ADRs: + investigative project

Choosing:

1. Project Architecture
2. Tool for build
3. Project folder structure
4. The architecture of one sharing feature
5. Design patterns and best practices
6. Language: Javascript, Typescript, flow or combined
7. Package manager
8. State manager
9. Lint, code style
10. Monitoring, log manager
11. Error Monitoring
12. Apis, communicate with a server
13. Routing
14. Ui kit
15. Charts
16. Css architecture
17. Authentications, authorization
18. Documentation (storybook, kitchen sink, …)
19. Helper libraries (Date, hooks, …) []
20. Testing (libraries, strategies) []

Project architecture

Status: Proposed

Context:

Our current project dashboard frontend is built with React. We need to extend this application to support multiple dashboards (two or three) that can share configurations, components, logic, and features. We are considering three architectural approaches to achieve this: Monorepo, Microfrontend, and Monolithic Application. This ADR evaluates the pros and cons of each approach to determine the best fit for our requirements.

Decision:

We will compare the following architectural approaches:

1. **Monorepo**: A single repository containing multiple projects and libraries.
2. **Microfrontend**: Each dashboard as a separate frontend application.
3. **Monolithic Application**: All dashboards within a single application.

Monorepo Approach:

Pros:

- **Code Reusability**: Shared configurations, components, and logic are easily accessible.
- **Simplified Dependency Management**: Dependencies are managed in a single repository.
- **Consistency**: Easier to maintain a consistent development environment and coding standards.
- **Collaborative Development**: Improved collaboration as all code is in one place.

Cons:

- **Repository Size**: The repository can become large and unwieldy.
- **Complexity**: Managing multiple projects within a single repository can add complexity.
- **Build Times**: Can lead to longer build times as the repository grows.

Microfrontend Approach:

Pros:

- **Independence**: Each dashboard can be developed, deployed, and scaled independently.
- **Modularity**: Clear separation of concerns between different dashboards.
- **Scalability**: Easier to scale specific parts of the application as needed.
- **Flexibility**: Different technologies and frameworks can be used for different dashboards.

Cons:

- **Complex Deployment**: Managing deployments of multiple frontends can be complex.
- **Inter-application Communication**: Handling communication and shared state between dashboards can be challenging.
- **Performance Overhead**: Potential performance overhead due to multiple frontend applications.

Monolithic Application Approach:

Pros:

- **Simplicity**: Simple project structure and build configuration.
- **Single Deployment**: Only one deployment process to manage.

- **Shared Resources**: Shared configurations, components, and logic are naturally available.

Cons:

- **Scalability Issues**: Harder to scale specific parts of the application independently.
- **Codebase Size**: The codebase can become large and difficult to manage.
- **Maintenance Challenges**: Maintaining a large monolithic codebase can be cumbersome.
- **Coupling**: High degree of coupling between different parts of the application.

Consequences:

- **Positive**:
    - **Monorepo**: Facilitates collaboration and code reuse, ensuring consistency across the project.
    - **Microfrontend**: Provides flexibility and scalability, allowing independent development and deployment.
    - **Monolithic**: Simplifies deployment and shared resource management.
- **Negative**:
    - **Monorepo**: May face issues with repository size and build times as the project grows.
    - **Microfrontend**: Introduces complexity in deployment and inter-application communication.
    - **Monolithic**: Scalability and maintenance can become significant challenges.

Decision:

After evaluating the pros and cons, we have decided to adopt the **Monorepo** approach. This approach provides a balance between code reuse, consistency, and collaborative development, aligning with our project goals and future expansion plans. While it may

introduce some complexity in managing the repository, the benefits of shared resources and simplified dependency management outweigh the drawbacks.

Alternatives Considered:

1. **Microfrontend**: Offers high flexibility and scalability but introduces complexity in deployment and inter-application communication.
2. **Monolithic**: Simplifies deployment but poses significant challenges in scalability and maintenance as the project grows.

Tool for build

Status: Proposed

Context:

We have adopted a monorepo approach for our React-based project dashboard, which will include multiple dashboards sharing configurations, components, logic, and features. To efficiently manage our monorepo, we need to select a suitable tool. The options we are considering are Turborepo, Lerna, and Nx.

Decision:

We will compare Turborepo, Lerna, and Nx based on various criteria such as features, performance, ease of use, community support, and scalability.

Turborepo

Pros:

1. **High Performance**: Built with performance in mind, Turborepo provides fast build times and efficient task execution.
2. **Incremental Builds**: Supports incremental builds, reducing build times by reusing previous build results.
3. **Modern Features**: Supports advanced features like parallel execution and caching.
4. **Ease of Use**: Simple setup and configuration, with a focus on developer experience.

Cons:

1. **Newer Tool**: Being relatively new, it may have fewer community resources and plugins compared to more established tools.
2. **Smaller Ecosystem**: Smaller ecosystem and potentially fewer integrations with other tools.

Lerna

Pros:

1. **Mature and Established**: Lerna is a well-established tool with a large community and extensive documentation.
2. **Flexible**: Provides flexible configuration options for managing dependencies and running scripts across packages.
3. **Widely Used**: Widely adopted in the industry, ensuring good community support and integration with other tools.
4. **Monorepo Management**: Efficiently manages dependencies and scripts in a monorepo, making it easy to work with multiple packages.

Cons:

1. **Performance**: May not be as performant as newer tools like Turborepo and Nx, especially for larger projects.
2. **Lack of Advanced Features**: Lacks some of the advanced features and optimizations found in Turborepo and Nx.

3. **Complexity**: Can become complex to configure and manage for larger monorepos.

Nx

Pros:

1. **Comprehensive Tooling**: Nx offers a complete suite of tools for monorepo management, including dependency graph visualization, caching, and code generation.
2. **Performance**: High performance with features like incremental builds, parallel execution, and distributed task execution.
3. **Modular Structure**: Encourages a modular approach, making it easier to manage large projects.
4. **Extensible**: Highly extensible with a rich plugin ecosystem and support for custom plugins.
5. **Community and Support**: Strong community support and extensive documentation.

Cons:

1. **Steeper Learning Curve**: More features and options can result in a steeper learning curve for new users.
2. **Complexity**: Can be overkill for smaller projects due to its comprehensive feature set.

Consequences:

- **Positive**:
    - **Turborepo**: Fast build times and modern features enhance developer productivity.
    - **Lerna**: Mature and flexible, with strong community support and integrations.
    - **Nx**: Comprehensive tooling and performance optimizations support large and complex projects.
- **Negative**:

    - **Turborepo**: Smaller ecosystem and fewer resources may limit integration options.
    - **Lerna**: Potential performance issues and complexity in managing large projects.
    - **Nx**: Steeper learning curve and potentially overkill for smaller projects.

Decision:

After evaluating the pros and cons, we have decided to use **Nx** for managing our monorepo. Nx provides comprehensive tooling, high performance, and strong community support, making it well-suited for our project's needs and future growth. Although it has a steeper learning curve, the benefits of its advanced features and extensibility outweigh the initial complexity.

Alternatives Considered:

1. **Turborepo**: Fast and modern but limited by a smaller ecosystem and fewer integrations.
2. **Lerna**: Mature and flexible but may face performance and complexity issues for larger projects.

Project folder structure

Status: Proposed

Context:

We have decided to adopt Nx for managing our monorepo for the React-based project dashboard. To ensure maintainability, scalability, and ease of development, we need to define a clear and organized folder structure that leverages Nx's features.

Decision:

We will implement the following folder structure for our Nx monorepo:

css
Copy code

```
root
├── apps
│   ├── dashboard-one
│   │   ├── src
│   │   └── project.json
│   ├── dashboard-two
│   │   ├── src
│   │   └── project.json
│   └── dashboard-core
│       ├── src
│       └── project.json
├── libs
│   ├── shared
│   │   ├── components
│   │   │   ├── src
│   │   │   └── index.ts
│   │   ├── utils
│   │   │   ├── src
│   │   │   └── index.ts
│   │   └── styles
│   │       ├── src
│   │       └── index.ts
│   └── another-shared-lib
│       ├── src
│       └── index.ts
```

```
├── tools
│   ├── scripts
│   ├── generators
│   └── executors
├── nx.json
├── workspace.json
├── package.json
├── tsconfig.base.json
└── .gitignore
```

Description:

1. **root/apps**: Contains individual applications.
    - **dashboard-one**: Specific implementation for Dashboard One.
        - **src**: Source code for Dashboard One.
        - **project.json**: Nx configuration for Dashboard One.
    - **dashboard-two**: Specific implementation for Dashboard Two.
        - **src**: Source code for Dashboard Two.
        - **project.json**: Nx configuration for Dashboard Two.
    - **dashboard-core**: Core functionality and shared logic that are specific to dashboards.
        - **src**: Source code for Dashboard Core.
        - **project.json**: Nx configuration for Dashboard Core.
2. **root/libs**: Contains shared libraries that can be used across multiple applications.
    - **shared**: Libraries that are shared across the monorepo.
        - **components**: Shared UI components.
            - **src**: Source code for shared components.
            - **index.ts**: Entry point for shared components.
        - **utils**: Shared utility functions.
            - **src**: Source code for shared utilities.

            - **index.ts**: Entry point for shared utilities.
        - **styles**: Shared styles and themes.
            - **src**: Source code for shared styles.
            - **index.ts**: Entry point for shared styles.
    - **another-shared-lib**: Another example of a shared library.
        - **src**: Source code for this shared library.
        - **index.ts**: Entry point for this shared library.
3. **root/tools**: Custom scripts, generators, and executors for Nx.
    - **scripts**: Custom scripts for automation.
    - **generators**: Custom code generators.
    - **executors**: Custom executors for Nx tasks.
4. **root/nx.json**: Nx configuration file.
5. **root/workspace.json**: Nx workspace configuration.
6. **root/package.json**: Root package.json for managing root-level dependencies and scripts.
7. **root/tsconfig.base.json**: Base TypeScript configuration for the entire monorepo.
8. **root/.gitignore**: Git ignore file for the monorepo.

Pros:

1. **Organization**: Clear separation of applications and shared libraries, enhancing code organization.
2. **Reusability**: Shared libraries promote code reuse, reducing duplication and maintenance overhead.
3. **Scalability**: New applications and libraries can be added easily, supporting project scalability.
4. **Maintainability**: Centralized configurations and scripts improve maintainability and consistency.
5. **Nx Features**: Leveraging Nx's powerful features like dependency graph visualization, caching, and code generation.

Cons:

1. **Initial Complexity**: Initial setup of the Nx monorepo and folder structure can be complex.
2. **Learning Curve**: Developers need to learn Nx-specific concepts and configurations.
3. **Dependency Management**: Managing dependencies across multiple libraries and applications requires careful attention.

Consequences:

- **Positive**:
    - Improved code organization, reusability, and maintainability.
    - Easier to scale and add new applications and libraries.
    - Consistent development environment and processes leveraging Nx's powerful features.
- **Negative**:
    - Potentially increased complexity in the initial setup.
    - Steeper learning curve for developers new to Nx.

Decision:

We will proceed with the proposed folder structure for our Nx monorepo. This structure provides a good balance between organization, scalability, and maintainability, leveraging Nx's features to support our project's growth and future expansion.

Alternatives Considered:

1. **Flat Structure**: All applications and libraries at the root level.
    - **Pros**: Simpler initial setup.
    - **Cons**: Poor organization, difficult to scale and maintain.
2. **Hierarchical Structure**: Nested applications and libraries within specific feature directories.
    - **Pros**: Clear separation by feature.
    - **Cons**: Increased complexity and potential for deep nesting.

The architecture of one sharing feature

Status: Proposed

Context:

We have decided to adopt Nx for managing our monorepo for the React-based project dashboard. To ensure maintainability, scalability, and ease of development, we need to define a clear and organized folder structure that leverages Nx's features.

