Table of contents

Overview

The purpose of this framework is to provide a clean and simple way to get started with CQRS (Command Query Responsibility Separation) and Event-Sourcing.

There are many ways to achieve this, but we specifically wanted to have to code align very closely to how you would model a process using event-modeling.

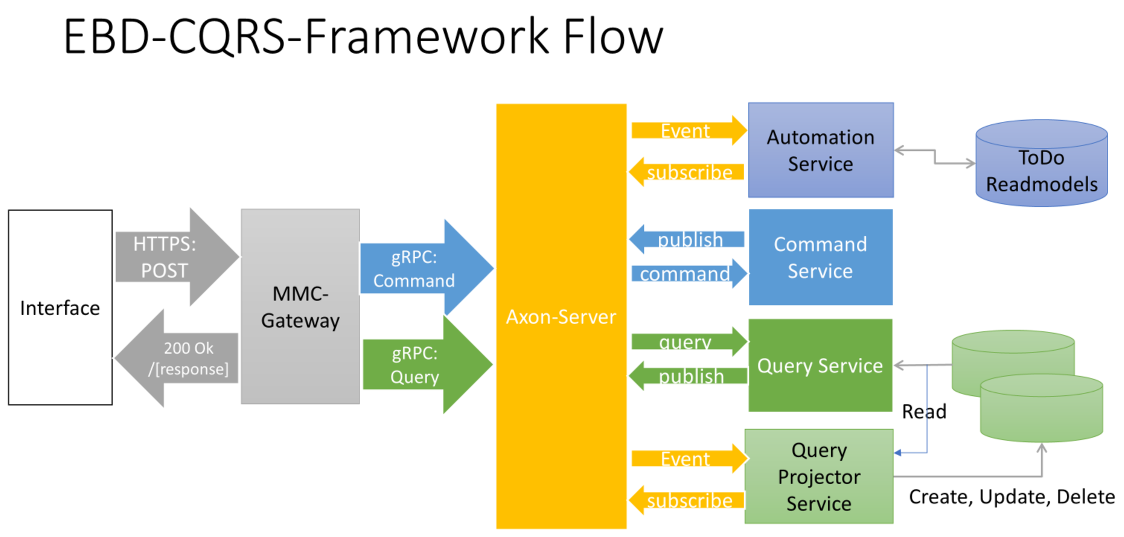


image.png

What does the framework provide?

The framework allows you to define, build and deploy 5 types of services:

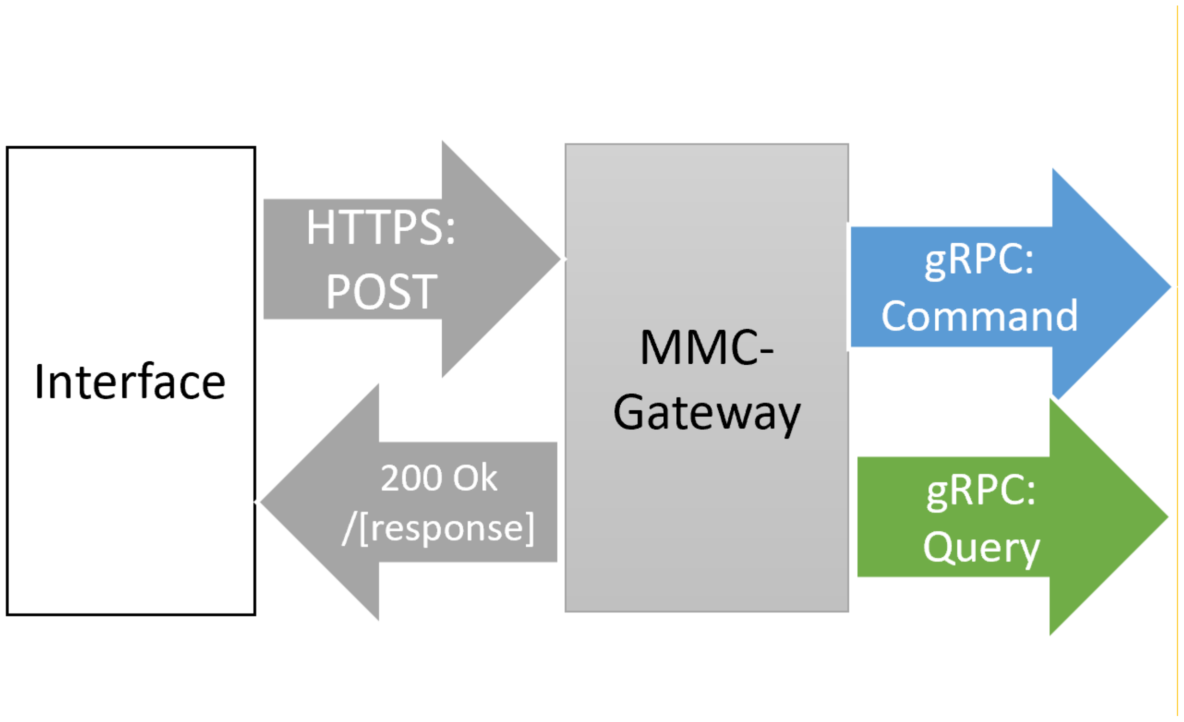
- A gateway service
- A command service
- A automation service
- A query service
- A query projection service

A key tool in setting up the framework is **NX** (https://www.nx.dev). This provides a mono-repo structure to the project and makes it very easy to expand projects while protecting the boundaries if each library.

