Change Log

| **Revision** | **Submission Date** | **Affected Sections or Pages** | **Change Summary** |
| --- | --- | --- | --- |
| Initial | 2019-02-26 | All | Initial issue of document. |
| Revision | 2019-04-08 | 1, 2, and 6 | Add correct revisions to document numbers. Add NgRX docs link. Add “Software License” column to Table 3. |

Table of Contents

[1 Introduction 1](#_Toc484433983)

[1.1 Identification 1](#_Toc484433984)

[1.2 Purpose 1](#_Toc484433985)

[1.3 Overview 1](#_Toc484433986)

[1.4 Terminology and Notation 1](#_Toc484433987)

[1.5 References **Error! Bookmark not defined.**](#_Toc484433988)

[2 Allocated Requirements 2](#_Toc484433989)

[3 Design Philosophy, Assumptions, and Constraints 2](#_Toc484433990)

[4 Design Framework or Skeleton 3](#_Toc484433991)

[4.1 Program Set Architecture 3](#_Toc484433992)

[4.2 External Interface Design 4](#_Toc484433993)

[4.2.1 External Hardware Interface Design 4](#_Toc484433994)

[4.2.2 Internal Program Set Interfaces 4](#_Toc484433995)

[4.3 Shared Data 4](#_Toc484433996)

[4.4 Requirements Allocation 5](#_Toc484433997)

[4.5 Error Detection and Recovery 5](#_Toc484433998)

[5 Analysis 5](#_Toc484433999)

[5.1 Trade-offs and Decisions 5](#_Toc484434000)

[5.2 Imported Software 5](#_Toc484434001)

[5.3 Technology Profile 6](#_Toc484434002)

Table of Tables

[Table 1: Applicable JPL Rules Documents 1](#_Toc483906730)

[Table 2: Applicable MGSS Documents 2](#_Toc483906731)

[Table 3: Allocated Requirements **Error! Bookmark not defined.**](#_Toc483906732)

[Table 4: Used Technologies 6](#_Toc483906733)

# Introduction

## Identification

| **Property** | **Value** |
| --- | --- |
| Configuration ID (CI) | 621.712 |
| Element | MPSA |
| Program Set | SEQ |
| Version | 2.1.0 |

## Purpose

NEST is a single-page web application (SPA) used for planning, sequencing, and visualization in mission operations. This document describes software design and implementation decisions of NEST.

## Overview

NEST is a subsystem of the Aerie Architecture. The Aerie Architecture is a collection of software systems used for Activity Planning (Merlin), Sequencing (Falcon), Spacecraft Analysis (Eagle), Science Planning (Osprey), and Operations Process Support (Kite). Nest provides a web-based User Interface (UI) for all of the software systems in the Aerie Architecture.

## Terminology and Notation

Aerie – A collection of software systems used for Activity Planning (Merlin), Sequencing (Falcon), Spacecraft Analysis (Eagle), Science Planning (Osprey), and Operations Process Support (Kite).

Angular Component - <https://angular.io/guide/architecture-components>

Angular Module - <https://angular.io/guide/ngmodules>

CBSE – Component-based Software Engineering

CDK – Component Dev Kit (<https://material.angular.io/cdk/categories>)

CLI – Command Line Interface

FP – Functional Programming

FRP – Functional Reactive Programming

IDE – Integrated Development Environment

JSON – JavaScript Object Notation

JSON Schema - <https://json-schema.org/>

MIT License - <https://opensource.org/licenses/MIT>

NPM – Node Package Manager (<https://www.npmjs.com/>)

Programming Paradigm – A style of building the structure and elements of computer programs

SPA – Single Page Web Application (<https://en.wikipedia.org/wiki/Single-page_application>)

Transient Dependencies – Dependencies of dependencies of dependencies…

UI – User Interface

Web Components and Custom Elements - <https://github.com/w3c/webcomponents/>

Table 1: Applicable JPL Rules Documents

| **Title** | **DocID** |
| --- | --- |
| Software Development | 57653 |

Table 2: Applicable MGSS Documents

| **Title** | **Document Number** |
| --- | --- |
| AMMOS Technical Standards Profile | DOC-001101 Rev A |
| MGSS Implementation and Maintenance Task Requirements | DOC-001455 Rev C |
| NEST Operations Concept | DOC-002181 |
| NEST Software Requirements Document (NEST SRD) | DOC-001934 Rev9 |

# Allocated Requirements

Requirements are defined in the NEST Software Requirements Document (DOC-001934).

