Open MCT Web Developer Guide

Victor Woeltjen
victor.woeltjen@nasa.gov

June 4, 2015
Document Version 1.0

Date	Version	Summary of Changes	Author
April 29, 2015	0	Initial Draft	Victor Woeltjen
May 12, 2015	0.1		Victor Woeltjen
June 4, 2015	1.0	Name changes	Victor Woeltjen

Introduction

The purpose of this guide is to familiarize software developers with the Open MCT Web platform.

What is Open MCT Web?

Open MCT Web is a platform for building user interface and display tools, developed at the NASA Ames Research Center in collaboration with teams at the Jet Propulsion Laboratory. It is written in HTML5, CSS3, and JavaScript, using AngularJS (http://www.angularjs.org) as a framework. Its intended use is to create single-page web applications which integrate data and behavior from a variety of sources and domains.

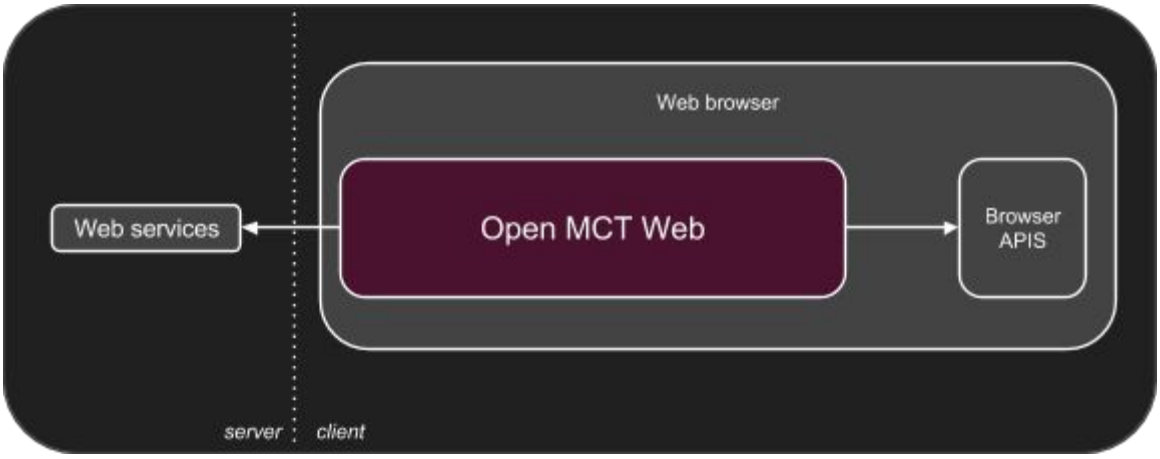
Open MCT Web has been developed to support the remote operation of space vehicles, so some of its features are specific to that task; however, it is flexible enough to be adapted to a variety of other application domains where a display tool oriented toward browsing, composing, and visualizing would be useful.

Open MCT Web provides:

- A common user interface paradigm which can be applied to a variety of domains and tasks. Open MCT Web is more than a widget toolkit - it provides a standard tree-on-the-left, view-on-the-right browsing environment which you customize by adding new browsable object types, visualizations, and back-end adapters.
- A plugin framework and an extensible API for introducing new application features of a variety of types.
- A set of general-purpose object types and visualizations, as well as some visualizations and infrastructure specific to telemetry display.

Client-Server Relationship

Open MCT Web is client software - it runs entirely in the user's web browser. As such, it is largely "server agnostic"; any web server capable of serving files from paths is capable of providing Open MCT Web.



While Open MCT Web can be configured to run as a standalone client, this is rarely very useful. Instead, it is intended to be used as a display and interaction layer for information obtained from a variety of back-end services. Doing so requires authoring or utilizing adapter plugins which allow Open MCT Web to interact with these services.

Typically, the pattern here is to provide a known interface that Open MCT Web can utilize, and implement it such that it interacts with whatever back-end provides the relevant information. Examples of back-ends that can be utilized in this fashion include databases for the persistence of user-created objects, or sources of telemetry data.

Developing with Open MCT Web

Building applications with Open MCT Web typically means authoring and utilizing a set of plugins which provide application-specific details about how Open MCT Web should behave.

Technologies

Open MCT Web sources are written in JavaScript, with a number of configuration files written in JSON. Displayable components are written in HTML5 and CSS3.

Open MCT Web is built using AngularJS (http://www.angularjs.org) from Google. A good understanding of Angular is recommended for developers working with Open MCT Web.

Forking

Open MCT Web does not currently have a single stand-alone artifact that can be used as a library. Instead, the recommended approach for creating a new application is to start by forking/branching Open MCT Web, and then adding new features from there. Put another way, Open MCT Web's source structure is built to serve as a template for specific applications.

Forking in this manner should *not* require that you edit Open MCT Web's sources. The preferred approach is to create a new directory (peer to `index.html`) for the new application, then add new bundles (as described in the Framework chapter) within that directory.

To initially clone the Open MCT Web repository:

```
git clone <repository URL> <local repo directory> -b
open-master
```

To create a fork to begin working on a new application using Open MCT Web:

```
cd <local repo directory>
git checkout open-master
git checkout -b <new branch name>
```

As a convention used internally, applications built using Open MCT Web have master branch names with an identifying prefix. For instance, if building an application called "Foo", the last statement above would look like:

```
git checkout -b foo-master
```

This convention is not enforced or understood by Open MCT Web in any way; it is mentioned here as a more general recommendation.

Overview

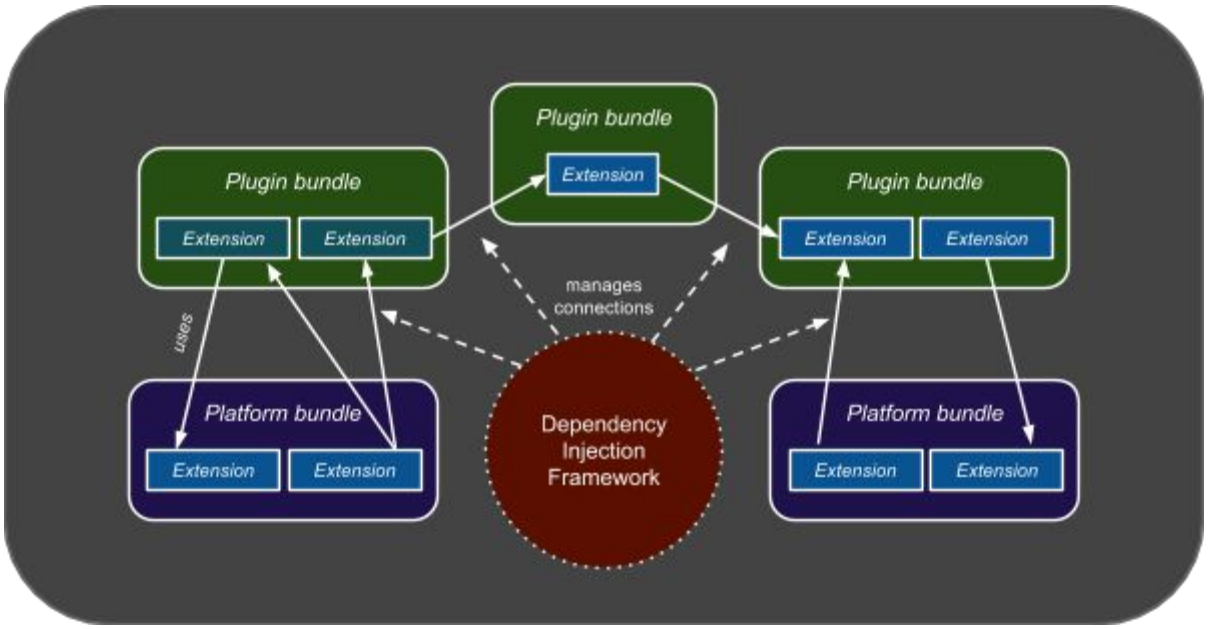
Open MCT Web is implemented as a framework component which manages a set of other components. These components, called “bundles”, act as containers to group sets of related functionality; individual units of functionality are expressed within these bundles as “extensions.”

Extensions declare dependencies on other extensions (either individually or categorically), and the framework provides actual extension instances at run-time to satisfy these declared dependency. This dependency injection approach allows software components which have been authored separately (e.g. as plugins) but to collaborate at run-time.

Open MCT Web’s framework layer is implemented on top of AngularJS’s dependency injection mechanism (https://docs.angularjs.org/guide/di) and is modelled after OSGi (http://www.osgi.org/) and its Declarative Services component model (http://wiki.osgi.org/wiki/Declarative_Services). In particular, this is where the term “bundle” comes from.

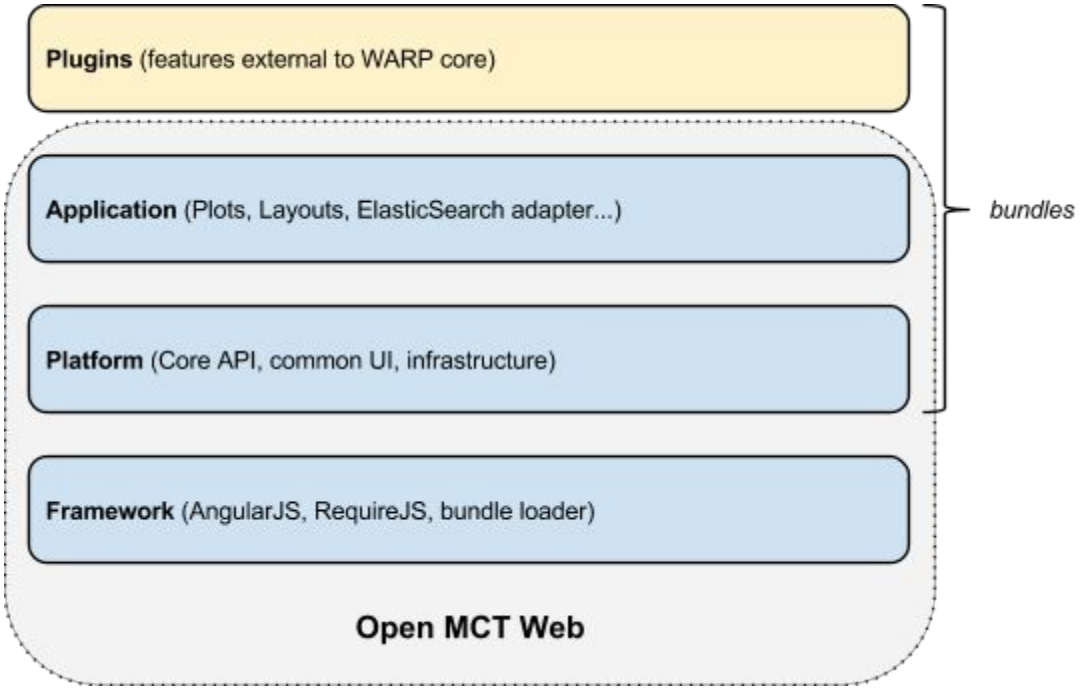
Framework Overview

The framework’s role in the application is to manage connections between bundles. All application-specific behavior is provided by individual bundles, or as the result of their collaboration.



Tiers

While all bundles in a running Open MCT Web instance are effectively peers, it is useful to think of them as a tiered architecture, where each tier adds more specificity to the application.



- **Framework**: This tier is responsible for wiring together the set of configured components (called bundles) together to instantiate the running application. It is responsible for mediating between AngularJS (in particular, its dependency injection mechanism) and RequireJS (to load scripts at run-time.) It additionally interprets bundle definitions (see explanation below, as well as further detail in the Framework chapter.) At this tier, we are at our most general: We know only that we are a plugin-based application.
- **Platform:** Components in the Platform tier describe both the general user interface and corresponding developer-facing interfaces of Open MCT Web. This tier provides the general infrastructure for applications. It is less general than the framework tier, insofar as this tier introduces a specific user interface paradigm, but it is still non-specific as to what useful features will be provided. Although they can be removed or replaced easily, bundles provided by the Platform tier generally should not be thought of as optional.
- **Application:** The application tier consists of components which utilize the infrastructure provided by the Platform to provide functionality which will (or could) be useful to specific applications built using Open MCT Web. These include adapters to specific persistence back-ends (such as ElasticSearch or CouchDB) as well as bundles which describe more user-facing features (such as Plot views for visualizing time series data, or Layout objects for display-building.) Bundles from this tier can be added or removed without

compromising basic application functionality, with the caveat that at least one persistence adapter needs to be present.

