(sys.argv[1]))["project"] +
"_quinzee")' "$book_env/environment.json")
book_dns=$(python3 -c 'import json,sys; print(json.load(open(sys.argv[1]))["networkPrefix"] +
".53")' "$book_env/environment.json")
docker run --rm -it \
  --user "$(id -u):$(id -g)" --cap-drop ALL \
  --security-opt no-new-privileges \
  --network "$book_network" --dns "$book_dns" \
  --mount "type=bind,src=$PWD/.local/book/project,dst=/work" \
  --mount "type=bind,src=$PWD/.local/book/settings.json,dst=/run/book/settings.json,readonly" \
  --mount "type=bind,src=$book_env/client/trust.pem,dst=/run/book/trust.pem,readonly" \
  --mount "type=bind,src=$book_env/client/engine-ca.crt,dst=/run/book/engine-ca.crt,readonly" \
  --env SSL_CERT_FILE=/run/book/trust.pem \
  --env DOTNET_CLI_HOME=/tmp --env NUGET_PACKAGES=/work/.nuget \
  --env DOTNET_CLI_TELEMETRY_OPTOUT=1 \
  --env APPLICATIONINSIGHTS_STATSBEAT_DISABLED=true \
  --workdir /work \

mcr.microsoft.com/dotnet/sdk:10.0@sha256:fe0fd4e93bd2fbb47c883b7a6369e5c8d3db8bf01cdc5981f6ee35fb
df3ff4fb bash
```

The SDK image is pinned to the digest used to verify these examples, and the NuGet dependency graph is locked. Update the image deliberately when adopting .NET servicing updates, then rerun your application checks. The `trust.pem` bundle contains public roots for HTTPS services and native engines, as well as ordinary public internet roots used for package restoration.

Step 3: verify DNS and trust before authenticating

In the **client container**:

```
dotnet --version
getent hosts management.azure.com
getent hosts login.microsoftonline.com
curl --fail --silent --show-error https://management.azure.com/health
```

Both names must resolve to the application environment's cloud address (`networkPrefix` plus `.10` in `environment.json`). The HTTPS response must identify `quinzee.cloud`. If either check fails, fix DNS/trust before running credentialed examples. Joining the network alone does not select Quinzee DNS; `--dns` is essential.

Azure emulation is already enabled by the application helper. On a separately started `make cloud-up` environment, enable it explicitly in QHQ's DNS page; see certificates and DNS.

Step 4: restore and begin editing

In the **client container**:

```
dotnet restore --locked-mode
```

On the **host**, open `.local/book/project` in your editor. In chapter 4, create `BookContext.cs` and `Program.cs` there. Build/run in this container with:

```
dotnet run --no-restore
```

Each following chapter replaces `Program.cs`. The context, project, lock file, compiled template and deployment outputs remain in the same directory. You can use your editor's container-attach feature if you prefer running its terminal and debugger in the SDK container. No host-wide CA installation or DNS change is needed for this development path.

Moving your application to Azure

Keep credentials and resource endpoints in configuration. The C# service clients use normal Microsoft URLs; remove the local DNS/trust setup and supply actual Azure credentials/resources in the Azure environment. First check your app against real Azure permissions and any features outside the local subset. Compatibility here is not evidence that production RBAC, resilience or scale has been tested.

Related reading

- Quinzee Administrator’s Guide: Start with a small, understood installation

- Quinzee Administrator’s Guide: Connect QHQ and make changes deliberately
- Quinzee Internals: Architecture and trust boundaries
- Quinzee Internals: The compatibility application
- Scenario overview: Developing for Azure Without Using Azure
- Scenario overview: Learning How the Cloud Works
- Scenario overview: Building a Complete Cloud Application from Scratch

4. Sign in and create a resource group

An application signs in using a **tenant ID**, **client ID** and **client secret**. Quinzee creates a development tenant, subscription and application identity on first startup. The application environment supplies these values to the private tutorial settings created in chapter 3. No real Azure subscription or `az login` is involved.

`ClientSecretCredential` requests tokens from `login.microsoftonline.com`. `ArmClient` uses the token for `management.azure.com`. Your container's DNS and CA setup make both requests local while retaining HTTPS verification.

Step 1: create the shared context

Save the following as **BookContext.cs** in `.local/book/project` using your host editor. Keep it for all subsequent examples. It reads private configuration, constructs the ordinary Azure clients and supplies a small helper for ARM operations that the examples call directly.

```
using Azure.Core;
using Azure.Identity;
using Azure.ResourceManager;
using System.Net.Http.Headers;
using System.Text.Json;

public sealed record BookSettings(
    string TenantId, string ClientId, string ClientSecret,
    string SubscriptionId, string Prefix, string Group, string DatabasePassword);

public sealed class BookContext
{
    public BookSettings Settings { get; }
    public TokenCredential Credential { get; }
    public ArmClient Arm { get; }
    public string GroupId =>
        $"/subscriptions/{Settings.SubscriptionId}/resourceGroups/{Settings.Group}";

    private BookContext(BookSettings settings)
    {
        Settings = settings;
        Credential = new ClientSecretCredential(
            settings.TenantId, settings.ClientId, settings.ClientSecret);
        Arm = new ArmClient(Credential, settings.SubscriptionId);
    }

    public static BookContext Load()
    {
        var settings = JsonSerializer.Deserialize<BookSettings>(
```

```
            File.ReadAllText("/run/book/settings.json"),
            new JsonSerializerOptions { PropertyNameCaseInsensitive = true })
            ?? throw new InvalidOperationException("Missing tutorial settings");
        return new BookContext(settings);
    }

    public Dictionary<string, string> Outputs() =>
        JsonSerializer.Deserialize<Dictionary<string, string>>(
            File.ReadAllText("outputs.json"))
        ?? throw new InvalidOperationException("Run the deployment chapter first");

    public async Task<JsonDocument> PostArmAsync(string providerPath, string version)
    {
        var token = await Credential.GetTokenAsync(
            new TokenRequestContext(["https://management.azure.com/.default"]),
            CancellationToken.None);
        using var http = new HttpClient { Timeout = TimeSpan.FromSeconds(30) };
        using var request = new HttpRequestMessage(HttpMethod.Post,
            $"https://management.azure.com{GroupId}/providers/{providerPath}?api-version=
{version}");
        request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token.Token);
        using var response = await http.SendAsync(request);
        response.EnsureSuccessStatusCode();
        return JsonDocument.Parse(await response.Content.ReadAsStringAsync());
    }
}
```

The client secret is never printed. The settings mount is read-only, and only this development container needs to read it. `BookContext` is example application code, not a Quinzee-specific SDK dependency.

Step 2: create the group

Save this as **Program.cs**, next to `BookContext.cs`:

```
using Azure;
using Azure.Core;
using Azure.ResourceManager.Resources;

var book = BookContext.Load();
var subscription = book.Arm.GetSubscriptionResource(
    new ResourceIdentifier($"/subscriptions/{book.Settings.SubscriptionId}"));

var operation = await subscription.GetResourceGroups().CreateOrUpdateAsync(
    WaitUntil.Completed,
    book.Settings.Group,
    new ResourceGroupData(AzureLocation.WestEurope));

Console.WriteLine($"Ready: {operation.Value.Data.Name}");
await foreach (var group in subscription.GetResourceGroups().GetAllAsync())
    Console.WriteLine($"{group.Data.Name}: {group.Data.Location}");
```

In the **client container**, run `dotnet run --no-restore`. Expect a group named `quinzee-book-bk…`. Creating it again updates the same group. Other groups listed belong to other local exercises; do not delete them as part of this tutorial.

`WestEurope` is resource metadata here. Choosing it does not contact Azure or select a remote machine. The emulator uses the configured local services.

Understand authentication errors

The browser access token, an Azure bearer token and a client secret serve different purposes. Only the client secret belongs in `ClientSecretCredential`. The SDK gets different bearer tokens for ARM, Storage, Key Vault and monitoring; an ARM token cannot be substituted for a monitoring query token.

Tokens expire after an hour and the cloud's signing key changes on restart. Create a fresh credential/client after restarting the cloud if a cached token keeps receiving 401. Each `dotnet run` here starts a new process. An `invalid_client` error usually means the settings came from a different local environment.

This tutorial uses the generated bootstrap application, which retains broad local privileges. New applications and users start without permissions. Quinzee also supports service-principal management, Functions managed identities, scoped ARM/Service Bus roles and local browser login with PKCE. See application identities and user sign-in for the separate examples and their compatibility limits.

Related reading

- Quinzee Administrator’s Guide: Operate the shared Entra and LDAP directory
- Quinzee Internals: TLS, identity and DNS
- Quinzee Internals: One directory behind HTTPS and LDAP
- Scenario overview: Testing Security and Access Controls

5. Deploy your application's resources with Bicep

Bicep describes the resources your application needs. Microsoft's compiler turns it into an ARM JSON template; the deployment SDK sends that template to Quinzee. Resource creation and readiness can take time, so the example waits for the deployment to finish before reading its outputs.

Step 1: copy and compile the complete example

On the **host**, from the repository root:

```
cp test/application.bicep .local/book/application.bicep
python3 scripts/deps/bicep.py build .local/book/application.bicep \
  --outfile .local/book/project/application.json
```

The helper downloads and verifies the repository's recorded official Bicep compiler if necessary. No Azure CLI login is needed. Edit your copy when experimenting, then compile it again.

The copied file is shown here in full. It creates the application's storage, secret/configuration services, SQL/cache engines and monitoring resources. `stem` normalizes the supplied prefix into service-compatible resource names. `@secure()` protects the database input from deployment history; it is still a secret in your private input file. Resource references create dependency order.

```
targetScope = 'resourceGroup'

@minLength(3)
@maxLength(24)
param prefix string
var stem = take(toLower(replace(prefix, '-', '')), 12)
param tenantId string
param location string = resourceGroup().location

resource storage 'Microsoft.Storage/storageAccounts@2023-05-01' = {
  name: '${stem}store'
  location: location
  kind: 'StorageV2'
  sku: { name: 'Standard_LRS' }
}
resource vault 'Microsoft.KeyVault/vaults@2023-07-01' = {
  name: '${stem}-vault'
  location: location
```

```
  properties: {
    tenantId: tenantId
    sku: { family: 'A', name: 'standard' }
    enableRbacAuthorization: true
    accessPolicies: []
  }
  dependsOn: [storage]
}
resource configuration 'Microsoft.AppConfiguration/configurationStores@2024-05-01' = {
  name: '${stem}-config'
  location: location
  sku: { name: 'standard' }
  dependsOn: [vault]
}
resource container 'Microsoft.Storage/storageAccounts/blobServices/containers@2023-05-01' = {
  name: '${storage.name}/default/documents'
  properties: { publicAccess: 'None' }
}
resource queue 'Microsoft.Storage/storageAccounts/queueServices/queues@2023-05-01' = {
  name: '${storage.name}/default/documents'
  properties: { metadata: { application: 'quinzee-test' } }
}
output storageName string = storage.name
output queueEndpoint string = storage.properties.primaryEndpoints.queue
output blobEndpoint string = storage.properties.primaryEndpoints.blob
output vaultEndpoint string = vault.properties.vaultUri
output configurationEndpoint string = configuration.properties.endpoint

param administrator string = 'appadmin'
@secure()
param password string

resource postgres 'Microsoft.DBforPostgreSQL/flexibleServers@2024-08-01' = {
  name: '${stem}-postgres'
  location: location
  properties: {
    administratorLogin: administrator
    administratorLoginPassword: password
    version: '18'
  }
}
resource pgDatabase 'Microsoft.DBforPostgreSQL/flexibleServers/databases@2024-08-01' = {
  parent: postgres
  name: 'application'
  properties: {}
}
resource mysql 'Microsoft.DBforMySQL/flexibleServers@2023-12-30' = {
  name: '${stem}-mysql'
  location: location
  properties: {
    administratorLogin: administrator
    administratorLoginPassword: password
    version: '8.4'
  }
}
resource myDatabase 'Microsoft.DBforMySQL/flexibleServers/databases@2023-12-30' = {
  parent: mysql
  name: 'application'
  properties: {}
}
resource cache 'Microsoft.Cache/redisEnterprise@2025-04-01' = {
```

```
  name: '${stem}-cache'
  location: location
  sku: { name: 'Balanced_B0' }
  properties: { minimumTlsVersion: '1.2' }
}
resource cacheDatabase 'Microsoft.Cache/redisEnterprise/databases@2025-04-01' = {
  parent: cache
  name: 'default'
  properties: {
    clientProtocol: 'Encrypted'
    clusteringPolicy: 'NoCluster'
    evictionPolicy: 'NoEviction'
  }
}
output postgresHost string = postgres.properties.fullyQualifiedDomainName
output mysqlHost string = mysql.properties.fullyQualifiedDomainName
output cacheHost string = cache.properties.hostName

resource workspace 'Microsoft.OperationalInsights/workspaces@2023-09-01' = {
  name: '${stem}-logs'
  location: location
  properties: {
    sku: { name: 'PerGB2018' }
    retentionInDays: 30
  }
}
resource insights 'Microsoft.Insights/components@2020-02-02' = {
  name: '${stem}-insights'
  location: location
  kind: 'web'
  properties: {
    Application_Type: 'web'
    WorkspaceResourceId: workspace.id
    DisableLocalAuth: true
  }
}
output workspaceId string = workspace.properties.customerId
output insightsResourceId string = insights.id
output insightsConnectionString string = insights.properties.ConnectionString
```

The later infrastructure workshop breaks this combined deployment into complete per-service lessons and explains the ARM JSON emitted by the compiler. Here the goal is to provision everything the C# application chapters will use.

Step 2: deploy with the Microsoft ARM SDK

Replace **Program.cs** with:

```
using Azure;
using Azure.Core;
using Azure.ResourceManager.Resources.Deployments;
using Azure.ResourceManager.Resources.Deployments.Models;
using System.Text.Json;

var book = BookContext.Load();
var group = book.Arm.GetResourceGroupResource(new ResourceIdentifier(book.GroupId));
```

```
var parameters = new Dictionary<string, object>
{
    ["prefix"] = new { value = book.Settings.Prefix },
    ["tenantId"] = new { value = book.Settings.TenantId },
    ["password"] = new { value = book.Settings.DatabasePassword }
};
var content = new ArmDeploymentContent(
    new ArmDeploymentProperties(ArmDeploymentMode.Incremental)
    {
        Template = BinaryData.FromString(await File.ReadAllTextAsync("application.json")),
        Parameters = BinaryData.FromObjectAsJson(parameters)
    });

var deployment = (await ResourcesDeploymentsExtensions.GetArmDeployments(group)
    .CreateOrUpdateAsync(WaitUntil.Completed, "book", content)).Value;
using var document = JsonDocument.Parse(deployment.Data.Properties.Outputs.ToString());
var outputs = document.RootElement.EnumerateObject().ToDictionary(
    property => property.Name,
    property => property.Value.GetProperty("value").GetString()!);
await File.WriteAllTextAsync("outputs.json", JsonSerializer.Serialize(outputs));
if (OperatingSystem.IsLinux())
    File.SetUnixFileMode("outputs.json", UnixFileMode.UserRead | UnixFileMode.UserWrite);
Console.WriteLine($"Deployment ready; saved {outputs.Count} outputs.");
```

Run `dotnet run --no-restore` in the **client container**. Leave the application environment terminal running: it operates the native-engine supervisor and the certificate revocation service needed by database clients.

The result is `outputs.json` in the project directory. Subsequent chapters read it instead of constructing service endpoints by hand. Treat it as local configuration; do not publish telemetry connection strings unnecessarily. The database password is not an output.

Step 3: inspect the result

In QHQ, connect to this application environment as described in chapter 14. Open **Resource Manager → Deployments**, select your subscription and group, and refresh. Inspect the `book` deployment's state, outputs and per-resource operations. Use the individual service pages to inspect the deployed resources; ARM also exposes inventory across providers.

A deployment can fail after some resources have succeeded. Fix the reported resource setting or unavailable engine supervisor, then rerun the same incremental deployment. It updates resources without intentionally deleting those omitted from the template. It does not promise transactional rollback.

Step 4: redeploy and understand deletion

Run the same program again. The existing Blob/SQL/Redis data should remain. Deletion of a completed **deployment history** entry leaves its resources in place. Deletion of the **resource group** removes resources and their data, including the labelled native database volumes. Cleanup is a separate exercise in chapter 15.

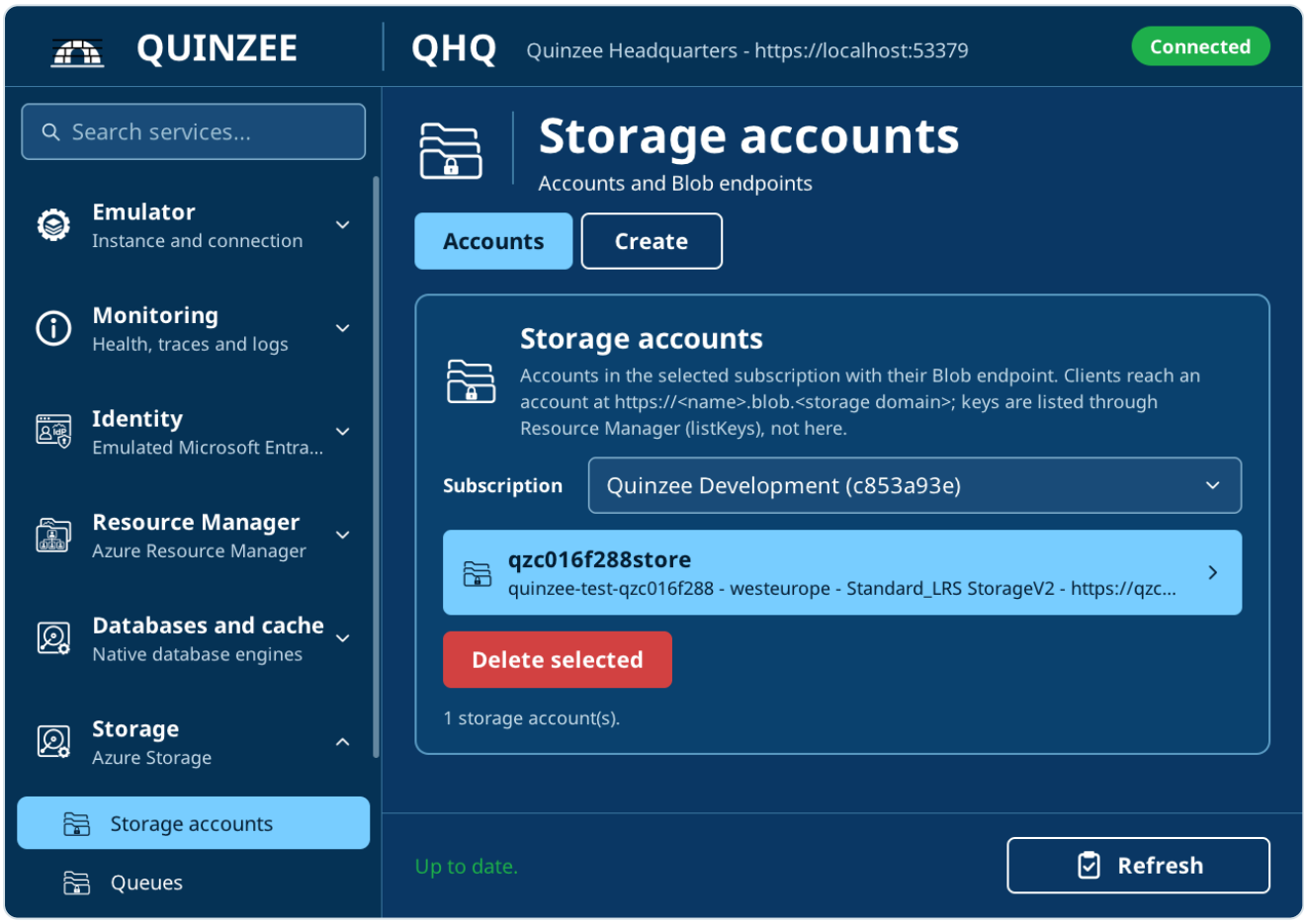
Supported deployments are resource-group, inline, incremental templates. Complete mode, linked/nested deployments, resource loops and arbitrary ARM expressions are not supported. Use the API versions in the supplied template; provider discovery reports the supported versions. An unsupported feature returns an error rather than pretending that it was applied.

Related reading

- Quinzee Administrator’s Guide: Control resources and infrastructure deployments
- Quinzee Internals: Deployments and native engines
- Scenario overview: Testing Bicep Deployments
- Scenario overview: Learning Infrastructure as Code with Bicep
- Scenario overview: Creating Reproducible Development Environments

6. Store documents in Blob Storage

A storage account contains containers; a container contains named blobs. In this example `documents` is private, and `hello.txt` is a block blob. The Azure Blob client obtains a storage-audience token from the credential automatically. Run the deployment chapter first so that `outputs.json` exists.



QHQ shows storage accounts and their Blob endpoints. This capture uses the browser demo's resource names; your book resources use the `bk…` *prefix.*

Step 1: upload, read and list

Replace **Program.cs** with this complete example, then run it in the client container with `dotnet run --no-restore`:

```
using Azure.Storage.Blobs;

var book = BookContext.Load();
var outputs = book.Outputs();
var service = new BlobServiceClient(new Uri(outputs["blobEndpoint"]), book.Credential);
var container = service.GetBlobContainerClient("documents");
await container.CreateIfNotExistsAsync();
var blob = container.GetBlobClient("hello.txt");

await blob.UploadAsync(BinaryData.FromString("Hello from Quinzee."), overwrite: true);
await blob.SetMetadataAsync(new Dictionary<string, string> { ["source"] = "book" });
var downloaded = await blob.DownloadContentAsync();
Console.WriteLine(downloaded.Value.Content.ToString());
await foreach (var item in container.GetBlobsAsync())
    Console.WriteLine($"{item.Name}: {item.Properties.ContentLength} bytes");
```

Expect `Hello from Quinzee.` and a listing containing `hello.txt`. Rerunning replaces that blob because `overwrite: true` is explicit. For a real file use `await blob.UploadAsync("/work/document.txt", overwrite: true)`; do not load a large file into a string merely to upload it.

Metadata belongs to the blob, separately from its content. Blob names can contain `/` to represent an application naming hierarchy. They are still blob names, not host filesystem paths.

Step 2: share read access with a scoped SAS

A service SAS grants a limited operation on a named blob until an expiry time. Constructing one requires an account key; a bearer credential alone cannot sign it. Replace `Program.cs` with this example after creating `hello.txt` above:

```
using Azure.Storage;
using Azure.Storage.Blobs;
using Azure.Storage.Sas;

var book = BookContext.Load();
var outputs = book.Outputs();
var account = outputs["storageName"];
using var keyResponse = await book.PostArmAsync(
    $"Microsoft.Storage/storageAccounts/{account}/listKeys", "2023-05-01");
var key = keyResponse.RootElement.GetProperty("keys")[0].GetProperty("value").GetString()!;
var signer = new StorageSharedKeyCredential(account, key);
var blob = new BlobServiceClient(new Uri(outputs["blobEndpoint"]), signer)
    .GetBlobContainerClient("documents").GetBlobClient("hello.txt");
var sas = new BlobSasBuilder
{
    BlobContainerName = "documents", BlobName = "hello.txt", Resource = "b",
```

```
    StartsOn = DateTimeOffset.UtcNow.AddMinutes(-1),
    ExpiresOn = DateTimeOffset.UtcNow.AddMinutes(5),
    Protocol = SasProtocol.Https
};
sas.SetPermissions(BlobSasPermissions.Read);
var delegated = new BlobClient(blob.GenerateSasUri(sas));
Console.WriteLine((await delegated.DownloadContentAsync()).Value.Content.ToString());
```

The delegated client sends no bearer token. Possession of its signed URL grants read access to `hello.txt`; it does not grant write access or access to another blob. Do not print or log that URL or the account key. Both account keys work; rotating the key used to sign a SAS invalidates signatures based on the old key.

Step 3: choose the right local limits

Quinzee supports block blobs, conditional operations with ETags, metadata, range reads and staged block uploads. Single PUT/block request bodies are limited to 256 MiB, and a committed block blob to 1 GiB. Large transfers stream through bounded buffers; these limits are not a global disk-space quota.

Use request conditions when several writers might update the same document. An `If-Match` failure is a concurrency conflict, not a reason to disable the condition and overwrite someone else's result blindly.

Page/append blobs, leases, snapshots, versions, user-delegation/account SAS, stored access policies and Queue SAS are outside this implementation. The supported service SAS permissions are `r`, `w`, `d` and `l`; do not assume that every option exposed by `BlobSasBuilder` is implemented.

Delete a tutorial blob with `await blob.DeleteIfExistsAsync()` when it is no longer needed. Account or resource-group deletion removes all of its Blob and Queue data.

Related reading

- Quinzee Administrator’s Guide: Administer storage, secrets and configuration
- Quinzee Internals: Module host and service state
- Quinzee Internals: Files, transactions and what a crash can leave behind

DEVELOPER'S GUIDE

7. Process work with Queue Storage

Queue messages tell a worker that there is something to do. Store the document in Blob Storage and put its name in the message instead of copying a large payload into the queue. The template created a `documents` queue; the sample below uses a separate `book-work` queue so it does not consume application work.

Step 1: send, peek, receive and acknowledge

Replace **Program.cs** and run `dotnet run --no-restore`:

```
using Azure.Storage.Queues;
using Azure.Storage.Queues.Models;
using System.Text.Json;

var book = BookContext.Load();
var outputs = book.Outputs();
var queue = new QueueClient(
    new Uri(new Uri(outputs["queueEndpoint"]), "book-work"), book.Credential,
    new QueueClientOptions { MessageEncoding = QueueMessageEncoding.Base64 });
await queue.CreateIfNotExistsAsync();
await queue.SendMessageAsync(JsonSerializer.Serialize(new { blob = "hello.txt" }));

var peek = await queue.PeekMessagesAsync(maxMessages: 1);
Console.WriteLine($"Messages visible to peek: {peek.Value.Length}");
var received = await queue.ReceiveMessagesAsync(
    maxMessages: 1, visibilityTimeout: TimeSpan.FromSeconds(30));
foreach (var message in received.Value)
{
    using var work = JsonDocument.Parse(message.Body.ToString());
    Console.WriteLine($"Process blob: {work.RootElement.GetProperty("blob").GetString()}");
    // Perform and durably record the application operation before acknowledging.
    await queue.DeleteMessageAsync(message.MessageId, message.PopReceipt);
}
```

Peek inspects a message without claiming it. Receive makes it invisible to other consumers for the visibility period and returns a **pop receipt**. Deleting with that receipt acknowledges successful processing.

Step 2: handle slow work and redelivery

If processing takes longer, renew the visibility period and retain the new receipt. Inside the receive loop, before deletion, the operation looks like:

```
var renewed = await queue.UpdateMessageAsync(
    message.MessageId, message.PopReceipt, message.Body.ToString(),
    TimeSpan.FromMinutes(1));
await queue.DeleteMessageAsync(message.MessageId, renewed.Value.PopReceipt);
```

Use this in place of the original delete, not after it. Updating or receiving a message rotates its receipt; an old receipt fails. If a worker exits before acknowledgement, the message becomes visible again after the lease expires. Queue state, visibility deadlines and dequeue counts survive cloud restart.

This is **at-least-once delivery**. Use an application work ID and an idempotent write, such as a database uniqueness constraint, to avoid processing the same business operation twice. There is no automatic exactly-once transaction across Queue, Blob, SQL and Redis. A repeatedly failing message needs an application retry/dead-letter policy; Quinzee does not move it automatically.

Step 3: understand capacity

The message text limit is 64 KiB **after Base64 encoding**; encoding reduces the raw payload you can fit. The default TTL is seven days. Receives accept 1–32 messages and a positive visibility period; visibility is bounded to seven days. The local capacity is 1,024 queues per account and 1,024 messages per queue, with a 16 MiB serialized queue-file limit. Capacity failures are explicit.

Use QHQ **Storage → Queues** to inspect queues and peek without consuming messages. Its approximate message count is not a guarantee that a receive will find work at that instant. The platform's sampled queue gauges are also not a complete inventory when their truncation indicator is set.

Related reading

- Quinzee Administrator’s Guide: Administer storage, secrets and configuration
- Quinzee Internals: Module host and service state
- Quinzee Internals: Files, transactions and what a crash can leave behind

DEVELOPER'S GUIDE

8. Keep application secrets in Key Vault

Key Vault separates secret retrieval from ordinary application settings. A secret has a name and one or more versions. Setting an existing name creates a new version; reading without a version selects the current one.

The local service supports **secrets**. Key Vault cryptographic keys and certificates are separate Azure products/APIs and are not emulated here.

Step 1: store and retrieve the database password

The deployment created the vault. Replace **Program.cs** and run it:

```
using Azure.Security.KeyVault.Secrets;

var book = BookContext.Load();
var outputs = book.Outputs();
var vault = new SecretClient(new Uri(outputs["vaultEndpoint"]), book.Credential);
var created = await vault.SetSecretAsync("database-password", book.Settings.DatabasePassword);
var current = await vault.GetSecretAsync("database-password");
if (current.Value.Value != book.Settings.DatabasePassword)
    throw new InvalidOperationException("Secret round trip failed");
Console.WriteLine($"Stored secret version: {created.Value.Properties.Version}");
await foreach (var properties in vault.GetPropertiesOfSecretsAsync())
    Console.WriteLine(properties.Name);
```

The sample validates the value without printing it. The SDK follows Key Vault's bearer challenge and obtains the vault-specific token. Do not replace its certificate checks or disable challenge-resource verification.

Reading a fixed version is useful when a configuration references a particular credential: `await vault.GetSecretAsync("database-password", version)`. Setting another version does not change an existing PostgreSQL or MySQL password; the local database service does not implement administrator credential rotation.

Step 2: try deletion and recovery on a separate secret

This exercise uses `disposable-example`, leaving the database password intact. Replace `Program.cs` with:

```
using Azure.Security.KeyVault.Secrets;

var book = BookContext.Load();
var vault = new SecretClient(new Uri(book.Outputs()["vaultEndpoint"]), book.Credential);
await vault.SetSecretAsync("disposable-example", "synthetic-example-value");
var deletion = await vault.StartDeleteSecretAsync("disposable-example");
await deletion.WaitForCompletionAsync();
var recovery = await vault.StartRecoverDeletedSecretAsync("disposable-example");
await recovery.WaitForCompletionAsync();
var recovered = await vault.GetSecretAsync("disposable-example");
Console.WriteLine($"Recovered: {recovered.Value.Name}");
```

A soft-deleted secret can be recovered. Purging it removes that recovery path. Deleting the vault or its resource group is a separate permanent deletion of the local vault and its data; there is no soft-deleted-vault recovery facility.

Step 3: design with the local boundary in mind

Secret values can be up to 25 KB, and the emulator keeps up to 500 versions per secret. Disabled versions are refused. Version attributes and tags are supported; secret `nbf`/`exp` metadata is stored rather than enforced as an access timer.

Local state is private to the operating user but is not an encrypted vault at rest. Use development credentials. RBAC/access-policy properties are not per-principal permission enforcement here. The current data plane retains the bootstrap/administrator access model and refuses new scoped identities until that service has a scoped policy integration. Test production permission assignments against Azure separately.

Related reading

- Quinzee Administrator’s Guide: Administer storage, secrets and configuration
- Quinzee Internals: Module host and service state
- Quinzee Internals: Files, transactions and what a crash can leave behind

9. Manage settings with App Configuration

App Configuration stores application settings by **key and label**. The label lets `Book:Queue` have different values for development and another environment. A null label is distinct from a label named `development`.

Keep nonsecret settings here, such as a queue name. Put passwords in Key Vault. A feature flag can be represented as a setting, but this emulator does not provide a separate server-side feature-management system.

Step 1: set, read and enumerate settings

Replace **Program.cs** with:

```
using Azure.Data.AppConfiguration;

var book = BookContext.Load();
var outputs = book.Outputs();
var configuration = new ConfigurationClient(
    new Uri(outputs["configurationEndpoint"]), book.Credential);
await configuration.SetConfigurationSettingAsync("Book:Queue", "book-work", "development");
await configuration.SetConfigurationSettingAsync("Book:Document", "hello.txt", "development");

var selected = await configuration.GetConfigurationSettingAsync("Book:Queue", "development");
Console.WriteLine($"Worker queue: {selected.Value.Value}");
var selector = new SettingSelector { KeyFilter = "Book:*", LabelFilter = "development" };
await foreach (var setting in configuration.GetConfigurationSettingsAsync(selector))
    Console.WriteLine($"{setting.Key} [{setting.Label}] = {setting.Value}");
```

Run `dotnet run --no-restore`. Expect `Worker queue: book-work` and the two `Book:` settings. Rerunning updates those same key/label pairs. Retrieving `Book:Queue` without a label would address the null-labelled setting instead.

Step 2: handle concurrent changes

Settings have ETags. Use conditional updates when you must preserve another writer's changes; an ETag mismatch returns 412. Read-only settings reject writes with 409 until unlocked. QHQ's configuration page can list, set and delete settings with an optional label.

The local store retains the last 1,000 revisions. Revision history is useful for inspection, but snapshots and point-in-time reads are not supported. The combined key and value size limit is 10 KB. Lists currently return their results in one page rather than implementing Azure's

continuation paging.

Step 3: choose an authentication method

The example uses a bearer credential. App Configuration also accepts its connection-string HMAC authentication; `listKeys` returns read-write and read-only keys. A read-only key cannot write. Setting `disableLocalAuth` on the store disables those key-based requests.

Do not put an access-key connection string in a browser bundle or print it in logs. This tutorial's .NET backend retrieves settings; the browser receives only the information the application deliberately exposes. Neither local labels nor a stored Azure RBAC property isolate tenants or applications.

Related reading

- Quinzee Administrator’s Guide: Administer storage, secrets and configuration
- Quinzee Internals: Module host and service state
- Quinzee Internals: Files, transactions and what a crash can leave behind

10. Use PostgreSQL, MySQL and Redis

Quinzee provisions real PostgreSQL 18, MySQL 8.4 and Valkey servers. ARM/Bicep manages their lifecycle; native clients execute SQL or Redis commands. Npgsql, MySqlConnector and StackExchange.Redis are independent .NET libraries, not Microsoft Azure management SDKs.

Complete chapters 5, 6 and 8 first. Keep the application environment terminal running: the supervisor reconciles engines and serves their signed certificate revocation list. The SDK container trusts the engine CA and can resolve the hostnames in `outputs.json`. Database ports are not published on the host.

Step 1: insert a document into PostgreSQL

Replace **Program.cs** and run it:

```
using Azure.Security.KeyVault.Secrets;
using Azure.Storage.Blobs;
using Npgsql;

var book = BookContext.Load();
var outputs = book.Outputs();
var vault = new SecretClient(new Uri(outputs["vaultEndpoint"]), book.Credential);
var password = (await vault.GetSecretAsync("database-password")).Value.Value;
var blob = new BlobClient(new Uri(new Uri(outputs["blobEndpoint"]), "documents/hello.txt"),
book.Credential);
var body = (await blob.DownloadContentAsync()).Value.Content.ToString();
var options = new NpgsqlConnectionStringBuilder
{
    Host = outputs["postgresHost"], Database = "application", Username = "appadmin",
    Password = password, SslMode = SslMode.VerifyFull,
    RootCertificate = "/run/book/engine-ca.crt", Timeout = 15, CommandTimeout = 15
};
await using var connection = new NpgsqlConnection(options.ConnectionString);
await connection.OpenAsync();
await using (var create = new NpgsqlCommand(
    "CREATE TABLE IF NOT EXISTS book_documents (id varchar(64) PRIMARY KEY, body text NOT NULL)",
connection))
    await create.ExecuteNonQueryAsync();
await using (var insert = new NpgsqlCommand(
    "INSERT INTO book_documents(id,body) VALUES (@id,@body) ON CONFLICT(id) DO UPDATE SET
body=EXCLUDED.body", connection))
{
    insert.Parameters.AddWithValue("id", "hello");
    insert.Parameters.AddWithValue("body", body);
    await insert.ExecuteNonQueryAsync();
}
await using var read = new NpgsqlCommand("SELECT body FROM book_documents WHERE id=@id",
```

```
connection);
read.Parameters.AddWithValue("id", "hello");
Console.WriteLine(await read.ExecuteScalarAsync());
```

Expect `Hello from Quinzee.`. The primary key and upsert make repeated delivery of this example work item safe. Values are SQL parameters; never concatenate a queue message or document into SQL text. `VerifyFull` validates both the certificate chain and hostname.

Step 2: do the same with MySQL

Replace **Program.cs** with:

```
using Azure.Security.KeyVault.Secrets;
using Azure.Storage.Blobs;
using MySqlConnector;

var book = BookContext.Load();
var outputs = book.Outputs();
var vault = new SecretClient(new Uri(outputs["vaultEndpoint"]), book.Credential);
var password = (await vault.GetSecretAsync("database-password")).Value.Value;
var blob = new BlobClient(new Uri(new Uri(outputs["blobEndpoint"]), "documents/hello.txt"),
book.Credential);
var body = (await blob.DownloadContentAsync()).Value.Content.ToString();
var options = new MySqlConnectionStringBuilder
{
    Server = outputs["mysqlHost"], Database = "application", UserID = "appadmin",
    Password = password, SslMode = MySqlSslMode.VerifyFull,
    SslCa = "/run/book/engine-ca.crt", ConnectionTimeout = 15, DefaultCommandTimeout = 15
};
await using var connection = new MySqlConnection(options.ConnectionString);
await connection.OpenAsync();
await using (var create = new MySqlCommand(
    "CREATE TABLE IF NOT EXISTS book_documents (id varchar(64) PRIMARY KEY, body text NOT NULL)",
connection))
    await create.ExecuteNonQueryAsync();
await using (var insert = new MySqlCommand(
    "INSERT INTO book_documents(id,body) VALUES (@id,@body) ON DUPLICATE KEY UPDATE body=@body",
connection))
{
    insert.Parameters.AddWithValue("id", "hello");
    insert.Parameters.AddWithValue("body", body);
    await insert.ExecuteNonQueryAsync();
}
await using var read = new MySqlCommand("SELECT body FROM book_documents WHERE id=@id",
connection);
read.Parameters.AddWithValue("id", "hello");
Console.WriteLine(await read.ExecuteScalarAsync());
```

MySqlConnector retains certificate revocation checks. A TLS failure after the supervisor has been stopped can therefore be a CRL availability problem, even if the database container itself is still running. Restart the supervisor; do not replace validation with an always-true callback.

Step 3: cache a result through the Redis protocol

Replace **Program.cs** with:

```
using StackExchange.Redis;

var book = BookContext.Load();
var outputs = book.Outputs();
using var keys = await book.PostArmAsync(
    $"Microsoft.Cache/redisEnterprise/{book.Settings.Prefix}-cache/databases/default/listKeys",
    "2025-04-01");
var options = new ConfigurationOptions
{
    Ssl = true, SslHost = outputs["cacheHost"],
    Password = keys.RootElement.GetProperty("primaryKey").GetString(),
    AbortOnConnectFail = true, ConnectTimeout = 10000,
    SyncTimeout = 10000, AsyncTimeout = 10000
};
options.EndPoints.Add(outputs["cacheHost"], 10000);
using var connection = await ConnectionMultiplexer.ConnectAsync(options);
var cache = connection.GetDatabase();
await cache.StringSetAsync("book:hello:status", "processed", TimeSpan.FromMinutes(5));
Console.WriteLine(await cache.StringGetAsync("book:hello:status"));
```

Expect `processed`. The access key stays in memory; the connection verifies the server name and uses the public CA bundle mounted in chapter 3. Treat a cache miss as a normal case and recover from your authoritative data store.

Step 4: manage lifecycle and data

Use QHQ **Databases and cache → PostgreSQL/MySQL** to inspect servers and children and request start, stop or restart. ARM supports the corresponding operations. A provisioning operation waits for real native-client readiness; if the supervisor cannot complete it within 180 seconds, it fails explicitly. Fix the underlying issue and resubmit rather than assuming that an ARM object means the database is already ready.

SQL stop and restart preserve data. Resource deletion removes the labelled data volume. A resource-group deletion is therefore destructive even if the engine is currently stopped. The application helper's Ctrl+C shutdown preserves native volumes for its next start; a standalone engine helper's Ctrl+C leaves its engine containers running, so manage those resources explicitly.

Each engine has local Docker limits of 1 GiB RAM, two CPUs and 256 processes. SKU names are metadata, not simulated Azure performance. No HA, replicas, backup/restore, engine upgrades, administrator credential rotation or managed- identity SQL login is implemented. The cache uses `databases/default`, encrypted connections and `NoCluster`; it is not a Redis cluster test environment.

Related reading

- Quinzee Administrator’s Guide: Run PostgreSQL, MySQL and Valkey safely
- Quinzee Internals: Deployments and native engines
- Quinzee Internals: From desired state to a real container
- Scenario overview: Testing Application Behaviour During Service Failures

11. Observe your application

There are two useful views of telemetry. **Application Insights and Log Analytics** are Azure-compatible resources used by your application and its Microsoft SDKs. **Platform diagnostics and Grafana** help you operate Quinzee itself. They have separate stores: sending an event to Application Insights does not automatically copy it into Grafana's Tempo/Loki backends.

Step 1: understand the resources

Chapter 5 created a Log Analytics workspace and a workspace-backed Application Insights component. `outputs.json` contains `workspaceId`, `insightsConnectionString` and `insightsResourceId`.

Use the workspace's **customer ID** for Logs Query, not its ARM resource ID. Use the returned component connection string so that ingestion targets Quinzee. The template sets `DisableLocalAuth: true`; the exporter therefore needs the same `TokenCredential` used elsewhere. It requests a monitoring-audience token.

Step 2: emit a request, log and metric

Replace **Program.cs** with this complete example. It uses the official `Azure.Monitor.OpenTelemetry.Exporter` library and `Azure.Monitor.Query.Logs`, already restored by chapter 3.

```
using Azure.Monitor.OpenTelemetry.Exporter;
using Azure.Monitor.Query.Logs;
using Microsoft.Extensions.Logging;
using OpenTelemetry;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;
using System.Diagnostics;
using System.Diagnostics.Metrics;

var book = BookContext.Load();
var outputs = book.Outputs();
Environment.SetEnvironmentVariable("APPLICATIONINSIGHTS_STATSBEAT_DISABLED", "true");
void Configure(AzureMonitorExporterOptions options)
{
    options.ConnectionString = outputs["insightsConnectionString"];
    options.Credential = book.Credential;
    options.DisableOfflineStorage = true;
    options.TracesPerSecond = null;
    options.SamplingRatio = 1;
```

```
    options.Retry.MaxRetries = 1;
    options.Retry.NetworkTimeout = TimeSpan.FromSeconds(5);
}
using var traces = Sdk.CreateTracerProviderBuilder()
    .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("quinzee.book"))
    .AddSource("Quinzee.Book")
    .AddAzureMonitorTraceExporter(Configure).Build();
using var metrics = Sdk.CreateMeterProviderBuilder()
    .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("quinzee.book"))
    .AddMeter("Quinzee.Book").AddAzureMonitorMetricExporter(Configure).Build();
using var source = new ActivitySource("Quinzee.Book");
using var meter = new Meter("Quinzee.Book");
string traceId;
using (var request = source.StartActivity("book.request", ActivityKind.Server)
    ?? throw new InvalidOperationException("Trace listener is missing"))
{
    traceId = request.TraceId.ToHexString();
    request.SetTag("http.request.method", "GET");
    request.SetTag("http.response.status_code", 200);
    request.SetStatus(ActivityStatusCode.Ok);
    var exporterOptions = new AzureMonitorExporterOptions();
    Configure(exporterOptions);
    using var logs = LoggerFactory.Create(builder =>
builder.SetMinimumLevel(LogLevel.Information)
        .AddOpenTelemetry(options =>
        {
            options.IncludeFormattedMessage = true;

options.SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("quinzee.book"));
            options.AddProcessor(new SimpleLogRecordExportProcessor(
                new AzureMonitorLogExporter(exporterOptions)));
        }));
    logs.CreateLogger("Book").LogInformation("Book request completed");
}
meter.CreateCounter<long>("book_documents_total").Add(1);
if (!traces.ForceFlush(10000) || !metrics.ForceFlush(10000))
    throw new InvalidOperationException("Telemetry flush timed out");
Console.WriteLine($"Trace ID: {traceId}");

var query = new LogsQueryClient(book.Credential);
async Task<long> WaitForCountAsync(string kql)
{
    using var timeout = new CancellationTokenSource(TimeSpan.FromSeconds(30));
    while (true)
    {
        var result = await query.QueryWorkspaceAsync(outputs["workspaceId"], kql,
            new LogsQueryTimeRange(TimeSpan.FromHours(1)),
            cancellationToken: timeout.Token);
        var count = result.Value.Table.Rows[0].GetInt64(0)
            ?? throw new InvalidOperationException("Query did not return a count");
        if (count > 0) return count;
        await Task.Delay(250, timeout.Token);
    }
}
foreach (var table in new[] { "AppRequests", "AppTraces" })
{
    var count = await WaitForCountAsync(
        $"{table} | where OperationId == '{traceId}' | count");
    Console.WriteLine($"{table}: {count}");
}
```

```
var metricCount = await WaitForCountAsync(
    "AppMetrics | where Name == 'book_documents_total' | count");
Console.WriteLine($"Metric records: {metricCount}");
```

Run `dotnet run --no-restore`. Expect a trace ID and a positive count for each signal. An `ActivityKind.Server` activity becomes an `AppRequests` record; client activities represent dependencies. The log shares the activity's operation ID. The explicit flush drains the providers; the query then waits up to 30 seconds for a visible record instead of assuming immediate visibility. A timeout is a failed verification: check ingestion, credentials and the workspace ID. Keep providers alive throughout the lifetime of a long-running application. The synchronous log processor here is for the small demonstration, not a high-throughput logging design.

Sampling must be 100 percent for this local ingestion subset. Set `TracesPerSecond = null` as well as `SamplingRatio = 1`: the exporter's default rate limit otherwise takes precedence over the ratio. Statsbeat is disabled for the local example, and failed exports are not saved to an offline cache. Avoid putting passwords, signed URLs, SQL parameters or document content in tags or log messages. Arbitrary application messages are not automatically sanitized just because the destination is a local monitoring workspace.

The exporter batches supported application envelopes and sends them to the component's ingestion endpoint. The query client reads workspace rows through a separate API and audience. Treat a successful export and a successful query as separate checks, and use only the query stages demonstrated below.

Step 3: query useful records

In QHQ **Monitoring → Observability → Workspace logs**, enter the customer ID from `workspaceId`, then run one query at a time, for example:

```
AppRequests | order by TimeGenerated desc | take 50
AppRequests | where Success == false | project Name, ResultCode, DurationMs
AppTraces | where TimeGenerated >= ago(1h) | project TimeGenerated, Message
AppDependencies | where OperationId == 'your-32-character-trace-id' | take 100
AppMetrics | where Name == 'book_documents_total' | count
```

The tables are `AppRequests`, `AppDependencies`, `AppTraces`, `AppExceptions`, `AppEvents` and `AppMetrics`. Supported stages are `where` comparisons, `project`, one-column `order by`, `take`/`limit`, `count` and `summarize count()`. Unsupported joins, render operations and other KQL return errors. Queries return at most 1,000 rows; narrow or aggregate a larger result. QHQ displays up to 100 of the returned rows and tells you that limit.

ARM Metrics also exposes request/dependency count, failed and duration metrics on the component. Use `requests/count`, `requests/failed`, `requests/duration` and the corresponding `dependencies/…` names, with `Count`, `Total`, `Average`, `Minimum` or `Maximum` as appropriate. Duration

is milliseconds. Custom `AppMetrics` names are queried through Logs Query rather than that ARM endpoint.

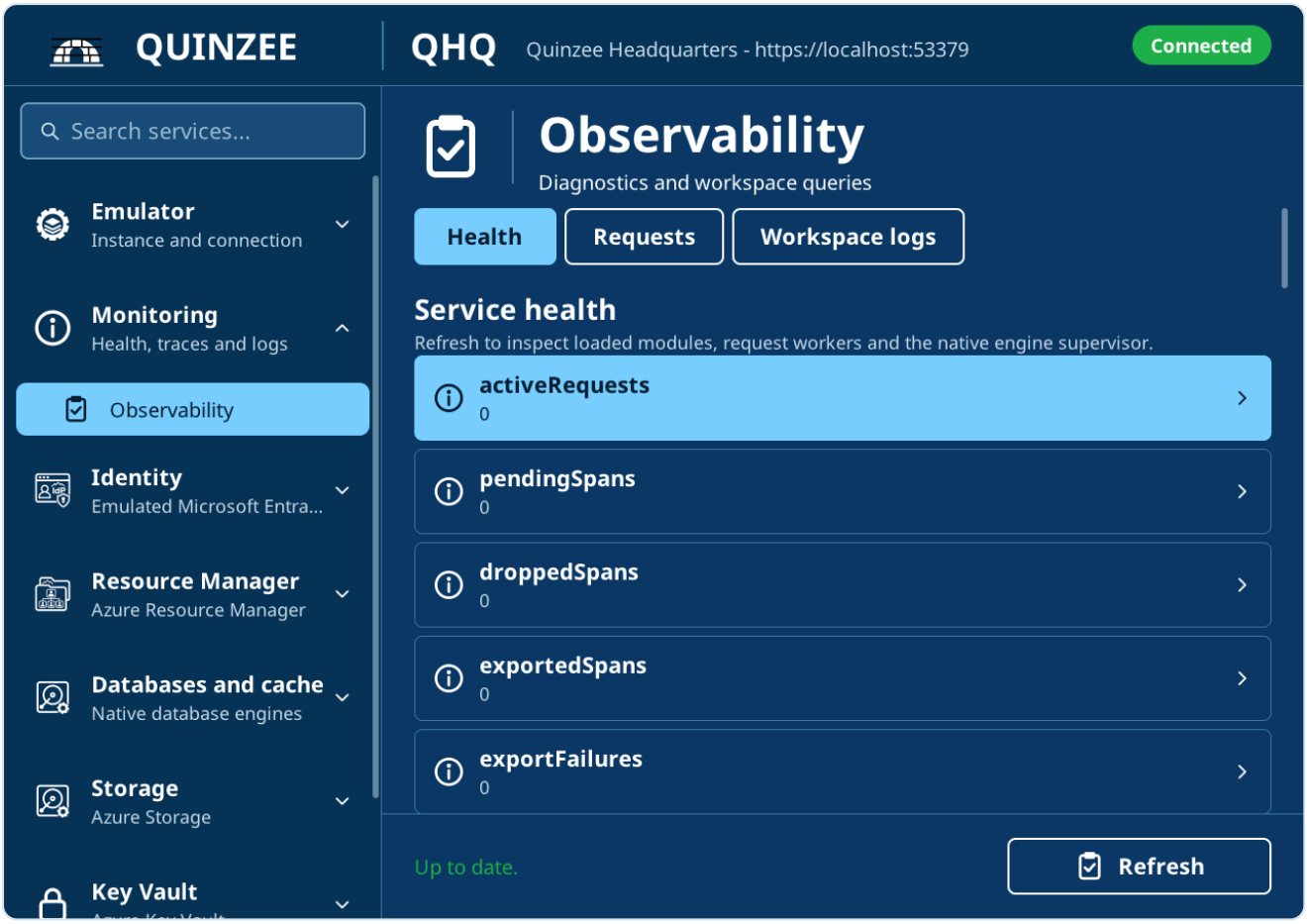
Step 4: diagnose the local platform

For the environment started with `make application-observability-up`:

1. Sign into **https://localhost:13001**, user `admin`, using `.local/observability/grafana-password` and its public CA.
2. Open **Quinzee observability** for request rates, handler latency/errors, transport throughput, worker activity, sampled queue depth and engine health.
3. In Explore, select Tempo and search a trace ID from the browser workflow. Select Loki for correlated cloud, application and engine logs.
4. In QHQ's Health and Requests tabs, refresh diagnostics or filter recent requests by trace ID. This uses authenticated administration APIs.

The browser workflow already exports to this platform stack. The standalone example above exports only to its Application Insights component. To send your own application's traces to Tempo as well, configure an OTLP exporter and its trusted HTTPS collector endpoint separately. The host endpoint is normally `https://localhost:14318`; in another container, `localhost` means that container, not Quinzee. Join the collector's network and configure its DNS/name and CA before using its internal endpoint. Do not substitute an Application Insights connection string for an OTLP URL.

`/health` reports process liveness. `/ready` reports that identity and Resource Manager are loaded; a stopped SQL server does not make the whole control plane unready. Engine heartbeat/state and worker/queue diagnostics provide the additional evidence. Handler latency excludes transfer time; transport metrics include parsed HTTP exchange/body transfer time. A queue sampling truncation flag means the displayed depth is partial.



QHQ separates platform health, recent requests and workspace log queries. The capture shows a local environment with external OTLP export disabled.

Step 5: understand retention

Workspaces retain 30–730 days by configuration, but also have a hard cap of 4,096 accepted rows per workspace, so a busy local workspace can evict older rows much sooner. Batch ingestion is capped at 1 MiB/256 envelopes, with 8 KiB normalized rows. Queries and ingested state survive cloud restart.

The optional tools keep logs/traces for 24 hours; Prometheus also targets 512 MB retention. Backend WALs and compaction need additional disk space. Stopping the tools preserves their data in `.local/observability/data`.

Alerts, diagnostic settings, workbooks, profiling, live metrics, DCR/DCE collection, gzip ingestion, sampled envelopes and full KQL are not supported. A local workspace and Grafana dashboard are separate facilities, not a complete Azure Monitor service replica.

Related reading

- Quinzee Administrator’s Guide: Observe the platform and the application separately
- Quinzee Administrator’s Guide: Diagnose failures from the outside inward
- Quinzee Internals: HTTP library and asynchronous execution

- Quinzee Internals: Connection and streaming lifecycle
- Scenario overview: Debugging Cloud Applications Locally
- Scenario overview: AI-Assisted Cloud Debugging

12. Deliver work with Service Bus

Use a Service Bus queue when workers must explicitly acknowledge work. Use a topic when each subscriber needs its own copy of an event. Service Bus is a separate service from Queue Storage: its .NET client is `Azure.Messaging.ServiceBus`, and its native transport is AMQP over TLS.

This chapter and the next use separate example projects and a standalone cloud. They do not replace the earlier tutorial's `Program.cs` or use its credentials. The application environment from chapter 2 does not enable the AMQP listener.

Step 1: run the complete SDK example

On the **host**, install the native build prerequisites, .NET 10 SDK, Python 3 and OpenSSL, then run:

```
make service-bus-sdk-test
```

The command builds Quinzee, compiles the Service Bus Bicep example, restores the locked SDK project and starts a temporary cloud with private certificates. It deploys an `orders` queue and an `events` topic with `first` and `second` subscriptions. The client sends, peeks, receives, renews a lock, abandons, completes, dead-letters and fans out messages. It also checks bearer authentication and rejection of invalid SAS and unsupported batch sends. Success ends with `Service Bus SDK smoke passed`; the fixture removes its temporary state. No Azure account or host trust/DNS change is needed.

The command supplies a temporary connection, namespace and trusted root to its client. The persistent lab below explains how to prepare those same prerequisites for your own program; a client cannot connect using the namespace name alone.

Step 2: understand delivery and settlement

This excerpt assumes an already authenticated `ServiceBusClient` named `client`, as created by the complete example:

```
await using var sender = client.CreateSender("orders");
await using var receiver = client.CreateReceiver("orders");
await sender.SendMessageAsync(new ServiceBusMessage("invoice-42")
{
    MessageId = "invoice-42", ContentType = "text/plain"
});
var message = await receiver.ReceiveMessageAsync(TimeSpan.FromSeconds(5));
```

```
if (message is null) throw new InvalidOperationException("No work available");
Console.WriteLine(message.Body.ToString());
await receiver.CompleteMessageAsync(message);
```

The default receive mode is PeekLock. Receipt reserves the message; successful completion removes it. In a real worker, complete only after committing the application's work. An abandoned or expired lock makes the message available again, so handlers must tolerate redelivery. A `MessageId` does not enable duplicate detection. Renew a lock before it expires if processing needs longer.

Operation	Effect
Peek	Inspect a message without reserving it
Receive in PeekLock mode	Obtain an exclusive, expiring lock
Complete	Remove the message using its current lock
Abandon	Release the lock for another delivery
Dead-letter	Move failed work to the entity's `$DeadLetterQueue`
ReceiveAndDelete	Remove immediately, before processing can succeed

For topics, `CreateSender("events")` publishes one message, and `CreateReceiver("events", "first")` receives the first subscription's copy. The second subscription has an independent copy and lock. Create subscriptions before publishing: a topic does not keep a copy for a future subscription.

Step 3: provision a persistent messaging lab

Use the following **host** commands when you want resources to remain available in QHQ after the isolated example finishes:

```
make certs
make cloud-up
make bicep-examples
install -d -m 700 .local/messaging
make qhq-run
```

These commands use `.local/quinzee`, the default HTTPS endpoint `https://localhost:11443`, and a loopback AMQP port of 5671. `QUINZEE_AMQP_PORT` changes the host AMQP port. Keep clients on the private Docker network when using service DNS names; native namespace names end in `.servicebus.quinzee.cloud`. Azure names require explicitly enabled emulation and its matching client trust root.

In QHQ, select the standalone subscription and create resource group `quinzee-messaging` in `westeurope`. Under **Resource Manager → Deployments**, submit `.packages/tools/service_bus.json`. Save this parameter document as `.local/messaging/service-bus.parameters.json` and select it in the deployment:

```
{
  "parameters": {
    "namespaceName": { "value": "bookbus" }
  }
}
```

Use a different namespace if `bookbus` already belongs to another exercise. After the deployment succeeds, open **Service Bus → Namespaces**. Its Namespaces, Queues, Topics and Subscriptions controls inspect the resources; they do not reveal keys or consume messages.

Step 4: obtain connection details privately

Save the following as `.local/messaging/client.py`. Run it from the repository root with `python3 .local/messaging/client.py`. It signs in to the standalone cloud, obtains the primary namespace connection string through ARM, and writes it to a private file for the next chapter. It uses the first local subscription; adjust the group or namespace if you selected different names in QHQ.

```
import http.client
import json
import os
from pathlib import Path
import ssl
import urllib.parse

ROOT = Path(__file__).resolve().parents[2]
context = ssl.create_default_context(cafile=str(
    ROOT / ".local/certificates/ca/quinzee-dev-root-ca.crt"))
port = int(os.environ.get("QUINZEE_HTTPS_PORT", "11443"))
token = ""

def request(method, path, body=None, *, host=None, headers=None):
    fields = dict(headers or {})
    if token and not host:
        fields["Authorization"] = "Bearer " + token
    if host:
        fields["Host"] = host
    if isinstance(body, dict):
        body = json.dumps(body)
        fields["Content-Type"] = "application/json"
    connection = http.client.HTTPSConnection(
        "localhost", port, context=context, timeout=15)
    try:
        connection.request(method, path, body, fields)
        response = connection.getresponse()
        raw = response.read()
```

```
        if not 200 <= response.status < 300:
            raise RuntimeError(f"Request failed: HTTP {response.status}")
        return json.loads(raw) if raw else None
    finally:
        connection.close()

def access_token(scope):
    credentials = json.loads(
        (ROOT / ".local/quinzee/credentials.json").read_text())
    form = dict(grant_type="client_credentials",
                client_id=credentials["clientId"],
                client_secret=credentials["clientSecret"], scope=scope)
    return request("POST",
        f"/{credentials['tenantId']}/oauth2/v2.0/token",
        urllib.parse.urlencode(form),
        host="localhost",
        headers={"Content-Type": "application/x-www-form-urlencoded"}
    )["access_token"]

token = access_token("https://management.azure.com/.default")
subscription = request("GET", "/subscriptions?api-version=2023-05-01")
sub = subscription["value"][0]["subscriptionId"]
group = f"/subscriptions/{sub}/resourceGroups/quinzee-messaging"

if __name__ == "__main__":
    keys = request("POST", group +
        "/providers/Microsoft.ServiceBus/namespaces/bookbus" +
        "/authorizationRules/RootManageSharedAccessKey/listKeys" +
        "?api-version=2024-01-01", {})
    os.umask(0o077)
    destination = ROOT / ".local/messaging/connection.txt"
    with destination.open("x") as output:
        output.write(keys["primaryConnectionString"])
    print("Saved private Service Bus connection file")
```

The helper deliberately refuses to overwrite an existing connection file. It verifies the local HTTPS endpoint's CA and hostname; the `Host` header selects the local service without changing system DNS. Keep this directory private and out of source control. The connection string grants broad namespace access; never include it in screenshots, logs or command-line arguments.

For your own .NET client, read that file into the `ServiceBusClient` constructor, select `AmqpTcp`, and provide the environment's public CA. On a forwarded port, set `CustomEndpointAddress` to the local TLS endpoint while preserving the namespace in the connection string. The complete SDK example demonstrates private CA trust with hostname checking retained. An HTTPS health response alone does not prove that AMQP is listening or that a message was settled.

For capacity planning, this local broker retains at most 10,000 message copies in a 256 MiB store. Topic subscriptions each consume a copy. Use small synthetic messages while learning, settle completed work, and inspect dead letters when receives stop succeeding. Unsupported batch sends, sessions, transactions, scheduling and WebSockets return errors; use the single-message TLS workflow shown in this chapter.

Related reading

- Quinzee Administrator’s Guide: Diagnose failures from the outside inward
- Quinzee Administrator’s Guide: Keep events and messages moving
- Quinzee Internals: Testing and verification
- Quinzee Internals: The compatibility application
- Scenario overview: Integration Testing Across Multiple Azure Services
- Scenario overview: Running Azure Integration Tests in CI/CD
- Scenario overview: Testing Application Upgrades and Regressions

13. Run HTTP and Service Bus functions

Quinzee runs your .NET isolated functions in a real Azure Functions v4 host. ARM/Bicep creates the plan and function app; a separate Docker helper starts its registered application image. HTTP requests enter through Quinzee's HTTPS gateway. Service Bus triggers connect from the function host to the broker.

Step 1: try the complete application

On the **host**, use the native build prerequisites, .NET 10 SDK, Python 3, OpenSSL and Docker. Build the fixture's local runner image, then run the example:

```
docker build --target lint --tag triglav-http-lint:local .
make functions-test
```

This builds the locked .NET project and application image, creates a temporary cloud/network/helper, deploys compiled Bicep, calls an HTTP function with and without its key, and sends work to the Service Bus trigger. It also exercises settings changes, restart, stop/start, helper recovery and deletion. Test-owned containers and the network are removed afterwards; this does not create a persistent QHQ environment. The initial image/package download needs network access. The following steps show the persistent deployment and invocation sequence so you can reproduce it without depending on the test's temporary state.

Step 2: inspect the function code

Start from the locked project in `tests/compatibility/functions`. Its packages provide the isolated worker, HTTP trigger and Service Bus extension. This is the complete Program.cs; use this separate Functions project, not the earlier console application's Program.cs:

```
// Real isolated-worker HTTP and Service Bus functions used by the emulator test.
// Quinzee Cloud - proprietary and confidential. All rights reserved.
using System.Collections.Concurrent;
using System.Net;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Extensions.Hosting;

await new HostBuilder().ConfigureFunctionsWorkerDefaults().Build().RunAsync();
```

```
public static class Functions
{
    private static readonly ConcurrentQueue<string> Received = new();

    [Function("Echo")]
    public static async Task<HttpResponseData> Echo(
        [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = "echo")] HttpRequestData
request)
    {
        var response = request.CreateResponse(HttpStatusCode.OK);
        await response.WriteAsJsonAsync(new { message = "hello from a real function", setting =
Environment.GetEnvironmentVariable("Greeting") });
        return response;
    }

    [Function("Messages")]
    public static async Task<HttpResponseData> Messages(
        [HttpTrigger(AuthorizationLevel.Function, "get", Route = "messages")] HttpRequestData
request)
    {
        var response = request.CreateResponse(HttpStatusCode.OK);
        await response.WriteAsJsonAsync(Received.ToArray());
        return response;
    }

    [Function("Consume")]
    public static void Consume([ServiceBusTrigger("orders", Connection = "ServiceBusConnection")]
string body)
    {
        if (Received.Count < 100) Received.Enqueue(body);
    }
}
```

HostBuilder starts the worker. Each Function attribute declares an entry point. The HTTP trigger requires a Function key. The Service Bus trigger names `orders` and reads its connection settings through the `ServiceBusConnection` prefix. The in-memory queue exists only to make trigger delivery visible in the tutorial; a production handler would write durable application state before completing.

Step 3: register an image for the persistent lab

Continue from the standalone `quinzee-messaging` group, `bookbus` namespace, `orders` queue and private connection file in chapter 12. Keep that cloud running. On the **host**:

```
make functions-example
python3 infrastructure/functions/register.py \
  --image quinzee-functions-example:local \
  --registry .local/functions-helper/images.json
make functions-run
```

Leave the last command running in its terminal. The helper uses the default standalone network/DNS, `.local/quinzee` cloud state and its own `.local/functions-helper` private state. It needs Docker access. Neither the cloud nor the function container receives a Docker socket

or a CA signing key. The default trust root supports native service names; an Azure-named Service Bus connection additionally needs the emulation CA through the helper's `--trust-ca` option, so the hosted client verifies the Azure-name certificate as well as the native one. The helper accepts repeatable public CA files; it does not need the CA key.

Registration resolves the local tag to an immutable image ID and approves that code for execution. After rebuilding an image, register it again, restart the helper to read the registry, and restart the app. The helper never pulls an arbitrary image named by a deployment. To use your own app, publish its .NET 10 isolated project and build that output with `infrastructure/functions/Dockerfile`; then register the resulting local image.

Step 4: deploy the function app

In another **host** terminal, create a private ARM parameters file without printing the Service Bus credential:

```
python3 - <<'PY'
import json
import os
from pathlib import Path
os.umask(0o077)
directory = Path(".local/messaging")
parameters = {"parameters": {
    "appName": {"value": "book-functions"},
    "serviceBusConnection": {
        "value": (directory / "connection.txt").read_text().strip()}
}}
with (directory / "functions.parameters.json").open("x") as output:
    json.dump(parameters, output)
PY
make bicep-examples
```

In QHQ's deployment page, select `quinzee-messaging` and deploy `.packages/tools/functions.json` with `.local/messaging/functions.parameters.json`. It creates a Linux B1 plan and `book-functions` app, with `Greeting` set to `from Bicep`. Secure parameters remain sensitive in local files even though the template declares them secure.

Open **Functions → Function apps** and Refresh until the app is Running and provisioning succeeds. Updating means the host is not ready yet; keep the helper running. An unregistered image fails, and a missing helper times out after 180 seconds. A successful ARM lifecycle POST acknowledges the requested state; poll the resource to see whether the runtime has reached it.

Step 5: invoke HTTP and queue triggers

Save this as `.local/messaging/invoke.py` beside `client.py`, then run `python3 .local/messaging/invoke.py` from the **host**. The helper from chapter 12 keeps ARM credentials and function keys out of terminal output. This example uses default names and

ports; adjust them together if you changed the lab.

```
import time
from client import access_token, group, request

app = group + "/providers/Microsoft.Web/sites/book-functions"
keys = request("POST",
    app + "/host/default/listkeys?api-version=2024-11-01", {})
headers = {"x-functions-key": keys["functionKeys"]["default"]}
host = "book-functions.functions.quinzee.cloud"
print(request("GET", "/api/echo", host=host, headers=headers))

bearer = access_token("https://servicebus.azure.net/.default")
request("POST", "/orders/messages", "invoice-42",
        host="bookbus.servicebus.quinzee.cloud",
        headers={"Authorization": "Bearer " + bearer,
                 "Content-Type": "text/plain"})
for _ in range(30):
    messages = request("GET", "/api/messages", host=host, headers=headers)
    if "invoice-42" in messages:
        print("Service Bus function received invoice-42")
        break
    time.sleep(1)
else:
    raise RuntimeError("Service Bus trigger did not receive the message")
```

The first response contains `hello from a real function` and `from Bicep`. The next success line confirms that the queue trigger ran the application's code. Do not run a competing receiver on `orders` during this check: either consumer can receive a queued message. Use a fresh message value for independent repeat checks, because the sample retains earlier deliveries until restart.

The real host enforces `AuthorizationLevel.Function`; an ARM bearer token is not the invocation key. Only `/api/...` routes are forwarded, and host administration paths are blocked. The gateway permits 64 KiB request/response bodies and a 10-second exchange; streaming and WebSockets are unsupported.

Use managed identity instead of a connection string

Run `make functions-identity-test` to verify the same Function with a user-assigned identity, Receiver role and SAS disabled. The complete Bicep is `examples/provisioning/functions_identity.bicep`. See the identity walkthrough for binding settings and system-assigned identity. The helper owns the token endpoint/header; application settings never contain its credential.

Step 6: stop or remove the lab

Use QHQ's Start, Stop and Restart actions and Refresh to check completion. To pause, Stop the app while the helper is running and wait for Stopped before ending the helper. Stopping only the helper leaves containers and Service Bus triggers running; gateway forwarding

stops when the heartbeat becomes stale.

To remove the exercise, deliberately delete `quinzee-messaging` through QHQ while the helper is still running, and allow it to remove the owned runtime. This also deletes the lab's queue/topic messages. Deleted app names remain reserved by tombstones; choose a fresh app name for a new exercise. Private helper key files and local parameter files remain for explicit operator cleanup. Then close QHQ and use `make qhq-env-down` for its standalone environment.

This is a single-instance Linux .NET development runtime. Consumption/Flex scaling, slots, ZIP/Kudu deployment, other language workers, timer/Blob/Queue Storage triggers and Durable Functions are outside the verified subset. Use a separate compatibility exercise before adopting another binding. A package that compiles is not evidence that its dependent service exists locally.

Related reading

- Quinzee Administrator’s Guide: Operate Functions and App Service workloads
- Quinzee Administrator’s Guide: Keep events and messages moving
- Quinzee Internals: Service Bus and Functions execution
- Quinzee Internals: From desired state to a real container
- Scenario overview: Developing an Azure Functions Application Locally
- Scenario overview: Experimenting with Cloud Architecture
- Scenario overview: Using Quinzee as an Internal Server for a Small Office

14. Manage the environment with QHQ

QHQ—Quinzee Headquarters—is the native management application. It uses the same public HTTPS APIs as your C# clients. A graphical Linux session and SDL2 are needed for its Rasta development window.

Step 1: connect to the book's application environment

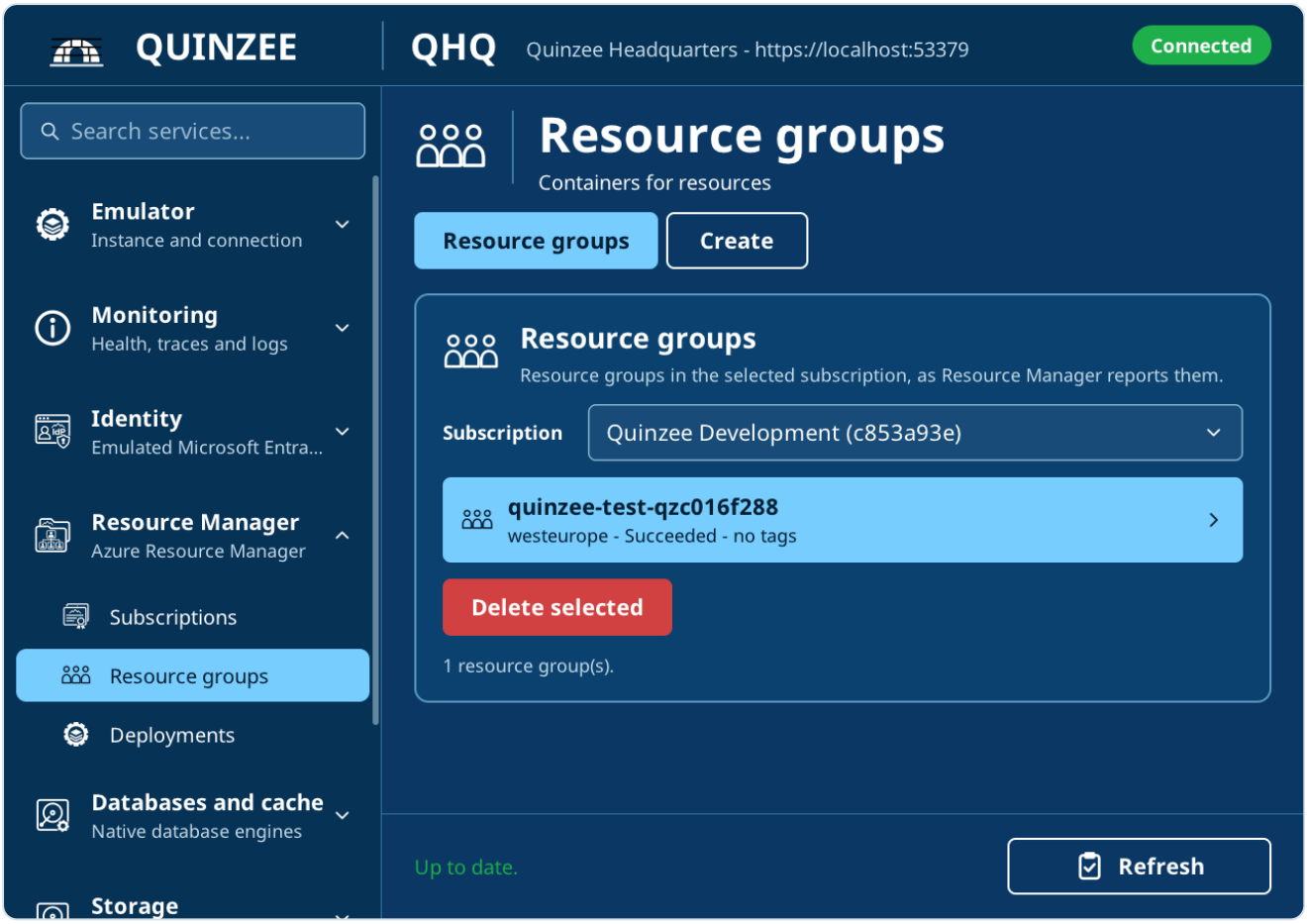
The tutorial environment has its own cloud port, data directory and CA. `make qhq-run` normally starts a separate default cloud environment, so use the following commands to connect to the one already running for the book.

On the **host**, from the repository root:

```
make qhq qhq-rasta-start
book_env="$PWD/.local/application/development"
book_port=$(python3 -c 'import json,sys; print(json.load(open(sys.argv[1]))["cloudPort"])'
"$book_env/environment.json")
QUINZEE_HTTPS_PORT="$book_port" \
QUINZEE_DATA_DIR="$book_env/data" \
QUINZEE_CERT_DIR="$book_env/certificates" \
  bash scripts/dev/qhq.sh run
```

When you close QHQ, stop its viewer with `bash scripts/dev/qhq.sh rasta-stop`. The application environment stays running in its original terminal. If you started QHQ against the wrong instance, use **Emulator → Connection** to set the endpoint, credentials file and CA above, then connect. The credentials file is `data/credentials.json`, not the browser access-token file or the tutorial's filtered settings file.

For a separate standalone management environment, `make qhq-run` remains the convenient command; finish that session with `make qhq-env-down`.



Choose a subscription, then inspect its resource groups. The endpoint port and resource names come from this disposable example environment; use your own. This 15 November capture predates the new Service Bus and Functions pages.

Step 2: navigate the service pages

Categories start collapsed. Expand a category, then choose its page. Pages may have tabs. Tab/Shift+Tab move between controls; Enter activates buttons. Use Refresh after an external SDK changes resources, and check the selected subscription/group before making changes.

Page	Use it to
Emulator → Overview	Check connection, endpoints, uptime and loaded services
Emulator → Connection	Select the cloud URL, credential file and trusted CA
Emulator → Modules	Inspect module state/dependencies and load or unload services
Emulator → DNS	Inspect service/custom records and enable or disable Azure-name emulation
Azure / Entra ID → Tenant	Inspect local tenant/authority details and obtain an access token

Page	Use it to
Azure / Entra ID → App registrations / Enterprise applications	Create and edit registrations, credentials and service principals
Azure / Entra ID → Users / Groups	Manage shared accounts, passwords and memberships
Azure / Entra ID → App roles / App role assignments	Define API roles and grant them to users, groups and applications
Azure / Resource access → Managed identities / RBAC	Create identities and assign scoped resource permissions
Windows / Directory → LDAP	Inspect the shared backend/listener and open shared user/group management
Azure / App Service → Web apps / App Service plans	Provision Linux hosting, attach identities, manage settings and lifecycle
Resource Manager → Subscriptions	Inspect or manage local subscriptions
Resource Manager → Resource groups	Create/list/delete groups with location and tags
Resource Manager → Deployments	Submit compiled ARM JSON, inspect history and resource operations
Storage → Storage accounts	Manage accounts and inspect endpoints
Storage → Queues	Manage queues, send messages and peek without consuming work
Storage → Tables	Inspect tables and work with typed entities
Event Grid → Topics	Inspect topics and event subscriptions
Service Bus → Namespaces	Inspect namespaces, queues, topics and subscriptions
Functions → Function apps	Create/edit apps, attach identities, manage settings and Start/Stop/Restart
Key Vault → Vaults	Manage vaults and secret values/metadata through the Secrets tab
App Configuration → Configuration stores	Manage stores and labelled key-values
Databases and cache → PostgreSQL/MySQL	Manage servers/database children and start/stop/restart SQL servers
Databases and cache → Managed Redis	Manage the local encrypted cache and its database
Monitoring → Observability	Inspect health, recent requests and workspace query results

Workspace and Application Insights component provisioning is through ARM/Bicep; there is no separate component-creation wizard in QHQ. Use the monitoring page to query their data after deployment. Service Bus provisioning uses the deployment page. Functions and App Service also have creation/editing forms. Their public inventory omits secret values; the App settings tab changes one private setting at a time. See the identity management walkthrough and web-app hosting guide.

Step 3: inspect your deployment

Choose the tutorial's `quinzee-book-bk…` group on the deployment page. Refresh, select `book`, and inspect its operations. To submit a new deployment in QHQ, provide a compiled JSON template and, where needed, an ARM parameters JSON file. A `.bicep` file is source code: compile it first. Keep secure parameter files private and do not paste passwords into tags or resource names.

A server can exist while its operation is still running. Refresh until the operation succeeds before using its native protocol. The engine helper must be running; QHQ does not have Docker-administrator powers of its own.

Step 4: use modules deliberately

An unloaded module retains its state, but requests for its service return 503. Reload it to resume. Module dependencies prevent removing a provider that is still required by another loaded module; identity cannot be unloaded. Unloading SQL modules is not the same as stopping their native database containers. Unloading Functions also leaves its external hosts and queue triggers running. Use Stop while the runtime helper is available and wait for the state change.

A resource Delete action is different: it removes the resource and may remove stored data. Use SQL Stop for a temporary pause, and reserve resource-group Delete for deliberate cleanup.

Related reading

- Quinzee Administrator’s Guide: Connect QHQ and make changes deliberately
- Quinzee Administrator’s Guide: Find every control in QHQ
- Quinzee Internals: QHQ and libsimple
- Quinzee Internals: Inspect a disposable instance from Linux
- Scenario overview: Practising Azure Administration

15. Stop, restart and clean up safely

There are two useful environment entry points. Do not mix their state paths, credentials or Compose commands accidentally.

Environment	Start	State	Stop
Application tutorial	`make application-up` or `make application-observability-up`	`.local/application/development` plus owned engine volumes	Exit attached development containers, then Ctrl+C in the helper terminal
Standalone cloud + DNS	`make cloud-up`	`.local/quinzee`, certificates in `.local/certificates`	`make cloud-down`
Functions helper for standalone cloud	`make functions-run`	`.local/functions-helper`, plus cloud mailbox state	Stop apps through ARM/QHQ and wait, then Ctrl+C in the helper terminal
Optional telemetry tools	`make observability-up`	`.local/observability`	`make observability-down`

The application helper generates its own Compose project, subnet and cloud port. Its browser port is normally 3000. The standalone cloud uses `https://localhost:11443`, AMQP/TLS on loopback port 5671, DNS diagnostics on loopback port 10053 and the `triglav-http_quinzee` network by default. Native engine ports are accessible inside their private network, not published on the host.

Step 1: stop without deleting data

1. Finish or cancel your client work. Cancellation can leave already-created resources; it does not roll back a deployment.
2. Exit the SDK container with `exit`, so it no longer holds the private network open.
3. Close QHQ and stop its Rasta viewer if you started it separately.
4. Press Ctrl+C in the application helper terminal. It stops the environment and preserves its state and engine volumes.
5. Run `make observability-down` if you also started the external tools.

Run the same application command later to reuse the environment. Existing accepted monitoring rows, blobs, queue leases, secrets, settings and native volumes survive this stop/start cycle. A deployment interrupted while running is marked failed on restart; redeploy it after checking what completed. Create fresh credential/client instances after a cloud restart if old cached bearer tokens are rejected.

For the standalone setup, `make cloud-down` stops the cloud/DNS services but does not manage separately operated engine containers. Stop SQL resources through QHQ/ARM or delete the resources deliberately. Simply stopping the standalone supervisor leaves engines running. Likewise, stop function apps through QHQ/ARM while the Functions helper is running and wait for Stopped before ending that helper. Otherwise Service Bus triggers can keep consuming messages even after HTTP forwarding has stopped. Service Bus queues, topic copies and locks persist across a normal cloud restart.

Step 2: delete only the tutorial resources

This step destroys the book's resources and their data, including SQL/Redis volumes. Run it only when you have finished the service exercises. Keep the application helper and SDK container running until deletion completes. Replace **Program.cs** with:

```
using Azure;
using Azure.Core;

var book = BookContext.Load();
if (!book.Settings.Group.StartsWith("quinzee-book-", StringComparison.Ordinal))
    throw new InvalidOperationException("This example only deletes its tutorial group");
Console.WriteLine($"Deleting tutorial group: {book.Settings.Group}");
var group = book.Arm.GetResourceGroupResource(new ResourceIdentifier(book.GroupId));
await group.DeleteAsync(WaitUntil.Completed);
Console.WriteLine("Tutorial resources deleted; unrelated groups were not selected.");
```

The browser application's Cleanup action targets its own application group, not this book's separate group. Deleting a completed deployment history entry also does not clean up its resources. Re-run chapters 4 and 5 to provision the book group again; the local project files and settings remain available.

The standalone messaging lab has a separate `quinzee-messaging` group. Follow its cleanup procedure while the Functions helper remains running to process runtime removal. Deleted function app names stay reserved by tombstones, and private helper keys/parameter files remain for explicit cleanup. Group deletion also destroys Service Bus messages.

Step 3: keep state and trust together

The application environment's private directory holds identity configuration, HTTP-service state, certificate material and helper logs. Native database data also lives in labelled Docker volumes; the engine supervisor's private CA/state is kept separately from the cloud's mount. Copying just the cloud directory is not a backup of the complete environment.

For a manual backup, stop client writes and all relevant processes, then save the cloud state, certificates, corresponding helper state and native volumes as one consistent set. Protect those copies as secrets. No automatic Azure-style backup/restore service is provided. Do not use broad Docker prune commands as a routine Quinzee shutdown procedure.

`make clean` removes build outputs, not runtime state. `make purge` additionally removes dependency caches and the default Compose project's containers/images; it is not needed for ordinary use or to refresh a browser page.

Logs and quick checks

For the application environment, inspect `.local/application/development/environment.log`. For the standalone cloud, use `make cloud-logs`. For telemetry tools, use `make observability-status` and `make observability-logs`.

Cloud request logs are structured JSON. Trace IDs help connect related events; HTTP health does not prove that a particular database or queue worker is ready. Use QHQ's observability page or the authenticated diagnostics endpoint for that additional state.

Related reading

- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Quinzee Administrator’s Guide: Configure the cloud and its launch environment
- Quinzee Internals: Build System and Dependency Cache
- Quinzee Internals: Containers and runtime ownership
- Scenario overview: Working Without an Internet Connection
- Scenario overview: Building a Personal Cloud on a Home Server
- Scenario overview: Testing Backup and Recovery

16. Troubleshoot a connection or workflow

Work from the network inward: correct environment, DNS, TLS, authentication, resource readiness, then the requested operation. Preserve the failing step, status/error code and trace ID; keep tokens, passwords and signed URLs out of shared diagnostics.

Step 1: identify the environment

Chapters 2–11 use `.local/application/development`; the messaging lab in chapters 12–13 uses the standalone cloud. `make cloud-up` and the default `make qhq-run` use `.local/quinzee`. Each has its own credentials and CA. The application environment's network, DNS subnet and cloud port are recorded in `environment.json`; do not assume it uses 11443 or `172.29.53.53`.

Inside the chapter 3 SDK container:

```
getent hosts management.azure.com
curl --fail --silent --show-error https://management.azure.com/health
```

The address must be the local cloud container, and the response must name `quinzee.cloud`. If not, stop credentialed requests until the client is using Quinzee DNS and the matching public trust bundle.

Step 2: match the symptom

Symptom	Check and action
SDK sees real Azure or cannot find the tenant	Check both `--network` and `--dns`. Enable Azure emulation on a standalone cloud. The application helper enables it for its own environment.
`SSL certificate problem` or .NET TLS failure	Check the exact environment's CA and `SSL_CERT_FILE`; verify the hostname matches the certificate. Do not disable verification.
Monitoring hostname fails while other services work	Certificates issued before monitoring names were added may need deliberate renewal and refreshed client trust.
Browser certificate warning	Import the public development CA printed by this environment. Grafana has a separate CA under `.local/observability`.

Symptom	Check and action
`invalid_client`	The tenant/client/secret must all come from this environment. A browser access token is not a client secret.
401 after cloud restart	Recreate the client/credential or allow its refresh path; old signing-key tokens are invalid.
Monitoring query rejects an ARM token	Use `LogsQueryClient` with `TokenCredential`; it requests the correct audience.
503 for one service	Check QHQ Modules; the responsible service may be unloaded.
Deployment is Running/Updating for a long time	Keep the supervisor running; inspect engine heartbeat and operation error. Native provisioning fails explicitly after its readiness timeout.
MySQL certificate revocation error	Ensure the supervisor's signed CRL endpoint is reachable on the private bridge; retain `VerifyFull`.
SQL/Redis host cannot be reached from the host desktop	Native ports are private. Run the client on the engine network and use its DNS hostname.
A stopped database is still listed	Resource existence and native readiness differ. Start it and wait for the operation.
Queue receive returns no message	It may be leased, expired, consumed by another worker or in another queue/account. Peek is not a reservation.
Message delete/update rejects the pop receipt	Retain the latest receipt returned by receive or update; do not reuse an earlier one.
Service Bus HTTPS works but the SDK cannot connect	Check the separate AMQP/TLS listener, forwarded port, private DNS and CA. Select `AmqpTcp`; WebSockets are unsupported.
Service Bus settlement reports a lost lock	Retain the current received message/lock, settle before expiry, and renew before long processing. Redelivery requires idempotent handlers.
Service Bus message disappears from the active queue	Inspect `$DeadLetterQueue`, TTL and other consumers. A running Functions trigger competes with manual receivers.
Function app stays Updating or fails readiness	Keep the helper running on the matching state/network; check image registration. A missing helper times out after 180 seconds.
HTTP function returns 401	Supply its function key, obtained through ARM `host/default/listkeys`, in `x-functions-key`. An ARM token is not an invocation key.
HTTP function returns 503 or 502	Check app state, helper heartbeat and upstream TLS/reachability. A stale helper stops forwarding; keep certificate verification enabled.

Symptom	Check and action
Function queue inspection is empty after restart	The example keeps its received messages in memory. Use persistent application storage for durable results.
Deleted function name cannot be reused	Tombstones reserve names; choose a fresh name rather than editing private runtime state.
Blob overwrite or setting update returns 412	An ETag condition failed. Re-read and resolve the concurrent change.
Blob SAS returns 403	Check expiry, scope, permissions, signing key and HTTPS; do not log the signed URL while diagnosing it.
Secret name cannot be recreated	A soft-deleted secret may need recovery or intentional purge.
Query returns unsupported syntax/result-too-large	Use the documented KQL stages and add a filter, `take`, or aggregation.
No telemetry from a short C# program	Keep the provider alive, end the activity, flush it and check the component connection string/credential. Set `TracesPerSecond = null` and `SamplingRatio = 1` for 100% sampling; the rate limit otherwise overrides the ratio.
QHQ request trace is missing	The native history holds only 128 recent requests; use exported traces for a longer view.
Grafana has no record for an Application Insights event	The two destinations are separate; configure an OTLP exporter if that application should also send to the platform stack.
Application Verify fails on a second attempt	Verification consumes the recovery message. Run the full workflow to create a new one.
Shutdown cannot remove a Docker network	Exit SDK/debug containers still attached to that network, then stop the owning environment.

Step 3: resolve setup errors

If `docker-dotnet:latest` is missing, build it in the Triglav-OS repository with `make -j8 docker-dotnet`, or set `TRIGLAV_OS_DIR` so Quinzee can load the export. A runtime image does not contain the .NET SDK; use the Microsoft SDK container for `dotnet build` and `dotnet run`.

If restore/build cannot download packages, check network access and the exact locked package sources. Do not replace a checksum to make a failed verification pass. A QHQ build also needs access to the independent Simple Git repository; server-only Docker builds do not need that UI package.

For QHQ's `rasta did not start`, check the graphical session, SDL2 availability and `build/qhq/rasta.log`. For a used port, stop the process you own or select a different environment port; do not kill unrelated applications.

Step 4: reproduce with the supplied application

Run `make application-test` for a disposable complete workflow, or `make application-observability-test` for the additional backend and outage checks. These commands create their own groups, networks and engine resources. They clean successful disposable application state; a failed run retains private logs/results under `.local/application/test-*` for diagnosis. The optional telemetry stack is separately operated; stop it with `make observability-down` when you are done.

For messaging, use `make service-bus-sdk-test`; for real HTTP/queue-trigger execution, follow the prerequisites and run command in the Functions chapter. These fixtures use isolated state rather than the persistent application environment.

Share the failed operation/status and a redacted trace ID, not a copy of the private environment directory.

Related reading

- Quinzee Administrator’s Guide: Observe the platform and the application separately
- Quinzee Administrator’s Guide: Diagnose failures from the outside inward
- Quinzee Internals: HTTP library and asynchronous execution
- Quinzee Internals: Connection and streaming lifecycle
- Scenario overview: Debugging Cloud Applications Locally
- Scenario overview: AI-Assisted Cloud Debugging

17. Build Quinzee from source

You need a native build to modify the C++ emulator or run QHQ. The application quickstart builds its cloud image separately, so it does not require a prior native build.

On Linux x86-64, install GCC/G++ 13 or newer, CMake 3.28 or newer, GNU Make, Bash, Git, curl, tar, Perl, Python 3.9+, `sha256sum`, `flock` and binutils. Docker Engine, Compose v2 and BuildKit support the container targets. A graphical QHQ session also needs SDL2. The verified build uses GCC 15.2, CMake 4.2, Boost 1.92.0 and OpenSSL 3.5.9. The repository manifest selects exact dependencies; `make deps` verifies and builds them instead of linking arbitrary system libraries.

From the repository root:

```
make deps
make
make test
```

Builds run in parallel. Boost, OpenSSL and the independent Simple UI toolkit come from verified sources in `.packages/`. The Simple revision is obtained from GitHub; a nearby `../simple` checkout is not used for linking. Existing Git credentials may be needed to fetch its repository. No credentials should be copied into dependency manifests or container images.

Command	Output/use
`make`	Debug binaries under `bin/debug/`
`make PRESET=release`	Hardened release binaries under `bin/release/`
`make qhq`	QHQ and its assets
`make rasta`	The development viewer under `bin/tools/rasta`
`make test`	C++ unit/integration/dependency and helper contract tests
`make docker-test`	Disposable container, HTTPS, DNS and SDK checks
`make application-check`	.NET/frontend formatting/build and browser accessibility checks
`make docker-lint`	C++ formatting and static analysis using the container toolchain

For a native service-only build, configure with `cmake --preset debug -DTRIGLAV_BUILD_QHQ=OFF`, then build with `make`. Set it back to `ON` before building QHQ. Do not install unrelated system Boost or OpenSSL versions to bypass the project's pinned dependencies.

`make clean` removes `build/` and `bin/` while keeping the dependency cache. `make purge` is broader and forces downloads/rebuilds; inspect its effects before using it. Neither command is an application data-backup procedure.

For memory checking, build with `make PRESET=asan`, then run `ctest --preset asan --output-on-failure`. For race checking use the tsan preset. If the sanitizer cannot start because the environment blocks its address-space requirements, report the run as incomplete. Do not weaken security settings to turn that startup failure into a claimed pass.

PostgreSQL client source builds also require Bison, Flex and bzip2. The Docker builder installs these tools; native builds need them on PATH.

Related reading

- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Quinzee Administrator’s Guide: Configure the cloud and its launch environment
- Quinzee Internals: Build System and Dependency Cache
- Quinzee Internals: Containers and runtime ownership
- Scenario overview: Working Without an Internet Connection
- Scenario overview: Building a Personal Cloud on a Home Server
- Scenario overview: Giving AI Coding Agents Their Own Cloud

18. Certificates, DNS and Azure URLs

A correct hostname and a trusted certificate are separate requirements. DNS chooses where to connect; certificate verification establishes that the peer is allowed to serve that name. Changing one does not replace the other.

Trust domains

Public trust material	Purpose
Quinzee Development Root CA	Native `quinzee.cloud`/service names and the localhost application UI
Quinzee Azure Emulation Root CA	The explicitly supported Microsoft/Azure hostnames inside a development environment
Engine CA	PostgreSQL, MySQL and cache hostnames; maintained by the separate supervisor
Functions runtime CA	Private gateway-to-host HTTPS; maintained by the separate Functions helper
Local telemetry CA	Grafana and the OTLP collector

These CAs have private signing keys that stay with their owner. Clients need only public `.crt` files or a public trust bundle. The HTTP server receives its own leaf certificate/key, not a CA key. Application settings and leaf keys must not be copied into a shared image.

The application helper creates its certificates under `.local/application/development/certificates`, and puts the client's public bundle in `client/trust.pem`. The standalone cloud uses `.local/certificates`. Mixing these environments is a common cause of TLS failures.

Native and Azure addresses

Service	Azure URL/name in the prepared container	Native local name
Identity	`login.microsoftonline.com`	`quinzee.cloud:11443` (tenant paths)
ARM	`management.azure.com`	`quinzee.cloud:11443`

Service	Azure URL/name in the prepared container	Native local name
Blob	`<account>.blob.core.windows.net`	`<account>.blob.storage.quinzee.cloud:11443`
Queue	`<account>.queue.core.windows.net`	`<account>.queue.storage.quinzee.cloud:11443`
Table	`<account>.table.core.windows.net`	`<account>.table.storage.quinzee.cloud:11443`
Service Bus AMQP	`<namespace>.servicebus.windows.net:5671`	`<namespace>.servicebus.quinzee.cloud:5671`
Functions HTTP	`<app>.azurewebsites.net`	`<app>.functions.quinzee.cloud:11443`
Key Vault	`<vault>.vault.azure.net`	`<vault>.vault.quinzee.cloud:11443`
App Configuration	`<store>.azconfig.io`	`<store>.appconfig.quinzee.cloud:11443`
PostgreSQL	`<server>.postgres.database.azure.com:5432`	`<server>.postgres.quinzee.cloud:5432`
MySQL	`<server>.mysql.database.azure.com:3306`	`<server>.mysql.quinzee.cloud:3306`
Redis-compatible cache	`<cache>.<region>.redis.azure.net:10000`	`<cache>.redis.quinzee.cloud:10000`
Monitoring ingestion	`dc.applicationinsights.azure.com`	`dc.monitor.quinzee.cloud:11443`
Monitoring queries	`api.loganalytics.azure.com`, `api.loganalytics.io`, `api.applicationinsights.io`	`api.monitor.quinzee.cloud:11443`

HTTP services use HTTPS; native SQL/cache ports use their own TLS protocols. The table's native HTTP port is the default. Prefer resource/deployment output endpoints so port overrides and returned names remain correct.

Service Bus uses a separate AMQP/TLS listener; its HTTPS messaging API uses the normal cloud HTTPS port on the namespace host. Functions clients trust the public cloud gateway, not the private runtime CA. The helper verifies its own upstream connection. Existing installations need refreshed leaf certificates for newly supported service names; if the constrained Azure CA changes, refresh that public CA in clients that explicitly use Azure emulation too.

The Azure listener uses port 443 **inside the private Docker network**. It is not exposed as your host's public port 443. Therefore changing a host's DNS alone is not sufficient to make an unmodified host SDK use the container listener. The client-container approach provides the right network reachability, DNS and trust without changing the host.

Enable Azure names on a standalone cloud

1. Run `make cloud-up` on the host. It prepares the native and constrained Azure leaf certificates and starts cloud/DNS.
2. Connect QHQ to this instance, open **Emulator → DNS**, enable Azure emulation and apply the change.
3. Join your client container to `triglav-http_quinzee` and set `--dns 172.29.53.53`, unless you changed the standalone network settings.
4. Trust the public Azure-emulation CA in that client. Add the engine CA as well if using SQL/Redis. Never mount CA private keys.
5. Confirm DNS gives the emulator address and verify `/health` before sending development credentials.

For the book's application environment, the helper already performs these steps with its own private network. Use the dynamic values from chapter 3. Unsupported Azure names continue resolving normally; this is not an offline replacement for all Microsoft endpoints. Only use explicit local application credentials, not your production account, for emulator exercises.

Certificate lifecycle

Standalone certificates are prepared by `make certs`; the Azure certificate is also prepared by `make cloud-up`. Valid existing certificates are reused. Leaf certificates have a 90-day development lifetime. Engine certificates and telemetry certificates have their own owners and renewal procedures.

If certificates predate newly supported hostnames, a still-valid leaf may not contain those names. Deliberate regeneration of standalone certificates with `make certs FORCE=1` replaces trust material: stop affected work, preserve any needed backup, restart the services and update every client's public trust bundle. Do not use this command as a routine response to an unrelated 401. The application environment's certificates require using its own directory; the default `make certs` does not update them.

Never use `curl -k`, a permissive certificate callback or a disabled hostname check to work around a setup error. Keep the supervisor running for the native engine CRL endpoint so clients can retain revocation verification.

Related reading

- Quinzee Administrator’s Guide: Make names, certificates and permissions agree
- Quinzee Internals: TLS, identity and DNS
- Quinzee Internals: Follow a packet through names, sockets and namespaces
- Scenario overview: Running Applications at the Edge
- Scenario overview: Building an Isolated Private Application Environment

19. Debug your application and QHQ

Debug C# where its network configuration works

Keep source on the host and run the chapter 3 SDK container. Build with `dotnet build --no-restore` and run with `dotnet run --no-restore`. Your editor can attach to that development container using its normal C# debugging tools; keep the same `SSL_CERT_FILE`, DNS configuration and private settings mount. Do not move the process to the host and assume its Azure-name resolution is unchanged.

Start with breakpoints around the application operation that fails. Inspect status/error codes and resource names. Avoid copying token, connection-string or secret values out of the debugger. For the supplied web application, use its step status and trace ID to find the corresponding platform trace.

An application exception can occur after provisioning succeeded. Inspect resources before rerunning; an incremental deployment may safely reuse them, whereas cleanup deliberately removes data.

Debug QHQ with F5

In VS Code, install the C/C++ extension and GDB, open the repository and choose **QHQ (Rasta) + quinzee.cloud (Docker)**. F5 builds QHQ and its viewer, prepares certificates and starts the default cloud environment. Breakpoints in `src/qhq/` run against that instance, not automatically against the book's application environment.

Stopping that debugger session stops its viewer/environment while preserving its state. To manage the already-running application environment instead, use the QHQ chapter's explicit connection.

Debug the native cloud

Build a debug binary and run it under `gdb --args` with its explicit data and TLS paths as in native operation. Use a separate data directory for debugging; two cloud processes must not write the same state.

The HTTP demonstration server also supports `make debug-up` and `make debug-down`, with gdbserver on loopback port 2345. That program tests the transport library; it does not provide Azure services.

Related reading

- Quinzee Administrator’s Guide: Observe the platform and the application separately
- Quinzee Administrator’s Guide: Diagnose failures from the outside inward
- Quinzee Internals: HTTP library and asynchronous execution
- Quinzee Internals: Connection and streaming lifecycle
- Scenario overview: Debugging Cloud Applications Locally
- Scenario overview: AI-Assisted Cloud Debugging

20. Run the emulator natively

The cloud and QHQ can be built and run as native Linux binaries. Docker is the integrated development setup used by the book; there is no packaged native installer or ready-made service-management setup in this repository.

After a native debug build, on the **host**:

```
make certs
bin/debug/quinzee.cloud \
  --address 127.0.0.1 --port 11444 \
  --data-dir "$PWD/.local/quinzee-native" \
  --tls-cert "$PWD/.local/certificates/quinzee.cloud/server.crt" \
  --tls-key "$PWD/.local/certificates/quinzee.cloud/server.key"
```

In another terminal:

```
curl --fail --cacert .local/certificates/ca/quinzee-dev-root-ca.crt \
  https://localhost:11444/health
```

This starts the HTTPS API and creates private identity/state for that directory. Add `--service-bus-port 5671` to enable AMQP/TLS with the same TLS files; without it the Service Bus HTTPS API can be available while AMQP remains disabled. It does not automatically configure a DNS server, Azure listener, native engine supervisor or your application's trust store. `--help` lists configuration options, including module selection and the separate Azure-emulation listener.

Native SQL/Redis provisioning still uses the separate Docker-based supervisor. It needs a private state directory, the matching cloud mailbox directory and a Docker network reachable by the clients. Merely running the cloud binary natively does not replace the engine runtime with embedded databases.

Functions also needs its separate Docker helper, registered images and a reachable private bridge. A loopback-only cloud cannot serve the broker to function containers; configure the reachable address, DNS and trust explicitly. The helper takes the same cloud data directory, a separate private state directory, an approved image registry and an existing reachable Docker network. Run it as a non-root Docker operator; keep its private state outside cloud data. The hosting chapter gives the complete invocation.

For unattended use, arrange process ownership, startup ordering, logs, certificate renewal and consistent data backups explicitly. Bind only the interfaces you intend clients to reach. Never run two hosts against one writable data directory.

QHQ's Rasta backend provides a graphical development window. Its framebuffer backend can run on a Linux console with the required display/input access; those device permissions and an installation service are operator setup, not provided automatically by this tutorial.

Related reading

- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Quinzee Administrator’s Guide: Configure the cloud and its launch environment
- Quinzee Internals: Build System and Dependency Cache
- Quinzee Internals: Containers and runtime ownership
- Scenario overview: Working Without an Internet Connection
- Scenario overview: Building a Personal Cloud on a Home Server
- Scenario overview: Giving AI Coding Agents Their Own Cloud

21. Give applications and people the right identity

An app registration describes a client or API. Its service principal is the application's identity inside the local tenant. A managed identity is a service principal whose credentials the hosting environment supplies. A user is a person who can authenticate interactively. A group connects a set of users to grants. AAD, Azure AD and Microsoft Entra ID are names for the identity technology; they do not describe three separate databases in Quinzee.

Use a fresh development group and synthetic accounts in this lesson. Start with the bootstrap identity to register and assign objects, then run the application with the restricted identity you created. Keep those two roles separate.

Manage identities in QHQ

Connect QHQ with your environment's bootstrap credentials. Expand **Azure / Entra ID** in the navigation. Every form uses the same public HTTPS APIs as clients. Use **Browse** to select an object, then **Create / Edit** to change it. **Save** updates the selection; **Create** makes a new object. Deleting requires the confirmation checkbox. Refresh after changing records through another client.

1. In **Users**, enter a display name, username such as `alice@example.test`, and a password of at least 12 characters; choose **Create**. Select a user to rename, enable/disable, reset the password or delete it. Password input clears after submission.
2. In **Groups**, create a security group with a display name and stable group account name. Select it, open **Members**, choose a user or group, and use **Add member** or **Remove member**. Browse shows the current member IDs. Cycles are rejected.
3. In **App registrations**, create a display name and exact redirect URI. Select **Public client** for an interactive client without a secret. Select **Expose API** for an API that will issue application roles. The local **Application enabled** switch controls this emulator's sign-in policy.
4. Select a registration and open **Credentials** to add a secret. Copy the one-time value into private application configuration; it cannot be retrieved later. Refresh lists credential IDs for removal. Public clients need no secret.
5. In **Enterprise applications**, choose the app registration and create its service principal. This is the tenant identity used for assignments. Select it to edit its display name, sign-in status or **Require an app role assignment**.
6. In **App roles**, choose a registration in Browse, then define a role in Create / Edit, for example `Orders.Read`. Choose whether users/groups, applications, or both may receive it.

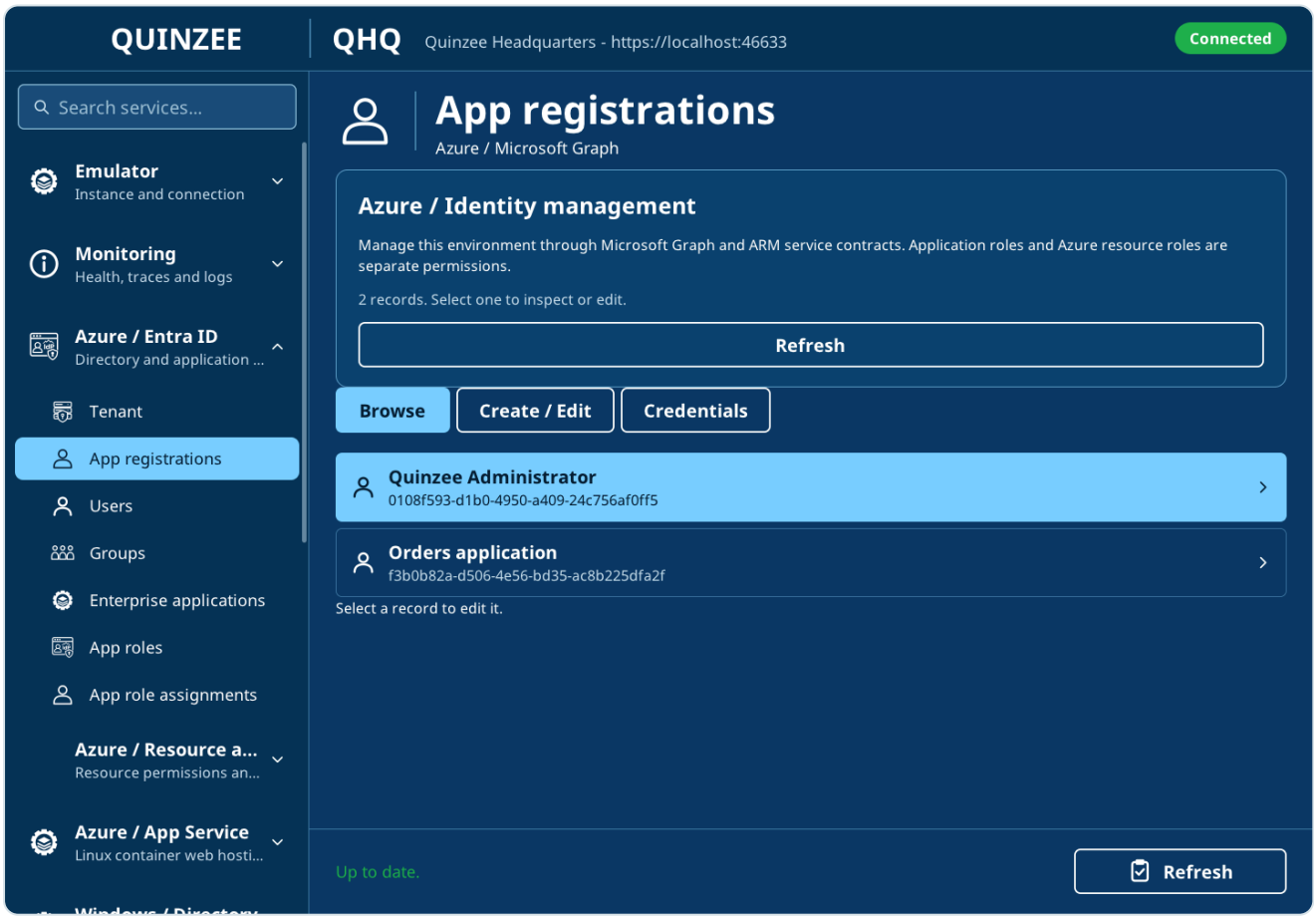
Disable a role and remove its assignments before deleting the definition.

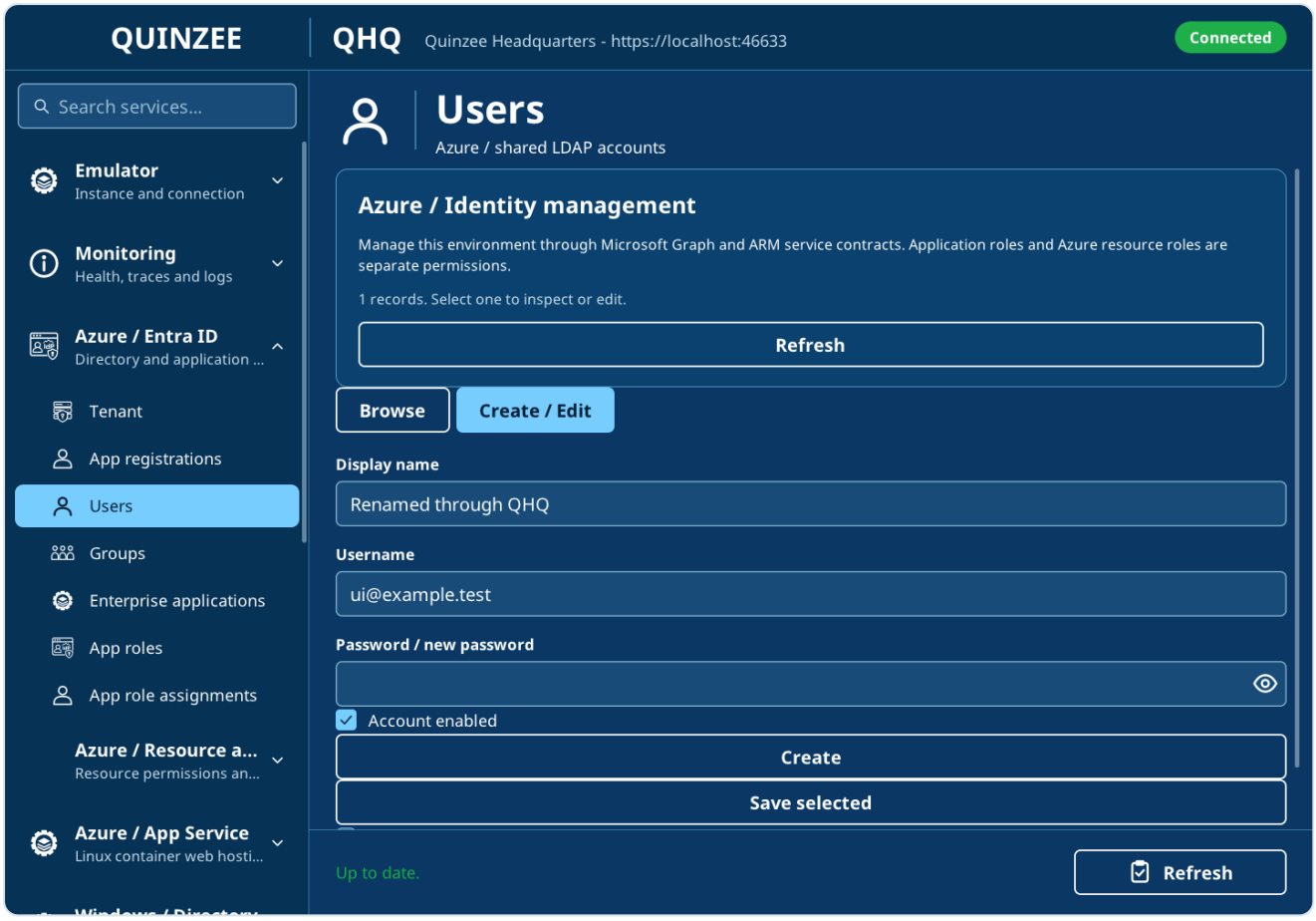
7. In **App role assignments**, choose the target Enterprise application in Browse. In Create / Edit choose a user, group or service principal and role, then **Create**. Group assignments grant roles to direct user members only; nested membership does not grant application roles.

Under **Azure / Resource access**, **Managed identities** creates standalone user-assigned identities in a resource group. **Role assignments** assigns a resource role at the entered subscription, resource-group or resource scope. Choose a principal and role, then Create. Group-based RBAC includes nested user membership. App roles and Azure resource roles are independent; neither grants directory administration.

Under **Windows / Directory**, **LDAP directory** shows the shared backend and listener status and links to the same Users and Groups. LDAP clients can use the POSIX view or the Windows-oriented user/group view. This module does not implement Windows domain join or Kerberos. Configure PostgreSQL and OpenLDAP using the instructions below before connecting LDAP clients.

Emulator → Modules lists each Azure/Windows frontend separately, including its dependencies. Unloading a frontend preserves records in the shared identity authority. The labels describe the API family; they do not indicate a connection to a Microsoft cloud tenant. Full cloud integration remains separate work.





Captured from a disposable HTTPS test environment. The accounts and IDs are synthetic. Scroll the form to reach additional actions below the visible area.

Step 1: understand the IDs before writing code

Creating `/v1.0/applications` returns `id` and `appId`. Use `id` when editing the registration and `appId` as the OAuth client ID. Creating its service principal returns another `id`: assign Azure roles and application permissions to that principal ID. Display names are labels and need not be unique. Resource IDs, object IDs and client IDs are not interchangeable.

The native HTTPS `/v1.0` directory API requires a token for `https://graph.microsoft.com/.default`. An ARM call requires `https://management.azure.com/.default`. Graph-shaped management is exposed at the local native origin; this does not add graph.microsoft.com DNS interception. QHQ's forms already select the correct audience for you.

To automate registration, POST the following JSON bodies to the named paths, using a Graph-audience administrator bearer token and verified HTTPS:

```
{ "displayName": "Orders worker" }
```

Send that to `/v1.0/applications`, then use its appId:

```
{ "appId": "<returned-client-id>" }
```

Send the second body to `/v1.0/servicePrincipals`. POST `{}` to `/v1.0/applications/<registration-id>/addPassword`; store the returned secretText privately immediately. GET later exposes only credential key IDs. A public interactive application uses `isFallbackPublicClient: true` and needs no secret. For browser authorization code flow also register its exact web.redirectUris; for device code, no callback URI is needed.

Step 2: give the identity a resource permission

At the intended group scope, PUT a UUID-named role assignment using API 2022-04-01. The complete resource body for Reader is:

```
{
  "properties": {
    "principalId": "<user-or-service-principal-object-id>",
    "roleDefinitionId": "/subscriptions/<subscription-
id>/providers/Microsoft.Authorization/roleDefinitions/acdd72a7-3385-48ef-bd42-f606fba81ae7"
  }
}
```

The path is `<group-resource-id>/providers/Microsoft.Authorization/roleAssignments/<assignment-uuid>`. Save that UUID for redeployment; another UUID creates another assignment attempt, not an update to the original identity. The IaC hosting lesson gives the complete Bicep/ARM version. Reader permits resource reads, Contributor permits resource changes, and Owner additionally permits role assignment changes. Service Bus Data Sender/Receiver/Owner are separate messaging grants. New identities without an applicable role are denied. None of these resource roles grants directory administrator access.

Step 3: sign a person in using MSAL device code

Create a public registration and local user in QHQ. Give the user Reader on your training group. A new user's password is 12–1,024 bytes; there is no default human password. Save a private `login.json` with your values:

```
{
  "origin": "https://localhost:11443",
  "tenantId": "<local-tenant-uuid>",
  "clientId": "<public-application-client-id>",
  "groupId": "/subscriptions/<subscription-id>/resourceGroups/book-iac"
}
```

Prepare an isolated .NET 10 console project on the host. Reuse the exact verified MSAL dependency and lock, then place the complete code below in Program.cs:

```
mkdir -p .local/book-login
cp tests/compatibility/identity/IdentitySmoke.csproj .local/book-login/BookLogin.csproj
cp tests/compatibility/identity/packages.lock.json .local/book-login/packages.lock.json
```

```
using Microsoft.Identity.Client;
using System.Text.Json;

using var document = JsonDocument.Parse(await File.ReadAllTextAsync(args[0]));
var settings = document.RootElement;
string Read(string key) => settings.GetProperty(key).GetString()!;
var client = PublicClientApplicationBuilder.Create(Read("clientId"))
    .WithAuthority(new Uri(Read("origin") + "/" + Read("tenantId")), false)
    .WithInstanceDiscovery(false)
    .Build();
var scopes = new[] { "https://management.azure.com/.default" };
using var timeout = new CancellationTokenSource(TimeSpan.FromMinutes(5));
var result = await client.AcquireTokenWithDeviceCode(scopes, code =>
{
    Console.WriteLine(code.Message);
    return Task.CompletedTask;
}).ExecuteAsync(timeout.Token);
using var http = new HttpClient();
using var request = new HttpRequestMessage(HttpMethod.Get,
    Read("origin") + Read("groupId") + "?api-version=2023-05-01");
request.Headers.Authorization = new("Bearer", result.AccessToken);
using var response = await http.SendAsync(request, timeout.Token);
response.EnsureSuccessStatusCode();
Console.WriteLine("Signed in as " + result.Account.Username);
Console.WriteLine("The assigned resource group is readable.");
// The SDK cache is process-local here. Do not print or persist access tokens.
```

Build with the local dependency cache, then run with the native development root trusted for this process:

```
DOTNET_CLI_HOME="$PWD/.packages/dotnet-home" \
NUGET_PACKAGES="$PWD/.packages/nuget" \
  dotnet build .local/book-login/BookLogin.csproj --property:RestoreLockedMode=true
SSL_CERT_FILE="$PWD/.local/certificates/ca/quinzee-dev-root-ca.crt" \
  dotnet run --no-build --project .local/book-login/BookLogin.csproj \
  -- "$PWD/.local/book-login/login.json"
```

Open the printed local verification URL in a browser that trusts this lab's public development root. Check the application name and code, sign in as the new user and approve. The program should print the username and confirmation that the group is readable. It does not print the token. MSAL handles polling, expiry and token acquisition; the application uses the returned access token only for the intended resource API. Do not approve a device code supplied by someone else. Instance discovery is disabled because this is a private local authority, while normal TLS verification remains enabled.

For browser code flow, use S256 PKCE, state and an exact redirect URI. MSAL's interactive acquisition handles those protocol pieces. `make identity-sdk-test` verifies the real browser flow, refresh rotation and Reader restrictions in an isolated environment. MFA, federation, Conditional Access and Azure CLI login are not implemented by the local authority.

Step 4: use one group for LDAP visibility and application roles

Create an Engineering group, add your user as a direct member, and create an Orders API registration with Expose API selected. Define an app role with value `Orders.Read`, allow Users, create the API's Enterprise application and set assignment required. Assign Engineering to Orders.Read.

The equivalent registration PATCH body is:

```
{
  "identifierUris": ["api://<api-client-id>"],
  "appRoles": [{
    "id": "<new-role-uuid>", "displayName": "Read orders",
    "description": "Read application orders", "value": "Orders.Read",
    "isEnabled": true, "allowedMemberTypes": ["User"]
  }]
}
```

POST an assignment to `/v1.0/servicePrincipals/<api-principal-id>/appRoleAssignedTo`:

```
{
  "principalId": "<engineering-group-id>",
  "resourceId": "<api-principal-id>",
  "appRoleId": "<role-uuid>"
}
```

Request `api://<api-client-id>/.default`. Its v2 token audience is the API's client ID and its roles claim contains Orders.Read. The API must validate signature, issuer, audience, lifetime and the required role before reading orders. A group app-role grant applies only to direct user members; nested groups do not convey it. Azure resource RBAC uses transitive group membership, so do not use the two permission models interchangeably. Removing a grant affects newly issued tokens; an already-issued external API token can remain valid until expiry.

When your administrator has enabled the shared PostgreSQL/LDAPS service, inspect the same group with an LDAP client:

```
LDAPTLS_CACERT=/absolute/path/to/lab-ca.crt ldapsearch -LLL -x \
  -H ldaps://127.0.0.1:11636 \
  -D 'uid=alice@example.test,ou=users,dc=quinzee,dc=cloud' -W \
  -b 'ou=windows,dc=quinzee,dc=cloud' '(objectClass=group)' \
```