The current version of the framework only supports **Axon-Server** (https://www.axoniq.io) as the event-store and message-bus. However, the framework has been set up to allow for different ES and message-bus solutions.

The Gateway Service

A minimal implementation of a single entry-point to all services is provided through the gateway-service.



gateway-service.png

The gateway is an express (https://expressjs.com/) server listening by default on **port: 3100.**

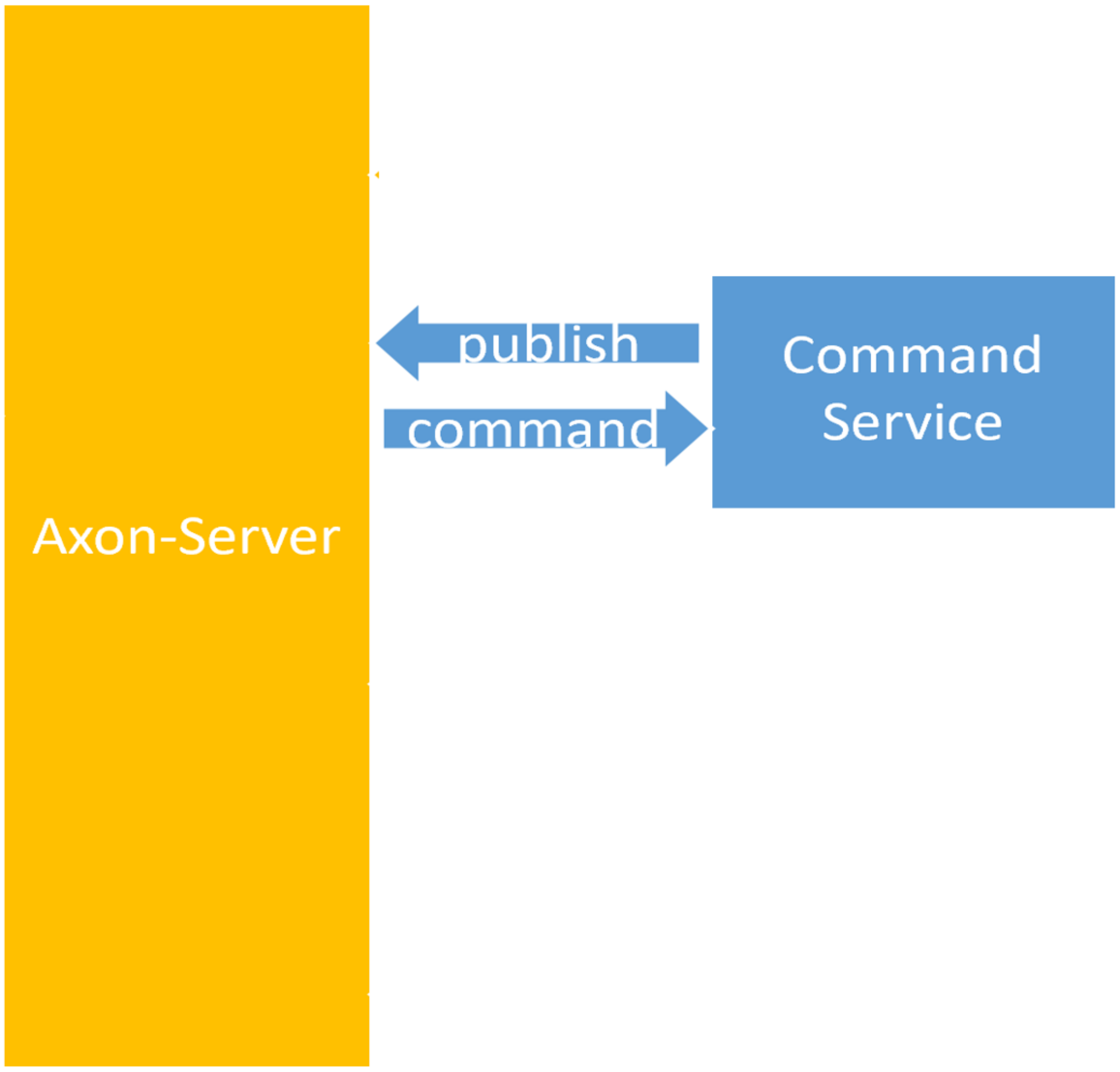
A client app can call a command or query using the following:

The gateway-service is a bare minimum implementation which only concern at the moment is mapping command and query requests to command bus and query bus of Axon Server.

⚠ **Authorisation** and other routing concerns should be added to the gateway.