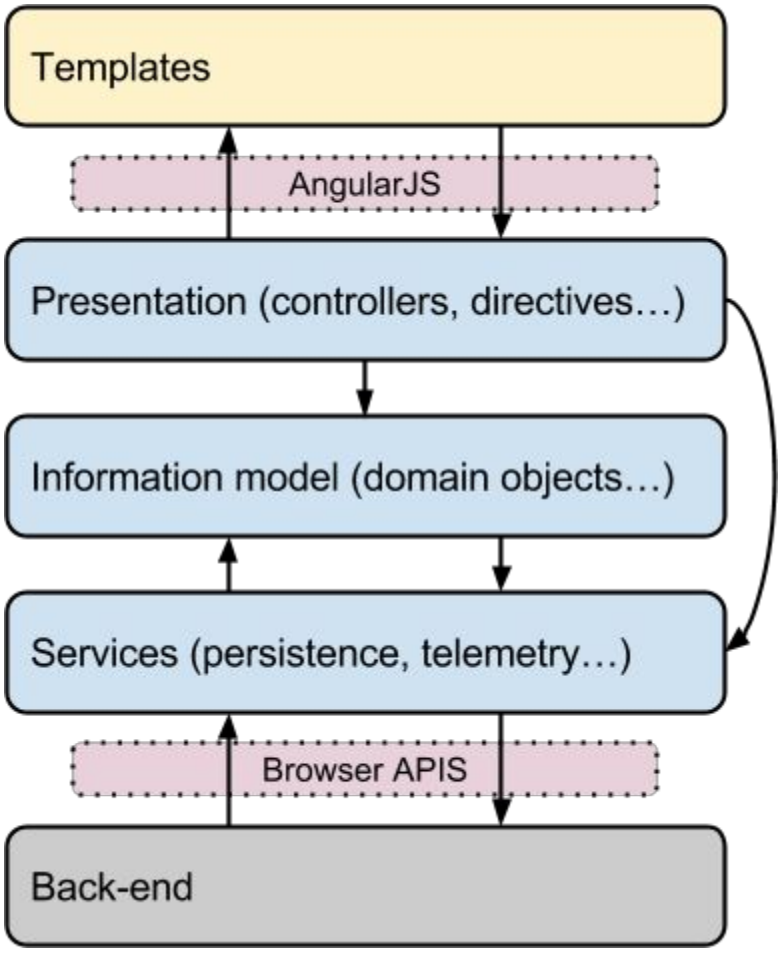
- **Plugins:** Conceptually, this tier is not so different from the application tier; it consists of bundles describing new features, back-end adapters, that are specific to the application being built on Open MCT Web. It is described as a separate tier here because it has one important distinction from the application tier: It consists of bundles that are not included with the platform (either authored anew for the specific application, or obtained from elsewhere.)

Note that bundles in any tier can go off and consult back-end services. In practice, this responsibility is handled at the Application and/or Plugin tiers; Open MCT Web is built to be server-agnostic, so any back-end is considered an application-specific detail.

Platform Overview

The “tiered” architecture described in the preceding text describes a way of thinking of and categorizing software components of a Open MCT Web application, as well as the framework layer’s role in mediating between these components. Once the framework layer has wired these software components together, however, the application’s logical architecture emerges.

Logical Architecture

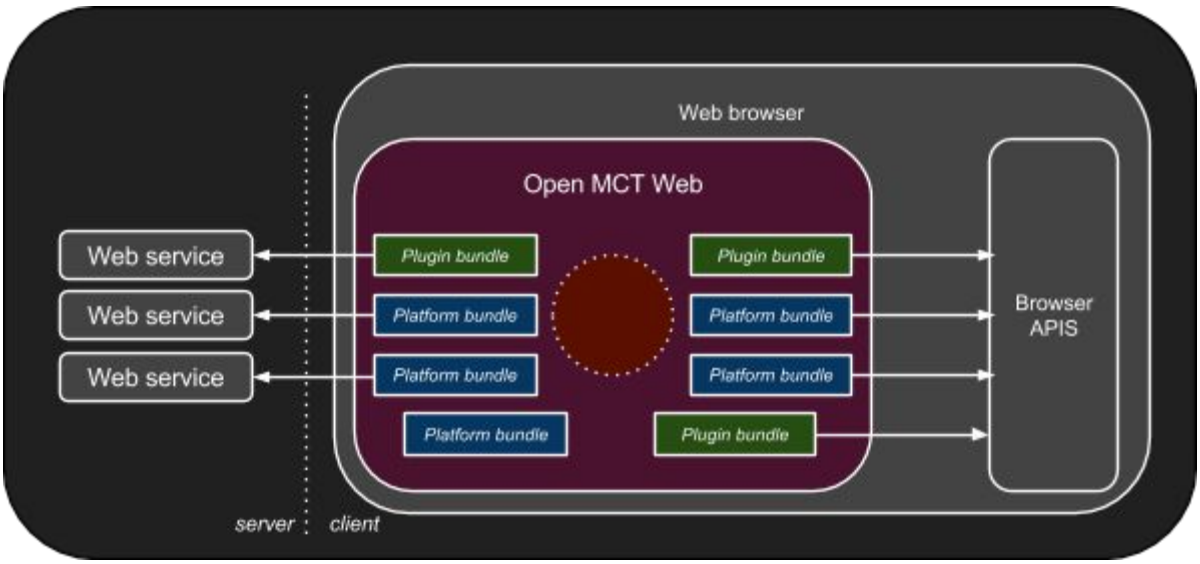


- *Templates*: HTML templates written in Angular's template syntax; see https://docs.angularjs.org/guide/templates. These describe the page as actually seen by the user. Conceptually, stylesheets (controlling the look-and-feel of the rendered templates) belong in this grouping as well.
- *Presentation:* Responsible for providing information to be displayed in templates, and managing interactions with the information model. Provides the logic and behavior of the user interface itself.
- *Information model:* Provides a common (within Open MCT Web) set of interfaces for dealing with "things" - domain objects - within the system. User-facing concerns in a Open MCT Web application are expressed as domain objects; examples include folders (used to organize other domain objects), layouts (used to build displays), or telemetry points (used as handles for streams of remote measurements.) These domain objects expose a common set of interfaces to allow reusable user interfaces to be built in the presentation and template tiers; the specifics of these behaviors are then mapped to interactions with underlying services.
- *Services:* A set of interfaces for dealing with back-end services.

- *Back-end*: External to the Open MCT Web client; the underlying persistence stores, telemetry streams, and so forth which the Open MCT Web client is being used to interact with.

Web Services

As mentioned in the Introduction, Open MCT Web is a platform single-page applications which runs entirely in the browser. Most applications will want to additionally interact with server-side resources, to (for example) read telemetry data or store user-created objects. This interaction is handled by individual bundles using APIs which are supported in browser (such as `XMLHttpRequest`, typically wrapped by Angular's `$http`.)



This architectural approach ensures a loose coupling between applications built using Open MCT Web and the backends which support them.

Glossary

Certain terms are used throughout Open MCT Web with consistent meanings or conventions. Other developer documentation, particularly in-line documentation, may presume an understanding of these terms.

- *bundle*: A bundle is a removable, reusable grouping of software elements. The application is composed of bundles. Plug-ins are bundles.
- *capability*: A JavaScript object which exposes dynamic behavior or non-persistent state associated with a domain object.
- *category*: A machine-readable identifier for a group that something may belong to.

- *composition*: In the context of a domain object, this refers to the set of other domain objects that compose or are contained by that object. A domain object's composition is the set of domain objects that should appear immediately beneath it in a tree hierarchy. A domain object's composition is described in its model as an array of identifiers; its composition capability provides a means to retrieve the actual domain object instances associated with these identifiers asynchronously.
- *description*: When used as an object property, this refers to the human-readable description of a thing; usually a single sentence or short paragraph. (Most often used in the context of extensions, domain object models, or other similar application-specific objects.)
- *domain object*: A meaningful object to the user; a distinct thing in the work support by Open MCT Web. Anything that appears in the left-hand tree is a domain object.
- *extension*: An extension is a unit of functionality exposed to the platform in a declarative fashion by a bundle. The term “extension category” is used to distinguish types of extensions from specific extension instances.
- *id*: A string which uniquely identifies a domain object.
- *key*: When used as an object property, this refers to the machine-readable identifier for a specific thing in a set of things. (Most often used in the context of extensions or other similar application-specific object sets.) This term is chosen to avoid attaching ambiguous meanings to “id”.
- *model*: The persistent state associated with a domain object. A domain object's model is a JavaScript object which can be converted to JSON without losing information (that is, it contains no methods.)
- *name*: When used as an object property, this refers to the human-readable name for a thing. (Most often used in the context of extensions, domain object models, or other similar application-specific objects.)
- *navigation*: Refers to the current state of the application with respect to the user's expressed interest in a specific domain object; e.g. when a user clicks on a domain object in the tree, they are *navigating* to it, and it is thereafter considered the *navigated* object (until the user makes another such choice.) This term is used to distinguish navigation from selection, which occurs in an editing context.
- *space*: A machine-readable name used to identify a persistence store. Interactions with persistence with generally involve a space parameter in some form, to distinguish multiple persistence stores from one another (for cases where there are multiple valid persistence locations available.)
- *source*: A machine-readable name used to identify a source of telemetry data. Similar to “space”, this allows multiple telemetry sources to operate side-by-side without conflicting.

Framework

Open MCT Web is built on the AngularJS framework (http://www.angularjs.org). A good understanding of that framework is recommended.

Open MCT Web adds an extra layer on top of AngularJS to (a) generalize its dependency injection mechanism slightly, particularly to handle many-to-one relationships; and (b) handle script loading. Combined, these features become a plugin mechanism.

This framework layer operates on two key concepts:

- **Bundle.** A bundle is a collection of related functionality that can be added to the application as a group. More concretely, a bundle is a directory containing a JSON file declaring its contents, as well as JavaScript sources, HTML templates, and other resources used to support that functionality. (The term bundle is borrowed from OSGi - http://www.osgi.org/ - which has also inspired many of the concepts used in the framework layer. A familiarity with OSGi, particularly Declarative Services, may be useful when working with Open MCT Web.)
- **Extension.** An extension is an individual unit of functionality. Extensions are collected together in bundles, and may interact with other extensions.

The framework layer, loaded and initiated from `index.html`, is the main point of entry for an application built on Open MCT Web. It is responsible for wiring together the application at run time (much of this responsibility is actually delegated to Angular); at a high-level, the framework does this by proceeding through four stages:

1. **Loading definitions.** JSON declarations are loaded for all bundles which will constitute the application, and wrapped in a useful API for subsequent stages.
2. **Resolving extensions.** Any scripts which provide implementations for extensions exposed by bundles are loaded, using Require.
3. **Registering extensions.** Resolved extensions are registered with Angular, such that they can be used by the application at run-time. This stage includes both registration of Angular built-ins (directives, controllers, routes, constants, and services) as well as registration of non-Angular extensions.
4. **Bootstrapping.** The Angular application is bootstrapped; at that point, Angular takes over and populates the body of the page using the extensions that have been registered.

Bundles

The basic configurable unit of Open MCT Web is the bundle. This term has been used a bit already; now we'll get to a more formal definition.

