Weather Forecast System

Software Architecture Document

Version 0.3

Author: Saad Al-Marsumi

Revision History

Version	Date	Author	Changes
0.1	18/06/2022	Saad Al-Marsumi	Initial draft.
0.2	23/06/2022	Saad Al-Marsumi	Redesign commands and queries to use repositories and query operation instead of relying on ApiDbContext directly.
0.3	03/07/2022	Saad Al-Marsumi	Add the performance monitor and health check components.

Table of Contents

1. Introduction....4
1.1. Purpose....4
1.2. Scope....4
1.3. Abbreviations....4
1.4. References....5
1.5. Overview....5
2. Architecture....6
2.1. Architectural Views....6
2.2. Constraints....6
3. System Context View....8
3.1. System Users....9
3.2. Administrators....9
3.3. Identity Service Provider....9
3.4. Grafana Loki....10
3.5. Grafana....10
3.6. Prometheus....10
4. Container View....11
4.1. Web Application....11
4.2. Single-Page Application....12
4.3. API Application....12
4.4. Authorization Service....13
5. Component View....14
5.1. Component Types....14
5.2. Shared Components....14
5.2.1 Logging Middleware....14
5.2.2 MediatR Pipeline....15
5.2.3 Authorization Component....15
5.2.4 Metrics Middleware....15
5.2.5 Metrics Endpoints....15
5.2.6 Health Check Endpoints....16
5.3. Web Application Components....16
5.3.1 Force Authentication Middleware....16
5.3.2 Account Endpoints....16
5.3.3 YARP Reverse Proxy....18
5.3.4 OpenID Connect Middleware....18
5.3.5 File Fallback....18
5.4. API Application Components....18
5.4.1 JWT Bearer Authentication Middleware....19
5.4.2 Weather Endpoints....19
5.4.3 Commands Component....20
5.4.4 Queries Component....20
5.4.5 Infrastructure Data Component....20
5.5. Authorization Service Components....20
5.5.1 JWT Bearer Authentication Middleware....21
5.5.2 Policy Endpoints....21

5.5.3 Policy Operations Component....21
5.6. Single-Page Application....21
6. Code View....23
6.1. Shared Components....23
6.1.1 MediatR Pipeline....23
6.1.2 Authorization Component....25
6.2. API Application....27
6.2.1 Commands Component....27
6.2.2 Queries Component....29
6.2.3 Infrastructure Data Component....30
6.3. Authorization Service....32
6.3.1 Policy Operations Component....32
7. Deployment View....33
7.1. Local Docker Deployment....33
8. Security....34
8.1. Authentication....34
8.2. Authorization....36
8.3. CSRF....38
9. Solution and Project Breakdown....39
9.1. Shared Projects....39
9.2. Services....39
9.3. Web....40
9.4. Test....40
10. Standards....42
10.1. Coding Conventions....42
10.2. ASP.NET Core....43
10.3. Logging....44
10.4. Testing....45
10.5. Real Project Recommendation....45

1. Introduction

1.1. Purpose

This document provides a top-level architectural overview of the Weather Forecast System (WFS). The document will use a number of different views to depict different aspects of the system. It is intended to capture and describe the significant architectural decisions made on the system.

1.2. Scope

The architecture described in this document will be used as a reference by development team during the development of the WFS project.

1.3. Abbreviations

Abbreviation	None-abbreviated
BFF	Backend for Frontend
CQRS	Command Query Responsibility Segregation
CSRF	Cross-Site Request Forgery
JWT	Json Web Token
PKCE	Proof Key for Code Exchange
UI	User Interface
SPA	Single-Page Application
WFS	Weather Forecast System
XSS	Cross-Site Scripting

1.4. References

C4 Model at: https://c4model.com

OAuth 2.0 PKCE at: https://oauth.net/2/pkce/

Duende IdentityServer at: https://docs.duendesoftware.com/identityserver/v6

1.5. Overview

The Weather Forecast System (WFS) was commissioned with the view to provide end users with a way to publish and view weather forecast data.

The WFS solution to be introduced will provide this functionality through a web portal accessible to end users. The portal will be an enterprise level web-based solution allowing users to easily manage weather forecast data.

2. Architecture

This document presents the architecture as a series of abstraction levels based on the C4 Model. These levels will be the high-level context abstraction, the container abstraction representing applications and data stores the overall system is comprised of, the individual components making up the containers in the system, and finally the code abstraction that components are made of in the form of class diagrams.

2.1. Architectural Views

- System Context View: The system context is the starting point for the weather forecast system architecture aiming to show the big picture of the system. The focus will be on people (users) interacting with the system and any external dependencies that are outside the system's scope. This view will not focus on technologies, protocols, or any other low-level details.
- Container View: The container view is the next step zoom-in to the system boundary. A single container represents a separately runnable/deployable unit that executes code or stores data. Container examples would be a server-side web application, single-page application, database schema, file system, etc. The container view shows the high-level shape of the software architecture and the major technology choices and how containers communicate with one another.
- Component View: The component view shows the decomposition of each container to identify the major structural building blocks and their interactions. It shows how a container is made up of a number of components, what each component is, their responsibility and the technology/implementation details.
- Code View: Code View shows how each component is implemented as code using class diagrams and entity relationship diagrams.

2.2. Constraints

Architectural constraints:

- The system should have a single point of entry the end user will be interacting with implemented as a web portal that isolates the inner components of the system from public access.
- The system should reuse an already existing identity server as part of a single sign-on solution shared with external applications.

Technology constraints:

- The proposed system should support multi-platform deployments specifically Windows and Linux deployments.
- Memory based storage as a replacement for a database management system as the solution is just an architecture reference intended to be a proof-of-concept solution.
- ASP.NET Core 6 as a web framework and C# as a programming language. ASP.NET Core is a cross-platform framework that can satisfy the multi-platform deployment scenarios.
- Angular 13 with Typescript as a front-end web application framework and language.

3. System Context View

The Weather Forecast System (WFS) at its highest level of abstraction will be represented by Figure 3-1. At this level the WFS has five major components the WFS itself, the user, the identity service provider, analytics application (Grafana), log aggregation system (Grafana Loki), monitoring system (Prometheus), and the system administrator. In order for the end users to use the WFS they need to authenticate with the identity service provider and request an access token. The identity service provider, analytics application, and log aggregation are considered as an external dependencies and their implementation details are outside the scope of this document.

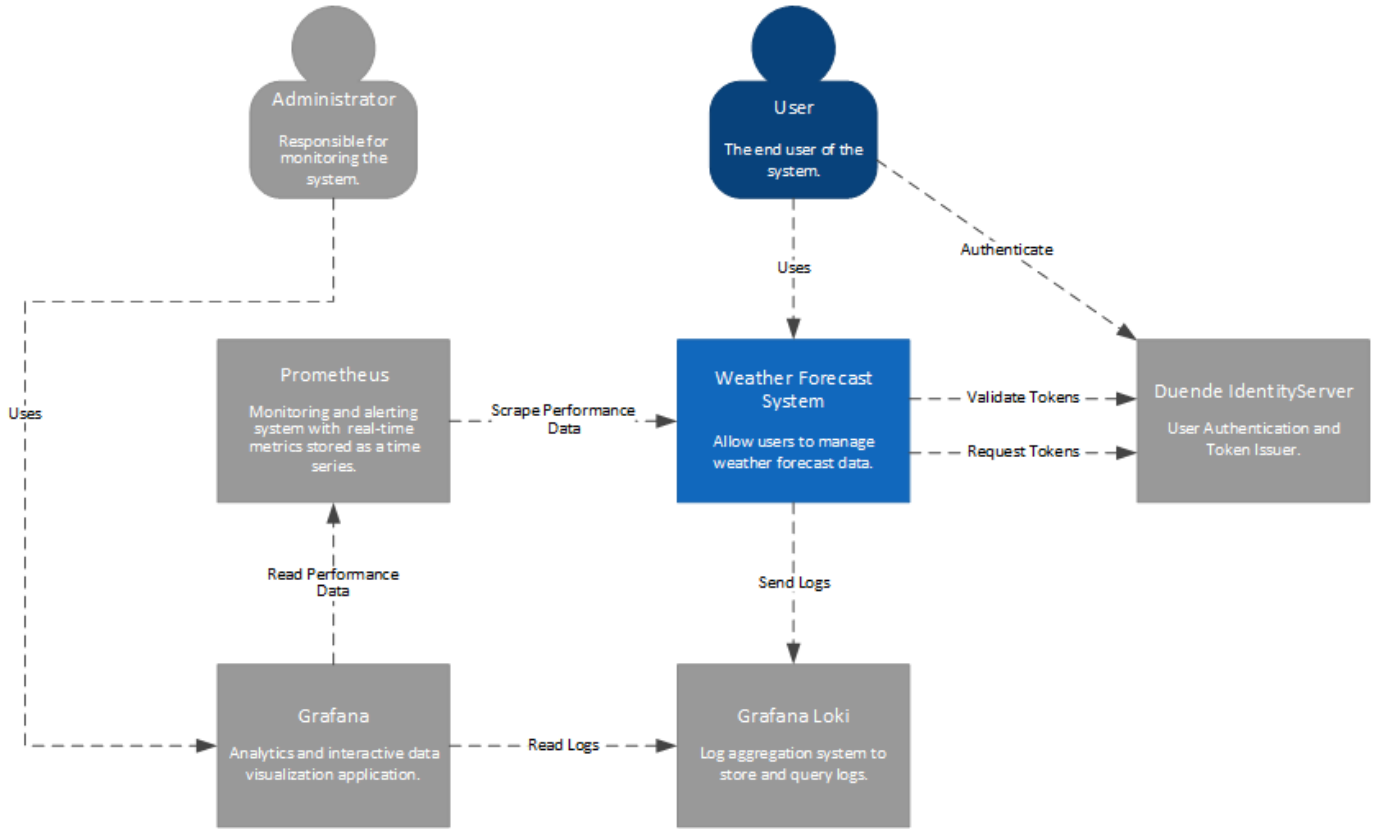

Figure 3-1 System Context

3.1. System Users

The WFS supports one category of users that are authenticated and authorized to interact with the system. The WFS does not support anonymous access.

Each authorized user will have a set of permissions that determine what the user can perform once successfully authenticated by the system.

3.2. Administrators

The WFS provides system administrators with monitoring tools to make sure service outage or any other issues will go unnoticed. a system administrator is a totally separate entity from a system user. Alerts can be set through Prometheus for certain conditions to help detect service degradation and other issues as they happen.

3.3. Identity Service Provider

The WFS will use Duende IdentityServer to provide identity services for the system. Duende IdentityServer is an OpenID Connect and OAuth 2.0 engine, it implements the OpenID Connect and OAuth 2.0 protocols.

IdentityServer will provide the following functionality:

- manage access to resources
- authenticate users using a local account store
- provide session management and single sign-on
- manage and authenticate clients
- issue identity and access tokens to clients

Duende IdentityServer was chosen as the identity solution for the WFS system because it is easy to setup and achieve the required functionality within the scope of this proof-of-concept solution.

Alternative solutions that provide the same functionality:

- OpenIddict
- Keycloak
- Gluu

3.4. Grafana Loki

The WFS will use Grafana Loki to centralize all logging storage and retrieval requirements. Grafana Loki is a log aggregation system, it is used within the WFS context to provide a centralized way to store and query logs generated by different parts of the system.