```
cn member memberOf objectGUID sAMAccountName
```

Use the actual account you created, its password prompt and the certificate's matching address. LDAP and the HTTPS directory share the record; no synchronization step is needed. This Windows-oriented view is read-only and supports nested member DNs. The POSIX view retains direct memberUid values. There is no Kerberos, Windows domain join or LDAP write API. LDAP lookup does not grant an app role; it only exposes directory membership.

Give a Function its own identity

Deploy `identity: { type: 'SystemAssigned' }` on a site, or create and attach a user-assigned identity. Assign the appropriate Service Bus role to its principal. For a binding named ServiceBusConnection use these app settings:

```
ServiceBusConnection__fullyQualifiedNamespace=booklab-bus.servicebus.quinzee.cloud
ServiceBusConnection__credential=managedidentity
ServiceBusConnection__clientId=<user-assigned-client-id>
```

Omit clientId for the system identity. The helper supplies IDENTITY_ENDPOINT and IDENTITY_HEADER privately. Do not set them yourself or expose tokens through a public web route. `make functions-identity-test` verifies a real queue trigger with SAS disabled. The complete identity-hosting IaC lesson explains each resource and permission. Managed identities cannot bind to LDAP with a password.

Clean up the identities you created

Remove application-role and resource-role assignments first, then delete the training user/group or application through QHQ. Disable and unassign a role before deleting its definition. Delete disposable client secrets by key ID. System identities disappear with their site; user-assigned resources survive until explicitly deleted. Keep the bootstrap identity for administration and avoid deleting unrelated records merely because their display names resemble the exercise's names.

Related reading

- Quinzee Administrator’s Guide: Operate the shared Entra and LDAP directory
- Quinzee Internals: TLS, identity and DNS
- Quinzee Internals: One directory behind HTTPS and LDAP
- Scenario overview: Testing Security and Access Controls

22. Host a web application

Azure App Service runs an application as a service. Quinzee's Linux container subset runs your real web process behind HTTPS. Start with the disposable example on the host:

```
make app-service-test
```

It builds a small ASP.NET Core app and verifies its gateway, identity and lifecycle in a disposable network. That test image has a token-returning probe for verification; use the minimal application below for your own shared lab.

Step 1: build your own web image

Create a private working project on the host:

```
mkdir -p .local/book-web
cp tests/compatibility/app_service/WebFixture.csproj .local/book-web/
cp tests/compatibility/app_service/packages.lock.json .local/book-web/
```

Save this complete Program.cs there. It has one public route and never returns credentials or workload tokens:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => new {
    message = "Hello from my web app",
    greeting = Environment.GetEnvironmentVariable("Greeting")
});
app.Run();
```

The project targets .NET 10 using the ASP.NET Core shared framework. Publish and package it with the verified runtime Dockerfile:

```
DOTNET_CLI_HOME="$PWD/.packages/dotnet-home" \
NUGET_PACKAGES="$PWD/.packages/nuget" \
  dotnet publish .local/book-web/WebFixture.csproj -c Release \
  --property:RestoreLockedMode=true --output "$PWD/.local/book-web/publish"
docker build --file tests/compatibility/app_service/Dockerfile \
  --tag quinzee-book-web:local .local/book-web/publish
```

Step 2: register the image and start its helper

Leave the standalone cloud from `make qhq-run` running. In another host terminal:

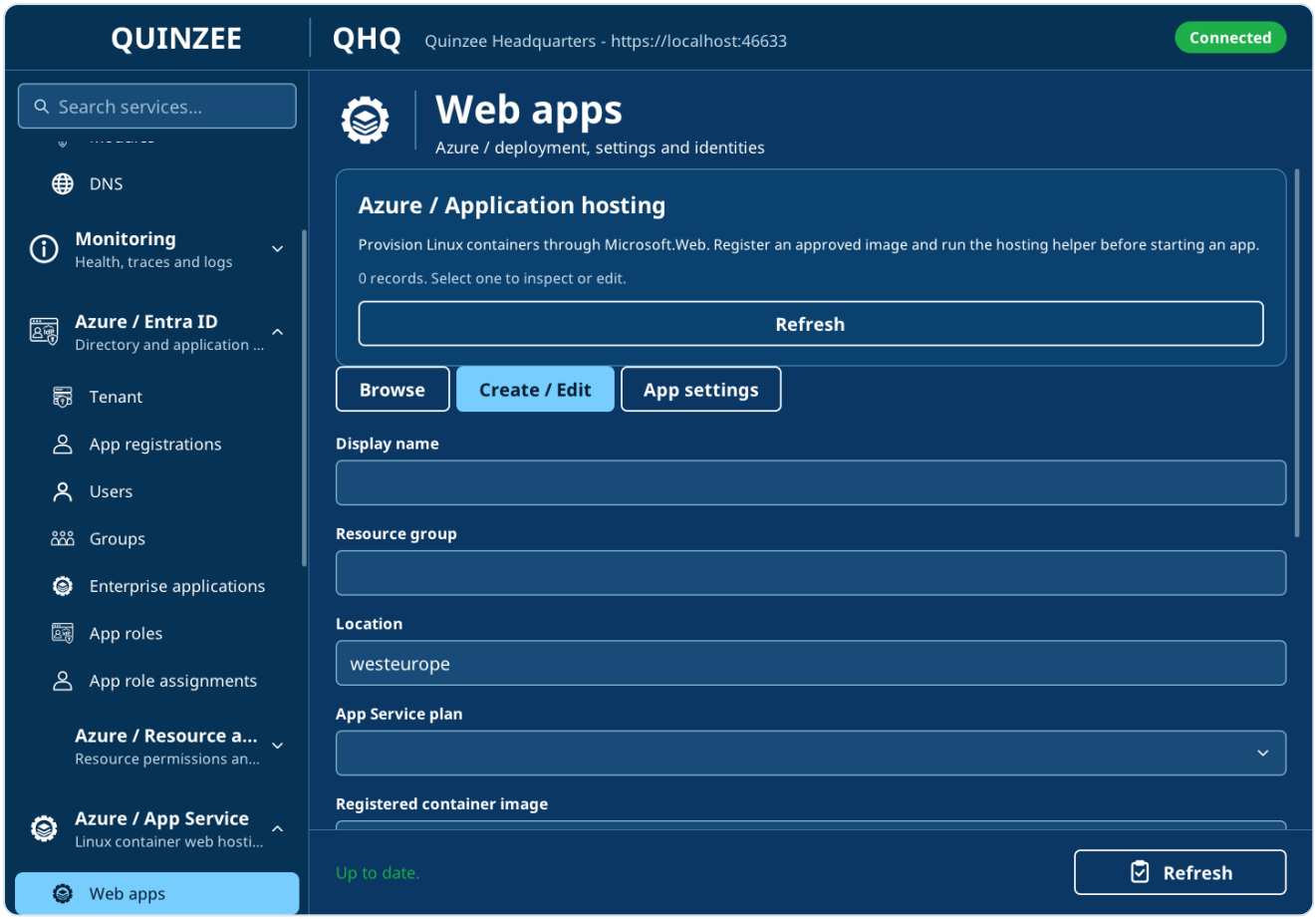
```
python3 infrastructure/functions/register.py \
  --image quinzee-book-web:local \
  --registry .local/functions-helper/images.json
make functions-run
```

The registry maps the approved name to an immutable local image ID. The helper runs separately with Docker access, the cloud's data path, its private state, the matching network/DNS and the development CA. QHQ never launches Docker itself. Restart the helper after changing the registry. Keep it running while creating, updating, stopping or deleting apps.

ASP.NET Core uses the helper's injected Kestrel TLS certificate settings and listens on port 8443. The helper checks `/` for a successful response. An arbitrary HTTP-only container does not satisfy this contract. The process runs non-root with a read-only root and bounded resources; write temporary data only to the provided writable locations and use a real storage service for durable data.

Provision through QHQ

1. Create a resource group under Resource Manager.
2. Expand **Azure / App Service → App Service plans**. Check the subscription Scope in Browse. In Create / Edit enter a name, resource group and location, then Create. The supported plan is Linux B1/Basic with one instance.
3. Open **Web apps**. Check the subscription scope, then enter the app name, resource group, location and approved image name `quinzee-book-web:local` in Create / Edit. Select the plan. Optionally enable the system identity or attach a user-assigned identity. Create the app and refresh until the runtime is Running.
4. Open `https://<app-name>.appservice.quinzee.cloud/` using the configured DNS and trusted CA. The container must serve TLS on port 8443 and respond at `/`; the provided ASP.NET example already follows this contract.
5. Select the app in Browse. Use App settings to load setting names, add/replace a private value or remove a selected setting. Existing settings are preserved. Saving settings restarts the container. Secret values are masked and cleared from the input after submission.
6. Use Start, Stop or Restart for runtime control. Refresh to observe completion. To remove the app, select it and confirm Delete selected. Delete its plan only after removing the apps that use it.



The plan selector is populated after creating a plan. Scroll to reach the image, identity and lifecycle controls below this screenshot.

Functions use the same plan/identity/settings controls under **Functions → Function apps** and retain their real Azure Functions runtime. Application images must match the selected hosting kind. Attachments selected in QHQ preserve other existing user-assigned identities; clearing the attachment checkbox removes all user-assigned attachments. Use ARM for finer-grained multi-identity edits.

A managed identity grants no permissions by itself. Assign Azure resource roles or API app roles using the identity pages. The app receives a protected workload token endpoint, so it need not store a client secret. The normal web endpoint remains the application's responsibility: implement its user authentication and authorization in your code.

Emulator → Modules shows App Service separately from Functions. Unloading App Service blocks its gateway while retaining configuration and running containers; use Stop first if the process must stop. Dependency checks require unloading App Service before Functions. Slots, autoscale, Windows containers, Kudu deployment, Easy Auth and VNet integration are outside this implementation. There is no Azure tenant connection yet.

Related reading

- Quinzee Administrator’s Guide: Operate Functions and App Service workloads

- Quinzee Internals: Service Bus and Functions execution
- Quinzee Internals: From desired state to a real container
- Scenario overview: Developing an Azure Functions Application Locally
- Scenario overview: Using Quinzee as an Internal Server for a Small Office

23. Service compatibility and limits

Use this page when an SDK offers an operation that the emulator rejects. Client libraries generally expose more APIs than the local service implements. The main tutorial uses the package versions locked in `test/backend/packages.lock.json`. Service Bus and Functions have their own locks under `tests/compatibility/`; the Functions build also locks the generated binding-extension project. A newer package can select a different API version; upgrading it is a separate compatibility check, not a reason to silently change the emulator's API version.

Area	Supported contract	Important exclusions/limits
Identity	Client credentials, browser PKCE/device sign-in, managed identities, app roles, shared groups and scoped ARM/Service Bus RBAC; QHQ management forms	Single local tenant; no MFA, federation, full Graph, delegated consent or other data-plane RBAC
LDAP / Windows directory	PostgreSQL-backed POSIX and Windows-oriented user/group views, nested group DNs, verified LDAPS bind/search/compare	No LDAP writes, Kerberos, SIDs, domain join or full AD DS schema
App Service	Real Linux web containers, B1 plans, settings, lifecycle, managed identity and QHQ editing	TLS 8443 container contract; no Windows hosting, slots, autoscale, Kudu or cloud synchronization
ARM	Local subscriptions/groups, provider discovery, inventory, asynchronous operations	No real Azure subscription/billing backend
Deployments	Resource-group Incremental mode; inline array-form resources, supported expressions/dependencies/outputs	1 MiB request, 128 resources, 16 queued deployments; no Complete/nested/linked/loop-based deployment
Blob	Block blobs, metadata, ranges, ETags, staged block commit, account-key service SAS	256 MiB PUT/block requests; 1 GiB committed blob; no leases, snapshots, page/append blobs, account/delegation SAS
Queue	Queue/message CRUD, peek, leases, receipt rotation, TTL, metadata	64 KiB encoded text, 1,024 queues/account and messages/queue, 16 MiB queue file; no Queue SAS

Area	Supported contract	Important exclusions/limits
Table	Tables, typed entities, ETags, merge/replace/upsert, OData filters, `$select`, paging, entity group transactions; SharedKey/SharedKeyLite/bearer	Azure's entity limits; 1,000 tables/account, 4 GiB per emulator; no Table SAS, ACLs or service properties
Event Grid	Custom topics (Event Grid schema, CloudEvents 1.0), storage system topics (Blob Created/Deleted), WebHook and StorageQueue subscriptions with filters and retries	No advanced filters, dead-lettering, other destinations, namespaces or MQTT; 10,000 pending deliveries
Service Bus	Standard namespaces, queues, topics/subscriptions, SAS/bearer auth, AMQP/TLS and HTTPS, durable locks and settlement	256 KiB AMQP messages; no sessions, batch sends, transactions, scheduling or WebSockets
Functions	Linux B1 plans, registered containers, real .NET 10 isolated HTTP/Service Bus queue triggers, ARM settings and lifecycle	One instance, 64 KiB HTTP bodies, 10-second gateway exchange; no autoscaling, slots, ZIP/Kudu, other workers or triggers
Key Vault	Secret versions, attributes, soft delete, recover/purge	25 KB values, 500 versions/secret; no keys, certificates, backup/restore or policy enforcement
App Configuration	Key/label pairs, filters, ETags, locks, revisions; bearer/HMAC auth	10 KB key plus value, 1,000 revisions; no snapshots, replicas or continuation paging
SQL	PostgreSQL 18/MySQL 8.4; databases; native TLS/password connections; start/stop/restart	128 server records/provider, 64 databases/server; no HA/backup/upgrade/credential rotation
Cache	Managed Redis ARM shape with Valkey; encrypted native protocol, default database	No cluster mode, key rotation or Azure SKU performance simulation
Workspaces/components	ARM lifecycle, stable IDs, workspace binding, local-auth option, group cleanup	64 workspaces and 64 components; component workspace binding is immutable
Application ingestion	Request/dependency/log/exception/event/metric JSON envelopes; 100% sampling	1 MiB/256 envelopes, 8 KiB rows; no gzip or sampled envelopes
Logs Query	Six App* tables; comparisons, projection, sorting, take/count	4,096 query characters, 16 stages, 1,000 result rows; no joins, cross-workspace/batch queries or full KQL
Monitoring storage	Persistent workspace rows and configured retention	4,096 rows/workspace can shorten effective retention; no unlimited analytics store

Area	Supported contract	Important exclusions/limits
ARM Metrics	Requests/dependencies count, failed, duration; bounded aggregations and intervals	Up to 10 names, 30 days, 1,440 buckets; no arbitrary custom metrics or dimension splitting
Platform diagnostics	JSON logs, recent requests, bounded metrics, OTLP export and readiness	Native recent history 128; export queue 512; delivery is best effort

Event Hubs, Cosmos DB, Azure File Storage, Azure DNS resource-zone APIs, monitoring alerts and diagnostic settings are not supplied services. Local DNS resolves emulator names; it does not emulate Azure DNS management APIs. QHQ is a local management client, not an Azure Portal replica.

For every service, test real Azure's authorization and failure/scale behavior separately before deployment. The most useful local workflow is to keep your business logic and client calls unchanged while making the environment and its deliberately limited guarantees explicit.

For messaging walkthroughs, see Service Bus and Functions. Both exercises use the same endpoints, credentials and lifecycle operations explained in the chapters. A passing smoke test covers that workflow; it does not certify every API in the Microsoft service.

Related reading

- Quinzee Administrator’s Guide: Diagnose failures from the outside inward
- Quinzee Administrator’s Guide: Understand modules and their dependencies
- Quinzee Internals: Testing and verification
- Quinzee Internals: The compatibility application
- Scenario overview: Integration Testing Across Multiple Azure Services
- Scenario overview: Running Azure Integration Tests in CI/CD
- Scenario overview: Testing Application Upgrades and Regressions

24. Turn infrastructure into repeatable deployments

Your application needs more than executable code. It needs named storage, queues, identities and endpoints. Infrastructure as code records those choices in a file so that the next developer can build the same environment. In this workshop, Bicep is the source language and ARM JSON is the document sent to Quinzee. You can author either; you do not deploy a `.bicep` file directly.

Step 1: create a small ARM template by hand

Create `.local/book-iac` with `mkdir -p .local/book-iac`, then save this complete file as `.local/book-iac/storage.json`. The resource group is the deployment boundary; the template does not create the subscription or group itself.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "accountName": { "type": "string", "defaultValue": "bookhandstore" }
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2023-05-01",
      "name": "[parameters('accountName')]",
      "location": "[resourceGroup().location]",
      "kind": "StorageV2",
      "sku": { "name": "Standard_LRS" }
    }
  ],
  "outputs": {
    "accountName": { "type": "string", "value": "[parameters('accountName')]" }
  }
}
```

`type` identifies the provider and resource. `apiVersion` identifies the contract, not the installed software version. The expression in `name` reads an input; `resourceGroup().location` uses the group's location. `outputs` return useful results to the next step. Outputs are readable deployment metadata, so never put a password or client secret there.

Step 2: express the same intention in Bicep

Save `.local/book-iac/storage.bicep`:

```
targetScope = 'resourceGroup'
param accountName string = 'bookhandstore'
resource account 'Microsoft.Storage/storageAccounts@2023-05-01' = {
  name: accountName
  location: resourceGroup().location
  kind: 'StorageV2'
  sku: { name: 'Standard_LRS' }
}
output accountName string = account.name
```

Compile it on the host:

```
python3 scripts/deps/bicep.py build .local/book-iac/storage.bicep \
  --outfile .local/book-iac/storage.json
```

Open the resulting JSON. Find the same resource type, version, name expression and SKU. The compiler may add metadata and explicit dependency expressions; these are normal. Keep Bicep as the file you edit, and compile after every change. The repository helper verifies and caches the official compiler. It does not log in to Azure or send the template to Microsoft.

Step 3: deploy to a deliberately chosen environment

For this separate workshop, run `make qhq-run` on the host and leave it running. It creates a default local cloud and trusted certificate material. It is separate from the browser application's environment used earlier in the book. Open a second host terminal. The following command uses a new `book-iac` resource group; change the name if it already contains work you want to keep separate.

```
python3 examples/book-iac/deploy.py \
  --endpoint https://localhost:11443 \
  --ca .local/certificates/ca/quinzee-dev-root-ca.crt \
  --credentials .local/quinzee/credentials.json \
  --group book-iac --name hand-storage \
  --template .local/book-iac/storage.json \
  --outputs .local/book-iac/hand-storage-outputs.json
```

The deployment client below is printed in full so you can understand and adapt it. It reads a private credential file, obtains an ARM token, creates the group if absent, validates the template, submits an Incremental deployment, polls to completion and writes outputs with

mode 0600. It chooses a subscription only when exactly one exists; otherwise supply `--subscription <UUID>`. It rejects redirects and verifies the supplied CA. No secret appears in a command argument.

```
#!/usr/bin/env python3
"""Deploy a compiled ARM template to a chosen local HTTPS environment.
Credentials and secure parameters stay in private files; no Azure login required.
"""
import argparse
import json
import os
from pathlib import Path
import ssl
import time
import urllib.error
import urllib.parse
import urllib.request


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('--endpoint', required=True)
    parser.add_argument('--ca', type=Path, required=True)
    parser.add_argument('--credentials', type=Path, required=True)
    parser.add_argument('--subscription')
    parser.add_argument('--group', required=True)
    parser.add_argument('--location', default='westeurope')
    parser.add_argument('--name', default='book')
    parser.add_argument('--template', type=Path, required=True)
    parser.add_argument('--parameters', type=Path)
    parser.add_argument('--outputs', type=Path, required=True)
    parser.add_argument('--validate-only', action='store_true')
    args = parser.parse_args()
    origin = args.endpoint.rstrip('/')
    parsed = urllib.parse.urlsplit(origin)
    if parsed.scheme != 'https' or not parsed.hostname or parsed.path or parsed.query or
parsed.fragment or parsed.username:
        parser.error('endpoint must be an HTTPS origin without credentials or a path')
    for name in (args.group, args.name):
        if not name or any(c not in
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-_.()' for c in name):
            parser.error('group and deployment names must be simple ASCII identifiers')
    trust = ssl.create_default_context(cafile=str(args.ca))
    # Reject HTTP redirects rather than sending a token to another destination.
    class NoRedirect(urllib.request.HTTPRedirectHandler):
        def redirect_request(self, req, fp, code, msg, headers, newurl):
            return None
    client = urllib.request.build_opener(urllib.request.HTTPSHandler(context=trust),
NoRedirect())
    token = ''
    def request(method, path, body=None):
        headers = {'Authorization': 'Bearer ' + token} if token else {}
        if isinstance(body, dict):
            body = json.dumps(body).encode()
            headers['Content-Type'] = 'application/json'
        elif isinstance(body, str):
            body = body.encode()
            headers['Content-Type'] = 'application/x-www-form-urlencoded'
        with client.open(urllib.request.Request(origin + path, data=body, headers=headers,
```

```
method=method), timeout=30) as response:
            raw = response.read()
            return json.loads(raw) if raw else {}
    credentials = json.loads(args.credentials.read_text())
    token = request('POST', '/' + credentials['tenantId'] + '/oauth2/v2.0/token',
urllib.parse.urlencode({
        'grant_type': 'client_credentials', 'client_id': credentials['clientId'],
        'client_secret': credentials['clientSecret'], 'scope':
'https://management.azure.com/.default'}))['access_token']
    subscription = args.subscription
    if not subscription:
        subscriptions = request('GET', '/subscriptions?api-version=2023-05-01')['value']
        if len(subscriptions) != 1:
            parser.error('specify --subscription when the environment has multiple
subscriptions')
        subscription = subscriptions[0]['subscriptionId']
    import uuid
    subscription = str(uuid.UUID(subscription))
    group = '/subscriptions/' + subscription + '/resourceGroups/' + args.group
    try:
        request('GET', group + '?api-version=2023-05-01')
    except urllib.error.HTTPError as error:
        if error.code != 404: raise
        if args.validate_only:
            parser.error('create the group before validating, or deploy without --validate-only')
        request('PUT', group + '?api-version=2023-05-01', {'location': args.location})
    parameters = json.loads(args.parameters.read_text()) if args.parameters else {}
    parameters = parameters.get('parameters', parameters)
    body = {'properties': {'mode': 'Incremental', 'template':
json.loads(args.template.read_text()), 'parameters': parameters}}
    path = group + '/providers/Microsoft.Resources/deployments/' + args.name
    request('POST', path + '/validate?api-version=2022-09-01', body)
    print('Template validation succeeded.')
    if args.validate_only: return
    request('PUT', path + '?api-version=2022-09-01', body)
    deadline = time.monotonic() + 240
    while time.monotonic() < deadline:
        result = request('GET', path + '?api-version=2022-09-01')['properties']
        state = result['provisioningState']
        if state == 'Failed':
            raise RuntimeError('Deployment failed; inspect its operations in QHQ. Completed
resources remain.')
        if state == 'Succeeded':
            args.outputs.parent.mkdir(mode=0o700, parents=True, exist_ok=True)
            fd = os.open(args.outputs, os.O_WRONLY | os.O_CREAT | os.O_TRUNC | os.O_NOFOLLOW,
0o600)
            with os.fdopen(fd, 'w') as out:
                os.fchmod(out.fileno(), 0o600)
                json.dump(result.get('outputs', {}), out, indent=2)
                out.write('\n')
            print('Deployment succeeded. Outputs saved privately to', args.outputs)
            return
        time.sleep(1)
    raise TimeoutError('Deployment still running; inspect QHQ before retrying.')

if __name__ == '__main__':
    try: main()
    except urllib.error.HTTPError as error:
        raise SystemExit(f'HTTP {error.code}; check the endpoint, permissions and deployment
operations.') from None
```

The expected result is “Template validation succeeded” followed by “Deployment succeeded”. In QHQ open Resource Manager → Deployments, select `book-iac`, refresh, and inspect `hand-storage`. The storage page should contain `bookhandstore`. The outputs file contains the resolved account name.

Step 4: use parameters without baking secrets into source

A parameters file maps names to `value` objects. Save this as `.local/book-iac/parameters.json`:

```
{ "accountName": { "value": "booksecondstore" } }
```

Add `--parameters .local/book-iac/parameters.json` and use a new deployment name. This creates another named resource: changing a name does not rename the old one. For database examples, generate the password in a private file:

```
umask 077
python3 - <<'PYSECURE'
from pathlib import Path
import json, secrets
p = Path('.local/book-iac/database-parameters.json')
p.write_text(json.dumps({'password': {'value': secrets.token_urlsafe(24)}}))
p.chmod(0o600)
PYSECURE
```

Bicep `@secure()` becomes ARM `secureString`. Quinzee omits those input values from deployment history, but the local parameter file still contains a secret. Protect it, back it up only with private backups, and never put it in Git.

Step 5: reason about dependencies and failure

A Bicep `parent` makes a child resource's full name and dependency explicit. Using another resource's properties usually creates an implicit dependency; `dependsOn` expresses an additional ordering requirement. ARM JSON carries those relationships in `dependsOn`. A resource reference reads the provisioned result, such as an identity's generated principal ID. It cannot substitute for an unsupported cross-scope or external deployment.

Validation catches template syntax, unsupported expressions, API versions and dependency mistakes. It does not prove that an external runtime is reachable or that every service property will pass provider validation. A later failure can leave earlier resources created. Inspect the per-resource operation, fix the cause, and redeploy the same intended resource names. Incremental redeployment preserves resources omitted from the new file. Deleting deployment history also preserves resources. Deleting a resource group removes its resources and data.

Quinzee accepts resource-group, inline, Incremental templates. It rejects Complete mode, linked/nested templates, resource loops, resource conditions and arbitrary ARM functions. Use the examples' supported expressions rather than assuming any Bicep feature will work merely because the compiler accepts it.

Know what belongs outside an ARM template

ARM describes resources exposed by a resource provider. It does not configure every process or directory object in Quinzee. Keep these boundaries explicit:

Task	Interface to use
Storage, vaults, configuration, events, messaging, SQL/cache, monitoring, hosted apps	The Bicep/ARM service lessons that follow
User-assigned managed identities and Azure resource role assignments	Microsoft.ManagedIdentity and Microsoft.Authorization resources
App registrations, service principals, people, groups and API app roles	Graph-shaped directory APIs or QHQ, as practiced in the identity chapter
LDAP listener, dedicated directory PostgreSQL, certificates and helper processes	Operator configuration; these are prerequisites, not template resources
Local DNS records and module loading	Quinzee management controls; no Azure DNS zone provider is implied

In particular, an application registration is not a Microsoft.Web site. Creating a site gives you hosting; registering an application gives OAuth a client/API identity. The identity chapter connects them through a service principal and explicit permissions. Do not invent an ARM resource type to replace that step.

Step 6: work through the service lessons

The following chapters contain complete Bicep files plus the corresponding ARM resource objects. To hand-author ARM, place those objects in the complete `resources` array shown above and include the stated parameters. Compiling the Bicep creates the complete executable ARM document automatically, including parameters, dependencies and outputs. Each lesson states the expected result, runtime prerequisites and what you should deliberately change next.

Related reading

- Quinzee Administrator’s Guide: Control resources and infrastructure deployments
- Quinzee Internals: Deployments and native engines
- Scenario overview: Testing Bicep Deployments

- Scenario overview: Learning Infrastructure as Code with Bicep
- Scenario overview: Creating Reproducible Development Environments

25. Provision storage, secrets and configuration

Run the infrastructure workshop first. Each file below is complete and deploys at resource-group scope. Use a separate deployment name for each lesson so its operation history is easy to find.

Give each kind of application data a home

A storage account supplies three different services. Put file content in Blob Storage, work notifications in Queue Storage and keyed records in Table Storage. The child service name is `default`; it is not your container, queue or table name.

Step 1: save the Bicep

The repository copy is `examples/book-iac/storage.bicep`. To experiment, save this complete sample to `.local/book-iac/storage.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

resource account 'Microsoft.Storage/storageAccounts@2023-05-01' = {
  name: '${prefix}store'
  location: location
  kind: 'StorageV2'
  sku: { name: 'Standard_LRS' }
}
resource blobs 'Microsoft.Storage/storageAccounts/blobServices@2023-05-01' = {
  parent: account
  name: 'default'
}
resource documents 'Microsoft.Storage/storageAccounts/blobServices/containers@2023-05-01' = {
  parent: blobs
  name: 'documents'
  properties: { publicAccess: 'None' }
}
resource queues 'Microsoft.Storage/storageAccounts/queueServices@2023-05-01' = {
  parent: account
  name: 'default'
}
resource jobs 'Microsoft.Storage/storageAccounts/queueServices/queues@2023-05-01' = {
  parent: queues
  name: 'jobs'
}
resource tables 'Microsoft.Storage/storageAccounts/tableServices@2023-05-01' = {
  parent: account
```

```
  name: 'default'
}
resource orders 'Microsoft.Storage/storageAccounts/tableServices/tables@2023-05-01' = {
  parent: tables
  name: 'Orders'
}
output blobEndpoint string = account.properties.primaryEndpoints.blob
output queueEndpoint string = account.properties.primaryEndpoints.queue
output tableEndpoint string = account.properties.primaryEndpoints.table
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}store', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "kind": "StorageV2",
    "sku": {
      "name": "Standard_LRS"
    }
  },
  {
    "type": "Microsoft.Storage/storageAccounts/blobServices",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}', format('{0}store', parameters('prefix')), 'default')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts', format('{0}store',
parameters('prefix')))]"
    ]
  },
  {
    "type": "Microsoft.Storage/storageAccounts/blobServices/containers",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}/{2}', format('{0}store', parameters('prefix')), 'default',
'documents')]",
    "properties": {
      "publicAccess": "None"
    },
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts/blobServices', format('{0}store',
parameters('prefix')), 'default')]"
    ]
  },
  {
    "type": "Microsoft.Storage/storageAccounts/queueServices",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}', format('{0}store', parameters('prefix')), 'default')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts', format('{0}store',
parameters('prefix')))]"
    ]
```

```
  },
  {
    "type": "Microsoft.Storage/storageAccounts/queueServices/queues",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}/{2}', format('{0}store', parameters('prefix')), 'default',
'jobs')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts/queueServices', format('{0}store',
parameters('prefix')), 'default')]"
    ]
  },
  {
    "type": "Microsoft.Storage/storageAccounts/tableServices",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}', format('{0}store', parameters('prefix')), 'default')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts', format('{0}store',
parameters('prefix')))]"
    ]
  },
  {
    "type": "Microsoft.Storage/storageAccounts/tableServices/tables",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}/{2}', format('{0}store', parameters('prefix')), 'default',
'Orders')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts/tableServices', format('{0}store',
parameters('prefix')), 'default')]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/storage.bicep \
  --outfile .local/book-iac/storage.json
```

This file has defaults for every parameter. Run the deployment command from the workshop with `--name storage`, `--template .local/book-iac/storage.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

The `documents` container is private, `jobs` is an empty queue and `Orders` is an empty table. ARM creates their infrastructure; upload blobs, send messages and insert entities with the data-plane clients from the earlier chapters. Change a queue name and redeploy: the old queue remains until explicitly deleted.

Create a vault before storing a secret

Supply your local tenant ID in a parameters file. The vault is infrastructure; its secret values belong in the authenticated secret API after deployment. `enableRbacAuthorization` is accepted metadata in this local subset and does not implement Azure Key Vault data-plane RBAC.

Step 1: save the Bicep

The repository copy is `examples/book-iac/key-vault.bicep`. To experiment, save this complete sample to `.local/book-iac/key-vault.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

param tenantId string
resource vault 'Microsoft.KeyVault/vaults@2023-07-01' = {
  name: '${prefix}-vault'
  location: location
  properties: {
    tenantId: tenantId
    sku: { family: 'A', name: 'standard' }
    enableRbacAuthorization: true
    accessPolicies: []
  }
}
output endpoint string = vault.properties.vaultUri
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.KeyVault/vaults",
    "apiVersion": "2023-07-01",
    "name": "[format('{0}-vault', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "properties": {
      "tenantId": "[parameters('tenantId')]",
      "sku": {
        "family": "A",
        "name": "standard"
      },
      "enableRbacAuthorization": true,
      "accessPolicies": []
    }
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/key-vault.bicep \
  --outfile .local/book-iac/key-vault.json
```

Supply these required parameters: `tenantId`. Run the deployment command from the workshop with `--name key-vault`, `--template .local/book-iac/key-vault.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

QHQ should show the vault and its endpoint. Use the Key Vault chapter to store and retrieve a synthetic secret. Redeploy the same vault and confirm the secret version still exists. Never output a secret from your ARM template.

Separate deployable settings from source code

Create a named configuration store, then use the App Configuration client to write key-values and labels. Keep ordinary feature settings here and credentials in a secret store. The store declaration is separate from individual runtime key-values.

Step 1: save the Bicep

The repository copy is `examples/book-iac/app-configuration.bicep`. To experiment, save this complete sample to `.local/book-iac/app-configuration.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

resource settings 'Microsoft.AppConfiguration/configurationStores@2024-05-01' = {
  name: '${prefix}-config'
  location: location
  sku: { name: 'standard' }
}
output endpoint string = settings.properties.endpoint
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.AppConfiguration/configurationStores",
    "apiVersion": "2024-05-01",
    "name": "[format('{0}-config', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "sku": {
      "name": "standard"
    }
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/app-configuration.bicep \
  --outfile .local/book-iac/app-configuration.json
```

This file has defaults for every parameter. Run the deployment command from the workshop with `--name app-configuration`, `--template .local/book-iac/app-configuration.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

QHQ should show the store and endpoint. Write `app:color=blue`, redeploy, and read it again. The setting should survive because redeploying the store does not reset its data.

Related reading

- Quinzee Administrator’s Guide: Control resources and infrastructure deployments
- Quinzee Administrator’s Guide: Administer storage, secrets and configuration
- Quinzee Internals: Deployments and native engines
- Quinzee Internals: Module host and service state
- Scenario overview: Testing Bicep Deployments
- Scenario overview: Learning Infrastructure as Code with Bicep
- Scenario overview: Creating Reproducible Development Environments

26. Provision event delivery and durable messaging

Run the infrastructure workshop first. Each file below is complete and deploys at resource-group scope. Use a separate deployment name for each lesson so its operation history is easy to find.

Route changes into a queue

This complete lesson creates a storage account, a queue and a table, then connects both a storage system topic and a custom topic to that queue. A system topic observes Blob changes. A custom topic accepts application-published events. Both use explicit destination resource IDs.

Step 1: save the Bicep

The repository copy is `examples/book-iac/event-grid.bicep`. To experiment, save this complete sample to `.local/book-iac/event-grid.bicep`:

```
// Table Storage and Event Grid in one deployment: a storage account with a
// table and a queue, a system topic that sends Blob Created events to the
// queue, and a custom topic whose events go to the same queue.
targetScope = 'resourceGroup'

@minLength(3)
@maxLength(12)
param prefix string
param location string = resourceGroup().location

resource storage 'Microsoft.Storage/storageAccounts@2023-05-01' = {
  name: '${prefix}events'
  location: location
  kind: 'StorageV2'
  sku: { name: 'Standard_LRS' }
}
resource tables 'Microsoft.Storage/storageAccounts/tableServices@2023-05-01' = {
  parent: storage
  name: 'default'
}
resource orders 'Microsoft.Storage/storageAccounts/tableServices/tables@2023-05-01' = {
  parent: tables
  name: 'Orders'
}
resource queues 'Microsoft.Storage/storageAccounts/queueServices@2023-05-01' = {
  parent: storage
  name: 'default'
}
```

```
resource inbox 'Microsoft.Storage/storageAccounts/queueServices/queues@2023-05-01' = {
  parent: queues
  name: 'inbox'
}
resource blobEvents 'Microsoft.EventGrid/systemTopics@2025-02-15' = {
  name: '${prefix}-blobs'
  location: location
  properties: {
    source: storage.id
    topicType: 'Microsoft.Storage.StorageAccounts'
  }
}
resource blobToQueue 'Microsoft.EventGrid/systemTopics/eventSubscriptions@2025-02-15' = {
  parent: blobEvents
  name: 'blob-created'
  properties: {
    destination: {
      endpointType: 'StorageQueue'
      properties: {
        resourceId: storage.id
        queueName: inbox.name
      }
    }
    filter: { includedEventTypes: ['Microsoft.Storage.BlobCreated'] }
  }
}
resource orderTopic 'Microsoft.EventGrid/topics@2025-02-15' = {
  name: '${prefix}-orders'
  location: location
}
resource orderToQueue 'Microsoft.EventGrid/topics/eventSubscriptions@2025-02-15' = {
  parent: orderTopic
  name: 'orders-to-queue'
  properties: {
    destination: {
      endpointType: 'StorageQueue'
      properties: {
        resourceId: storage.id
        queueName: inbox.name
      }
    }
  }
}
output tableEndpoint string = storage.properties.primaryEndpoints.table
output queueEndpoint string = storage.properties.primaryEndpoints.queue
output blobEndpoint string = storage.properties.primaryEndpoints.blob
output topicEndpoint string = orderTopic.properties.endpoint
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}events', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "kind": "StorageV2",
    "sku": {
      "name": "Standard_LRS"
    }
  },
  {
    "type": "Microsoft.Storage/storageAccounts/tableServices",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}', format('{0}events', parameters('prefix')), 'default')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts', format('{0}events',
parameters('prefix')))]"
    ]
  },
  {
    "type": "Microsoft.Storage/storageAccounts/tableServices/tables",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}/{2}', format('{0}events', parameters('prefix')), 'default',
'Orders')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts/tableServices', format('{0}events',
parameters('prefix')), 'default')]"
    ]
  },
  {
    "type": "Microsoft.Storage/storageAccounts/queueServices",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}', format('{0}events', parameters('prefix')), 'default')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts', format('{0}events',
parameters('prefix')))]"
    ]
  },
  {
    "type": "Microsoft.Storage/storageAccounts/queueServices/queues",
    "apiVersion": "2023-05-01",
    "name": "[format('{0}/{1}/{2}', format('{0}events', parameters('prefix')), 'default',
'inbox')]",
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts/queueServices', format('{0}events',
parameters('prefix')), 'default')]"
    ]
  },
  {
    "type": "Microsoft.EventGrid/systemTopics",
    "apiVersion": "2025-02-15",
    "name": "[format('{0}-blobs', parameters('prefix'))]",
```

```
    "location": "[parameters('location')]",
    "properties": {
      "source": "[resourceId('Microsoft.Storage/storageAccounts', format('{0}events',
parameters('prefix')))]",
      "topicType": "Microsoft.Storage.StorageAccounts"
    },
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts', format('{0}events',
parameters('prefix')))]"
    ]
  },
  {
    "type": "Microsoft.EventGrid/systemTopics/eventSubscriptions",
    "apiVersion": "2025-02-15",
    "name": "[format('{0}/{1}', format('{0}-blobs', parameters('prefix')), 'blob-created')]",
    "properties": {
      "destination": {
        "endpointType": "StorageQueue",
        "properties": {
          "resourceId": "[resourceId('Microsoft.Storage/storageAccounts', format('{0}events',
parameters('prefix')))]",
          "queueName": "inbox"
        }
      },
      "filter": {
        "includedEventTypes": [
          "Microsoft.Storage.BlobCreated"
        ]
      }
    },
    "dependsOn": [
      "[resourceId('Microsoft.EventGrid/systemTopics', format('{0}-blobs',
parameters('prefix')))]",
      "[resourceId('Microsoft.Storage/storageAccounts/queueServices/queues', format('{0}events',
parameters('prefix')), 'default', 'inbox')]",
      "[resourceId('Microsoft.Storage/storageAccounts', format('{0}events',
parameters('prefix')))]"
    ]
  },
  {
    "type": "Microsoft.EventGrid/topics",
    "apiVersion": "2025-02-15",
    "name": "[format('{0}-orders', parameters('prefix'))]",
    "location": "[parameters('location')]"
  },
  {
    "type": "Microsoft.EventGrid/topics/eventSubscriptions",
    "apiVersion": "2025-02-15",
    "name": "[format('{0}/{1}', format('{0}-orders', parameters('prefix')), 'orders-to-queue')]",
    "properties": {
      "destination": {
        "endpointType": "StorageQueue",
        "properties": {
          "resourceId": "[resourceId('Microsoft.Storage/storageAccounts', format('{0}events',
parameters('prefix')))]",
          "queueName": "inbox"
        }
      }
    },
    "dependsOn": [
      "[resourceId('Microsoft.Storage/storageAccounts/queueServices/queues', format('{0}events',
```

```
parameters('prefix')), 'default', 'inbox')]",
      "[resourceId('Microsoft.EventGrid/topics', format('{0}-orders', parameters('prefix')))]",
      "[resourceId('Microsoft.Storage/storageAccounts', format('{0}events',
parameters('prefix')))]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/event-grid.bicep \
  --outfile .local/book-iac/event-grid.json
```

Supply these required parameters: `prefix`. Run the deployment command from the workshop with `--name event-grid`, `--template .local/book-iac/event-grid.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

After deployment, upload a Blob and inspect the `inbox` queue. Publish an event to the custom topic and expect another notification. Event delivery retries may duplicate notifications, so make the consumer idempotent. The destination queue is a service resource; its name and account ID must match the subscription.

Give independent consumers their own subscriptions

Supply a unique `namespaceName`, for example `booklab-bus`. Queues distribute competing work. A topic copies each message to every existing subscription, so the `first` and `second` consumers each receive their own copy. The AMQP listener must be enabled before using the Service Bus SDK.

Step 1: save the Bicep

The repository copy is `examples/book-iac/service-bus.bicep`. To experiment, save this complete sample to `.local/book-iac/service-bus.bicep`:

```
// Durable messaging: a queue and a topic with two independent subscriptions.
// Quinzee Cloud - proprietary and confidential. All rights reserved.
targetScope = 'resourceGroup'

@minLength(6)
@maxLength(50)
param namespaceName string
param location string = resourceGroup().location

resource bus 'Microsoft.ServiceBus/namespaces@2024-01-01' = {
  name: namespaceName
  location: location
  sku: { name: 'Standard', tier: 'Standard' }
}
resource orders 'Microsoft.ServiceBus/namespaces/queues@2024-01-01' = {
```

```
  parent: bus
  name: 'orders'
  properties: {
    lockDuration: 'PT1M'
    maxDeliveryCount: 10
  }
}
resource events 'Microsoft.ServiceBus/namespaces/topics@2024-01-01' = {
  parent: bus
  name: 'events'
}
resource first 'Microsoft.ServiceBus/namespaces/topics/subscriptions@2024-01-01' = {
  parent: events
  name: 'first'
}
resource second 'Microsoft.ServiceBus/namespaces/topics/subscriptions@2024-01-01' = {
  parent: events
  name: 'second'
}
output namespaceId string = bus.id
output queueId string = orders.id
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.ServiceBus/namespaces",
    "apiVersion": "2024-01-01",
    "name": "[parameters('namespaceName')]",
    "location": "[parameters('location')]",
    "sku": {
      "name": "Standard",
      "tier": "Standard"
    }
  },
  {
    "type": "Microsoft.ServiceBus/namespaces/queues",
    "apiVersion": "2024-01-01",
    "name": "[format('{0}/{1}', parameters('namespaceName'), 'orders')]",
    "properties": {
      "lockDuration": "PT1M",
      "maxDeliveryCount": 10
    },
    "dependsOn": [
      "[resourceId('Microsoft.ServiceBus/namespaces', parameters('namespaceName'))]"
    ]
  },
  {
    "type": "Microsoft.ServiceBus/namespaces/topics",
    "apiVersion": "2024-01-01",
    "name": "[format('{0}/{1}', parameters('namespaceName'), 'events')]",
    "dependsOn": [
      "[resourceId('Microsoft.ServiceBus/namespaces', parameters('namespaceName'))]"
```

```
    ]
  },
  {
    "type": "Microsoft.ServiceBus/namespaces/topics/subscriptions",
    "apiVersion": "2024-01-01",
    "name": "[format('{0}/{1}/{2}', parameters('namespaceName'), 'events', 'first')]",
    "dependsOn": [
      "[resourceId('Microsoft.ServiceBus/namespaces/topics', parameters('namespaceName'),
'events')]"
    ]
  },
  {
    "type": "Microsoft.ServiceBus/namespaces/topics/subscriptions",
    "apiVersion": "2024-01-01",
    "name": "[format('{0}/{1}/{2}', parameters('namespaceName'), 'events', 'second')]",
    "dependsOn": [
      "[resourceId('Microsoft.ServiceBus/namespaces/topics', parameters('namespaceName'),
'events')]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/service-bus.bicep \
  --outfile .local/book-iac/service-bus.json
```

Supply these required parameters: `namespaceName`. Run the deployment command from the workshop with `--name service-bus`, `--template .local/book-iac/service-bus.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

QHQ lists the namespace, `orders` queue, `events` topic and two subscriptions. Run the Service Bus chapter to send and settle messages. Change `maxDeliveryCount` and redeploy without deleting the queue; existing messages remain. Use supported Standard tier and AMQP/TLS, not WebSockets.

Related reading

- Quinzee Administrator’s Guide: Control resources and infrastructure deployments
- Quinzee Administrator’s Guide: Keep events and messages moving
- Quinzee Internals: Deployments and native engines
- Quinzee Internals: Service Bus and Functions execution
- Scenario overview: Testing Bicep Deployments
- Scenario overview: Learning Infrastructure as Code with Bicep
- Scenario overview: Experimenting with Cloud Architecture

27. Provision databases and a cache with real engines

Run the infrastructure workshop first. Each file below is complete and deploys at resource-group scope. Use a separate deployment name for each lesson so its operation history is easy to find.

Provision PostgreSQL and an application database

Run `make engine-images` once, then leave `make engines-run` running against the same cloud data directory and Docker network. Supply `password` using the private parameter-file procedure. The provider owns lifecycle; SQL clients connect to a separate real PostgreSQL process.

Step 1: save the Bicep

The repository copy is `examples/book-iac/postgresql.bicep`. To experiment, save this complete sample to `.local/book-iac/postgresql.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

param administrator string = 'appadmin'
@secure()
param password string
resource server 'Microsoft.DBforPostgreSQL/flexibleServers@2024-08-01' = {
  name: '${prefix}-postgresql'
  location: location
  properties: {
    administratorLogin: administrator
    administratorLoginPassword: password
    version: '18'
  }
}
resource database 'Microsoft.DBforPostgreSQL/flexibleServers/databases@2024-08-01' = {
  parent: server
  name: 'application'
  properties: {}
}
output hostname string = server.properties.fullyQualifiedDomainName
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.DBforPostgreSQL/flexibleServers",
    "apiVersion": "2024-08-01",
    "name": "[format('{0}-postgresql', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "properties": {
      "administratorLogin": "[parameters('administrator')]",
      "administratorLoginPassword": "[parameters('password')]",
      "version": "18"
    }
  },
  {
    "type": "Microsoft.DBforPostgreSQL/flexibleServers/databases",
    "apiVersion": "2024-08-01",
    "name": "[format('{0}/{1}', format('{0}-postgresql', parameters('prefix')), 'application')]",
    "properties": {},
    "dependsOn": [
      "[resourceId('Microsoft.DBforPostgreSQL/flexibleServers', format('{0}-postgresql',
parameters('prefix')))]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/postgresql.bicep \
  --outfile .local/book-iac/postgresql.json
```

Supply these required parameters: `password`. Run the deployment command from the workshop with `--name postgresql`, `--template .local/book-iac/postgresql.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

Wait for the deployment to succeed and server readiness before connecting on TLS port 5432. The database should be named `application`. Create a table with Npgsql, stop/start the server, and verify the row survives. Deleting the server removes its managed data volume.

Use the same lifecycle with MySQL

Use a distinct server name and the MySQL 8.4 version accepted by this provider. The administrator account is local database authentication, not an Entra user. Supply a private password parameter and keep the engine supervisor running.

Step 1: save the Bicep

The repository copy is `examples/book-iac/mysql.bicep`. To experiment, save this complete sample to `.local/book-iac/mysql.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

param administrator string = 'appadmin'
@secure()
param password string
resource server 'Microsoft.DBforMySQL/flexibleServers@2023-12-30' = {
  name: '${prefix}-mysql'
  location: location
  properties: {
    administratorLogin: administrator
    administratorLoginPassword: password
    version: '8.4'
  }
}
resource database 'Microsoft.DBforMySQL/flexibleServers/databases@2023-12-30' = {
  parent: server
  name: 'application'
  properties: {}
}
output hostname string = server.properties.fullyQualifiedDomainName
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.DBforMySQL/flexibleServers",
    "apiVersion": "2023-12-30",
    "name": "[format('{0}-mysql', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "properties": {
      "administratorLogin": "[parameters('administrator')]",
      "administratorLoginPassword": "[parameters('password')]",
      "version": "8.4"
    }
  },
  {
    "type": "Microsoft.DBforMySQL/flexibleServers/databases",
    "apiVersion": "2023-12-30",
    "name": "[format('{0}/{1}', format('{0}-mysql', parameters('prefix')), 'application')]",
    "properties": {},
    "dependsOn": [
      "[resourceId('Microsoft.DBforMySQL/flexibleServers', format('{0}-mysql',
parameters('prefix')))]"
```

```
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/mysql.bicep \
  --outfile .local/book-iac/mysql.json
```

Supply these required parameters: `password`. Run the deployment command from the workshop with `--name mysql`, `--template .local/book-iac/mysql.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

Connect with MySqlConnector on verified TLS port 3306 and select `application`. A created ARM record is not proof that the engine is ready. A missing helper leaves provisioning pending until it fails; repair the helper before redeploying.

