opentext™

D2 Smartview SDK

version: 23.4.0

opentext™

Table of contents:

- Documentum D2 Smartview SDK
  - How to prepare and start with the development environment
    - 1. Download the developer tools for your OS:
    - 2. Install the developer tools for your OS
    - 3. Create the development workspace:
    - 4. Get familiar with Workspace assistant
    - 5. Create a plugin project:
    - 6. Start coding
- Architecture
  - Technology
- Setup SDK workspace in IDE
    - Preface
  - Steps to setup workspace in IntelliJ IDE
  - Steps to setup workspace in Eclipse IDE
- Debugging D2 Smartview UI
  - How to setup?
- Extending/Overriding D2FS service through Service Plugin
- Custom Widget Type
  - Custom shortcut type in D2
- Delta Menus in D2
- Icons in D2 Smartview
  - Specification
  - How to use an icon
  - Sample non-interactive icon
  - Sample action icon
- Upgrading Documentum D2 Smartview SDK to the latest version
- Workspace & Assistant
  - What is a workspace?
  - What is the Assistant?
- Creating a plugin
- Add Smartview application scope perspective
- Add smartview UI support to an existing plugin project

- Remove a plugin from workspace
- Add D2-REST controller support to a plugin project
- Build all plugins in the workspace
- Checkout documentation
- Checkout samples
- Add smartview shortcut behavior
- Add smartview list tile
- Add smartview shortcut tile
- Add D2FS dialog to a plugin
- Add new metadata view to plugin
- Add new task details view to plugin
- Packaged Samples
  - List of samples
- D2 Admin-Groups Sample
  - Instruction to try out the sample
  - Source code structure
- D2SV Read-Only Permission View Sample
  - Instruction to try out the sample
  - Source code structure
  - Files and their purpose
  - REST Implementations
  - View permission menu configuration in back-end and its display & execution on the front-end
  - The side-panel dialog that displays permissions
- D2SV Custom Widget Type Tile
  - Instruction to try out the sample
  - Source code structure
  - Files and their purpose
  - REST Implementations
  - Custom widget type references needed for the D2-Config widget configuration and D2 Smartview landing page configuration
  - D2 Smartview UI changes for the plugin
- D2SV client to server logging
  - Instruction to try out the sample
  - Source code structure

opentext™

- D2SV Custom Dialogs(D2FS) sample
  - Instruction to try out the sample
  - Source code structure
- Custom Table Cell View sample
  - Instruction to try out the sample
  - Source code structure
- Action icons catalog
  - D2 Smartview icons
  - CSUI icons
  - SVF icons
- D2FS REST services developer guide
- Understanding D2SV plugin project
  - Plugin project layout
  - Files and their purpose
- Where to start?
- Overview
- Context
  - Factory
  - Fetchable Factory
  - Configurable Factory
  - Detached Objects
  - Permanent Objects
  - Temporary Objects
  - Factory Life-Cycle
  - Methods
    - getObject(factory, options): object
    - getCollection(factory, options): object
    - getModel(factory, options): object
    - hasObject(factory, options): boolean
    - hasCollection(factory, options): boolean
    - hasModel(factory, options): boolean
    - clear(options): void
    - fetch(options): Promise
    - suppressFetch(): boolean

- Properties
  - fetching: Promise
  - fetched: boolean
  - error: Error
- Events
  - 'before:clear', context
  - 'clear', context
  - 'request', context
  - 'sync', context
  - 'error', error, context
  - 'add:factory', context, propertyName, factory
  - 'remove:factory', context, propertyName, factory
- Context Fragment
  - Details
  - Methods
    - constructor(context)
    - fetch(options): Promise
    - clear(): void
    - destroy(): void
  - Properties
    - fetching: Promise?
    - fetched: boolean
    - error: Error
  - Events
    - 'request', context
    - 'sync', context
    - 'error', error, context
    - 'add:factory', context, propertyName, factory
    - 'before:clear', context
    - 'clear', context
    - 'destroy', context
- PageContext
  - Plugins
- SynchronizedContext

opentext™

- I18n
  - Accept-Language in AJAX Calls
- RequireJS
  - Changes
  - pkgs mergeable
  - Attribute data-csui-required
  - rename map
  - Merged module configuration
  - moduleConfig method
- Browsable Support for Backbone.Collections
  - Browsing Support Modules
  - Examples
  - Remarks
- BrowsableMixin
    - Example
  - makeBrowsable(options) : this
  - Browsing State Properties
  - Browsing State Changing Methods
- Browsing Methods
  - fetch(options) : promise
  - See Also
- ClientSideBrowsableMixin
    - Example
  - makeClientSideBrowsable(options) : this
  - fetch(options) : promise
  - populate(models, options) : promise
  - add(models, options) : models
  - remove(models, options) : models
  - reset(models, options) : models
  - set(models, options) : models
  - compareObjectsForSort(property, left, right) : -1|0|1
  - See Also
- BrowsableV1RequestMixin
    - Example

    - makeBrowsableV1Request(options) : this
    - getBrowsableUrlQuery(options) : object literal or string
    - See Also
- BrowsableV1ResponseMixin
    - Example
    - makeBrowsableV1Response(options) : this
    - parseBrowsedState(response, options) : nothing
    - parseBrowsedItems(response, options) : array of object literals
    - See Also
- BrowsableV2ResponseMixin
    - Example
    - makeBrowsableV2Response(options) : this
    - parseBrowsedState(response, options) : nothing
    - parseBrowsedItems(response, options) : array of object literals
    - See Also
- AutoFetchableMixin
    - How to apply the mixin to a model
    - How use the mixin
    - makeAutoFetchable(options) : this
    - automateFetch(boolean) : void
    - isFetchable() : boolean
    - See Also
- CommandableMixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeCommandable(options) : this
    - commands
    - setCommands(names) : void
    - resetCommands(names) : void
    - getRequestedCommandsUrlQuery() : string
    - See Also
- ConnectableMixin
    - Remarks
        - How to apply the mixin to a model

    - How use the mixin
  - makeConnectable(options) : this
  - connector
  - See Also
- DelayedCommandableMixin
  - How to apply the mixin to a model
  - How to use the mixin
  - makeDelayedCommandable(options) : this
  - commands
  - defaultActionCommands
  - delayRestCommands
  - delayRestCommandsForModels
  - delayedActions
  - setCommands(names) : void
  - resetCommands(names) : void
  - setDefaultActionCommands(names) : void
  - resetDefaultActionCommands(names) : void
  - getRequestedCommandsUrlQuery() : string
  - See Also
- ExpandableMixin
  - How to apply the mixin to a model
  - How to use the mixin
  - makeExpandable(options) : this
  - expand
  - setExpand(name) : void
  - resetExpand(name) : void
  - getExpandableResourcesUrlQuery() : string
  - See Also
- FetchableMixin
  - Remarks
    - How to apply the mixin to a collection
    - How use the mixin
  - makeFetchable(options) : this
  - fetching

    - fetched
    - error
    - ensureFetched(options) : promise
    - invalidateFetch
    - See Also
- IncludingAdditionalResourcesMixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeIncludingAdditionalResources(options) : this
    - _additionalResources
    - includeResources(names) : void
    - excludeResources(names) : void
    - getAdditionalResourcesUrlQuery() : string
    - See Also
- Mixins for Models
- NodeAutoFetchableMixin
    - How to apply the mixin to a model
    - How use the mixin
    - makeNodeAutoFetchable(options) : this
    - automateFetch(boolean) : void
    - isFetchable() : boolean
    - See Also
- NodeConnectableMixin
    - Remarks
        - How to apply the mixin to a model
        - How use the mixin
    - makeNodeConnectable(options) : this
    - node
    - connector
    - See Also
- NodeResourceMixin
    - How to apply the mixin to a collection
    - Remarks
        - How use the mixin

    - makeNodeResource(options) : this
    - See Also
- ResourceMixin
    - How to apply the mixin to a model
    - Remarks
        - How use the mixin
    - makeResource(options) : this
    - See Also
- RulesMatchingMixin
    - How to apply the mixin to a model
    - How use the mixin
    - makeRulesMatching(options) : this
    - Conditions
    - Properties
- StateCarrierMixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeStateCarrier(options) : this
    - state : Backbone.Model
    - parseState(response, role) : void
    - See Also
- StateRequestorMixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeFieldsV2(options) : this
    - stateEnabled : boolean
    - enableState() : void
    - disableState() : void
    - getStateEnablingUrlQuery() : object
    - See Also
- SyncableFromMultipleSourcesMixin
    - How to apply the mixin to a model
    - How to combine multiple $.ajax calls
    - How to combine multiple model/collection fetches

    - makeSyncableFromMultipleSources(options) : this
    - syncFromMultipleSources(promises, mergeSources, convertError, options) : promise
- UploadableMixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeUploadable(options) : this
    - prepare(data, options) : object
    - See Also
    - Motivation
        - PATCH Semantics Support
        - File Upload Support
- AdditionalResourcesV2Mixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeAdditionalResourcesV2Mixin(options) : this
    - _additionalResources
    - includeResources(names) : void
    - excludeResources(names) : void
    - getAdditionalResourcesUrlQuery() : string
    - See Also
- CommandableMixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeCommandable(options) : this
    - commands
    - setCommands(names) : void
    - resetCommands(names) : void
    - getRequestedCommandsUrlQuery() : string
    - See Also
- DelayedCommandableV2Mixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeDelayedCommandableV2(options) : this
    - commands

    - defaultActionCommands
    - delayRestCommands
    - delayRestCommandsForModels
    - promoteSomeRestCommands
    - delayedActions
    - setCommands(names) : void
    - resetCommands(names) : void
    - setDefaultActionCommands(names) : void
    - resetDefaultActionCommands(names) : void
    - getRequestedCommandsUrlQuery() : string
    - setEnabledDelayRestCommands(enabled, promoted) : void
    - See Also
- ExpandableV2Mixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeExpandableV2(options) : this
    - expand
    - hasExpand(role) : boolean
    - setExpand(role, names) : void
    - resetExpand(role, names) : void
    - getExpandableResourcesUrlQuery() : string
    - See Also
- FieldsV2Mixin
    - How to apply the mixin to a model
    - How to use the mixin
    - makeFieldsV2(options) : this
    - fields
    - hasFields(role) : boolean
    - setFields(role, names) : void
    - resetFields(role, names) : void
    - getResourceFieldsUrlQuery() : string
    - See Also
- NodeAddableTypeCollection
    - AddableTypeModel Attributes

    - Examples
        - See Also
- NodeModel
    - Examples
- NodeChildren2Collection
    - Examples
    - See Also
- NodeColumn2Collection
- ToolItemMaskCollection
- Authenticators
    - Common Methods
        - unauthenticate(options) : void
    - Base Classes
        - Authenticator
        - RequestAuthenticator
- Final Classes
    - BasicAuthenticator
    - CredentialsAuthenticator
    - InitialHeaderAuthenticator
    - InteractiveCredentialsAuthenticator
    - RegularHeaderAuthenticator
    - TicketAuthenticator
- Base
    - autoAlignDropDowns(inputElement, dropdownContainer, applyWidth, view, callback)
- Module nuc/util/connector
    - Connector
        - Constructor
        - makeAjaxCall(options) : promise
- Load extensions
- Log
    - Synopsis
    - Usage
    - Configuration Parameters
    - Configuration Updates

    - Beyond Console
- PageLeavingBlocker
    - Examples
    - Methods
        - isEnabled() : boolean
        - enabled(message) : void
        - disable() : void
- Date Parsing and Formatting
- Localizable Strings
    - getClosestLocalizedString(value, fallback) : string
        - Parameters
        - Result
        - Example
    - locale*String (...) : ...
        - localeCompareString(left, right, options): -1|0|1
        - localeContainsString(hay, needle): boolean
        - localeEndsWithString(full, end): boolean
        - localeIndexOfString(hay, needle): number
        - localeStartsWithString(full, start): boolean
    - formatMessage (count, messages, ...) : string
        - Example
- Member Name Formatting (nuc/utils/types/member)
- Number Formatting (nuc/utils/types/number)

opentext™

Documentum D2 Smartview SDK

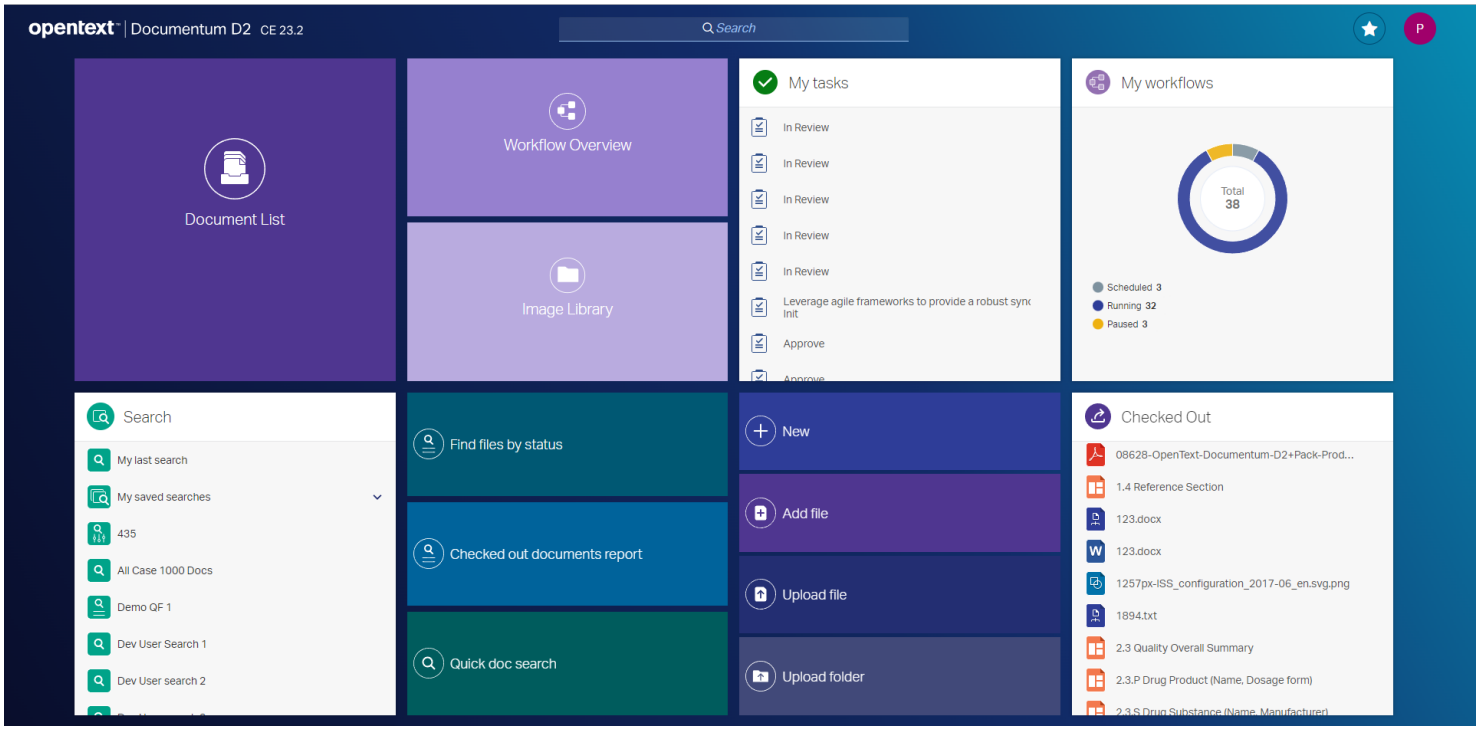
The D2 Smart View SDK consists of sources, binaries, documentation, and samples for -

- D2 Smartview UI extension enviornment.
- D2-REST services extension enviornment.
- D2 plugin development enviornment.

It also includes a few tools to create and maintain a development workspace.

With the D2 Smart View SDK you can build enterprise-ready software components for Documentum D2 Smartview runtime to cater custom business needs.

Out of the box, D2 Smart View landing page looks like:



How to prepare and start with the development environment

1. Download developer tools
2. Install developer tools

3. Create the development workspace
4. Get familiar with SDK tools
5. Create a plugin project
6. Start coding

1. Download the developer tools for your OS:

```
JDK         - JDK is required to compile Java code present within a development
workspace.
              Use JDK 1.8 or later.
              See https://openjdk.java.net
Maven       - Apache maven is the secondary build tool used in this SDK development
workspace.
              Recommended version is 3.8.2. A different version may not be fully
compatible.
              See https://maven.apache.org
NodeJS      - JavaScript VM to execute the SDK tools, build tools and to run the
development web server for UI code.
              Recommended version is 16 LTS. A different version may not be fully
compatible.
              See http://nodejs.org.
Grunt       - JavaScript task runner for building and testing UI code.
              See http://gruntjs.com. Nothing to be downloaded from this URL though.
```

2. Install the developer tools for your OS

```
JDK         - Run installer. Set the JAVA_HOME path variable to point to the JDK root
directory.
Maven       - Unzip & extract to a directory. Set MAVEN_HOME environment variable
pointing to the directory.
              Update PATH variable accordingly so that Maven commands can be executed
from command-line/terminal.
NodeJS      - Install the package for your OS. Set NPM_HOME path variable pointing to
the NodeJS
              installation directory. Update PATH variable so that Node & NPM commands
can be executed from
              command-line.
              It is recommended to avoid installing NodeJS under 'Program Files' as
```

```
doing that has been known to create
              problem some times.
NPM         - Update the NPM module management tool to the latest version:
                npm install -g npm@latest

Grunt       - Install the command line task runner client as a global NPM module
                npm install -g grunt-cli
```

3. Create the development workspace:

```
# 1. Extract the SDK
# 2. Open a command prompt at the extracted folder

# Execute batch script ws-init.bat
>ws-init.bat

# It will take a while to fully initialize the workspace.
# Once initialization completes successfully, the workspace assistant starts
automatically. Select "Check out documentation" option to open documentation in default
browser.
#
# The directory where SDK was extracted becomes the root of the development workspace.
# It doesn't require to run ws-init.bat inside the initialized workspace again, unless
some other instructions specifically says to do so.
# If you want to run the workspace assistant anytime later, open a command
prompt/terminal at workspace root directory and run
>npm start

# Select "Nothing", to terminate the workspace assitant, if wanted.

# To access the documentation without the workspace assistant, you can run the following
command in a command prompt/terminal at the workspace root.
>npm run documentation
```

4. Get familiar with Workspace assistant

Check out the Workspace assistant. It's a good idea to familiarize yourself with the general aspects of the SDK, this can be done later though.