The Command Service

The write side of CQRS is handled by the command service. Only the command service uses the concept of aggregates, as the command service is the only service that will actually change the state of a system. There are alternatives that do not use aggregates at all, but for now we prefer to use them as it allows nice way to enforce consistent use of applying state.



command-service.png

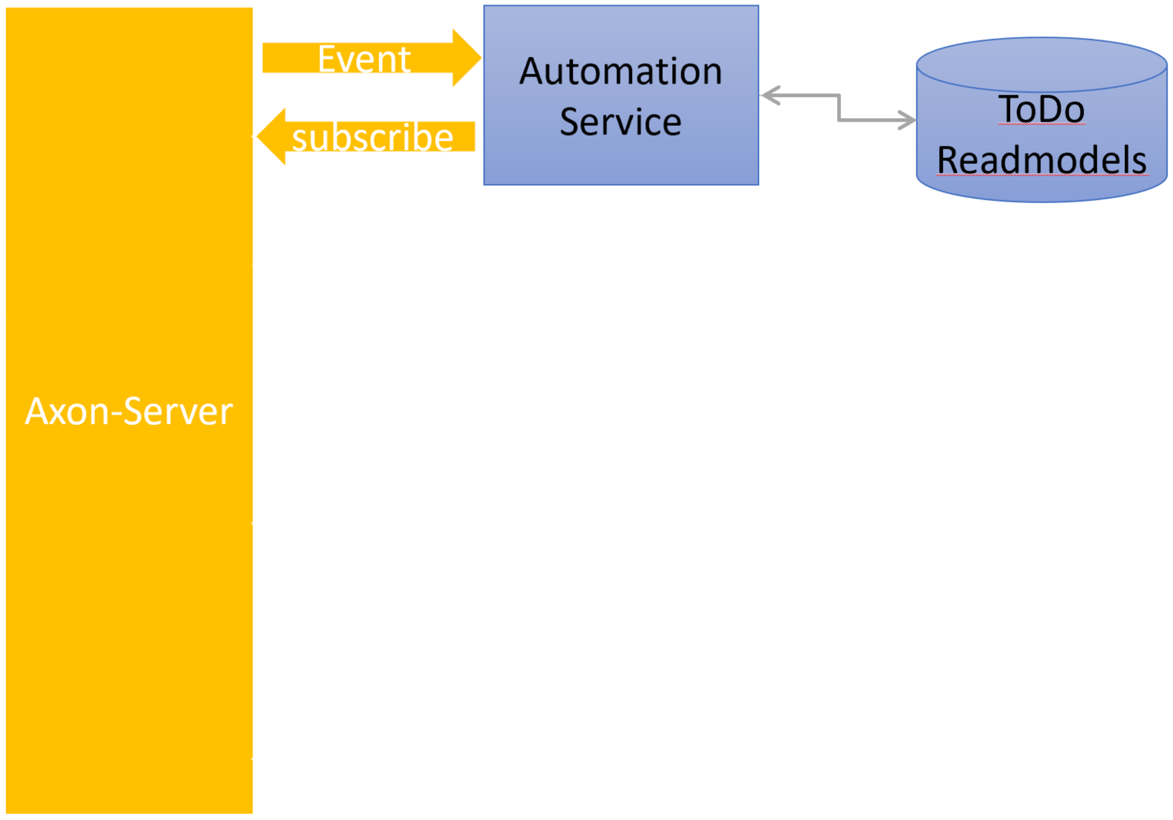
This service contains all functionality to implement:

- communicate with axon server using command and event bus.
- command handlers:

  - validate incoming commands (Anti Corruption Layer) for the application service/command handlers.
  - create or load (de-hydrate) an aggregate
  - inject additional services required by the domain layer
  - call aggregate function
  - persist and publish events
  - optional return response value
- aggregates:
  - execute the business rules that need to be checked to determine what event(s) should be applied.
  - return an array of events to the calling command handler
  - event sourcing handlers that are used when loading an aggregate to rebuild the current state of the aggregate.

The Automation Service

The automation service uses the approach of todo list processing instead of sagas.



automation-service.png

A automation process handler consists of 4 elements:

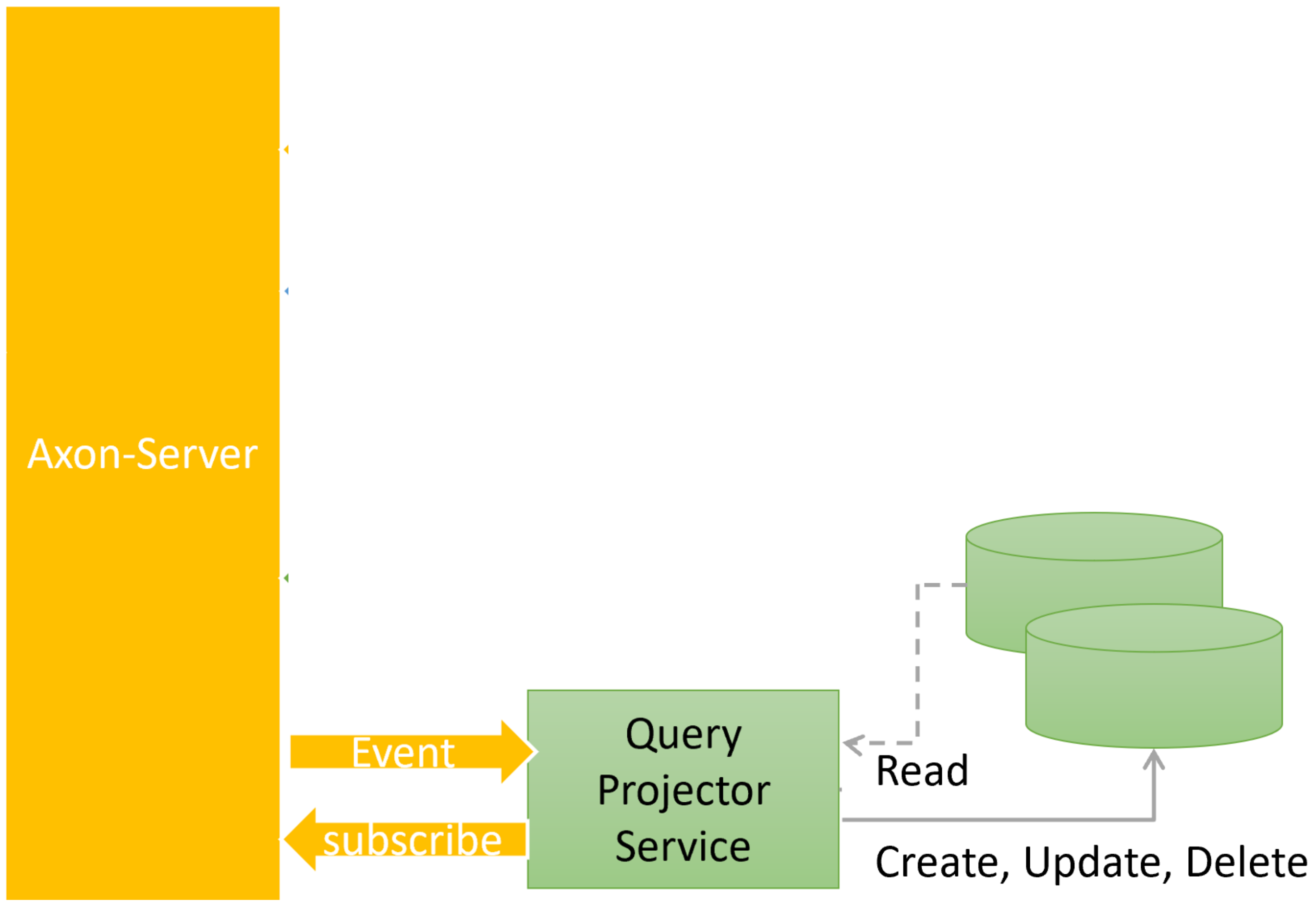
- The read-model which contains only just enough facts to make the decision when to send a command and provide the command with the required parameters.
- The business rules required to check if a command can be sent.
- Sending a command
- Event handlers that support:
  - Adding a new todo record containing the start state.
  - Updating the read-model and updating the record in the todo-list
  - Removing (completing) the record in the todo-list when command published event

successfully.

There can be multiple event handlers for each of the above. The challenge with automation is to have a common key in all subscribed events that we can use to identify the record in the todo-list.