# Design Philosophy, Assumptions, and Constraints

The NEST software design uses the NgRX Architecture combined with component-based software engineering and functional-reactive programming.

The NgRX Architecture provides centralized state management, isolation of side effects, entity collection management, router bindings, code generation, and developer tools. Its core principles include:

* State is a single, immutable data structure.
* Components delegate responsibilities to side effects, which are handled in isolation.
* Type-safety is promoted throughout the architecture with reliance on TypeScript’s compiler for program correctness.
* Actions and state are serializable to ensure state is predictable stored, rehydrated, and replayed.
* Promotes the use of functional programming when building reactive applications.
* Provides straightforward testing strategies for validation of functionality.

Component-based software engineering (CBSE) emphasizes separation of concerns with respect to the wide-ranging functionality throughout a given software system. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software.

Functional-reactive programming (FRP) is a programming paradigm for reactive programming. Reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change. It is sometimes called asynchronous dataflow programming. FRP uses the building blocks of functional programming (e.g. map, reduce, filter) to express and operate on data streams.

Functional programming (FP) is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. This makes reasoning about the overall state of the software much easier since the functions used in the application don’t have side-effects (no mutation of local static variables, non-local variables, mutable reference arguments or I/O streams).

# Design Framework or Skeleton

## Program Set Architecture

The core of NEST is built using the NgRX Architecture. The NgRX Architecture is a centralized store architecture, where the entire application state is kept in a single location called a store. The logic of every NgRX Architecture program breaks down into five cleanly separated parts:

1. Actions
2. Reducers
3. Effects
4. Selectors
5. Components (Containers and Presentational)

Actions describe unique events in the system. They are typically dispatched from components, services, or Effects. For example, if a user clicks on a button somewhere in the component tree, we could dispatch an action called “ClickedButton”.

State change is handled by pure functions called reducers. Reducers take the current state and the latest action as arguments, and return a new state for the application based on those arguments. For example if the state for an application is a single variable called “clickCount”, we could call a reducer with the signature: reducer(clickCount, ClickedButton) -> ++clickCount. We get a brand-new state based on the old state and the action.

Effects handle side-effects in the NgRX architecture. They use FRP to provide new sources of actions to reduce state based on external interactions like network requests or time-based events. For example, if an Effect is created for the “ClickedButton” action, the Effect would be called after the reducer has updated the state, and we could return another action from the Effect like “NotifyButtonClicked” which could show a popup notifying the user of the clicked button. Effects all follow the same pattern of: actions in, actions out; the Effect gets an action, and always return 0 or more actions which start the cycle over again.

Selectors are pure functions that are used to select, derive, and compose piece of state. For example, we could create a selector that returns a “clickCount” state. Selectors are useful because they memorize our state selection, and are only called when the state we want actually changes.

There are two types of Angular components that are used in NEST: containers and presentation:

1. Containers are Angular Components responsible for connecting to the NgRX Store to select state and dispatch events. You can think of a container as being tightly coupled to its application. It would be hard to move a container to use in another application.
2. Presentation components (or pure components) are Angular Components that have no knowledge of the NgRX store. A presentation components state is passed in through its Inputs, and anytime it needs to communicate with a parent it dispatches an Output. It is up to container components to catch these Outputs if dispatching to the Store is needed. It should be easy to move a pure component into another application with little to no work at all. In fact, we compile presentation components to Web Components via Angular Elements for use in other applications as needed.

See Figure 1 blow for a complete picture of the NgRX Architecture data flow.