A bundle is a directory which contains:

- A bundle definition; a file named `bundle.json`.
- Subdirectories for sources, resources, and tests.
- Optionally, a `README.md` Markdown file describing its contents (this is not used by Open MCT Web in any way, but it's a helpful convention to follow.)

The bundle definition is the main point of entry for the bundle. The framework looks at this to determine which components need to be loaded and how they interact.

A plugin in Open MCT Web is a bundle. The platform itself is also decomposed into bundles, each of which provides some category of functionality. The difference between a "bundle" and a "plugin" is purely a matter of the intended use; a plugin is just a bundle that is meant to be easily added or removed. When developing, it is typically more useful to think in terms of bundles.

Configuring Active Bundles

To decide *which* bundles should be loaded, the framework loads a file named `bundles.json` (peer to the index.html file which serves the application) to determine which bundles should be loaded. This file should contain a single JSON array of strings, where each is the path to a bundle. These paths should not include `bundle.json` (this is implicit) or a trailing slash.

For instance, if bundles.json contained:

```
[
      "example/builtins",
      "example/extensions"
]
```

...then the Open MCT Web framework would look for bundle definitions at `example/builtins/bundle.json` and `example/extensions/bundle.json`, relative to the path of `index.html`. No other bundles would be loaded.

Bundle Definition

A bundle definition (the `bundle.json` file located within a bundle) contains a description of the bundle itself, as well as the information exposed by the bundle.

This definition is expressed as a single JSON object with the following properties (all of which are optional, falling back to reasonable defaults):

- `key`: A machine-readable name for the bundle. (Currently used only in logging.)
- `name`: A human-readable name for the bundle. (Also only used in logging.)
- `sources`: Names a directory in which source scripts (which will implement extensions) are located. Defaults to “src”
- `resources`: Names a directory in which resource files (such as HTML templates, images, CS files, and other non-JavaScript files needed by this bundle) are located. Defaults to “res”
- `libraries`: Names a directory in which third-party libraries are located. Defaults to “lib”
- `configuration`: A bundle’s configuration object, which should be formatted as would be passed to require.config (see RequireJS documentation at http://requirejs.org/docs/api.html); note that only paths and shim have been tested.
- `extensions`: An object containing key-value pairs, where keys are extension categories, and values are extension definitions. See the section on Extensions for more information.

For example, the bundle definition for `example/policy` looks like:

```
{
    "name": "Example Policy",
    "description": "Provides an example of using policies.",
    "sources": "src",
    "extensions": {
        "policies": [
            {
                "implementation": "ExamplePolicy.js",
                "category": "action"
            }
        ]
    }
}
```

Bundle Directory Structure

In addition to the directories defined in the bundle definition, a bundle will typically contain other directories not used at run-time. Additionally, some useful development scripts (such as the command line build and the test suite) expect this directory structure to be in use, and may ignore options chosen by `bundle.json`. It is recommended that the directory structure described below be used for new bundles.

- `src`: Contains JavaScript sources for this bundle. May contain additional subdirectories to organize these sources; typically, these subdirectories are named to correspond to the extension categories they contain and/or support, but this is only a convention.
- `res`: Contains other files needed by this bundle, such as HTML templates. May contain additional subdirectories to organize these sources.
- `lib`: Contains JavaScript sources from third-party libraries. These are separated from bundle sources in order to ignore them during code style checking from the command line build.
- `test`: Contains JavaScript sources implementing Jasmine (http://jasmine.github.io/) tests, as well as a file named `suite.json` describing which files to test. Should have the same folder structure as the src directory; see the section on automated testing for more information.

For example, the directory structure for bundle `platform/commonUI/about` looks like:

Extensions

While bundles provide groupings of related behaviors, the individual units of behavior are called extensions.

Extensions belong to categories; an extension category is the machine-readable identifier used to identify groups of extensions. In the `extensions` property of a bundle definition, the keys are extension categories and the values are arrays of extension definitions.

General Extensions

Extensions are intended as a general-purpose mechanism for adding new types of functionality to Open MCT Web.

An extension category is registered with Angular under the name of the extension, plus a suffix of two square brackets; so, an Angular service (or, generally, any other extension) can access the full set of registered extensions, from all bundles, by including this string (e.g. `types[]` to get all type definitions) in a dependency declaration.

As a convention, extension categories are given single-word, plural nouns for names within Open MCT Web (e.g. `types`.) This convention is not enforced by the platform in any way. For extension categories introduced by external plugins, it is recommended to prefix the extension category with a vendor identifier (or similar) followed by a dot, to avoid collisions.

Extension Definitions

The properties used in extension definitions are typically unique to each category of extension; a few properties have standard interpretations by the platform.

- `implementation`: Identifies a JavaScript source file (in the sources folder) which implements this extension. This JavaScript file is expected to contain an AMD module (see http://requirejs.org/docs/whyamd.html#amd) which gives as its result a single constructor function.
- `depends`: An array of dependencies needed by this extension; these will be passed on to Angular's dependency injector, https://docs.angularjs.org/guide/di. By default, this is treated as an empty array. Note that `depends` does not make sense without `implementation` (since these dependencies will be passed to the implementation when it is instantiated.)
- `priority`: A number or string indicating the priority order (see below) of this extension instance. Before an extension category is registered with AngularJS, the extensions of this category from all bundles will be concatenated into a single array, and then sorted by priority.

Extensions do not need to have an implementation. If no implementation is provided, consumers of the extension category will receive the extension definition as a plain JavaScript object. Otherwise, they will receive the partialized (see below) constructor for that implementation, which will additionally have all properties from the extension definition attached.

Partial Construction

In general, extensions are intended to be implemented as constructor functions, which will be used elsewhere to instantiate new objects of that type. However, the Angular-supported method for dependency injection is (effectively) constructor-style injection; so, both declared dependencies and run-time arguments are competing for space in a constructor's arguments.

To resolve this, the Open MCT Web framework registers extension instances in a *partially constructed* form. That is, the constructor exposed by the extension's implementation is effectively decomposed into two calls; the first takes the dependencies, and returns the constructor in its second form, which takes the remaining arguments.

This means that, when writing implementations, the constructor function should be written to include all declared dependencies, followed by all run-time arguments. When using extensions, only the run-time arguments need to be provided.

Priority

Within each extension category, registration occurs in priority order. An extension's priority may be specified as a `priority` property in its extension definition; this may be a number, or a symbolic string. Extensions are registered in reverse order (highest-priority first), and symbolic strings are mapped to the numeric values as follows:

- `fallback`: Negative infinity. Used for extensions that are not intended for use (that is, they are meant to be overridden) but are present as an option of last resort.
- `default`: -100. Used for extensions that are expected to be overridden, but need a useful default.
- `none`: 0. Also used if no priority is specified, or if an unknown or malformed priority is specified.
- `optional`: 100. Used for extensions that are meant to be used, but may be overridden.
- `preferred`: 1000. Used for extensions that are specifically intended to be used, but still may be overridden in principle.
- `mandatory`: Positive infinity. Used when an extension should definitely not be overridden.

These symbolic names are chosen to support usage where many extensions may satisfy a given need, but only one may be used; in this case, as a convention it should be the lowest-ordered (highest-priority) extensions available. In other cases, a full set (or multi-element

subset) of extensions may be desired, with a specific ordering; in these cases, it is preferable to specify priority numerically when declaring extensions, and to understand that extensions will be sorted according to these conventions when using them.

Angular Built-ins

Several entities supported Angular are expressed and managed as extensions in Open MCT Web. Specifically, these extension categories are `directives`, `controllers`, `services`, `constants`, `runs`, and `routes`.

Directives

New directives (see https://docs.angularjs.org/guide/directive) may be registered as extensions of the `directives` category. Implementations of directives in this category should take only dependencies as arguments, and should return a directive definition object.

The directive's name should be provided as a `key` property of its extension definition, in camel-case format.

Controllers

New controllers (see https://docs.angularjs.org/guide/controller) may be registered as extensions of the `controllers` category. The implementation is registered directly as the controller; its only constructor arguments are its declared dependencies.

The directive's identifier should be provided as a `key` property of its extension definition.

Services

New services (see https://docs.angularjs.org/guide/services) may be registered as extensions of the `services` category. The implementation is registered via a service call (https://docs.angularjs.org/api/auto/service/$provide#service), so it will be instantiated with the `new` operator.

Constants

Constant values may be registered as extensions of the `constants` category; see https://docs.angularjs.org/api/ng/type/angular.Module#constant. These extensions have no

implementation; instead, they should contain a property `key`, which is the name under which the constant will be registered, and a property `value`, which is the constant value that will be registered.

Runs

In some cases, you want to register code to run as soon as the application starts; these can be registered as extensions of the `runs` category; see https://docs.angularjs.org/api/ng/type/angular.Module#run. Implementations registered in this category will be invoked (with their declared dependencies) when the Open MCT Web application first starts. (Note that, in this case, the implementation is better thought of as just a function, as opposed to a constructor function.)

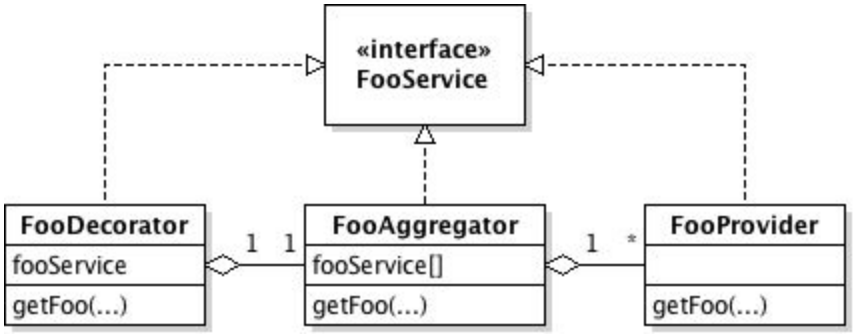
Routes

Extensions of category `routes` will be registered with Angular's route provider, https://docs.angularjs.org/api/ngRoute/provider/$routeProvider. Extensions of this category have no implementations, and need only two properties in their definition:

- `when`: The value that will be passed as the path argument to `$routeProvider.when`; specifically, the string that will appear in the trailing part of the URL corresponding to this route. This property may be omitted, in which case this extension instance will be treated as the default route.
- `templateUrl`: A path to the template to render for this route. Specified as a path relative to the bundle's resource directory (`res` by default.)

Composite Services

A special category of extensions recognized by the framework are `components`; these are parts of services intended to be fit together in a common pattern.



Components all implement the same interface, which is the interface expected for services of the type that they create. Components fall into three types:

- `provider`: Provides an actual implementation of the service in question.
- `aggregator`: Makes many implementations of the service in question appear as one.
- `decorator`: Modifies the inputs or outputs of another implementation of the service.

When the framework layer encounters components, it assembles them into single service instances that can be referred to elsewhere as single dependencies. All providers are instantiated, and passed to the first available aggregator; decorators are then layered on in priority order to create the final form of the service.

A component should include the following properties in its extension definition:

- `provides`: The symbolic identifier for the service that will be composed. The fully-composed service will be registered with Angular under this name.
- `type`: One of `provider`, `aggregator`, or `decorator` (as above)

In addition to any declared dependencies, aggregators and decorators both receive one more argument (immediately following declared dependencies) that is provided by the framework. For an aggregator, this will be an array of all providers of the same service (that is, with matching `provides` properties); for a decorator, this will be whichever provider, decorator, or aggregator is next in the sequence of decorators.

Services exposed by the Open MCT Web platform are often declared as composite services, as this form is open for a variety of common modifications.

Core API

Most of Open MCT Web's relevant API is provided and/or mediated by the framework; that is, much of developing for Open MCT Web is a matter of adding extensions which access other parts of the platform by means of dependency injection.

The core bundle (`platform/core`) introduces a few additional object types meant to be passed along by other services.

Domain Objects

Domain objects are the most fundamental component of Open MCT Web's information model. A domain object is some distinct thing relevant to a user's work flow, such as a telemetry channel, display, or similar. Open MCT Web is a tool for viewing, browsing, manipulating, and otherwise interacting with a graph of domain objects.