Decision:

We will implement the following folder structure for our Nx monorepo:

css
Copy code

```
root
├── apps
│   ├── dashboard-one
│   │   ├── src
│   │   └── project.json
│   ├── dashboard-two
│   │   ├── src
│   │   └── project.json
│   └── dashboard-core
│       ├── src
│       └── project.json
├── libs
│   ├── shared
```

```
│   │   ├── components
│   │   │   ├── src
│   │   │   └── index.ts
│   │   ├── utils
│   │   │   ├── src
│   │   │   └── index.ts
│   │   └── styles
│   │       ├── src
│   │       └── index.ts
│   └── another-shared-lib
│       ├── src
│       └── index.ts
├── tools
│   ├── scripts
│   ├── generators
│   └── executors
├── nx.json
├── workspace.json
├── package.json
├── tsconfig.base.json
└── .gitignore
```

Description:

1. **root/apps**: Contains individual applications.
    - **dashboard-one**: Specific implementation for Dashboard One.
        - **src**: Source code for Dashboard One.
        - **project.json**: Nx configuration for Dashboard One.
    - **dashboard-two**: Specific implementation for Dashboard Two.
        - **src**: Source code for Dashboard Two.

        - **project.json**: Nx configuration for Dashboard Two.
    - **dashboard-core**: Core functionality and shared logic that are specific to dashboards.
        - **src**: Source code for Dashboard Core.
        - **project.json**: Nx configuration for Dashboard Core.
2. **root/libs**: Contains shared libraries that can be used across multiple applications.
    - **shared**: Libraries that are shared across the monorepo.
        - **components**: Shared UI components.
            - **src**: Source code for shared components.
            - **index.ts**: Entry point for shared components.
        - **utils**: Shared utility functions.
            - **src**: Source code for shared utilities.
            - **index.ts**: Entry point for shared utilities.
        - **styles**: Shared styles and themes.
            - **src**: Source code for shared styles.
            - **index.ts**: Entry point for shared styles.
    - **another-shared-lib**: Another example of a shared library.
        - **src**: Source code for this shared library.
        - **index.ts**: Entry point for this shared library.
3. **root/tools**: Custom scripts, generators, and executors for Nx.
    - **scripts**: Custom scripts for automation.
    - **generators**: Custom code generators.
    - **executors**: Custom executors for Nx tasks.
4. **root/nx.json**: Nx configuration file.
5. **root/workspace.json**: Nx workspace configuration.
6. **root/package.json**: Root package.json for managing root-level dependencies and scripts.
7. **root/tsconfig.base.json**: Base TypeScript configuration for the entire monorepo.
8. **root/.gitignore**: Git ignore file for the monorepo.

Pros:

1. **Organization**: Clear separation of applications and shared libraries, enhancing code organization.
2. **Reusability**: Shared libraries promote code reuse, reducing duplication and maintenance overhead.
3. **Scalability**: New applications and libraries can be added easily, supporting project scalability.
4. **Maintainability**: Centralized configurations and scripts improve maintainability and consistency.
5. **Nx Features**: Leveraging Nx's powerful features like dependency graph visualization, caching, and code generation.

Cons:

1. **Initial Complexity**: Initial setup of the Nx monorepo and folder structure can be complex.
2. **Learning Curve**: Developers need to learn Nx-specific concepts and configurations.
3. **Dependency Management**: Managing dependencies across multiple libraries and applications requires careful attention.

Consequences:

- **Positive**:
    - Improved code organization, reusability, and maintainability.
    - Easier to scale and add new applications and libraries.
    - Consistent development environment and processes leveraging Nx's powerful features.
- **Negative**:
    - Potentially increased complexity in the initial setup.
    - Steeper learning curve for developers new to Nx.

Decision:

We will proceed with the proposed folder structure for our Nx monorepo. This structure provides a good balance between

organization, scalability, and maintainability, leveraging Nx's features to support our project's growth and future expansion.

Alternatives Considered:

1. **Flat Structure**: All applications and libraries at the root level.
    - **Pros**: Simpler initial setup.
    - **Cons**: Poor organization, difficult to scale and maintain.
2. **Hierarchical Structure**: Nested applications and libraries within specific feature directories.
    - **Pros**: Clear separation by feature.
    - **Cons**: Increased complexity and potential for deep nesting.

Status: Proposed

Context:

We have decided to adopt Nx for managing our monorepo for the React-based project dashboard. To ensure maintainability, scalability, and ease of development, we need to define a clear and organized folder structure. One approach is to organize the folder structure based on features. This will group related code together, making it easier to manage and understand.

Decision:

We will implement the following feature-based folder structure for our Nx monorepo:

css
Copy code

```
root
├── apps
│   ├── dashboard-one
│   │   ├── src
│   │   └── project.json
│   ├── dashboard-two
│   │   ├── src
│   │   └── project.json
│   └── dashboard-core
│       ├── src
│       └── project.json
├── libs
│   ├── feature-xyz
│   │   ├── components
│   │   │   ├── src
│   │   │   └── index.ts
│   │   ├── services
│   │   │   ├── src
│   │   │   └── index.ts
│   │   ├── hooks
│   │   │   ├── src
│   │   │   └── index.ts
│   │   └── utils
│   │       ├── src
│   │       └── index.ts
│   ├── feature-abc
```

```
|   |   ├── components
|   |   |   ├── src
|   |   |   └── index.ts
|   |   ├── services
|   |   |   ├── src
|   |   |   └── index.ts
|   |   ├── hooks
|   |   |   ├── src
|   |   |   └── index.ts
|   |   └── utils
|   |       ├── src
|   |       └── index.ts
|   └── shared
|       ├── components
|       |   ├── src
|       |   └── index.ts
|       ├── services
|       |   ├── src
|       |   └── index.ts
|       ├── hooks
|       |   ├── src
|       |   └── index.ts
|       └── utils
|           ├── src
|           └── index.ts
├── tools
|   ├── scripts
|   ├── generators
|   └── executors
├── nx.json
```

```
├── workspace.json
├── package.json
├── tsconfig.base.json
└── .gitignore
```

Description:

1. **root/apps**: Contains individual applications.
    - **dashboard-one**: Specific implementation for Dashboard One.
        - **src**: Source code for Dashboard One.
        - **project.json**: Nx configuration for Dashboard One.
    - **dashboard-two**: Specific implementation for Dashboard Two.
        - **src**: Source code for Dashboard Two.
        - **project.json**: Nx configuration for Dashboard Two.
    - **dashboard-core**: Core functionality and shared logic that are specific to dashboards.
        - **src**: Source code for Dashboard Core.
        - **project.json**: Nx configuration for Dashboard Core.
2. **root/libs**: Contains feature-based libraries.
    - **feature-xyz**: Code related to the XYZ feature.
        - **components**: UI components specific to the XYZ feature.
            - **src**: Source code for XYZ components.
            - **index.ts**: Entry point for XYZ components.
        - **services**: Services and business logic related to the XYZ feature.
            - **src**: Source code for XYZ services.
            - **index.ts**: Entry point for XYZ services.
        - **hooks**: Custom hooks specific to the XYZ feature.
            - **src**: Source code for XYZ hooks.
            - **index.ts**: Entry point for XYZ hooks.
        - **utils**: Utility functions specific to the XYZ feature.

            - **src**: Source code for XYZ utils.
            - **index.ts**: Entry point for XYZ utils.
    - **feature-abc**: Code related to the ABC feature.
        - **components**: UI components specific to the ABC feature.
            - **src**: Source code for ABC components.
            - **index.ts**: Entry point for ABC components.
        - **services**: Services and business logic related to the ABC feature.
            - **src**: Source code for ABC services.
            - **index.ts**: Entry point for ABC services.
        - **hooks**: Custom hooks specific to the ABC feature.
            - **src**: Source code for ABC hooks.
            - **index.ts**: Entry point for ABC hooks.
        - **utils**: Utility functions specific to the ABC feature.
            - **src**: Source code for ABC utils.
            - **index.ts**: Entry point for ABC utils.
    - **shared**: Code that is shared across multiple features.
        - **components**: Shared UI components.
            - **src**: Source code for shared components.
            - **index.ts**: Entry point for shared components.
        - **services**: Shared services and business logic.
            - **src**: Source code for shared services.
            - **index.ts**: Entry point for shared services.
        - **hooks**: Shared custom hooks.
            - **src**: Source code for shared hooks.
            - **index.ts**: Entry point for shared hooks.
        - **utils**: Shared utility functions.
            - **src**: Source code for shared utils.
            - **index.ts**: Entry point for shared utils.
3. **root/tools**: Custom scripts, generators, and executors for Nx.
    - **scripts**: Custom scripts for automation.
    - **generators**: Custom code generators.
    - **executors**: Custom executors for Nx tasks.
4. **root/nx.json**: Nx configuration file.

5. **root/workspace.json**: Nx workspace configuration.
6. **root/package.json**: Root package.json for managing root-level dependencies and scripts.
7. **root/tsconfig.base.json**: Base TypeScript configuration for the entire monorepo.
8. **root/.gitignore**: Git ignore file for the monorepo.

Pros:

1. **Organization**: Clear separation of code by feature, making it easier to manage and understand.
2. **Reusability**: Shared libraries promote code reuse, reducing duplication and maintenance overhead.
3. **Scalability**: New features can be added easily, supporting project scalability.
4. **Maintainability**: Centralized configurations and scripts improve maintainability and consistency.
5. **Feature Isolation**: Isolating features reduces the risk of unintended side effects between different parts of the codebase.

Cons:

1. **Initial Complexity**: Initial setup of the feature-based folder structure can be complex.
2. **Learning Curve**: Developers need to understand the feature-based organization and Nx-specific configurations.
3. **Dependency Management**: Managing dependencies across multiple features requires careful attention.

Consequences:

- **Positive**:
    - Improved code organization, reusability, and maintainability.
    - Easier to scale and add new features.

    - Consistent development environment and processes leveraging Nx's powerful features.
    - Reduced risk of unintended side effects through feature isolation.
- **Negative**:
    - Potentially increased complexity in the initial setup.
    - Steeper learning curve for developers new to Nx and feature-based organization.

Decision:

We will proceed with the proposed feature-based folder structure for our Nx monorepo. This structure provides a good balance between organization, scalability, and maintainability, leveraging Nx's features to support our project's growth and future expansion.

Alternatives Considered:

1. **Package-Based Structure**: Organizing by packages instead of features.
    - **Pros**: Simpler initial setup.
    - **Cons**: Poor organization, difficult to scale and maintain.
2. **Domain-Based Structure**: Organizing by business domains.
    - **Pros**: Aligns with business logic.
    - **Cons**: Increased complexity and potential for deep nesting.

Design patterns and best practices

Status: Proposed

Context:

We have adopted Nx for managing our monorepo for the React-based project dashboard. To ensure the project is scalable, maintainable, and efficient, we need to establish best practices and patterns that all developers should follow. These practices will help in maintaining code quality, consistency, and development efficiency across the entire monorepo.

Decision:

We will implement the following best practices and patterns in our Nx monorepo project:

Code Organization and Structure

1. **Feature-Based Folder Structure**:
    - Organize code by feature to enhance maintainability and scalability.
    - Each feature should have its own directory within the `libs` folder.
2. **Shared Libraries**:
    - Create shared libraries for reusable components, services, hooks, and utilities.
    - Use `libs/shared` for common functionality used across multiple features.

Coding Standards

1. **Consistent Coding Style**:
    - Enforce consistent coding styles using Prettier and ESLint.
    - Add configuration files for Prettier and ESLint in the root directory.
2. **TypeScript**:
    - Use TypeScript for type safety and improved code quality.
    - Ensure all TypeScript configurations are centralized in `tsconfig.base.json`.