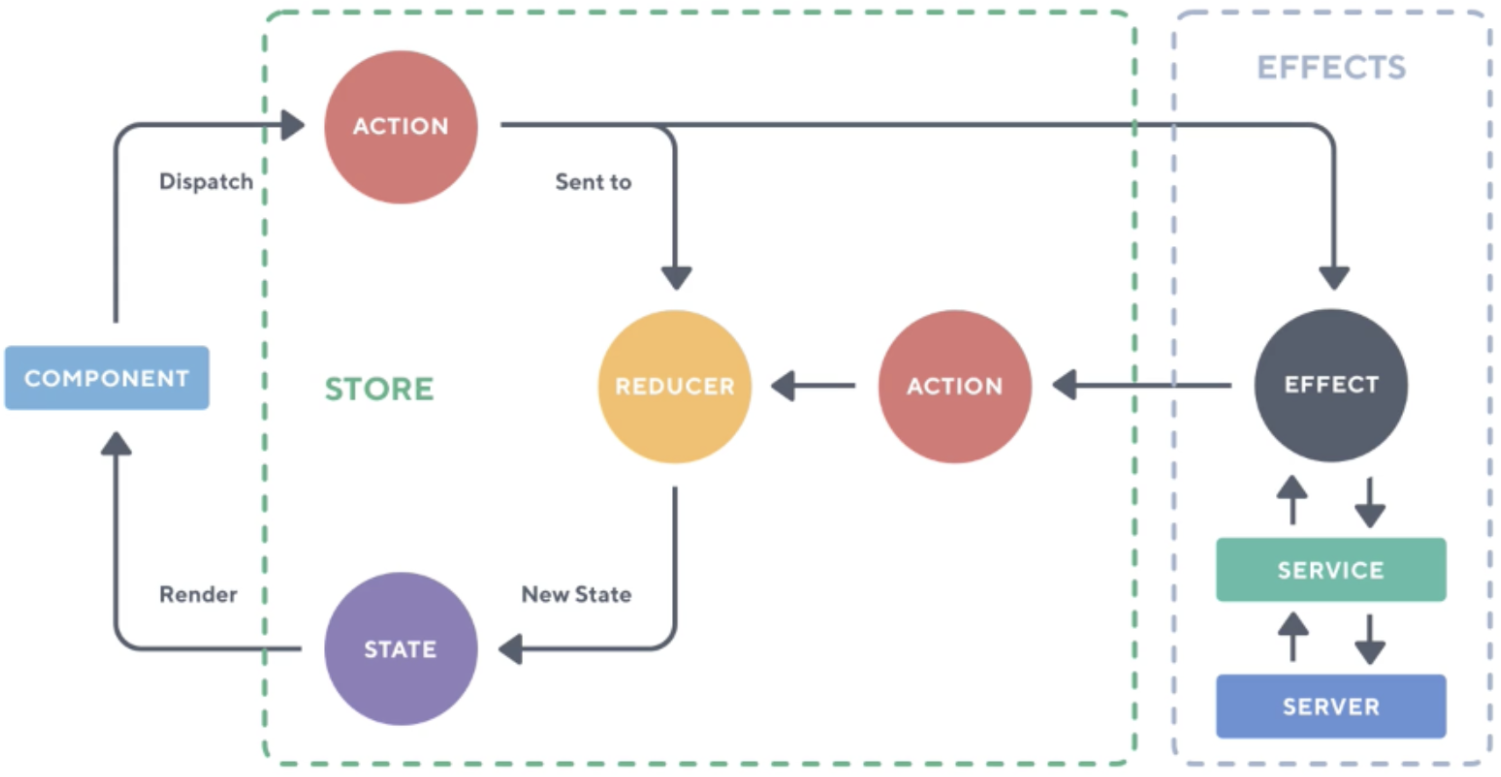


Figure 1 (NgRX Architecture)

## External Interface Design

### External Hardware Interface Design

There is no external hardware interface associated with this software.

#### External Software Interface Design

#### Aerie Interface Design

NEST is built to interface with the Aerie services. All software interfaces are defined using JSON Schema. A list of these interfaces can be found here: <https://github.jpl.nasa.gov/MPS/aerie/tree/develop/schemas>

JSON Schema was chosen because the Aerie services use Java as the main programming language, and NEST uses TypeScript as the main programming language. We use JSON Schema as an intermediate “universal” language that can compile to Java and TypeScript so all of our systems use the same interfaces. This helps insure data consistency across the different sub-systems of Aerie.

A more detailed description of the Aerie Interface Design can be found in the Aerie SDD.

