**Project shape**

ClaimsFunctions/

Program.cs

host.json

local.settings.json

Models/Claim.cs

Data/ClaimRepository.cs

Functions/ClaimsHttp.cs

host.json

{

"version": "2.0",

"extensions": { "http": { "routePrefix": "api" } }

}

local.settings.json (local only)

{

"IsEncrypted": false,

"Values": {

"FUNCTIONS\_WORKER\_RUNTIME": "dotnet-isolated",

"AzureWebJobsStorage": "UseDevelopmentStorage=true"

},

"Cosmos": {

"AccountEndpoint": "https://<your-account>.documents.azure.com:443/",

"Key": "<your-key>",

"Database": "ClaimsDb",

"Container": "Claims"

}

}

For quick testing, create the **database** ClaimsDb and **container** Claims with **partition key /id** (simple for this demo).  
In real apps, prefer a domain key like /tenantId or /policyId.

**Program.cs (DI, Cosmos client)**

using Microsoft.Azure.Functions.Worker;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Azure.Cosmos;

using ClaimsFunctions.Data;

var host = new HostBuilder()

.ConfigureAppConfiguration(cfg =>

{

cfg.AddJsonFile("appsettings.json", optional: true)

.AddJsonFile("local.settings.json", optional: true)

.AddEnvironmentVariables();

})

.ConfigureServices((ctx, services) =>

{

var c = ctx.Configuration.GetSection("Cosmos");

var endpoint = c["AccountEndpoint"]!;

var key = c["Key"]!;

var database = c["Database"]!;

var container = c["Container"]!;

// CosmosClient singleton

var client = new CosmosClient(endpoint, key, new CosmosClientOptions

{

ConnectionMode = ConnectionMode.Gateway

});

services.AddSingleton(client);

// Container singleton

services.AddSingleton(sp =>

sp.GetRequiredService<CosmosClient>().GetContainer(database, container));

services.AddSingleton<IClaimRepository, ClaimRepository>();

})

.ConfigureFunctionsWorkerDefaults()

.Build();

await host.RunAsync();

**Model**

Models/Claim.cs

namespace ClaimsFunctions.Models;

public class Claim

{

// Using id as both document id and partition key for demo

public string id { get; set; } = Guid.NewGuid().ToString();

public string PolicyNumber { get; set; } = default!;

public int Amount { get; set; }

public string Status { get; set; } = "Open"; // Open|Approved|Rejected|Paid

public DateTime CreatedAtUtc { get; set; } = DateTime.UtcNow;

// optional tenant key (good real-world partition key)

// public string TenantId { get; set; } = default!;

}

**Repository with Cosmos SDK + LINQ**

Data/ClaimRepository.cs

using ClaimsFunctions.Models;

using Microsoft.Azure.Cosmos;

using System.Net;

using System.Linq;

using System.Text.Json;

namespace ClaimsFunctions.Data;

public interface IClaimRepository

{

Task<Claim> CreateAsync(Claim claim, CancellationToken ct = default);

Task<Claim?> GetAsync(string id, CancellationToken ct = default);

Task<Claim?> UpdateAsync(string id, Action<Claim> apply, CancellationToken ct = default);

Task<bool> DeleteAsync(string id, CancellationToken ct = default);

// LINQ query samples

Task<IReadOnlyList<Claim>> ListOpenAsync(int max = 50, CancellationToken ct = default);

Task<(IReadOnlyList<Claim> items, string? continuation)> SearchAsync(

string? policyNumber = null,

string? status = null,

int pageSize = 20,

string? continuation = null,

CancellationToken ct = default);

// Aggregates via LINQ (translated)

Task<int> TotalForPolicyAsync(string policyNumber, CancellationToken ct = default);

}

public class ClaimRepository : IClaimRepository

{

private readonly Container \_container;

public ClaimRepository(Container container) => \_container = container;

public async Task<Claim> CreateAsync(Claim claim, CancellationToken ct = default)

{

var resp = await \_container.CreateItemAsync(claim, new PartitionKey(claim.id), cancellationToken: ct);

return resp.Resource;

}

public async Task<Claim?> GetAsync(string id, CancellationToken ct = default)

{

try

{

var resp = await \_container.ReadItemAsync<Claim>(id, new PartitionKey(id), cancellationToken: ct);

return resp.Resource;

}

catch (CosmosException ex) when (ex.StatusCode == HttpStatusCode.NotFound)

{

return null;

}

}

public async Task<Claim?> UpdateAsync(string id, Action<Claim> apply, CancellationToken ct = default)

{

// Read → apply mutation → Replace

var existing = await GetAsync(id, ct);

if (existing is null) return null;

apply(existing);

var resp = await \_container.ReplaceItemAsync(existing, existing.id, new PartitionKey(existing.id), cancellationToken: ct);

return resp.Resource;

}

public async Task<bool> DeleteAsync(string id, CancellationToken ct = default)

{

try

{

await \_container.DeleteItemAsync<Claim>(id, new PartitionKey(id), cancellationToken: ct);

return true;

}

catch (CosmosException ex) when (ex.StatusCode == HttpStatusCode.NotFound)

{

return false;

}

}

// ------- LINQ queries --------

// Cosmos LINQ starts with GetItemLinqQueryable<T>() which returns IQueryable<T>.

