

*Multimission Ground Systems and Services*

RAVEN Software Design Document (SDD)

Change Log

| Revision | Submission Date | Affected Sections or Pages | Change Summary |
| --- | --- | --- | --- |
| Initial | 04-04-2017 | All | Initial issue of document. |
| First Revision | 04-05-2017 | All | Revised sections and its contents to address feedback provided. |
| Rev C | 08-14-2017 | All | Modified architecture to truly reflect the design behind RAVEN. |

Table of Contents

Table of Figures iii

Table of Tables iv

1 Introduction 5

1.1 Identification 5

1.2 Purpose 5

1.3 Overview 5

1.4 Stakeholders 5

1.5 Terminology and Notation 6

1.6 References 7

2 Allocated Requirements 7

3 Design Philosophy, Assumptions, and Constraints 8

3.1 Architectural Drivers 10

4 Design Framework 12

4.1 Program Set Architecture 12

4.2 External Interface Design 15

4.2.1 External Hardware Interface Design 15

4.2.2 External Software Interface Design 15

4.2.3 Graphical User Interface (GUI) Design 16

4.3 Internal Program Set Interfaces 21

4.4 Shared Data 22

4.4.1 Source Manager 23

4.4.2 Resource Data Handler 23

4.4.3 Data Renderer 25

4.4.4 Action Handler 25

4.4.5 Display Manager 26

4.5 Error Detection and Recovery 27

5 Analysis 27

5.1 Trade-offs and Decisions 27

5.2 Imported Software 30

5.3 Technology Profile 31

5.4 Design Feasibility, Performance and Margins 33

5.5 Reliability, Maintainability, and Related Concerns 34

5.6 Adaptability and Reusability 35

6 Detailed Design 35

## Table of Figures

Figure 1. JavaScript Memory Heap 10

Figure 2. ReactJS Virtual DOM from https://auth0.com/blog/face-off-virtual-dom-vs-incremental-dom-vs-glimmer/ 11

Figure 3. RAVEN in relationship SEQ Subsystems and Other Components 13

Figure 4. Redux 14

Figure 5. RAVEN Sequence Diagram 15

Figure 6. RAVEN Copyright Display 17

Figure 7. RAVEN Copyright Display 18

Figure 8. RAVEN UI map 19

Figure 9. Resource Band 24

Figure 10. State Band 24

Figure 11. Activity Band 24

Figure 12. Composite Band 24

Figure 13. ITAR message 27

Figure 14. Composite Design Pattern 33

Figure 15. RAVEN Detailed Design with Redux Architecture 36

## Table of Tables

Table 1: Stakeholders’ Interests 6

Table 2: Stakeholders’ Interests 7

Table 3: Applicable JPL Rules documents 8

Table 4: Applicable MGSS documents 8

Table 5: Allocated Requirements 8

Table 6: Trade-Offs and Decisions matrix 27

Table 7: Flux Vs Redux 29

Table 8: Technology Profile 30

# Introduction

## Identification

| **Property** | **Value** |
| --- | --- |
| Element | MPSA |
| Program Set | RAVEN |
| Version | 34.5.0 |

## Purpose

This document describes the architectural approach to the Resource and Activity Visualization ENgine application, also known as **RAVEN**. The design is based on the SRD. In addition, it forms the basis for planning the development cycles and it will be updated using the results from the various development iterations to keep the architectural design in synch.

## Overview

RAVEN is a web-based application included in the SEQ subsystem of the Advanced Multimission Operations System (AMMOS) and managed by the Mission Planning, Sequencing and Analysis (MPSA) element. It allows users to view science planning, spacecraft activities, resource usage and predicted data, or any time-based data, displayed in a timeline format via web browser. Subsequently, it can be viewed simultaneously by distributed users/teams for collaboration when creating, updating and validating activity plans and command sequences.

## Stakeholders

It is important to identify the person, group or organization that has interest or concern in RAVEN. They can affect or be affected by the actions, objectives and policies the team decides to take. This section identifies these stakeholders and briefly describes their problems or constraints.

Table : Stakeholders’ Interests

|  |  |  |
| --- | --- | --- |
| Stakeholder | Description | General Interest |
| MGSS | Provide common functionality in the same software product without forking deliverables for specific mission situations. More importantly, fitting the work within allocated budget and schedule. | * Maintainability * Cost * Portability |
| Flight Projects | Share specific views amongst project personnel. | * Stability * Interoperability * Performance * Usability * Deployability * Security |
| Visualize environment elements and events with respect to time in order to comprehend their meaning. |
| A user can visualize and share the same plan in multiple work stations. |
| Developers | Add new features and capabilities based on user’s feedback and strategic goals.  Provide support to users for the application. | * Maintainability * Efficiency (related to speed of runtime execution for software) * Separation of Concerns |
| MPSA QA Team | Perform new feature, regression, integration testing to ensure quality in the deliverable product. | * Testability * Usability |

## Terminology and Notation

Table : Stakeholders’ Interests

|  |  |
| --- | --- |
| Acronym or Abbreviation | Definition |
| AMMOS | Advanced Multimission Operations System |
| APGen | Activity Plan Generator |
| API | Application Program Interface |
| CAM | Common Access Manager |
| CSV | Comma Separated Value |
| CTL | Canvas Timeline Library |
| DMS | Document Management System |
| DOM | Document Object Model |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| MGSS | Multimission Ground System & Services |
| MPSA | Mission Planning, Sequencing & Analysis |
| MPSServer | Mission Planning & Sequencing Server |
| MPSViewer | A wrapper to the CTL library. It manages rows in the chart. Also, it transforms data from MPSServer to the data format needed for CTL. |
| RAVEN | Resource and Activity Visualization ENgine |
| REST | Representational State Transfer |
| SEQ | Sequencing Subsystem |
| SEQGen | Sequence Generator |
| TMS | Timeline Management System |
| XML | Extensible Markup Language |
| XMLTOL | Extensible Markup Language Time-Ordered Listing (APGen) |

## References

Table 3: Applicable JPL Rules documents

| Title | DocID |
| --- | --- |
| Software Development | 57653-Rev 9 |
| Document Management | 78420-Rev 0 |
| Overview of Software Development Standard Processes | 78187-Rev 0 |

Table 4: Applicable MGSS documents

| Title | Document Number |
| --- | --- |
| AMMOS Technical Standards Profile | DOC-001101, Rev A |
| MGSS Implementation and Maintenance Task Requirements | DOC-001455, Rev B |
| *MGSS Document Structure, Standards, and Definitions* | DOC-000016, Rev C |
| *AMMOS Architecture Principles* | DOC-000485, Rev A |
| *Mission Planning and Sequencing (MPS) Subsystem Software Management Plan* | DOC-000178 |
| *RAVEN Software Requirements Document (SRD)* | DOC-001498, Rev E |
| *RAVEN User’s Guide* | DOC-001507, Rev E |
| *MPSServer Web Services Software Interface Specification*  *(SIS)* | DOC-001472, Rev E |
| *APGen User’s Guide* | DOC-000429, Rev H |
| *SEQGEN for Multimission Sequence (SEQ) User’s Guide* | DOC-000426, Rev H |

# Allocated Requirements

Table 5: Allocated Requirements

| **ID** | **Requirement Statement** |
| --- | --- |
| MW-01 | RAVEN shall retrieve data to be displayed from one or more user-selected databases. |
| MW-05 | RAVEN shall display up to 100 rows of data simultaneously in the same chart when requested by the user |
| MW-06 | RAVEN shall display 10,000 points for a single resource within 20 seconds |
| MW-09 | RAVEN shall provide an interface for remote access through a web browser. |
| MW-10 | RAVEN shall provide a copyright statement that may be viewed by the user |
| MW-11 | RAVEN shall have an option to perform user authentication before allowing a user to access or display any data. |
| MW-30 | RAVEN shall operate on the RedHat Linux Enterprise operating system |
| MW-27 | RAVEN shall filter data sources based on user input |
| MW-28 | RAVEN shall encrypt all communication with the server |
| MW-34 | RAVEN shall manage display configurations. |
| MW-46 | RAVEN shall change the displayed time to show the format specified by the user. |
| MW-48 | RAVEN shall display an indicator that represents current time |
| MW-53 | RAVEN shall configure the structure of data in a user-selected database. |
| MW-60 | RAVEN shall display an ITAR Marking Language |
| MW-63 | RAVEN shall provide an option to export a timeline to different formats. |
| MW-64 | RAVEN shall display activities, sequences, commands, and events (hereafter referred to collectively as "plan elements") with start times and end times on a timeline. |
| MW-65 | RAVEN shall plot sets of numerical values, each associated with a time. |
| MW-66 | RAVEN shall display sets of string values, each associated with a time. |
| MW-67 | RAVEN shall temporally align the information displayed in all rows |
| MW-68 | RAVEN shall allow me to work with annotations in the time lanes |

When a symbol like this is visible: It means it is addressing a requirement.

The requirement ID will be indicated inside the sticker for reference.