opentext™

5. Create a plugin project:

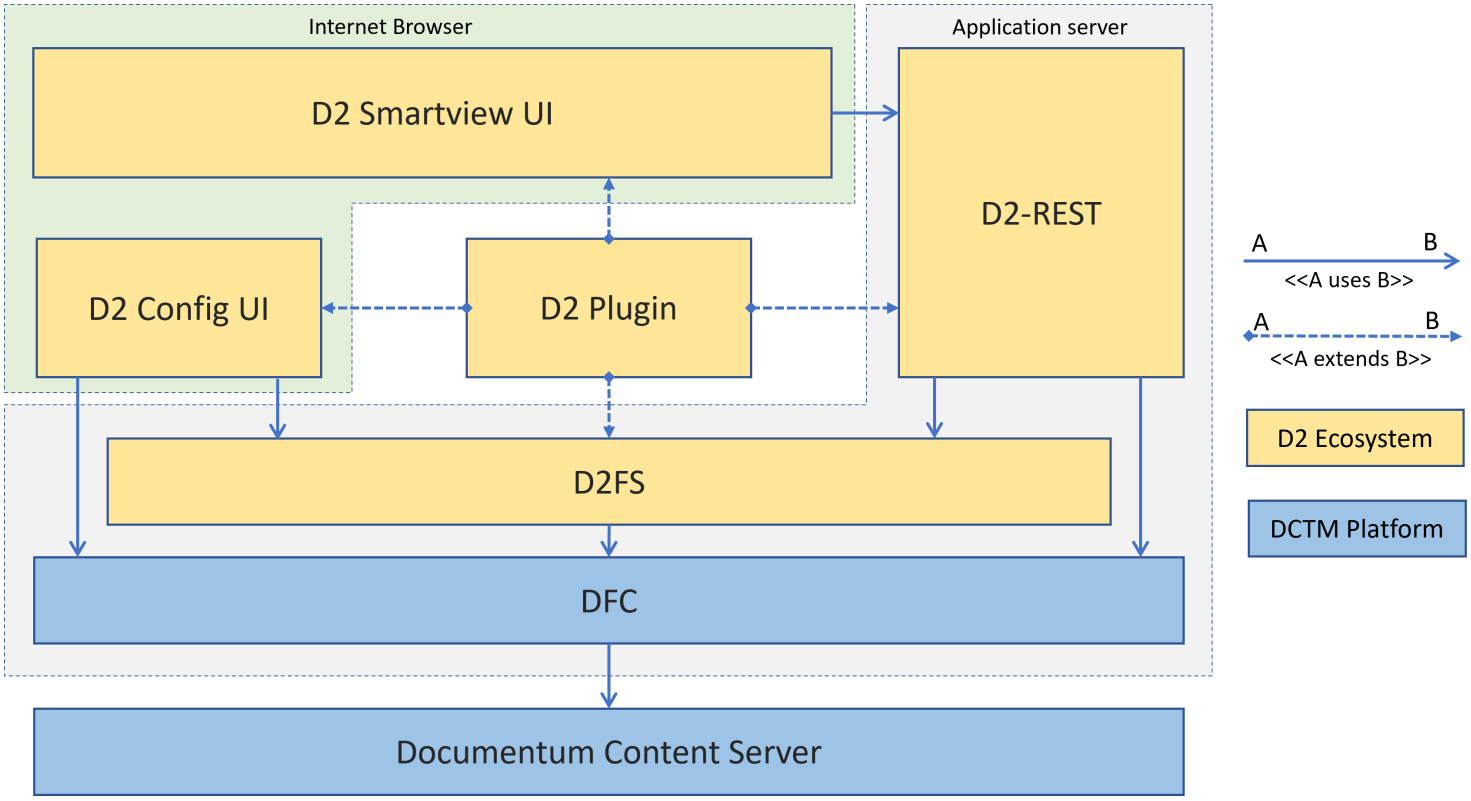
```
# Open command prompt at workspace root and run
>npm start

# Select "Create a new plugin project" from the workspace assistant options.
# Follow on-screen instruction and anser questions to create your first plugin project.
# Once done, type and run-
>npm run build

# Or, alternatively run the workspace assistant again and select "Build all plugins in
this workspace" option.
# This will build all projects in the workspace
```

6. Start coding

Check out the API documentation and start coding as per business requirement.

Architecture

A simplified representation of the D2 Smartview(D2SV) runtime.



A central component of the runtime is D2 Plugins a.k.a D2FS Plugins. A D2 Plugin is loaded dynamically in the D2SV Ecosystem and it can primarily augment functions of D2 Foundation Services(D2FS).

The D2SV SDK API is built around the same D2 Plugins architecture and additionally it can augment functions of D2SV UI & D2FS-REST runtime.

The SDK deals with hybrid Maven + NodeJS project which has both Java & Javascript code along with other static resources organized in a certain structure. Upon build, the SDK compiles and packages the built output into a Jar. This Jar file can then be dropped inside the `lib` folder of a D2 Smartview runtime. The D2SV runtime loads the pluggable components from within the Jar dynamically.

opentext™

Technology

The D2 Smartview is a web application and requires a hybrid middleware runtime. JVM runs the Java written back-end code and an Internet browser's Javascript VM runs the front-end. All the communication between FE & BE happens through AJAX request-response.

- The Java based back-end uses the Spring WebMVC Framework, configured to run in an application container, along with other proprietary and open-source libraries.
- The Javascript front-end uses Backbone & Marionette UI framework along with jQuery, RequireJS, Underscore, Handlebars etc. libraries.

opentext™

Setup SDK workspace in IDE

Preface

All D2SV plugin projects are made of Java & Javascript source codes. Naturally the project setup also has to be hybrid to compile and package all parts of the source code. This is the reason the SDK uses a mixed tooling approach towards the same.

All the plugins inside the workspace, are layed out in a Maven project structure where the `pom.xml` found at the workspace root directory serves as the aggregator-parent and each plugin is linked to it as Maven module.

The NodeJS specific portion does not require a parent-child relationship between the plugins and the workspace. However to optimize dependency management, the workspace declares itself as a NodeJS project through its

`package.json` and each plugin in turn uses directory shortcuts to refer to the same set of dependencies even though the plugin declares a separate NodeJS project for its Javascript code through `package.json` found in its `src/main/smartview` directory.
While building, Maven is used as the primary tool to trigger it. Internally Maven uses `maven-antrun-plugin` to execute NodeJS script through shell and that builds the Javascript portion.

Any IDE that works with Maven projects, can recognize this hybrid setup. All that it takes is the support to be able to import a Maven project from existing source code.

> **TIP**
>
> Before trying any of the following steps to setup IDE, make sure to either create a plugin project or extract a packaged sample in the SDK workspace by running either Create a plugin project or Checkout some samples option from Workspace Assistant.

Steps to setup workspace in IntelliJ IDE

- Select **File -> New -> Project from Existing Sources** from menu.

  Note. If starting with a fresh IDE installation select **Import Project** option from welcome screen.

- In **Select File or Directory to Import** dialog, locate and select `pom.xml` from the workspace root.
- In **Import Project from Maven** dialog, keep the default values and deselect *Search for projects recursively* checkbox if it is selected. Then, click **Next** button to go to next screen.
- In the current screen select the checkbox against a group-id and artifact-id combination that correctly represents the workspace root pom. By default it may look something like *com.opentext.d2.smartview:D2-Plugin-Projects:1.0.0*. After selection, click **Next** button to proceed to the next screen.
- Select an available JDK to use for the imported projects and click **Next** to proceed.
- In current screen keep default values for **Project name** and **Project file location** input fields and click **Finish** to start import.

  Note. It might ask whether to open the project in current window or new window, please select an appropriate option. **New window** is could be a preferred option.

- After the project import completes, it might take a while for the IDE to index files from the workspace. To cut short on the indexing time, it is advised to mark all directories from workspace root except **plugins**(or whichever directory you chose to store plugins) as **Excluded**.

> **INFO**
>
> As you keep adding/removing new plugin projects in the workspace using the workspace assistant, the IDE automatically catches up with the change.

Steps to setup workspace in Eclipse IDE

- Select **File -> Import** from menu.
- In **Import** dialog, expand **Maven** and select **Existing Maven Projects** and click **Next**.
- In **Import Maven Projects** dialog, Click **Browse...** button beside the **Root Directory** field.

- In **Select Root Folder** file-selection dialog, navigate to SDK workspace root directory and click **Select Folder** button to go back to **Import Maven Projects** dialog.
- In **Import Maven Projects** dialog, click **Select All** to select all discovered projects. Then click **Finish** to close the dialog and start importing the projects.
- Once the project import completes, right-click on root project's **pom.xml** and select **Run as -> Maven build** from the menu to open **Edit Configuration** dialog.
- In **Edit Configuration** dialog type `clean install` in **Goals** field then click **Run** button to start building the plugins in the workspace.

> **CAUTION**
>
> Eclipse is unable to automatically detect plugin projects added to/removed from the workspace using the workspace assistant.

To detect newly added project, in Eclipse Project Explorer

- On the root project, **right click -> Refresh**.
- Select the **plugins** folder(or whichever folder you're using to store plugins) and select **Right click -> New -> Project**
- In **New Project** dialog, select **General -> Project** and click **Next**.
- In **New Project** dialog, deselect **Use default location** checkbox and click **Browse**
- In **Select Folder** dialog, navigate inside the root folder of newly created plugin project on disk and click **Select Folder** button
- In **New Project** dialog, copy the last part of path value from **Location** field and paste it into **Project name** field.
- Select the chekcbox **Add project to working sets** and select value **D2-Plugin-Projects** for the field **Working sets** then click **Finish** to close the dialog.
- Once the project creation completes, in **Project Explorer** of Eclipse, expand the **plugins**(or whichever folder you're using) folder and gesture **Right-click -> Configure -> Convert to Maven Project** to finish setting up the new plugin project.

To detect removed project, in Eclipse Project Explorer

- Gesture **Right-click -> Maven -> Update Project** on the root project's **pom.xml**
- In the **Update Maven Project** dialog, deselect every project except the root one and click **OK**
- If the above steps do not automatically remove the plugin project entry, then you can safely **Right-click -> Delete** the project.

Debugging D2 Smartview UI

D2 Smartview UI being written purely in Javascript, HTML, CSS has the benefit of debugging its client-side source code directly from an Internet Browser. Debugging can happen in either mode

1. with the source code as is
2. with compiled & minifed version of the source code

In distribution, D2 Smartview front-end and back-end is packaged as a single web archive however these two parts are very loosely coupled and communicates via HTTP request-responses.

This loosely coupled nature helps keep the front-end and back-end clearly separate even upto an extent where these two parts are hosted and served by two different application servers. This technique is recommended and also used by us to debug D2 Smartview UI such as we setup and configure D2 Smartview UI to make it talk to a running Smartview instance as backend and then host only the UI part on a lightweight NodeJS server.

> **INFO**
>
> The following method only allows debugging client-side Javascript code. If a plugin has some server-side component like REST-Controller, D2FS dialog, D2FS service plugin, menu confguration etc. then the plugin has to be built and deployed on the D2 Smartview application server followed by a server restart in order to make them available to use for the corresponding client-side code.

How to setup?

For D2SV plugin developers, the heavy lifting is already done for you. All you need is to -

1. Open to edit the `server.conf.js` from `src/main/smartview` directory of your plugin.
2. For `D2SV_APP_SERVER_URL` property, set an appripriate URL to a running instance of D2 Smartview

E.g. http://my.domain.com:port/D2-Smartview

> **TIP**
>
> The D2-Smartview installation, which is being referred to by the URL, must be setup to produce either relative linkrels by setting `rest.use.relative.url=true` in its rest-api-runtime.properties file, or alternatively it should be configured to allow Cross-Origin requests by setting other appropriate properties(refer to rest-api-runtime.properties.template file from D2-Smartview distribution).

3. Save `server.conf.js`
4. Open a Terminal/Command Prompt in the same `src/main/smartview` directory of the plugin.
5. Execute command `npm start`
6. Navigate to URL `http://localhost:6989/ui/pages/debug/app.html` in a browser to start debugging code in as-is format.

   Or, alternatively navigate to `http://localhost:6989/ui/pages/release/app.html` to debug the compiled & minified code. Please remember that command `grunt compile` from `src/main/smartview` or `npm run build` from workspace root directory has to be executed before you can debug the compiled and minified code.

Extending/Overriding D2FS service through Service Plugin

If developer wants to create a customization which needs to override/extend the existing functionality of a D2FS service, the developer can create custom class with the **"(D2FS Services Name)Plugin.java'** which extends the D2FS service class and implements **ID2fsPlugin** class. For an example -

```
package com.opentext.d2.smartview.d2svdialogs.webfs.dialog;

import com.emc.d2fs.dctm.web.services.ID2fsPlugin;
import com.emc.d2fs.dctm.web.services.dialog.D2DialogService;
import com.emc.d2fs.models.context.Context;
import com.emc.d2fs.models.attribute.Attribute;
import com.emc.d2fs.models.dialog.Dialog;
import java.util.List;

public class D2DialogServicePlugin extends D2DialogService implements ID2fsPlugin {
    ...
    public Dialog validDialog(Context context, String id, String dialogName,
List<Attribute> parameters) throws Exception {
        //custom logic

        //If the following line is executed during an invocation then it becomes an
extension, otherwise it becomes an override.
        return super.validDialog(context, id, dialogName, parameters);
    }
    ...
}
```

Following services can only be extended/overriden for the supported methods. Below table lists both overidable and non-overridable methods for D2-Smartview

Services	Overridable Methods	Non-Overridable Methods

opentext™

Services	Overridable Methods	Non-Overridable Methods
D2CreationService	applyVdTemplate getVDTemplates setTemplate getTemplates getTemplates updateTemplatesListwithFilter getConvertStructureConfig createTemplateFromServer createProperties getRecentlyUsedVDTemplates getImportStructureConfigs getRecentlyUsedTemplates	hasAnyAttachments removeAttachments getUIMaxSize hasAttachments hasAttachments getFilteredTemplates isAFolderOrACabinet getTemplateFilterOptions isNoCreationProfile isNoContentAuthorized
D2DialogService	getOptions getDialog validDialog cancelDialog	getTaxonomy getLabels isMemberOfGroup getImportValuesUrl getSubforms getExportValuesUrl
D2PropertyService	dump saveProperties saveProperties	getProperties
D2WorkflowService	rerunAutoActivity resumeTask pauseTask setTaskPriority updateWorkflowSupervisor getWorkflowTemplatesByWidgetName removeWorkflowSupportingDocuments	isTaskAcquired getTaskMode acquireTask getTaskPermissions canRejectTask canForwardTask canDelegateTask

opentext™

Services	Overridable Methods	Non-Overridable Methods
	isTaskQueueItemRead addWorkflowSupportingDocuments getWorkflowUsersByWidgetName getWorkflowWorkingDocumentsCount getWorkflowSupportingDocumentsCount completeAutoActivity getWorkflowAttachments checkLifeCycle pauseWorkflow processTask addNoteToTask launchWorkflow abortWorkflow canAddTaskNote delegateTask setTaskReadState resumeWorkflow updatePerformer fetchWorkflowConfig checkAttachmentLockState getWorkflowStatusSummary verifyEntryCriterias verifyEntryCriteria launchScheduledWorkflow acquireTaskAndGetState getUnreadTasks delegateTaskEx addNoteToWorkflow doAutoTaskAction	checkPropertyPage checkPropertyPage getTaskFolderLabel canAbortWorkflow getConfigurationsNames getWorkflowDisplayName checkWorkflowAliases delegateTaskOnError isManualAcquisitionTask

opentext™

Services	Overridable Methods	Non-Overridable Methods
D2CheckoutService	checkout cancelCheckout testCheckout testControlledPrint	cancelCheckoutAll checkoutAsNew getNumberOfCheckoutDocument
D2CheckinService	getCheckinConfig	checkin
D2DownloadService	checkin getUploadUrls getUploadUrls getUploadUrls getPageServingUrl getCheckinUrls getDownloadUrls getExtractUrls getFile setFile hasAnyValidC2ExportAndRenditionConfig getDefaultServerInfo checkinAndLifeCycle extractDcoumentProperty getImportStructureUrls setDocumentProperty extractProperties getDownloadObjectId	getFileInfo getObjectsDownloadUrls canPrintControlledView getDispatchDownloadUrl getDownloadFileDetails isProtectedInControlledView addRendition

INFO

In case any unsupported methods are overriden by a plugin, it will be shown as warning in `D2-Smartview.log` during startup of the application

opentext™

Custom Widget Type

Developers can define custom widget types if the default set of widgets provided from OOTB D2. This helps in developing custom views and business operation to perform

Custom shortcut type in D2

D2-Smartview landing page configuration elements like `<tile>` requires a `type` attribute to be set. By default all "Widget type" found in D2-Config are accepted as valid values. However, while defining new shortcuts for the landing page tile one might need to use a value that is not a "Widget type" at all or in other words the value is not pre-defined. Additionally, while defining new application scope perspective one might need to declare a default widget name for the corresponding perspective to use when a direct URL based navigation is taking place in the UI.

To facilitate this, an SDK developer can create a properties file **{Plugin}/resources/strings/config/U4Landing.properties**. There are two properties that can be defined in the file -

- *default_widget_tags*: This allows declaring new valid XML tag names to go under the `<default-widgets>` section in the D2SV landing page file. Multiple comma separated values could be specified.
- *shortcut_types*: This allows declaring valid values for `type` attribute of `<tile>`. Multiple comma separated values could be specified.

Example

```
default_widget_tags=abcd,defg
shortcut_types=custom1,custom2
```

If this is the content of the **U4Landing.properties** file, then the following hypothetical landing page structure is accepted as valid configuration

```
<root>
    <space>
        <flow-layout-container>
            <content>
                <tile-container size="full">
                    <tile name="one" type="custom1">
                        <theme color="shade1" type="stone1"/>
                    </tile>
                    <tile name="two" type="custom2">
                        <theme color="shade1" type="indigo1"/>
                    </tile>
                </tile-container>
            </content>
        </flow-layout-container>
    </space>
    <default-widgets>
        <abcd>SOME_NAME</abcd>
        <defg>ANOTHER</defg>
    </default-widgets>
</root>
```

The developer needs to register the custom widget type if needed to create widgets which can driven through the D2-Config context matrix evaluation. To do this, developer needs to create a properties file **{Plugin}/resources/strings/config/WidgetSubtypelist.properties**. This will include all the widget type created in the format **{Widget Type Name}=true**

Example

```
CustomWidgetType=true
```

If developer wants the widget to inherit the properties of some other OOTB D2 widget types, then developer needs to prefix the parent widget type name in the format **{Parent Widget Type Name}.{Widget Type Name}=true**

Example

```
DocListWidget.CustomWidgetType=true
```

opentext™

If the developer want to have custom parameters as part of the custom widget type, developer needs to add those in the properties file **{Plugin}/resources/strings/config/WidgetSubtype.properties**. This will include the parameters for all the widget type created for plugin in the format **{Widget type Name}.{{Parameter name}={Default value}**.

Example

```
CustomerCustomType.sample1 = Text1
CustomerCustomType.sample2 = Text2
```

In order to provide custom labels for the custom parameters which are created needs to be add in the properties file **{Plugin}/resources/strings/actions/config/modules/widget/WidgetDialog/WidgetDialog_en.properties**. Entries will be in the format **label_{parameter name}={display label}**.

