

Introduction

How It Works

Quickie operates by injecting essential middleware and services into your application pipeline. It provides base classes that you can implement to create APIs and backend services with minimal effort and focus solely on your logic. With Quickie, all you need to do is define your required classes and you're ready to go — no endless boilerplate, no hassle. Quickie is perfect for building tracer bullet APIs, prototypes, or any sorta proof-of-concept projects. Whether you're experimenting with ideas or rapidly iterating, Quickie keeps things simple and fun, without feeling like you're trapped in enterprise-grade complexity.

Here's a high-level overview of its workflow:

1. **Service Registration:** Quickie sets up services for rate limiting, idempotency, and other configurable options through `QuickieConfig()` and complete the setup with `AddQuickie()` which resolve all dependencies.
2. **Idempotency made simple:** Ensure API calls yield consistent results when retried, with support for custom idempotency providers.
3. **Built-In Rate Limiting:** Protect your APIs from overuse with an easy-to-setup rate-limiting mechanism.
4. **Customizable Options:** You can fine-tune configurations, such as permitting limits for rate limiting or choosing a custom idempotency provider (Redis, or any external databases).
5. **Custom Error Messages:** Fine-tune responses with the option to show user-friendly error messages or show exact exception message.

Getting Started

To start using Quickie in your .NET project, follow these steps:

Adding Quickie to Your Project

1. Install Quickie via NuGet:

```
dotnet add package Quickie
```

2. Add Quickie configuration in your `Program.cs` file:

Default Configuration:



```
builder.Services.QuickieConfig();  
app.AddQuickie();
```

Customized Configuration

```
builder.Services.QuickieConfig(options => {  
    options.ShowCustomErrorMessage = false;  
    options.RateLimitingConfiguration = new RateLimitConfiguration  
    {  
        DisableRateLimiting = false  
    };  
    options.IdempotencyConfiguration = new IdempotentConfiguration  
    {  
        Enable = true  
    };  
});  
  
app.AddQuickie();
```

Simple demo it is, but you can go nuts with configuration. More [here](#)

Explanation of Key Features

- **Rate Limiting:** Manage API usage with customizable rate-limiting policies. To know more click [here](#) .
- **Idempotency:** Ensure consistent results for repeated API calls, with support for custom providers. To know more click [here](#) .

Configuration

Quickie offers flexible configuration options to customize its behavior according to your application needs. This guide covers all available configuration options and their usage.

Basic Setup

To configure Quickie in your application, you need to add it to both your service collection and middleware pipeline:

```
// In Program.cs
builder.Services.QuickieConfig();

// In middleware pipeline
app.AddQuickie();
```

Above configuration will enable default options. Default options include:

- Rate limiting
 - Accept 1 request (**PermitLimit**) every 6 seconds (**FromSeconds**) or simply understanding: allow only 10 API request every 1 minute.
 - Policy name as Quickie-RI-Policy.
- Idempotency is disabled.
- Custom error message is enabled. API client will see custom generic message instead of the exception messages.

Idempotency Configuration

Idempotency prevents duplicate API request. You can make your POST call idempotent. To enable idempotency:

```
builder.Services.QuickieConfig(options => {
    options.IdempotencyConfiguration = new IdempotentConfiguration {
        Enable = true, // enable idempotency support
        IdempotencyLifespan = TimeSpan.FromHours(2), // set key lifespan (default:
1 hour)
        RunBackgroundServiceEveryHour = 2, // cleanup interval (default:
1 hour)
        Provider = new CustomIdempotencyProvider() // optional: custom provider
    };
});
```

Property	Type	Default	Description
Enable	bool	false	Enables/disables idempotency support
Provider	IImpotencyProvider	InMemoryIdempotencyProvider	Custom provider for idempotency handling. By default, its handle by Quickie using in-memory configuration more
IdempotencyLifespan	TimeSpan	1 hour	Duration for which idempotency keys remain valid
RunBackgroundServiceEveryHour	int	1	Interval (in hours) for cleanup service (0-24)

View [doc](#)

Default Idempotency provider

Quickie uses in-memory option to provide idempotency.

Example:

```
builder.Services.QuickieConfig(options => {
    options.IdempotencyConfiguration = new IdempotentConfiguration {
        Enable = true
    };
});
```

By default idempotency is disabled, above configuration enables it. Now, every POST request requires **X-Idempotency-Key** header. Header's value will be saved in-mem. Every 1 hour background service will reset this in-memory pool. You can customize idempotency lifespan and interval period to run background service.

Example:

```
builder.Services.QuickieConfig(options => {
    options.IdempotencyConfiguration = new IdempotentConfiguration {
        Enable = true,
        RunBackgroundServiceEveryHour = 2,
        IdempotencyLifespan = TimeSpan.FromHours(5)
    };
});
```

Above config will run background service every 2 hours and sets idempotency key's lifespan to 5 hours. It means each key's lifespan is 5 hours and any key whose lifespan has completed will be removed by background service from in-memory pool in every 2 hours.