Grafana Loki was chosen as the log aggregator because of its very small resource footprint and ease of operation. Grafana Loki provides an easy to setup solution for local environments with the ability to scale horizontally to reflect a more distributed solution.

Alternative solutions that provide the same functionality:

- Elasticsearch
- Seq
- Azure Application Insight

3.5. Grafana

The WFS will use Grafana as a data analytics and visualization tool. Grafana will provide the WFS with the ability to query, visualize, and explore data stored in multiple data sources like logs from Grafana Loki and performance data from Promethues.

Alternative solutions that provide the same functionality:

- Kibana
- Azure Application Insight

3.6. Prometheus

The WFS will use Prometheus as a monitoring and alerting system with real-time metrics stored as a time series data.

Alternative solutions that provide the same functionality:

- Elastic APM
- Azure Application Insight

4. Container View

The Weather Forecast System (WFS) at the container level of abstraction will be represented by Figure 4-1. At this level the WFS has Four major components the user facing Web Application, the Single-Page Application, the back-end API Application, and the Authorization Service. Only Authenticated requests can take place among the different system components.

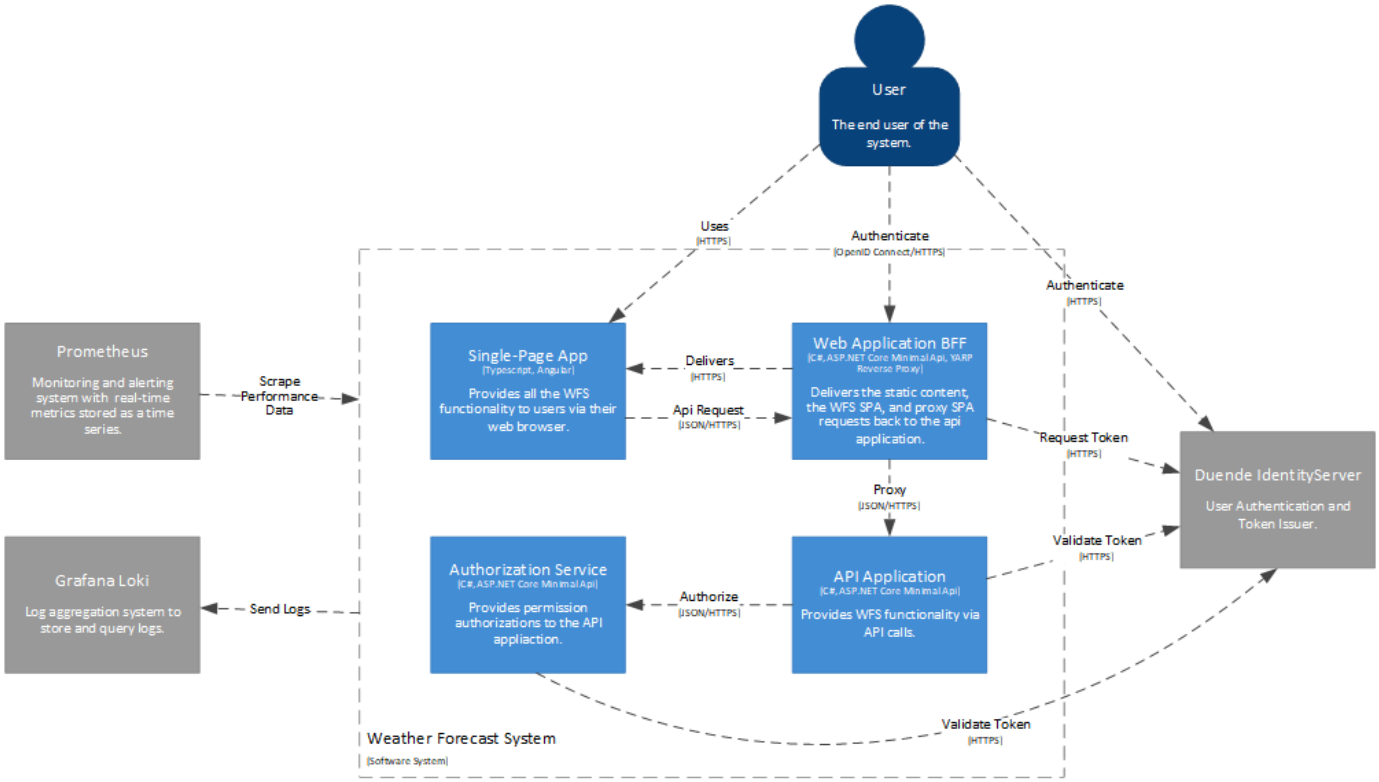

Figure 4-1 Container Diagram

4.1. Web Application

The web application will serve as the single point of entry that users of the WFS will interact with either directly or through the use of the Single-Page Application.

Responsibilities:

- Ensure all user interactions with the system are authenticated and initiate the authentication process otherwise.
- Request access token from the identity service provider on behalf of the users using OpenID Connect/OAuht 2.0 flow (see security section for more details).

- Serve all static contents and the SPA Angular application to authenticated users.
- Act as a Back-end for Front-end (BFF) and proxy all requests coming from the Angular app back to the API back-end with the appropriate authorization header forwarding the JWT access token to the API application.

Technologies used:

- C# as the programming language.
- ASP.NET Core 6.0 web framework.
- Minimal API for creating HTTP endpoints.
- YARP reverse proxy.
- OpenID Connect/OAuth 2.0.

4.2. Single-Page Application

The single-page application provides the end users with all the WFS functionality via the web browser, delivered over HTTPS by the Web Application BFF to authenticated users only.

Responsibilities:

- Implement all the UI functionality provided by the WFS.
- Communicate with the BFF to access the WFS back-end services over HTTPS.

Technologies used:

- Typescript as the programming language.
- Angular 13 as the front-end framework.
- JSON over HTTPS.

4.3. API Application

The API Application exposes the WFS functionality as a Restful service providing a predefined interface for clients to invoke. The API Application should not be accessible directly by end clients and all traffic should be proxied by the BFF web application.

Responsibilities:

- Provide a RESTful interface for all the WFS functionality.

Technologies used:

- C# as the programming language.

- ASP.NET Core 6.0 web framework.
- Minimal API for creating HTTP endpoints.
- JSON over HTTPS.
- OpenID Connect/OAuth 2.0.

4.4. Authorization Service

The Authorization Service provides a centralized solution for configuring and enforcing access control for the WFS. Access control in the WFS will be implemented in the form of permissions where every user will be a member of a user group (can map to a specific role) and every user group can be assigned a set of permissions. Full permission management capabilities will be outside the scope of this document and the proof-of-concept solution it's describing.

Responsibilities:

- Store and manage user permissions.
- Provide HTTP endpoints to evaluate user permissions.

Technologies used:

- C# as the programming language.
- ASP.NET Core 6.0 web framework.
- Minimal API for creating HTTP endpoints.
- JSON over HTTPS.
- OpenID Connect/OAuth 2.0.

5. Component View

5.1. Component Types

The component view will be describing different types of components used to implement the WFS containers described in section 4. These types are as follows:

- Middleware: A middleware component is either a custom or third party ASP.NET Core middleware that integrates with the Request/Response pipeline. This type of component represents a small snippet of code that is executed as part of the ASP.NET Core pipeline.
- Component: This type of component represents a functionality that the proposed system will be fully implementing in code. The size of these components can vary from a single class to multiple classes and interfaces that interact with each other to achieve the required result.
- Endpoint: An endpoint component represents the implementation of the HTTP endpoints exposed by one of the system containers (applications and services). For the WFS all endpoint components will be using the ASP.NET Core minimal Api mapping.

5.2. Shared Components

Some of the components in the WFS will be used by multiple containers, they will be described in this section and will not be repeated for every container.

5.2.1 Logging Middleware

The Logging Middleware implements the Serilog ASP.NET Core HTTP structured request logging configuration. The middleware needs to be added to the ASP.NET Core pipeline before any request handlers to be able to intercept all incoming requests.

The configuration implemented in this component will extend the log events with any required properties. Additionally, any excluded paths that should not be logged outside of debug or trace level should be identified in this component. The Logging Middleware will also be responsible of configuring any Serilog sinks needed by the system like Grafana Loki sink.

5.2.2 MediatR Pipeline

MediatR is an in-process messaging that supports request/response, commands, queries, notifications and events dispatching. The WFS uses MediatR to implement a CQRS pattern to process incoming requests.

The MediatR Pipeline component implements any configuration required to integrate MediatR into the ASP.NET Core dependency injection container. Additionally, any cross-cutting concerns needed for the CQRS pattern like logging, validation, and transactions will be implemented in this component.

5.2.3 Authorization Component

The Authorization Component implements the authorization policy engine the WFS will use to decide which users can access what parts of the system. The component implements all the integration points that allows it to execute as part of the ASP.NET Core authorization system. Additionally, two types of clients will be implemented as part of this component, remote and local clients. A remote client will be used by a container like the Application API to contact the authorization service. The authorization service will use the local client to resolve user policies.

The WFS implements the following application wide permissions:

Permissoin	Description
CreateWeather	The permission needed for creating new weather forecast data entry.
DeleteWeather	The permission needed for deleting and existing weather forecast data entry.
ViewWeather	The permission needed for reading weather forecast data entries.

5.2.4 Metrics Middleware

The Metrics Middleware Component configures the Prometheus-Net metrics middleware. The Prometheus-Net middleware instruments the different system containers to expose metrics data through the metrics endpoint.

5.2.5 Metrics Endpoints

The Metrics Endpoints component is responsible for mapping all performance monitoring related HTTP endpoints that will be exposed by the system containers. The component will implement the HTTP endpoints using ASP.NET Core endpoints builder. The component will handle requests on the /metrics path. The metrics endpoints will be scraped by Prometheus to obtain performance data.

Metrics endpoints implements the following RESTful api:

Path	HTTP Method	Description
/metrics	GET	Returns the metrics data collected by the instrumentation middleware.

5.2.6 Health Check Endpoints

The Health Check Endpoints component is responsible for mapping all status and service health related HTTP endpoints that will be exposed by the system containers. The component will implement the HTTP endpoints using ASP.NET Core endpoints builder. The component will handle requests on both /live and /ready paths.

Health check endpoints implements the following RESTful api:

Path	HTTP Method	Description
/live	GET	Returns a response indicating the called container is alive.
/ready	GET	Returns a response indicating the health of the container after checking the availability of its dependencies. The result will indicate whether the container is ready to accept requests.

5.3. Web Application Components

The Web Application container at the components level of abstraction will be represented by Figure 5-1. At this level the Web Application container has six major components the logging middleware, force authentication middleware, account endpoints, YARP reverse proxy, OpenID Connect middleware, metrics middleware, metrics endpoints, health check endpoints, and file fallback.

The logging middleware is described in the shared component section and will not be covered again.

5.3.1 Force Authentication Middleware

The Force Authentication Middleware is a single ASP.NET Core middleware that is added before any request handling takes place on the pipeline. The implementation of this component checks every incoming request for an authenticated user, if it is not found a challenge is returned to the caller to force the authentication flow to take place. The component acts as a guard that only allows authenticated users to access the functionality and features provided by the WFS.