A domain object should be conceived of as the union of the following:

- Identifier: A machine-readable string that uniquely identifies the domain object within this application instance.
- Model: The persistent state of the domain object. A domain object's model is a JavaScript object that can be losslessly converted to JSON.
- Capabilities: Dynamic behavior associated with the domain object. Capabilities are JavaScript objects which provide additional methods for interacting with the domain objects which expose those capabilities. Not all domain objects expose all capabilities.

At run-time, a domain object has the following interface:

- `getId()`: Get the identifier for this domain object.
- `getModel()`: Get the plain state associated with this domain object. This will return a JavaScript object that can be losslessly converted to JSON. Note that the model returned here can be modified directly but should not be; instead, use the `mutation` capability.
- `getCapability(key)`: Get the specified capability associated with this domain object. This will return a JavaScript object whose interface is specific to the type of capability being requested. If the requested capability is not exposed by this domain object, this will return `undefined`.

- `hasCapability(key)`: Shorthand for checking if a domain object exposes the requested capability.
- `useCapability(key, arguments…)`: Shorthand for `getCapability(key).invoke(arguments)`, with additional checking between calls. If the provided capability has no invoke method, the return value here functions as `getCapability`, including returning `undefined` if the capability is not exposed.

Actions

An `Action` is behavior that can be performed upon/using a `DomainObject`. An Action has the following interface:

- `perform()`: Do this action. For example, if one had an instance of a `RemoveAction`, invoking its `perform` method would cause the domain object which exposed it to be removed from its container.
- `getMetadata()`: Get metadata associated with this action. Returns an object containing:
    - `name`: Human-readable name.
    - `description`: Human-readable summary of this action.
    - `glyph`: Single character to be displayed in Open MCT Web's icon font set.
    - `context`: The context in which this action is being performed (see below)

Action instances are typically obtained via a domain object's `action` capability.

Action Contexts

An action context is a JavaScript object with the following properties:

- `domainObject`: The domain object being acted upon.
- `selectedObject`: Optional; the selection at the time of action (e.g. the dragged object in a drag-and-drop operation.)

Telemetry

Telemetry series data in Open MCT Web is represented by a common interface, and packaged in a consistent manner to facilitate passing telemetry updates around multiple visualizations.

Telemetry Requests

A telemetry request is a JavaScript object containing the following properties:

- `source`: A machine-readable identifier for the source of this telemetry. This is useful when multiple distinct data sources are in use side-by-side.
- `key`: A machine-readable identifier for a unique series of telemetry within that source.
- *Note: This API is still under development; additional properties, such as start and end time, should be present in future versions of Open MCT Web.*

Additional properties may be included in telemetry requests which have specific interpretations for specific sources.

Telemetry Responses

When returned from the `telemetryService` (see Services section), telemetry series data will be packaged in a `source -> key -> TelemetrySeries` fashion. That is, telemetry is passed in an object containing key-value pairs. Keys identify telemetry sources; values are objects containing additional key-value pairs. In this object, keys identify individual telemetry series (and match they `key` property from corresponding requests) and values are `TelemetrySeries` objects (see below.)

Telemetry Series

A telemetry series is a specific sequence of data, typically associated with a specific instrument. Telemetry is modeled as an ordered sequence of domain and range values, where domain values must be non-decreasing but range values do not. (Typically, domain values are interpreted as UTC timestamps in milliseconds relative to the UNIX epoch.) A series must have at least one domain and one range, and may have more than one.

Telemetry series data in Open MCT Web is expressed via the following `TelemetrySeries` interface:

- `getPointCount()`: Returns the number of unique points/samples in this series.
- `getDomainValue(index, [domain])`: Get the domain value at the specified `index`. If a second `domain` argument is provided, this is taken as a string identifier indicating which domain option (of, presumably, multiple) should be returned.
- `getRangeValue(index, [range])`: Get the domain value at the specified `index`. If a second `range` argument is provided, this is taken as a string identifier indicating which range option (of, presumably, multiple) should be returned.

Telemetry Metadata

Domain objects which have associated telemetry also expose metadata about that telemetry; this is retrievable via the `getMetadata()` of the telemetry capability. This will return a single JavaScript object containing the following properties:

- `source`: The machine-readable identifier for the source of telemetry data for this object.
- `key`: The machine-readable identifier for the individual telemetry series.
- `domains`: An array of supported domains (see `TelemetrySeries` above.) Each domain should be expressed as an object which includes:
    - `key`: Machine-readable identifier for this domain, as will be passed into a `getDomainValue(index, domain)` call.
    - `name`: Human-readable name for this domain.
- `ranges`: An array of supported ranges; same format as `domains`.

Note that this metadata is also used as the prototype for telemetry requests made using this capability.

Types

A domain object's type is represented as a `Type` object, which has the following interface:

- `getKey()`: Get the machine-readable identifier for this type.
- `getName()`: Get the human-readable name for this type.
- `getDescription()`: Get a human-readable summary of this type.
- `getGlyph()`: Get the single character to be rendered as an icon for this type in Open MCT Web's custom font set.
- `getInitialModel()`: Get a domain object model that represents the initial state (before user specification of properties) for domain objects of this type.
- `getDefinition()`: Get the extension definition for this type, as a JavaScript object.
- `instanceOf(type)`: Check if this type is (or inherits from) a specified `type`. This type can be either a string, in which case it is taken to be that type's `key`, or it may be a `Type` instance.
- `hasFeature(feature)`: Returns a boolean value indicating whether or not this type supports the specified `feature`, which is a symbolic string.
- `getProperties()`: Get all properties associated with this type, expressed as an array of `TypeProperty` instances.

Type Features

Features of a domain object type are expressed as symbolic string identifiers. They are defined in practice by usage; currently, the Open MCT Web platform only uses the `creation` feature to determine which domain object types should appear in the Create menu.

Type Properties

Types declare the user-editable properties of their domain object instances in order to allow the forms which appear in the Create and Edit Properties dialogs to be generated by the platform. A `TypeProperty` has the following interface:

- `getValue(model)`: Get the current value for this property, as it appears in the provided domain object `model`.
- `setValue(model, value)`: Set a new `value` for this property in the provided domain object `model`.
- `getDefinition()`: Get the raw definition for this property as a JavaScript object (as it was declared in this type's extension definition.)

Extension Categories

The information in this section is focused on registering new extensions of specific types; it does not contain a catalog of the extension instances of these categories provided by the platform. Relevant summaries there are provided in subsequent sections.

Actions

An action is a thing that can be done to or using a domain object, typically as initiated by the user.

An action's implementation:

- Should take a single `context` argument in its constructor. (See Action Contexts, under Core API.)
- Should provide a method `perform`, which causes the behavior associated with the action to occur.
- May provide a method `getMetadata`, which provides metadata associated with the action. If omitted, one will be provided by the platform which includes metadata from the action's extension definition.
- May provide a static method `appliesTo(context)` (that is, a function available as a property of the implementation's constructor itself), which will be used by the platform to filter out actions from contexts in which they are inherently inapplicable.

An action's bundle definition (and/or `getMetadata()` return value) may include:

- `category`: A string or dearray of strings identifying which category or categories an action falls into; used to determine when an action is displayed. Categories supported by the platform include:
    - `contextual`: Actions in a context menu.
    - `view-control`: Actions triggered by buttons in the top-right of Browse view.
- `key`: A machine-readable identifier for this action.
- `name`: A human-readable name for this action (e.g. to show in a menu)
- `description`: A human-readable summary of the behavior of this action.
- `glyph`: A single character which will be rendered in Open MCT Web's custom font set as an icon for this action.

Capabilities

Capabilities are exposed by domain objects (e.g. via the `getCapability` method) but most commonly originate as extensions of this category.

Extension definitions for capabilities should include both an implementation, and a property named `key` whose value should be a string used as a machine-readable identifier for that capability, e.g. when passed as the argument to a domain object's `getCapability(key)` call.

A capability's implementation should have methods specific to that capability; that is, there is no common format for capability implementations, aside from support for `invoke` via the `useCapability` shorthand.

A capability's implementation will take a single argument (in addition to any declared dependencies), which is the domain object that will expose that capability.

A capability's implementation may also expose a static method `appliesTo(model)` which should return a boolean value, and will be used by the platform to filter down capabilities to those which should be exposed by specific domain objects, based on their domain object models.

Controls

Controls provide options for the `mct-control` directive.

Four standard control types are included in the forms bundle:

- `textfield`: An area to enter plain text.
- `select`: A drop-down list of options.
- `checkbox`: A box which may be checked/unchecked.
- `color`: A color picker.
- `button`: A button.
- `datetime`: An input for UTC date/time entry; gives result as a UNIX timestamp, in milliseconds since start of 1970, UTC.

New controls may be added as extensions of the controls category. Extensions of this category have two properties:

- `key`: The symbolic name for this control (matched against the control field in rows of the form structure).
- `templateUrl`: The URL to the control's Angular template, relative to the resources directory of the bundle which exposes the extension.

Within the template for a control, the following variables will be included in scope:

- `ngModel`: The model where form input will be stored. Notably we also need to look at `field` (see below) to determine which field in the model should be modified.
- `ngRequired`: True if input is required.
- `ngPattern`: The pattern to match against (for text entry.)
- `options`: The options for this control, as passed from the `options` property of an individual row definition.
- `field`: Name of the field in `ngModel` which will hold the value for this control.

Gestures

A gesture is a user action which can be taken upon a representation of a domain object. Examples of gestures included in the platform are:

- `drag`: For representations that can be used to initiate drag-and-drop composition.
- `drop`: For representations that can be drop targets for drag-and-drop composition.
- `menu`: For representations that can be used to pop up a context menu.

Gesture definitions have a property `key` which is used as a machine-readable identifier for the gesture (e.g. `drag`, `drop`, `menu` above.)

A gesture's implementation is instantiated once per representation that uses the gesture. This class will receive the jqLite-wrapped `mct-representation` element and the domain object being represented as arguments, and should do any necessary "wiring" (e.g. listening for events) during its constructor call. The gesture's implementation may also expose an optional `destroy()` method which will be called when the gesture should be removed, to avoid memory leaks by way of unremoved listeners.

Indicators

An indicator is an element that should appear in the status area at the bottom of a running Open MCT Web client instance.

Standard Indicators

Indicators which wish to appear in the common form of an icon-text pair should provide implementations with the following methods:

- `getText()`: Provides the human-readable text that will be displayed for this indicator.
- `getGlyph()`: Provides a single-character string that will be displayed as an icon in Open MCT Web's custom font set.
- `getDescription()`: Provides a human-readable summary of the current state of this indicator; will be displayed in a tooltip on hover.
- `getClass()`: Get a CSS class that will be applied to this indicator.
- `getTextClass()`: Get a CSS class that will be applied to this indicator's text portion.
- `getGlyphClass()`: Get a CSS class that will be applied to this indicator's icon portion.
- `configure()`: If present, a configuration icon will appear to the right of this indicator, and clicking it will invoke this method.

Note that all methods are optional, and are called directly from an Angular template, so they should be appropriate to run during digest cycles.

Custom Indicators

Indicators which wish to have an arbitrary appearance (instead of following the icon-text convention commonly used) may specify a `template` property in their extension definition. The value of this property will be used as the `key` for an `mct-include` directive (so should refer to an extension of category `templates`.) This template will be rendered to the status area. Indicators of this variety do not need to provide an implementation.

Licenses

The extension category `licenses` can be used to add entries into the "Licensing information" page, reachable from Open MCT Web's About dialog.

Licenses may have the following properties, all of which are strings:

- `name`: Human-readable name of the licensed component. (e.g. "AngularJS".)
- `version`: Human-readable version of the licensed component. (e.g. "1.2.26".)
- `description`: Human-readable summary of the component.
- `author`: Name or names of entities to which authorship should be attributed.
- `copyright`: Copyright text to display for this component.
- `link`: URL to full license text.