State Management

1. **Redux/Context API**:
    - Use Redux or Context API for state management.
    - Organize state management logic within each feature or shared library as needed.

Component Design

1. **Presentational and Container Components**:
    - Separate presentational (dumb) components from container (smart) components.
    - Place presentational components in `components` and container components in `containers`.
2. **Reusable Components**:
    - Develop reusable UI components and place them in `libs/shared/components`.
    - Ensure components are well-documented and tested.

Testing

1. **Unit Testing**:
    - Write unit tests for all components, services, and utilities.
    - Use Jest for unit testing and place test files alongside the source files.
2. **End-to-End Testing**:
    - Implement end-to-end tests using Cypress.
    - Organize e2e tests in the `apps` directory corresponding to each application.

Build and Deployment

1. **Incremental Builds**:
    - Leverage Nx's incremental build capabilities to optimize build times.
    - Configure appropriate cache and build settings in `nx.json`.

2. **Continuous Integration/Continuous Deployment (CI/CD)**:
    - Set up CI/CD pipelines using tools like GitHub Actions or Jenkins.
    - Ensure automated testing and linting are part of the CI process.

Version Control

1. **Git Workflow**:
    - Use Git flow or trunk-based development for version control.
    - Ensure all code changes go through code reviews via pull requests.
2. **Commit Messages**:
    - Follow a conventional commit message format for clarity and consistency.
    - Use tools like Commitizen to standardize commit messages.

Documentation

1. **Code Documentation**:
    - Use JSDoc or TypeDoc for documenting code.
    - Ensure all public APIs, functions, and classes are well-documented.
2. **Project Documentation**:
    - Maintain a `README.md` file at the root level with setup instructions and project overview.
    - Create additional documentation for specific features or libraries as needed.

Performance Optimization

1. **Lazy Loading**:
    - Implement lazy loading for feature modules to improve initial load time.

    - Use React's `Suspense` and dynamic imports for code splitting.
2. **Caching**:
    - Utilize browser caching strategies for static assets.
    - Configure service workers for caching using tools like Workbox.

Pros:

1. **Maintainability**: Clear organization and consistent coding standards improve code maintainability.
2. **Scalability**: Feature-based structure and reusable components support project scalability.
3. **Code Quality**: Enforcing testing, documentation, and type safety enhances code quality.
4. **Development Efficiency**: Best practices and standardized workflows streamline development processes.

Cons:

1. **Initial Setup**: Implementing and enforcing these best practices may require significant initial effort.
2. **Learning Curve**: Developers need to familiarize themselves with the established practices and tools.

Consequences:

- **Positive**:
    - Enhanced code quality, maintainability, and scalability.
    - Consistent development environment and processes.
    - Improved team collaboration and development efficiency.
- **Negative**:
    - Potentially increased complexity in the initial setup.
    - Steeper learning curve for developers new to these practices and tools.

Decision:

We will adopt the proposed best practices and patterns for our Nx monorepo project. These practices will help ensure the project remains maintainable, scalable, and efficient as it grows.

Alternatives Considered:

1. **Ad-Hoc Practices**: Allow each team or developer to follow their own practices.
    - **Pros**: More flexibility for individual developers.
    - **Cons**: Inconsistent codebase, reduced maintainability, and scalability challenges.
2. **Minimal Practices**: Implement only a minimal set of practices.
    - **Pros**: Simpler initial setup.
    - **Cons**: Potential issues with code quality, maintainability, and scalability over time.

Language: Javascript, Typescript, flow or combined

Title: Language Choice for Dashboard Development

Status: Proposed

Context:

We are developing a React-based project dashboard that will include multiple dashboards sharing configurations, components, logic, and features. To ensure maintainability, scalability, and developer productivity, we need to choose the most suitable language or combination of languages. The options considered are JavaScript, TypeScript, Flow, or a combination of these.

Decision:

We will compare JavaScript, TypeScript, Flow, and a combination of these languages based on various criteria such as type safety, developer productivity, community support, tooling, and scalability.

JavaScript

Pros:

1. **Familiarity**: Widely known and used, making it easier to onboard new developers.
2. **Flexibility**: Dynamic typing allows for rapid prototyping and flexible coding styles.
3. **Tooling and Ecosystem**: Rich ecosystem with extensive libraries and frameworks.

Cons:

1. **Lack of Type Safety**: No compile-time type checking, which can lead to runtime errors.
2. **Maintainability**: Harder to maintain large codebases due to the absence of static types.
3. **Error Detection**: Increased potential for bugs and errors that are hard to detect early.

TypeScript

Pros:

1. **Type Safety**: Compile-time type checking helps catch errors early and improves code quality.
2. **Developer Productivity**: Enhanced IDE support with autocompletion, refactoring, and navigation.
3. **Maintainability**: Easier to maintain and scale large codebases with static typing.
4. **Community Support**: Strong community and growing adoption in the industry.

Cons:

1. **Learning Curve**: Steeper learning curve for developers unfamiliar with TypeScript.
2. **Initial Setup**: Requires additional configuration and setup compared to plain JavaScript.
3. **Compilation Overhead**: Requires a build step to compile TypeScript to JavaScript.

Flow

Pros:

1. **Type Safety**: Provides static type checking similar to TypeScript.
2. **Flexibility**: Can be gradually introduced to an existing JavaScript codebase.
3. **Developer Productivity**: Enhanced IDE support with type inference and checking.

Cons:

1. **Community Support**: Smaller community and ecosystem compared to TypeScript.
2. **Tooling**: Limited tooling and library support relative to TypeScript.
3. **Learning Curve**: Steeper learning curve for developers unfamiliar with Flow.

Combined Approach

Pros:

1. **Gradual Adoption**: Allows for gradual adoption of type safety in an existing codebase.
2. **Flexibility**: Developers can choose the appropriate level of type safety for each part of the codebase.

Cons:

1. **Inconsistency**: Mixing languages can lead to inconsistencies and increased complexity.
2. **Tooling Conflicts**: Potential conflicts and issues with tooling and build processes.
3. **Maintenance Overhead**: Increased maintenance overhead due to managing multiple languages and configurations.

Consequences:

- **Positive**:
    - **TypeScript**: Improved code quality, maintainability, and developer productivity with strong type safety and community support.
    - **JavaScript**: Familiarity and flexibility for rapid prototyping and development.
    - **Flow**: Gradual adoption of type safety with minimal disruption.
    - **Combined Approach**: Flexibility to choose the appropriate level of type safety.
- **Negative**:
    - **TypeScript**: Steeper learning curve and initial setup effort.
    - **JavaScript**: Lack of type safety and potential maintainability issues.
    - **Flow**: Smaller community and limited tooling support.
    - **Combined Approach**: Inconsistencies and increased complexity in the codebase.

Decision:

After evaluating the pros and cons, we have decided to use **TypeScript** for our project dashboard development. TypeScript provides strong type safety, improved maintainability, and enhanced developer productivity. The benefits of using TypeScript outweigh the initial setup effort and learning curve, making it the best choice for our project's long-term success.

Alternatives Considered:

1. **JavaScript**: Provides familiarity and flexibility but lacks type safety and maintainability for large codebases.
2. **Flow**: Offers type safety but has limited community support and tooling compared to TypeScript.
3. **Combined Approach**: Provides flexibility but introduces inconsistencies and increased complexity.

Package manager

Title: Package Manager Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure efficient dependency management, consistent workflows, and optimal performance, we need to select the most suitable package manager. The options considered are Yarn, npm, pnpm, and others.

Decision:

We will compare Yarn, npm, and pnpm based on various criteria such as performance, workspace support, community support, and feature set.

npm

Pros:

1. **Widely Used**: npm is the default package manager for Node.js and has wide adoption.
2. **No Additional Setup**: Comes pre-installed with Node.js, reducing setup effort.
3. **Rich Ecosystem**: Extensive registry of packages and strong community support.

Cons:

1. **Performance**: Historically slower performance compared to Yarn and pnpm.
2. **Workspaces**: Basic support for workspaces, not as feature-rich as Yarn or pnpm.
3. **Lockfile Issues**: Lockfile format can be less efficient, leading to potential issues with deterministic builds.

Yarn

Pros:

1. **Performance**: Faster dependency installation and improved caching mechanisms.
2. **Workspaces**: Robust support for workspaces, making it easy to manage monorepos.
3. **Deterministic Installs**: Yarn’s lockfile ensures deterministic installs across environments.
4. **Plug'n'Play**: Advanced feature for improved module resolution and performance.

Cons:

1. **Additional Setup**: Requires additional installation step.
2. **Compatibility**: Some advanced features (like Plug'n'Play) may have compatibility issues with certain packages.

pnpm

Pros:

1. **Performance**: Extremely fast and efficient due to a unique approach to storing dependencies.
2. **Workspaces**: Excellent support for workspaces, ideal for managing monorepos.
3. **Disk Space**: Uses hard links to save disk space and avoid duplication of dependencies.
4. **Strictness**: Enforces strict dependency resolutions, reducing potential for conflicts.

Cons:

1. **Learning Curve**: May have a steeper learning curve for developers unfamiliar with its concepts.
2. **Community**: Smaller community compared to npm and Yarn, though growing rapidly.

Others (e.g., Lerna with npm/Yarn)

Pros:

1. **Enhanced Monorepo Management**: Tools like Lerna can enhance npm/Yarn workspaces.
2. **Flexibility**: Combines features of multiple tools for a tailored solution.

Cons:

1. **Complexity**: Increased complexity in setup and maintenance.
2. **Overhead**: Additional overhead in learning and managing multiple tools.

Consequences:

- **Positive**:
    - **npm**: Simplifies setup with no additional installation, widely adopted.
    - **Yarn**: Improved performance, robust workspace support, deterministic installs.

    - **pnpm**: Superior performance, efficient disk usage, strict dependency management.
    - **Others**: Tailored solutions combining strengths of multiple tools.
- **Negative**:
    - **npm**: Slower performance, basic workspace support, potential lockfile issues.
    - **Yarn**: Requires additional setup, potential compatibility issues.
    - **pnpm**: Steeper learning curve, smaller community.
    - **Others**: Increased complexity, overhead in learning and managing multiple tools.

Decision:

After evaluating the pros and cons, we have decided to use **pnpm** for our project. pnpm offers superior performance, efficient disk usage, and robust support for workspaces, making it well-suited for managing our monorepo. Despite the steeper learning curve, the benefits of using pnpm outweigh the potential drawbacks.

Alternatives Considered:

1. **npm**: Simplifies setup but lacks the performance and advanced workspace features needed for our monorepo.
2. **Yarn**: Offers robust workspace support and performance improvements but has potential compatibility issues and requires additional setup.
3. **Others**: Provide flexibility but introduce additional complexity and overhead.

State manager

Title: State Management Library Selection for React Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure efficient state management, scalability, and developer productivity, we need to select the most suitable state management library. The options considered are Redux, Zustand, and others (e.g., Context API, MobX).

Decision:

We will compare Redux, Zustand, and other potential state management solutions based on various criteria such as performance, ease of use, community support, and scalability.

Redux

Pros:

1. **Predictable State Management**: Centralized and predictable state management with strict unidirectional data flow.
2. **DevTools**: Powerful debugging tools like Redux DevTools for state inspection and time-travel debugging.
3. **Ecosystem**: Rich ecosystem with middleware (e.g., Redux Thunk, Redux Saga) and integrations.
4. **Community Support**: Large community, extensive documentation, and widespread adoption.