### Internal Program Set Interfaces

Internally, NEST components communicate via JSON (JavaScript Object Notation). All internal messages exchanged between the NEST components are defined as TypeScript interfaces. You can find the complete set here: <https://github.jpl.nasa.gov/MPS/aerie/tree/develop/nest/src/app/shared/models>

## Shared Data

Any shared data between NEST subsystems is kept in a “shared” directory in the Aerie/NEST repository: <https://github.jpl.nasa.gov/MPS/aerie/tree/develop/nest/src/app/shared>. This directory contains any actions, pure components, effects, models, reducers, services, or utility functions that can be used freely across NEST subsystems.

## Requirements Allocation

There are no derived design requirements.

## Error Detection and Recovery

Errors mostly commonly arrive to NEST through the event boundaries defined by the NgRX Effects. When an NgRX Effect is invoked, we are typically expecting the application to perform some sort of asynchronous side-effect.

The most common side-effect is an HTTP Request, which can fail for a number of reasons:

1. Server availability
2. Network connectivity
3. Misaligned client-server interface

We detect NgRX Effect errors using the RxJS “catchError” function (<https://www.learnrxjs.io/operators/error_handling/catch.html>). This is similar to a try-catch construct commonly used in synchronous programming. If an error occurs in an Effect the catch is called. In the catch we both report and gracefully recover from the error.

# Analysis

## Trade-offs and Decisions

The main trade off the NEST design makes is using a centralized state (via NgRX) vs a non-centralized state. This means the entire application state is co-located in a single place, and updates to that state are made through a strict interface via “actions”. In this architecture, portions of the state that you want to mutate are not actually mutated, where instead an entire new state is returned when an update is made. This ensures no race-condition-like side effects occur when mutating state.

This design offers a trade-off. It asks you to describe application state as plain objects and arrays, describe changes in the system as plain objects, and describe the logic for handling changes as pure functions. These are strong constraints but there are good reasons for adopting such constraints. These include providing alternate UIs while reusing most business logic, time traveling between state history in development and re-evaluating the current state from the action history when the code changes, maintaining an undo history, passing action objects over the network to implement collaborative environments, serializing user actions with a state snapshot for automated bug reports so that developers can replay them to reproduce errors, prefilling state on the server, and easily persisting application state. The tradeoff that NgRX offers is to add indirection to decouple “what happened” to “how things change”, and it has allowed us to more easily reason about our application state as it changes over time.

## Imported Software

All third-party dependencies are downloaded via NPM. You can find a complete list of these dependencies here: <https://github.jpl.nasa.gov/MPS/aerie/blob/develop/nest/package.json>.

NPM dependencies are broken down into production dependencies and development dependencies. NEST absolutely cannot run without production dependencies, and only uses development dependencies during development. It is important that both production dependencies and development dependencies are installed before development. See the NEST Readme for more information on how to install the dependencies: <https://github.jpl.nasa.gov/MPS/aerie/blob/develop/nest/README.md>.

None of the NPM dependencies are tested explicitly in NEST, and we instead rely on the third-party software authors to test their software before we use it. We always make sure an NPM dependency is tested and well-maintained before using it in NEST. We also make frequent use of NPM’s auditing tool (<https://docs.npmjs.com/cli/audit>) to make sure all dependencies and their transient-dependencies have no known security vulnerabilities.