Custom Idempotency provider

Instead of Quickie's default idempotency provider, you can have your own custom provider using Redis, MongoDB, SQL databases, or any other storage solution. Implement your provider with **IIIdempotencyProvider**. Here is an example using [Redis](#):

Run Redis server

```
docker run -d --name redis -p 6379:6379 redis
```

Check process status

```
docker ps
```

You are good to go if Redis is running.

```
using Quickie.Configuration.Idempotency;
using StackExchange.Redis;

namespace sample.idempotent.redis.Configuration;

public class RedisIdempotencyProvider : IIIdempotencyProvider
{
    private readonly IDatabase _redisDatabase;
```

```

public RedisIdempotencyProvider(IConnectionMultiplexer redisConnection)
{
    _redisDatabase = redisConnection.GetDatabase();
}

public async ValueTask<bool> ExistsAsync(string key)
{
    var check = await _redisDatabase.KeyExistsAsync(key);
    return check;
}

public async ValueTask MarkAsync(string key)
{
    var lifespan = TimeSpan.FromHours(1); // holds this key only for 1 hr
    await _redisDatabase.StringSetAsync(key, DateTime.UtcNow.ToString("o"), lifespan);
}

public async ValueTask RemoveExpiredKeys()
{
    // redis automatically removes keys when their lifespan.
    await ValueTask.CompletedTask;
}
}

```

Now, configuration look something like this:

```

builder.Services.QuickieConfig(options => {
    options.IdempotencyConfiguration = new IdempotentConfiguration {
        Enable = true,
        RunBackgroundServiceEveryHour = 1,
        Provider = new RedisIdempotencyProvider(redisConnection)
    };
});

```

The background service will run once every hour to clean up expired idempotency keys from Redis. Although Redis automatically removes expired keys, the background service can be used for additional tasks like manually cleaning up or performing other maintenance if needed.

IdempotencyLifespan is only for quickie's default idempotency provider. If you're using a custom provider like redis or other options, you'll need to implement the lifespan logic for the keys yourself as demonstrated in the example above. **RedisIdempotencyProvider** will be automatically resolved by Quickie as well.

Rate Limiting Configuration

Quickie provides rate limiting options based on IP address by default. To customize:

- If you want to disable:

```
builder.Services.QuickieConfig(options => {
    options.RateLimitingConfiguration = new RateLimitConfiguration {
        DisableRateLimiting = true // this disables rate limiting
    };
});
```

Customize:

- Scenario: Allow 100 request in duration of 60 seconds.

```
builder.Services.QuickieConfig(options => {
    options.RateLimitingConfiguration = new RateLimitConfiguration {
        PermitLimit = 100,
        FromSeconds = 60
    };
});
```

Property	Type	Default	Description
<code>DisableRateLimiting</code>	bool	false	Disables/enables rate limiting
<code>PolicyName</code>	string	"Quickie-RI-Policy"	Name of the rate limit policy. This cannot be changed.
<code>PermitLimit</code>	int	1	Number of allowed requests per window
<code>FromSeconds</code>	int	6	Time window duration in seconds

Internally, Quickie implements rate limiting using the [Fixed Window algorithm](#), which tracks requests based on IP addresses. This allows you to limit the number of requests a user can make within a specific time window. For example, the configuration in the code above sets a limit on the number of requests that can be made from a given IP address within a defined time window. If the limit is exceeded, Quickie will return a 429 Too Many Requests status code. [src](#)

Error Message Configuration

Configure how you want to show your error messages: show actual error or generic message.

- Show generic message

```
builder.Services.QuickieConfig(options => {  
    options.ShowCustomErrorMessage = true;  
});
```

Above configuration will show custom generic error message for all sorta errors.

Example:

If database related occur while creating an entity, instead of showing actual exception message, it will show: **Data not created.**

- Show actual error

```
builder.Services.QuickieConfig(options => {  
    options.ShowCustomErrorMessage = false;  
});
```

Above configuration will show actual exception message for all sorta errors.

Example:

If database related occur while creating an entity, it will show: **The specified key 'user_id' was not ...**

Configuration src doc [here](#)

Build By Example

Here is a simple Web API with controllers for a **Todo App** using Quickie.