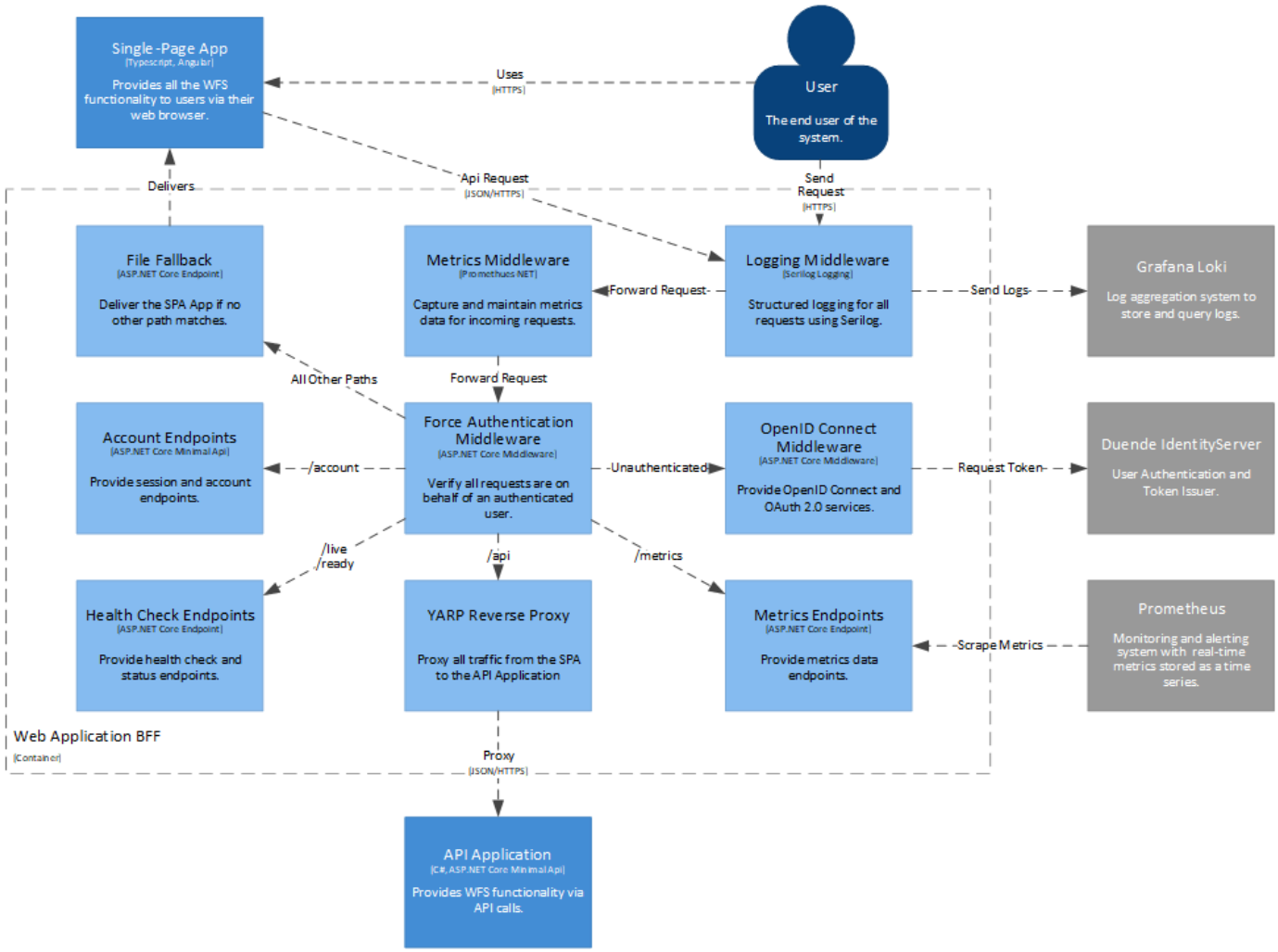

Figure 5-1 Web Application (BFF) Component Diagram.

5.3.2 Account Endpoints

The Account Endpoints component is responsible for mapping all the account and session related HTTP endpoints that will be exposed by the BFF Web Application. The component will implement the HTTP endpoints using ASP.NET Core minimal Api mappings. The component will handle requests starting with the /account path.

Account Endpoints implements the following RESTful api:

Path	HTTP Method	Description
/account/session	GET	Returns a session object of the authenticated user containing the user information.
/account/postlogin	GET	Performs a server-side redirect after the user authentication to circumvent the restrictions applied by the Angular development proxy server.
/account/logout	POST	Returns a sign-out challenge.

5.3.3 YARP Reverse Proxy

YARP is a reverse proxy library for ASP.NET Core hosted applications. YARP provides a simple configuration model that will help implement the Back-end for Front-end pattern required in the WFS. YARP will simplify the implementation of the BFF by eliminating the need to create, configure, and manage HTTP Client instances for each service that needs to be called by the end users.

The YARP Reverse Proxy component will be responsible for setting up and configuring the YARP runtime to proxy Api requests received from the Single-Page Application to the back-end API Application. The YARP component will be configured to handle all requests coming on the /api path. Additionally, the component will be responsible for forwarding JWT access tokens found in the user session to any downstream back-end services.

5.3.4 OpenID Connect Middleware

The OpenID Connect Middleware component will be responsible for configuring the ASP.NET Core authentication library to enable the use of the OpenID Connect authentication flow. The component will add the required functionality needed by the WFS to properly initialize the authentication flow and process the required redirects between the user browser and the identity service provider. Additionally, the component will manage the storage of any tokens obtained from the identity service provider.

5.3.5 File Fallback

The File Fallback component is an ASP.NET Core default endpoint that will be used to deliver the static files of the Single-Page Applications for any request that is not handled by any other handlers. The component assumes that any request that has no other handler will have to be passed down to the SPA. This component will only be in effect in a production build due to the use of the Angular development server to serve the SPA.

5.4. API Application Components

The API Application container at the components level of abstraction will be represented by Figure 5-2. At this level the API Application container has seven major components the logging middleware, JWT bearer authentication middleware, authorization component, weather endpoints, metrics middleware, metrics endpoints, health check endpoints, MediatR pipeline, commands component, queries component, and infrastructure data component.

The logging middleware, authorization component, and the MediatR pipeline are described in the shared component section and will not be covered again.

The API application uses a simplified Command and Query Responsibility (CQRS) pattern to implement all use cases required. The API Application makes use of the shared MediatR Component to achieve the required CQRS behavior. The API Application implements two separate models for read and write operations. By recognizing that both of these models have different requirements the possibility of evolving them independently from each other becomes more streamlined.

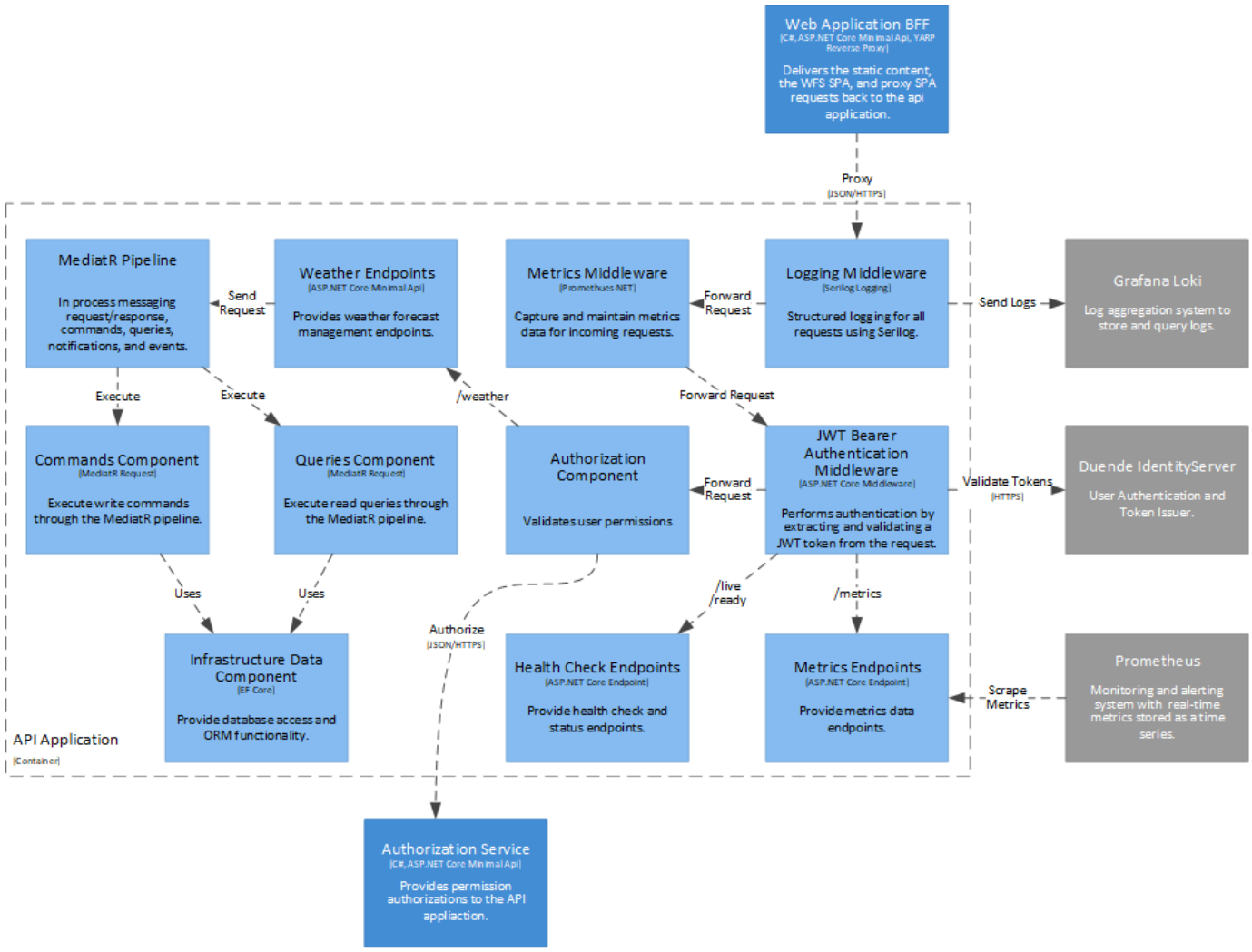

Figure 5-2 API Application component diagram.

5.4.1 JWT Bearer Authentication Middleware

The JWT Bearer Authentication Middleware component will be responsible of configuring the ASP.NET Core authentication library to enable an application to receive and validate an OpenID Connect JWT bearer token. The component will add the required functionality needed by the API Application to validate received user tokens by contacting the identity service provider.

5.4.2 Weather Endpoints

The Weather Endpoints component is responsible for mapping all the weather-related HTTP endpoints that will be exposed by the API Application. The component will implement the HTTP endpoints using ASP.NET Core minimal Api mappings. The component will handle requests starting with the /weather path.

Weather Endpoints implements the following RESTful api:

Path	HTTP Method	Description
/weather	GET	Returns a collection of all weather forecast data.
/weather	POST	Creates a new weather forecast data entry.
/wearther/{id}	DELETE	Deletes an existing weather forecast data entry.