// It's DEFERRED until you enumerate via FeedIterator.

public async Task<IReadOnlyList<Claim>> ListOpenAsync(int max = 50, CancellationToken ct = default)

{

var queryable =

\_container.GetItemLinqQueryable<Claim>(allowSynchronousQueryExecution: false)

.Where(c => c.Status == "Open")

.OrderByDescending(c => c.CreatedAtUtc)

.Take(max);

var it = queryable.ToFeedIterator(); // IMMEDIATE when iterated

var results = new List<Claim>();

while (it.HasMoreResults && results.Count < max)

{

var page = await it.ReadNextAsync(ct);

results.AddRange(page);

}

return results;

}

public async Task<(IReadOnlyList<Claim> items, string? continuation)> SearchAsync(

string? policyNumber = null,

string? status = null,

int pageSize = 20,

string? continuation = null,

CancellationToken ct = default)

{

var q = \_container.GetItemLinqQueryable<Claim>(requestOptions: new QueryRequestOptions

{

MaxItemCount = pageSize

});

if (!string.IsNullOrWhiteSpace(policyNumber))

q = q.Where(c => c.PolicyNumber == policyNumber);

if (!string.IsNullOrWhiteSpace(status))

q = q.Where(c => c.Status == status);

q = q.OrderByDescending(c => c.CreatedAtUtc);

// Deferred until ToFeedIterator(...)

var it = q.ToFeedIterator(continuationToken: continuation);

if (!it.HasMoreResults)

return (Array.Empty<Claim>(), null);

var page = await it.ReadNextAsync(ct); // Immediate execution here

return (page.Resource.ToList(), page.ContinuationToken);

}

public async Task<int> TotalForPolicyAsync(string policyNumber, CancellationToken ct = default)

{

// Sum via LINQ projection (translated to server-side)

var q = \_container.GetItemLinqQueryable<Claim>()

.Where(c => c.PolicyNumber == policyNumber)

.Select(c => c.Amount);

var it = q.ToFeedIterator();

var total = 0;

while (it.HasMoreResults)

{

var page = await it.ReadNextAsync(ct);

// Aggregation must be client-side when paging,

// but the filtering/projection are server-translated.

total += page.Sum();

}

return total;

}

}

**Deferred vs Immediate in Cosmos LINQ**

* Building the IQueryable<T> (Where, Select, OrderBy, Take) is **deferred**.
* It becomes **immediate** when you call **ToFeedIterator()** and then iterate by calling **ReadNextAsync()**.
* There is **no** ToListAsync() directly on Cosmos LINQ; the iterator/pages are the execution boundary.

**HTTP Functions (CRUD + query)**

Functions/ClaimsHttp.cs

using System.Net;

using System.Text.Json;

using ClaimsFunctions.Data;

using ClaimsFunctions.Models;

using Microsoft.Azure.Functions.Worker;

using Microsoft.Azure.Functions.Worker.Http;

namespace ClaimsFunctions.Functions;

public class ClaimsHttp(IClaimRepository repo)

{

private static readonly JsonSerializerOptions json = new(JsonSerializerDefaults.Web);

[Function("CreateClaim")]

public async Task<HttpResponseData> Create(

[HttpTrigger(AuthorizationLevel.Function, "post", Route = "claims")] HttpRequestData req)

{

var claim = await JsonSerializer.DeserializeAsync<Claim>(req.Body, json);

if (claim is null)

{

var bad = req.CreateResponse(HttpStatusCode.BadRequest);

await bad.WriteStringAsync("Invalid body");

return bad;

}

var created = await repo.CreateAsync(claim);

var res = req.CreateResponse(HttpStatusCode.Created);

await res.WriteAsJsonAsync(created, json);

return res;

}

[Function("GetClaim")]

public async Task<HttpResponseData> Get(

[HttpTrigger(AuthorizationLevel.Function, "get", Route = "claims/{id}")] HttpRequestData req, string id)

{

var claim = await repo.GetAsync(id);

if (claim is null) return req.CreateResponse(HttpStatusCode.NotFound);

var res = req.CreateResponse(HttpStatusCode.OK);

await res.WriteAsJsonAsync(claim, json);

return res;

}

[Function("UpdateClaimStatus")]

public async Task<HttpResponseData> UpdateStatus(

[HttpTrigger(AuthorizationLevel.Function, "patch", Route = "claims/{id}/status/{next}")]

HttpRequestData req, string id, string next)