Example

```
label_sample1 = Sample text field 1
label_sample2 = Sample text field 2
```

Delta Menus in D2

Developers can create default menus as part of any context from the plugin. Any menu item can be appended in context.

Following are the currently supported contexts in D2 Smartview.

- MenuContext
- MenuContextDetailRelations
- MenuContextDetailRenditions
- MenuContextDetailVersions
- MenuContextSavedSearchObject
- MenuContextTaskDocument
- MenuContextTaskPriority
- MenuContextTasksList
- MenuContextVD
- MenuContextVDAddChildOption
- MenuContextVDReplaceChildOption
- MenuContextWorkflowOverview
- MenuDocumentD2LifeCycleAttach
- MenuDocumentLifeCycle
- MenuDocumentLifeCycleAttach
- MenuDocumentShare
- MenuDocumentWorkflow
- MenuNewObject
- MenuToolsMassUpdate
- MenuUser

opentext™

If the develop wants to define a OOTB default menu in the following path and format. `resources/xml/unitymenu/<context_name>Delta.xml`.

Menu item in the D2 will follow the below structure

```
<menuitem id="MenuId">
     <dynamic-action class="ClassName"/>
     <condition class="ClassName" equals="value"/>
  </menuitem>
```

Here we have mainly 3 part for the menu as follows

- **menuitem**: define the menu item which will have a id attribute. The id attribute can be used to define label and is also a unique identifier.
- **dynamic-action**: developer can define a class which is extended from the `IDynamicAction`. This tag is used to define the action that has to be performed when the menu item is clicked.
- **condition**: developer can define condition which has to be extended from `ICondition`

> **TIP**
>
> Depending on the class custom attributes can be passed to the tag for both `dynamic-action` and `condition`

In order to define the menu item we need to understand the root tag as `delta` tag. There are mainly 3 types of operation done on the menu item

- **insert** - This will be used to insert a new item. There are a couple of attribute which are `position-before` and `position-after` which will contain attribute value to menu id before or after which the current node has to be placed.
- **append** - This will be used to append item to the end of the.
- **modify** - this will need an id attribute which is a reference to any existing menu id which has to be modified. This can can be used to `delete`, `insert`, `append` and `insert-before`.
- **delete** - This will need an id attribute which will refer to the menu id that has to deleted.
- **insert-before** - This can be used along with the `modify` to add new conditions.

- **move**- This can be used to move and exiting menu by using the menu id and `position-before` and `position-after`.

> **TIP**
>
> 1. `position-before` and `position-after` attributes containes the menu id of other menus
> 2. menu id, class names for the dynamic actions and conditions can be identified by creating menu items in D2-Config and exporting the menus.

Find below some of the ways to use delta menus:

1. Insert new menu

```
<delta>
  <insert position-before="menuToolsMassUpdate">
    <menuitem id="menuContextViewPermission">
      <dynamic-action class="com.emc.d2fs.dctm.ui.dynamicactions.actions.U4Generic"
eMethod="getPermissions" eMode="SINGLE" eService="PermissionActionService"
rAction="${pluginNamespace}/dialogs/permissions/permissions.dialog:showPermissions"
rType="JS"/>
      <condition
class="com.emc.d2fs.dctm.ui.conditions.interfaces.IsMultipleSelection"
equals="false"/>
      <condition class="com.emc.d2fs.dctm.ui.conditions.options.IsPluginActivated"
value="${pluginName}"/>
    </menuitem>
  </insert>
  <insert position-before="menuToolsMassUpdate">
    <separator/>
  </insert>
</delta>
```

2. Modify an existing menu with new conditions

```
<delta>
    <modify id="menuDocumentEdit">
        <insert-before>
            <condition class="com.emc.d2fs.dctm.ui.conditions.typeparts.IsObjectType"
value="d2c_bin_deleted_document¬d2c_bin_deleted_folder¬d2c_bin_deleted_folder_dump¬d2c_
equals="false"/>
```

```
        </insert-before>
    </modify>
</delta>
```

Icons in D2 Smartview

Like any other application, D2 Smartview also uses different icons for visual context of data, infromation or action.

D2 Smartview uses Scalable Vector Graphic(svg) images to implement an icon. All the icons used in D2SV can be broadly classified into

- Non interactive

  Used to attach a context to a piece of data or information.

- Interactive a.k.a action icons

  Used to represent an action

Action icons are different semantically compared to the other type in the sense that action icons are reactive and respond to keyboard focus, blur or mosue events. Implementation wise, the SVG behind an action icon has a certain element structure whereas the other type don't have any such restriction.

Specification

Being SVG, a D2SV icon can be upscaled or downscaled to any size to fit in a UI element's boundary, however we recommend and follow that each SVG is defined w.r.t a view box of size `32px x 32px`.

Action icons must have 3 svg sub-elements with id `state`, `metaphor` and `focus`. The `state` element reacts to mouse position w.r.t to icon itself and changes its color accent. The `metaphor` element holds the visual graphic of the icon. And the `focus` element reacts to keyboard focus gain, giving itself a highlighted border.

How to use an icon

- For non-interactive icons, place the svg file at any source folder location and refer to it as `background-image` property in any CSS file using relative(from css file location to svg file locaiton) url.

  We follow a convention where an svg file is placed inside `impl/images` folder w.r.t to CSS file's location where the svg image will be used.

- For interactive icons, drop the svg file inside `utils/theme/action_icons` folder w.r.t the smartview source code directory. Then run `grunt compile` in the smartview source code directory. And finally, at the place of use (which is mostly inside node.actions.rules module), set property `iconName: '<pluginNamespace>_<svg_file_name_without_extension>'` after replacing the format with appropriate values.

Sample non-interactive icon

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 19.1.0, SVG Export Plug-In . SVG Version: 6.00 Build
0)  -->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
     viewBox="-379 291 32 32" enable-background="new -379 291 32 32"
xml:space="preserve">
<g>
    <circle fill="#7E929F" cx="-363" cy="307" r="16"/>
</g>
<circle fill="#FFFFFF" cx="-363" cy="315" r="2"/>
<path fill="#FFFFFF" d="M-361,309c0,1.1-0.9,2-2,2l0,0c-1.1,0-2-0.9-2-2v-11c0-1.1,0.9-
2,2-2l0,0c1.1,0,2,0.9,2,2V309z"/>
</svg>
```

Sample action icon

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 24.0.1, SVG Export Plug-In . SVG Version: 6.00 Build 0)
```

```
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
     viewBox="0 0 32 32" style="enable-background:new 0 0 32 32;" xml:space="preserve">
<style type="text/css">
    .st0{fill:none;}
    .st1{fill-rule:evenodd;clip-rule:evenodd;fill:#333333;}
    .st2{fill:none;stroke:#2E3D98;}
</style>
<circle id="state" class="st0" cx="16" cy="16" r="14"/>
<path id="metaphor" class="st1" d="M22.4,12.5H9.6c-0.331,0-0.6-0.336-0.6-
0.75S9.269,11,9.6,11h12.8c0.331,0,0.6,0.336,0.6,0.75
    S22.731,12.5,22.4,12.5z
M23.53,16.5H9.685C9.307,16.5,9,16.164,9,15.75S9.307,15,9.685,15H23.53c0.378,0,0.684,0.336,0
    S23.908,16.5,23.53,16.5z M9.628,20.5h10.785c0.346,0,0.628-0.336,0.628-
0.75S20.759,19,20.412,19H9.628C9.281,19,9,19.336,9,19.75
    S9.281,20.5,9.628,20.5z"/>
<circle id="focus" class="st2" cx="16" cy="16" r="15.5"/>
</svg>
```

TIP

For a list of built-in action icons refer to the Catalog

Upgrading Documentum D2 Smartview SDK to the latest version

With Documentum D2 Smartview SDK, new features will be added and delivered in each release. So,the SDK workspace need to be updated for consuming newly delivered feature.

With the Current version of D2 Smartview SDK,the existing workspace cannot be upgraded with the new version of SDK automatically. Developer need to migrate to the new version of SDK manually.

For manual migration, extract the latest version of D2 Smartview SDK for follow the steps for creating new development workspace. Once the new workspace is created, then copy the source code from old workspace to new workspace.

Ideally all the plugin project are created inside the "plugins" folder by default. Same workspace plugins folder name can be changed while creating the new plugin.

While creating a new plugin project, it's recommended to provide the same workspace assistance inputs which is captured in the prior version of SDK. But its not mandatory to provide the same value. Providing the same value will make upgrade process easy.

Please refer the following list for finding the workspace assistance inputs provided in prior version of SDK.

- Directory name to save this plugin project in - Refer the "modules" tag in workspace root "pom.xml"
- Maven group-id of the plugin - Refer the "groupId" tag in first level within the plugin's pom.xml.
- Name(maven artifact-id) of the plugin - Refer the "artifactId" tag in first level within the plugin's pom.xml.
- Version of the plugin - Refer the "version" tag in first level within the plugin's pom.xml.
- One liner description of the plugin - Refer the "name" and "description" tag in first level within the plugin's pom.xml.

- Enter package namespace for the plugin - Refer the content of property file in [plugin_name]/src/main/resources/D2Plugin.properties
- Include support for D2SV UI - Check if "smartview" folder exist in [plugin_name]/src/main

Once the new workspace is created with new version of D2 Smartview SDK, few files needs to be copied from old to new workspace and few files need to be merged.

- Files/Folders to be copied from [plugin_name]/src/main/resources
    - [plugin_name]/src/main/resources/strings
    - [plugin_name]/src/main/resources/xml
- Files/Folders to be copied from [plugin_name]/src/main/java
    - [plugin_name]/src/main/java/com/opentext/d2/smartview
- Files/Folders to be merged in [plugin_name]/src/main/java
    - [plugin_name]/src/main/java/com/opentext/d2/rest/context/jc
    - Spring bean configuration need to be merged in java configuration file which is located in mentioned path.
- Files/Folders to be copied from [plugin_name]/src/main/smartview
    - Following items must be ignored while moving the source code from old workspace to new workspace.
        - [plugin_name]/src/main/smartview/src/bundles
        - [plugin_name]/src/main/smartview/src/component.js
        - [plugin_name]/src/main/smartview/src/config-build.js
        - [plugin_name]/src/main/smartview/src/Gruntfile.js
        - [plugin_name]/src/main/smartview/src/extensions.json.js
        - [plugin_name]/src/main/smartview/src/[plugin_namespace]-init.js
    - Copy the rest of the files and folders from old workspace to new workspace.
- Files/Folders to be merged in [plugin_name]/src/main/smartview

- [plugin_name]/src/main/smartview/src/extensions.json
- UI extension need to be merged. So that customization from previous version will be included in latest version as well.

After moving and merging the source code to new workspace, open workspace assistance to compile all the plugins in the SDK workspace. If build is successfull, then the workspace upgrade is done.

opentext™

Workspace & Assistant

What is a workspace?

A workspace is the collection of tools, libraries, documentations put together in a specific directory structure that acts as a **Development Environment** and facilitates creation, management and build of D2 Plugin projects. D2SV SDK distributable comes in the form of a zip file named as `D2SV-SDK-<Version>.zip`. After extracting the zip and subsequently executing `ws-init.bat` successfully in the extracted folder turns it into a workspace.

What is the Assistant?

The workspace assistant is a command line utility that lives within a workspace & provides the functional aspects of the workspace with help of `NodeJS` & `Apache Maven` runtimes. While running, the assistant presents users with options to get something done within the corresponding workspace.

To run the assistant, open a command prompt/terminal in a workspace root directory and run

```
$npm start
```

Things that the workspace assistant is capable of doing are:

- Create a new plugin project
- Add smartview application scope perspective
- Add smartview UI support to an existing plugin
- Remove a plugin from workspace
- Add D2-REST controller to a plugin
- Build all plugins in the workspace
- Check out the documentation

opentext™

- Check out some samples
- Add smartview shortcut behavior
- Add smartview list tile
- Add smartview shortcut tile
- Add D2FS dialog to a plugin
- Add new metadata view to plugin
- Add new task details view to plugin

opentext™

Creating a plugin

This option is used to initiate a fresh D2SV plugin project. A plugin project is basically a maven project with all its dependencies pre-declared from the workspace `lib` folder. Based on selected options a plugin project may optionally have a Smartview UI component. If there is a D2SV UI component in a plugin, it requires NodeJS runtime to compile and package that specific component. All the relevant NodeJS and Javascript dependencies will be initialized upon plugin project creation.

To create a new plugin, a developer has to choose the **Create a new plugin project** option from the D2SV SDK workspace assistant.

Upon selecting the specific option in workspace assistant, a developer has to answer a few questions before the assistant can create and initialize the plugin project. For some of these questions asked, the workspace assistant will provide a meaningful contextual default answer based on usage, the default answer is enclosed within a pair of parentheses `()`, to choose the default value, one has to only press *Enter* key on the keyboard. These questions are self-explanatory however, here are a list of those questions and their meaning -

- Directory name to save this plugin project in

  Where to save the newly created project, defaults to `plugins` directory within the workspace.

- Maven group-id of the plugin

  Since all the plugin projects are maven projects, each project requires a group-id to be specified.

- Name(maven artifact-id) of the plugin

  Artifact identifier of the maven project to uniquely identify this plugin within the provided maven group ID.

- Version of the plugin

  Version of the plugin project.

- One liner description

  Used as the name and description for the underlying maven project. This is also shown as part of installed plugin data in D2 runtime.

- Package namespace

  A unique name used as prefix/suffix for generating the source code & properties in the maven project. The lowercase version of the given package name is used as part of the Java package name and also used as unique identifier for the Smartview UI code in the project, if any. For an example, if a Plugin project is created with Maven group-id `a.b.c` and it is given a package name `MyPlugin` then the base package for all Java source code becomes `a.b.c.myplugin` and the Smartview specific UI code is identified by `myplugin`.

- Include support for D2SV UI

  Whether to include D2 Smartview UI specific code infrastructure in the created plugin project. This question should be answered with an `Yes (Y)` only if the plugin is meant to develop, override or complement any D2SV front-end functionality. However, even for a plugin project initially created to develop or complement back-end oriented functionality, D2SV UI support can be added later through Add smartview UI support to an existing plugin project option.

Add Smartview application scope perspective

A perspective in D2 Smartview is loosely defined as something similar to a web-page in case of multi page web application. A perspective renders a view of semantically similar data with relevant UI controls and interactions to operate on the data. D2 Smartview switches from one perspective to another based on user interaction.

D2 Smartview has several perspective implementations like Landing, Doclist, Virtual Documents, Tasks & Workflows etc. Out of all these the **Landing persepctive** is the default and shown immediately after user login, unless the URL in browser's address bar points to a different hint.

The Landing perspective shows a collection of widget & shortcut tiles, some of them, upon click, opens up another persepctive with a more specialized representation of the corresponding data. The semantic associated with the data in such perspective is generally termed as **application scope**. All such application scope perspectives (tied to different widget tiles in Landing perspective) visually look similar irrespective of the data semantic that they represent.

This **Add smartview application scope perspective** option of workspace assistant, helps create the boilerplate if we want to define a new data semantic in addition to those already defined by D2 Smartview.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- This perspective goes to plugin

  Select a plugin project, inside which this boilerplate will be created.

- Name of the widget type to associate

Which landing page widget this perspective will associate itself with. Each landing page widget tile is configured with a `type` attribute, the possible values that can go against the `type` attribute can be used here.

- Name of the default smartview widget type(from landing config) to associate

  The Smartview landing page has a bunch of default widget configuration name defined. When an end-user navigates directly to an application scope using URL, the widget configuration data might be missin in the URL, in thsoe cases the default widget configuration is used to resolve the metadata requirement while opening the perspective.
  Answer to this question specifies which default widget configuration from the landing page is to be associated with the application scope being defined.

- Application scope of this perspective(also used as URL fragment)

  Name of this application scope. Also used as base part of the URL when this perspective is activated.

- Directory name where generated code will be put into

  Relative location of the boilerplate code w.r.t selected plugin projects widget source directory

- Default title of this perspective

  Default displayable label when this perspective is active.

- How would you describe this perspective?

  Description associated wtih this perspective

Add smartview UI support to an existing plugin project

This options of the workspace assistant can be used to add Smartview UI support to an existing D2SV plugin project, if the project was created initially without Smartview support.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- Add D2SV UI to plugin

  Select the D2SV plugin project, from the list of options, where Smartview UI code will be injected.

- Selected plugin seems to have D2SV UI support enabled already, overwrite it ?

  Confirm whether to overwrite the boilerplate code related to enabling D2SV UI support for the plugin. This question is asked only when the assistant detects that the selected plugin project already has D2SV UI support enabled in it.

Remove a plugin from workspace

This option is used to remove a plugin project from the corresponding workspace.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- Select plugin to remove from workspace

  Select which plugin project is to be removed from the workspace.

- Remove it completely from file system?

  Whether to remove the project completely from filesystem. In case of a soft removal, the project is left intact on the disk, however its entry from the aggregator POM in workspace is removed to exclude it from build order.

Add D2-REST controller support to a plugin project

If a D2SV plugin intends to deploy new D2-REST endpoints in addition to the factory endpoints, then this option of the workspace assistant comes in handy.

Upon successful execution, this option, can add a boilerplate REST controller definition to the plugin such that the controller handles HTTP GET(Retrieve), POST(Create), DELETE(Delete) operation on the specified endpoint resource matching part of the CRUD style transaction.

The complete URL path of the endpoint created is represented as -

```
/D2-Smartview/repositories/<repo_name>/<group_name>/<endpoint_name>
```

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- Select plugin to add REST support

  Specify the plugin project where to add boilerplate for this controller

- Group name of controller

  Group name for the service that this REST controller implements. Used as `<group_name>` part of the URL format mentioned above. Usually same group name is used across multiple services if they all happen to be correlated.

- Endpoint name of controller

A meaningful name that should uniquely identify this endpoint in the group of correlated services. This name is used as `<endpoint_name>` part of the URL format mentioned above.

- Service name to use against the controller

  Name of the service that this endpoint represents. This name is used to form the name of Java classes and interfaces which are finally going to implement the service.

Build all plugins in the workspace

This option builds all the plugin projects in the workspace whose entries are found in the aggregator `pom.xml` file in the workspace.

Basically it runs `mvn clean package` command in the workspace root directory. As part of this option, final build output from each of the plugin project is collected in the `dist` directory right under the workspace root directory.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Checkout documentation

This option starts the embedded documentation server and opens the home page of it in the default browser.

Checkout samples

The D2SV SDK packs a few working samples to demonstrate the building blocks of D2SV runtime APIs. This option of the workspace assitant is used to unpack a sample into the workspace such that subsequently it can be built and deployed on a running D2 Smartview or simly the source code could be checked out as tutorial.

Upon selecting this option, the only additional question to be answered is to pick the sample that is to be extracted.

Add smartview shortcut behavior

D2 Smartview landing perspective can be configured to have a number of shortcut tiles. Each of these shortcut tiles can do something specific and different from others based on its type, however their visual representations are same irrespective of what they do.