5.4.3 Commands Component

The Commands component will be responsible for implementing all the commands used by the API Application. Commands in the context of the WFS are requests that perform actions with the aim of changing the state of the system. Commands in WFS are responsible for implementing any required business logic and rules. All commands are implemented as MediatR requests sent from the weather endpoint and executed by the MediatR pipeline.

5.4.4 Queries Component

The Queries component will be responsible for implementing all the queries used by the API Application. Queries in the context of the WFS are requests that perform actions with the aim of retrieving data without changing the state of the system. All queries are implemented as MediatR requests sent from the weather endpoint and executed by the MediatR pipeline.

5.4.5 Infrastructure Data Component

The Infrastructure Data Component is responsible for abstracting data access and implementing all the infrastructure storage concerns. The component will be implemented using Entity Framework Core to map system entities to database objects. The infrastructure component will be only used by the commands and queries within the API Application.

5.5. Authorization Service Components

The Authorization Service container at the components level of abstraction will be represented by Figure 5-3. At this level the Authorization Service container has four major components the logging middleware, JWT bearer authentication middleware, metrics middleware, metrics endpoints, health check endpoints, policy endpoints, and policy operations.

The logging middleware is described in the shared component section and will not be covered again.

5.5.1 JWT Bearer Authentication Middleware

The JWT Bearer Authentication Middleware component will be responsible of configuring the ASP.NET Core authentication library to enable an application to receive and validate an OpenID Connect JWT bearer token. The component will add the required functionality needed by API Application to validate received user tokens by contacting the identity service provider.

5.5.2 Policy Endpoints

The Policy Endpoints component is responsible for mapping all the policy related HTTP endpoints that will be exposed by the Authorization Service. The exposed policy endpoints will be used by other containers in the WFS to evaluate user permissions. The component will implement the HTTP endpoints using ASP.NET Core minimal Api mappings. The component will handle requests starting with the /policy path.

5.5.3 Policy Operations Component

The Policy Operations Component is responsible for implementing the policy runtime and any infrastructure concerns needed to store and retrieve user permissions. A full implementation is outside the scope of this document and its proof-of-concept solution. The component will be implementing a memory based storage for the purpose of providing a simple functional solution.

5.6. Single-Page Application

The single-page application components will be omitted for brevity in favor of focusing on server side components only.

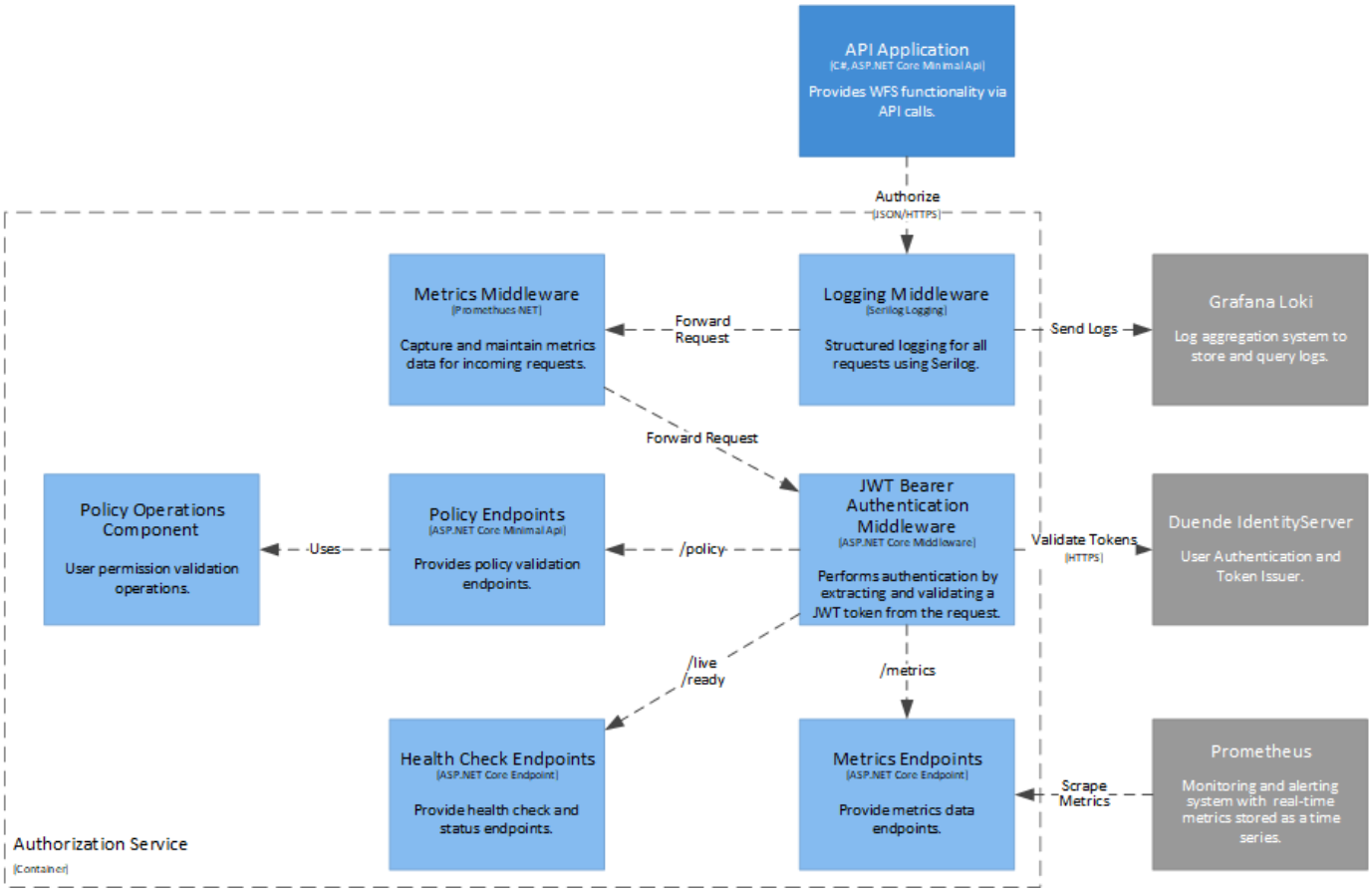

Figure 5-3 Authorization Service component diagram.

6. Code View

This section will provide class diagrams for a select number of components that needs expanded details about their implementation.

6.1. Shared Components

6.1.1 MediatR Pipeline

Figure 6-1 MediatR Pipeline Component class diagram

Code elements:

Element	Project/ Package	Description
`IRequest<T>`	MediatR	Part of the MediatR runtime to indicate a class as a request. Implemented buy all commands and queries in the system.
`IRequestHandler<TRequest, TResponse>`	MediatR	Part of the MediatR runtime to indicate a class as a request handler for a specific request.
`INotification`	MediatR	Part of the MediatR runtime ton indicate a class as a notification (an event with multiple subscribers).
`INotificationHandler<T>`	MediatR	Part of the MediatR runtime to indicate a class as a notification handler.
`IPipelineBehavior<TResponse, TRequest>`	MediatR	Part of the MediatR runtime to to indicate a class as a behavior (implements cross cutting concerns for all requests).
`LoggingBehavior`	Common.MediatR	Logs the request object being handled by the MediatR pipeline.
`ValidatorBehavior`	Common.MediatR	Apply all registered validation rules for the current request.
`EFTransactionBehavior`	Common.MediatR	Provide a transactional context for the current request.
`Extensions`	Common.MediatR	Add the MediatR services to the ASP.NET Core dependency injection container.
`ISecuritySensitive`	Common.MediatR	Implemented by the request to indicate the need to redact sensitive information from the request before logging it.
`ITransactionalRequest`	Common.MediatR	Implemented by the request to indicate the need for a transactional context.
`IRequestResult`	Common.MediatR	Defines the members and operations implemented by RequestResult.
`RequestResult`	Common.MediatR	A wrapper class around the request result. Implements IRequestResult. All requests passing through the MediatR pipeline should return their result as a RequestResult. The RequestResult indicates the success or the failure of the operation. The use of a result class adds the benefit of not relying on throwing exceptions to interrupt the execution flow.
`DomainEvent`	Common	The base class for all system events.

`EventNotification<T>`	Common.MediatR	A wrapper class around domain events to make them publishable through the MediatR pipeline by implementing the INotification interface.
`DomainEventService`	Common.MediatR	The service used to convert DomainEvent instances to EventNotification instances and publish them through MediatR.

6.1.2 Authorization Component

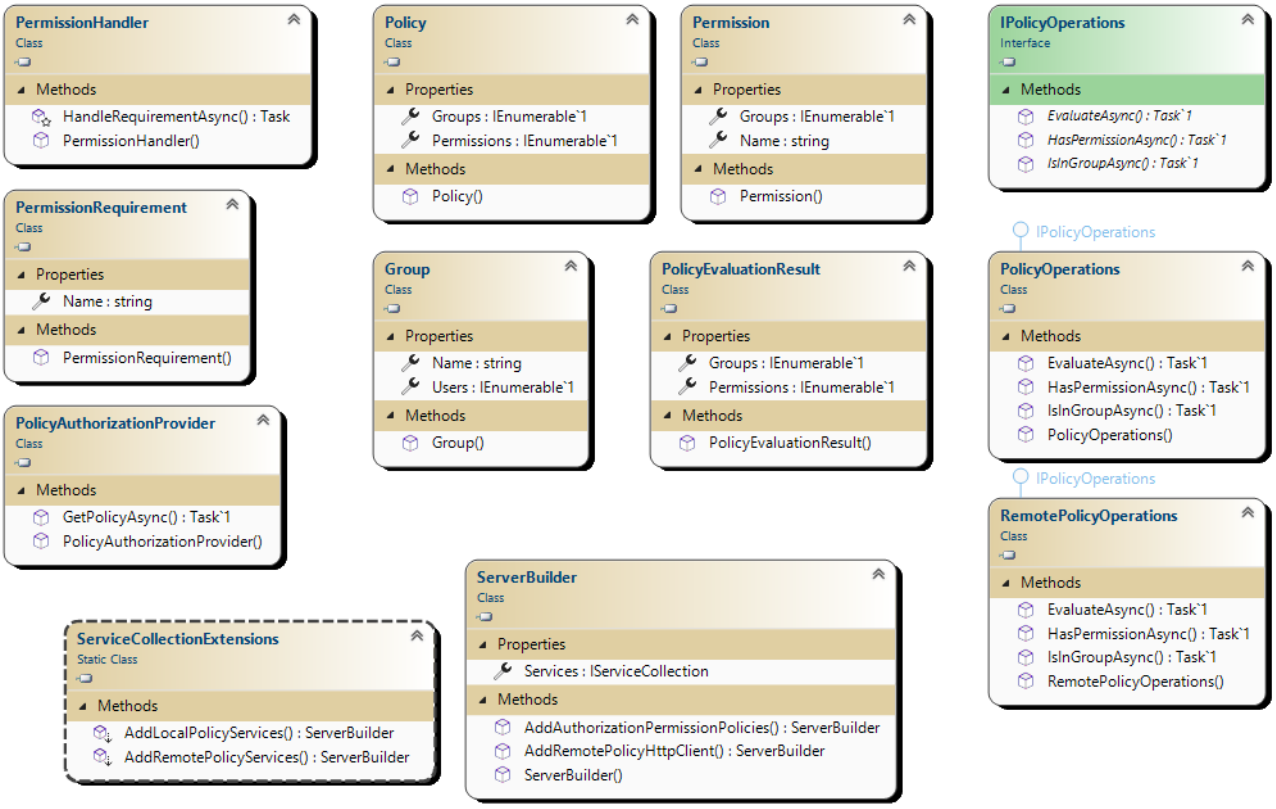

Figure 6-2 Authorization Component class diagram.