Provision an encrypted disposable cache

The ARM shape is Managed Redis; the local process is Valkey. This lesson creates one `default` database with encrypted clients, no clustering and no eviction. It requires the engine supervisor just like SQL. Its access key is obtained through an authenticated operation, not an ARM output.

Step 1: save the Bicep

The repository copy is `examples/book-iac/redis.bicep`. To experiment, save this complete sample to `.local/book-iac/redis.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

resource cache 'Microsoft.Cache/redisEnterprise@2025-04-01' = {
  name: '${prefix}-cache'
  location: location
  sku: { name: 'Balanced_B0' }
  properties: { minimumTlsVersion: '1.2' }
}
resource database 'Microsoft.Cache/redisEnterprise/databases@2025-04-01' = {
  parent: cache
  name: 'default'
  properties: {
    clientProtocol: 'Encrypted'
    clusteringPolicy: 'NoCluster'
    evictionPolicy: 'NoEviction'
```

```
  }
}
output hostname string = cache.properties.hostName
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.Cache/redisEnterprise",
    "apiVersion": "2025-04-01",
    "name": "[format('{0}-cache', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "sku": {
      "name": "Balanced_B0"
    },
    "properties": {
      "minimumTlsVersion": "1.2"
    }
  },
  {
    "type": "Microsoft.Cache/redisEnterprise/databases",
    "apiVersion": "2025-04-01",
    "name": "[format('{0}/{1}', format('{0}-cache', parameters('prefix')), 'default')]",
    "properties": {
      "clientProtocol": "Encrypted",
      "clusteringPolicy": "NoCluster",
      "evictionPolicy": "NoEviction"
    },
    "dependsOn": [
      "[resourceId('Microsoft.Cache/redisEnterprise', format('{0}-cache',
parameters('prefix')))]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/redis.bicep \
  --outfile .local/book-iac/redis.json
```

This file has defaults for every parameter. Run the deployment command from the workshop with `--name redis`, `--template .local/book-iac/redis.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

Use the Redis chapter to retrieve the key privately and connect on TLS port 10000. Write and read a value, then decide whether the application can reconstruct it. The fixed container limits do not simulate the performance promised by an Azure SKU.

Related reading

- Quinzee Administrator’s Guide: Control resources and infrastructure deployments
- Quinzee Administrator’s Guide: Run PostgreSQL, MySQL and Valkey safely
- Quinzee Internals: Deployments and native engines
- Quinzee Internals: From desired state to a real container
- Scenario overview: Testing Bicep Deployments
- Scenario overview: Learning Infrastructure as Code with Bicep
- Scenario overview: Creating Reproducible Development Environments

28. Deploy application hosts and their identities

Run the infrastructure workshop first. Each file below is complete and deploys at resource-group scope. Use a separate deployment name for each lesson so its operation history is easy to find.

Create a workload identity with a scoped role

Generate one UUID for `assignmentName` and retain it in the private parameters file for future redeployments. The template creates a user-assigned identity and gives it Reader at the deployment resource-group scope. The generated principal ID identifies the tenant object; the client ID identifies the OAuth client.

Step 1: save the Bicep

The repository copy is `examples/book-iac/identity-rbac.bicep`. To experiment, save this complete sample to `.local/book-iac/identity-rbac.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

param assignmentName string
resource identity 'Microsoft.ManagedIdentity/userAssignedIdentities@2023-01-31' = {
  name: '${prefix}-reader'
  location: location
}
resource reader 'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: assignmentName
  properties: {
    principalId: identity.properties.principalId
    principalType: 'ServicePrincipal'
    roleDefinitionId:
'/subscriptions/${subscription().subscriptionId}/providers/Microsoft.Authorization/roleDefinition
s/acdd72a7-3385-48ef-bd42-f606fba81ae7'
  }
}
output clientId string = identity.properties.clientId
output principalId string = identity.properties.principalId
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.ManagedIdentity/userAssignedIdentities",
    "apiVersion": "2023-01-31",
    "name": "[format('{0}-reader', parameters('prefix'))]",
    "location": "[parameters('location')]"
  },
  {
    "type": "Microsoft.Authorization/roleAssignments",
    "apiVersion": "2022-04-01",
    "name": "[parameters('assignmentName')]",
    "properties": {
      "principalId": "[reference(resourceId('Microsoft.ManagedIdentity/userAssignedIdentities',
format('{0}-reader', parameters('prefix'))), '2023-01-31').principalId]",
      "principalType": "ServicePrincipal",
      "roleDefinitionId": "
[format('/subscriptions/{0}/providers/Microsoft.Authorization/roleDefinitions/acdd72a7-3385-48ef-
bd42-f606fba81ae7', subscription().subscriptionId)]"
    },
    "dependsOn": [
      "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', format('{0}-reader',
parameters('prefix')))]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/identity-rbac.bicep \
  --outfile .local/book-iac/identity-rbac.json
```

Supply these required parameters: `assignmentName`. Run the deployment command from the workshop with `--name identity-rbac`, `--template .local/book-iac/identity-rbac.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

QHQ shows the identity and role assignment. It has no password and cannot obtain a token from an arbitrary laptop: attach it to a supported hosted app. Reader grants resource reads, not message send/receive and not directory administration. Use one stable assignment UUID per intended principal/role/scope tuple.

Run a web application behind HTTPS

Build the minimal web image in the App Service walkthrough, register `quinzee-book-web:local` with the hosting helper, and keep the helper running. The App Service walkthrough explains those commands. The plan and app use the same group and location. The system identity is created with the site but receives no resource permissions automatically.

Step 1: save the Bicep

The repository copy is `examples/book-iac/app-service.bicep`. To experiment, save this complete sample to `.local/book-iac/app-service.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

param image string = 'quinzee-book-web:local'
resource plan 'Microsoft.Web/serverfarms@2024-11-01' = {
  name: '${prefix}-web-plan'
  location: location
  kind: 'linux'
  sku: { name: 'B1', tier: 'Basic', capacity: 1 }
  properties: { reserved: true }
}
resource web 'Microsoft.Web/sites@2024-11-01' = {
  name: '${prefix}-web'
  location: location
  kind: 'app,linux,container'
  identity: { type: 'SystemAssigned' }
  properties: {
    serverFarmId: plan.id
    httpsOnly: true
    reserved: true
    siteConfig: {
      linuxFxVersion: 'DOCKER|${image}'
      alwaysOn: true
      appSettings: [{ name: 'Greeting', value: 'Hello from infrastructure as code' }]
    }
  }
}
output hostname string = web.properties.defaultHostName
output principalId string = web.identity.principalId
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.Web/serverfarms",
    "apiVersion": "2024-11-01",
    "name": "[format('{0}-web-plan', parameters('prefix'))]",
```

```
    "location": "[parameters('location')]",
    "kind": "linux",
    "sku": {
      "name": "B1",
      "tier": "Basic",
      "capacity": 1
    },
    "properties": {
      "reserved": true
    }
  },
  {
    "type": "Microsoft.Web/sites",
    "apiVersion": "2024-11-01",
    "name": "[format('{0}-web', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "kind": "app,linux,container",
    "identity": {
      "type": "SystemAssigned"
    },
    "properties": {
      "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', format('{0}-web-plan',
parameters('prefix')))]",
      "httpsOnly": true,
      "reserved": true,
      "siteConfig": {
        "linuxFxVersion": "[format('DOCKER|{0}', parameters('image'))]",
        "alwaysOn": true,
        "appSettings": [
          {
            "name": "Greeting",
            "value": "Hello from infrastructure as code"
          }
        ]
      }
    },
    "dependsOn": [
      "[resourceId('Microsoft.Web/serverfarms', format('{0}-web-plan', parameters('prefix')))]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/app-service.bicep \
  --outfile .local/book-iac/app-service.json
```

This file has defaults for every parameter. Run the deployment command from the workshop with `--name app-service`, `--template .local/book-iac/app-service.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

The deployment completes when the helper verifies the app. Open the returned hostname through the lab DNS and trusted CA. The included web app returns its Greeting at `/`. Change Greeting, redeploy, and observe the replacement container serving the new value.

Deploy HTTP and queue-triggered code

Build `make functions-example`, register the image and keep the helper running. Deploy the Service Bus lesson first, then obtain its connection string privately as shown in the messaging chapter. Supply `serviceBusConnection` as a secure parameter; use a unique `appName`.

Step 1: save the Bicep

The repository copy is `examples/book-iac/functions.bicep`. To experiment, save this complete sample to `.local/book-iac/functions.bicep`:

```
param appName string = 'quinzee-functions'
param image string = 'quinzee-functions-example:local'
@secure()
param serviceBusConnection string
param location string = resourceGroup().location
resource plan 'Microsoft.Web/serverfarms@2024-11-01' = {
  name: '${appName}-plan'
  location: location
  kind: 'linux'
  sku: { name: 'B1', tier: 'Basic', capacity: 1 }
  properties: { reserved: true }
}
resource app 'Microsoft.Web/sites@2024-11-01' = {
  name: appName
  location: location
  kind: 'functionapp,linux,container'
  properties: {
    serverFarmId: plan.id
    httpsOnly: true
    reserved: true
    siteConfig: {
      linuxFxVersion: 'DOCKER|${image}'
      alwaysOn: true
      appSettings: [
        { name: 'Greeting', value: 'from Bicep' }
        { name: 'ServiceBusConnection', value: serviceBusConnection }
      ]
    }
  }
}
output hostname string = app.properties.defaultHostName
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.Web/serverfarms",
    "apiVersion": "2024-11-01",
    "name": "[format('{0}-plan', parameters('appName'))]",
    "location": "[parameters('location')]",
    "kind": "linux",
    "sku": {
      "name": "B1",
      "tier": "Basic",
      "capacity": 1
    },
    "properties": {
      "reserved": true
    }
  },
  {
    "type": "Microsoft.Web/sites",
    "apiVersion": "2024-11-01",
    "name": "[parameters('appName')]",
    "location": "[parameters('location')]",
    "kind": "functionapp,linux,container",
    "properties": {
      "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', format('{0}-plan',
parameters('appName')))]",
      "httpsOnly": true,
      "reserved": true,
      "siteConfig": {
        "linuxFxVersion": "[format('DOCKER|{0}', parameters('image'))]",
        "alwaysOn": true,
        "appSettings": [
          {
            "name": "Greeting",
            "value": "from Bicep"
          },
          {
            "name": "ServiceBusConnection",
            "value": "[parameters('serviceBusConnection')]"
          }
        ]
      }
    },
    "dependsOn": [
      "[resourceId('Microsoft.Web/serverfarms', format('{0}-plan', parameters('appName')))]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/functions.bicep \
  --outfile .local/book-iac/functions.json
```

Supply these required parameters: `serviceBusConnection`. Run the deployment command from the workshop with `--name functions`, `--template .local/book-iac/functions.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

The returned host serves `/api/echo`; the Functions host enforces its function key. Sending to `orders` runs the Service Bus trigger. The real application image must contain the referenced binding. ARM provisions infrastructure; it does not compile C# or install missing extensions.

Replace the messaging secret with a managed identity

Use a fresh app name and the existing namespace name. Retain a stable `roleAssignmentName` UUID. This template grants Receiver to the user-assigned identity at resource-group scope, attaches it, and supplies the binding's identity settings. For a narrower grant, use the explicit namespace/entity role-assignment API instead.

Step 1: save the Bicep

The repository copy is `examples/book-iac/functions-identity.bicep`. To experiment, save this complete sample to `.local/book-iac/functions-identity.bicep`:

```
// Secret-free queue processing with a user-assigned identity and scoped role.
// Quinzee Cloud - proprietary and confidential. All rights reserved.
param appName string = 'quinzee-identity-functions'
param namespaceName string
param image string = 'quinzee-functions-example:local'
param roleAssignmentName string = '5486a231-5229-46aa-913d-3fd405af9958'
param location string = resourceGroup().location
resource identity 'Microsoft.ManagedIdentity/userAssignedIdentities@2023-01-31' = {
  name: '${appName}-identity'
  location: location
}
resource receiver 'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: roleAssignmentName
  properties: {
    principalId: identity.properties.principalId
    principalType: 'ServicePrincipal'
    roleDefinitionId:
'/subscriptions/${subscription().subscriptionId}/providers/Microsoft.Authorization/roleDefinition
s/4f6d3b9b-027b-4f4c-9142-0e5a2a2247e0'
  }
}
resource plan 'Microsoft.Web/serverfarms@2024-11-01' = {
  name: '${appName}-plan'
  location: location
```

```
  kind: 'linux'
  sku: { name: 'B1', tier: 'Basic', capacity: 1 }
  properties: { reserved: true }
}
resource app 'Microsoft.Web/sites@2024-11-01' = {
  name: appName
  location: location
  kind: 'functionapp,linux,container'
  identity: {
    type: 'UserAssigned'
    userAssignedIdentities: { '${identity.id}': {} }
  }
  dependsOn: [receiver]
  properties: {
    serverFarmId: plan.id
    httpsOnly: true
    reserved: true
    siteConfig: {
      linuxFxVersion: 'DOCKER|${image}'
      alwaysOn: true
      appSettings: [
        { name: 'Greeting', value: 'from Bicep' }
        { name: 'ServiceBusConnection__fullyQualifiedNamespace', value:
'${namespaceName}.servicebus.quinzee.cloud' }
        { name: 'ServiceBusConnection__credential', value: 'managedidentity' }
        { name: 'ServiceBusConnection__clientId', value: identity.properties.clientId }
      ]
    }
  }
}
output hostname string = app.properties.defaultHostName
output principalId string = identity.properties.principalId
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.ManagedIdentity/userAssignedIdentities",
    "apiVersion": "2023-01-31",
    "name": "[format('{0}-identity', parameters('appName'))]",
    "location": "[parameters('location')]"
  },
  {
    "type": "Microsoft.Authorization/roleAssignments",
    "apiVersion": "2022-04-01",
    "name": "[parameters('roleAssignmentName')]",
    "properties": {
      "principalId": "[reference(resourceId('Microsoft.ManagedIdentity/userAssignedIdentities',
format('{0}-identity', parameters('appName'))), '2023-01-31').principalId]",
      "principalType": "ServicePrincipal",
      "roleDefinitionId": "
[format('/subscriptions/{0}/providers/Microsoft.Authorization/roleDefinitions/4f6d3b9b-027b-4f4c-
9142-0e5a2a2247e0', subscription().subscriptionId)]"
```

```
    },
    "dependsOn": [
      "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', format('{0}-identity',
parameters('appName')))]"
    ]
  },
  {
    "type": "Microsoft.Web/serverfarms",
    "apiVersion": "2024-11-01",
    "name": "[format('{0}-plan', parameters('appName'))]",
    "location": "[parameters('location')]",
    "kind": "linux",
    "sku": {
      "name": "B1",
      "tier": "Basic",
      "capacity": 1
    },
    "properties": {
      "reserved": true
    }
  },
  {
    "type": "Microsoft.Web/sites",
    "apiVersion": "2024-11-01",
    "name": "[parameters('appName')]",
    "location": "[parameters('location')]",
    "kind": "functionapp,linux,container",
    "identity": {
      "type": "UserAssigned",
      "userAssignedIdentities": {
        "[format('{0}', resourceId('Microsoft.ManagedIdentity/userAssignedIdentities',
format('{0}-identity', parameters('appName'))))]": {}
      }
    },
    "properties": {
      "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', format('{0}-plan',
parameters('appName')))]",
      "httpsOnly": true,
      "reserved": true,
      "siteConfig": {
        "linuxFxVersion": "[format('DOCKER|{0}', parameters('image'))]",
        "alwaysOn": true,
        "appSettings": [
          {
            "name": "Greeting",
            "value": "from Bicep"
          },
          {
            "name": "ServiceBusConnection__fullyQualifiedNamespace",
            "value": "[format('{0}.servicebus.quinzee.cloud', parameters('namespaceName'))]"
          },
          {
            "name": "ServiceBusConnection__credential",
            "value": "managedidentity"
          },
          {
            "name": "ServiceBusConnection__clientId",
            "value": "[reference(resourceId('Microsoft.ManagedIdentity/userAssignedIdentities',
format('{0}-identity', parameters('appName'))), '2023-01-31').clientId]"
          }
        ]
```

```
      }
    },
    "dependsOn": [
      "[resourceId('Microsoft.ManagedIdentity/userAssignedIdentities', format('{0}-identity',
parameters('appName')))]",
      "[resourceId('Microsoft.Web/serverfarms', format('{0}-plan', parameters('appName')))]",
      "[resourceId('Microsoft.Authorization/roleAssignments', parameters('roleAssignmentName'))]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/functions-identity.bicep \
  --outfile .local/book-iac/functions-identity.json
```

Supply these required parameters: `namespaceName`. Run the deployment command from the workshop with `--name functions-identity`, `--template .local/book-iac/functions-identity.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

With SAS disabled on the namespace, the trigger still receives work using a Service Bus audience token. Removing the receiver assignment prevents further receives. The role is an Azure resource permission; app roles on an Enterprise application do not grant Service Bus access.

Related reading

- Quinzee Administrator’s Guide: Operate Functions and App Service workloads
- Quinzee Administrator’s Guide: Control resources and infrastructure deployments
- Quinzee Internals: Service Bus and Functions execution
- Quinzee Internals: From desired state to a real container
- Scenario overview: Developing an Azure Functions Application Locally
- Scenario overview: Testing Bicep Deployments
- Scenario overview: Learning Infrastructure as Code with Bicep

29. Deploy monitoring with the application

Run the infrastructure workshop first. Each file below is complete and deploys at resource-group scope. Use a separate deployment name for each lesson so its operation history is easy to find.

Connect an Application Insights component to a workspace

A workspace stores telemetry; a component identifies its producer and points to that workspace. The resource reference makes the workspace a prerequisite. `DisableLocalAuth: true` requires a supported bearer token for ingestion rather than relying only on an instrumentation key.

Step 1: save the Bicep

The repository copy is `examples/book-iac/monitoring.bicep`. To experiment, save this complete sample to `.local/book-iac/monitoring.bicep`:

```
targetScope = 'resourceGroup'
param prefix string = 'booklab'
param location string = resourceGroup().location

resource workspace 'Microsoft.OperationalInsights/workspaces@2023-09-01' = {
  name: '${prefix}-logs'
  location: location
  properties: {
    sku: { name: 'PerGB2018' }
    retentionInDays: 30
  }
}
resource insights 'Microsoft.Insights/components@2020-02-02' = {
  name: '${prefix}-insights'
  location: location
  kind: 'web'
  properties: {
    Application_Type: 'web'
    WorkspaceResourceId: workspace.id
    DisableLocalAuth: true
  }
}
output workspaceId string = workspace.properties.customerId
output connectionString string = insights.properties.ConnectionString
```

Step 2: inspect the ARM equivalent

The compiler emits these resource objects. The surrounding template also contains the parameters and outputs declared above. Expressions in square brackets are evaluated by the deployment engine, not by your shell.

```
[
  {
    "type": "Microsoft.OperationalInsights/workspaces",
    "apiVersion": "2023-09-01",
    "name": "[format('{0}-logs', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "properties": {
      "sku": {
        "name": "PerGB2018"
      },
      "retentionInDays": 30
    }
  },
  {
    "type": "Microsoft.Insights/components",
    "apiVersion": "2020-02-02",
    "name": "[format('{0}-insights', parameters('prefix'))]",
    "location": "[parameters('location')]",
    "kind": "web",
    "properties": {
      "Application_Type": "web",
      "WorkspaceResourceId": "[resourceId('Microsoft.OperationalInsights/workspaces',
format('{0}-logs', parameters('prefix')))]",
      "DisableLocalAuth": true
    },
    "dependsOn": [
      "[resourceId('Microsoft.OperationalInsights/workspaces', format('{0}-logs',
parameters('prefix')))]"
    ]
  }
]
```

Step 3: compile, deploy and check

```
python3 scripts/deps/bicep.py build .local/book-iac/monitoring.bicep \
  --outfile .local/book-iac/monitoring.json
```

This file has defaults for every parameter. Run the deployment command from the workshop with `--name monitoring`, `--template .local/book-iac/monitoring.json` and a distinct outputs filename. Add `--parameters` when supplying inputs. Select the same deployment in QHQ and wait for Succeeded.

Use the returned connection string with the monitoring chapter's exporter and send a trace. Query `AppTraces | take 10` in QHQ. Changing the workspace binding in place is unsupported: plan a new component when moving to another workspace. Retention is also bounded by the local 4,096-row cap.

Related reading

- Quinzee Administrator’s Guide: Control resources and infrastructure deployments
- Quinzee Administrator’s Guide: Observe the platform and the application separately
- Quinzee Internals: Deployments and native engines
- Quinzee Internals: HTTP library and asynchronous execution
- Scenario overview: Testing Bicep Deployments
- Scenario overview: Debugging Cloud Applications Locally
- Scenario overview: Learning Infrastructure as Code with Bicep

PUBLICATION NOTES

About this edition

This book is generated from the Quinzee Markdown manuals. Screenshots show the real application and QHQ in an isolated local environment; names, identifiers and ports vary between runs.

The PDF includes DejaVu font subsets. Their copyright and permission notice follows.

Fonts are (c) Bitstream (see below). DejaVu changes are in public domain. Glyphs imported from Arev fonts are (c) Tavmjong Bah (see below)

Bitstream Vera Fonts Copyright ----------

Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

Arev Fonts Copyright ----------

Copyright (c) 2006 by Tavmjong Bah. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the modifications to the Bitstream Vera Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Tavmjong Bah" or the word "Arev".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Tavmjong Bah Arev" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL TAVMJONG BAH BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the name of Tavmjong Bah shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from Tavmjong Bah. For further information, contact: tavmjong @ free . fr.

TeX Gyre DJV Math ---------- Fonts are (c) Bitstream (see below). DejaVu changes are in public domain.

Math extensions done by B. Jackowski, P. Strzelczyk and P. Pianowski (on behalf of TeX users groups) are in public domain.

Letters imported from Euler Fraktur from AMSfonts are (c) American Mathematical Society (see below). Bitstream Vera Fonts Copyright Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license (“Fonts”) and associated documentation files (the “Font Software”), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words “Bitstream” or the word “Vera”.

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the “Bitstream Vera” names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE. Except as contained in this notice, the names of GNOME, the GNOME Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the GNOME Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

AMSFonts (v. 2.2) copyright

The PostScript Type 1 implementation of the AMSFonts produced by and previously distributed by Blue Sky Research and Y&Y, Inc. are now freely available for general use. This has been accomplished through the cooperation of a consortium of scientific publishers with Blue Sky Research and Y&Y. Members of this consortium include:

Elsevier Science IBM Corporation Society for Industrial and Applied Mathematics (SIAM) Springer-Verlag American Mathematical Society (AMS)

In order to assure the authenticity of these fonts, copyright will be held by the American Mathematical Society. This is not meant to restrict in any way the legitimate use of the fonts, such as (but not limited to) electronic distribution of documents containing these fonts, inclusion of these fonts into other public domain or commercial font collections or computer applications, use of the outline data to create derivative fonts and/or faces, etc. However, the AMS does require that the AMS copyright notice be removed from any derivative versions of the fonts which have been altered in any way. In addition, to ensure the fidelity of TeX documents using Computer Modern fonts, Professor Donald Knuth, creator of the Computer Modern faces, has requested that any alterations which yield different font metrics be given a different name.

Id