Policies

Policies are used to handle decisions made using Open MCT Web's `policyService`; examples of these decisions are determining the applicability of certain actions, or checking whether or not a domain object of one type can contain a domain object of a different type. See the section on the Policies for an overview of Open MCT Web's policy model.

A policy's extension definition should include:

- `category`: The machine-readable identifier for the type of policy decision being supported here. For a list of categories supported by the platform, see the section on Policies. Plugins may introduce and utilize additional policy categories not in that list.
- `message`: Optional; a human-readable message describing the policy, intended for display in situations where this specific policy has disallowed something.

A policy's implementation should include a single method, `allow(candidate, context)`. The specific types used for `candidate` and `context` vary by policy category; in general, what is being asked is "is this candidate allowed in this context?" This method should return a boolean value.

Open MCT Web's policy model requires consensus; a policy decision is allowed when and only when all policies choose to allow it. As such, policies should generally be written to reject a certain case, and allow (by returning true) anything else.

Representations

A representation is an Angular template used to display a domain object. The `representations` extension category is used to add options for the `mct-representation` directive.

A representation definition should include the following properties:

- `key`: The machine-readable name which identifies the representation.
- `templateUrl`: The path to the representation's Angular template. This path is relative to the bundle's resources directory.
- `uses`: Optional; an array of capability names. Indicates that this representation intends to use those capabilities of a domain object (via a `useCapability` call), and expects to find the latest results of that `useCapability` call in the scope of the presented template (under the same name as the capability itself.) Note that, if `useCapability` returns a promise, this will be resolved before being placed in the representation's scope.

- `gestures`: An array of keys identifying gestures (see the `gestures` extension category) which should be available upon this representation. Examples of gestures include `drag` (for representations that should act as draggable sources for drag-drop operations) and `menu` (for representations which should show a domain-object-specific context menu on right-click.)

Representation Scope

While `representations` do not have implementations, per se, they do refer to Angular templates which need to interact with information (e.g. the domain object being represented) provided by the platform. This information is passed in through the template's scope, such that simple representations may be created by providing only templates. (More complex representations will need controllers which are referenced from templates. See https://docs.angularjs.org/guide/controller for more information on controllers in Angular.)

A representation's scope will contain:

- `domainObject`: The represented domain object.
- `model`: The domain object's model.
- `configuration`: An object containing configuration information for this representation (an empty object if there is no saved configuration.) The contents of this object are managed entirely by the view/representation which receives it.
- `representation`: An empty object, useful as a "scratch pad" for representation state.
- `ngModel`: An object passed through the `ng-model` attribute of the `mct-representation`, if any.
- `parameters`: An object passed through the `parameters` attribute of the `mct-representation`, if any.
- Any capabilities requested by the `uses` property of the representation definition.

Representers

The `representers` extension category is used to add additional behavior to the `mct-representation` directive. This extension category is intended primarily for use internal to the platform.

Unlike represent*ations*, which describe specific ways to represent domain objects, represent*ers* are used to modify or augment the process of representing domain objects in general. For example, support for the `gestures` extension category is added by a representer.

A representer needs only provide an implementation. When an `mct-representation` is linked (see https://docs.angularjs.org/guide/directive) or when the domain object being represented changes, a new representer of each declared type is instantiated. The constructor arguments for a representer are the same as the arguments to the link function in an Angular

directive: `scope`, the Angular scope for this representation; `element`, the jqLite-wrapped `mct-representation` element, and `attrs`, a set of key-value pairs of that element's attributes. Representers may wish to populate the scope, attach event listeners to the element, etc.

This implementation must provide a single method, `destroy()`, which will be invoked when the representer is no longer needed.

Roots

The extension category `roots` is used to provide root-level domain object models. Root-level domain objects appear at the top-level of the tree hierarchy. For example, the "My Items" folder is added as an extension of this category.

Extensions of this category should have the following properties:

- `id`: The machine-readable identifier for the domain object being exposed.
- `model`: The model, as a JSON object, for the domain object being exposed.

Stylesheets

The `stylesheets` extension category is used to add CSS files to style the application. Extension definitions for this category should include one property:

- `stylesheetUrl`: Path and filename, including extension, for the stylesheet to include. This path is relative to the bundle's resources folder (by default, `res`)

To control the order of CSS files, use `priority` (see the section on Extension Definitions above.)

Templates

The `templates` extension category is used to expose Angular templates under symbolic identifiers. These can then be utilized using the `mct-include` directive, which behaves similarly to `ng-include`, except that it uses these symbolic identifiers instead of paths.

A template's extension definition should include the following properties:

- `key`: The machine-readable name which identifies this template, matched against the value given to the key attribute of the mct-include directive.
- `templateUrl`: The path to the relevant Angular template. This path is relative to the bundle's resources directory.

Note that, when multiple templates are present with the same `key`, the one with the highest priority will be used from mct-include. This behavior can be used to override templates exposed by the platform (to change the logo which appears in the bottom right, for instance.)

Templates do not have implementations.

Types

The `types` extension category describes types of domain objects which may appear within Open MCT Web.

A type's extension definition should have the following properties:

- `key`: The machine-readable identifier for this domain object type. Will be stored to and matched against the `type` property of domain object models.
- `name`: The human-readable name for this domain object type.
- `description`: A human-readable summary of this domain object type.
- `glyph`: A single character to be rendered as an icon in Open MCT Web's custom font set.
- `model`: A domain object model, used as the initial state for created domain objects of this type (before any properties are specified.)
- `features`: Optional; an array of strings describing features of this domain object type. Currently, only `creation` is recognized by the platform; this is used to determine that this type should appear in the Create menu. More generally, this is used to support the `hasFeature(...)` method of the `type` capability.
- `properties`: An array describing individual properties of this domain object (as should appear in the Create or the Edit Properties dialog.) Each property is described by an object containing the following properties:

    - `control`: The key of the control (see mct-control and the controls extension category) to use for editing this property.
    - `property`: A string which will be used as the name of the property in the domain object's model that the value for this property should be stored under. If this value should be stored in an object nested within the domain object model, then property should be specified as an array of strings identifying these nested objects and, finally, the property itself.
    - ...other properties as appropriate for a control of this type (each property's definition will also be passed in as the structure for its control.) See documentation of `mct-form` for more detail on these properties.

Types do not have implementations.

Versions

The `versions` extension category is used to introduce line items in Open MCT Web's About dialog. These should have the following properties:

- `name`: The name of this line item, as should appear in the left-hand side of the list of version information in the About dialog.
- `value`: The value which should appear to the right of the name in the About dialog.

To control the ordering of line items within the About dialog, use `priority`. (See section on Extension Definitions above.)

This extension category does not have implementations.

Views

The `views` extension category is used to determine which options appear to the user as available views of domain objects of specific types. A view's extension definition has the same properties as a representation (and views can be utilized via `mct-representation`); additionally:

- `name`: The human-readable name for this view type.
- `description`: A human-readable summary of this view type.
- `glyph`: A single character to be rendered as an icon in Open MCT Web's custom font set.
- `type`: Optional; if present, this representation is only applicable for domain object's of this type.

- `needs`: Optional array of strings; if present, this representation is only applicable for domain objects which have the capabilities identified by these strings.
- `delegation`: Optional boolean, intended to be used in conjunction with `needs`; if present, allow required capabilities to be satisfied by means of capability delegation. (See the `delegation` capability, in the Capabilities section.)
- `toolbar`: Optional; a definition for the toolbar which may appear in a toolbar when using this view in Edit mode. This should be specified as a structure for `mct-toolbar`, with additional properties available for each item in that toolbar:
    - `property`: A property name. This will refer to a property in the view's current selection; that property on the selected object will be modifiable as the `ng-model` of the displayed control in the toolbar. If the value of the property is a function, it will be used as a getter-setter (called with no arguments to use as a getter, called with a value to use as a setter.)
    - `method`: A method to invoke (again, on the selected object) from the toolbar control. Useful particularly for buttons (which don't edit a single property, necessarily.)

View Scope

Views do not have implementations, but do get the same properties in scope that are provided for `representations`.

When a view is in Edit mode, this scope will additionally contain:

- `commit()`: A function which can be invoked to mark any changes to the view's `configuration` as ready to persist.
- `selection`: An object representing the current selection state.

Selection State

A view's selection state is, conceptually, a set of JavaScript objects. The presence of methods/properties on these objects determine which toolbar controls are visible, and what state they manage and/or behavior they invoke.

This set may contain up to two different objects: The *view proxy*, which is used to make changes to the view as a whole, and the *selected object*, which is used to represent some state within the view. (Future versions of Open MCT Web may support multiple selected objects.)

The `selection` object made available during Edit mode has the following methods:

- `proxy([object])`: Get (or set, if called with an argument) the current view proxy.
- `select(object)`: Make this object the selected object.
- `deselect()`: Clear the currently selected object.
- `get()`: Get the currently selected object. Returns `undefined` if there is no currently selected object.
- `selected(object)`: Check if the JavaScript object is currently in the selection set. Returns `true` if the object is either the currently selected object, or the current view proxy.
- `all()`: Get an array of all objects in the selection state. Will include either or both of the view proxy and selected object.

Directives

Open MCT Web defines several Angular directives that are intended for use both internally within the platform, and by plugins.

Before Unload

The `mct-before-unload` directive is used to listen for (and prompt for user confirmation) of navigation changes in the browser. This includes reloading, following links out of Open MCT Web, or changing routes. It is used to hook into both `onbeforeunload` event handling as well as route changes from within Angular.

This directive is useable as an attribute. Its value should be an Angular expression. When an action that would trigger an unload and/or route change occurs, this Angular expression is evaluated. Its result should be a message to display to the user to confirm their navigation change; if this expression evaluates to a falsy value, no message will be displayed.

Chart

The `mct-chart` directive is used to support drawing of simple charts. It is present to support the Plot view, and its functionality is limited to the functionality that is relevant for that view.

This directive is used at the element level and takes one attribute, `draw`, which is an Angular expression which will should evaluate to a drawing object. This drawing object should contain the following properties:

- `dimensions`: The size, in logical coordinates, of the chart area. A two-element array or numbers.
- `origin`: The position, in logical coordinates, of the lower-left corner of the chart area. A two-element array or numbers.
- `lines`: An array of lines (e.g. as a plot line) to draw, where each line is expressed as an object containing:
    - `buffer`: A Float32Array containing points in the line, in logical coordinates, in sequential x,y pairs.
    - `color`: The color of the line, as a four-element RGBA array, where each element is a number in the range of 0.0-1.0.
    - `points`: The number of points in the line.
- `boxes`: An array of rectangles to draw in the chart area. Each is an object containing:
    - `start`: The first corner of the rectangle, as a two-element array of numbers, in logical coordinates.

    - `end`: The opposite corner of the rectangle, as a two-element array of numbers, in logical coordinates.
    - `color`: The color of the line, as a four-element RGBA array, where each element is a number in the range of 0.0-1.0.

While `mct-chart` is intended to support plots specifically, it does perform some useful management of canvas objects (e.g. choosing between WebGL and Canvas 2D APIs for drawing based on browser support) so its usage is recommended when its supported drawing primitives are sufficient for other charting tasks.

Container

The `mct-container` is similar to the `mct-include` directive insofar as it allows templates to be referenced by symbolic keys instead of by URL. Unlike `mct-include`, it supports transclusion.

Unlike `mct-include`, `mct-container` accepts a `key` as a plain string attribute, instead of as an Angular expression.

Control

The `mct-control` directive is used to display user input elements. Several controls are included with the platform to wrap default input types. This directive is primarily intended for internal use by the `mct-form` and `mct-toolbar` directives.

When using `mct-control`, the attributes `ng-model`, `ng-disabled`, `ng-required`, and `ng-pattern` may also be used. These have the usual meaning (as they would for an input element) except for `ng-model`; when used, it will actually be `ngModel[field]` (see below) that is two-way bound by this control. This allows `mct-control` elements to more easily delegate to other `mct-control` instances, and also facilitates usage for generated forms.