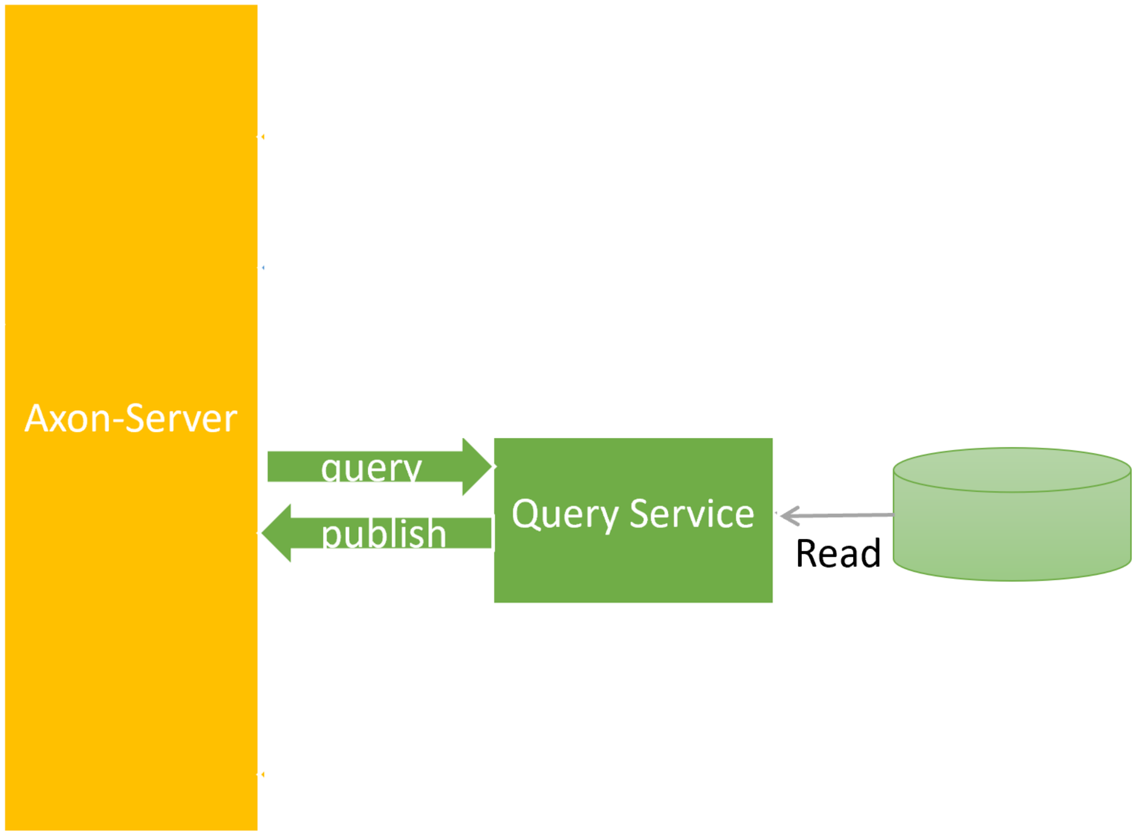
The Query Projector Service

Start typing here...



The Query Service

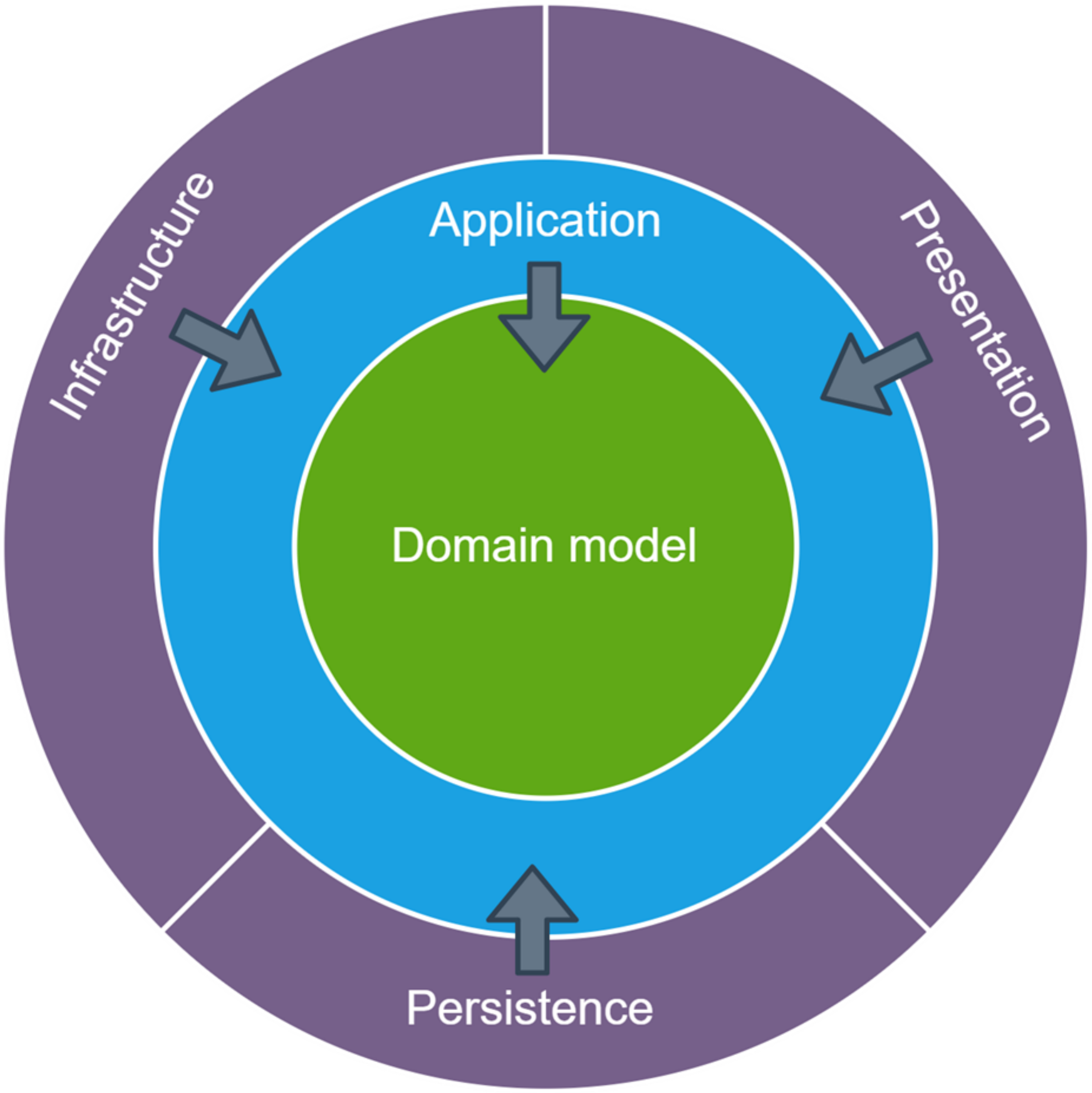
Start typing here...