The function behind a particular type of shortcut is defined by a shortcut behavior attached to the shortcut type.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

This assistant option lets define a new shortcut behavior that can be attached to a shortcut type.

Associated questions and their meanings -

- This shortcut goes to plugin

  Select the plugin project where to create the boilerplate for the shortcut behavior

- Type of the target shortcut

  Declare the type of shortcut tile that are to be associated with this particular behavior

Add smartview list tile

Using this option of the assistant, a new list type of tile a.k.a widget tile definition could be added.

All widget tiles shown in D2 Smartview landing perspective are visually similar, however they are backed by different type of widget conifiguration and display data evaluated in the context of that widget.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- This shortcut goes to plugin

  To select the plugin project where to create boilerplate associated with creating a widget tile.

- Name of Widget type to associate

  Specify which type of widget configuration will drive this list tile data.

- CSS class name for the icon

  CSS class selector to put in the HTML element hosting the icon for this widget tile. Later this same class name could be used to render a specific icon for this tile.

- Directory name where generated code will be put into

  Relative location of the boilerplate code w.r.t selected plugin projects widget source directory

- Default title of this tile

  Specifies an optional default name for the tile which is shown at the header region of this tile incase the underlying widget configuration is not able to provide a name for it.

- How would you describe this tile?

  Description metadata for this tile.

- Will it expand to own perspective?

  Whether this tile will expand to its own application scope perspective. If anwered with `yes` then a bunch of followup question related to the application scope perspective are asked. Add smartview application scope perspective can be referred for meaning of such questions.

Add smartview shortcut tile

Using this option of the assistant, a shortcut type of tile definition could be created.

D2 Smartview has a bunch of shortcut tile implemenation on its own. Visually all the shortcuts look similar except their representation icon. However, each type of shortcut has its own mechanism to react to click event.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- This shortcut goes to plugin

  To select the plugin project where to create boilerplate associated with creating a widget tile.

- Type of this shortcut

  Declare the type of this shortcut tile definition. Creating custom widget type can be referred for more information.

- CSS class name for the icon

  CSS class selector to put in the HTML element hosting the icon for this shortcut tile. Later this same class name could be used to render a specific icon for this tile.

- Does it require a widget config?

  Shortcut tiles can optionally be backed by a widget config(E.g. Doclist type of shortcut requires a `DoclistWidget` config). Answering with `yes` associates this shortcut with a widget config.

- What this shortcut should do on click?

Defines what happens when end user clicks on this tile. Can be one of -

- Execute inline handler
  A JS callback function will implement the action.
- Execute a behavior
  The action is delegated to a shortcut behavior implementaion. The actual definition of the beahvior is to be coded into the boilerplate created.
- Expand to perspective
  The action is to open an application scope perspective. Further questions are asked for creating the application scope perspective. Add smartview application scope perspective can be referred for meanings of such questions.

Add D2FS dialog to a plugin

If a D2SV plugin intends to define a property-page like form then this assistant option could be used to create the boilerplate associated with such form, which is also known as D2FS dialog.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- Select plugin to add new D2FS dialog

  Specify the plugin project where to add boilerplate for this dialog

- Enter name of the dialog

  A unique name of the dialog. It is also used as an ID to refer to the dialog from other part of D2SV runtime.

- Title of the dialog

  The title to be displayed when the dialog is visible on screen. It defaults to the given name of the dialog.

- Create a menu to open the dialog?

  Whether to also define a menu item in D2SV context menu so that clicking that menu would show the dialog. Defaults to `Yes`.

- Label for the menu

  Define an English label for the menu. This question is asked only if the previous question was answered with an `Yes`.

- Pick a toolbar to add this menu to

  D2-Smartview UI shows different type of menu bars depending on the application context. By answering this option we can specify the menubar where this menu is going to be added. This question is asked only if 'Create menu to open the dialog?' was answered with `Yes`.

Add new metadata view to plugin

If a D2SV plugin intends to define new views like properties, versions, permissions, task performers then this assistant option could be used to create the boilerplate code associated with such views, which is also known as metadata panel.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- Select plugin to add new metadata view

  Specify the plugin project where to add boilerplate code for the new metadata view.

- Enter name for the view

  A unique name for the new metadata view.

  This will be the option name shown in the dropdown by default.

Associated generated boilerplate files and their use -

- The generated files would be present under `<Plugins Directory>\<Selected Plugin>\src\main\smartview\src\widgets\metadata\panels\<View Name>`

- `impl` folder

  This folder contains the handlebar template file and the style sheet file used by the view.

  It also contains the `nls` folder which contains the lang files used for the translation strings.

The option name shown in the dropdown could be changed by changing the translation string value for viewName in `lang.js` file under `nls/root` folder.

- `metadata.<View Name>.view.js`

  This is the main view file for the new metadata view created.

  This will have code for a simple helloworld view.

  The template file, style sheet and lang file used by this view are already loaded.

  A wrapper class name for this view, ui, regions and events are defined in this view for reference.

  Modify this view based on the usecase.

  Incase of complex view, break it into smaller independent views and keep them under `impl` folder, specify regions in the main view and show these smaller views using regions.

  By default, this view will be shown for all metadata dropdowns. Add conditions in enabled function to restrict this view based on the usecase.

Add new task details view to plugin

If a D2SV plugin intends to define new views like working files, supporting files, task notes then this assistant option could be used to create the boilerplate code associated with such views, which is also known as task details panel.

> **CAUTION**
>
> Use this option when at least one plugin project exists in the workspace.

Associated questions and their meanings -

- Select plugin to add new task details view

  Specify the plugin project where to add boilerplate code for the new task details view.

- Enter name for the view

  A unique name for the new task details view.

  This will be the name shown for the tab by default.

Associated generated boilerplate files and their use -

- The generated files would be present under `<Plugins Directory>\<Selected Plugin>\src\main\smartview\src\widgets\task.details\panels\<View Name>`

- `impl` folder

  This folder contains the handlebar template file and the style sheet file used by the view.

  It also contains the `nls` folder which contains the lang files used for the translation strings.

The tab name could be changed by changing the translation string value for tabName in `lang.js` file under `nls/root` folder.

- `task.<View Name>.view.js`

  This is the main view file for the new task details view created.

  This will have code for a simple helloworld view.

  The template file, style sheet and lang file used by this view are already loaded.

  A wrapper class name for this view, ui, regions and events are defined in this view for reference.

  Modify this view based on the usecase.

  By default, this view will be visible in task and workflow tab panel. Add conditions to restrict this view based on the usecase.

opentext™

Packaged Samples

D2SV SDK includes a few sample plugins as part of its distribution. They could be extracted in a workspace by using Checkout some samples option of the workspace assistant.

List of samples

- D2 Admin Groups Sample
- D2SV client to server logging
- D2SV Custom Dialogs(D2FS) sample
- D2SV Read-Only permission display

opentext™

D2 Admin-Groups Sample

D2 Smartview does not ship with an Out-Of-The-Box(OOTB) group management widget like D2 Classic. However, the D2 Admin-Groups sample plugin fills-in the gap functionally and serves as a complete example of how to use SDK to

- Define a landing page widget tile.
- Define a perspective and stich it up with the landing page widget.
- Define and use a custom menu type to go with the widget.
- Define a few REST endpoints to serve data to the widget.

Instruction to try out the sample

As this plugin implements a landing page tile as part of it, some configuration changes are required in the D2 Smartview landing page before the tile is made available for the end users. Here are the list of steps required to completely deploy and configure this plugin.

- Build the plugin using `npm run build` from SDK workspace root.
- Copy `D2-AdminGroups-1.0.0.jar` from 'dist' folder in workspace root and paste it inside `WEB-INF/lib` folder of a deployed D2 Smartview application.
- Restart application server on which D2 Smartview is deployed.
- Open D2-Config web application in browser, login and then navigate to **Widget view -> Widget...** from menu.
- Create a new widget configuration, and put "Manage Groups"(or whatever you wish) as **Name** and select "AdminGroupsWidget" for **Widget type** field.
- Fill-in other fields as necessary and save the configuration.
- From the toolbar, click **Matrix** to go to D2-config matrix and enable the widget configuration, you just created, against appropriate contexts.
- Select **Widget view -> Smart View Landing Page...** from menubar to navigate to landing page configurations.

- Select an applicable configuration from the left side and click **Download** to get the structure file locally and then open in notepad to edit.

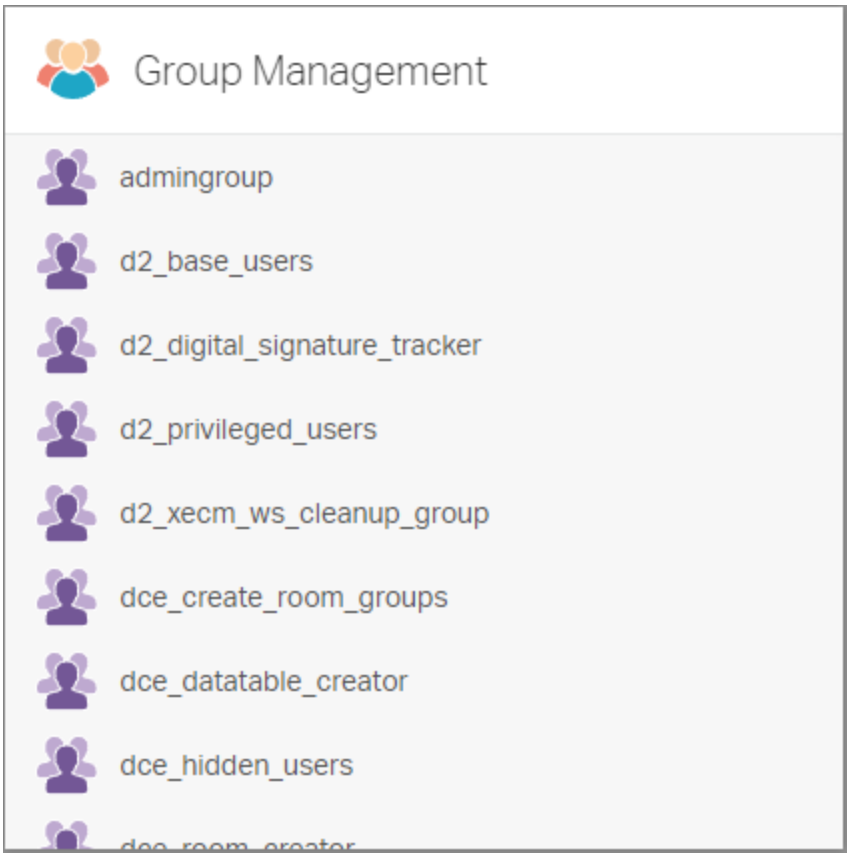
  > **TIP**
  >
  > If a pre-created Smartview landing page configuration does not exist, then refer to D2 Administration Guide documentation to create the same and learn basics of landing page structure file.

- Paste the following piece of xml anywhere right under the `<content>` tag

```
<widget-container>
    <widget name="Manage Groups" type="AdminGroupsWidget"/>
</widget-container>
```

> **TIP**
>
> If you've used a different name while creating the widget configuration, use that name as the value for **name** attribute.

- Save the landing structure xml file and upload it to D2-Config under the same landing page configuration from where we downloaded it before.
- Save the configuration change in D2-Config and click **Tools -> Refresh cache** from menubar.

- Reloading the D2 Smartview at this point should show an additional widget in the landing page, similar to following 

Source code structure

```
D2-AdminGroups
|
|
|   pom.xml
|
+---src
|   \---main
|       +---java
|       |   \---com
|       |       +---emc
|       |       |       D2PluginVersion.java
|       |       |
|       |       \---opentext
|       |           \---d2
|       |               +---rest
|       |               |   \---context
|       |               |       \---jc
```

```
|       |               |               PluginRestConfig_AdminGroups.java
|       |               |
|       |               \---smartview
|       |                   \---admingroups
|       |                       |   AdminGroupsPlugin.java
|       |                       |
|       |                       +---api
|       |                       |       AdminGroupsVersion.java
|       |                       |
|       |                       \---rest
|       |                           |   package-info.java
|       |                           |
|       |                           +---controller
|       |                           |       AdminGroupsController.java
|       |                           |       AdminGroupsMembersController.java
|       |                           |
|       |                           +---dfc
|       |                           |   |   AdminGroupsManager.java
|       |                           |   |
|       |                           |   \---impl
|       |                           |           AdminGroupsManagerImpl.java
|       |                           |
|       |                           +---model
|       |                           |       GroupMembers.java
|       |                           |       GroupModel.java
|       |                           |       UserModel.java
|       |                           |
|       |                           \---view
|       |                                   GroupMembersFeedView.java
|       |                                   GroupsFeedView.java
|       |                                   GroupView.java
|       |                                   UsersFeedView.java
|       |                                   UserView.java
|       |
|       +---resources
|       |   |   admingroups-version.properties
|       |   |   D2Plugin.properties
|       |   |
|       |   +---smartview
|       |   |       SmartView.properties
|       |   |
|       |   +---strings
|       |   |   \---menu
|       |   |       \---PMenuContextAdminGroups
```

```
|       |   |               PMenuContextAdminGroups_en.properties
|       |   |
|       |   \---xml
|       |       \---unitymenu
|       |               PMenuContextAdminGroups.xml
|       |
|       \---smartview
|           |   .csslintrc
|           |   .eslintrc-html.yml
|           |   .eslintrc.yml
|           |   .npmrc
|           |   admingroups.setup.js
|           |   config-editor.js
|           |   esmhelper.js
|           |   Gruntfile.js
|           |   package.json
|           |   proxy.js
|           |   server.conf.js
|           |   server.js
|           |
|           +---pages
|           |   |   config-d2.js
|           |   |   config-dev.js
|           |   |   favicon.ico
|           |   |   initialize.js
|           |   |   loader.css
|           |   |   otds.html
|           |   |   redirect.html
|           |   |
|           |   +---debug
|           |   |       app.html
|           |   |
|           |   \---release
|           |           app.html
|           |
|           +---src
|           |   |   admingroups-init.js
|           |   |   component.js
|           |   |   config-build.js
|           |   |   extensions.json
|           |   |   Gruntfile.js
|           |   |
|           |   +---bundles
|           |   |       admingroups-bundle.js
```

```
|           |   |
|           |   +---commands
|           |   |   |   manage.group.js
|           |   |   |   node.actions.rules.js
|           |   |   |
|           |   |   \---nls
|           |   |       |   lang.js
|           |   |       |
|           |   |       \---root
|           |   |               lang.js
|           |   |
|           |   +---dialogs
|           |   |   \---manage.group
|           |   |       |   manage.group.dialog.js
|           |   |       |   manage.group.view.js
|           |   |       |
|           |   |       \---impl
|           |   |           |   group.members.form.view.js
|           |   |           |   manage.group.css
|           |   |           |   manage.group.hbs
|           |   |           |
|           |   |           \---nls
|           |   |               |   lang.js
|           |   |               |
|           |   |               \---root
|           |   |                       lang.js
|           |   |
|           |   +---extensions
|           |   |   |   admin.groups.icon.sprites.js
|           |   |   |   admin.groups.perspective.js
|           |   |   |   admin.groups.tile.js
|           |   |   |
|           |   |   \---nls
|           |   |       |   lang.js
|           |   |       |
|           |   |       \---root
|           |   |               lang.js
|           |   |
|           |   +---models
|           |   |       admin.groups.collection.js
|           |   |       group.members.collection.js
|           |   |       group.model.js
|           |   |       member.model.js
|           |   |
```

```
|           |   +---test
|           |   |       extensions.spec.js
|           |   |
|           |   +---utils
|           |   |   |   alert.util.js
|           |   |   |   constants.js
|           |   |   |   menu.utils.js
|           |   |   |   startup.js
|           |   |   |
|           |   |   +---contexts
|           |   |   |   \---factories
|           |   |   |           admin.groups.collection.factory.js
|           |   |   |           next.group.factory.js
|           |   |   |
|           |   |   +---perspectives
|           |   |   |       admin.groups.perspective.json
|           |   |   |
|           |   |   \---theme
|           |   |       |   action.icons.js
|           |   |       |
|           |   |       \---action_icons
|           |   |               action_sample_icon.svg
|           |   |
|           |   \---widgets
|           |       +---admin.groups
|           |       |   |   admin.groups.manifest.json
|           |       |   |   admin.groups.view.js
|           |       |   |   toolitems.js
|           |       |   |
|           |       |   \---impl
|           |       |       |   admin.groups.css
|           |       |       |
|           |       |       +---images
|           |       |       |       group-svgrepo-com.svg
|           |       |       |
|           |       |       \---nls
|           |       |           |   admin.groups.manifest.js
|           |       |           |   lang.js
|           |       |           |
|           |       |           \---root
|           |       |                   admin.groups.manifest.js
|           |       |                   lang.js
|           |       |
|           |       \---admin.groups.members
```

```
|           |           |   admin.groups.members.view.js
|           |           |
|           |           \---impl
|           |               |   admin.groups.members.css
|           |               |
|           |               \---nls
|           |                   |   lang.js
|           |                   |
|           |                   \---root
|           |                           lang.js
|           |
|           \---test
|                   Gruntfile.js
|                   karma.js
|                   test-common.js
|
\---target
```

Files and their purpose

Following are the list of function oriented source files and their purpose. Other source files present within the plugin are part of common infrastructure code and explained in Understanding D2SV plugin project.

REST Controller

- src/main/java/com/opentext/d2/rest/context/jc/PluginRestConfig_AdminGroups.java - Declares Spring Bean `AdminGroupsManager` through `AdminGroupsManagerImpl`.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/controller/AdminGroupsController.java - Defines a REST controller with two endpoints to list all the users and groups from Documentum.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/controller/AdminGroupsMembersController.java - Defines a REST controller with two endpoints to list and edit members of a group.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/dfc/AdminGroupsManager.java - Declares the data manager interface used by above REST controllers to get/set the data they deal with.

- src/main/java/com/opentext/d2/smartview/admingroups/rest/dfc/impl/AdminGroupsManagerImpl.java - Data manager that interacts with Documentum through DQL and exchanges data as per `AdminGroupsManager` interface.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/model/GroupMembers.java - Serializable POJO that represents members of a group while editing.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/model/GroupModel.java - Serializable POJO that represents a single group.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/model/UserModel.java - Serializable POJO that represents a single user.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/view/GroupMembersFeedView.java - Spring view used to wrap and serialize a list of group member data. Uses `UserView` in turn to serialize each individual member.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/view/GroupsFeedView.java - Spring view used to wrap and serialize a list of group data. Uses `GroupView` in turn to serialize each individual group.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/view/GroupView.java - Spring view used to seralize a single group data.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/view/UsersFeedView.java - Spring view used to wrap and serialize a list of user data. Uses `UserView` in turn to serialize each individual user.
- src/main/java/com/opentext/d2/smartview/admingroups/rest/view/UserView.java - Spring view used to serialize a single user data.

