

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 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()`.
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 (simple demo it is, but you can go nuts with configuration):

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

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](#) .

Build By Example

Here is a simple Web API of 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](#) .