query-service.png

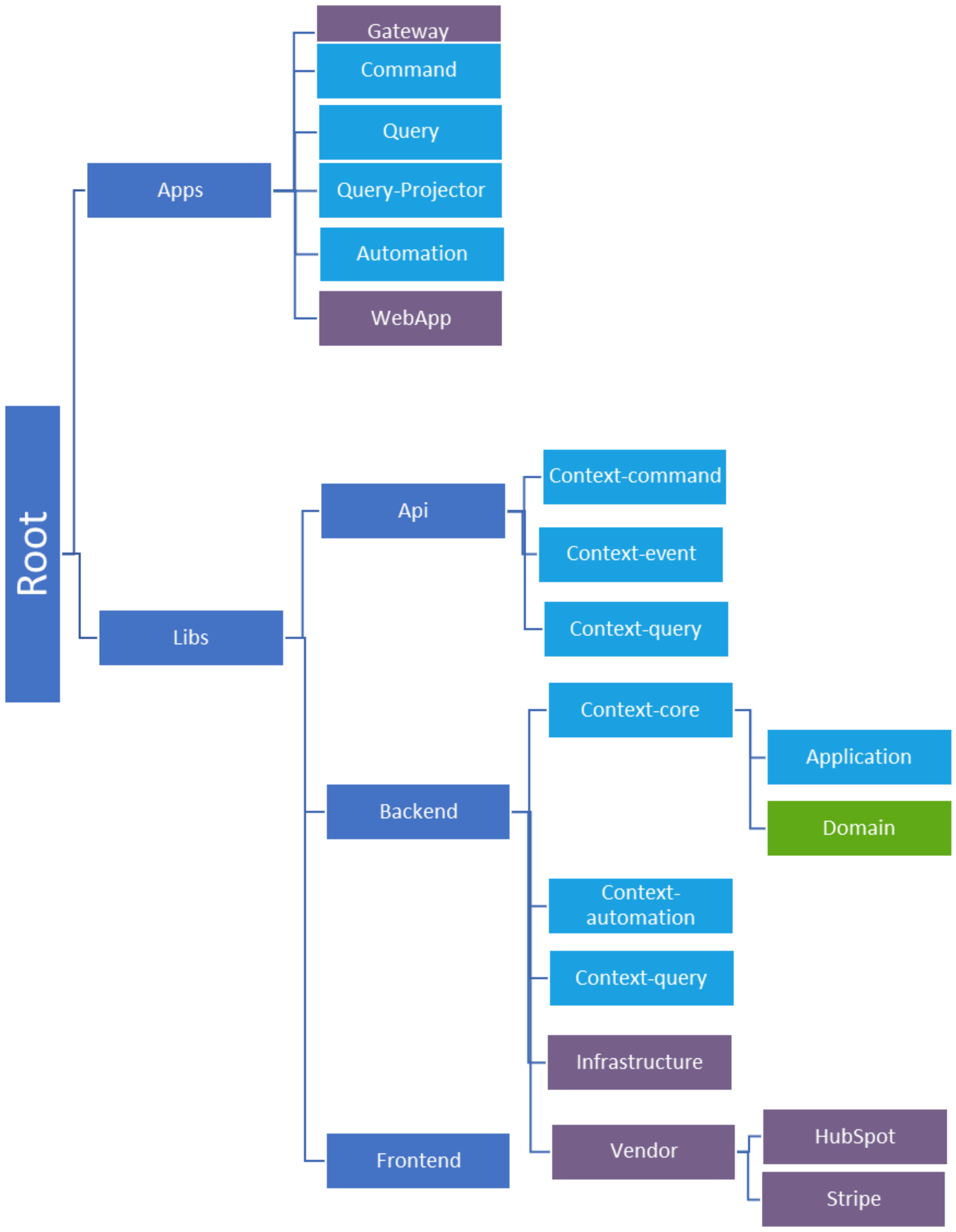
Project Structure

The project structure is based on following the onion-architecture approach. The separation between infrastructure, application and domain is strictly enforced. Each area will have its own projects and the separation is enforced by the linter.



onion_architecture.png

In addition to the onion structure we also take into account that different contexts should have no knowledge of eachother. This means we will have projects/folders repeated per instance of a context. Infrastructure libraries can be consumed throughout the project, but application layers and domain-layers are specific to contexts and should not be linked to other contexts. Only use events for inter-context communication!