Group-manage menu configuration in back-end and its display & execution on the front-end

- src/main/resources/strings/menu/PMenuContextAdminGroups/PMenuContextAdminGroups_en.properties - Labels associated with the dynamically configured menu.
- src/main/resources/xml/unitymenu/PMenuContextAdminGroups.xml - The menu definition file that dynamically adds a new type(PMenuContextAdminGroups) of menu for the D2FS `D2MenuService` to discover and return for D2 Smartview.
- src/main/smartview/src/bundles/admingroups-bundle.js - A portion of this file is used to refer to key RequireJS modules that define the extensions to the toolbar and menu related D2SV UI API.

```
define([
    ...
    'admingroups/utils/startup',
    'admingroups/commands/node.actions.rules',
    'admingroups/commands/manage.group',
    ...
], {});
```

- src/main/smartview/src/commands/manage.group.js - A `CommandModel` that implements the executable logic when a user clicks the `Manage` menu on the UI. It dynamically loads and displays `manage.group.dialog.js` dialog. When the dialog closes(without cancel flag), it makes an AJAX call to group-update related REST endpoint created by the Java code from above and then finally shows a toast message on successful completion.
- src/main/smartview/src/commands/node.actions.rules.js - Defines client side filtering and association logic to attach the above `manage.group.js` command model implementaion to a UI toolbar item.
- src/main/smartview/src/utils/menu.utils.js - Parses the data for `PMenuContextAdminGroups` type of menu into a client-side toolbar definition that in turn is used by landing page tile & perspective.
- src/main/smartview/src/utils/startup.js - Runs as part of D2 Smartview client-side startup flow. This flow is executed everytime endusers reload the D2 Smartview application in their internet browser. As part of this startup hook, an AJAX call is made to get the menu configuration data for type `PMenuContextAdminGroups`, then the response is trasformed into a toolbar definition by `menu.utils.js`.
- src/main/smartview/src/extensions.json - A portion of this file registers extensions to the toolbar and menu related D2SV UI API. The corresponding portion is highlighted below

```
"d2/sdk/commands/node.actions.rules": {
    "extensions": {
      "admingroups": [
        "admingroups/commands/node.actions.rules"
      ]
    }
  },
  "d2/sdk/utils/commands": {
    "extensions": {
      "admingroups": [
```

```
        "admingroups/commands/manage.group"
      ]
    }
  }
```

Tile on the landing page

- src/main/smartview/src/bundles/admingroups-bundle.js - A portion of this file refers to the the RequireJS modules that implement the landing page tile. These references are only used for RequireJS modules that are otherwise not referenced statically from any other RequireJS module.

```
define([
   ...
   'admingroups/extensions/admin.groups.icon.sprites',
   ...
   'admingroups/extensions/admin.groups.tile',
   'admingroups/widgets/admin.groups/admin.groups.view'
   ...
   'json!admingroups/widgets/admin.groups/admin.groups.manifest.json',
   'i18n!admingroups/widgets/admin.groups/impl/nls/admin.groups.manifest'
], {});
```

- src/main/smartview/src/extensions/admin.groups.icon.sprites.js - Defines the icon to represent a user and a group by means of extension.
- src/main/smartview/src/extensions/admin.groups.tile.js - Declares a new widget type handler for the landing page tiles by means of extension.
- src/main/smartview/src/extensions.json - A portion of this file registers extensions to the landing page tile & icon related D2SV UI API. The corresponding portion is highlighted below

```
"d2/sdk/utils/landingpage/tiles": {
    "extensions": {
      "admingroups": [
        "admingroups/extensions/admin.groups.tile"
      ]
    }
  },
  ...
  "d2/sdk/controls/icon.sprites/node.icon.sprites": {
```

```
      "extensions": {
        "admingroups": [
          "admingroups/extensions/admin.groups.icon.sprites"
        ]
      }
    }
```

- src/main/smartview/src/models/admin.groups.collection.js - A `BackboneJS` collection that holds all available groups data. Uses `group.model.js` to represent each group in the collection. Makes an AJAX call to one of the REST endpoint, created by Java code from above, to get the available groups data.
- src/main/smartview/src/models/group.model.js - A `BackboneJS` model that holds data for a single group.
- src/main/smartview/src/utils/contexts/factories/admin.groups.collection.factory.js - A factory to control creation of initialized/uninitialized group collection instances.
- src/main/smartview/src/utils/contexts/factories/next.group.factory.js - A factory to control creation of group-model instances. Purpose of this factory is to create a single instance of group model and use that to reflect current selected group item in the UI. This instance is used by `admin.groups.view.js` to constantly update the selected group data as user clicks an item in the list of visible groups in UI.
- src/main/smartview/src/utils/constants.js - A portion of the file defines a few frequently used constant values in the context of landing tile implementation.
- src/main/smartview/src/widgets/admin.groups/admin.groups.view.js - A `MarionetteJS` view that implements the UI and function for the widget. An instance of this view is dynamically created by the D2SV runtime and this instance manages and renders DOM elements to represent the list of all available groups. It also handles user interaction with the DOM elements.
- src/main/smartview/src/widgets/admin.groups/toolitems.js - The toolbar configuration used by `admin.groups.view.js` to display inline `Manage` menu. An instance of it is manipulated by `menu.utils.js` to dynamically inject the menu items in the toolbar at the time of D2SV UI startup.

The perspective, landing tile expands into

The perspective is defined in a two panel layout where the left-side re-uses the same `admin.groups.view.js` from landing page tile. Apart from the RequireJS modules and other source

code resources referred by the landing page tile, here's the other files involved in defining the perspective itself besides the right-side part of it.

- src/main/smartview/src/bundles/admingroups-bundle.js - A portion of this file refers to the the RequireJS modules that implement the perspective and right-side part of it. These references are only used for RequireJS modules that are otherwise not referenced statically from any other RequireJS module.

```
define([
   ...
   'admingroups/extensions/admin.groups.perspective',
   ...
   'admingroups/widgets/admin.groups.members/admin.groups.members.view',
   'json!admingroups/utils/perspectives/admin.groups.perspective.json',
   ...
], {});
```

- src/main/smartview/src/extensions/admin.groups.perspective.js - Declares an application scope handler and associates a perspective definition file with it.
- src/main/smartview/src/extensions.json - A portion of this file registers extensions toward application scope perspective related D2SV UI API. The corresponding portion is highlighted below

```
"d2/sdk/utils/perspectives/app.scope.perspectives": {
    "extensions": {
      "admingroups": [
        "admingroups/extensions/admin.groups.perspective"
      ]
    }
  }
```

- src/main/smartview/src/models/group.members.collection.js - A `BackboneJS` collection that holds membership data for a given group. Uses `member.model.js` to represent each member within the group. Makes an AJAX call to one of the REST endpoint, created by Java code from above, to get the members data for a selected group.
- src/main/smartview/src/models/member.model.js - A `BackboneJS` model that holds data for a member within a group.

- src/main/smartview/src/utils/contexts/factories/next.group.factory.js - A factory to control creation of group-model instances. Purpose of this factory is to create a single instance of group model and use that to reflect current selected group item in UI. This instance is used by `admin.groups.members.view.js` to constantly monitor change to the selected group in UI.
- src/main/smartview/src/utils/perspectives/admin.groups.perspective.json - Defines the layout for the perspective and associates `admin.groups` & `admin.groups.members` as the widgets to go on the left and right side respectively. The D2SV runtime dynamically creates the associated view instances when the perspective comes alive.
- src/main/smartview/src/widgets/admin.groups.members/admin.groups.members.view.js - A `MarionetteJS` view implementation that displays the members of a selected group. An instance of this view is dynamically created by D2SV runtime to show list of members in the perspective. The instance automatically updates itself as a result of end users selecting a group from the left-side by means of constant watch over the group model instance acquired using `next.group.factory.js`.

The side-panel dialog that manages group member

- src/main/smartview/src/bundles/admingroups-bundle.js - A portion of this file refers to the the RequireJS modules that implements the dialog. These references are only used for RequireJS modules that are otherwise not referenced statically from any other RequireJS module.

```
define([
   ...
   'admingroups/dialogs/manage.group/manage.group.dialog',
   ...
], {});
```

- src/main/smartview/src/dialogs/manage.group/manage.group.dialog.js - Uses D2SV UI API to create a side panel and host an instance of `manage.group.view.js` to show the related UI. Also collects information about updated group members and relays that to the caller.
- src/main/smartview/src/dialogs/manage.group/manage.group.view.js - A `MarionetteJS` view that wraps an instance of `group.members.form.view.js` to make it renderable within the side panel and defers the instance creation & rendering until required membership data has been fetched through an instance of `group.members.collection.js`. Also defines utility methods to have a check on

whether the membership data has changed from what is loaded initially. Uses `manage.group.hbs` & `manage.group.css` files for HTML templating and CSS styling respectively.

- src/main/smartview/src/dialogs/manage.group/impl/group.members.form.view.js - Uses D2SV UI API to create a statically defined form with multi-select list field to show membership information for the selcted group. Also makes an AJAX call to one of the REST endpoint, created by Java code from above, to get all available users data that serves as the options shown while editing the membership data. Also defines utility method to get the membership information for the selected group at any time.

D2SV Read-Only Permission View Sample

D2 Read-Only permission view sample plugin fills-in the gap functionally and serves as a complete example of how to use SDK to

- Define a custom action services
- Define a custom menu by default and initiate a dialog
- Define a custom dialog view to display the information and show the form view of the data

Instruction to try out the sample

Developer can extract the sample and build it using the workspace assistant. Once built, the distribution is collected in 'dist' folder as **D2SV-ReadOnlyPermission-View-1.0.0.jar** which can placed in `WEB-INF/lib` directory of a deployed D2 Smartview. The application server needs to be restarted post deployment.

As result of deploying this plugin, it will introduce a new menu in `Doclist` widget as `View Permission`

Source code structure

```
D2SV-ReadOnlyPermission-View
    |   pom.xml
    |
    +---src
    |   \---main
    |       +---java
    |       |   \---com
    |       |       +---emc
    |       |       |       D2PluginVersion.java
    |       |       |
    |       |       \---opentext
    |       |           \---d2
    |       |               +---rest
    |       |               |   \---context
    |       |               |       \---jc
```

```
|       |               |               PluginRestConfig_D2SVROPView.java
|       |               |
|       |               \---smartview
|       |                   \---d2svropview
|       |                       |   D2SVROPViewPlugin.java
|       |                       |
|       |                       +---api
|       |                       |       D2SVROPViewVersion.java
|       |                       |
|       |                       +---rest
|       |                       |       package-info.java
|       |                       |
|       |                       \---webfs
|       |                           \---custom
|       |                                   PermissionActionService.java
|       |
|       +---resources
|       |   |   D2Plugin.properties
|       |   |   d2svropview-version.properties
|       |   |
|       |   +---smartview
|       |   |       SmartView.properties
|       |   |
|       |   +---strings
|       |   |   \---menu
|       |   |       \---MenuContext
|       |   |               MenuContext_en.properties
|       |   |
|       |   \---xml
|       |       \---unitymenu
|       |               MenuContextDelta.xml
|       |
|       \---smartview
|           |   .csslintrc
|           |   .eslintrc-html.yml
|           |   .eslintrc.yml
|           |   .npmrc
|           |   config-editor.js
|           |   d2svropview.setup.js
|           |   esmhelper.js
|           |   Gruntfile.js
|           |   package.json
|           |   proxy.js
|           |   server.conf.js
```

```
|           |   server.js
|           |
|           +---pages
|           |   |   config-d2.js
|           |   |   config-dev.js
|           |   |   favicon.ico
|           |   |   initialize.js
|           |   |   loader.css
|           |   |   otds.html
|           |   |
|           |   +---debug
|           |   |       app.html
|           |   |
|           |   \---release
|           |           app.html
|           |
|           +---src
|           |   |   component.js
|           |   |   config-build.js
|           |   |   d2svropview-init.js
|           |   |   extensions.json
|           |   |   Gruntfile.js
|           |   |
|           |   +---bundles
|           |   |       d2svropview-bundle.js
|           |   |
|           |   +---commands
|           |   |   |   node.actions.rules.js
|           |   |   |   view.permission.js
|           |   |   |
|           |   |   \---impl
|           |   |       \---nls
|           |   |           |   lang.js
|           |   |           |
|           |   |           \---root
|           |   |                   lang.js
|           |   |
|           |   +---dialogs
|           |   |   \---permissions
|           |   |       |   permissions.dialog.js
|           |   |       |   permissions.view.js
|           |   |       |
|           |   |       \---impl
|           |   |           |   permission.collection.js
```

```
|           |   |           |   permissions.css
|           |   |           |   permissions.hbs
|           |   |           |   table.columns.js
|           |   |           |
|           |   |           \---nls
|           |   |               |   lang.js
|           |   |               |
|           |   |               \---root
|           |   |                       lang.js
|           |   |
|           |   +---test
|           |   |       extensions.spec.js
|           |   |
|           |   \---utils
|           |       |   startup.js
|           |       |
|           |       \---theme
|           |           |   action.icons.js
|           |           |
|           |           \---action_icons
|           |                   action_sample_icon.svg
|           |                   action_view_perms32.svg
|           |
|           \---test
|                   Gruntfile.js
|                   karma.js
|                   test-common.js
|
\---target
```

Files and their purpose

Following are the list of function oriented source files and their purpose. Other source files present within the plugin are part of common infrastructure code and explained in Understanding D2SV plugin project.

REST Implementations

- pom.xml - Defines the maven project for this plugin.

- src/main/java/com/emc/D2PluginVersion.java - Declares identification information for the entire plugin using `D2SVROPViewVersion` class
- src/main/java/com/opentext/d2/rest/context/jc/PluginRestConfig_D2SVROPView.java - Java configuration for spring components like REST controller, Beans etc.
- src/main/java/com/opentext/d2/smartview/d2svropview/D2SVROPViewPlugin.java - Declares a plugin component for D2FS.
- src/main/java/com/opentext/d2/smartview/d2svropview/api/D2SVROPViewVersion.java - Holder for plugin identification information. Loads relevant data from `d2svropview-version.properties` file resource.
- src/main/java/com/opentext/d2/smartview/d2svropview/rest/package-info.java - Declares package metadata for JDK and IDE.
- src/main/java/com/opentext/d2/smartview/d2svropview/rest/webfs.custom/PermissionActionService.java - Defines a custom service to fetch the basic permissions for the given object id. So the menu will have reference to the method 'getPermissions' which will triggered from the menu action.

View permission menu configuration in back-end and its display & execution on the front-end

- src/main/resources/strings/menu/MenuContext/MenuContext_en.properties - Labels associated with the dynamically configured menu.
- src/main/resources/xml/unitymenu/MenuContextDelta.xml - This delta menu will be used to dynamically load the custom OOTB menu to view permissions in the D2 Smartview for the default MenuContext.

```
<delta>
  <insert position-before="menuToolsMassUpdate">
    <menuitem id="menuContextViewPermission">
      <dynamic-action class="com.emc.d2fs.dctm.ui.dynamicactions.actions.U4Generic"
eMethod="getPermissions" eMode="SINGLE" eService="PermissionActionService"
rAction="d2svropview/dialogs/permissions/permissions.dialog:showPermissions"
rType="JS"/>
      <condition class="com.emc.d2fs.dctm.ui.conditions.interfaces.IsMultipleSelection"
equals="false"/>
      <condition class="com.emc.d2fs.dctm.ui.conditions.options.IsPluginActivated"
value="D2SV-ReadOnlyPermission-View"/>
```

```
    </menuitem>
  </insert>
  <insert position-before="menuToolsMassUpdate">
    <separator/>
  </insert>
</delta>
```

Here the `dynamic-action` is used to map the method `getPermissions` in `PermissionActionService` when the menuAction is triggered from UI. `dynamic-action` will also have reference to the target UI action to perform using the `rAction`

- src/main/smartview/src/bundles/d2svropview-bundle.js - A portion of this file is used to refer to key RequireJS modules that define the extensions to the toolbar and menu related D2SV UI API.

```
define([
    'd2svropview/utils/theme/action.icons',
    'd2svropview/utils/startup',
    'd2svropview/commands/node.actions.rules',
    'd2svropview/commands/view.permission',
], {});
```

- src/main/smartview/src/commands/view.permission.js - A `CommandModel` that implements the executable logic when a user clicks the `View Permission` menu on the UI. It is an extension of `CallServiceCommand` which is used to take care of the forming the service method request
- src/main/smartview/src/commands/node.actions.rules.js - Defines client side filtering and association logic to attach the above `view.permission.js` command model implementaion to a UI toolbar item.
- src/main/smartview/src/utils/startup.js - Runs as part of D2 Smartview client-side startup flow. This flow is executed everytime endusers reload the D2 Smartview application in their internet browser.
- src/main/smartview/src/extensions.json - A portion of this file registers extensions to the toolbar and menu related D2SV UI API. The corresponding portion is highlighted below

```
  "d2/sdk/controls/action.icons/action.icons": {
    "extensions": {
      "d2svropview": [
        "d2svropview/utils/theme/action.icons"
```

```
      ]
    }
  },
  "d2/sdk/commands/node.actions.rules": {
    "extensions": {
      "d2svropview": [
        "d2svropview/commands/node.actions.rules"
      ]
    }
  },
  "d2/sdk/utils/commands": {
    "extensions": {
      "d2svropview": [
        "d2svropview/commands/view.permission"
      ]
    }
  }
```

The side-panel dialog that displays permissions

- src/main/smartview/src/bundles/d2svropview-bundle.js - A portion of this file is used to refer to key RequireJS modules that define the extensions to dialog used for the side panel as part of the response from the menu action.

```
define([
    ...
    'd2svropview/dialogs/permissions/permissions.dialog'
], {});
```

- src/main/smartview/src/dialogs/permissions/permissions.dialog.js - This dialog will be used to show a stepper wizard view. The stepper wizard will be a single step having `permissions.view`.
- src/main/smartview/src/dialogs/permissions/permissions.view.js - The view will be used to render data as table view. Data returned as part fo the response from the `PermissionActionService` be managed by a `MarionetteJS` `Collection` `permission.collection.js`.
- src/main/smartview/src/dialogs/permissions/impl/table.columns.js - Its a `Backbone JS` collection which is used to map the columns information such as the key,title etc

Example:

```
{
      key: 'base_permission',
      column_key: 'base_permission',
      sequence: 3,
      sort: false,
      definitions_order: 3,
      title: lang.colNameBasePermissions,
      type: -1,
      widthFactor: 0.7,
      permanentColumn: true // don't wrap column due to responsiveness into details row
}
```

- src/main/smartview/src/extensions.json - A portion of this file registers extensions toward collections used in handling the list of permissions. The corresponding portion is highlighted below

```
"d2/sdk/models/module.collection": {
    "modules": {
      "d2svropview": {
        "title": "D2SV-ReadOnlyPermission-View",
        "version": "1.0.0"
      }
    }
  }
```

- src/main/smartview/src/dialogs/permissions/impl/permissions.collection.js - Collection is used to parse the unformatted response data from the `PermissionActionService` to collection of models. This collection is included in `BrowsableMixin` to have filter and sorting capability of the result set.