This directive supports the following additional attributes, all specified as Angular expressions:

- `key`: A machine-readable identifier for the specific type of control to display.
- `options`: A set of options to display in this control.
- `structure`: In practice, contains the definition object which describes this form row or toolbar item. Used to pass additional control-specific parameters.
- `field`: The field in the `ngModel` under which to read/store the property associated with this control.

Drag

The `mct-drag` directive is used to support drag-based gestures on HTML elements. Note that this is not “drag” in the “drag-and-drop” sense, but “drag” in the more general “mouse down, mouse move, mouse up” sense.

This takes the form of three attributes:

- `mct-drag`: An Angular expression to evaluate during drag movement.
- `mct-drag-down`: An Angular expression to evaluate when the drag starts.
- `mct-drag-up`: An Angular expression to evaluate when the drag ends.

In each case, a variable `delta` will be provided to the expression; this is a two-element array or the horizontal and vertical pixel offset of the current mouse position relative to the mouse position where dragging began.

Form

The `mct-form` directive is used to generate forms using a declarative structure, and to gather back user input. It is applicable at the element level and supports the following attributes:

- `ng-model`: The object which should contain the full form input. Individual fields in this model are bound to individual controls; the names used for these fields are provided in the form structure (see below).
- `structure`: The structure of the form; e.g. sections, rows, their names, and so forth. The value of this attribute should be an Angular expression.
- `name`: The name in the containing scope under which to publish form "meta-state", e.g. `$valid`, `$dirty`, etc. This is as the behavior of `ng-form`. Passed as plain text in the attribute.

Form Structure

Forms in Open MCT Web have a common structure to permit consistent display. A form is broken down into sections, which will be displayed in groups; each section is broken down into rows, each of which provides a control for a single property. Input from this form is two-way bound to the object passed via `ng-model`.

A form's structure is represented by a JavaScript object in the following form:

```
{
    "name": ... title to display for the form, as a string ...,
    "sections": [
        {
            "name": ... title to display for the section ...,
            "rows": [
                {
                    "name": ... title to display for this row ...,
                    "control": ... symbolic key for the control ...,
                    "key": ... field name in ng-model ...
                    "pattern": ... optional, reg exp to match against ...
                    "required": ... optional boolean ...
                    "options": [
                        "name": ... name to display (e.g. in a select) ...,
                        "value": ... value to store in the model ...
                    ]
                },
                ... and other rows ...
            ]
        },
        ... and other sections ...
    ]
}
```

Note that `pattern` may be specified as a string, to simplify storing for structures as JSON when necessary. The string should be given in a form appropriate to pass to a `RegExp` constructor.

Form Controls

A few standard control types are included in the `platform/forms` bundle:

- `textfield`: An area to enter plain text.
- `select`: A drop-down list of options.
- `checkbox`: A box which may be checked/unchecked.
- `color`: A color picker.
- `button`: A button.
- `datetime`: An input for UTC date/time entry; gives result as a UNIX timestamp, in milliseconds since start of 1970, UTC.

Include

The `mct-include` directive is similar to `ng-include`, except that it takes a symbolic identifier for a template instead of a URL. Additionally, templates included via `mct-include` will have an isolated scope.

The directive should be used at the element level and supports the following attributes, all of which are specified as Angular expressions:

- `key`: Machine-readable identifier for the template (of extension category `templates`) to be displayed.
- `ng-model`: Optional; will be passed into the template's scope as `ngModel`. Intended usage is for two-way bound user input.
- `parameters`: Optional; will be passed into the template's scope as `parameters`. Intended usage is for template-specific display parameters.

Representation

The `mct-representation` directive is used to include templates which specifically represent domain objects. Usage is similar to `mct-include`.

The directive should be used at the element level and supports the following attributes, all of which are specified as Angular expressions:

- `key`: Machine-readable identifier for the representation (of extension category `representations` or `views`) to be displayed.
- `mct-object`: The domain object being represented.
- `ng-model`: Optional; will be passed into the template's scope as `ngModel`. Intended usage is for two-way bound user input.
- `parameters`: Optional; will be passed into the template's scope as `parameters`. Intended usage is for template-specific display parameters.

Resize

The `mct-resize` directive is used to monitor the size of an HTML element. It is specified as an attribute whose value is an Angular expression that will be evaluated when the size of the HTML element changes. This expression will be provided a single variable, `bounds`, which is an object containing two properties, `width` and `height`, describing the size in pixels of the element.

When using this directive, an attribute `mct-resize-interval` may optionally be provided. Its value is an Angular expression describing the number of milliseconds to wait before next checking the size of the HTML element; this expression is evaluated when the directive is linked and reevaluated whenever the size is checked.

Scroll

The `mct-scroll-x` and `mct-scroll-y` directives are used to both monitor and control the horizontal and vertical scroll bar state of an element, respectively. They are intended to be used as attributes whose values are assignable Angular expressions which two-way bind to the scroll bar state.

Toolbar

The `mct-toolbar` directive is used to generate toolbars using a declarative structure, and to gather back user input. It is applicable at the element level and supports the following attributes:

- `ng-model`: The object which should contain the full toolbar input. Individual fields in this model are bound to individual controls; the names used for these fields are provided in the form structure (see below).
- `structure`: The structure of the toolbar; e.g. sections, rows, their names, and so forth. The value of this attribute should be an Angular expression.
- `name`: The name in the containing scope under which to publish form "meta-state", e.g. `$valid`, `$dirty`, etc. This is as the behavior of `ng-form`. Passed as plain text in the attribute.

Toolbars support the same `control` options as forms.

Toolbar Structure

A toolbar's structure is defined similarly to forms, except instead of `rows` there are `items`.

```
{
    "name": ... title to display for the form, as a string ...,
    "sections": [
        {
            "name": ... title to display for the section ...,
            "items": [
                {
                    "name": ... title to display for this row ...,
                    "control": ... symbolic key for the control ...,
                    "key": ... field name in ng-model ...
                    "pattern": ... optional, reg exp to match against ...
                    "required": ... optional boolean ...
                    "options": [
                        "name": ... name to display (e.g. in a select) ...,
                        "value": ... value to store in the model ...
                    ],
                    "disabled": ... true if control should be disabled ...
                    "size": ... size of the control (for textfields) ...
                    "click": ... function to invoke (for buttons) ...
                    "glyph": ... glyph to display (for buttons) ...
                    "text": ... text within control (for buttons) ...
```

```
                },
                ... and other rows ...
            ]
        },
        ... and other sections ...
    ]
}
```

Services

The Open MCT Web platform provides a variety of services which can be retrieved and utilized via dependency injection. These services fall into two categories:

- Composite Services are defined by a set of `components` extensions; plugins may introduce additional components with matching interfaces to extend or augment the functionality of the composed service. (See the Framework section on Composite Services.)
- Other services which are defined as standalone service objects; these can be utilized by plugins but are not intended to be modified or augmented.

Composite Services

This section describes the composite services exposed by Open MCT Web, specifically focusing on their interface and contract.

In many cases, the platform will include a provider for a service which consumes a specific extension category; for instance, the `actionService` depends on `actions[]` and will expose available actions based on the rules defined for that extension category.

In these cases, it will usually be simpler to add a new extension of a given category (e.g. of category `actions`) even when the same behavior could be introduced by a service component (e.g. an extension of category `components` where `provides` is `actionService`, and `type` is `provider`.)

Occasionally, the extension category does not provide enough expressive power to achieve a desired result. For instance, the Create menu is populated with `create` actions, where one such action exists for each creatable type. Since the framework does not provide a declarative means to introduce a new action per type declaratively, the platform implements this explicitly in an `actionService` component of type `provider`. Plugins may use a similar approach when the normal extension mechanism is insufficient to achieve a desired result.

Action Service

The `actionService` provides `Action` instances which are applicable in specific contexts. See Core API for additional notes on the interface for actions.

The `actionService` has the following interface:

- `getActions(context)`: Returns an array of `Action` objects which are applicable in the specified action context.

Capability Service

The `capabilityService` provides constructors for capabilities which will be exposed for a given domain object.

The `capabilityService` has the following interface:

- `getCapabilities(model)`: Returns a an object containing key-value pairs, representing capabilities which should be exposed by the domain object with this model. Keys in this object are the capability keys (as used in a `getCapability(...)` call) and values are either:
    - Functions, in which case they will be used as constructors, which will receive the domain object instance to which the capability applies as their sole argument. The resulting object will be provided as the result of a domain object's `getCapability(...)` call. Note that these instances are cached by each object, but may be recreated when an object is mutated.
    - Other objects, which will be used directly as the result of a domain object's `getCapability(...)` call.

Dialog Service

The `dialogService` provides a means for requesting user input via a modal dialog. It has the following interface:

- `getUserInput(formStructure, formState)`: Prompt the user to fill out a form. The first argument describes the form's structure (as will be passed to `mct-form`) while the second argument contains the initial state of that form. This returns a `Promise` for the state of the form after the user has filled it in; this promise will be rejected if the user cancels input.
- `getUserChoice(dialogStructure)`: Prompt the user to make a single choice from a set of options, which (in the platform implementation) will be expressed as buttons in the displayed dialog. Returns a `Promise` for the user's choice, which will be rejected if the user cancels input.

Dialog Structure

The object passed as the `dialogStructure` to `getUserChoice` should have the following properties:

- `title`: The title to display at the top of the dialog.
- `hint`: Short message to display below the title.
- `template`: Identifying `key` (as will be passed to `mct-include`) for the template which will be used to populate the inner area of the dialog.
- `model`: Model to pass in the `ng-model` attribute of `mct-include`.
- `parameters`: Parameters to pass in the `parameters` attribute of `mct-include`.
- `options`: An array of options describing each button at the bottom. Each option may have the following properties:
    - `name`: Human-readable name to display in the button.
    - `key`: Machine-readable key, to pass as the result of the resolved promise when clicked.
    - `description`: Description to show in tooltip on hover.

Domain Object Service

The `objectService` provides domain object instances. It has the following interface:

- `getObjects(ids)`: For the provided array of domain object identifiers, returns a `Promise` for an object containing key-value pairs, where keys are domain object identifiers and values are corresponding `DomainObject` instances. Note that the result may contain a superset or subset of the objects requested.

Gesture Service

The `gestureService` is used to attach gestures (see extension category `gestures`) to representations. It has the following interface:

- `attachGestures(element, domainObject, keys)`: Attach gestures specified by the provided gesture `keys` (an array of strings) to this jqLite-wrapped HTML `element`, which represents the specified `domainObject`. Returns an object with a single method `destroy()`, to be invoked when it is time to detach these gestures.

Model Service

The `modelService` provides domain object models. It has the following interface:

- `getModels(ids)`: For the provided array of domain object identifiers, returns a `Promise` for an object containing key-value pairs, where keys are domain object identifiers and values are corresponding domain object models. Note that the result may contain a superset or subset of the models requested.

Persistence Service

The `persistenceService` provides the ability to load/store JavaScript objects (presumably serializing/deserializing to JSON in the process.) This is used primarily to store domain object models. It has the following interface:

- `listSpaces()`: Returns a `Promise` for an array of strings identifying the different persistence spaces this service supports. Spaces are intended to be used to distinguish between different underlying persistence stores, to allow these to live side by side.
- `listObjects()`: Returns a Promise for an array of strings identifying all documents stored in this persistence service.
- `createObject(space, key, value)`: Create a new document in the specified persistence `space`, identified by the specified `key`, the contents of which shall match the specified `value`. Returns a promise that will be rejected if creation fails.
- `readObject(space, key)`: Read an existing document in the specified persistence `space`, identified by the specified `key`. Returns a promise for the specified document; this promise will resolve to `undefined` if the document does not exist.
- `updateObject(space, key, value)`: Update an existing document in the specified persistence `space`, identified by the specified `key`, such that its contents match the specified `value`. Returns a promise that will be rejected if the update fails.
- `deleteObject(space, key)`: Delete an existing document from the specified persistence `space`, identified by the specified `key`. Returns a promise which will be rejected if deletion fails.