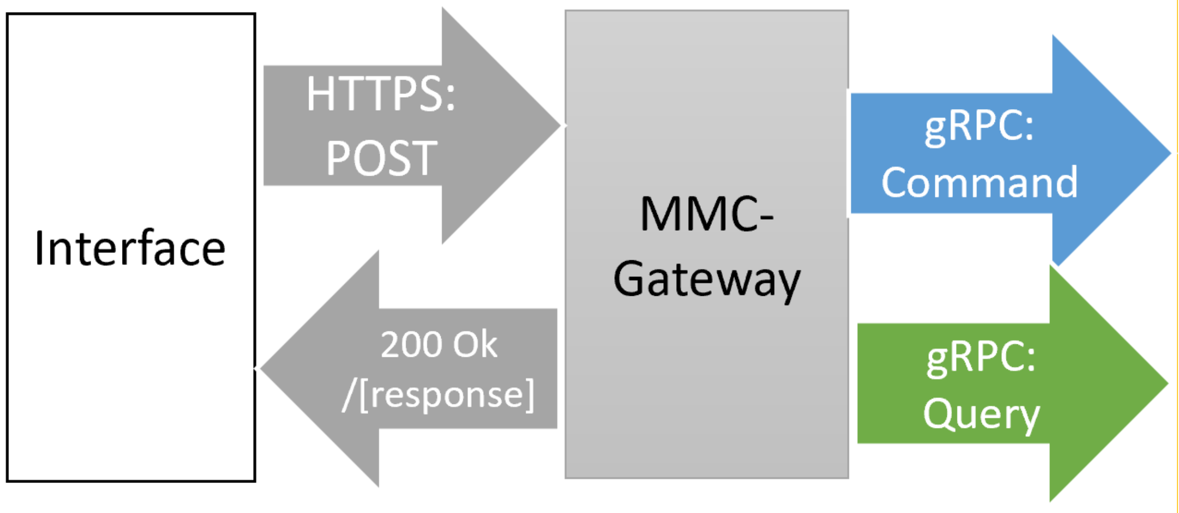


folder_structure.png

The lighter blue color is to indicate that these folders/files are all changed by a context. In the case of the apps it will be just registering the command/query and query-handlers. The apps/WebApp is to refer to one or more front-end applications. The principle of nx is that you have as little as possible in the actual application and push everything else into libraries.

The Gateway Service

A minimal implementation of a single entry-point to all services is provided through the gateway-service.



gateway-service.png

The gateway is an express (https://expressjs.com/) server listening by default on **port: 3100.**

A client app can call a command or query using the following:

The gateway-service is a bare minimum implementation which only concern at the moment is mapping command and query requests to command bus and query bus of Axon Server.

NodeJs

```
const requestOptions = {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
      command: props.command,
      payload: props.data,
  });
  //query example
  //body: JSON.stringify({
```

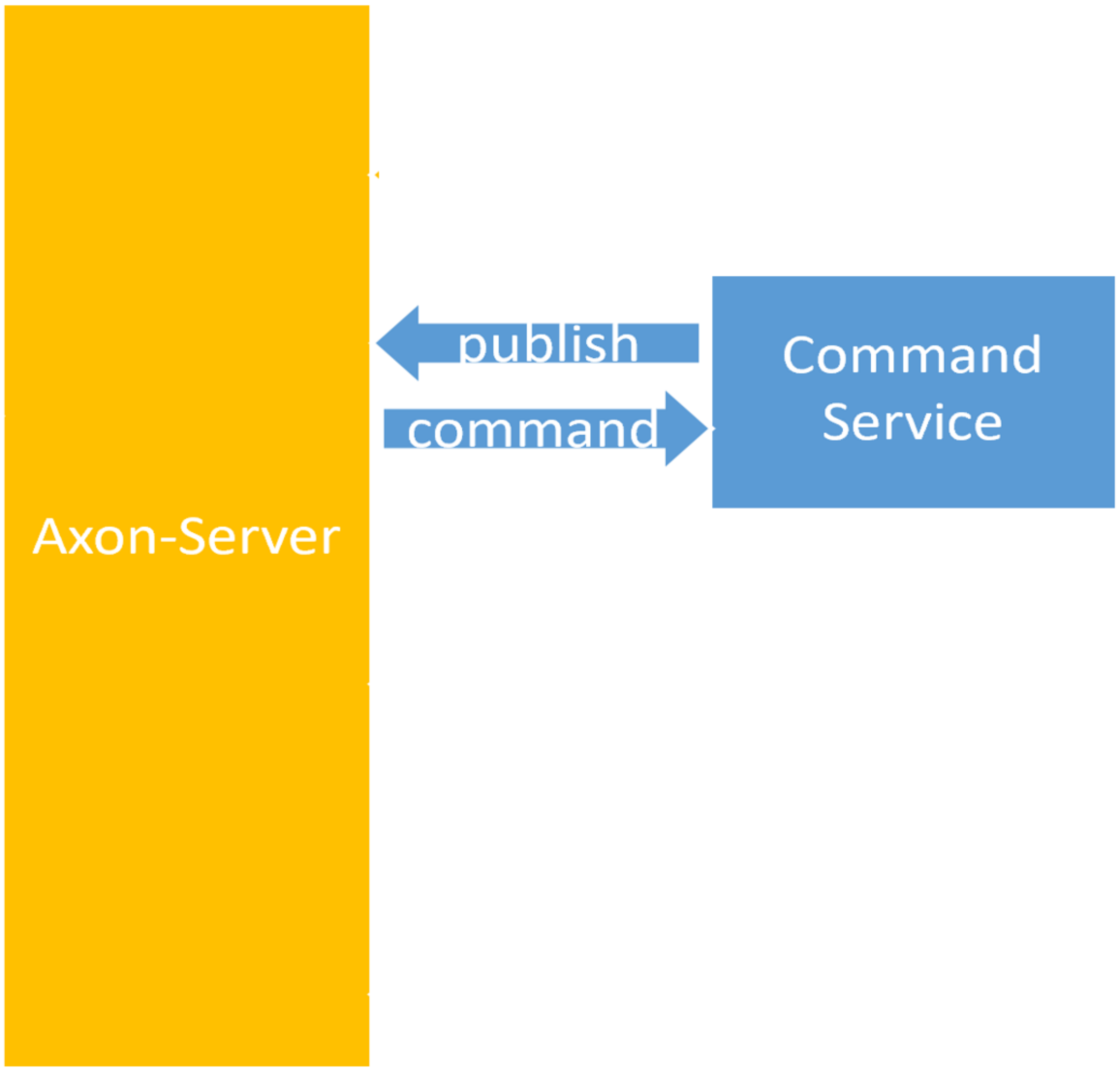
```
  //    query: props.query,
  //    payload: props.filter,
  //});
}

await fetch(props.gatewayUrl, requestOptions)
  .then(async (response) => {
    if (response.status ===200 && response.status ===201) {
      return response.json();
    }
    console.error(`Send command failed : ${response.statusText}`)
    return
  })
  .catch((error) =>
    console.error(`Send command failed : ${error}`),
  );
```