D2SV Custom Widget Type Tile

D2SV custom widget type tile plugin will help the developer in solving the following scenarios

- Define a custom widget type
- Define a custom widget type parameter
- Define a custom widget type in the landing pange
- Define a shortcut tile with behavior to access the custom widget type parameter.

Instruction to try out the sample

Developer can extract the sample and build it using the workspace assistant. Once built, the distribution is collected in 'dist' folder as **D2SV-ReadOnlyPermission-View-1.0.0.jar** which can placed in `WEB-INF/lib` directory of a deployed D2 Smartview. In the case of D2-Config, it has to be placed in the `WEB-INF/CustomWidgetType`. Also update the plugin details in the `WEB-INF/classes/D2-Config.properties` The application server needs to be restarted post deployment.

As result of deploying this plugin, it will introduce a new widget type in the D2-Config widget types.

Source code structure

```
\---D2SV-Custom-Widget-Type
    |   pom.xml
    |
    +---src
    |   \---main
    |       +---java
    |       |   \---com
    |       |       +---emc
    |       |       |       D2PluginVersion.java
    |       |       |
    |       |       \---opentext
    |       |           \---d2
    |       |               +---rest
```

```
|       |               |   \---context
|       |               |       \---jc
|       |               |               PluginRestConfig_CustomWidgetType.java
|       |               |
|       |               \---smartview
|       |                   \---customwidgettype
|       |                       |   CustomWidgetTypePlugin.java
|       |                       |
|       |                       +---api
|       |                       |       CustomWidgetTypeVersion.java
|       |                       |
|       |                       \---rest
|       |                               package-info.java
|       |
|       +---resources
|       |   |   customwidgettype-version.properties
|       |   |   D2Plugin.properties
|       |   |
|       |   +---smartview
|       |   |       SmartView.properties
|       |   |
|       |   \---strings
|       |       +---actions
|       |       |   \---config
|       |       |       \---modules
|       |       |           \---widget
|       |       |               \---WidgetDialog
|       |       |                       WidgetDialog_en.properties
|       |       |
|       |       \---config
|       |               U4Landing.properties
|       |               WidgetSubtype.properties
|       |               WidgetSubtypelist.properties
|       |
|       \---smartview
|           |   .csslintrc
|           |   .eslintrc-html.yml
|           |   .eslintrc.yml
|           |   .npmrc
|           |   config-editor.js
|           |   customwidgettype.setup.js
|           |   esmhelper.js
|           |   Gruntfile.js
|           |   package.json
```

```
|           |   proxy.js
|           |   server.conf.js
|           |   server.js
|           |

|           +---pages
|           |   |   config-d2.js
|           |   |   config-dev.js
|           |   |   favicon.ico
|           |   |   initialize.js
|           |   |   loader.css
|           |   |   otds.html
|           |   |
|           |   +---debug
|           |   |       app.html
|           |   |
|           |   \---release
|           |           app.html
|           |
|           +---src
|           |   |   component.js
|           |   |   config-build.js
|           |   |   customwidgettype-init.js
|           |   |   extensions.json
|           |   |   Gruntfile.js
|           |   |
|           |   +---bundles
|           |   |       customwidgettype-bundle.js
|           |   |
|           |   +---extensions
|           |   |       custom.type.tile.behaviors.js
|           |   |       custom.type.tiles.js
|           |   |
|           |   +---test
|           |   |       extensions.spec.js
|           |   |
|           |   +---utils
|           |   |   |   startup.js
|           |   |   |
|           |   |   \---theme
|           |   |       |   action.icons.js
|           |   |       |
|           |   |       \---action_icons
|           |   |               action_sample_icon.svg
```

```
    |           |   |
    |           |   \---widgets
    |           |       \---shortcut.tile
    |           |               custom.type.shortcut.behavior.js
    |           |
    |           \---test
    |                   Gruntfile.js
    |                   karma.js
    |                   test-common.js
    |
    \---target
        \---antrun
                build-main.xml
```

Files and their purpose

Following are the list of function oriented source files and their purpose. Other source files present within the plugin are part of common infrastructure code and explained in Understanding D2SV plugin project.

REST Implementations

- pom.xml - Defines the maven project for this plugin.
- src/main/java/com/emc/D2PluginVersion.java - Declares identification information for the entire plugin using `D2SVROPViewVersion` class
- src/main/java/com/opentext/d2/rest/context/jc/PluginRestConfig_CustomWidgetType.java - Java configuration for spring components like REST controller, Beans etc.
- src/main/java/com/opentext/d2/smartview/customwidgettype/CustomWidgetTypePlugin.java - Declares a plugin component for D2FS.
- src/main/java/com/opentext/d2/smartview/customwidgettype/api/CustomWidgetTypeVersion.java - Holder for plugin identification information. Loads relevant data from `d2svropview-version.properties` file resource.
- src/main/java/com/opentext/d2/smartview/customwidgettype/rest/package-info.java - Declares package metadata for JDK and IDE.

opentext™

Custom widget type references needed for the D2-Config widget configuration and D2 Smartview landing page configuration

- src/main/resources/strings/config/U4Landing.properties - Defines the custom widget type which are supported

```
shortcut_types=CustomType
```

- src/main/resources/strings/config/WidgetSubtypelist.properties - Defines the custom widget type

```
CustomType=true
```

- src/main/resources/strings/config/WidgetSubtype.properties - Defines the custom widget type parameter for the custom widget types mentioned which are supported Here the default value for the paramter can also be provided which can be changed in the D2 Config widget configuration

```
CustomType.sample1 = Hello World
```

- src/main/resources/strings/actions/config/modules/widget/WidgetDialog/WidgetDialog_en.properties - This properties file will contain the labels used for parameters for the custom type

```
label_sample1 = Sample text field 1
```

D2 Smartview UI changes for the plugin

- src/main/smartview/src/bundles/customwidgettype-bundle.js - A portion of this file is used to refer to key RequireJS modules that define the extensions shortcut behavior API and click of the widget tile

```
define([
    'customwidgettype/utils/theme/action.icons',
    'customwidgettype/utils/startup',
```

```
    'customwidgettype/extensions/custom.type.tiles',
    'customwidgettype/extensions/custom.type.tile.behaviors'
], {});
```

- src/main/smartview/src/utils/theme/action.icons.js - Defines the default icon for the widgets
- src/main/smartview/src/utils/startup.js - Runs as part of D2 Smartview client-side startup flow. This flow is executed everytime end users reload the D2 Smartview application in their internet browser.
- src/main/smartview/src/extensions.json - A portion of this file registers extensions to enable the custom shortcut tile for the widget configuration and also to have custom shortcut tile behavior.

```
  {
  "d2/sdk/models/module.collection": {
    "modules": {
      "customwidgettype": {
        "title": "D2SV-Custom-Dialogs",
        "version": "1.0.0"
      }
    }
  },
  "d2/sdk/controls/action.icons/action.icons": {
    "extensions": {
      "customwidgettype": [
        "customwidgettype/utils/theme/action.icons"
      ]
    }
  },
  "d2/sdk/utils/landingpage/tiles": {
    "extensions": {
      "customwidgettype": [
        "customwidgettype/extensions/custom.type.tiles"
      ]
    }
  },
  "d2/sdk/widgets/shortcut.tile/shortcut.tile.behaviors": {
    "extensions": {
      "customwidgettype": [
        "customwidgettype/extensions/custom.type.tile.behaviors"
      ]
    }
  }
}
```

- src/main/smartview/src/extensions/custom.type.tiles.js - This will define the custom widget type to the tile containers in the UI

```
define(['d2/sdk/utils/widget.constants'], function(widgetConstants) {
    'use strict';

    function validateConfigCustomType0() {
        var validation = {
            status: true
        };
        // TODO: Validates widgetConfig. Set validation.status = false if the validation
should fail.
        return validation;
    }
    // List of landing tile definitions
    return [{
        type: 'CustomType',
        icon: 'custom-widget-type',
        isShortcut: true,
        tileConfigType: widgetConstants.TileConfigTypes.WIDGET,
        validateConfig: validateConfigCustomType0
    }];
});
```

- src/main/smartview/src/extensions/custom.type.tile.behaviors.js - This will define the custom shortcut behavior to the custom widget type `CustomType`

```
define(['customwidgettype/widgets/shortcut.tile/custom.type.shortcut.behavior'],
function(CustomTypeShortcutBehavior) {
    'use strict';
    return [{
        type: 'CustomType',
        behaviorClass: CustomTypeShortcutBehavior
    }];
});
```

- src/main/smartview/src/widgets/shortcut.tile/custom.type.shortcut.behavior.js -Define custom shortcut behavior with the onclick action to perform. This will prompt the user with the default parameter value configured for the `CustomType`

```
define([
    'd2/sdk/widgets/shortcut.tile/shortcut.tile.behavior'
], function(ShortcutTileBehaviorImpl){
    'use strict';

    var CustomTypeShortcutBehavior = ShortcutTileBehaviorImpl.extend({
        constructor: function CustomTypeShortcutBehavior() {
            CustomTypeShortcutBehavior.__super__.constructor.apply(this, arguments);
        },
        onClick: function() {
            alert('custom parameter sample1 : '+this.options.widgetParams.sample1);
        }
    });

    return CustomTypeShortcutBehavior;
});
```

D2SV client to server logging

D2SV UI uses `log4javascript` to enable logging for UI components. By default, the library is configured to channel log output to web browser console. In the past while debugging for some issue in D2 Smartview, we felt the need to correlate this client-side log output with server-side log output generated by back-end components and usually saved in "D2-Smartview.log" file. Driven by this need, we've created this sample plugin which re-configures the `log4javascript` and channels the log output to the same server-side log file. Key concepts explored in this plugin are

- REST endpoint with un-conventional input/output.
- RequireJS module re-configuration

Instruction to try out the sample

- Build the plugin using `npm run build` from SDK workspace root.
- Copy `D2SV-Client2Server-Logging-1.0.0.jar` from "dist" folder in workspace root and paste it inside `WEB-INF/lib` folder of a deployed D2 Smartview application.
- Edit D2 Smartview logging configuration file "logback.xml" and set the root logging level to `INFO`.
- Restart application server on which D2 Smartview is deployed.
- Reload D2-Smartview application in web-browser with additional query parameter `loglevel=info`.

  > **TIP**
  >
  > Complete URL might look like `https://mydomain.com/D2-Smartview/ui?loglevel=info#d2`

- Open console for the web-browser and check if some INFO level log output is present.
- On the server-side open **D2-Smartview.log** file and check for the same statements as from web-browser console.

Source code structure

```
D2SV-Client2Server-Logging
|
|   pom.xml
|
+---src
|   \---main
|       +---java
|       |   \---com
|       |       +---emc
|       |       |       D2PluginVersion.java
|       |       |
|       |       \---opentext
|       |           \---d2
|       |               +---rest
|       |               |   \---context
|       |               |       \---jc
|       |               |               PluginRestConfig_C2SLogging.java
|       |               |
|       |               \---smartview
|       |                   \---c2slogging
|       |                       |   C2SLoggingPlugin.java
|       |                       |
|       |                       +---api
|       |                       |       C2SLoggingVersion.java
|       |                       |
|       |                       \---rest
|       |                           |   package-info.java
|       |                           |
|       |                           +---api
|       |                           |   |   IClientLogManager.java
|       |                           |   |
|       |                           |   \---impl
|       |                           |           ClientLogManager.java
|       |                           |
|       |                           +---controller
|       |                           |       InboundExternalLogController.java
|       |                           |
|       |                           \---model
|       |                                   HelpModel.java
|       |                                   LogEntry.java
|       |                                   LogLevel.java
|       |                                   LogRequest.java
|       |
```

```
|       +---resources
|       |   |   c2slogging-version.properties
|       |   |   D2Plugin.properties
|       |   |
|       |   \---smartview
|       |           SmartView.properties
|       |
|       \---smartview
|           |   .csslintrc
|           |   .eslintrc-html.yml
|           |   .eslintrc.yml
|           |   .npmrc
|           |   c2slogging.setup.js
|           |   config-editor.js
|           |   esmhelper.js
|           |   Gruntfile.js
|           |   package.json
|           |   proxy.js
|           |   server.conf.js
|           |   server.js
|           |
|           +---pages
|           |   |   config-d2.js
|           |   |   config-dev.js
|           |   |   favicon.ico
|           |   |   initialize.js
|           |   |   loader.css
|           |   |   otds.html
|           |   |   redirect.html
|           |   |
|           |   +---debug
|           |   |       app.html
|           |   |
|           |   \---release
|           |           app.html
|           |
|           +---src
|           |   |   c2slogging-init.js
|           |   |   component.js
|           |   |   config-build.js
|           |   |   extensions.json
|           |   |   Gruntfile.js
|           |   |
|           |   +---bundles
```

```
|           |   |       c2slogging-bundle.js
|           |   |
|           |   +---test
|           |   |       extensions.spec.js
|           |   |
|           |   \---utils
|           |       |   startup.js
|           |       |
|           |       \---theme
|           |           |   action.icons.js
|           |           |
|           |           \---action_icons
|           |                   action_sample_icon.svg
|           |
|           \---test
|                   Gruntfile.js
|                   karma.js
|                   test-common.js
|
\---target
```

Files and their purpose

Following are the list of function oriented source files and their purpose. Other source files present within the plugin are part of common infrastructure code and explained in Understanding D2SV plugin project.

REST Controller

- src/main/java/com/opentext/d2/rest/context/jc/PluginRestConfig_C2SLogging.java - Declares Spring Bean `IClientLogManager` through `ClientLogManager`.
- src/main/java/com/opentext/d2/smartview/c2slogging/rest/api/IClientLogManager.java - Declares log manager interface for REST controllers to use.
- src/main/java/com/opentext/d2/smartview/c2slogging/rest/api/impl/ClientLogManager.java - Log manager implementation that parses and maps input log statements and relays those statements into server side log.
- src/main/java/com/opentext/d2/smartview/c2slogging/rest/controller/InboundExternalLogController.java - Defines two REST endpoints, one receives HTTP POST request with log statements as part

of request body, the other responds to HTTP GET requests with help information on how to use the first endpoint.

- src/main/java/com/opentext/d2/smartview/c2slogging/rest/model/HelpModel.java - Serializable POJO that holds help information.
- src/main/java/com/opentext/d2/smartview/c2slogging/rest/model/LogEntry.java - Serializable POJO that represents a single log statement.
- src/main/java/com/opentext/d2/smartview/c2slogging/rest/model/LogLevel.java - Enum that represents D2SV client-side log levels.
- src/main/java/com/opentext/d2/smartview/c2slogging/rest/model/LogRequest.java - Serializable POJO that holds a bunch of log statements together.

RequireJS module configuration

- src/main/smartview/src/c2slogging-init.js - This file is used to re-configure module `nuc/utils/log` so that it channels log statements to the endpoint created by Java code from above. It also configures `nuc/lib/log4javascript` to customize the request body format sent to the REST endpoint.

  > **INFO**
  >
  > The module `nuc/utils/log` encapsulates the `log4javascript` library and provides managed logging API to D2SV UI components.

opentext™

D2SV Custom Dialogs(D2FS) sample

D2 Custom Dialog sample provide an option to modify the metadata for a document with any available properties page created in D2-Config. As out of the box, the document metadata can be modified only using the properties page which is resolved after configuration matrix against the context.

This sample shows

- How to define a D2 Dialog service plugin which implements ID2fsPlugin.
- How to define a D2FS state method to make dialog chaining as context less. So that the last step Submit will be performed on OOTB property dialog service instead of original dialog service.

Instruction to try out the sample

- Build the plugin using `npm run build` from SDK workspace root.
- Copy `D2SV-Custom-Dialogs-1.0.0.jar` from 'dist' folder in workspace root and paste it inside `WEB-INF/lib` folder of a deployed D2 Smartview application.
- Restart application server on which D2 Smartview is deployed.
- Open D2-Config web application in browser, login and then create few properties page configuration.

Source code structure

```
D2SV-Custom-Dialogs
|   pom.xml
|
+---src
|   \---main
|       +---java
|       |   \---com
|       |       +---emc
|       |       |       D2PluginVersion.java
|       |       |
|       |       \---opentext
```

```
|       |           \---d2
|       |               +---rest
|       |               |   \---context
|       |               |       \---jc
|       |               |               PluginRestConfig_D2SVDialogs.java
|       |               |
|       |               \---smartview
|       |                   \---d2svdialogs
|       |                       |   D2SVDialogsPlugin.java
|       |                       |
|       |                       +---api
|       |                       |       D2SVDialogsVersion.java
|       |                       |
|       |                       +---dialogs
|       |                       |       SelectivePropertyDisplay.java
|       |                       |
|       |                       +---rest
|       |                       |       package-info.java
|       |                       |
|       |                       \---webfs
|       |                           \---dialog
|       |                                   D2DialogServicePlugin.java
|       |
|       +---resources
|       |   |   D2Plugin.properties
|       |   |   d2svdialogs-version.properties
|       |   |
|       |   +---smartview
|       |   |       SmartView.properties
|       |   |
|       |   +---strings
|       |   |   +---dialog
|       |   |   |   \---SelectivePropertyDisplay
|       |   |   |           SelectivePropertyDisplay_en.properties
|       |   |   |
|       |   |   \---menu
|       |   |       \---MenuContext
|       |   |               MenuContext_en.properties
|       |   |
|       |   \---xml
|       |       +---dialog
|       |       |       SelectivePropertyDisplay.xml
|       |       |
|       |       \---unitymenu
```

```
|       |               MenuContextDelta.xml
|       |
|       \---smartview
|           |   .csslintrc
|           |   .eslintrc-html.yml
|           |   .eslintrc.yml
|           |   .npmrc
|           |   config-editor.js
|           |   d2svdialogs.setup.js
|           |   esmhelper.js
|           |   Gruntfile.js
|           |   package.json
|           |   proxy.js
|           |   server.conf.js
|           |   server.js
|           |
|           +---pages
|           |   |   config-d2.js
|           |   |   config-dev.js
|           |   |   favicon.ico
|           |   |   initialize.js
|           |   |   loader.css
|           |   |   otds.html
|           |   |
|           |   +---debug
|           |   |       app.html
|           |   |
|           |   \---release
|           |           app.html
|           |
|           +---src
|           |   |   component.js
|           |   |   config-build.js
|           |   |   d2svdialogs-init.js
|           |   |   extensions.json
|           |   |   Gruntfile.js
|           |   |
|           |   +---bundles
|           |   |       d2svdialogs-bundle.js
|           |   |
|           |   +---dialogs
|           |   |   \---d2fs
|           |   |           context.less.d2fs.state.method.js
|           |   |
```

```
|           |   +---extensions
|           |   |       dialog.state.methods.js
|           |   |
|           |   +---test
|           |   |       extensions.spec.js
|           |   |
|           |   \---utils
|           |       |   startup.js
|           |       |
|           |       \---theme
|           |           |   action.icons.js
|           |           |
|           |           \---action_icons
|           |                   action_sample_icon.svg
|           |
|           \---test
|                   Gruntfile.js
|                   karma.js
|                   test-common.js
|
\---target
```

Files and their purpose

Following are the list of function oriented source files and their purpose. Other source files present within the plugin are part of common infrastructure code and explained in Understanding D2SV plugin project.