Cons:

1. **Boilerplate**: Can introduce significant boilerplate code for actions, reducers, and store configuration.
2. **Learning Curve**: Steeper learning curve, especially for beginners.
3. **Performance**: Potential performance overhead due to frequent re-renders if not optimized correctly.

Zustand

Pros:

1. **Simplicity**: Minimal boilerplate and straightforward API, making it easy to use and understand.
2. **Performance**: Highly performant with optimized re-renders and no need for context providers.
3. **Flexible**: Allows for both global and local state management.
4. **Small Bundle Size**: Lightweight library with a small footprint.

Cons:

1. **Smaller Ecosystem**: Smaller ecosystem and fewer middleware/integrations compared to Redux.
2. **Community Support**: Smaller community, though growing steadily.
3. **DevTools**: Limited DevTools support compared to Redux.

Context API

Pros:

1. **Built-in Solution**: Native to React, no need for external libraries.
2. **Simplicity**: Simple API for passing state through the component tree.
3. **Performance**: Optimized for small to medium-sized applications.

Cons:

1. **Scalability**: Can become cumbersome and lead to performance issues in larger applications due to frequent re-renders.
2. **Boilerplate**: Can introduce boilerplate code for managing complex state.

MobX

Pros:

1. **Reactive State Management**: Automatically tracks dependencies and optimizes re-renders.
2. **Ease of Use**: Simple and intuitive API for state management.
3. **Performance**: Efficient performance due to reactive state updates.

Cons:

1. **Community Support**: Smaller community and ecosystem compared to Redux.
2. **Learning Curve**: Requires learning new concepts (e.g., observables, actions) which can be a hurdle for new developers.

Others (e.g., Recoil, Apollo Client for GraphQL state management)

Pros:

1. **Tailored Solutions**: Specific libraries for particular use cases (e.g., Recoil for fine-grained state management, Apollo Client for GraphQL).
2. **Modern Features**: Leverages modern React features and best practices.

Cons:

1. **Niche Use Cases**: May be overkill or not suitable for general state management needs.
2. **Community Support**: Varies by library, with some having smaller communities and less mature ecosystems.

Consequences:

- **Positive**:
    - **Redux**: Provides a robust and predictable state management solution with powerful DevTools and strong community support.

    - **Zustand**: Offers a simple, performant, and flexible state management solution with minimal boilerplate.
    - **Context API**: Built-in and straightforward for small to medium-sized applications.
    - **MobX**: Provides reactive state management with efficient performance and a simple API.
- **Negative**:
    - **Redux**: Introduces significant boilerplate and has a steeper learning curve.
    - **Zustand**: Smaller ecosystem and fewer integrations compared to Redux.
    - **Context API**: Can lead to performance issues in larger applications.
    - **MobX**: Requires learning new concepts and has a smaller community.

Decision:

After evaluating the pros and cons, we have decided to use **Zustand** for our project. Zustand provides a balance between simplicity, performance, and flexibility, making it well-suited for our state management needs. Despite its smaller ecosystem, the benefits of minimal boilerplate, optimized re-renders, and straightforward API outweigh the potential drawbacks.

Alternatives Considered:

1. **Redux**: Provides a robust and predictable state management solution but introduces significant boilerplate and has a steeper learning curve.
2. **Context API**: Suitable for smaller applications but can lead to performance issues in larger applications.
3. **MobX**: Offers reactive state management but requires learning new concepts and has a smaller community.

Lint, code style

Title: Linting and Code Style Tools Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure code quality, consistency, and maintainability, we need to select suitable linting and code style tools. These tools will help enforce coding standards, catch potential errors early, and improve overall code readability.

Decision:

We will compare ESLint, Prettier, and other potential tools based on various criteria such as integration, community support, configurability, and ease of use.

ESLint

Pros:

1. **Customizability**: Highly configurable with support for custom rules and plugins.
2. **Integration**: Integrates seamlessly with popular code editors and CI/CD pipelines.
3. **Community Support**: Large community with extensive documentation and plugins.
4. **Error Detection**: Capable of catching a wide range of syntax and logical errors.

Cons:

1. **Complexity**: Can become complex to configure for larger projects with custom rules and plugins.
2. **Performance**: May introduce performance overhead for large codebases if not optimized.

Prettier

Pros:

1. **Code Formatting**: Enforces a consistent code style by formatting code automatically.
2. **Integration**: Integrates well with ESLint, code editors, and CI/CD pipelines.
3. **Ease of Use**: Simple to set up and use, reducing the need for manual code style enforcement.
4. **Community Support**: Large community and extensive plugin ecosystem.

Cons:

1. **Limited Configurability**: Offers limited configuration options to ensure consistency, which may not suit all projects.
2. **Overlap**: Some overlap with ESLint in terms of code style rules, requiring careful integration.

Stylelint

Pros:

1. **CSS Linting**: Specializes in linting CSS/SCSS/Less, ensuring consistent and error-free styles.
2. **Integration**: Works well with ESLint and Prettier for a complete linting setup.
3. **Customizability**: Highly configurable with support for custom rules and plugins.

Cons:

1. **Additional Setup**: Requires additional setup and configuration for full integration with other linting tools.
2. **Learning Curve**: May introduce a learning curve for developers unfamiliar with CSS linting.

Others (e.g., TSLint, StandardJS)

Pros:

1. **Specific Use Cases**: Tailored solutions for specific use cases (e.g., TSLint for TypeScript).
2. **Ease of Use**: Some tools offer simpler setups with predefined rulesets.

Cons:

1. **Deprecated/Reduced Support**: Some tools like TSLint are deprecated in favor of ESLint.
2. **Limited Flexibility**: May offer less flexibility compared to ESLint and Prettier.

Consequences:

- **Positive**:
    - **ESLint**: Ensures code quality and consistency through customizable linting rules and extensive plugin support.
    - **Prettier**: Enforces consistent code formatting, reducing the need for manual code style enforcement.
    - **Stylelint**: Provides specialized linting for CSS, ensuring consistent and error-free styles.
    - **Combined Approach**: Leveraging multiple tools ensures comprehensive linting and formatting coverage.
- **Negative**:
    - **ESLint**: Can become complex to configure and may introduce performance overhead.
    - **Prettier**: Limited configurability may not suit all projects.
    - **Stylelint**: Requires additional setup and introduces a learning curve.

- **Combined Approach**: Integrating multiple tools can increase complexity and maintenance overhead.

Decision:

After evaluating the pros and cons, we have decided to use a combination of **ESLint**, **Prettier**, and **Stylelint** for our project. This approach provides comprehensive linting and formatting coverage, ensuring code quality, consistency, and maintainability. Despite the initial setup complexity, the benefits of using these tools outweigh the potential drawbacks.

Configuration:

1. **ESLint**:
   - Use ESLint for JavaScript/TypeScript linting.
   - Integrate with Prettier to avoid conflicting rules.
   - Utilize popular ESLint plugins such as `eslint-plugin-react`, `eslint-plugin-import`, `eslint-plugin-jsx-a11y`, and `@typescript-eslint/eslint-plugin`.
2. **Prettier**:
   - Use Prettier for automatic code formatting.
   - Integrate with ESLint using `eslint-config-prettier` and `eslint-plugin-prettier` to ensure compatibility.
3. **Stylelint**:
   - Use Stylelint for linting CSS/SCSS/Less.
   - Integrate with Prettier using `stylelint-config-prettier` to avoid conflicting rules.

Alternatives Considered:

1. **ESLint Only**: Provides robust linting but lacks automatic code formatting.

2. **Prettier Only**: Ensures consistent formatting but lacks comprehensive linting rules.
3. **Other Tools**: Tailored solutions for specific use cases but may offer less flexibility and support.

Monitoring, log manager

Title: Log Management Solution for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure effective monitoring, debugging, and auditing, we need to select a suitable log management solution. The options considered are Log4js, Winston, and others (e.g., Pino, Bunyan).

Decision:

We will compare Log4js, Winston, Pino, and other potential log management solutions based on various criteria such as performance, ease of use, community support, and feature set.

Log4js

Pros:

1. **Feature-Rich**: Provides a wide range of logging features and configurations.
2. **Flexibility**: Highly configurable with various appenders, log levels, and formatting options.
3. **Community Support**: Well-established with extensive documentation and community support.

Cons:

1. **Complexity**: Can become complex to configure for larger projects with custom requirements.
2. **Performance**: May have performance overhead due to its feature-rich nature.

Winston

Pros:

1. **Modular**: Highly modular with support for multiple transports and formats.
2. **Flexibility**: Configurable log levels, formats, and transports (console, file, HTTP, etc.).
3. **Community Support**: Large community, extensive documentation, and widespread adoption.
4. **Async Logging**: Supports asynchronous logging to improve performance.

Cons:

1. **Complexity**: Initial setup can be complex, especially for advanced configurations.
2. **Bundle Size**: Larger bundle size compared to lightweight loggers like Pino.

Pino

Pros:

1. **Performance**: Extremely fast and efficient logging with minimal overhead.
2. **Simplicity**: Simple API with minimal configuration required.
3. **JSON Logging**: Outputs logs in JSON format by default, making it easy to integrate with log management systems.
4. **Ecosystem**: Growing ecosystem with plugins for various transports and integrations.

Cons:

1. **Less Feature-Rich**: Fewer features compared to Winston and Log4js, focusing more on performance.
2. **Community Size**: Smaller community compared to more established loggers like Winston.

Bunyan

Pros:

1. **JSON Logging**: Outputs logs in JSON format by default.
2. **Performance**: Focuses on performance with efficient logging.
3. **Built-in Support**: Provides built-in support for rotating files, streams, and more.

Cons:

1. **Complexity**: Configuration can be complex for advanced use cases.
2. **Community Support**: Smaller community compared to Winston and Log4js.

Consequences:

- **Positive**:
    - **Log4js**: Provides a feature-rich and highly configurable logging solution with strong community support.
    - **Winston**: Offers modularity, flexibility, and strong community support with a balanced feature set.
    - **Pino**: Focuses on performance and simplicity, making it ideal for high-performance applications.
    - **Bunyan**: Provides efficient JSON logging with built-in support for advanced use cases.
- **Negative**:
    - **Log4js**: Can become complex to configure and may introduce performance overhead.

- **Winston**: Initial setup can be complex, and it has a larger bundle size.
- **Pino**: Fewer features and smaller community compared to more established loggers.
- **Bunyan**: Smaller community and complex configuration for advanced use cases.

Decision:

After evaluating the pros and cons, we have decided to use **Winston** as the log management solution for our project. Winston offers a balance between flexibility, modularity, and performance, making it well-suited for our logging needs. Despite the initial setup complexity, the benefits of using Winston, such as its extensive feature set and strong community support, outweigh the potential drawbacks.

Configuration:

1. **Basic Setup**:
   - Configure Winston with default transports (console, file) for basic logging needs.
2. **Advanced Configuration**:
   - Utilize custom transports for specific requirements (e.g., HTTP, database).
   - Set up different log levels for development and production environments.
   - Integrate with monitoring and alerting systems for proactive issue resolution.
3. **Performance Optimization**:
   - Use asynchronous logging to minimize performance impact.
   - Leverage log rotation to manage log file sizes effectively.

Alternatives Considered:

1. **Log4js**: Provides a feature-rich solution but can become complex and may have performance overhead.
2. **Pino**: Offers high performance and simplicity but lacks some advanced features and has a smaller community.
3. **Bunyan**: Efficient JSON logging with built-in support for advanced use cases but has a smaller community and complex configuration.

Error Monitoring

Title: Error Monitoring Tool Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure effective error monitoring, debugging, and proactive issue resolution, we need to select a suitable error monitoring tool. The primary options considered are Sentry and other alternatives such as LogRocket, Bugsnag, and Rollbar.