cURL

The Command Service

The write side of CQRS is handled by the command service. Only the command service uses the concept of aggregates, as the command service is the only service that will actually change the state of a system. There are alternatives that do not use aggregates at all, but for now we prefer to use them as it allows nice way to enforce consistent use of applying state.



command-service.png

API definition

First step in being able to use commands will be to define the commands. This is pretty straight-forward but we do have to take into account that we have two ways to define a command:

- The first way, and preferred way, is sending a command as send and forget. We are living in an async world and should not expect results back. The only response we should expect is that the command has been received and will be processed.
- The second ways is a so called **responding** command. The definition will include what will be returned after the command has successfully been executed.

Command definition

//libs/api/web-translator-command/src/translation/approve-translation-price.ts

```
import { command, Command } from '@ebd-connect/cqrs-framework';

@command('ApproveTranslationPrice')
export class ApproveTranslationPrice implements Command {
constructor(
public readonly translationId : string,
public readonly approved : boolean,
) {}

}
```

Responding command definition

//libs/api/web-translator-command/src/translation/submit-translation-request.ts

```
import { command, RespondingCommand } from '@ebd-connect/cqrs-framework';

@command('SubmitTranslationRequest')
export class SubmitTranslationRequest implements RespondingCommand {
  constructor(
    public readonly text : string,
    public readonly translationId : string,
) {}
  $responseType!: { translationId : string }
}
```

Command Handler

A command-handler is the core functionality of the write side of a CQRS architecture. Because of this the project it belongs to is **ContextName-core.** For example **web-translator-core.**

It belongs to the application layer and needs to be registered with the command-service. Command-handlers are organised per aggregate and are called **AggregateName**ApplicationService.

```
import { ApproveTranslationPrice } from '@api/web-translator-command';
import { DetermineTranslationPrice } from '@api/web-translator-command';
import { translatorService } from '@backend/infrastructure';
import { SubmitTranslationRequest } from '@api/web-translator-command';
import { Translation } from '../domain/translation';
import { CommandContext, commandHandler,CommandReturnType } from '@ebd-
connect/cqrs-framework';

export class TranslationApplicationService {
  @commandHandler({ name: 'ApproveTranslationPrice' })
  async approveTranslationPrice(command: ApproveTranslationPrice, {
eventSourcing }: CommandContext) {
    await eventSourcing.load(Translation, command.translationId,
(eventStream) =>
      eventStream.approveTranslationPrice(command )
    );
  }

  @commandHandler({ name: 'SubmitTranslationRequest' })
  async submitTranslationRequest(command: SubmitTranslationRequest, {
eventSourcing }: CommandContext):
CommandReturnType<SubmitTranslationRequest> {
    const translationId= command?.translationId ? command?.translationId :
crypto.randomUUID();

    await eventSourcing.create(Translation, translationId, (eventStream) =>
eventStream.submitTranslationRequest(translationId, command)
  );
  return { translationId };
  }
}
```

To provide a clean way to register the command handlers we use a separate AggregateConfig

```
import { TranslationApplicationService } from './translation-application-
service';
import { Type } from '@ebd-connect/cqrs-framework';

export const webTranslatorCommandHandlers: Type[] = [
  //services
  TranslationApplicationService,
];
```

This is what we use in the command service to register the command handlers. // Apps/mmc-command/src/main.ts

```
import { webTranslatorCommandHandlers } from '@backend/web-translator-core';

import {
  AxonApplication,
  ClientIdentification,
  configLogger,
  credentials,
} from '@ebd-connect/cqrs-framework';

const isProduction = false;
configLogger();

const axonConnector = new AxonApplication({
  commandHandlers: [
    // contextCommandHandlers
    ...webTranslatorCommandHandlers,
  ],
  connection: {
    serviceClientInit: {
      address: process.env.AXON_HOST ?? 'localhost:8124',
      credentials: credentials.createInsecure(),
    },
    clientIdentification: new ClientIdentification()
      .setComponentName('mmc-command')
      .setClientId(isProduction ? crypto.randomUUID() : 'local'),
    forceStayOnSameConnection: !isProduction,
  },
});
axonConnector.connect().catch((error) => console.error(error.message));
```

The CommandHandler

Start typing here...

Application Layer

Start typing here...

Domain Layer

Start typing here...