Code elements:

Element	Project/Package	Description
`Policy`	Common.Authorization	Represents the entire policy configuration with all permissions and user groups defined in the system. This class will be the single source of truth regarding which user can have what permission. Given a user the policy instance can evaluate and return what permissions and membership the user has.

`Permission`	Common.Authorization	Represents a single system permission with all the groups assigned that specific permission. Given a collection of groups the permission instance can evaluate whether these groups have been granted the specified permission.
`Group`	Common.Authorization	Represents a single users group in the system with all the users that are members of that specific group. Given a user the group instance can evaluate whether this user is a member of the specified group.
`PolicyEvaluationResult`	Common.Authorization	Represents the result of a policy evaluation for a specific user.
`IPolicyOperations`	Common.Authorization	Defines the operations needed to evaluate users against a policy instance.
`PolicyOperations`	Common.Authorization	Provides an implementation to evaluate users against a policy instance. This class will be used through the IPolicyOperations interface for local evaluation operations.
`RemotePolicyOperations`	Common.Authorization	Provides an implementation to evaluate users through calling the Authorization Service. This class is will be used through the IPolicyOperations interface for remote evaluation operations.
`PermissionHandler`	Common.Authorization	Extends the AuthorizationHandler class to integrate with the ASP.NET Core authorization. The PermisionsHanlder will provide ASP.NET Core authorization the means to check if a specific user has a required permission through the use of IPolicyOperations.
`PermissionRequirement`	Common.Authorization	Represent the requirement needed to pass the authorization check, this will the name of the required permission.
`PolicyAuthorizationProvider`	Common.Authorization	Extends the DefaultAuthorizationPolicyProvider class to integrate with the ASP.NET Core authorization. This provider will automatically create ASP.NET Core authorization policies for all permissions used in the application without the need to manually define every single one of them.
`ServiceCollectionExtensions`	Common.Authorization	Add the needed authentication services to the ASP.NET Core dependency injection container.
`ServerBuilder`	Common.Authorization	Add the needed authentication services to the ASP.NET Core dependency injection container.

6.2. API Application

6.2.1 Commands Component

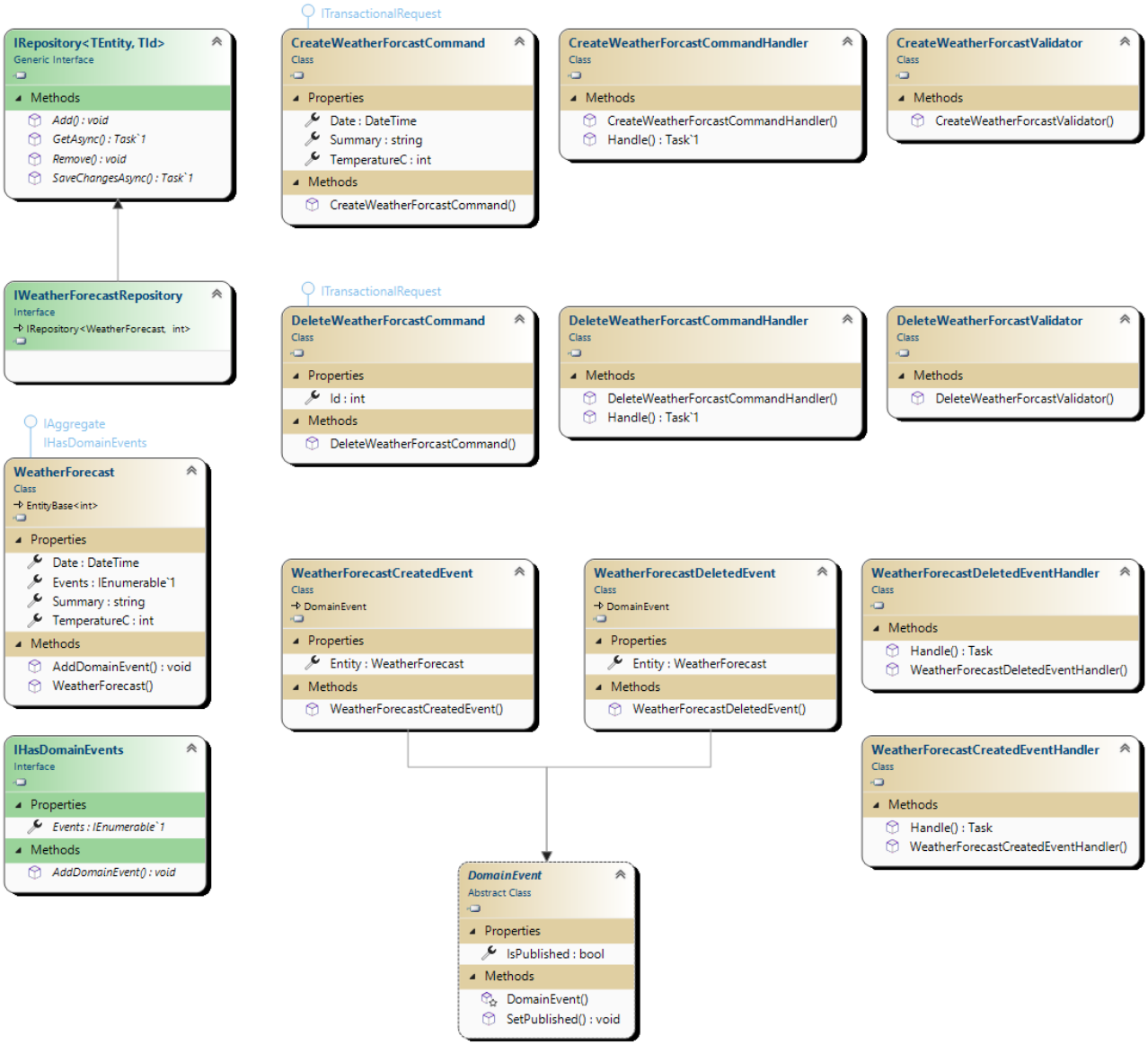

Figure 6-3 API Application Commands Component

Code elements:

Element	Project/ Package	Description
`CreateWeatherForecastCommand`	Api.Application	The command that will be send through the MediatR pipeline to create a new weather forecast. Has to implement IRequest to be accepted as a MediatR request. Implements the ITransactionalRequest to indicate the need for the request to be transactional.
`CreateWeatherForecastCommandHandler`	Api.Application	Handles the CreateWeatherForcastCommand request inside the MediatR pipeline. Implements the IRequestHandler interface.
`CreateWeatherForecastValidator`	Api.Application	Validate the incoming CreateWeatherForcastCommand inside the MediatR pipeline using FluentValidation.
`DeleteWeatherForecastCommand`	Api.Application	The command that will be send through the MediatR pipeline to delete a weather forecast. Has to implement IRequest to be accepted as a MediatR request. Implements the ITransactionalRequest to indicate the need for the request to be transactional.
`DeleteWeatherForcastCommandHandler`	Api.Application	Handles the DeleteWeatherForcastCommand request inside the MediatR pipeline. Implements the IRequestHandler interface.
`DeleteWeatherForcastValidator`	Api.Application	Validate the incoming DeleteWeatherForcastCommand inside the MediatR pipeline using FluentValidation.
`IWeatherForecastRepository`	Api.Domain	Defines the WeatherForecast repository operations. This class is used by the command hadlers.
`WeatherForecast`	Api.Domain	The Entity encapsulating the weather forecast data.
`WeatherForecastCreatedEvent`	Api.Domain	The event that will be published when a weather forecast entity is created. Implements the INotification interface.
`WeatherForecastDeletedEvent`	Api.Domain	The event that will be published when a weather forecast entity is deleted. Implements the INotification interface.
`WeatherForecastCreatedEventHandler`	Api.Application	Handles the WeatherForecastCreatedEvent event inside the MediatR pipeline. Implements the

		INotificationHandler interface.
`WeatherForecastDeletedEventHandler`	Api.Application	Handles the WeatherForecastDeletedEvent event inside the MediatR pipeline. Implements the INotificationHandler interface.

6.2.2 Queries Component

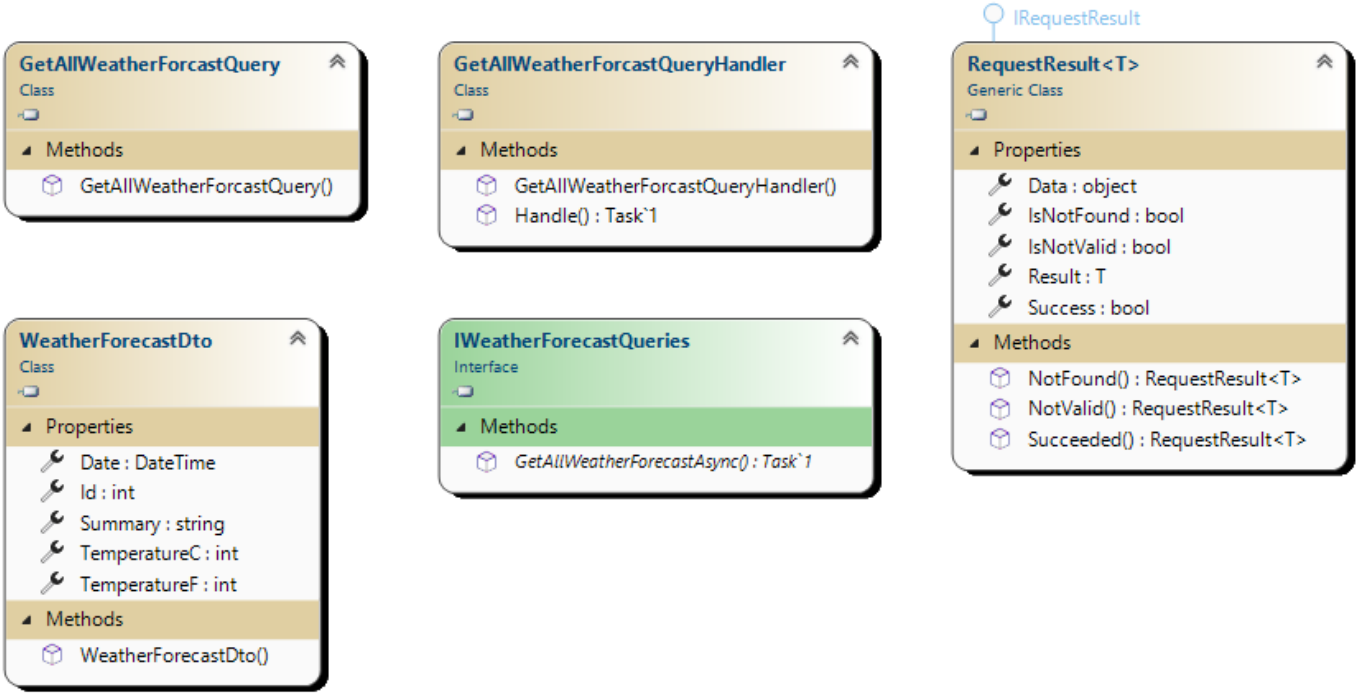

Figure 6-4 API Application Queries Component

Code elements:

Element	Project/ Package	Description
`GetAllWeatherForcastQuery`	Api.Application	The query that will be send through the MediatR pipeline to retrieve all the weather forecast records. Has to implement IRequest to be accepted as a MediatR request.