# Design Philosophy, Assumptions, and Constraints

RAVEN architectural design is created to adhere to established design patterns, as initially defined by the Gang Of Four[[1]](#footnote-1), so we can fulfill the following expectations from the system:

* Ease of maintenance. When a component has to change, or be updated, it must not break other parts of the system. Instead, it must follow a common established interface.
* Separation of Concerns. These are different aspects of software functionality. The separation is keeping the code for each functionality decoupled. Changing the interface should not require changing the business logic code, and vice-versa.

RAVEN is designed to be a light-weight and multi-browser compatible web application. It allows multiple users to view the same data set at the same time from different systems. It displays resources from multiple data sources to allow flexibility in modelling.

To achieve its design at implementation, it uses a recent architectural concept called virtual Document Object Model (DOM). It efficiently re-renders the components that have had their state changed, without reloading the whole application. In RAVEN, this can be applied throughout the application. The application will render in a plot a graphical representation of the data retrieved through MPSServer. If one of the data points changes, it would be very costly to compute the whole DOM again. Rather than doing that, by using the virtual DOM concept, the application can do a diff for the changes, patch it and only update the element on the tree that requires it.

The following assumptions have been made in the design of RAVEN:

1. A web browser is available on the user machine.
2. Access to a running MPSServer.
3. Access to a running Common Access Manager (CAM) server (when CAM is used).

RAVEN software design decisions considered the following constraints:

1. Constraining requirements listed in the RAVEN Software Requirements Document (SRD)
2. Performance impact by quantity of data items being plotted.
3. Integration into the SEQ subsystem cannot disrupt the delivery process of the current subsystem and it needs to be compatible with currently supported SEQ applications

The architecture design is focused on delivering a smooth experience using mainly JavaScript. In order to fulfill the charting and performance requirements we are using the Canvas Timeline Library (CTL).

A performance issue can be caused by filling the JavaScript memory with too many objects too quickly. JavaScript manages the memory allocation. Nevertheless, the downside to this automatic collection is that it takes some processing power from the browser to execute. This means that there is less power to keep up with the framerate running smoothly, causing animations to slow down.

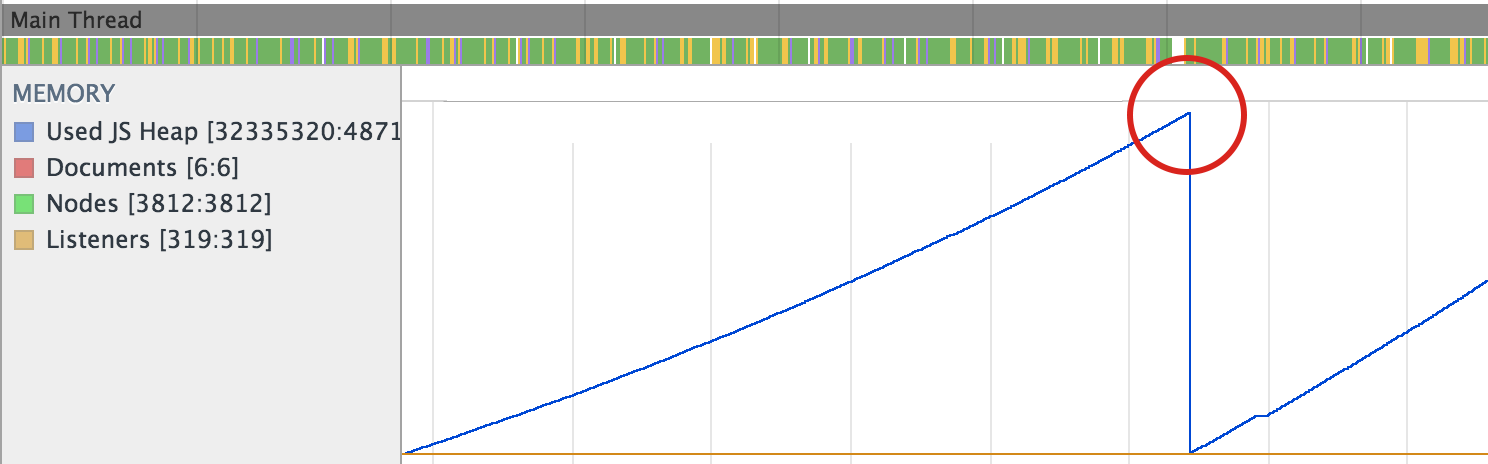


Figure 1. JavaScript Memory Heap

To minimize the impact, RAVEN will be reusing created objects as much as possible. As a consequence, we will reduce the frequency at which the garbage collection will have to run, resulting in better performant plotters.

Another important concept to understand is the use of the Virtual DOM by ReactJS. The HTML DOM is always tree-structured, which makes it easier to traverse. Unfortunately, easily doesn’t mean quickly. For an application like RAVEN, the DOM trees can be huge because of all the data it has to represent. In addition, the DOM tree needs to be modified incessantly and a lot. This represents a performance issue. This is where the capabilities of React and its use of the Virtual DOM are used.

It is an abstraction of the HTML DOM, lightweight and detached from the browser-specific implementation details. It allows to do its computations within this abstract world and skip the “real” DOM operations, often slow and browser-specific.

The central piece of Virtual DOM is its smart diffing algorithm: once the differences in the model have been mapped to the in-memory copy of the DOM, the algorithm finds the minimum number of operations required to update the real DOM. This results in two copies of the in-memory DOM being present during the diffing process.

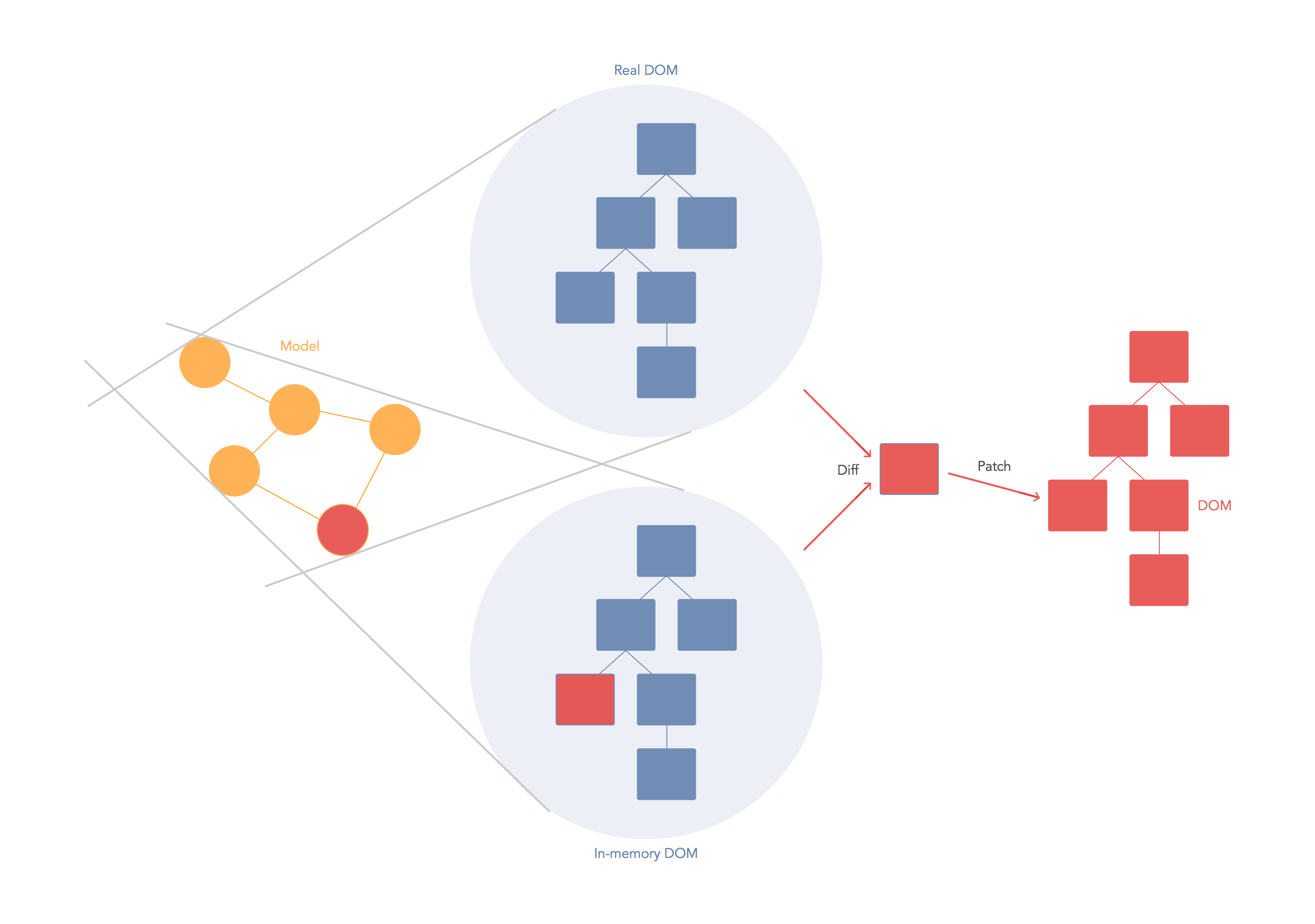


Figure 2. ReactJS Virtual DOM from <https://auth0.com/blog/face-off-virtual-dom-vs-incremental-dom-vs-glimmer/>