Decision:

We will compare Sentry with LogRocket, Bugsnag, and Rollbar based on various criteria such as features, ease of use, community support, integration capabilities, and cost.

Sentry

Pros:

1. **Comprehensive Error Tracking**: Provides detailed error tracking and performance monitoring.
2. **Ease of Use**: Simple setup with extensive documentation and integration guides.
3. **Alerts and Notifications**: Supports configurable alerts and notifications to quickly identify and address issues.
4. **Rich Context**: Captures additional context such as user information, breadcrumbs, and environment details.
5. **Community Support**: Large community and widespread adoption, ensuring continuous improvements and support.
6. **Performance Monitoring**: Built-in support for performance monitoring.

Cons:

1. **Cost**: Can become expensive for large projects with high error volumes.
2. **Performance Impact**: Potential performance impact due to additional error tracking overhead.

LogRocket

Pros:

1. **Session Replay**: Provides session replay functionality to see exactly what users experienced.
2. **Error Tracking**: Integrates error tracking with session replays for better context.
3. **User Insights**: Offers insights into user interactions and behaviors leading up to errors.
4. **Performance Monitoring**: Includes performance monitoring features to track application performance metrics.

Cons:

1. **Cost**: Can become costly, especially for projects with high user traffic.

2. **Setup Complexity**: Initial setup can be more complex compared to other tools.
3. **Limited Standalone Error Tracking**: Primarily focused on session replay, with error tracking as a secondary feature.

Bugsnag

Pros:

1. **Automated Error Detection**: Automatically detects and captures errors with minimal configuration.
2. **Rich Context**: Provides detailed error reports with additional context like user information and environment details.
3. **Alerts and Integrations**: Supports alerts and integrates with various third-party tools for incident management.
4. **Ease of Use**: Simple setup with extensive documentation.

Cons:

1. **Cost**: Can become expensive for large projects with high error volumes.
2. **Customization**: Limited customization options compared to Sentry.

Rollbar

Pros:

1. **Real-Time Error Tracking**: Provides real-time error tracking with detailed error reports.
2. **Ease of Use**: Simple setup and easy integration with various frameworks and languages.
3. **Automated Grouping**: Automatically groups similar errors to reduce noise.
4. **Rich Context**: Captures additional context such as user information and environment details.

Cons:

1. **Cost**: Pricing can become a concern for large projects with high error volumes.
2. **Performance Impact**: Potential performance impact due to additional error tracking overhead.

Consequences:

- **Positive**:
    - **Sentry**: Provides comprehensive error tracking, performance monitoring, and rich context for effective debugging.
    - **LogRocket**: Offers session replay with integrated error tracking for better user insights and debugging.
    - **Bugsnag**: Automated error detection with rich context and ease of use.
    - **Rollbar**: Real-time error tracking with detailed reports and automated grouping.
- **Negative**:
    - **Sentry**: Can become expensive and may introduce performance overhead.
    - **LogRocket**: Primarily focused on session replay with potentially complex setup and higher costs.
    - **Bugsnag**: Limited customization options and potential cost concerns.
    - **Rollbar**: Potential performance impact and cost considerations.

Decision:

After evaluating the pros and cons, we have decided to use **Sentry** as the error monitoring tool for our project. Sentry provides comprehensive error tracking, performance monitoring, and rich context for effective debugging. Despite potential cost and performance considerations, the benefits of using Sentry, such as its ease of use, extensive features, and strong community support, outweigh the potential drawbacks.

Configuration:

1. **Basic Setup**:
    - Integrate Sentry with the React application using the Sentry SDK.
    - Configure Sentry to capture errors, exceptions, and performance metrics.
2. **Advanced Configuration**:
    - Set up custom tags and contexts to capture additional information relevant to the project.
    - Configure alerts and notifications for proactive issue resolution.
    - Integrate with third-party tools like Slack, Jira, and GitHub for streamlined incident management.
3. **Performance Optimization**:
    - Optimize Sentry configuration to minimize performance impact.
    - Use sampling and rate limiting to control error volume and costs.

Alternatives Considered:

1. **LogRocket**: Offers session replay with integrated error tracking but is primarily focused on user insights and can become costly.
2. **Bugsnag**: Automated error detection with rich context but limited customization and potential cost concerns.
3. **Rollbar**: Real-time error tracking with detailed reports but potential performance impact and cost considerations.

Apis, communicate with a server

Title: Client-Server Communication Tool Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. Effective client-server communication is crucial for data fetching, caching, and state synchronization. We need to select a suitable tool to manage this communication efficiently. The primary options considered are React Query and other alternatives such as Redux Toolkit Query, Apollo Client, and Axios.

Decision:

We will compare React Query with Redux Toolkit Query, Apollo Client, and Axios based on various criteria such as features, ease of use, community support, and integration capabilities.

React Query

Pros:

1. **Simplified Data Fetching**: Abstracts away complex data fetching logic with a simple API.
2. **Caching**: Provides built-in caching, refetching, and synchronization capabilities.
3. **DevTools**: Includes powerful DevTools for inspecting query states.
4. **Automatic Refetching**: Supports automatic refetching and background updates.
5. **Community Support**: Strong community support and extensive documentation.

Cons:

1. **Learning Curve**: Requires understanding of React Query concepts and patterns.
2. **Bundle Size**: Adds to the bundle size of the application.

Redux Toolkit Query

Pros:

1. **Integration with Redux**: Seamless integration with Redux Toolkit for managing global state and data fetching.
2. **Simplified API**: Provides a simple and consistent API for data fetching.
3. **Caching**: Built-in caching and automatic refetching capabilities.
4. **DevTools**: Leverages Redux DevTools for state inspection and debugging.

Cons:

1. **Redux Dependency**: Requires Redux, which may introduce additional complexity and boilerplate.
2. **Learning Curve**: Requires knowledge of both Redux and Redux Toolkit Query.

Apollo Client

Pros:

1. **GraphQL Support**: Provides a powerful and flexible solution for GraphQL-based data fetching.
2. **Caching**: Advanced caching mechanisms with fine-grained control.
3. **DevTools**: Includes Apollo DevTools for inspecting and debugging GraphQL queries.
4. **Community Support**: Large community and extensive documentation.

Cons:

1. **Complexity**: More complex setup and learning curve, especially for REST APIs.
2. **Bundle Size**: Can significantly increase the bundle size.

Axios

Pros:

1. **Simplicity**: Simple and straightforward API for making HTTP requests.
2. **Flexibility**: Highly flexible and customizable for various use cases.
3. **Community Support**: Large community and well-documented.

Cons:

1. **No Built-in Caching**: Lacks built-in caching and automatic refetching capabilities.
2. **Manual State Management**: Requires manual handling of state management and side effects.

Consequences:

- **Positive**:
    - **React Query**: Simplifies data fetching with built-in caching, refetching, and synchronization.
    - **Redux Toolkit Query**: Seamless integration with Redux and consistent API for data fetching.
    - **Apollo Client**: Powerful GraphQL support with advanced caching and DevTools.
    - **Axios**: Simple and flexible HTTP requests with a straightforward API.
- **Negative**:
    - **React Query**: Learning curve and additional bundle size.
    - **Redux Toolkit Query**: Requires Redux dependency and associated complexity.

- **Apollo Client**: Complexity and increased bundle size, especially for REST APIs.
- **Axios**: No built-in caching or automatic refetching, requiring manual state management.

Decision:

After evaluating the pros and cons, we have decided to use **React Query** for client-server communication in our project. React Query provides a balance between simplicity, powerful features, and strong community support. Its built-in caching, refetching, and synchronization capabilities make it well-suited for managing data fetching and state synchronization efficiently.

Configuration:

1. **Basic Setup**:
   - Install React Query and set up the QueryClientProvider in the React application.
   - Define and use queries and mutations to manage data fetching and state updates.
2. **Advanced Configuration**:
   - Customize caching, refetching, and background synchronization strategies as needed.
   - Integrate React Query DevTools for enhanced debugging and inspection.
3. **Performance Optimization**:
   - Optimize query configurations to minimize unnecessary refetching and data fetching overhead.
   - Use pagination and infinite scrolling features to handle large data sets efficiently.

Alternatives Considered:

1. **Redux Toolkit Query**: Provides seamless integration with Redux but introduces additional complexity and boilerplate.

2. **Apollo Client**: Offers powerful GraphQL support but is more complex and increases bundle size.
3. **Axios**: Simple and flexible but lacks built-in caching and requires manual state management.

Routing

Title: Routing Library Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. Efficient and flexible routing is crucial for handling navigation and managing different views in our single-page application (SPA). We need to select a suitable routing library that meets our needs. The primary options considered are React Router, Next.js, and other alternatives such as Reach Router and Wouter.

Decision:

We will compare React Router, Next.js, Reach Router, and Wouter based on various criteria such as features, ease of use, community support, integration capabilities, and performance.

React Router

Pros:

1. **Feature-Rich**: Provides a comprehensive set of features for routing, including nested routes, dynamic routing, and route guards.

2. **Declarative Routing**: Uses a declarative approach to define routes, making it easy to understand and manage.
3. **Community Support**: Large community, extensive documentation, and widespread adoption.
4. **Integration**: Seamless integration with React and other libraries.

Cons:

1. **Complexity**: Can become complex for large applications with deeply nested routes.
2. **Learning Curve**: Requires understanding of routing concepts and patterns specific to React Router.

Next.js

Pros:

1. **Built-In Routing**: Provides built-in file-based routing, simplifying the routing setup.
2. **Server-Side Rendering (SSR)**: Supports SSR, static site generation (SSG), and client-side rendering (CSR) out of the box.
3. **Performance**: Optimized for performance with features like automatic code splitting and prefetching.
4. **Community Support**: Large community, extensive documentation, and growing adoption.

Cons:

1. **Opinionated**: Next.js is an opinionated framework, which may limit flexibility for some use cases.
2. **Learning Curve**: Requires learning Next.js-specific concepts and conventions.

Reach Router

Pros:

1. **Simplicity**: Designed to be simple and easy to use with a minimal API.
2. **Accessibility**: Focuses on accessibility by default, ensuring that routes are accessible to all users.
3. **Declarative Routing**: Uses a declarative approach similar to React Router.

Cons:

1. **Maintenance**: Reach Router is now merged with React Router, so it may not receive individual updates.
2. **Features**: Lacks some advanced features compared to React Router and Next.js.

Wouter

Pros:

1. **Lightweight**: Extremely lightweight with a small footprint, making it ideal for performance-sensitive applications.
2. **Simplicity**: Simple API with minimal configuration required.
3. **Flexibility**: Provides flexibility in defining routes and managing navigation.

Cons:

1. **Community Support**: Smaller community and less documentation compared to more established libraries.
2. **Features**: Lacks some advanced features provided by React Router and Next.js.

Consequences:

- **Positive**:
    - **React Router**: Provides a comprehensive and flexible routing solution with strong community support.

    - **Next.js**: Offers built-in routing with SSR and performance optimizations, suitable for server-rendered applications.
    - **Reach Router**: Simple and accessible routing with a declarative approach.
    - **Wouter**: Lightweight and flexible, ideal for performance-sensitive applications.
- **Negative**:
    - **React Router**: Can become complex for large applications with a steeper learning curve.
    - **Next.js**: Opinionated framework with a learning curve specific to Next.js concepts.
    - **Reach Router**: Lacks some advanced features and is now merged with React Router.
    - **Wouter**: Smaller community and fewer features compared to more established libraries.

Decision:

After evaluating the pros and cons, we have decided to use **React Router** as the routing library for our project. React Router provides a balance between comprehensive features, flexibility, and strong community support. Its declarative approach and seamless integration with React make it well-suited for managing navigation in our single-page application.