## Technology Profile

Table 3: Used Technologies

| **Technology Area** | **Technology Description** | **Technology Standards** | **Software License** |
| --- | --- | --- | --- |
| Angular | Angular is an open-source (MIT Licenced) framework used for building single-page applications. It has a wide set of features that enable rapid creation of robust and safe software that scales. | <https://angular.io/> | MIT - <https://github.com/angular/angular/blob/master/LICENSE> |
| TypeScript | A programming language that is a typed superset of JavaScript that compiles to plain JavaScript. This allows NEST to take advantage of all the features a statically typed language has to offer in the web browser. Some of these advantages include:   * Strict software interfaces * Code completion and hinting in your IDE * Performance optimizations * Compiler errors that help you catch bugs early | <https://www.typescriptlang.org/> | Apache License Version 2.0 - <https://github.com/Microsoft/TypeScript/blob/master/LICENSE.txt> |
| Angular CLI | A command-line interface that automatically creates, configures, tests, builds, serves, and updates your application. Without the CLI all of these processes would have to be created manually by the developers, which is both time consuming and error prone. Some example commands from the CLI are:   * *ng build*: Development and optimized production builds. * *ng serve*: Starts a development server and automatically recompiles/reloads the application when project files change. * *ng test*: Run all unit tests in the application. * *ng new*: Create a new Angular application with the recommended directory structure. | <https://cli.angular.io/> | MIT - <https://github.com/angular/angular-cli/blob/master/LICENSE> |
| Karma | Unit test runner | <https://karma-runner.github.io/latest/index.html> | MIT - <https://github.com/karma-runner/karma/blob/master/LICENSE> |
| Jasmine | Unit test spec language | https://jasmine.github.io/ | MIT - <https://github.com/jasmine/jasmine/blob/master/MIT.LICENSE> |
| Protractor | An end-to-end test framework for Angular. It runs tests against your application running in a real browser, interacting with it as a user would. | https://www.protractortest.org/#/ | MIT - <https://github.com/angular/protractor/blob/master/LICENSE> |
| TSLint | An extensible linter for the TypeScript language. It automatically checks TypeScript code for readability, maintainability, and functionality errors. | <https://palantir.github.io/tslint/> | Apache License Version 2.0 - <https://github.com/palantir/tslint/blob/master/LICENSE> |
| Codelyzer | A set of TSLint rules used for statically analysing Angular projects. | https://github.com/mgechev/codelyzer | MIT - <https://github.com/mgechev/codelyzer/blob/master/LICENSE> |
| Angular Router | Enables navigation from one view to the next as users perform application tasks. This also allows for lazy loading (<https://angular.io/guide/lazy-loading-ngmodules>) Angular modules, which means the application only has code in the browser it absolutely needs at any given time. When a new module is needed, it is requested and loaded in the browser. This decreases overall application download time and makes the application more performant on start-up. | <https://angular.io/guide/router> | MIT - <https://github.com/angular/angular/blob/master/LICENSE> |
| Style Guide | An opinionated guide to Angular syntax, conventions, and application structure. | https://angular.io/guide/styleguide | MIT - <https://github.com/angular/angular/blob/master/LICENSE> |
| RxJS | API for asynchronous programming with observable streams. | http://reactivex.io/ | Apache License Version 2.0 - <https://github.com/ReactiveX/rxjs/blob/master/LICENSE.txt> |
| NgRX | Reactive state management for Angular. | https://ngrx.io/docs | MIT - <https://github.com/ngrx/platform/blob/master/LICENSE> |
| Angular Elements | Package Angular Components as Custom Elements for reuse. This allows for other web-based applications not using Angular to consume our components. | https://angular.io/guide/elements | MIT - <https://github.com/angular/angular/blob/master/LICENSE> |
| Angular Material | Material design components for angular. | https://material.angular.io/ | MIT - <https://github.com/angular/material2/blob/master/LICENSE> |
| CDK | The Component Dev Kit is a set of tools that implement common interaction patters whilst being unopinionated about their presentation. This includes a virtual scroll and drag-and-drop component that can be easily used in other components. | https://material.angular.io/cdk/ | MIT - <https://github.com/angular/material2/blob/master/LICENSE> |
| Augury | A Chrome and Firefox Dev Tools extension for debugging Angular applications. | https://augury.rangle.io/ | MIT - <https://github.com/rangle/augury/blob/master/LICENSE> |
| Redux Dev Tools Extension | Time-travel debugging for the NgRX  store. | https://github.com/zalmoxisus/redux-devtools-extension | MIT - <https://github.com/zalmoxisus/redux-devtools-extension/blob/master/LICENSE> |
| Prettier | Opinionated code formatter. Code is automatically formatted so you don’t need to discuss style in code review. Saves team time and energy. | https://prettier.io/ | MIT - <https://github.com/prettier/prettier/blob/master/LICENSE> |
| Compodoc | A documentation tool for Angular applications. It generates a static documentation of your application. | https://compodoc.app/guides/getting-started.html | MIT - <https://github.com/compodoc/compodoc/blob/develop/LICENSE> |