`GetAllWeatherForcastQueryHandler`	Api.Application	Handles the GetAllWeatherForcastQuery request inside the MediatR pipeline. Implements the IRequestHandler interface.
`WeatherForecastDto`	Api.Application	The result type of the GetAllWeatherForcastQuery that will be wrapped with the RequestResult class.
`IWeatherForecastQueries`	Api.Application	Defines the WeatherForecast query operations. This class will be used by the query handlers.

6.2.3 Infrastructure Data Component

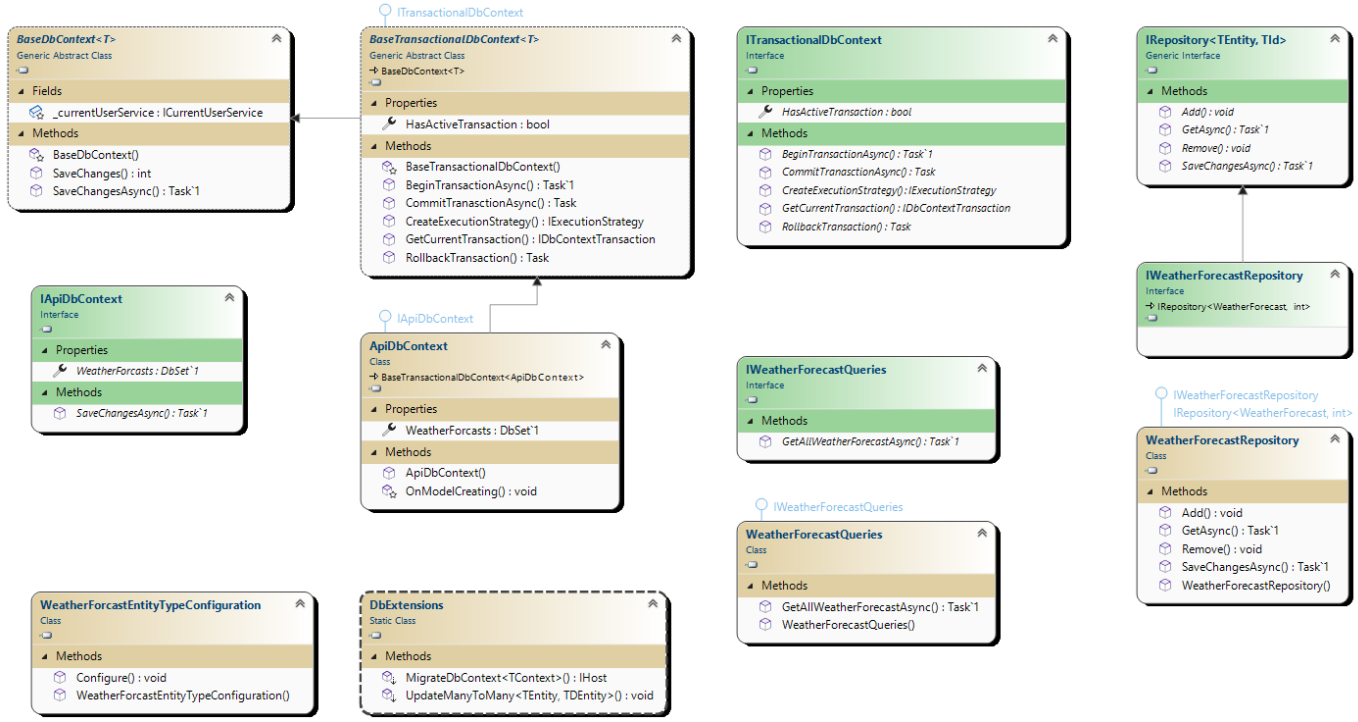

Figure 6-5 API Application Infrastructure Data Component

Code elements:

Element	Project/ Package	Description
`BaseDbContext`	Api.Infrastructure	Implement optimistic concurrency and audit functionality for all DbContext classes inheriting it.
`ITransactionalDbContext`	Api.Infrastructure	Defines transactional behavior functionality for all DbContext classes implementing it. This interface us used by the EFTransactionBehavior in the MediatR pipeline to manage transactions.
`WeatherForcastEntityTypeConfiguration`	Api.Infrastructure	Configure the WeatherForcast entity mapping to database objects.
`IApiDbContext`	Api.Infrastructure	Defines the interface used by the application to access the ApiDbContext. This interface will be used by the repositories and queries implementation.
`ApiDbContext`	Api.Infrastructure	Implements the IapiDbContext interface used by the application to provide data operations through Entity Framework Core.
`IRepository`	Api.Domain	Defines the interface used to implement repository operations. This is a generic class that takes an IAggregate entity as the generic type.
`WeatherForecastRepository`	Api.Infrastructure	Implements the WeatherForecast repository operations. This class will be used by the Commands Component.
`WeatherForecastQueries`	Api.Infrastructure	Implements the WeatherForecast query operations. This class will be used by the Queries Component.

6.3. Authorization Service

6.3.1 Policy Operations Component

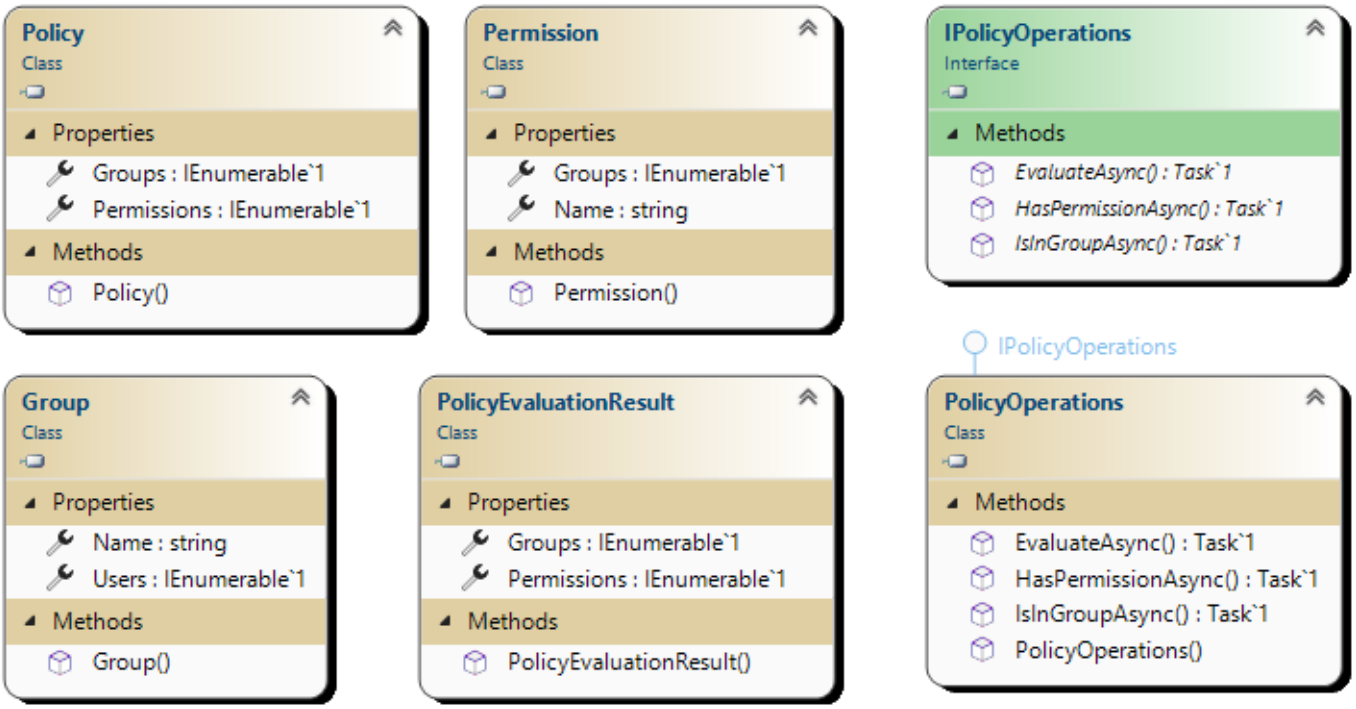

Figure 6-6 Authorization Service Policy Operations Component

The Policy Operations Component uses a subset of the Authorization Component described in section 6.1.2 (refer to section 6.1.2 for more details).

7. Deployment View

This section will provide an overview of the different deployment scenarios supported by the WFS and how the different system parts map to their correspondent infrastructure requirements.

7.1. Local Docker Deployment

Figure 7-1 shows how the WFS is deployed in a local Docker environment. All incoming traffic is routed through NGINX as a reverse proxy with only HTTPS support. Docker only publishs three ports used by NGINX to the outside. These ports are 8080 for the BFF, 8090 for the IdentityServer, and 3000 for Grafana.