Java Classes

- src/main/java/com/opentext/d2/smartview/d2svdialogs/dialogs/SelectivePropertyDisplay.java - Dialog class which implements ID2Dialog to serve the dialog for selecting the properties page configuration.
- src/main/java/com/opentext/d2/smartview/d2svdialogs/webfs/dialog/D2DialogServicePlugin.java - Dialog service class which implements ID2fsPlugin interface for validating the dialog request.

Dialog form definition

- src/main/resources/xml/dialog/SelectivePropertyDisplay.xml - Defines the form structure for rendering "SelectivePropertyDisplay" dialog. The same file will be processed in

"src/main/java/com/opentext/d2/smartview/d2svdialogs/dialogs/SelectivePropertyDisplay.java"

- src/main/resources/strings/dialog/SelectivePropertyDisplay/SelectivePropertyDisplay_en.properties - Label associated with the dialog.

Custom dialog menu configuration in back-end

- src/main/resources/strings/menu/MenuContext/MenuContext_en.properties - Labels associated with the dynamically configured menu.
- src/main/resources/xml/unitymenu/MenuContextDelta.xml - The menu definition file that dynamically adds a new type(MenuContext) of menu for the D2FS `D2MenuService` to discover and return for D2 Smartview.

Dialog state method override

As part of dialog state customization added extension for dialog state methods. This state method will be resolved based on "SelectivePropertyDisplay" dialog name. Intension of having custom dialog state method to override the default behavior of dialog state. With this override dialog state is decoupled between first form and second form.

- src/main/smartview/src/dialogs/d2fs/context.less.d2fs.state.method.js - This is a client side JavaScript file extends "d2/sdk/controls/dialogs/generic/d2fs.state.method". Dialog context is decoupled by having dummy override for "setDialogContextForm()" method.
- src/main/smartview/src/extensions/dialog.state.methods.js - This file is having rule for resolving the dialog state method based on dialog name.
- src/main/smartview/src/extensions.json - Adding the rule for dialog.state.method.

```
"d2/sdk/controls/dialogs/generic/dialog.state.methods": {
  "extensions": {
    "d2svdialogs": [
      "d2svdialogs/extensions/dialog.state.methods"
    ]
  }
}
```

opentext™

Custom Table Cell View sample

Custom Table cell view provides option to render column specific custom cell layout. With this cell view can be visually improved. As out of the box, document modified user column shows information only in text. In this sample modified user information is shown with initials.

This sample shows

- How to define a Custom table cell view implementation.

Instruction to try out the sample

- Build the plugin using `npm run build` from SDK workspace root.
- Copy `D2-CustomTableCell-1.0.0.jar` from 'dist' folder in workspace root and paste it inside `WEB-INF/lib` folder of a deployed D2 Smartview application.
- Restart application server on which D2 Smartview is deployed.
- Open D2-Config web application in browser, login and then create few properties page configuration.

Source code structure

```
D2-CustomTableCell
|   pom.xml
|
+---src
|   \---main
|       +---java
|       |   \---com
|       |       +---emc
|       |       |       D2PluginVersion.java
|       |       |
|       |       \---opentext
|       |           \---d2
|       |               +---rest
```

```
|       |               |   \---context
|       |               |       \---jc
|       |               |               PluginRestConfig_CustomTableCell.java
|       |               |
|       |               \---smartview
|       |                   \---customtablecell
|       |                       |   CustomTableCellPlugin.java
|       |                       |
|       |                       +---api
|       |                       |       CustomTableCellVersion.java
|       |                       |
|       |                       \---rest
|       |                               package-info.java
|       |
|       +---resources
|       |   |   customtablecell-version.properties
|       |   |   D2Plugin.properties
|       |   |
|       |   \---smartview
|       |           SmartView.properties
|       |
|       \---smartview
|           |   .csslintrc
|           |   .eslintrc-html.yml
|           |   .eslintrc.yml
|           |   .npmrc
|           |   config-editor.js
|           |   customtablecell.setup.js
|           |   esmhelper.js
|           |   Gruntfile.js
|           |   package.json
|           |   proxy.js
|           |   server.conf.js
|           |   server.js
|           |
|           +---src
|           |   |   component.js
|           |   |   config-build.js
|           |   |   customtablecell-init.js
|           |   |   extensions.json
|           |   |   Gruntfile.js
|           |   |
|           |   +---bundles
|           |   |       customtablecell-bundle.js
```

```
|           |   |
|           |   +---table
|           |   |   \---cell
|           |   |       \---modified.by
|           |   |           |   modified.by.view.js
|           |   |           |
|           |   |           \---impl
|           |   |                   modified.by.css
|           |   |                   modified.by.hbs
|           |   |
|           |   +---test
|           |   |       extensions.spec.js
|           |   |
|           |   \---utils
|           |       |   startup.js
|           |       |
|           |       \---theme
|           |           |   action.icons.js
|           |           |
|           |           \---action_icons
|           |                   action_sample_icon.svg
|           |
|           \---test
|                   Gruntfile.js
|                   karma.js
|                   test-common.js
\---target
```

Files and their purpose

Following are the list of function oriented source files and their purpose. Other source files present within the plugin are part of common infrastructure code and explained in Understanding D2SV plugin project.

Custom table cell view

- src/main/smartview/src/table/cell/modified.by/modified.by.view.js - Define the custom cell view implementation.
- src/main/smartview/src/table/cell/modified.by/impl/modified.by.hbs - Handlebar template for custom cell view.

- src/main/smartview/src/table/cell/modified.by/impl/modified.by.css - CSS for styling custom cell view.
- src/main/smartview/src/utils/startup.js - Loaded "modified.by.view.js" in "startup.js".

opentext™

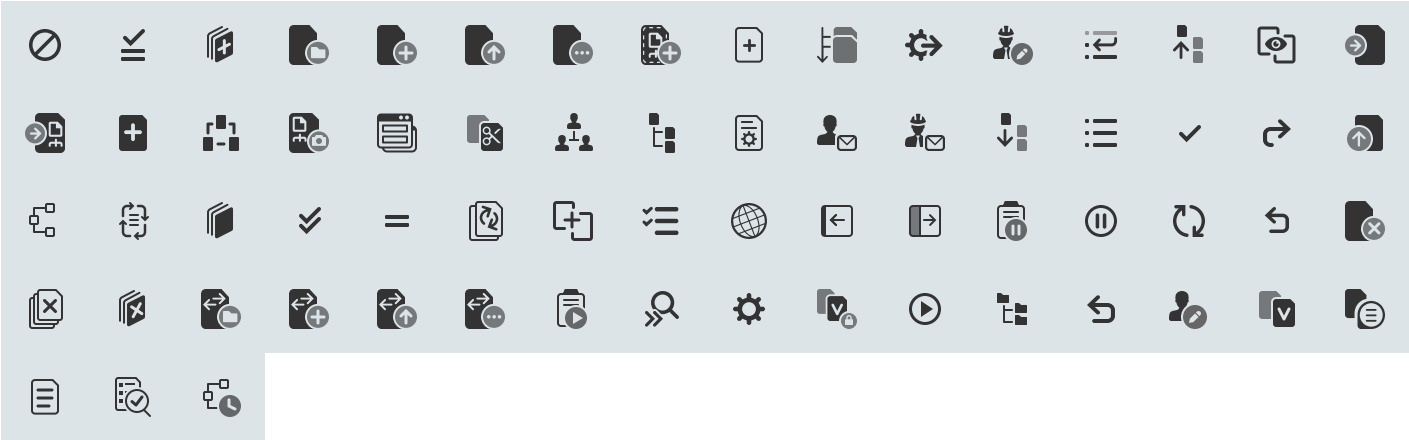
Catalog of built-in action icons

Action icons catalog

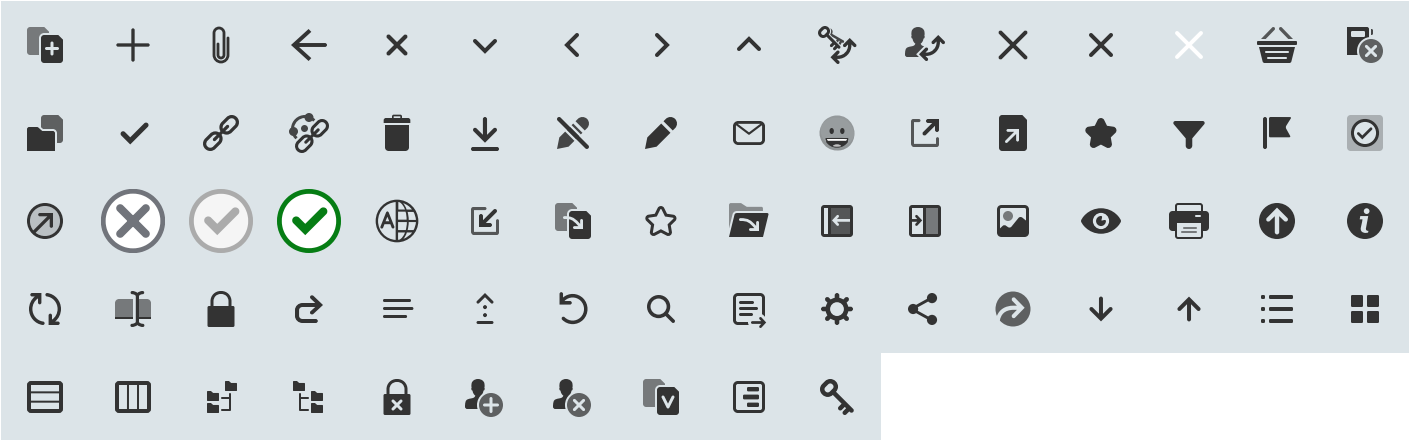
Here is a list of built-in action icons from D2 Smartview runtime and its framework.

> **CAUTION**
>
> Action icons listed under CSUI & SVF may change without notice.

D2 Smartview icons

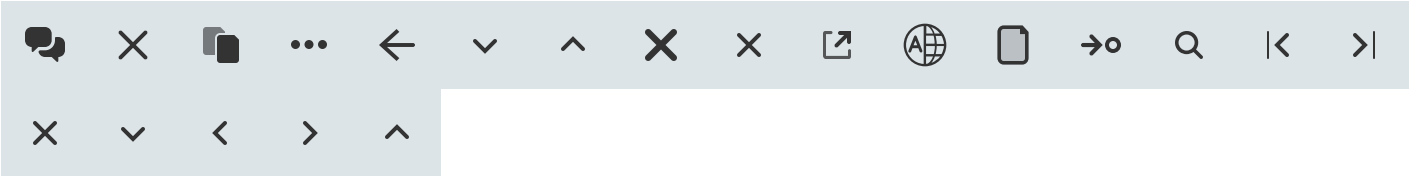


CSUI icons



opentext™

SVF icons

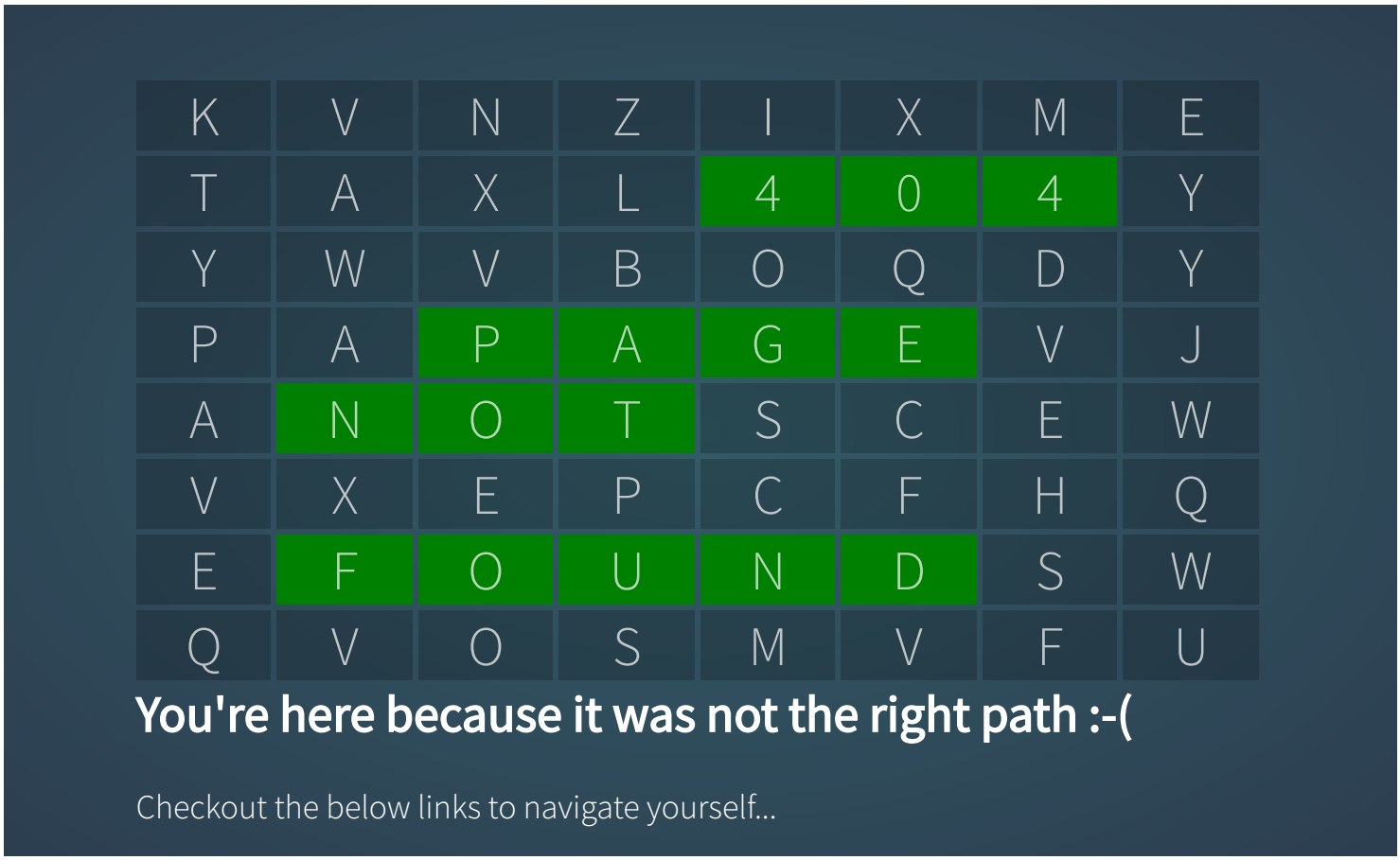


opentext™

D2FS REST services developer guide

D2FS REST services developer guide

This document helps to familiarize one with the existing D2FS REST endpoints and learn about the standards and conventions used in developing those endpoints.



opentext™

Understanding D2SV plugin project

Each D2SV plugin project is a hybrid Maven + NodeJS project having some Java, Javascript and a few static resources as part of its source code. On the outer side, the source code is organized in a Maven project layout and an additional source directory `src/main/smartview` is used to house Javascript source code and directory follows an NPM project structure.

Here we list the directories and files found in a bare-minimum project and outline their purpose. For the purpose of listing, we create a new plugin project by following -

- Execute `npm start` in the workspace root directory to fire up the Workspace Assistant
- Select option **Create a new plugin project**
- Answer the follow-up questions as -
  - Directory name to save this plugin project in: **plugins**
  - Maven group-id of the plugin: **com.opentext.d2.smartview**
  - Name(maven artifact-id) of the plugin: **D2SV-TEST**
  - Version of the plugin: **1.0.0**
  - One liner description of the plugin(shows up everywhere in D2 runtime): **D2SV-TEST**
  - Enter package namespace for the plugin(used as prefix/suffix to generate Java classes & properties, also its lowercase format is used as base Java package name for the plugin & D2SV UI bundle): **D2SVTEST**
  - Include support for D2SV UI: **Yes**

After the assistant runs successfully, it would create a new plugin project in `plugins/D2SV-TEST` directory.

Plugin project layout

```
D2SV-TEST
|
|   pom.xml
```

```
|
+---src
|   \---main
|       +---java
|       |   \---com
|       |       +---emc
|       |       |       D2PluginVersion.java
|       |       |
|       |       \---opentext
|       |           \---d2
|       |               +---rest
|       |               |   \---context
|       |               |       \---jc
|       |               |               PluginRestConfig_D2SVTEST.java
|       |               |
|       |               \---smartview
|       |                   \---d2svtest
|       |                       |   D2SVTESTPlugin.java
|       |                       |
|       |                       +---api
|       |                       |       D2SVTESTVersion.java
|       |                       |
|       |                       \---rest
|       |                               package-info.java
|       |
|       +---resources
|       |   |   D2Plugin.properties
|       |   |   d2svtest-version.properties
|       |   |
|       |   \---smartview
|       |           SmartView.properties
|       |
|       \---smartview
|           |   .csslintrc
|           |   .eslintrc-html.yml
|           |   .eslintrc.yml
|           |   .npmrc
|           |   config-editor.js
|           |   d2svtest.setup.js
|           |   esmhelper.js
|           |   Gruntfile.js
|           |   package.json
|           |   proxy.js
|           |   server.conf.js
```

```
|           |   server.js
|           |
|           +---lib (shortcut to javascript & java libraries)
|           +---node_modules (shortcut to NPM based dependencies used to build/serve the
Javascript portion of code)
|           |
|           +---pages
|           |   |   config-d2.js
|           |   |   config-dev.js
|           |   |   favicon.ico
|           |   |   initialize.js
|           |   |   loader.css
|           |   |   otds.html
|           |   |   redirect.html
|           |   |
|           |   +---debug
|           |   |       app.html
|           |   |
|           |   \---release
|           |           app.html
|           |
|           +---src
|           |   |   component.js
|           |   |   config-build.js
|           |   |   d2svtest-init.js
|           |   |   extensions.json
|           |   |   Gruntfile.js
|           |   |
|           |   +---bundles
|           |   |       d2svtest-bundle.js
|           |   |
|           |   +---test
|           |   |       extensions.spec.js
|           |   |
|           |   \---utils
|           |       |   startup.js
|           |       |
|           |       \---theme
|           |           |   action.icons.js
|           |           |
|           |           \---action_icons
|           |                   action_sample_icon.svg
|           |
|           \---test
```

```
|                   Gruntfile.js
|                   karma.js
|                   test-common.js
|
\---target
```

Files and their purpose

- `/`
  - pom.xml - Maven build file. Used to build the plugin from source code to its distributable format.
- `src/main/java/` - Directory containing all Java source code
  - com/emc/D2PluginVersion.java - Declares identification information for the entire plugin using `D2SVTESTVersion` class.
  - com/opentext/d2/rest/context/jc/PluginRestConfig_D2SVTEST.java - Java configuration for spring components like Beans, Interceptor, Filter etc. This class also declares a component scanner for Spring runtime to automatically load REST Controller and related components.
  - com/opentext/d2/smartview/d2svtest/D2SVTESTPlugin.java - Declares a blanket plugin. Additional code could be put inside this class to implement any functional service plugin.
  - com/opentext/d2/smartview/d2svtest/api/D2SVTESTVersion.java - Holder for plugin identification information. Loads relevant data from `d2svtest-version.properties` file resource.
  - com/opentext/d2/smartview/d2svtest/rest/package-info.java - Declares package metadata for JDK and IDE.
- `src/main/resources/` - Directory containing all plugin related metadata and other static resources
  - smartview/SmartView.properties - Descriptor for D2SV UI runtime. Content of this file is read by D2SV UI runtime and appropriate UI artifacts from this plugin are discovered and linked to the UI.