Configuration:

1. **Basic Setup**:
    - Install React Router and set up the Router, Routes, and Route components.
    - Define routes for different views and components in the application.
2. **Advanced Configuration**:
    - Implement nested routes, dynamic routing, and route guards as needed.

    - Use React Router hooks (e.g., `useHistory`, `useLocation`) for advanced navigation control.
3. **Performance Optimization**:
    - Optimize route-based code splitting to improve load times.
    - Use lazy loading for components to reduce initial bundle size.

Alternatives Considered:

1. **Next.js**: Provides built-in routing and SSR but is an opinionated framework with a specific learning curve.
2. **Reach Router**: Simple and accessible but lacks some advanced features and is now merged with React Router.
3. **Wouter**: Lightweight and flexible but has a smaller community and fewer features.

Ui kit

Title: UI Kit Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure a consistent and maintainable user interface, we need to select a suitable UI kit that allows for customization and flexibility while promoting reusability across multiple dashboards. The primary options considered are Headless UI, Material-UI (MUI), Ant Design, and Tailwind CSS.

Decision:

We will compare Headless UI with Material-UI (MUI), Ant Design, and Tailwind CSS based on various criteria such as customizability, ease of use, community support, and integration capabilities. The decision is to use Headless UI due to its focus on providing unstyled, accessible components that can be easily customized and reused.

Headless UI

Pros:

1. **Customizability**: Provides unstyled components that allow for full control over the design and styling.
2. **Accessibility**: Focuses on accessibility out of the box, ensuring components are accessible to all users.
3. **Flexibility**: Easily integrates with any styling solution, such as Tailwind CSS, CSS-in-JS, or traditional CSS.
4. **Reusability**: Promotes reusability across different parts of the application, enabling consistent UI patterns.

Cons:

1. **Learning Curve**: Requires additional effort to style components as needed.
2. **Limited Components**: Smaller set of pre-built components compared to more comprehensive UI kits.

Material-UI (MUI)

Pros:

1. **Comprehensive**: Offers a wide range of pre-built, styled components that follow Material Design guidelines.
2. **Customization**: Provides theming and styling options to customize components.
3. **Community Support**: Large community, extensive documentation, and widespread adoption.

Cons:

1. **Styling Overhead**: Customizing components to deviate from Material Design can be challenging.
2. **Bundle Size**: Larger bundle size due to comprehensive feature set and styles.

Ant Design

Pros:

1. **Comprehensive**: Offers a rich set of pre-built components with a clean and modern design.
2. **Internationalization**: Built-in support for internationalization and localization.
3. **Community Support**: Large community and extensive documentation.

Cons:

1. **Styling Overhead**: Customizing the default styles can be complex and time-consuming.
2. **Bundle Size**: Larger bundle size due to the extensive component library.

Tailwind CSS

Pros:

1. **Utility-First**: Provides a utility-first approach to styling, allowing for rapid and flexible UI development.
2. **Customizability**: High degree of customization with a small bundle size.
3. **Community Support**: Growing community and extensive documentation.

Cons:

1. **No Pre-Built Components**: Requires building and styling components from scratch.
2. **Learning Curve**: Requires understanding of utility classes and how to apply them effectively.

Consequences:

- **Positive**:
    - **Headless UI**: Provides unstyled, accessible components that are highly customizable and reusable across the application.
    - **Material-UI (MUI)**: Comprehensive set of pre-built components with strong community support.
    - **Ant Design**: Rich component library with built-in internationalization support.
    - **Tailwind CSS**: Utility-first approach that offers high flexibility and customizability.
- **Negative**:
    - **Headless UI**: Requires additional effort to style components, limited pre-built components.
    - **Material-UI (MUI)**: Customizing components can be challenging, larger bundle size.
    - **Ant Design**: Complex customization, larger bundle size.
    - **Tailwind CSS**: No pre-built components, requires understanding of utility classes.

Decision:

After evaluating the pros and cons, we have decided to use **Headless UI** as the UI kit for our project. Headless UI provides the flexibility and customizability needed to create a consistent and maintainable user interface. Its focus on accessibility and unstyled components allows us to define our own design system and reuse components across different parts of the application.

Configuration:

1. **Basic Setup**:
    - Install Headless UI and integrate it with the React application.
    - Set up basic components using Headless UI and style them according to the design requirements.
2. **Advanced Configuration**:
    - Create a custom design system using Tailwind CSS or another styling solution to ensure consistency across components.
    - Implement additional accessibility features as needed.
3. **Component Library**:
    - Develop a shared component library using Headless UI components to promote reusability and consistency.
    - Document the custom components and their usage for the development team.

Alternatives Considered:

1. **Material-UI (MUI)**: Comprehensive and well-supported but can be challenging to customize and has a larger bundle size.
2. **Ant Design**: Rich component library with internationalization support but complex customization and larger bundle size.
3. **Tailwind CSS**: Offers high flexibility and customizability but requires building components from scratch.

Charts

Title: Chart Library Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To visualize data effectively, we need to select a suitable chart library that provides a balance between features, performance, and ease of use. The primary options considered are Chart.js, Recharts, D3.js, and other alternatives such as Victory and Highcharts.

Decision:

We will compare Chart.js, Recharts, D3.js, Victory, and Highcharts based on various criteria such as features, ease of use, community support, integration capabilities, and performance. The decision is to use Recharts due to its React integration, ease of use, and sufficient feature set.

Chart.js

Pros:

1. **Simplicity**: Easy to set up and use, with a straightforward API.
2. **Performance**: Highly performant for basic charting needs.
3. **Community Support**: Large community, extensive documentation, and widespread adoption.

Cons:

1. **Integration**: Not specifically designed for React, requiring wrappers for seamless integration.
2. **Customization**: Limited customization options compared to more advanced libraries like D3.js.

Recharts

Pros:

1. **React Integration**: Built specifically for React, providing seamless integration with React components.
2. **Ease of Use**: Simple and intuitive API, making it easy to create and customize charts.

3. **Community Support**: Strong community and good documentation.
4. **Customizability**: Offers a good balance between ease of use and customizability.

Cons:

1. **Feature Set**: Less feature-rich compared to D3.js, but sufficient for most common use cases.
2. **Performance**: May not be as performant as more lightweight libraries for very large datasets.

D3.js

Pros:

1. **Flexibility**: Extremely flexible and powerful, allowing for highly customized and complex visualizations.
2. **Feature-Rich**: Comprehensive feature set for creating all types of visualizations.
3. **Performance**: High performance with the ability to handle large datasets efficiently.

Cons:

1. **Complexity**: Steeper learning curve and more complex API compared to other libraries.
2. **Integration**: Requires more effort to integrate with React, often needing wrappers or additional libraries.

Victory

Pros:

1. **React Integration**: Built specifically for React, providing a seamless integration.
2. **Ease of Use**: Intuitive and easy-to-use API.
3. **Customizability**: Highly customizable with a good balance of features.

Cons:

1. **Performance**: May not be as performant as more lightweight libraries for very large datasets.
2. **Community Support**: Smaller community and fewer resources compared to Chart.js and D3.js.

Highcharts

Pros:

1. **Feature-Rich**: Extensive feature set with support for a wide variety of chart types.
2. **Customization**: Highly customizable and flexible.
3. **Community Support**: Large community and extensive documentation.

Cons:

1. **Cost**: Requires a commercial license for most use cases.
2. **Integration**: Not specifically designed for React, requiring wrappers for seamless integration.

Consequences:

- **Positive**:
    - **Chart.js**: Easy to use and highly performant for basic charts with strong community support.
    - **Recharts**: Seamless React integration, easy to use, and sufficiently customizable for most use cases.
    - **D3.js**: Extremely flexible and powerful with high performance for complex visualizations.
    - **Victory**: Intuitive and customizable with strong React integration.
    - **Highcharts**: Feature-rich and highly customizable with extensive community support.
- **Negative**:

- **Chart.js**: Limited customization and requires wrappers for React integration.
- **Recharts**: Less feature-rich and potentially less performant for very large datasets.
- **D3.js**: Steeper learning curve and more complex integration with React.
- **Victory**: Smaller community and fewer resources.
- **Highcharts**: Requires a commercial license and wrappers for React integration.

Decision:

After evaluating the pros and cons, we have decided to use **Recharts** as the chart library for our project. Recharts provides a balance between ease of use, React integration, and sufficient customization options, making it well-suited for our data visualization needs. Despite having a less extensive feature set compared to D3.js, its simplicity and seamless integration with React outweigh the potential drawbacks.

Configuration:

1. **Basic Setup**:
   - Install Recharts and integrate it with the React application.
   - Create basic charts using Recharts components such as LineChart, BarChart, and PieChart.
2. **Advanced Configuration**:
   - Customize chart styles and behaviors using Recharts' props and customization options.
   - Implement interactive features such as tooltips, legends, and animations.
3. **Performance Optimization**:
   - Optimize chart rendering for large datasets by leveraging Recharts' performance features.
   - Use lazy loading for charts to reduce initial load time.

Alternatives Considered:

1. **Chart.js**: Easy to use and performant but requires wrappers for React integration and has limited customization.
2. **D3.js**: Extremely flexible and powerful but has a steeper learning curve and more complex integration.
3. **Victory**: Intuitive and customizable with strong React integration but smaller community support.
4. **Highcharts**: Feature-rich and highly customizable but requires a commercial license and wrappers for React integration.

Css architecture

Title: CSS Architecture Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure a scalable, maintainable, and consistent styling approach, we need to select a suitable CSS architecture. The primary options considered are BEM (Block Element Modifier), CSS Modules, Styled Components, and Tailwind CSS.

Decision:

We will compare BEM, CSS Modules, Styled Components, and Tailwind CSS based on various criteria such as modularity, ease of use, community support, integration capabilities, and performance. The decision is to use Tailwind CSS due to its utility-first approach and strong support for rapid UI development.

BEM (Block Element Modifier)

Pros:

1. **Modularity**: Provides a structured and modular approach to CSS, promoting reusable and maintainable code.
2. **Consistency**: Ensures a consistent naming convention, making the codebase easier to understand.
3. **Community Support**: Well-established methodology with extensive documentation and community support.

Cons:

1. **Verbosity**: Can result in verbose class names, making the HTML harder to read.
2. **Manual Management**: Requires manual management of styles and naming conventions, which can be error-prone.

CSS Modules

Pros:

1. **Encapsulation**: Encapsulates CSS locally, avoiding global scope conflicts and promoting modularity.
2. **Integration**: Seamlessly integrates with React and other modern frameworks.
3. **Maintainability**: Promotes maintainable and reusable styles by scoping CSS to components.

Cons:

1. **Setup Complexity**: Requires additional build configuration, which can add complexity to the project setup.
2. **Learning Curve**: Developers need to understand the concept of scoped styles and module-specific syntax.

Styled Components

Pros:

1. **Component-Based**: Allows styling directly within React components using JavaScript, promoting a cohesive development approach.
2. **Dynamic Styling**: Supports dynamic and conditional styling, making it easy to adapt styles based on props and state.
3. **Community Support**: Strong community support and extensive documentation.

Cons:

1. **Performance**: Can introduce runtime overhead due to dynamic style generation.
2. **Learning Curve**: Requires understanding of CSS-in-JS concepts and syntax.

Tailwind CSS

Pros:

1. **Utility-First**: Provides a utility-first approach to styling, enabling rapid UI development with pre-defined utility classes.
2. **Consistency**: Ensures consistent styling across the application with a small set of reusable classes.
3. **Customization**: Highly customizable through configuration files, allowing for a tailored design system.
4. **Performance**: Generates minimal CSS, reducing bundle size and improving performance.

Cons:

1. **Learning Curve**: Requires understanding of utility classes and how to apply them effectively.
2. **Class Name Proliferation**: Can result in a proliferation of class names in the HTML, making it harder to read.