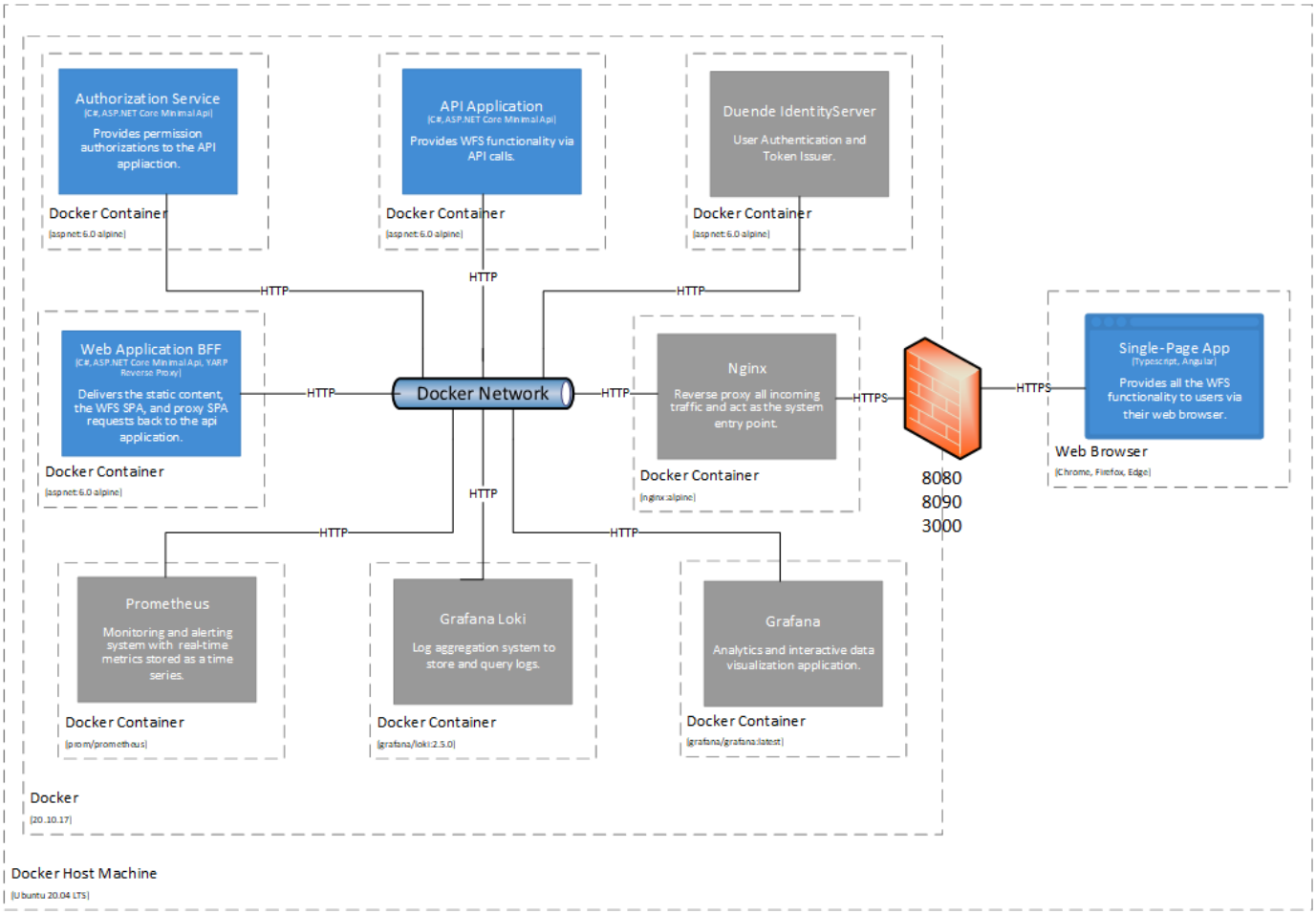

Figure 7-1 Local Docker Deployment

8. Security

8.1. Authentication

The Weather Forecast System (WFS) will be using OpenID Connect and OAuth 2.0 as the identity layer and the authorization protocol respectively to authenticate users. The Authorization Code Flow with Proof Key for Code Exchange (PKCE) is chosen as the authentication flow. The Authorization Code Flow + PKCE flow is specifically designed to authenticate native or mobile users and is considered best practice for Single-Page applications (see the OAuth 2.0 specification for more details).

The Web Application (BFF) will force all incoming requests to authenticate before accepting them. All unauthenticated requests will be responded to with a challenge to initialize the authentication flow. The WFS will not even deliver the SPA without an authenticated session already established.

After receiving an unauthenticated request and with PKCE enabled for the Authorization Code Flow the Web Application BFF generates a random value called a Code Verifier that is used to produce a Code Challenge. The Code Challenge will be sent to the Identity Service Provider (IdentityServer) as part of the Authorization Code Flow request. The IdenntityServer will store the Code Challenge to be used as a verification mechanism in a later step.

After authenticating the user at the IdentityServer, the user will be redirected back to the BFF with an Authorization Code. The BFF will use the Authorization Code and the Code Verifier to request a token on behalf of the user. The Code Verifier is hashed by the IdentityServer and checked against the stored Code Challenge. If the check is successful the requested tokens will be returned to the BFF where they will be encrypted and stored in the session cookie. Figure 7-1 shows the steps of the authentication flow.

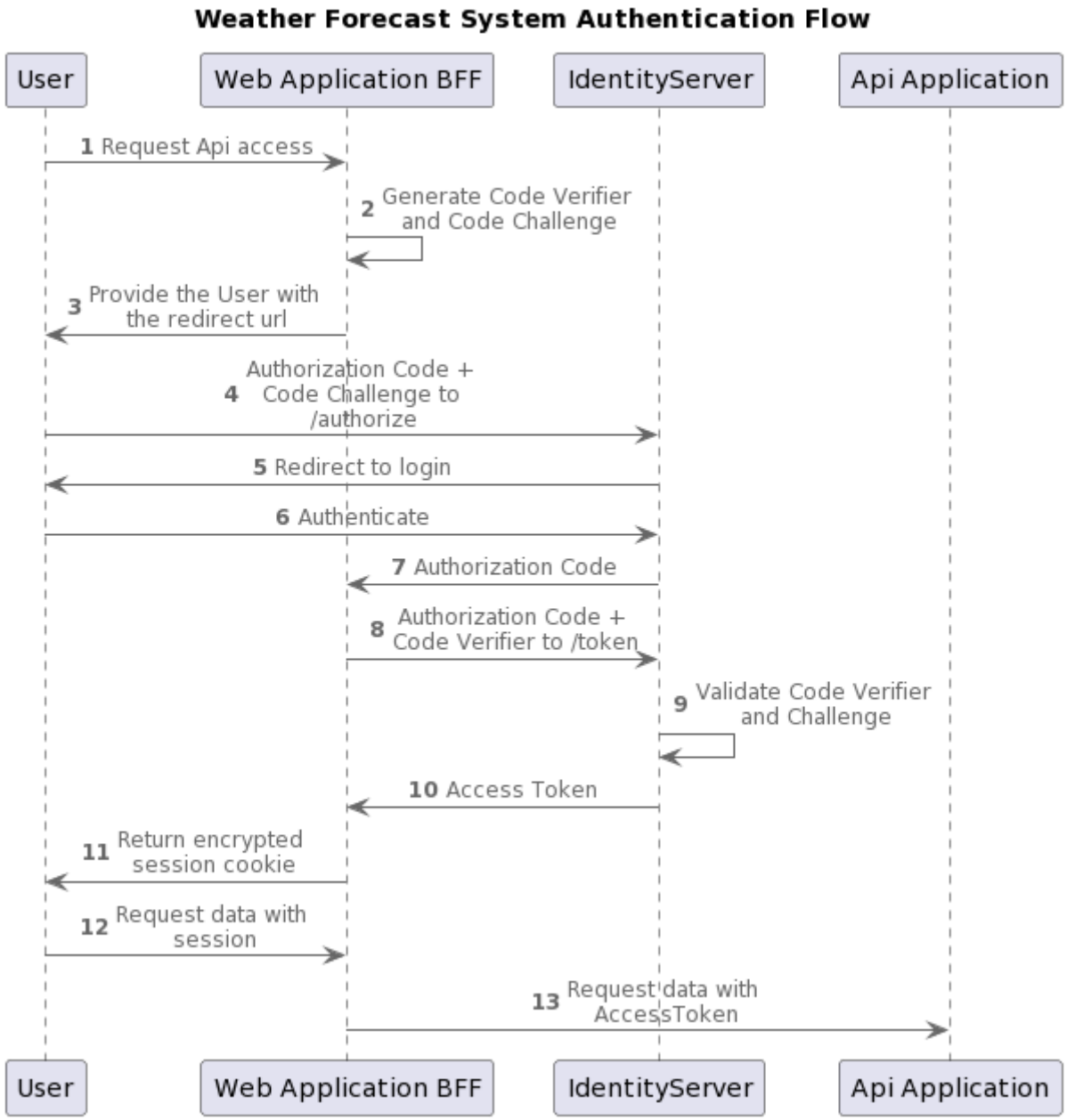

Figure 8-1 WFS Authentication Flow

8.2. Authorization

This section will describe the approach used to authorize system users. Authorization in the WFS context is the control over which user could perform what actions on different parts of the system. The WFS follows a permission for user group approach to enforce the access control for the authenticated users.

The WFS authorization approach involves two main concepts: groups and permissions. A group in a collection of users that require the same access control to perform a certain task. A permission on the other hand maps to a specific operation within the WFS that a user can or can not perform. within the WFS context , permissions are only assignable to groups. An example would be for viewing the weather data a user needs to be a member of a group that has the ViewWeather permission.

The WFS implement the relation between groups and permissions using Policy objects. A policy will have a collection of groups with their users and a collection of permissions with their groups. Policy objects are used to evaluate a user permissions by returning a collection of both groups and permissions for that specific user.

The Authorization Service is responsible for maintaining the system wide Policy and all other system components need to consult the Authorization Service to obtain evaluations of user policies. Within the WFS context, policy evaluation takes place at the API Application after retrieving the user policy from the Authorization Service and based on the result a request can be granted access or rejected with a 403 forbidden response. Figure 7-2 shows the authorization sequence of the WFS.

The choice of using a centralized Authorization service and not rely on roles or permissions stored in the JWT token itself can help make permission revocation take effect faster and almost immediately. Relying on permissions as part of the token will require either a sign-out to take a place or a token revocation from the Identity Service Provider that could take a while to take effect.

As for the choice of evaluating the user permissions on the API Application side will allow us to evaluate multiple permissions with one request to the Authorization Service. Additionally, the user permissions will be cached at the API level for a few seconds to allow for better scalability under heavy load. This will still achieve almost immediate permissions revocation with a delay of few seconds.

An alternative approach would be centralize the authorization evaluation at the Web Application BFF by having the BFF call the Authorization Service and if the user have the required permission the request would be forwarded to the API Application. This approach could raise scalability issues by giving the BFF more responsibilities than it needs to. It could also present a single point of failure from a security point of view for if the BFF is bypassed for any reason the service behind it will be fully exposed. For these reasons the WFS applies a service level authorization approach rather than an edge level authorization approach.

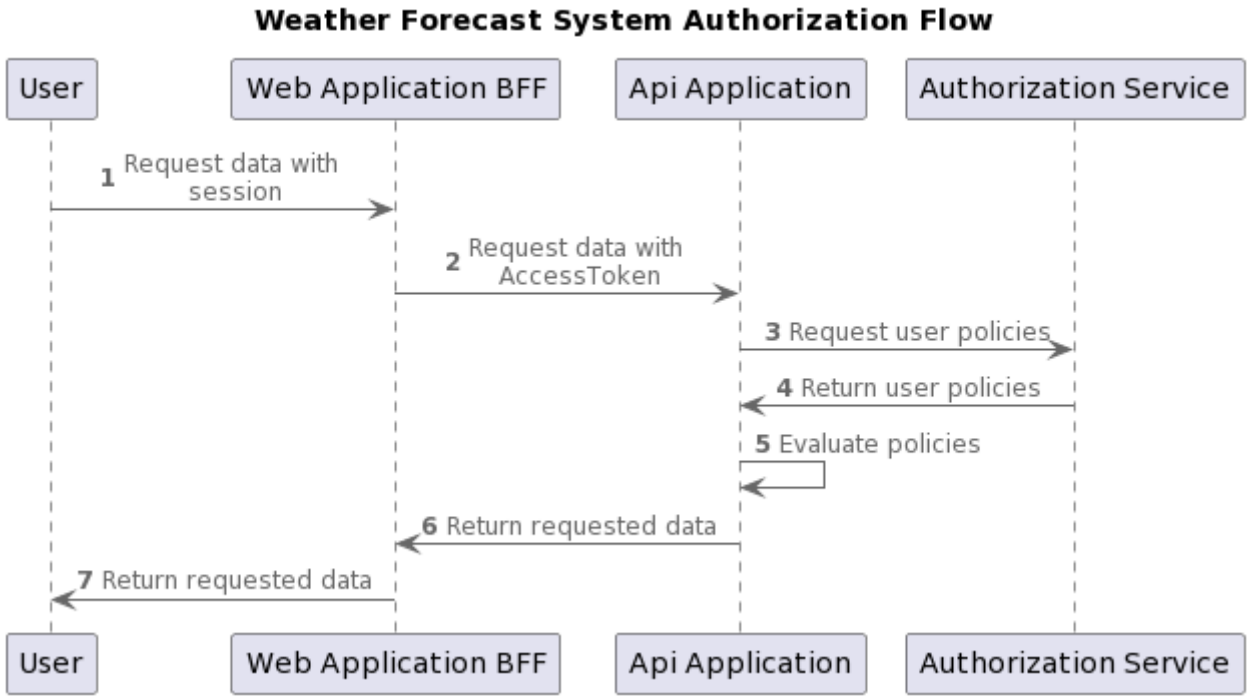

Figure 8-2 WFS uthorization Flow

8.3. CSRF

Cross-Site Request Forgery (CSRF) can be a concern for applications that use cookies as part of their authentication flow which is what the WFS is doing. An alternative approach to using cookies for storing authentication sessions would be the reliance on the browser local storage which can mitigate the CSRF risk but can be open for a Cross-Site Scripting (XSS) attack. Additionally, The OpenID Connect/OAuth 2.0 Implicit flow which was traditionally used for single-page applications which relies on storing tokens in the browser local storage has been deprecated by OAuth (see the OAuth 2.0 PKCE specification for more details).