Step 1: Create a new Web API Project

```
dotnet new webapi -n todo.apis --use-controllers
```

Step 2: Install Quickie

Install Quickie from NuGet:

```
dotnet add package Quickie
```

Step 3: Configure Quickie in `Program.cs`

In your `Program.cs`, configure Quickie as follows:

```
builder.Services.QuickieConfig(options => {
    options.ShowCustomErrorMessage = true;
    options.RateLimitingConfiguration = new RateLimitConfiguration
    {
        DisableRateLimiting = false
    };
    options.IdempotencyConfiguration = new IdempotentConfiguration
    {
        Enable = true
    };
});

app.AddQuickie();
```

Explanation:

- For this project, we are enabling **Idempotency**.
- Rate limiting is **enabled by default**, but we chose to explicitly configure it here.
- Custom error messages are shown when exceptions occur.

Step 4: Create a Dto and Entity

```
public record TodoDto(int Id, string Title, string Description) : CrudDto;
```

```

public class TodoEntity : CrudEntity
{
    [Key]
    public int Id { get; set; }
    public required string Title { get; set; }
    public required string Description { get; set; }
    public required DateTime CreatedDate { get; set; }
}

```

Step 5: Create a Controller

For our Todo app, we need CRUD operations:

- **C** -> Create Todo
- **R** -> Read Todo
- **U** -> Update Todo
- **D** -> Delete Todo

Here is the `TodoController`:

```

public class TodoController(ITodoService requestHandler) : CrudController<TodoDto,
ITodoService, int>(requestHandler);

```

Step 6: Request handler (Service layer) Setup Todo Service

```

public interface ITodoService : ICrudRequestHandler<TodoDto, int>;

public class TodoService(ICrudDataHandler<TodoEntity, int> dataHandler) :
    CrudRequestHandler<TodoDto, TodoEntity, ITodoRepo, int>(dataHandler), ITodoService
{
    protected override TodoEntity MapToEntity(TodoDto request)
    {
        var d = new TodoEntity()
        {
            Id = request.Id,
            Title = request?.Title,
            Description = request?.Description,
            CreatedDate = DateTime.Now
        };
        return d;
    }
}

```

```
protected override TodoDto MapToDto(TodoEntity request)
{
    var d = request is not null ? new TodoDto(request.Id, request?.Title + " id:" +
request?.Id, request?.Description) : default;
    return d;
}
}
```

Note: Mapping must be done manually. You can use any mapping library or write your own logic.

Todo Repository (Data handler)

```
public interface ITodoRepo : ICrudDataHandler<TodoEntity, int>;

public class TodoRepo(ApplicationDbContext dbContext) : CrudDataHandler<TodoEntity,
ApplicationDbContext, int>(dbContext), ITodoRepo;
```

Step 7: Configure Database Context

```
public class ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
: DbContext(options)
{
    public DbSet<TodoEntity> TodoEntity { get; set; }
}
```

Step 8: Register Services in DI

Register the services in `Program.cs`:

```
builder.Services.AddScoped<ITodoService, TodoService>();
builder.Services.AddScoped<ITodoRepo, TodoRepo>();
builder.Services.AddScoped<ICrudDataHandler<TodoEntity, int>, TodoRepo>();
```

Step 9: Making Requests with Idempotency

Since **Idempotency** is enabled, you must provide an **X-Idempotency-Key** with each request (POST calls). For duplicate requests, the API will respond with a **409 Conflict** status.

Example Request:

```
curl -X 'POST' \
'http://localhost:5162/api/Todo' \
```

```
-H 'accept: application/json' \  
-H 'X-Idempotency-Key: c311bef0-9953-45b1-bb73-70169e1a3de5' \  
-H 'Content-Type: application/json' \  
-d '{  
  "id": 0,  
  "title": "work",  
  "description": "feature 0"  
}'
```

Not Just CRUD

Quickie is versatile and supports scenarios beyond CRUD operations:

- **CRUD for Collection:** Bulk create, read, update, and delete operations are supported, making it easy to handle multiple entities in a single request.
- **Readonly:** For entities where only read operations are required.
- **Write-only:** For scenarios where entities can only be written to, but not read.
- **Edit-only:** For entities that support updates but not creation or deletion.
- **Readonly Collections:** You can define collections where only bulk read operations are required, and no modifications are allowed.

You can choose the appropriate functionality based on your application's needs.

API is Ready!

That's it! Your fully functional Web API with:

- CRUD functionality
- Built-in **Idempotency** (to prevent duplicate requests)
- Built-in **Rate Limiting** (enabled by default)

Things to Consider

- **DTOs** should be record types.
- **Entities** are class (reference types).

More examples [here](#) .

Build By Example

Here is a simple Web API of a **Todo App** using Quickie for **Minimal API** project.

Step 1: Create a new Web API Project

```
dotnet new webapi -n todo.apis
```

Step 2: Install Quickie

Install Quickie from NuGet:

```
dotnet add package Quickie
```

Step 3: Configure Quickie in `Program.cs`

In your `Program.cs`, configure Quickie as follows:

```
builder.Services.QuickieConfig();

app.AddQuickie();
```

Explanation:

- For this project, default configuration will be used as shown above.