## Architectural Drivers

The following drivers where taken from AMMOS Architecture Principles (MGSS DOC- 000485, Rev.A). These principles are used as the guidelines for the design and implementation of RAVEN. Their purpose is to guide the decision making when competing priorities or requirements need to be addressed.

| Principle Name | Principle Description |
| --- | --- |
| 1. Primacy of Principles | These core principles apply to all organizational entities that are responsible for the management and delivery of the AMMOS. |
| 2. Use of Common Services | AMMOS provides operations capabilities via adaptable, loosely-coupled common services. These expose adaptable portions to customers while maintaining key common aspects as part of the Multimission system. |
| 3. Customer Focus | AMMOS exists in order to provide operations capabilities that enable Missions to operate at lower total cost to NASA, while satisfying needs for reliability and performance. |
| 4. Learn from Experience | For the AMMOS to improve and/or maintain capabilities in the face of imperfection and a changing world, it must be maintained and improved over time. This requires intentional effort to acquire and apply the lessons that experience offers. |
| 5. Close the Loop | AMMOS enables closed-loop control of flight assets, including reconciling the reported state of a flight system with science and engineering plans. To enable missions to efficiently and effectively perform planned-to-actual reconciliation, it is necessary to identify, architect, design, build, maintain, and operate elements of the System that respond to this principle. |
| 6. Data/Information Visibility, Accessibility, and Understandability | Data/Information is defined externally to any given user (including software systems or services) and is readily visible, accessible, and understandable to all authorized AMMOS and external partner users, software systems, and services. |
| 7. Technology/Platform Independence | AMMOS software systems and its interfaces are architected independent of specific technology choices and therefore can operate on a variety of technology platforms. |
| 8. Interoperability | Data/Information is defined externally to any given user (including software systems or services) and is readily visible, accessible, and understandable to all authorized AMMOS and external partner users, software systems, and services. |
| 9. Universality of Data/Information Security | The confidentiality, integrity, and availability of data/information in all components of the AMMOS are a first-class concern and not designed into AMMOS software systems and services as an afterthought. |

# Design Framework

## Program Set Architecture

RAVEN is a ReactJS[[2]](#footnote-2) application that uses the JavaScript Canvas[[3]](#footnote-3) library to display any time-based data resource provided in either comma separated values (CSV) or JavaScript object notation (JSON) adhering to the MPSServer schema. It is designed to be deployed and tested in a Linux RedHat operating system.

Requests to retrieve data are made from RAVEN to the MPServer RESTful API. This works as an abstraction layer to the data stores. After data is retrieved RAVEN represents it in a timeline style. RAVEN provides interfaces defined in the MPSServer SIS that return resources with mission defined document-based or relational databases. In the figure below, RAVEN is shown in relationship with other products it interfaces directly and indirectly. This will provide context for the architecture of the application.

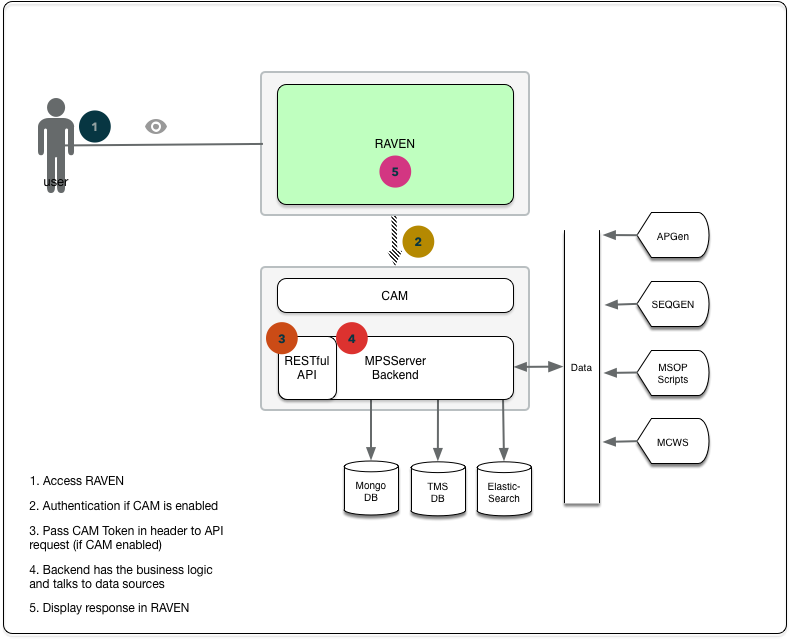


Figure 3. RAVEN in relationship SEQ Subsystems and Other Components

Requirements Addressed:

|  |  |
| --- | --- |
| ID | Requirement Statement |
| MW-09 | RAVEN shall provide an interface for remote access through a web browser. |
| MW-01 | RAVEN shall retrieve data to be displayed from one or more user-selected databases. |
| MW-11 | RAVEN shall have an option to perform user authentication before allowing a user to access or display any data. |

RAVEN utilizes a new paradigm called Flux architecture, where a dispatcher acts as traffic controller. The implementation chosen is called Redux[[4]](#footnote-4). This architecture was chosen because it makes the complicated parts, like state management, more predictable and easier to understand when a new developer is on-boarded. It also makes it easier to maintain since all states live in one single structure.

Flux is an architecture, not a framework or a library. It is simply a new kind of architecture that complements React and the concept of Unidirectional Data Flow. Flux has four main components: Actions, Dispatcher, Stores and Controller Views.

* Actions – Helper methods that facilitate passing data to the Dispatcher
* Dispatcher – Receives actions and broadcasts payloads to registered callbacks
* Stores – Containers for application state & logic that have callbacks registered to the dispatcher
* Controller Views – React Components that grab the state from Stores and pass it down via props to child components.

RAVEN also utilizes Redux, which is an evolution of Flux, avoiding complexity by taking the core idea that the code is built around a Model of the state, a way to update the model and a way to view it.

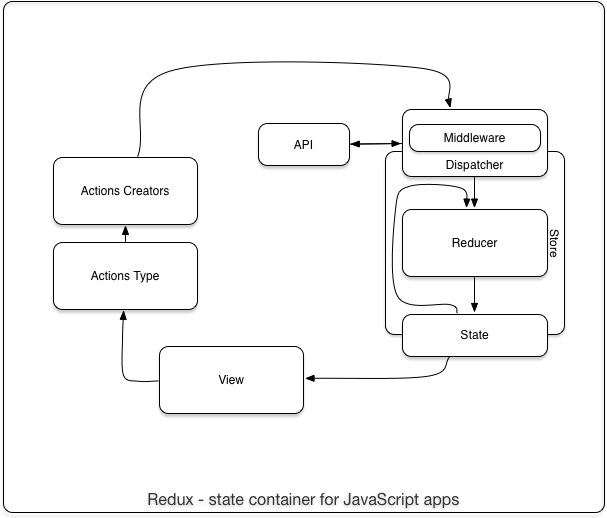


Figure 4. Redux

In Redux, a single data store for an entire application is great since it is saved in a single object. The components don't change that state directly, instead the components rely on the framework to change a global state. If the user changes something, component simply fires an event, and the state object catches and handles it.

Lastly, RAVEN supports the use of decimation to reduce the original sampling rate of a sequence to a lower rate. The decimation process filters the input to guard against aliasing and down samples the result.

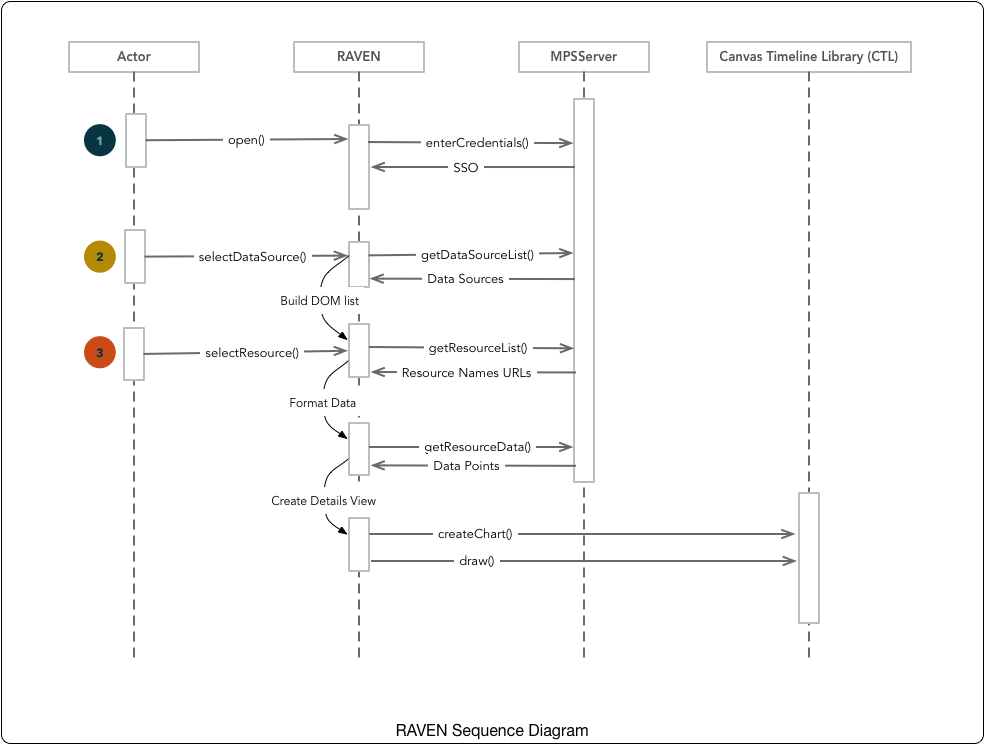
## External Interface Design

### External Hardware Interface Design

None.