Policy Service

The `policyService` may be used to determine whether or not certain behaviors are allowed within the application. It has the following interface:

- `allow(category, candidate, context, [callback])`: Check if this decision should be allowed. Returns a boolean. Its arguments are interpreted as:
    - `category`: A string identifying which kind of decision is being made. See the section on Policies for categories supported by the platform; plugins may define and utilize policies of additional categories, as well.
    - `candidate`: An object representing the thing which shall or shall not be allowed. Usually, this will be an instance of an extension of the category defined above. This does need to be the case; additional policies which are not specific to any extension may also be defined and consulted using unique category identifiers. In this case, the type of the object delivered for the candidate may be unique to the policy type.
    - `context`: An object representing the context in which the decision is occurring. Its contents are specific to each policy category.
    - `callback`: Optional; a function to call if the policy decision is rejected. This function will be called with the message string (which may be undefined) of whichever individual policy caused the operation to fail.

Telemetry Service

The `telemetryService` is used to acquire telemetry data. See the section on Telemetry in Core API for more information on how both the arguments and responses of this service are structured.

When acquiring telemetry for display, it is recommended that the `telemetryHandler` service be used instead of this service. The `telemetryHandler` has additional support for subscribing to and requesting telemetry data associated with domain objects or groups of domain objects. See the Other Services section for more information.

The `telemetryService` has the following interface:

- `requestTelemetry(requests)`: Issue a request for telemetry, matching the specified telemetry `requests`. Returns a `Promise` for a telemetry response object.
- `subscribe(callback, requests)`: Subscribe to real-time updates for telemetry, matching the specified `requests`. The specified `callback` will be invoked with telemetry response objects as they become available. This method returns a function which can be invoked to terminate the subscription.

Type Service

The `typeService` exposes domain object types. It has the following interface:

- `listTypes()`: Returns all domain object types supported in the application, as an array of `Type` instances.
- `getType(key)`: Returns the `Type` instance identified by the provided key, or `undefined` if no such type exists.

View Service

The `viewService` exposes definitions for views of domain objects. It has the following interface:

- `getViews(domainObject)`: Get an array of extension definitions of category `views` which are valid and applicable to the specified `domainObject`.

Other Services

Drag and Drop

The `dndService` provides information about the content of an active drag-and-drop gesture within the application. It is intended to complement the `DataTransfer` API of HTML5 drag-and-drop, by providing access to non-serialized JavaScript objects being dragged, as well as by permitting inspection during drag (which is normally prohibited by browsers for security reasons.)

The `dndService` has the following methods:

- `setData(key, value)`: Set drag data associated with a given type, specified by the `key` argument.
- `getData(key)`: Get drag data associated with a given type, specified by the `key` argument.
- `removeData(key)`: Clear drag data associated with a given type, specified by the `key` argument.

Navigation

The `navigationService` provides information about the current navigation state of the application; that is, which object is the user currently viewing? This service merely tracks this state and notifies listeners; it does not take immediate action when navigation changes, although its listeners might.

The `navigationService` has the following methods:

- `getNavigation()`: Get the current navigation state. Returns a `DomainObject`.
- `setNavigation(domainObject)`: Set the current navigation state. Returns a `DomainObject`.
- `addListener(callback)`: Listen for changes in navigation state. The provided `callback` should be a `Function` which takes a single `DomainObject` as an argument.
- `removeListener(callback)`: Stop listening for changes in navigation state. The provided `callback` should be a `Function` which has previously been passed to `addListener`.

Now

The service `now` is a function which acts as a simple wrapper for `Date.now()`. It is present mainly so that this functionality may be more easily mocked in tests for scripts which use the current time.

Telemetry Formatter

The `telemetryFormatter` is a utility for formatting domain and range values read from a telemetry series.

The `telemetryFormatter` has the following methods:

- `formatDomainValue(value)`: Format the provided domain value (which will be assumed to be a timestamp) for display; returns a string.
- `formatRangeValue(value)`: Format the provided range value (a number) for display; returns a string.

Telemetry Handler

The `telemetryHandler` is a utility for retrieving telemetry data associated with domain objects; it is particularly useful for dealing with cases where the `telemetry` capability is delegated to contained objects (as occurs in Telemetry Panels.)

The `telemetryHandler` has the following methods:

- `handle(domainObject, callback, [lossless])`: Subscribe to and issue future requests for telemetry associated with the provided `domainObject`, invoking the provided `callback` function when streaming data becomes available. Returns a `TelemetryHandle` (see below.)

Telemetry Handle

A `TelemetryHandle` has the following methods:

- `getTelemetryObjects()`: Get the domain objects (as a `DomainObject[]`) that have a `telemetry` capability and are being handled here. Note that these are looked up asynchronously, so this method may return an empty array if the initial lookup is not yet completed.
- `promiseTelemetryObjects()`: As `getTelemetryObjects()`, but returns a `Promise` that will be fulfilled when the lookup is complete.
- `unsubscribe()`: Unsubscribe to streaming telemetry updates associated with this handle.
- `getDomainValue(domainObject)`: Get the most recent domain value received via a streaming update for the specified `domainObject`.
- `getRangeValue(domainObject)`: Get the most recent range value received via a streaming update for the specified `domainObject`.
- `getMetadata()`: Get metadata (as reported by the `getMetadata()` method of a `telemetry` capability) associated with telemetry-providing domain objects. Returns an array, which is in the same order as `getTelemetryObjects()`.
- `request(request, callback)`: Issue a new `request` for historical telemetry data. The provided `callback` will be invoked when new data becomes available, which may occur multiple times (e.g. if there are multiple domain objects.) It will be invoked with the `DomainObject` for which a new series is available, and the `TelemetrySeries` itself, in that order.
- `getSeries(domainObject)`: Get the latest `TelemetrySeries` (as resulted from a previous `request(...)` call) available for this domain object.

Models

Domain object models in Open MCT Web are JavaScript objects describing the persistent state of the domain objects they describe. Their contents include a mix of commonly understood metadata attributes; attributes which are recognized by and/or determine the applicability of specific extensions; and properties specific to given types.

General Metadata

Some properties of domain object models have a ubiquitous meaning through Open MCT Web and can be utilized directly:

- `name`: The human-readable name of the domain object.

Extension-specific Properties

Other properties of domain object models have specific meaning imposed by other extensions within the Open MCT Web platform.

Capability-specific Properties

Some properties either trigger the presence/absence of certain capabilities, or are managed by specific capabilities:

- `composition`: An array of domain object identifiers that represents the contents of this domain object (e.g. as will appear in the tree hierarchy.) Understood by the `composition` capability; the presence or absence of this property determines the presence or absence of that capability.
- `modified`: The timestamp (in milliseconds since the UNIX epoch) of the last modification made to this domain object. Managed by the `mutation` capability.
- `persisted`: The timestamp (in milliseconds since the UNIX epoch) of the last time when changes to this domain object were persisted. Managed by the `persistence` capability.
- `relationships`: An object containing key-value pairs, where keys are symbolic identifiers for relationship types, and values are arrays of domain object identifiers. Used by the `relationship` capability; the presence or absence of this property determines the presence or absence of that capability.
- `telemetry`: An object which serves as a template for telemetry requests associated with this domain object (e.g. specifying `source` and `key`; see Telemetry Requests

under Core API.) Used by the `telemetry` capability; the presence or absence of this property determines the presence or absence of that capability.
- `type`: A string identifying the type of this domain object. Used by the `type` capability.

View Configurations

Persistent configurations for specific views of domain objects are stored in the domain object model under the property `configurations`. This is an object containing key-value pairs, where keys identify the view, and values are objects containing view-specific (and view-managed) configuration properties.

Modifying Models

When interacting with a domain object's model, it is possible to make modifications to it directly. **Don't!** These changes may not be properly detected by the platform, meaning that other representations of the domain object may not be updated, changes may not be saved at the expected times, and generally, that unexpected behavior may occur.

Instead, use the `mutation` capability.

Capabilities

Dynamic behavior associated with a domain object is expressed as capabilities. A capability is a JavaScript object with an interface that is specific to the type of capability in use.

Often, there is a relationship between capabilities and services. For instance, there is an `action` capability and an `actionService`, and there is a `telemetry` capability as well as a `telemetryService`. Typically, the pattern here is that the capability will utilize the service *for the specific domain object*.

When interacting with domain objects, it is generally preferable to use a capability instead of a service when the option is available. Capability interfaces are typically easier to use and/or more powerful in these situations. Additionally, this usage provides a more robust substitutability mechanism; for instance, one could configure a plugin such that it provided a totally new implementation of a given capability which might not invoke the underlying service, while user code which interacts with capabilities remains indifferent to this detail.

Action

The `action` capability is present for all domain objects. It allows applicable `Action` instances to be retrieved and performed for specific domain objects.

For example:

```
domainObject.getCapability("action").perform("navigate");
```

...will initiate a navigate action upon the domain object, if an action with key "navigate" is defined.

This capability has the following interface:

- `getActions(context)`: Get the actions that are applicable in the specified action `context`; the capability will fill in the `domainObject` field of this context if necessary. If `context` is specified as a string, they will instead be used as the `key` of the action context. Returns an array of `Action` instances.
- `perform(context)`: Perform an action. This will find and perform the first matching action available for the specified action `context`, filling in the `domainObject` field as necessary. If `context` is specified as a string, they will instead be used as the `key` of the action context. Returns a `Promise` for the result of the action that was performed, or undefined if no matching action was found.

Composition

The `composition` capability provides access to domain objects that are contained by this domain object. While the `composition` property of a domain object's model describes these contents (by their identifiers), the `composition` capability provides a means to load the corresponding `DomainObject` instances in the same order. The absence of this property in the model will result in the absence of this capability in the domain object.

This capability has the following interface:

- `invoke()`: Returns a `Promise` for an array of `DomainObject` instances.

Delegation

The `delegation` capability is used to communicate the intent of a domain object to delegate responsibilities, which would normally handled by other capabilities, to the domain objects in its composition.

This capability has the following interface:

- `getDelegates(key)`: Returns a `Promise` for an array of `DomainObject` instances, to which this domain object wishes to delegate the capability with the specified `key`.
- `invoke(key)`: Alias of `getDelegates(key)`.
- `doesDelegate(key)`: Returns `true` if the domain object does delegate the capability with the specified `key`.

The platform implementation of the `delegation` capability inspects the domain object's type definition for a property `delegates`, whose value is an array of strings describing which capabilities domain objects of that type wish to delegate. If this property is not present, the `delegation` capability will not be present in domain objects of that type.

Editor

The `editor` capability is meant primarily for internal use by Edit mode, and helps to manage the behavior associated with exiting Edit mode via Save or Cancel. Its interface is not intended for general use. However, `domainObject.hasCapability('editor')` is a useful way of determining whether or not we are looking at an object in Edit mode.

Mutation

The `mutation` capability provides a means by which the contents of a domain object's model can be modified. This capability is provided by the platform for all domain objects, and has the following interface:

- `mutate(mutator, [timestamp])`: Modify the domain object's model using the specified `mutator` function. After changes are made, the `modified` property of the model will be updated with the specified `timestamp`, if one was provided, or with the current system time.
- `invoke(...)`: Alias of `mutate`.

Changes to domain object models should only be made via the `mutation` capability; other platform behavior is likely to break (either by exhibiting undesired behavior, or failing to exhibit desired behavior) if models are modified by other means.

Mutator Function

The `mutator` argument above is a function which will receive a cloned copy of the domain object's model as a single argument. It may return:

- A `Promise`, in which case the resolved value of the promise will be used to determine which of the following forms is used.
- Boolean `false`, in which case the mutation is cancelled.
- A JavaScript object, in which case this object will be used as the new model for this domain object.