Consequences:

- **Positive**:

    - **BEM**: Provides a modular and consistent approach to CSS with strong community support.
    - **CSS Modules**: Promotes encapsulated and maintainable styles with seamless React integration.
    - **Styled Components**: Allows dynamic and component-based styling with strong community support.
    - **Tailwind CSS**: Enables rapid UI development with consistent and customizable utility classes.
- **Negative**:
    - **BEM**: Can result in verbose class names and requires manual management.
    - **CSS Modules**: Adds setup complexity and requires understanding of scoped styles.
    - **Styled Components**: Introduces runtime overhead and requires learning CSS-in-JS concepts.
    - **Tailwind CSS**: Requires understanding of utility classes and can result in class name proliferation.

Decision:

After evaluating the pros and cons, we have decided to use **Tailwind CSS** as the CSS architecture for our project. Tailwind CSS provides a utility-first approach that enables rapid UI development and ensures consistent styling across the application. Its high customizability and performance benefits make it well-suited for our needs.

Configuration:

1. **Basic Setup**:
    - Install Tailwind CSS and integrate it with the React application.
    - Configure the `tailwind.config.js` file to define the design system and custom utility classes.
2. **Advanced Configuration**:

    - Extend Tailwind CSS with custom plugins and components as needed.
    - Utilize Tailwind's purge feature to remove unused styles and optimize performance.
3. **Development Workflow**:
    - Encourage the use of Tailwind utility classes for rapid UI development.
    - Maintain consistency by adhering to the defined design system and utility classes.

Alternatives Considered:

1. **BEM**: Provides a modular approach but can result in verbose class names and requires manual management.
2. **CSS Modules**: Promotes encapsulated styles with React integration but adds setup complexity.
3. **Styled Components**: Allows dynamic styling within components but introduces runtime overhead.
4. **Tailwind CSS**: Enables rapid UI development with utility-first styling but requires understanding of utility classes.

Authentications, authorization

Title: Authentication Library Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure secure and efficient user authentication, we need to select a suitable authentication library. The primary options considered are Auth0, Firebase Authentication, NextAuth.js, and Passport.js.

Decision:

We will compare Auth0, Firebase Authentication, NextAuth.js, and Passport.js based on various criteria such as security features, ease of use, community support, integration capabilities, and scalability. The decision is to use Auth0 due to its comprehensive features, ease of integration, and strong security standards.

Auth0

Pros:

1. **Comprehensive Features**: Provides a wide range of authentication and authorization features, including social login, multifactor authentication (MFA), and single sign-on (SSO).
2. **Ease of Use**: Simple setup with extensive documentation and integration guides.
3. **Security**: Strong security standards and compliance with industry regulations.
4. **Scalability**: Scales easily with the growth of the application.
5. **Community Support**: Large community and enterprise-level support options.

Cons:

1. **Cost**: Can become expensive for larger applications with many users.
2. **Vendor Lock-In**: Dependence on a third-party service.

Firebase Authentication

Pros:

1. **Ease of Use**: Simple setup and integration with Firebase services.
2. **Comprehensive**: Provides various authentication methods, including email/password, phone, and social login.

3. **Real-Time**: Integrates well with other Firebase services, such as Firestore and Realtime Database.
4. **Community Support**: Large community and extensive documentation.

Cons:

1. **Vendor Lock-In**: Dependence on Firebase services.
2. **Scalability Costs**: Costs can increase with application growth and user base expansion.
3. **Customization**: Limited customization options compared to more specialized solutions like Auth0.

NextAuth.js

Pros:

1. **Built for Next.js**: Specifically designed for Next.js applications, providing seamless integration.
2. **Flexibility**: Supports various authentication providers, including email/password and OAuth.
3. **Open Source**: Free and open-source with a growing community.
4. **Customization**: Highly customizable to fit specific application needs.

Cons:

1. **Learning Curve**: Requires understanding of Next.js-specific concepts and configurations.
2. **Community Size**: Smaller community compared to Auth0 and Firebase.

Passport.js

Pros:

1. **Flexibility**: Provides a wide range of authentication strategies and supports various authentication methods.

2. **Framework Agnostic**: Can be used with any Node.js framework.
3. **Customizable**: Highly customizable to fit specific application requirements.

Cons:

1. **Complexity**: Requires more configuration and setup compared to other solutions.
2. **Maintenance**: Requires ongoing maintenance and updates to ensure security.
3. **Community Size**: Smaller community and fewer resources compared to Auth0 and Firebase.

Consequences:

- **Positive**:
    - **Auth0**: Provides comprehensive features, strong security, and scalability with enterprise-level support.
    - **Firebase Authentication**: Simple setup with integration into Firebase services and strong community support.
    - **NextAuth.js**: Seamless integration with Next.js, flexibility, and open-source.
    - **Passport.js**: Flexible, framework-agnostic, and highly customizable.
- **Negative**:
    - **Auth0**: Can become expensive and introduces vendor lock-in.
    - **Firebase Authentication**: Vendor lock-in and potential scalability costs with limited customization.
    - **NextAuth.js**: Requires understanding Next.js-specific concepts and has a smaller community.
    - **Passport.js**: Requires more configuration, maintenance, and has a smaller community.

Decision:

After evaluating the pros and cons, we have decided to use **Auth0** as the authentication library for our project. Auth0 provides a comprehensive set of features, strong security standards, and ease of integration, making it well-suited for our authentication needs. Despite potential costs and vendor lock-in, the benefits of using Auth0, such as its scalability, security, and enterprise-level support, outweigh the potential drawbacks.

Configuration:

1. **Basic Setup**:
    - Sign up for Auth0 and create an Auth0 application.
    - Integrate Auth0 with the React application using the Auth0 React SDK.
    - Configure authentication methods and user roles as needed.
2. **Advanced Configuration**:
    - Implement multifactor authentication (MFA) and single sign-on (SSO) for enhanced security.
    - Customize login and registration pages to match the application's branding.
    - Set up roles and permissions for fine-grained access control.
3. **Monitoring and Maintenance**:
    - Regularly monitor authentication logs and alerts for potential security issues.
    - Keep the Auth0 SDK and configuration up to date with the latest security patches and features.

Alternatives Considered:

1. **Firebase Authentication**: Simple setup and integration with Firebase services but introduces vendor lock-in and scalability costs.

2. **NextAuth.js**: Seamless integration with Next.js and flexibility but requires understanding Next.js-specific concepts and has a smaller community.
3. **Passport.js**: Flexible and highly customizable but requires more configuration and ongoing maintenance.

Documentation (storybook, kitchen sink, …)

Title: Documentation Strategy Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure clear, consistent, and comprehensive documentation, we need to select a suitable documentation strategy. The primary options considered are Storybook, Kitchen Sink, and traditional documentation tools like Markdown and Docsify.

Decision:

We will compare Storybook, Kitchen Sink, and traditional documentation tools based on various criteria such as ease of use, community support, integration capabilities, and comprehensiveness. The decision is to use **Storybook** due to its interactive and component-focused approach, which aligns well with our React development workflow.

Storybook

Pros:

1. **Component-Centric**: Focuses on documenting individual components, making it easier to develop and test components in isolation.
2. **Interactive**: Provides an interactive interface for viewing and testing components, which improves developer experience.
3. **Ease of Use**: Simple setup with extensive documentation and integration guides.
4. **Add-ons and Plugins**: Extensive ecosystem of add-ons and plugins to extend functionality (e.g., accessibility checks, design tokens).
5. **Community Support**: Large and active community, ensuring continuous improvements and support.

Cons:

1. **Learning Curve**: Requires understanding Storybook-specific configurations and workflows.
2. **Performance**: Can introduce performance overhead if not configured optimally.

Kitchen Sink

Pros:

1. **Comprehensive**: Provides a single place to view all components, utilities, and documentation.
2. **Customizable**: Highly customizable to fit specific project needs and workflows.
3. **Integration**: Can integrate various tools and technologies to create a unified documentation site.

Cons:

1. **Complex Setup**: Requires more effort to set up and maintain compared to dedicated tools like Storybook.
2. **Maintenance Overhead**: Keeping the documentation site updated and consistent with the codebase can be challenging.

Traditional Documentation Tools (Markdown, Docsify)

Pros:

1. **Simplicity**: Easy to write and maintain using simple Markdown files.
2. **Version Control**: Can be easily integrated with version control systems like Git.
3. **Flexibility**: Can be customized and extended using various static site generators and tools (e.g., Docsify, Docusaurus).

Cons:

1. **Lack of Interactivity**: Does not provide an interactive interface for viewing and testing components.
2. **Separation from Code**: Documentation can become outdated or inconsistent with the actual codebase.

Consequences:

- **Positive**:
    - **Storybook**: Provides an interactive and component-focused documentation approach, improving developer experience and ensuring consistency.
    - **Kitchen Sink**: Offers a comprehensive and customizable documentation site, combining various tools and technologies.
    - **Traditional Documentation Tools**: Simple and easy to maintain, with strong integration with version control systems.
- **Negative**:
    - **Storybook**: Requires learning specific configurations and can introduce performance overhead.
    - **Kitchen Sink**: Complex setup and higher maintenance overhead.
    - **Traditional Documentation Tools**: Lack of interactivity and potential for outdated documentation.

Decision:

After evaluating the pros and cons, we have decided to use **Storybook** as the documentation strategy for our project. Storybook provides an interactive, component-focused approach that aligns well with our React development workflow. Its extensive ecosystem, ease of use, and strong community support make it the best choice for maintaining clear, consistent, and comprehensive documentation.

Configuration:

1. **Basic Setup**:
    - Install Storybook and configure it to work with the Nx monorepo and React components.
    - Create stories for existing components to document their usage and variations.
2. **Advanced Configuration**:
    - Integrate add-ons for enhanced functionality, such as accessibility checks and design token support.
    - Customize Storybook's theme and layout to match the project's branding and design guidelines.
3. **Documentation Workflow**:
    - Encourage developers to create and update stories for new and modified components.
    - Use Storybook as part of the CI/CD pipeline to ensure documentation is always up to date with the latest code changes.

Alternatives Considered:

1. **Kitchen Sink**: Provides a comprehensive and customizable documentation site but requires more effort to set up and maintain.
2. **Traditional Documentation Tools**: Simple and easy to maintain but lacks interactivity and can become outdated.

Helper libraries (Date, hooks, …)

Title: Helper Methods Library Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure code consistency, efficiency, and maintainability, we need to select a suitable library for helper methods (e.g., utility functions, data manipulation, and common operations). The primary options considered are Lodash, Ramda, and native JavaScript functions.

Decision:

We will compare Lodash, Ramda, and native JavaScript functions based on various criteria such as functionality, ease of use, community support, performance, and integration capabilities. The decision is to use **Lodash** due to its comprehensive set of utility functions, ease of use, and strong community support.

Lodash

Pros:

1. **Comprehensive**: Offers a wide range of utility functions for various use cases, including array and object manipulation, string operations, and more.
2. **Ease of Use**: Simple and consistent API that is easy to learn and use.
3. **Community Support**: Large and active community with extensive documentation and numerous resources.

4. **Performance**: Optimized for performance, ensuring efficient execution of utility functions.
5. **Modular**: Allows importing specific functions to reduce bundle size.

Cons:

1. **Bundle Size**: Full library can add to the bundle size if not used modularly.
2. **Redundancy**: Some functionality may overlap with native JavaScript methods.

Ramda

Pros:

1. **Functional Programming**: Emphasizes a functional programming paradigm with immutability and currying.
2. **Composability**: Functions are designed to be easily composed, promoting cleaner and more maintainable code.
3. **Community Support**: Active community with good documentation and resources.
4. **Modular**: Allows importing specific functions to reduce bundle size.