### External Software Interface Design

RAVEN only has one external software interface; MPSServer abstracts all interactions with data sources through a RESTful layer. The interactions are shown in the following sequence diagram.

  
Figure 5. RAVEN Sequence Diagram

1. The user opens RAVEN in the browser. If CAM is enabled, the user enters credentials to authenticate and a token is generated. CAM is an intermediary in the operation by authenticating that the user is who it claims to be.
2. The user selects a data source to display from the data sources list. The application builds a DOM with the response of the service call to MPSServer.
3. The user selects a resource to visualize.

Data can be imported from various project datastores:

* MongoDB
* Relational DB
* ElasticSearch
* TMS

RAVEN relies on MPSServer as the backend to communicate to the above-mentioned datastores; supported data queries and ingestion types are:

* Any timed data in CSV or JSON format
* Plan data from APGen in XMLTOL
* PEF and Predicted telemetry generated by SEQGen
* Actual telemetry from MCWS (Mission Control Web Service –AMPCS)
* Sequence Tracker

### Graphical User Interface (GUI) Design

The project itself is developed under the ReactJS framework and uses Babel to transpile JavaScript from ES6 to ES5 specification. This is to support browsers that are ES5 compatible only.

ReactJS is a framework that guides the developer into practicing good design patterns. Taking advantage of these patterns allows for a rich, yet performant graphic user interface. These patterns are meant to be used as guidelines to influence implementation decisions. Among the patterns to be used in RAVEN we can list:

* **Composition**. Components written by different people should work well together. It is important that you can add functionality to a component without causing rippling changes throughout the codebase. In RAVEN, to achieve this, every component behaves like a black box, expecting input and producing output. The application achieves complex visualization by connecting these components together to accomplish a requirement. The same components might be used independent of each other.
* **Common Abstraction**. Resist adding features that can be implemented only for a particular use case. The application should not be bloated. For example, implementing a details table. A table should be a component that looks and feels the same regardless of what data is displaying. It also must be implemented once and reused in every instance where displaying tabular data is required. To accomplish this, the same table should be instantiated depending on the context it is being used.
* **Higher-order Components.** Higher-order components look really similar to the [decorator design pattern](http://robdodson.me/javascript-design-patterns-decorator/). It is wrapping a component and attaching some new functionalities or properties to it. This gives us control on the input. The data that needs to be sent as properties. In RAVEN, the application wraps basic components to aggregate functionality.
* **One-way direction data flow**. One-way direction data flow is a pattern that works nicely with React. It is around the idea that the components do not modify the data that they receive. They only listen for changes in this data and maybe provide the new value but they do not update the actual data store. This update happens following another mechanism in another place and the component just gets rendered with the new value.

#### Software Interface Design

Although RAVEN is written as a single JavaScript application, it can be logically broken up into functional components.

1. Source Manager – Gets data sources and build source tree with checkboxes for each source.
2. Display Manager – Creates Details Panel for data set.
3. Resource Data Handler – Gets data from MPSServer when resource boxes are checked.
4. Data Renderer – Invokes the CTL to create a new chart if chart does not exist. Otherwise, invoke the CTL to add new row.
5. Action Handler - Determines the plot type based on the data type

The user interface design aims at providing a familiar set of tools to an engineer or scientist to map data points into charts in relationship to time. RAVEN follows the western reading paradigm where we start from left to right and top to bottom. The starting point is the explorer panel, where the user selects a data source to load. Immediately after data loads, it is added to the center panel, where a timeline is displayed to provide time context. The user from this point onward can modify or customize different display options, including specifying the time format.

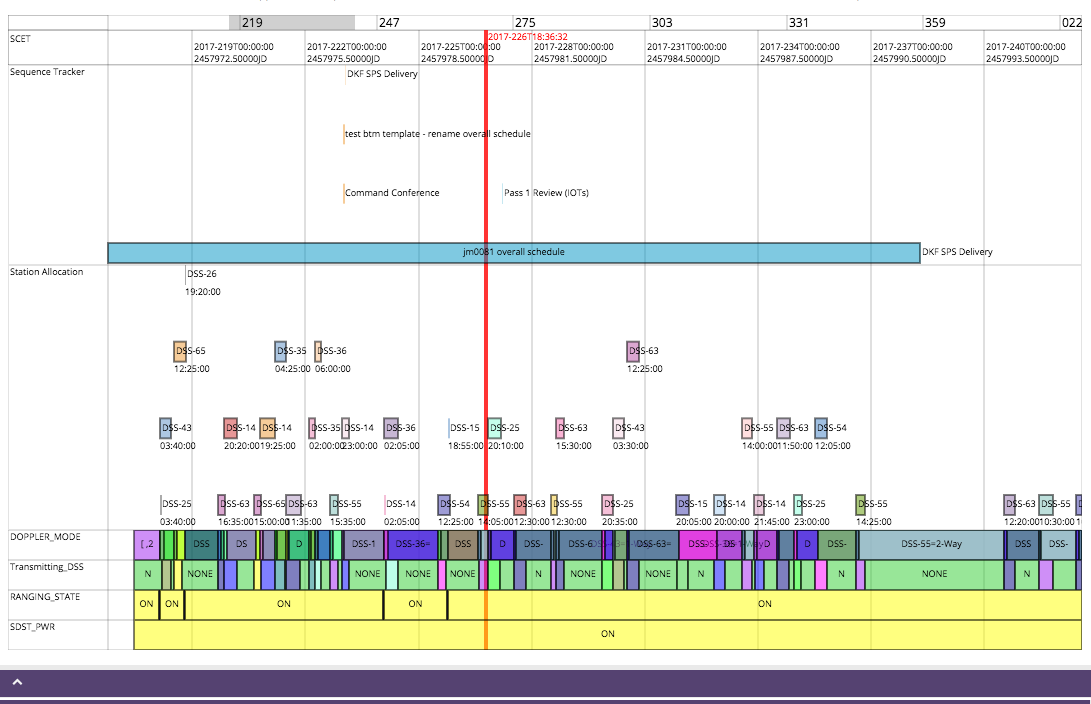


Figure 6. RAVEN Current Time Indicator

A copyright statement is displayed in the information section of the interface:

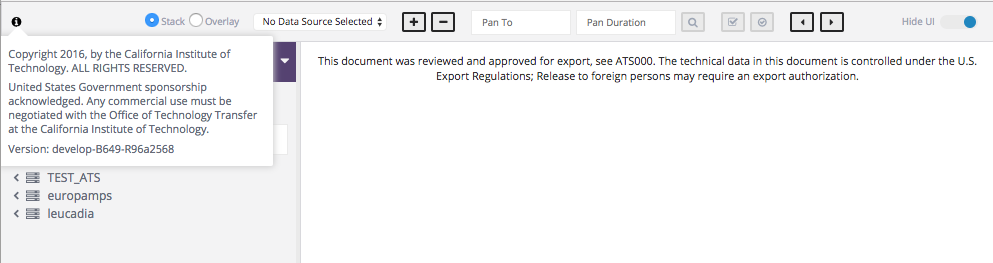


Figure 7. RAVEN Copyright Display

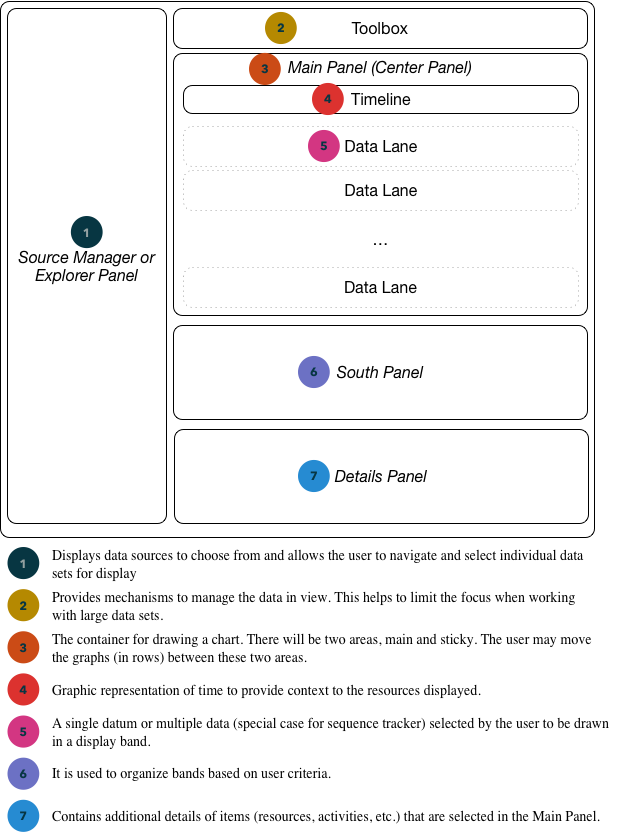


Figure 8. RAVEN UI map

Depending in the type of activities, the application has the capability to display slightly different context menus to support new features requests.