{

var updated = await repo.UpdateAsync(id, c => c.Status = next);

if (updated is null) return req.CreateResponse(HttpStatusCode.NotFound);

var res = req.CreateResponse(HttpStatusCode.OK);

await res.WriteAsJsonAsync(updated, json);

return res;

}

[Function("DeleteClaim")]

public async Task<HttpResponseData> Delete(

[HttpTrigger(AuthorizationLevel.Function, "delete", Route = "claims/{id}")]

HttpRequestData req, string id)

{

var ok = await repo.DeleteAsync(id);

return req.CreateResponse(ok ? HttpStatusCode.NoContent : HttpStatusCode.NotFound);

}

// LINQ list (Open claims, top N)

[Function("ListOpenClaims")]

public async Task<HttpResponseData> ListOpen(

[HttpTrigger(AuthorizationLevel.Function, "get", Route = "claims/open")]

HttpRequestData req)

{

var items = await repo.ListOpenAsync(50);

var res = req.CreateResponse(HttpStatusCode.OK);

await res.WriteAsJsonAsync(items, json);

return res;

}

// LINQ search with paging

[Function("SearchClaims")]

public async Task<HttpResponseData> Search(

[HttpTrigger(AuthorizationLevel.Function, "get", Route = "claims/search")]

HttpRequestData req)

{

var q = System.Web.HttpUtility.ParseQueryString(req.Url.Query);

var policy = q["policy"];

var status = q["status"];

var pageSize = int.TryParse(q["pageSize"], out var ps) ? Math.Clamp(ps, 1, 100) : 20;

var token = q["continuation"];

var (items, continuation) = await repo.SearchAsync(policy, status, pageSize, token);

var res = req.CreateResponse(HttpStatusCode.OK);

await res.WriteAsJsonAsync(new { items, continuation }, json);

return res;

}

// Aggregate: total claim amount per policy

[Function("TotalForPolicy")]

public async Task<HttpResponseData> TotalForPolicy(

[HttpTrigger(AuthorizationLevel.Function, "get", Route = "claims/total/{policy}")]

HttpRequestData req, string policy)

{

var total = await repo.TotalForPolicyAsync(policy);

var res = req.CreateResponse(HttpStatusCode.OK);

await res.WriteAsJsonAsync(new { policy, total }, json);

return res;

}

}

**Quick test endpoints (local)**

* **Create**: POST http://localhost:7071/api/claims
* { "policyNumber": "POL-123", "amount": 5000, "status": "Open" }
* **Get**: GET http://localhost:7071/api/claims/{id}
* **Update status**: PATCH http://localhost:7071/api/claims/{id}/status/Approved
* **Delete**: DELETE http://localhost:7071/api/claims/{id}
* **List open**: GET http://localhost:7071/api/claims/open
* **Search (paged)**:  
  GET http://localhost:7071/api/claims/search?policy=POL-123&status=Open&pageSize=5  
  → returns { items, continuation }  
  Next page: GET ...&continuation=<token>
* **Aggregate**: GET http://localhost:7071/api/claims/total/POL-123

**LINQ: deferred vs immediate (at a glance)**

* **Deferred**: building the query with .Where, .Select, .OrderBy, .Take on IQueryable<T> returned by container.GetItemLinqQueryable<T>().
* **Immediate** (executes against Cosmos): when you call **.ToFeedIterator()** and then **ReadNextAsync()** (page-by-page). Each ReadNextAsync() makes a query request and yields a page of results plus a continuation token.

**Useful notes**

* Use QueryRequestOptions.MaxItemCount to steer page size.
* For **server-side filtering/projection**, stick to Cosmos-supported LINQ operators. Unsupported constructs fall back to client-side only after fetching (inefficient) or may fail translation.
* For **count/sum** across many pages, aggregate on the client as you iterate pages.

**Cosmos LINQ vs LINQ to Objects / SQL / XML**

* **LINQ to Objects**: operates in-memory over IEnumerable<T>. Fully featured, but no server translation.
* **LINQ to SQL/EF**: rich translation to SQL; large operator surface. Cosmos LINQ is **similar in spirit** but has a **smaller supported set** tailored to its query engine (e.g., equality, comparisons, StartsWith, logical ops, OrderBy, Select, limited Any/All, etc.). Complex method calls, custom functions, and certain DateTime ops may not translate.
* **LINQ to XML**: operates on XDocument/XElement trees. No server translation.
* **Cosmos LINQ**: query is converted to **Cosmos SQL** (text like SELECT c.\* FROM c WHERE c.Status = 'Open' ORDER BY c.CreatedAtUtc DESC). If translation isn’t possible, it errors or degrades to client filtering (avoid that for scale).

**Tip:** when a query gets tricky, you can always drop to **GetItemQueryIterator<T>(new QueryDefinition("SELECT ..."))** and write native Cosmos SQL.