Cons:

1. **Learning Curve**: Requires understanding of functional programming concepts, which can be a hurdle for some developers.
2. **Performance**: May have performance overhead due to its functional nature and immutability focus.

Native JavaScript Functions

Pros:

1. **Built-In**: No need for external dependencies, reducing bundle size and potential security vulnerabilities.

2. **Performance**: Native implementations are typically highly optimized for performance.
3. **Standards-Based**: Follows ECMAScript standards, ensuring compatibility and future-proofing.

Cons:

1. **Limited Functionality**: Lacks some of the advanced utility functions provided by libraries like Lodash and Ramda.
2. **Consistency**: Requires manually implementing consistent utility functions, which can lead to code duplication and inconsistency.
3. **Development Effort**: Increased development effort to create and maintain custom utility functions.

Consequences:

- **Positive**:
    - **Lodash**: Provides a comprehensive, easy-to-use, and performant set of utility functions with strong community support.
    - **Ramda**: Emphasizes functional programming and composability, promoting cleaner code.
    - **Native JavaScript Functions**: Reduces bundle size and dependency overhead, with highly optimized performance.
- **Negative**:
    - **Lodash**: Can add to the bundle size if not used modularly and may overlap with native methods.
    - **Ramda**: Requires a learning curve for functional programming and may introduce performance overhead.
    - **Native JavaScript Functions**: Limited functionality and increased development effort for consistent utility functions.

Decision:

After evaluating the pros and cons, we have decided to use **Lodash** as the library for helper methods in our project. Lodash provides a comprehensive, easy-to-use, and performant set of utility functions that cover a wide range of use cases. Its strong community support and extensive documentation make it a reliable choice for maintaining code consistency and efficiency.

Configuration:

1. **Basic Setup**:
    - Install Lodash and integrate it with the React application.
    - Use ES6 imports to include only the specific Lodash functions needed, reducing bundle size.
2. **Advanced Configuration**:
    - Explore Lodash's advanced functions and utilities to streamline complex operations and data manipulations.
    - Regularly review and refactor code to leverage Lodash's capabilities effectively.
3. **Performance Optimization**:
    - Use Lodash's modular imports to minimize bundle size.
    - Profile and optimize performance-critical code sections that use Lodash functions.

Alternatives Considered:

1. **Ramda**: Emphasizes functional programming and composability but requires a learning curve and may introduce performance overhead.
2. **Native JavaScript Functions**: Reduces dependency overhead but lacks advanced functionality and requires increased development effort.

Title: Time Utility Library Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To handle date and time manipulation effectively, we need to select a suitable time utility library. The primary options considered are Day.js, Moment.js, Luxon, and date-fns.

Decision:

We will compare Day.js, Moment.js, Luxon, and date-fns based on various criteria such as functionality, ease of use, community support, performance, and integration capabilities. The decision is to use **Day.js** due to its lightweight nature, ease of use, and compatibility with Moment.js.

Day.js

Pros:

1. **Lightweight**: Very small footprint, roughly 2KB, making it ideal for performance-sensitive applications.
2. **Ease of Use**: Simple and straightforward API, similar to Moment.js, making it easy to learn and use.
3. **Compatibility**: Almost fully compatible with Moment.js, making migration easy.
4. **Plugin System**: Modular plugin system allows for extending functionality without increasing the core library size.
5. **Performance**: High performance due to its lightweight nature.

Cons:

1. **Feature Set**: May require additional plugins for advanced functionalities, which can add complexity.
2. **Community Size**: Smaller community compared to more established libraries like Moment.js.

Moment.js

Pros:

1. **Comprehensive**: Extensive feature set for date and time manipulation.
2. **Ease of Use**: Simple and consistent API, widely used and well-documented.
3. **Community Support**: Large community, extensive documentation, and numerous plugins.

Cons:

1. **Size**: Large bundle size (approximately 67KB minified), which can impact performance.
2. **Performance**: Known for performance issues, particularly in large-scale applications.
3. **Maintenance**: Officially in maintenance mode, with no new features planned, only critical fixes.

Luxon

Pros:

1. **Modern API**: Built with modern JavaScript features like Intl, providing a more intuitive and powerful API.
2. **Time Zone Support**: Comprehensive support for time zones and internationalization.
3. **Immutable**: Immutable data structures, promoting safer code.

Cons:

1. **Bundle Size**: Larger than Day.js, though still smaller than Moment.js.
2. **Learning Curve**: Requires learning a new API, which can be a barrier for developers familiar with Moment.js.
3. **Community Size**: Smaller community compared to Moment.js.

date-fns

Pros:

1. **Functional Approach**: Provides a functional programming approach to date manipulation, promoting composability.
2. **Modularity**: Highly modular, allowing the import of only the required functions to minimize bundle size.
3. **Performance**: High performance due to its modular and lightweight nature.
4. **TypeScript Support**: Built-in TypeScript support.

Cons:

1. **Learning Curve**: Requires understanding of functional programming concepts.
2. **Verbosity**: Can be more verbose compared to libraries like Moment.js and Day.js.

Consequences:

- **Positive**:
    - **Day.js**: Lightweight, easy to use, and compatible with Moment.js, making it an excellent choice for performance-sensitive applications.
    - **Moment.js**: Comprehensive feature set and large community support, though in maintenance mode.
    - **Luxon**: Modern API with excellent time zone support and immutability.
    - **date-fns**: Functional approach with high performance and modularity.
- **Negative**:
    - **Day.js**: May require additional plugins for advanced functionalities.
    - **Moment.js**: Large bundle size and performance issues, officially in maintenance mode.

- **Luxon**: Larger bundle size and requires learning a new API.
- **date-fns**: Requires understanding functional programming concepts and can be more verbose.

Decision:

After evaluating the pros and cons, we have decided to use **Day.js** as the time utility library for our project. Day.js provides a lightweight, high-performance solution with an easy-to-use API that is almost fully compatible with Moment.js, making it ideal for our needs.

Configuration:

1. **Basic Setup**:
   - Install Day.js and integrate it with the React application.
   - Use the core library for common date and time operations.
2. **Advanced Configuration**:
   - Add necessary plugins for advanced functionalities like time zone support, duration, and relative time.
   - Configure Day.js to use the required plugins without adding unnecessary overhead.
3. **Migration Strategy**:
   - For projects currently using Moment.js, gradually migrate to Day.js by replacing Moment.js imports and methods with their Day.js equivalents.
   - Leverage Day.js's compatibility with Moment.js to ensure a smooth transition.

Alternatives Considered:

1. **Moment.js**: Comprehensive and well-supported but with a large bundle size and performance issues.
2. **Luxon**: Modern API and excellent time zone support but with a larger bundle size and a steeper learning curve.

3. **date-fns**: High performance and modularity but requires understanding functional programming concepts and can be more verbose.

Testing (libraries, strategies)

Title: Testing Strategy and Library Selection for Nx Monorepo Project

Status: Proposed

Context:

We are developing a React-based project dashboard using Nx for managing our monorepo. To ensure high-quality, maintainable, and reliable code, we need to select an appropriate testing strategy and the corresponding testing libraries. The primary options considered are end-to-end (E2E) testing and unit testing. Additionally, we will evaluate libraries suitable for each strategy.

Decision:

We will compare end-to-end (E2E) testing and unit testing based on various criteria such as time, complexity, coverage, and ease of maintenance. Additionally, we will select the best libraries for implementing the chosen strategy. The decision is to use a combination of both unit testing and end-to-end testing, leveraging Jest for unit testing and Cypress for E2E testing.

End-to-End (E2E) Testing

Pros:

1. **Comprehensive Coverage**: Tests the entire application flow, including interactions between different components and services.

2. **Real-World Scenarios**: Mimics real user interactions, providing high confidence in application behavior.
3. **Reduced Integration Issues**: Identifies issues that arise from the integration of various parts of the application.

Cons:

1. **Time-Consuming**: Takes longer to execute compared to unit tests, especially for large applications.
2. **Complexity**: More complex to set up and maintain, particularly with dynamic data and environments.
3. **Debugging**: Debugging failures can be challenging due to the broad scope of the tests.

Unit Testing

Pros:

1. **Speed**: Executes quickly, providing immediate feedback during development.
2. **Isolation**: Tests individual components or functions in isolation, making it easier to pinpoint issues.
3. **Ease of Maintenance**: Easier to write, understand, and maintain compared to E2E tests.
4. **Granularity**: Provides detailed insights into specific parts of the codebase.

Cons:

1. **Limited Coverage**: Does not test the interactions between components or services, potentially missing integration issues.
2. **Mocking Overhead**: Requires extensive mocking of dependencies, which can be time-consuming and error-prone.

Combined Strategy

Pros:

1. **Balanced Coverage**: Combines the strengths of both strategies, ensuring comprehensive coverage and detailed insights.
2. **Risk Mitigation**: Reduces the risk of integration issues by complementing unit tests with E2E tests.
3. **Efficiency**: Unit tests provide quick feedback during development, while E2E tests ensure end-to-end functionality.

Cons:

1. **Resource Intensive**: Requires investment in both types of tests, increasing the overall testing effort.
2. **Complexity**: Involves managing two testing frameworks and strategies.

Libraries Selection

Unit Testing: Jest

Pros:

1. **Ease of Use**: Simple configuration and extensive documentation.
2. **Performance**: Optimized for speed with features like parallel test execution.
3. **Features**: Built-in mocking, assertion library, and snapshot testing.
4. **Community Support**: Large community and widespread adoption.

Cons:

1. **Learning Curve**: Requires learning Jest-specific configurations and best practices.

End-to-End Testing: Cypress

Pros:

1. **Ease of Use**: Simple setup with an intuitive API.
2. **Real-Time Feedback**: Provides real-time feedback with an interactive test runner.
3. **Comprehensive Documentation**: Extensive documentation and community support.
4. **Robustness**: Handles complex user interactions and provides powerful debugging tools.

Cons:

1. **Resource Intensive**: Can consume significant resources during test execution, especially for large test suites.
2. **Initial Setup**: Requires initial investment in setting up test environments and data.

Consequences:

- **Positive**:
    - **Combined Strategy**: Ensures comprehensive coverage, balances speed and reliability, and reduces integration issues.
    - **Jest for Unit Testing**: Provides fast, isolated tests with detailed insights into specific parts of the codebase.
    - **Cypress for E2E Testing**: Ensures real-world scenario testing with robust, interactive testing capabilities.
- **Negative**:
    - **Combined Strategy**: Requires investment in both types of tests, increasing the overall testing effort and complexity.
    - **Jest**: Learning curve for Jest-specific configurations and best practices.
    - **Cypress**: Resource-intensive and requires initial setup effort.

Decision:

After evaluating the pros and cons, we have decided to adopt a **combined testing strategy** using **Jest for unit testing** and **Cypress for end-to-end testing**. This approach ensures comprehensive test coverage, balancing the speed and reliability of unit tests with the thoroughness of E2E tests.

Configuration:

1. **Jest Setup**:
    - Install Jest and configure it within the Nx monorepo.
    - Write unit tests for individual components and functions, leveraging Jest's mocking and assertion capabilities.
2. **Cypress Setup**:
    - Install Cypress and configure it for the project.
    - Write E2E tests to cover critical user workflows and interactions, ensuring the application works as expected from the user's perspective.
3. **CI/CD Integration**:
    - Integrate both Jest and Cypress into the CI/CD pipeline to ensure tests run automatically on code changes, providing continuous feedback.

Alternatives Considered:

1. **E2E Only**: Provides comprehensive coverage but is resource-intensive and slower, potentially delaying feedback.
2. **Unit Testing Only**: Fast and isolated tests but limited in scope, potentially missing integration issues.