Requirements Addressed:

|  |  |
| --- | --- |
| ID | Requirement Statement |
| MW-10 | RAVEN shall provide a copyright statement that may be viewed by the user |
| MW-48 | RAVEN shall display an indicator that represents current time |
| MW-46 | RAVEN shall change the displayed time to show the format specified by the user. |

#### Communication Protocols

There are two types of communication that occur in the application depending on the component and the intention of the message:

**Communication to the server**. The communication happens over HTTP protocol, via a RESTful request of a resource in the server. RAVEN requests resources, or operations over resources, via an HTTP endpoint request that contains parameters in the query, path or payload. RAVEN uses GET, PUT. POST and DELETE verbs in the requests depending on the intent of the operation to perform. RAVEN is an expert at drawing charts, by design. Therefore, it knows nothing about data or how to process it. It expects the resources to be represented in such a way that they can be displayed as a resource, activity or state. The front-end relies on MPSServer to process and provide the necessary data.

**Communication over event.** The decoupling of the caller from the response allows for the JavaScript runtime to do other things while waiting for an asynchronous operation to complete and their callbacks to fire. JavaScript runtimes contain a message queue which stores a list of messages to be processed and their associated callback functions. These messages are queued in response to external events given a callback function has been provided. If, for example, a user were to click a button to select a data source and no callback function was provided- no message would have been added to the queue. In RAVEN, a request to an endpoint can be asynchronous, so when the response arrives, an event is fired for a method to handle the returned data. This happens throughout the application.

#### Error Communication Handling

For HTTP requests, there are different scenarios. These are the basic ones we catch:

* **200 Ok**.
* **202 Accepted**.
* **304 Not Modified**.
* **400 Bad Request**.
* **401 Unauthorized**.
* **403 Forbidden**.
* **404 Not Found**.
* **500 Internal Server Error**.
* **504 Gateway Timeout**.

Depending on the status code returned by the server, the client takes action by showing an alternative action or a user-friendly error in the screen via a toast alert or a dialog component. For 2xx and 3xx codes, we show the user in a toast alert that the action they have requested was successfully performed. This happens in case the UI doesn’t change enough or at all, so the user would know that something happened.

For 4xx errors, we capture them internally. The user is not interacting directly with the services, so we consider RAVEN to be the client in this context. The client has not requested resources in a right manner, so we trap the error and log it in the console log. If the actions are destructive we notify the user via a toast alert or a dialog box.

For 5xx errors, we know the server is experiencing issues that the client is unable to identify. We catch the error and notify the user of the application in a dialog box.

## Internal Program Set Interfaces

RAVEN uses the Flux architectural approach by implementing Redux. While there are no specific interfaces, it is important to describe its glossary, since the implementation depends on the correct understanding of its API.

**State**(also called the state tree). It is a broad term, but in the Redux API it usually refers to the single state value that is managed by the store and returned by getState(). It represents the entire state of a Redux application, which is often a deeply nested object.

**Action**. Plain object that represents an intention to change the state. Actions are the only way to get data into the store. Any data, whether from UI events, network callbacks, or other sources such as Web Sockets needs to eventually be dispatched as actions.

**Reducer** (also called a reducing function). It is a function that accepts an accumulation and a value and returns a new accumulation. They are used to reduce a collection of values down to a single value.

Reducers are not unique to Redux—they are a fundamental concept in functional programming. Even most non-functional languages, like JavaScript, have a built-in API for reducing. In JavaScript, it's Array.prototype.reduce().

In Redux, the accumulated value is the state object, and the values being accumulated are actions. Reducers calculate a new state given the previous state and an action. They must be pure functions that return the exact same output for given inputs. They should also be free of side-effects. This is what enables exciting features like hot reloading and time travel.

**Dispatching function** (or simply dispatch function). It is a function that accepts an action or an async action; it then may or may not dispatch one or more actions to the store.

It is important to distinguish between dispatching functions in general and the base dispatch function provided by the store instance without any middleware.

The base dispatch function always synchronously sends an action to the store's reducer, along with the previous state returned by the store, to calculate a new state. It expects actions to be plain objects ready to be consumed by the reducer.

An **action creator**is, quite simply, a function that creates an action. Do not confuse the two terms—again, an action is a payload of information, and an action creator is a factory that creates an action.

Calling an action creator only produces an action, but does not dispatch it. You need to call the store's dispatch function to actually cause the mutation. Sometimes it is referred to bound action creators to mean functions that call an action creator and immediately dispatch its result to a specific store instance. If an action creator needs to read the current state, perform an API call, or cause a side effect, like a routing transition, it should return an async action instead of an action.

An **async action**is a value that is sent to a dispatching function, but is not yet ready for consumption by the reducer. It will be transformed by middleware into an action (or a series of actions) before being sent to the base dispatch() function. Async actions may have different types, depending on the middleware you use. They are often asynchronous primitives, like a Promise or a thunk, which are not passed to the reducer immediately, but trigger action dispatches once an operation has completed.

***Middleware*.** It is a higher-order function that composes a dispatch function to return a new dispatch function. It often turns async actions into actions. Middleware is composable using function composition. It is useful for logging actions, performing side effects like routing, or turning an asynchronous API call into a series of synchronous actions.

***Store***. Object that holds the application's state tree. There should only be a single store in a Redux app, as the composition happens on the reducer level.

* dispatch(action) is the base dispatch function described above.
* getState() returns the current state of the store.
* subscribe(listener) registers a function to be called on state changes.
* replaceReducer(nextReducer) can be used to implement hot reloading and code splitting. Most likely you won't use it.

A ***store creator***is a function that creates a Redux store. Like with dispatching function, we must distinguish the base store creator, createStore(reducer, preloadedState) exported from the Redux package, from store creators that are returned from the store enhancers.

## Shared Data

One of RAVEN’s main purposes is to visualize and share this visualization

RAVEN is designed to provide the capability of exporting a timeline to different formats. In this way, data can be shared for different purposes. Another feature that supports sharing data with interface.

|  |  |
| --- | --- |
| ID | Requirement Statement |
| MW-63 | RAVEN shall provide an option to export a timeline to different formats. |

### Source Manager

The source manager exists to provide an interface that follows a friendly paradigm for visualizing and selecting sources grouped by categories. It allows the user to traverse the different nodes to arrive to the desire data set to select and visualize.

#### Source Manager Invocation and Parameters

RAVEN invokes the appropriate web services via the MPSServer API to obtain a list of data sources and a resource listing from each data source. All documents stored in a datastore can be queried using the generic ‘GET’ resources. The generic ‘GET’ endpoints support the ‘filter’, ‘project’ and ‘format’ parameters to specify the search criteria and fields (including nested records) to return and format the output. In addition, it also supports an ‘unwind’ parameter to allow unwinding of nested JSON documents to search data returned by MongoDB Web Services.

#### Source Manager Requirements

The Source Manager is developed in JavaScript using the MPSServer to access the data source via MPSServer. When the user selects to browse a data repository type, RAVEN invokes the appropriate web services to obtain a list of data source names and URLs for that data repository type. If the user selects ‘Browse Mongo’, RAVEN issues a GET request to MPSServer using the related endpoint. MPSServer returns a list of mongoDB instances known to the MPSServer. In addition to the instance names, the URL for each instance is included. When a user selects a Mongo instance, RAVEN will issue a GET request for the corresponding URL. Similarly, when a user selects a collection, RAVEN will issue a GET request for the collection URL. A list of displayable items with their URLs will be returned by MPSServer. Subsequently, when the user selects a resource, RAVEN will issue a GET request for the resource URL to obtain the data to be plotted.

|  |  |
| --- | --- |
| ID | Requirement Statement |
| MW-27 | RAVEN shall filter data sources based on user input |
| MW-01 | RAVEN shall retrieve data to be displayed from one or more user-selected databases. |

### Resource Data Handler

The Resource Data Handler retrieves resource data from MPSServer, formats the data for rendering by the CTL and populates the Details Panel with the resource data.

#### Resource Data Handler Invocation and Parameters

RAVEN retrieves data from selected data stores via MPSServer using REST calls. A CTL Interval object is created from each time-value pair resource. For a stack plot, a new Resource, State or Activity Band is created. For an overlaid plot, a Composite Band is created if it does not currently exist and the new Band is added to the Composite Band.

* **Resource Band**. Displays numeric data:

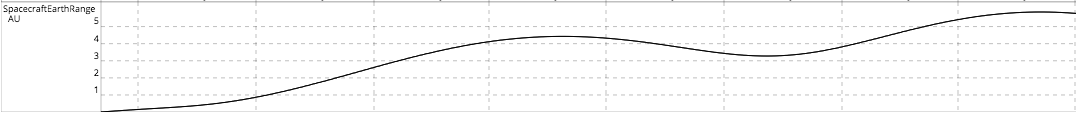
****

Figure 9. Resource Band

* **State Band**. Displays states:



Figure 10. State Band

* **Activity Band**. Displays activities in a row.