Based on the above, the authentication flow used by the WFS is still considered best practice but with the need to mitigate any CSRF risk that is associated with the use of authentication cookies.

Modern web browsers have come a long way since cookies were first introduced and they are adapting more strict rules when handling them. Even though more strict browser rules are helping mitigate risks associate with CSRF attacks, it should not be relied on solely to protect against CSRF.

Within the WFS context, CSRF is a browser only concern, and WFS will be mitigating its risk at the Web Application BFF level. The BFF is the only part of the system browsers can interact with as it is the entry point of the WFS. The Web Application BFF will be using a combination of cookie policies and the ASP.NET Core Antiforgery tokens to reduce the risk associated with CSRF. The following measures are taken to achieve that:

- Generate an Antiforgery token per user session and deliver it to the SPA.
- All incoming requests needs to include the Antiforgery token in the X-CSRF header.
- The BFF validates the incoming token for all authenticated requests.
- Invalid tokens will result in a 400 Bad Request response.
- All cookies used will have a Same Site mode set to Strict, Secure flag set to true, and a lifetime span set to the session length.
- All cookie names should be prefixed with __Host to prevent them being overridden by a malicious subdomain.
- Authentication cookies will be set to HTTP Only to prevent JavaScript access.

9. Solution and Project Breakdown

All the different components of the WFS will be encapsulated within a single Visual Studio solution which will contain the following .NET Core projects.

9.1. Shared Projects

- Common

  This is a .NET Core 6 class library project which contains code shared will all the WFS projects. The project will include shared exceptions, logging setup and middleware, exception handling, application permissions, base entity types, and shared services.

- Common.EF

  This is a .NET Core 6 class library project which contains code shared by projects implementing Entity Framework Core data contexts. The project will include base classes for data contexts with that will implement a unified way to deal with transactions, concurrency, and audit requirements.

- Common.MediatR

  This is a .NET Core 6 class library project which contains code shared by projects implementing the CQRS pattern using MediatR. The project will include logging, validation, and transactional behaviors. Additionally, any MediatR related interfaces will be contained in this project as well.

- Common.Authorization

  This is a .NET Core 6 class library project which contains code shared by all projects that need access to authorization capabilities using permissions. The project will include types and interfaces used to define and evaluate authorization policies. Additionally, this project will implement the remote access required to interact with the authorization service by client applications like the API Application.

9.2. Services

- Api

  The Api Application will be broken down into multiple projects as follows:

  - Api

    This is an ASP.NET Core 6 project which will be responsible of hosting the Api Application and implement any required endpoints.

- Api.Application

  This is a .NET Core 6 class library project which will be responsible of implementing all the commands and queries that provide the required Api functionality.

- Api.Domain

  This is a .NET Core 6 class library project which will be responsible of implementing core domain functionality like entities, exceptions, and interfaces.

- Api.Infrastructure

  This is a .NET Core 6 class library project which will be responsible of implementing all the infrastructure concerns needed for the Api Application. This project will have data access components implemented using Entity Framework Core data contexts.

- Authorization

  This is an ASP.NET Core 6 project which will be responsible of hosting the Authorization Service, implement policy endpoints, and any policy storage related operations.

- Identity

  This is an ASP.NET Core 6 project which will be responsible of hosting the Identity Service Provider using Duende IdentityServer.

9.3. Web

- BFF

  This is an ASP.NET Core 6 project which will be responsible of hosting the Web Application BFF. The project will include the Single-Page Application, YARP proxy, and any account endpoints.

9.4. Test

- Unit Tests

  - Api.UnitTests

    This is a .NET Core 6 class library project which will be responsible of implementing all unit tests needed for the Api Application. This project will be using XUnit library to provide the testing infrastructure.

- Common.Authorization.UnitTests

  This is a .NET Core 6 class library project which will be responsible of implementing all unit tests needed for the Authorization common project. This project will be using XUnit library to provide the testing infrastructure.

- Integration tests
  - Api.IntegrationTests

    This is a .NET Core 6 class library project which will be responsible of implementing all integration tests needed for the Api Application. This project will be using XUnit library to provide the testing infrastructure.

  - Common.Authorization.IntegrationTests

    This is a .NET Core 6 class library project which will be responsible of implementing all integration tests needed for the Authotization engine shared code. This project will be using XUnit library to provide the testing infrastructure.

10. Standards

10.1. Coding Conventions

- Clear and informative code commit messages.
- Class names should always be in Pascal Case (ClassName).
- Method names should always be in Pascal Case (MethodName).
- Variable names should always be in Camel Case (variableName).
- Class fields should always be prefixed by _ (_field).
- Constants should always be in uppercase (CONSTANT).
- Properties should have no blank lines between them.
- Method definitions should be separated from property definitions by at least one blank line.
- Comments should always be on a separate line.
- Comments should always begin with an uppercase letter and end with a period.
- XML comments (///) should be used to describe code purpose and behavior.
- String interpolation should be used to concatenate short strings.
- StringBuilder objects should be used for large amounts of text and appending strings in loops.
- Implicit typing (var) for local variables should only be used when the right hand side of the assignment is obvious otherwise explicit typing should be used.
- If statements should be written where it short circuits when ever possible.
- Object instantiation should be unified across the code base and object initializers should be used where applicable. (var instance = new Class { Property = value }).
- Static members should always be called by using the class name even from within the class itself or any derived classes.

10.2. ASP.NET Core

- User input should never be trusted and needs to be validated.
- Validation for free text should always white list what is allowed using regular expressions for example allowing only alphabet characters.
- The ASP.NET Core 6 minimalist hosting configuration should be used in the Program class.
- The application entry method (Main) should only be concerned with creating the application builder and running it.
- Configuring the ASP.NET Core pipeline and dependency injection container should be implemented as extension methods inside a single class named HostingExtensions.
- The new ASP.NET Core 6 Minimal Api conversion should be used to map HTTP endpoints.
- Only authenticated users should be allowed to interact with the system endpoints.
- All endpoint implementations should use the Result class to return any required response.
- As a result of a successful Api read operation (GET), an OK 200 should be returned.
- As a result of a successful Api create operation (POST), a created 201 should be returned.
- As a result of a successful Api update operation (PUT/PATCH), a no content 204 should be returned.
- The result of a successful Api delete operation (DELETE), a no content 204 should be returned.
- Open redirect attacks should always be considered when when redirecting users to a new URL. The use of local redirects should be used when appropriate.
- CORS should not be enabled for the BFF as it is not needed (the SPA is hosted on the same domain).
- Cookies used in the system should have a name set with the prefix __Host as this will protect the cookie from being overwritten by a subdomain.
- Cookies should have SameSite set to strict.

- Cookies should always have an expiration time set.
- All caching requirements should be implemented using the IDistributedCache interface to make it easy to change from memory to a distributed cache solution like Redis.

10.3. Logging

- Serilog logging should be configured to handle all logging requirements of the WFS. This will give the WFS flexibility to configure multiple sinks as needed.
- All logging should use the Microsoft.Extensions.Logging.ILogger<T> interface.
- No concatenation should be used when logging values, rather use the provided formatting capabilities of the logger to keep the logged events as structured objects.
- Generate correlation id to group log entries per request as it accesses multiple services.
- User id and user name should be present on every log entry.
- Sensitive data like user passwords should never be logged (see ISecuritySensitive in section 6.1.1).
- The program entry should be wrapped with try catch finally block to make sure initialization exceptions happening before the ASP.NET pipeline takes control are logged properly and any pending logs are flushed to their sinks.
- UseExceptionHandler should be used as the catch all exception handler to intercept any unhandled exceptions and return proper HTTP status responses.
- Microsoft.AspNetCore events should be overridden to warning level to reduce logging clutter.
- Microsoft.EntityFrameworkCore events should be overridden to warning level to reduce logging clutter.

10.4. Testing

- Testing projects should start with the name of the project under test ending with the type of test being performed. Example, Api.UnitTests and Api.IntegrationTests.
- Individual tests (test method names) should use the following template ActionName_ExpectedOutCome_Condition.
- Test codes should follow the Arrange, Act, and Assert order.
- All testing projects should be utilizing the XUnit testing library.
- All mocking requirements that can not be implemented using a fake implementation should be using the Moq library to achieve that.
- Unit tests will be concerned with logic validation and algorithm results independent from infrastructure concerns. An example of that is to test validation rules in the system.
- Integration tests will be responsible of testing all the infrastructure concerns in the system. An example of that would be all database related operation.
- Database integration tests needs to verify the output of their operations and not just the successful execution.

10.5. Real Project Recommendation

- Authorization engine needs to implement persistent storage and management of both group membership and group permissions.
- Queries can benefit from the changing of the infrastructure implementation from using Entity Framework Core to using a micro ORM like Dapper (one of the benefits of a CQRS pattern).
- The Grafana portal should have more strict access requirements, the use of client side certificates for administrators would be a suitable solution.
- Communication between the different system parts should always be kept at a minimal for better scalability. Outside the authorization service calls and the BFF calls to the backend API, communication should be performed asynchronously (not the use of async/await in C#) through the use of a service bus to publish events or a shared table with a worker process that reads its entries. MassTransit is a great tool that provides an enterprise service bus implementation while abstracting the underlying infrastructure details.

- A domain model derived from the system requirements can be added before starting to detail the different views of the proposed system.
- Idempotent requests should be considered when any retry logic is present in the application.
- The application should count for multi instance scenarios when different services are scaled out. In particular, ASP.NET Core Data Protection and Caching needs to be configured to use a shared storage to support a distributed scenario. The use of Redis or a relational database can be one approach to solve this problem.
- gRPC can be used for inter-service communication instead of HTTP for better performance. gRPC will provide better response times when speed and responsiveness is a critical criteria for the application.