- No value (or, equivalently, `undefined`), in which case the cloned copy (including any changes made in place by the mutator function) will be used as the new domain object model.

Persistence

The `persistence` capability provides a mean for interacting with the underlying persistence service which stores this domain object's model. It has the following interface:

- `persist()`: Store the local version of this domain object, including any changes, to the persistence store. Returns a `Promise` for a boolean value, which will be true when the object was successfully persisted.
- `refresh()`: Replace this domain object's model with the most recent version from persistence. Returns a `Promise` which will resolve when the change has completed.
- `getSpace()`: Return the string which identifies the persistence space which stores this domain object.

Relationship

The `relationship` capability provides a means for accessing other domain objects with which this domain object has some typed relationship. It has the following interface:

- `listRelationships()`: List all types of relationships exposed by this object. Returns an array of strings identifying the types of relationships.
- `getRelatedObjects(relationship)`: Get all domain objects to which this domain object has the specified type of `relationship`, which is a string identifier (as above.) Returns a `Promise` for an array of `DomainObject` instances.

The platform implementation of the `relationship` capability is present for domain objects which has a `relationships` property in their model, whose value is an object containing key-value pairs, where keys are strings identifying relationship types, and values are arrays of domain object identifiers.

Telemetry

The `telemetry` capability provides a means for accessing telemetry data associated with a domain object. It has the following interface:

- `requestData([request])`: Request telemetry data for this specific domain object, using telemetry request parameters from the specified `request` if provided. This capability will fill in telemetry request properties as-needed for this domain object. Returns a `Promise` for a `TelemetrySeries`.
- `subscribe(callback, [request])`: Subscribe to telemetry data updates for this specific domain object, using telemetry request parameters from the specified `request` if provided. This capability will fill in telemetry request properties as-needed for this domain object. The specified `callback` will be invoked with `TelemetrySeries` instances as they arrive. Returns a function which can be invoked to terminate the subscription, or `undefined` if no subscription could be obtained.
- `getMetadata()`: Get metadata associated with this domain object's telemetry.

The platform implementation of the `telemetry` capability is present for domain objects which has a `telemetry` property in their model and/or type definition; this object will serve as a template for telemetry requests made using this object, and will also be returned by `getMetadata()` above.

Type

The `type` capability exposes information about the domain object's type. It has the same interface as `Type`; see Core API.

View

The `view` capability exposes views which are applicable to a given domain object. It has the following interface:

- `invoke()`: Returns an array of extension definitions for views which are applicable for this domain object.

Actions

Actions are reusable processes/behaviors performed by users within the system, typically upon domain objects.

Action Categories

The platform understands the following action categories (specifiable as the `category` parameter of an action's extension definition.)

- `contextual`: Appears in context menus.
- `view-control`: Appears in top-right area of view (as buttons) in Browse mode

Platform Actions

The platform defines certain actions which can be utilized by way of a domain object's `action` capability. Unless otherwise specified, these act upon (and modify) the object described by the `domainObject` property of the action's context.

- `cancel`: Cancel the current editing action (invoked from Edit mode.)
- `compose`: Place an object in another object's composition. The object to be added should be provided as the `selectedObject` of the action context.
- `edit`: Start editing an object (enter Edit mode.)
- `fullscreen`: Enter full screen mode.
- `navigate`: Make this object the focus of navigation (e.g. highlight it within the tree, display a view of it to the right.)
- `properties`: Show the "Edit Properties" dialog.
- `remove`: Remove this domain object from its parent's composition. (The parent, in this case, is whichever other domain object exposed this object by way of its `composition` capability.)
- `save`: Save changes (invoked from Edit mode.)
- `window`: Open this object in a new window.

Policies

Policies are consulted to determine when certain behavior in Open MCT Web is allowed. Policy questions are assigned to certain categories, which broadly describe the type of decision being made; within each category, policies have a candidate (the thing which may or may not be allowed) and, optionally, a context (describing, generally, the context in which the decision is occurring.)

The types of objects passed for “candidate” and “context” vary by category; these types are documented below.

Policy Categories

The platform understands the following policy categories (specifiable as the `category` parameter of an policy’s extension definition.)

- `action`: Determines whether or not a given action is allowable. The candidate argument here is an `Action`; the context is its action context object.
- `composition`: Determines whether or not domain objects of a given type are allowed to contain domain objects of another type. The candidate argument here is the container’s `Type`; the context argument is the `Type` of the object to be contained.
- `view`: Determines whether or not a view is applicable for a domain object. The candidate argument is the view’s extension definition; the context argument is the `DomainObject` to be viewed.

Build, Test, Deploy

Open MCT Web is designed to support a broad variety of build and deployment options. The sources can be deployed in the same directory structure used during development. A few utilities are included to support development processes.

Command-line Build

Open MCT Web includes a script for building via command line using Maven 3.0.4 (https://maven.apache.org/).

Invoking `mvn clean install` will:

- Check code style using JSLint. The build will fail if JSLint raises any warnings.
- Run the test suite (see below.) The build will fail if any tests fail.
- Populate version info (e.g. commit hash, build time.)
- Produce a web archive (`.war`) artifact in the `target` directory.

The produced artifact contains a subset of the repository's own folder hierarchy, omitting tests and example bundles.

Note that an internet connection is required to run this build, in order to download build dependencies.

Test Suite

Open MCT Web uses Jasmine (http://jasmine.github.io/) for automated testing. The file `test.html`, included at the top level of the source repository, can be run from the browser to perform tests for all active bundles, as defined in `bundle.json`.

To define tests for a bundle:

- Include a directory named `test` within that bundle.
- In the `test` directory, include a file named `suite.json`. This will identify which scripts will be tested.
- The file `suite.json` must contain a JSON array of strings, where each string is the name of a script to be tested. These names should include any directory paths to the script after (but not including) the `src` folder, and should not include the file's `.js` extension. (Note that while Open MCT Web's framework allows a different name to be chosen for the `src` directory, the test runner does not: This directory must be named `src` for the test runner to find it.)

- For each script to be tested, a corresponding test script should be located in the bundle's `test` directory. This should include the suffix `Spec` at the end of the filename (but before the `.js` extension.) This test script should be an AMD module which uses the Jasmine API to declare its test behavior. It should declare an AMD dependency on the script to be tested, using a relative path.

  For example, if writing tests for a bundle at `example/foo` with two scripts:
- `example/foo/src/controllers/FooController.js`
- `example/foo/src/directives/FooDirective.js`

First, these scripts should be identified in `example/foo/test/suite.json`, e.g. with contents:

```
[ "controllers/FooController", "directives/FooDirective" ]
```

Then, scripts which describe these tests should be written. For example, test `example/foo/test/controllers/FooControllerSpec.js` could look like:

```
/*global define,Promise,describe,it,expect,beforeEach*/

define(
    ["../../src/controllers/FooController"],
    function (FooController) {
        "use strict";

        describe("The foo controller", function () {
            it("does something", function () {
                var controller = new FooController();
                expect(controller.foo()).toEqual("foo");
            });
        });
    }
);
```

Code Coverage

In addition to running tests, the test runner will also capture code coverage information using Blanket.JS (http://blanketjs.org/) and display this at the bottom of the screen. Currently, only statement coverage is displayed.

Deployment

Open MCT Web is built to be flexible in terms of the deployment strategies it supports. In order to run in the browser, Open MCT Web needs:

1. HTTP access to sources/resources for the framework, platform, and all active bundles.
2. Access to any external services utilized by active bundles. (This means that external services need to support HTTP or some other web-accessible interface, like WebSockets.)

Any HTTP server capable of serving flat files is sufficient for the first point. The command-line build also packages Open MCT Web into a `.war` file for easier deployment on containers such as Apache Tomcat.

The second point may be less flexible, as it depends upon the specific services to be utilized by Open MCT Web. Because of this, it is often the set of external services (and the manner in which they are exposed) that determine how to deploy Open MCT Web.

One important constraint to consider in this context is the browser's same origin policy. If external services are not on the same apparent host and port as the client (from the perspective of the browser) then access may be disallowed. There are two workarounds if this occurs:

- Make the external service appear to be on the same host/port, either by actually deploying it there, or by proxying requests to it.
- Enable CORS (cross-origin resource sharing) on the external service. This is only possible if the external service can be configured to support CORS. Care should be exercised if choosing this option to ensure that the chosen configuration does not create a security vulnerability.

Examples of deployment strategies (and the conditions under which they make the most sense) include:

- If the external services that Open MCT Web will utilize are all running on Apache Tomcat (https://tomcat.apache.org/), then it makes sense to run Open MCT Web from the same Tomcat instance as a separate web application. The `.war` artifact produced by the command line build facilitates this deployment option. (See https://tomcat.apache.org/tomcat-8.0-doc/deployer-howto.html for general information on deploying in Tomcat.)
- If a variety of external services will be running from a variety of hosts/ports, then it may make sense to use a web server that supports proxying, such as the Apache HTTP Server (http://httpd.apache.org/). In this configuration, the HTTP server would be configured to proxy (or reverse proxy) requests at specific paths to the various external services, while providing Open MCT Web as flat files from a different path.

- If a single server component is being developed to handle all server-side needs of an Open MCT Web instance, it can make sense to serve Open MCT Web (as flat files) from the same component using an embedded HTTP server such as Nancy (http://nancyfx.org/).
- If no external services are needed (or if the “external services” will just be generating flat files to read) it makes sense to utilize a lightweight flat file HTTP server such as Lighttpd (http://www.lighttpd.net/). In this configuration, Open MCT Web sources/resources would be placed at one path, while the files generated by the external service are placed at another path.
- If all external services support CORS, it may make sense to have an HTTP server that is solely responsible for making Open MCT Web sources/resources available, and to have Open MCT Web contact these external services directly. Again, lightweight HTTP servers such as Lighttpd (http://www.lighttpd.net/) are useful in this circumstance. The downside of this option is that additional configuration effort is required, both to enable CORS on the external services, and to ensure that Open MCT Web can correctly locate these services.

Another important consideration is authentication. By design, Open MCT Web does not handle user authentication. Instead, this should typically be treated as a deployment-time concern, where authentication is handled by the HTTP server which provides Open MCT Web, or an external access management system.

Configuration

In most of the deployment options above, some level of configuration is likely to be needed or desirable to make sure that bundles can reach the external services they need to reach. Most commonly this means providing the path or URL to an external service.

Configurable parameters within Open MCT Web are specified via constants (literally, as extensions of the `constants` category) and accessed via dependency injection by the scripts which need them. Reasonable defaults for these constants are provided in the bundle where they are used. Plugins are encouraged to follow the same pattern.

Constants may be specified in any bundle; if multiple constants are specified with the same `key`, the highest-priority one will be used. This allows default values to be overridden by specifying constants with higher priority.

This permits at least three configuration approaches:

- Modify the constants defined in their original bundles when deploying. This is generally undesirable due to the amount of manual work required and potential for error, but is viable if there are a small number of constants to change.
- Add a separate configuration bundle which overrides the values of these constants. This is particularly appropriate when multiple configurations (e.g. development, test,

production) need to be managed easily; these can be swapped quickly by changing the set of active bundles in `bundles.json`.
- Deploy Open MCT Web and its external services in such a fashion that the default paths to reach external services are all correct.

Configuration Constants

The following configuration constants are recognized by Open MCT Web bundles:

- CouchDB adapter, `platform/persistence/coucb`
  - `COUCHDB_PATH`: URL or path to the CouchDB database to be used for domain object persistence. Should not include a trailing slash.
- ElasticSearch adapter, `platform/persistence/elastic`
  - `ELASTIC_ROOT`: URL or path to the ElasticSearch instance to be used for domain object persistence. Should not include a trailing slash.
  - `ELASTIC_PATH`: Path relative to the ElasticSearch instance where domain object models should be persisted. Should take the form `<index>/<type>`.