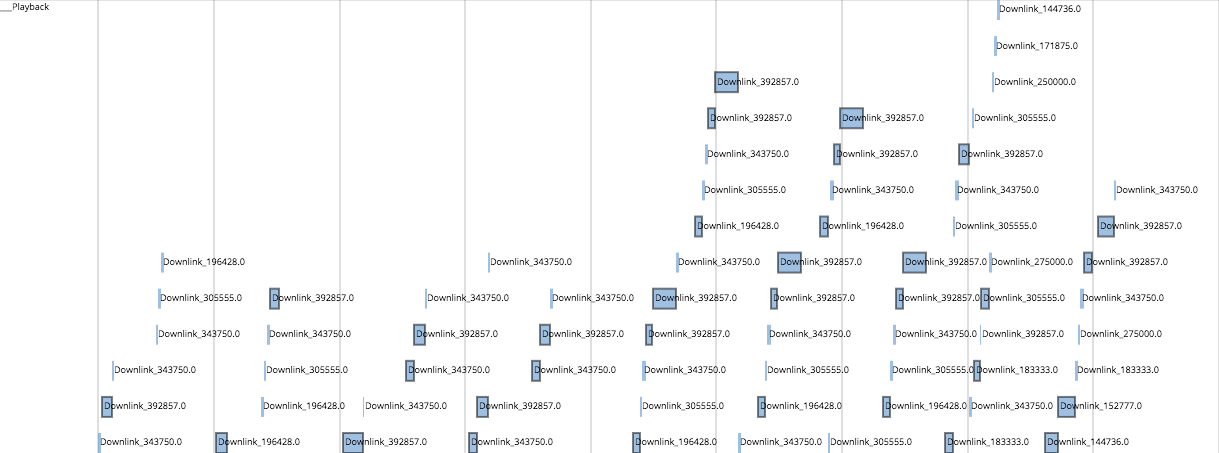


Figure 11. Activity Band

* **Composite Band**. Groups more than one band in a row:

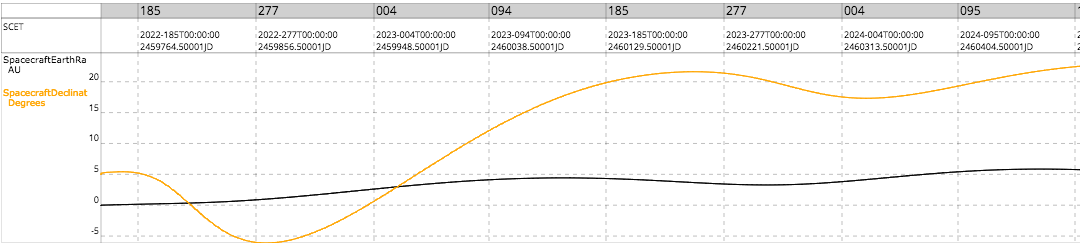


Figure 12. Composite Band

MPSViewer is a wrapper to the CTL library. It manages rows in the chart and transforms data referred by MPSServer to the data format needed for CTL.

Timed data set is added to the Details Panel to display resources in a tabular view. The Details Panel is implemented using the jqGrid plugin.

#### Resource Data Handler Requirements

|  |  |
| --- | --- |
| ID | Requirement Statement |
| MW-64 | RAVEN shall display activities, sequences, commands, and events (hereafter referred to collectively as "plan elements") with start times and end times on a timeline. |
| MW-65 | RAVEN shall plot sets of numerical values, each associated with a time. |
| MW-66 | RAVEN shall display sets of string values, each associated with a time. |
| MW-67 | RAVEN shall temporally align the information displayed in all rows |

### Data Renderer

The Data Renderer component deals with the drawing and plotting aspects of RAVEN via the CTL. This includes creating the chart, generating a row for each data type as well as overlaying one resource over another when this option is selected.

#### Data Renderer Invocation and Parameters

RAVEN uses the CTL for plotting the data resources. An instance of the MPSViewer, wrapping CTL, is initiated when the first data type is plotted. Each row in the chart is displayed by a Band object and then displayed in the center panel.

The MPSViewer component draws the chart by invoking the paint function on each band and then it listens for zoom and pan actions. A callback obtains new data from the server if necessary. When new data is received, the Band Intervals are updated and the Bands repaint functions are invoked. Note that it is only necessary to obtain new data from the server for decimated numeric data, where decimation is the filtering output of data.

#### Data Renderer Requirements

The Data Renderer is developed in JavaScript with React using the CTL to handle activity, command and resource plotting. Resources can be numeric and state. Numeric resources are plotted using line graphs and state resources are plotted using bar graphs with the state label.

### Action Handler

The Action Handler component is called whenever changes are requested to the rows and chart. This includes actions such as obtaining new display start and end times, retrieving data for a new time range and updating the plotting for a new time range.

#### Action Handler Invocation and Parameters

RAVEN uses the CTL for re-plotting and re-drawing of the data resources triggered by requests for changes to the timeline. Actions available for each timeline type are accessible via context menu. Context menu in RAVEN is implemented using jQuery contextMenu API. Available actions for a selected timeline depends on the type of timeline and displays the appropriate context menu. When a user right clicks on a timeline, RAVEN determines if it is an activity, state or numeric timeline. A context menu is dynamically built for the timeline. Callback functions are invoked to handle each of the user actions or selections.

#### Action Handler Requirements

The Action Handler is developed in JavaScript with Redux using the CTL to re-plot and/or re-draw data resources and MPSServer to retrieve new data.

### Display Manager

The display Manager creates the details panel for data sets. RAVEN allows a screen layout to be persisted in a document-based database. MongoDB is currently used to store screen layouts, nevertheless, any document-based database can be used. The database name for the screen layout is configured using the layout\_db property defined in “config.js”. The user is prompted for the collection name used to store the screen layout.

RAVEN can display data from any data source providing listing and data request service endpoints. RAVEN currently supports (but not limited to) displaying data from MongoDB and TMS via the MPSServer RESTful interface.

A configuration file should be adapted to set the required display properties in RAVEN. The RAVEN screen consists of four sections:

* **Top Panel**. Contains buttons to browse various data sources.
* **Left Panel**. It is the plot resource selection grouped by the data source. Data sources can be added and deleted.
* **Bottom Panel**. Details panel to display the time and value of the data point for the selected resource.
* **Middle Panel**. Resource drawing area. The drawing area of RAVEN contains one or more rows with a single time axis (X-axis). Each row can contain one of more activity, state or numeric plots, referred to an overlay plot.

#### Display Manager Invocation and Parameters

The Display Manager is invoked when the user saves a current layout or to recall a previously saved layout template. Each layout template is identified using a unique name provided by the user.

#### Display Manager Requirements

The Display Manager obtains the display template from MPSViewer for the current display layout by invoking the getCurrentLayout function. The display templates are saved in mongoDB via the PUT request. A display template can be applied to one or more selected collections to reproduce a previously created display.

The Display Manager can also generate a unique URL for the currently displayed data and layout. Using the unique URL, an exact display can be created without specifying a data source collection.

An ITAR message can also be configured to be displayed in the Top Panel to indicate that the visualized data has been approved.

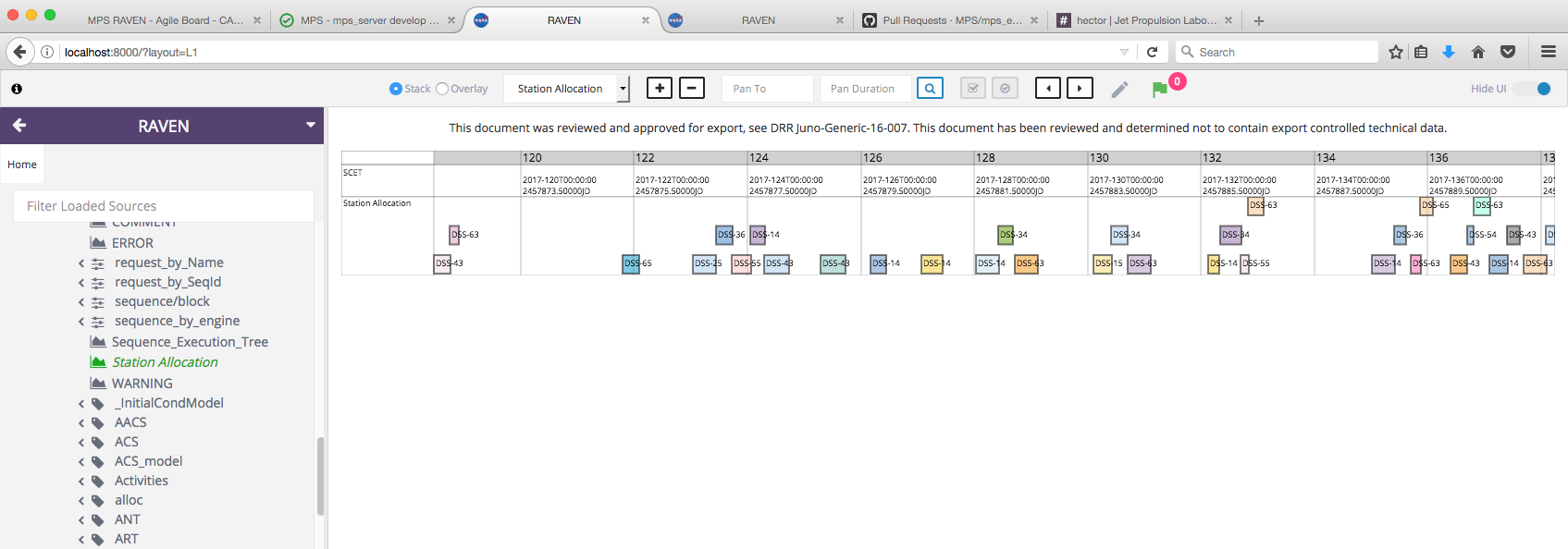


Figure 13. ITAR message

|  |  |
| --- | --- |
| ID | Requirement Statement |
| MW-34 | RAVEN shall manage display configurations. |
| MW-60 | RAVEN shall display an ITAR Marking Language |
| MW-53 | RAVEN shall configure the structure of data in a user-selected database. |

## Error Detection and Recovery

Please refer to section 4.2.1.3.3 for error handling information.