More on configuration [here](#)

Step 4: Create a Dto and Entity

```
public record TodoCrudableDto(int Id, string Title, string Description) : CrudDto;
```

```
public class TodoCrudableEntity : CrudEntity
{
    [Key]
    public int Id { get; set; }
    public required string Title { get; set; }
    public required string Description { get; set; }
    public required DateTime CreatedDate { get; set; }
}
```

Step 5: Create your apis

For our Todo app, we need CRUD operations:

- **C** -> Create Todo
- **R** -> Read Todo
- **U** -> Update Todo
- **D** -> Delete Todo

```
app.AddCrudEndpoints<TodoCrudableDto, IToDoService, int>("/api/todos");
```

Step 6: Request handler (Service layer) Setup

Todo Service

```
public interface IToDoService : ICrudRequestHandler<TodoCrudableDto, int>;
```

```
public class ToDoService(IToDoRepo dataHandler) : CrudRequestHandler<TodoCrudableDto,
    TodoCrudableEntity, IToDoRepo, int>(dataHandler), IToDoService
{
    protected override TodoCrudableEntity MapToEntity(TodoCrudableDto request)
    {
        var d = new TodoCrudableEntity()
            { Id = request.Id, Title = request?.Title, Description = request?.Description,
              CreatedDate = DateTime.Now };
        return d;
    }

    protected override TodoCrudableDto MapToDto(TodoCrudableEntity request)
    {
        var d = request is not null ? new TodoCrudableDto(request.Id, request?.Title + "
id:" + request?.Id, request?.Description) : default;
        return d;
    }
}
```

Note: Mapping must be done manually. You can use any mapping library or write your own logic.

Todo Repository (Data handler)

```
public interface IToDoRepo : ICrudDataHandler<TodoCrudableEntity, int>;
```

```
public class TodoRepo(ApplicationDbContext dbContext) : CrudDataHandler<TodoCrudableEntity,
ApplicationDbContext, int>(dbContext), ITodoRepo;
```

Step 7: Configure Database Context

```
public class ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
: DbContext(options)
{
    public DbSet<TodoCrudableEntity> TodoCrudableEntity { get; set; }
}
```

Step 8: Register Services in DI

DI registration:

```
builder.Services.AddScoped<ITodoService, TodoService>();
builder.Services.AddScoped<ITodoRepo, TodoRepo>();
builder.Services.AddScoped<ICrudDataHandler<TodoEntity, int>, TodoRepo>();
```

Step 9: Making Request

Since **Idempotency** is disabled, **X-Idempotency-Key** header is not required.

Example Request:

Create Todo

POST http://localhost:5220/api/todos
Content-Type: application/json

```
{
  "title": "Test Todo yayay",
  "description": "Testing CRUD operations"
}
```

Get Todo by ID

GET http://localhost:5220/api/todos/3

Update Todo

PUT http://localhost:5220/api/todos/1
Content-Type: application/json

```
{
  "title": "Updated Todo",
```

```
"description": "Updated description"
}

### Delete Todo
DELETE http://localhost:5220/api/todos/1
```

Your Minimal API is now ready with:

- CRUD functionality
- Built-in Idempotency (to prevent duplicate requests which is disabled for default configuration)
- Built-in Rate Limiting (enabled by default)

Not Just CRUD

Quickie is versatile and supports scenarios beyond CRUD operations:

- **CRUD for Collection:** Bulk create, read, update, and delete operations are supported, making it easy to handle multiple entities in a single request.
- **Readonly:** For entities where only read operations are required.

```
// Get only (Single Entity)
app.AddReadOnlyEndpoints<TodoDto, ISingleTodoReqHandler, string>("/api/get/todos");

// Get only (Collection)
app.AddReadOnlyCollectionEndpoints<TodoDto, IReadTodoReqHandler, DataFilterRequest,
string>("/api/getcollection/todos");
```

- **Write-only:** For scenarios where entities can only be written to, but not read.

```
// Write-only API
app.AddWriteOnlyEndpoints<WriteOnlyTodoDto, IWriteOnlyTodoReqHandler>("/api/create/todos");
```

- **Edit-only:** For entities that support updates but not creation or deletion.

```
// Edit-only API
app.AddEditOnlyEndpoints<PastTodo_EditOnlyDto, IEditOnlyTodoReqHandler,
string>("/api/editonly/todos");
```

- **Readonly Collections:** You can define collections where only bulk read operations are required, and no modifications are allowed.

You can choose the appropriate functionality based on your application's needs.

That's it! Your fully functional Web API with:

- CRUD functionality
- Built-in **Idempotency** (to prevent duplicate requests)
- Built-in **Rate Limiting** (enabled by default)

Things to Consider

- **DTOs** should be record types.
- **Entities** are class (reference types).

More examples [here](#)[↗].