# Analysis

RAVEN is developed as a web-based application that allows users to view the data without the need of having SEQ components installed in their local hardware. Moreover, users search for platform independent applications that are well performant and can be shared among peers to collaborate and contribute for analyzing data.

## Trade-offs and Decisions

Table 6: Trade-Offs and Decisions matrix

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **Desktop Applications** | **Web Applications** | **Stakeholder Interest** |
| **Rapid Development** | Designed from the beginning to be a quick and easy solution to building graphical user interfaces (GUIs) | Never designed for rapid development. Using MVC (model, view, control) style is typically seen as the “correct” way to create web applications. | MGSS, Developers - Maintainability |
| **Security** | Since the user keeps the data on their own computer systems this makes it harder for hackers to gain access to people’s data. Nevertheless, the way data is requested is by service calls, so it can potentially be intercepted anyway. | Since all the user’s communication with data happens online, we need to be cognizant of clean security practices in development to avoid holes in the application. | Flight Projects - Security |
| **Flexibility** | It is very easy to write desktop applications that take advantage of the user’s hardware. Nevertheless, the flexibility exists if you have to interact with a hardware interface. In our case, we do not have that requirement. | Web application rely on HTML, JavaScript and CSS to provide a friendly user interface. Now, for our requirements in RAVEN, we don’t need to interface to any peripherals in the user’s hardware. | Flight Projects, QA – Usability |
| **Portability** | Desktop applications can be portable, but most are NOT portable and require manual installation from the user. | Web applications are very portable and will work with almost any computer with a modern web browser. | MGSS - Portability |
| **Maintenance** | Desktop applications usually need to be manually updated (or at least have manual approval) to install updates. | Very maintainable (assuming that the code and its structure are written with strong design patterns). End-users do not have to install any updates. All the updates are already taken care of by the web application administrators. | MGSS, Developers - Maintainability |
| **Performance** | Typically, you will find that well-written desktop software running on a decent computer run faster than web applications. | Web applications typically have slower performance than desktop applications, due to having to transmit data across the web. | Flight Projects – Performance |

Does not address the concern

Somewhat addresses the concern

Addresses the concern

Based on the categories analyzed, a web application fulfills the stakeholder’s concerns better than a desktop application. Flexibility and portability are high in priority.

While a desktop application can be better performant than a web application, it is not definite. Good architecture and design patterns implementation help to speed up the performance perception. Another technique used that can be presented as an example, is lazy loading. Not all data/features need to be loaded at the beginning of the application, but can be requested when in need to be used.

In regards to security, it is something that must be taken very seriously. While design and implementation should have security in mind, relying in other tools for authentication and authorization, such as CAM, is beneficial. Requirements for data sharing and export controls are also taken into consideration.

To conclude, flexibility, portability and maintenance build a stronger case for implementing a web application, considering that there are things that can be done in the implementation to address security and performance concerns.

After concluding that a web application needs to be implemented, the design must tell what architecture the application must follow. The benefits of using a virtual DOM for processing large amount of data providing a smooth and friendly experience to the user geared the application development towards React JS as a framework. The team looked at two possible architecture choices:

Table 7: Flux Vs Redux

|  |  |  |
| --- | --- | --- |
|  | Flux | Redux |
| Complexity | Uses Callback registration. It means we must chain requests to get data and then manage aggregation. | Uses Functional Composition, allowing us to map the results of one function to the arguments of another |
| Server Rendering | Stores are singletons. This means it's hard to separate the data for different requests on the server. | **Since there is just a single store (managed by many reducers), you don't need any special API to manage**  data propagation to registered stores. |
| Ecosystem | Not very extensible by design. | provides a few extension points such as middleware. It was designed with use cases such as logging, support for promises, observables, routing, immutability dev checks and persistence. |

Overall, Redux preserves all the benefits of Flux (recording and replaying of actions, unidirectional data flow, dependent mutations) and adds new benefits (easy undo-redo, hot reloading) without introducing Dispatcher and store registration.

## Imported Software

The Node Package Manager (NPM) is used to handle dependencies in RAVEN. The definition lives in a package.json file in the root of this project’s codebase. Besides containing metadata for the project, it describes the dependencies and the scripts available to run RAVEN.

Main Dependencies:

* **"express"**: **"^4.14.0".** Basic Routing library for NodeJS.
* **“typescript”**: **“^2.2.1”.** JavaScript superset which provides optional static typing, classes and interfaces.
* **"babel"**: **"^6.5.2".** Compiles ES6 code into ES5 for browsers compatibility.
* **"eslint"**: **"^3.11.1".** Detect JavaScript code problems and errors.
* **"react"**: **"^15.4.1".** ReactJS library.
* **"redux"**: **"^3.6.0".** Flux architecture implementation for state store.
* **“webpack”**: **“^1.13.3”.** Module bundler.

For the dependencies in the project, semver[[5]](#footnote-5) is used. It is a specification outlining a method of encoding the nature of change between releases of a public interface, directly into the version string.

* **version**Must match version exactly
* **>version**Must be greater than version
* **>=version**etc
* **<version**
* **<=version**
* **~version**"Approximately equivalent to version"
* **^version**"Compatible with version"
* 1.2.x 1.2.0, 1.2.1, etc., but not 1.3.0
* Matches any version
* **""** (just an empty string) Same as \*
* version1 - version2 Same as >=version1 <=version2.
* range1 || range2 Passes if either range1 or range2 are satisfied.
* user/repo
* tag A specific version tagged and published as tag
* path/path/path

## Technology Profile

Table 8: Technology Profile

| **Technology Area** | **Technology Description** | **Technology Standards** | **Rationale** | **License** |
| --- | --- | --- | --- | --- |
| ReactJS | Open-source **JavaScript** library providing a view for data rendered as HTML. React’s primary goal is to provide a different and more efficient way of performing DOM updates. Instead of mutating the DOM directly, your app builds a “virtual DOM”, and React handles updating the real DOM to match. | Virtual DOM | React listens from the components what they want to render – then React will efficiently update and render just the right components when the state changes. This helps us to achieve high performance since we are dealing with millions of points to plot. | MIT License (MIT) |
| Babel | Babel is a Javasctipr transpiler. It takes ES2015 (ES6) code and makes it work on all browsers, even the ones that might not support it. Basically, it compiles it into ES5 Code. | ECMAScript 6 | For development purposes, it makes it more convenient to use ES6 features. Unfortunately, not all broswsers will support it, so we need to compile it back to ES5 to provide older browsers compatibility. | MIT License (MIT) |
| Canvas Time Library (CTL) | The HTML5 <canvas> tag is used to draw graphics, on the fly, via scripting. | Javascript | We need a library that is flexible and efficient to draw graphs to represent the data provided to RAVEN. | CalTech |
| Node Packaged Modules (NPM) | Online repository for the publishing of open-source Node.js projects; AS well as, command-line utility for interacting with said repository that aids in package installation, version management, and dependency management. | NodeJS (Javascript) | It helps to organize and manage our dependencies in an efficient way. | The Artistic License 2.0 |
| Typescript | Superset of JavaScript which primarily provides optional static typing, classes and interfaces. One of the big benefits is to enable IDEs to provide a richer environment for spotting common errors as you type the code. | Javascript superset (maintained by Microsoft) | Provides massive producitity boosts because of its cleaner ECMAScriupt 6 syntax and optional typing. | The Artistic License 2.0 |
| Webpack | Module bundler that works great with the most modern of front-end workflows including Babel, ReactJS, CommonJS, among others. | NodeJS (Javascript) | When building complex front end applications with many non-code static assets such as CSS, images and fonts, it provides great benefits by creating a dependency graph and packaging the application in a portable and efficient manner. | MIT License (MIT) |
| Polymer | Polymer is a lightweight library that helps you take full advantage of Web Components. You can create reusable custom elements that interoperate seamlessly with the browser’s built-in elements, or break your app up into right-sized components, making your code cleaner and less expensive to maintain. | HTML, CSS, Javascript | Polymer helps us leveraging the browser’s native technologies to create portable and re-usable components. Reduces the gap between developer and designer and makes our code less expensive to maintain. | BSD License |
| Cascading Style Sheets (CSS) Level 2 | Used to define styles for web pages, including the design, layout and variations in display for different devices and screen sizes. | CSS | Helps us styling the application by separating content of an HTML document from the style and layout. | Free License |

## Design Feasibility, Performance and Margins

RAVEN is designed to be loosely coupled with the data provider, by interacting with a services abstraction layer in MPSServer. RAVEN is only an expert in drawing charts and it does not care where the data is coming from or any rules associated with it.

When we create components for RAVEN in React JS we follow a Composite Design Pattern, where every component is defined with inputs and outputs but have no dependencies in other components. In addition, it does not keep state since everything goes to a data store internally, where state is propagated by the components.



Figure 14. Composite Design Pattern

By definition, it allows us to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

In order to fulfill the performance requirements, the development team needs to take into consideration two aspects: The time the response comes back from the server after requesting data and the time it will take to render the data in a band. The design takes into consideration that no data processing should occur in the front-end, in order to display data as quick as it comes back from the server.

|  |  |
| --- | --- |
| ID | Requirement Statement |
| MW-06 | RAVEN shall display 10,000 points for a single resource within 20 seconds |
| MW-05 | RAVEN shall display up to 100 rows of data simultaneously in the same chart when requested by the user |

## Reliability, Maintainability, and Related Concerns

Any software application must be reliable and maintainable. Nevertheless, defining discrete values for “how much” an application is reliable or “how easy” is to maintain it is difficult to define.

The team follows good design and implementation practices that computer scientists have studied and identified as a way to provide reliability and maintainability in software. Stakeholders concerns are usually around these features in software.

The architecture followed in the implementation revolves around two concepts:

* Separation of Concerns
* Reusability of Components

By separating a computer program into distinct sections, such that each section addresses a separate concern, the team simplifies development and maintenance of the application. This happens due to the fact that changes in once component, would not affect other components. The team follows the motto that “each component is an expert in only one thing”. Each component must be reliable in its task.

The idea behind reusing component is that having an expert in one and only one thing can be leveraged in a composition pattern. By defining the blocks, the application can compose functionality. If a component is reliable to draw states, it is always reliable that drawing states will be achieved.

RAVEN has been moving towards a loosely coupled approach, where it is an expert in displaying data. It does not care about rules, states, data sources or business logic. This is important because a stakeholder’s concern is that RAVEN is interoperable, communicating and exchanging data across a range of applications.

## Adaptability and Reusability

RAVEN reuses components, having less code to maintain. Furthermore, when there is a need to add components, the same structure is maintained, avoiding the need to change other parts of the system. When bugs are found, the fix happens within the component.

In terms of configuration, this happens during installation of the application. The RAVEN Viewer is configurable to fit the needs of each individual user by modifying a configuration file.

Structure:

* Available buttons
* Location of data sources

Layout and Content:

* Screen layout with or without data can be saved and retrieved later

Having a clear separation of concerns between data layer, services and presentation allows RAVEN to be flexible to adapt to different mission’s users.

# Detailed Design

The detailed design of RAVEN can be found as comments in the code itself, where it expresses the purpose behind the different parts and how the components interact with each other. Nevertheless, it is important to provide the context of the design in this section so that connecting the dots when looking at the code is easier.

The RAVEN project contains, mainly, three abstract layers:

* **Server**. Contains the controller where the HTTP requests are sent and the responses are handled.
* **Client**. Lives inside the src folder. And it is divided into content, pages and scripts. The content refers to all resources like CSS, fonts and images to be used. The pages define the starting point of the application. Finally, the scripts folder contains the React specific JavaScript and JSX files where the Redux architecture is implemented.
* **Configuration**. In the main level of the folder structure, things like the dependencies management, build scripts and README files are defined.

Having defined the logical parts of the application, the redux architecture can be described below:

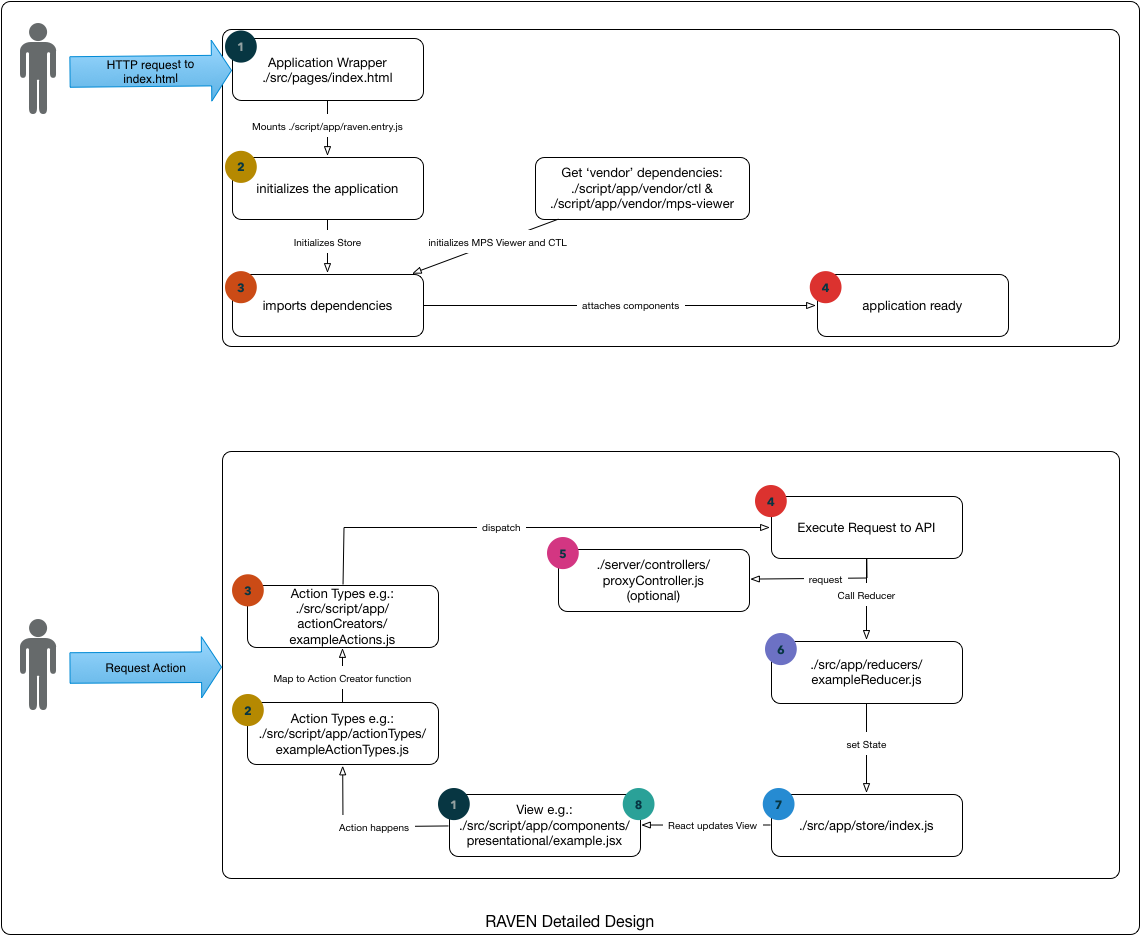


Figure 15. RAVEN Detailed Design with Redux Architecture

Every action from the user follows the same flow in the Redux architecture implementation. An action is initiated from a view in a component, it determines the action type and calls an action creator. As a consequence, it dispatches an action and requests an endpoint (if required). Then, it updates the state in the store and React does a comparison to update only the part in the view that is needed.

1. *Design Patterns: Elements of Reusable Object-Oriented Software is a*[*software engineering*](https://en.wikipedia.org/wiki/Software_engineering)*book describing*[*software design patterns*](https://en.wikipedia.org/wiki/Software_design_pattern)*. The book's authors are*[*Erich Gamma*](https://en.wikipedia.org/wiki/Erich_Gamma)*, Richard Helm,*[*Ralph Johnson*](https://en.wikipedia.org/wiki/Ralph_Johnson_(computer_scientist))*and*[*John Vlissides*](https://en.wikipedia.org/wiki/John_Vlissides)*with a foreword by*[*Grady Booch*](https://en.wikipedia.org/wiki/Grady_Booch)*, known as the Gang of Four.* [↑](#footnote-ref-1)
2. Please refer to section 5.1 for the trade-offs and decisions analysis. [↑](#footnote-ref-2)
3. Please refer to section 5.3 for the technology stack selected. [↑](#footnote-ref-3)
4. Please refer to section 5.1 for trade-offs and decisions [↑](#footnote-ref-4)
5. Semver stands for semantic versioning. Given a version number MAJOR.MINOR.PATCH, increment the:

   MAJOR version when you make incompatible API changes,

   MINOR version when you add functionality in a backwards-compatible manner, and

   PATCH version when you make backwards-compatible bug fixes. [↑](#footnote-ref-5)