    > **INFO**

> This file won't be present for the plugins where Smartview UI support is not enabled.

> **DANGER**
>
> Changing content of this file will bring about runtime incompatibility and the UI artifacts will never get discovered by D2SV UI runtime.

- D2Plugin.properties - Another descriptor for D2SV plugin system to identify this plugin separately from other plugins deployed in the same runtime. Content of this file is interpreted as a uniqueue namespace identifier.

  > **DANGER**
  >
  > Changing content of this file will lose the namespace convention used throughout the source code and makes this plugin unmanageable by the SDK as well as D2SV runtime.

- d2svtest-version.properties - Contains name, version and base package for Java classes defined by this plugin. This file is read by `D2SVTESTVersion` class and supplies the metadata information to D2SV plugin system.

  > **DANGER**
  >
  > Changing content of this file may render D2SV runtime to correctly load class files from the deployed plugin jar.

- `src/main/smartview/` - Home of the NPM project that represents the Javascript and related source code for D2SV UI.

  > **DANGER**
  >
  > Even though the source code here is layed out as an NPM project but you should never execute `npm install` command in this directory. Doing so will completely break the setup as the `node_modules` folder is a softlinked directory and managed by the `package.json` from SDK workspace root.

> **TIP**
>
> If for some reason an additional NPM based dependency is required for a plugins UI code, the dependency should be added in the `package.json` from SDK workspace root, subsequently followed by `npm install` command execution in the workspace root directory itself. After that the plugin specific Javascript code can refer to it in usual manner.

- `lib` - Shortcut to the `lib` directory from SDK workspace root. The directory hosts all D2SV UI dependency libraries, that are used while running the Javascript only portion through a NodeJS light-server.
- `node_modules` - Shortcut to the `node_modules` directory from SDK workspace root. The directory hosts all NPM packages used to test, build the Javascript source code and pack those into distributable format, besides serving them through NodeJS server for debugging/testing purposes.
- pages/debug/app.html - HTML page used to run the application with as-is source code using NodeJS server.
- pages/release/app.html - HTML page used to run the application with "built" source code using NodeJS server.
- pages/config-d2.js - Primary `RequireJS` configuration used to run the application using NodeJS server.
- pages/config-dev.js - Additional `RequireJS` configuration to run the application using NodeJS server.
- pages/favicon.ico - Tab icon for the browser to use while accessing the application started using NodeJS server.
- pages/initialize.js - Primary Javascript loaded by the HTML pages to kick-off the bootstrapping of application.
- pages/loader.css - Basic styling using during bootstrap phase of the application.
- pages/otds.html - Token capturer page used when the back-end is configured with OTDS for authentication.
- config-editor.js - Temporary RequireJS configuration override file used while building the UI code.

- d2svtest.setup.js - NodeJS module to setup/re-instate the directory softlinks for `lib` and `node_modules`.
- esmhelper.js - ESM to AMD conversion helper used while running the application using NodeJS server.
- Gruntfile.js - Master task definition file used by `Grunt` while testing/building the source code.
- package.json - NPM package manifest for the UI code.
- proxy.js - Creates local equivalent of a foreign URL through HTTP proxy.
- server.conf.js - Lets configure the remote URL to use as back-end while serving the application through NodeJS.
- server.js - Javascript module to spawn a NodeJS server that supplies D2SV application to the browser.
- .csslintrc - Linter rules for CSS files. Used to validate if all CSS source files meet D2SV standard.
- .eslintrc.yml - Linter rules for Javascript files. Used to validate if all JS source code meet D2SV standard.
- .eslintrc-html.yml - Linter rules for HTML & HBS(Handlebars HTML template) files. Used to validate if matching source files meet D2SV standard.
- .npmrc - Local NPM configuration, used by NodeJS engine before running any NPM/NodeJS scripts.

  > **INFO**
  >
  > All these HTML, JS, JSON, CSS etc files are not part of the actual plugin source code. Some of them facilitate build, test, packaging of the actual source code whereas, others enable serving the entire front-end of D2SV application including this plugin and connects it to a remote back-end for testing/debugging purposes. Any of these files should not be modified.

- src/bundles/d2svtest-bundle.js - Plugin UI bundle, it must contain direct/indirect reference to all AMD modules defined in this plugin so that they are correctly packaged during build.

  > **INFO**
  >
  > AMD modules, that are referred by atleast one other AMD module through `define()`, are called statically referred modules. Dynamically referred AMD modules are never referred

> by any other module using `define()` rather they are referred only through `require()`. The UI bundle must contain direct reference to all dynamically referred modules. It's very important to know that for an entry, all other statically referred modules from that module are automatically added e.g. suppose, A statically refers to B and that statically refers to C but C dynamically refers to D, in this scenario, you should put an entry for A and an entry for C to completely refer all the modules A, B, C, D.

- src/test/extensions.spec.js - Sample unit test. Shows how to write unit tests for each JS module in this plugin.
- src/utils/theme/action_icons/action_sample_icon.svg - A sample icon resource, could be used to represent a menu action. To know more about action icons and how to use them in UI refer to Use icons in D2SV
- src/utils/theme/action.icons.js - Holder of all the action icons. This file is auto-generated everytime the UI code is being built.

  > **TIP**
  >
  > After adding a new svg file in `action_icons` directory, you need to build the UI code once to re-generate this holder file.

- src/utils/startup.js - Hooks to D2SV application startup phase. Generally used to run additional custom logic that has to happen during the startup i.e before any of the UI components starts to render on the page. Provides two hook points, namely `beforeStartup()` and `afterStartup()` their purpose is pretty much self explanatory.
- src/component.js - Declares the UI bundles in this plugin. Used while building the UI code as well as used by `esmhelper.js` while serving the code through NodeJS server. This file normally wouldn't require any change unless the UI source code declares a RequireJS plugin.

  > **CAUTION**
  >
  > At this time only one UI bundle per D2SV plugin is supported by the runtime. So you must not try to split the `d2svtest-bundle.js` file into multiple smaller bundles.

- src/config-build.js - RequireJS configuration used while building the UI code. This file is auto-generated every time the UI code is being built. So any changes done to this file gets

automatically discarded.

- src/d2svtest-init.js - Used to supply additional RequireJS configuration to AMD modules in this plugin. Also can be used to re-configure any other AMD modules defined by the D2SV UI itself or any of its dependencies.

  > **INFO**
  >
  > All the plugins deployed on D2 Smartview do not have a mechanism to specify their loading order. So if multiple plugins try to configure the same AMD module then whichever plugin is loaded last, configuration from that plugin will apply.

- src/extensions.json - Single file to register all the D2SV UI API extensions defined by this plugin.
- src/Gruntfile.js - Task definition file used to build the source code.
- test/Gruntfile.js - Task definition file used to start karma server to run unit test on source code/build output.
- test/karma.js - Karma configuration file used while running unit tests.
- test/test-common.js - Defines pre-conditions and init configuration used to run unit tests.

opentext™

Where to start?

Well, D2 Smartview UI has many UI constructs like command, shotcut tile, list tile, application scope perspective, rest-controller etc. Answer to the question, depends on what you're trying to accomplish.

A good starting point might be to look at the packaged samples. `D2-AdminGroups` sample specifically covers all of the above stated constructs.

After extracting the sample, go thorugh D2 Admin Groups Sample documentation to know about the key concepts and strctures implemented in the sample. Then -

1. Build it once
2. Deploy the compiled artifact in `WEB-INF\lib` folder of a running D2-Smartview
3. Follow How to debug to run the sample in debug mode and put break points(in internet browser's developer console) at at different javascript modules in the project.

Once familiarized, try exploring different workspace assistant options to add new components to the sample or create a fresh plugin project and add it there to see how it works.

> **TIP**
>
> Familiarize yourself with the rest of "How to" topics. The API documentation helps getting to know different parts of the front-end & back-end components.

opentext™

Overview

Overview

The API includes classes, objects, methods & extension points that could be used to enhance/alter existing components of D2SV runtime or write brand new components for it.

The D2SV runtime is comprised of

FRONT-END

Runs on Internet browsers, written using Javascript, HTML, CSS.

Components of the application are loaded using RequireJS framwork that comes bundled with the runtime. A bunch of other open-source libraries are pre-packed as part of the runtime. Below is a list of libraries and their RequireJS module dependency path -

- BackboneJS (nuc/lib/backbone)
- MarionetteJS (nuc/lib/marionette)
- UnderscoreJS (nuc/lib/underscore)
- jQyery (nuc/lib/jquery)
- MomentJS (nuc/lib/moment)
- jQuery Fancy tree (d2/lib/fancytree/jquery.fancytree)
- D3 JS (csui/lib/d3)

  Available requirejs plugins -

  - i18n - Loads a localization module
  - hbs - Loads handlebar template files (*.hbs)
  - json - Loads JSON files (*.json)

- css - Loads CSS files (*.css)

BACK-END

Runs on Web application container, Written in Java.

opentext™

Context

Gathers models, collections, or plain objects to be shared among multiple scenarios and fetch them together. Objects in context are managed by their *factories*.

This is a base class. `PageContext`, `PortalContext`, `BrowsingContext` or `PerspectiveContext` are classes to create instances of.

```
// Create a new context.
var context = new PageContext();
// Get the (main contextual) authenticated user
var currentUser = context.getModel(UserModelFactory);
```

Factory

Is the "overlord" of objects in the context. The parent class returned from 'nuc/contexts/factories/factory' is usually called by different names like `ObjectFactory`, `ModelFactory` or `CollectionFactory` to express what the descended factory will take care of.

- Creates an instance of the object, which will be returned to the caller.
- Assigns a unique prefix to the object, so that the same object can be obtained using the factory at different places.
- Can override how the model or collection is fetched.

A factory has to specify a unique `propertyPrefix` in the prototype and set the object managed by it to `this.property`:

```
var TestObjectFactory = ObjectFactory.extend({

  propertyPrefix: 'test',

  constructor: function TestObjectFactory(context, options) {
```

```
    ObjectFactory.prototype.constructor.apply(this, arguments);

    this.property = new TestObject();
  }

});

// Request an object with the default identifier
// (internally stored with prefix 'test')
var test = context.getObject(TestObjectFactory);
```

Objects are stored using `propertyPrefix` in the context. The `propertyPrefix` is used alone for globally unique objects, or as a base for multiple objects having the same factory, but different attributes:

```
// Request a separate object with a specific identifier
// (internally stored with prefix 'test-id-1')
var test = context.getObject(TestObjectFactory, {
  attributes: {id: 1}
});
```

Factory can be used just for the object creation, if you don't want to learn about its constructor parameters.

```
// Request a standalone object, not shareable by the context
var test = context.getObject(TestObjectFactory, {
  unique: true,
  temporary: true,
  detached: true
});
```

Fetchable Factory

Exposes `fetch` method, which should fetch its model. Whenever the context is fetched, this method will be called.

```

```

```
var FavoriteCollectionFactory = CollectionFactory.extend({

  propertyPrefix: 'favorites',

  constructor: function FavoritesCollectionFactory(context, options) {
    CollectionFactory.prototype.constructor.apply(this, arguments);

    var connector = context.getObject(ConnectorFactory, options);
    this.property = new FavoritesCollection(undefined, {
      connector: connector,
      autoreset: true
    });
  },

  fetch: function (options) {
    return this.property.fetch(options);
  }

});
```

The `isFetchable` method can be added to be able to check dynamically, if the object is fetchable or not.

```
var NodeModelFactory = ModelFactory.extend({

  propertyPrefix: 'node',

  constructor: function NodeModelFactory(context, options) {
    ModelFactory.prototype.constructor.apply(this, arguments);

    var connector = context.getObject(ConnectorFactory, options);
    this.property = new NodeModel(undefined, {connector: connector});
  },

  isFetchable: function () {
    return this.property.isFetchable();
  },

  fetch: function (options) {
    return this.property.fetch(options);
  }
```

```
});
```

Configurable Factory

Factories are usually created once per object type, but they need to be able to create multiple object instances. With just the factory provided, the object will be constructed with default options:

```
// Get the (main contextual) node
var currentNode = context.getModel(NodeModelFactory);
```

With the second argument, additional options can be passed to control the object creation. The `attributes` will be used to uniquely stamp the new object, so future calls to `getObject` with the same attributes will return the same object. Also the `attributes` will be passed to the constructor of the object, if it is a `Backbone.Model`:

```
// Get original where the (main contextual) node points to, if it is
// a shortcut
var originalId = currentNode.get('original_id'),
    original = context.getModel(NodeModelFactory, {
      attributes: {id: originalId}
    });
```

Below the property called like the factory prefix you can pass additional options to the newly created object's constructor by the `options` property:

```
// Get original where the (main contextual) node points to, if it is
// a shortcut and make it fetchable by the connector
var originalId = currentNode.original.get('id'),
    original = context.getModel(NodeModelFactory, {
      attributes: {id: originalId},
      node: {
        options: {connector: currentNode.connector}
      }
    });
```

If the new object is a `Backbone.Model`, you can specify different attributes for the constructor, than the attributes, which control the unique stamp of the object. While the former should be as minimum as to compose the unique stamp, the latter could be more complete to pre-initialize the new object:

```
// Get original where the (main contextual) node points to, if it is
// a shortcut, make it fetchable by the connector, but pre-initialize
// it will all properties available so far
var originalId = node.original.get('id'),
    original = context.getModel(NodeModelFactory, {
      attributes: {id: originalId},
      node: {
        attributes: node.original.attributes,
        options: {connector: currentNode.connector}
      }
    });
```

Finally, if you already have the new object created and you only need the context to make it shareable, you can pass it to the property called like the factory as-is:

```
// Get original where the (main contextual) node points to, if it is
// a shortcut, and share the same object, which has been obtained
// with the contextual node
var originalId = node.original.get('id'),
    original = context.getModel(NodeModelFactory, {
      attributes: {id: originalId},
      node: node.original
    });
```

Detached Objects

Objects, which are added to the context after the context was fetched are needed to be fetched manually, if they need fetching at all. Also, as manually fetched objects, when the context is re-fetched, they are not re-fetched again. Their users decide, when they should be re-fetched.

```
// User information, which does not refresh automatically and will be
// discarded, when clear() is called on the context
```

```
var ownerId = node.get('owner_user_id'),
    owner = context.getModel(MemberModelFactory, {
      attributes: {id: ownerId},
      detached: true
    });
```

Detached objects should merge the `Fetchable` mixin, which allows fetching only once on demand by `ensureFetched`:

```
// Make sure, that the model was fetched once, before accessing
// its properties
owner
    .ensureFetched()
    .done(function () {
      console.log('Login:', owner.get('name'));
    });
```

Permanent Objects

Objects like the authenticated user need not be re-created during the application lifecycle. After being requested for the first time, they should remain in the context for all scenarios. (The only way how to re-create them is to reload the entire application - the application page.)

```
// User information, which does not refresh automatically and will not be
// discarded, when clear() or fetch() is called on the context
var ownerId = node.get('owner_user_id'),
    owner = context.getModel(MemberModelFactory, {
    attributes: {id: ownerId},
    permanent: true,
    detached: true
  });
```

Permanent objects are usually detached too, unless they should be re-fetched with every context re-fetch.

opentext™

Temporary Objects

Objects like the original node need to be shared across function scopes and object boundaries, but should not be re-created and re-fetched multiple times. When the lifecycle of the current (main contextual) node ends, they should be discarded from the context, so that they would not get re-fetched with the new context content.

```
// Shareable original node information, which will be discarded, as soon
// as clear() or fetch() is called on the context
var originalId = shortcut.original.get('id'),
    original = context.getModel(NodeModelFactory, {
      attributes: {id: originalId},
      temporary: true
    });
```

Factory Life-Cycle

The context is a single-instance object that lives as long as the web page lives. (There may be multiple contexts, if parts of the page were supposed to work separately, but that would be a rare case.) The web page serves different purposes during its life. Having just single context instance means that the content of the context has to be able to be exchanged to reflect the current page content.

The context supports two changes of the page content:

- refresh - the page (views) will be reused, only the data will be reloaded
- exchange - the page will be rebuilt (current views will be destroyed and new ones will be created) and new data wil be loaded

These changes can be induced by the following methods of the context: `clear` and `fetch`. The `clear` removes the factories and thus their data from the context. The `fetch` reloads (or loads, initially) the data by letting the factories fetch.

```
// render a new page    <----------------------------------+
context.getObject(...) // get objects from the context     |
context.fetch()        // fetch collected factories  <--+  |
```

```
// work with the page                                  |  |
// open another object on the same page  ---------------+  |
context.clear()        // prepare for the next page        |
// navigate to other page  --------------------------------+
```

If the page has to show a different scenario (exchange), the `clear` will be called, then the page will be rebuilt and eventually the `fetch` will be called to load the data. If the page should show the same scenario with different data (refresh), just `fetch` will be called.

Factories together with the objects that they maintain can be removed from the context when `fetch` and `clear` are called to allow some objects to stay forever and the other objects temporarily only after new data are to be loaded. When factories are used to request objects from context, they can be passed options, or these options can be set to `this.options` in the factory's constructor: `permanent` and `temporary` take care of the life-cycle, `detached` and `unique` have other purposes.

How factories are removed from the context when `clear` and `fetch` are called:

operation / flag	refresh (fetch)	exchange (clear + fetch)
permanent	stay	stay
normal	stay	drop
temporary	drop	drop

The `detached` flag does not affect the factory's life. It prevents the factory ever getting fetched. The `unique` flag appends a unique number to the factory prefix, so that one factory can be put to the context multiple times to maintain different objects.

Declarative options control what factories are allowed to fetch when `fetch` is called. In addition to the static rules below, the actual fetchability is checked by the `isFetchable` method of the context:

method / flag	fetch

method / flag	fetch
permanent	allowed
normal	allowed
temporary	N/A (*)
detached	forbidden

(*) Temporary factories are removed from the context when the `fetch` method starts executing. It does not make sense to discuss their fetchability.

Methods

getObject(factory, options): object

Returns an object maintained by the specified factory. If the object has not existed yet, it will be created, otherwise the previously created instance will be returned.

The object existence is made unique by the property prefix defined by the factory. The full unique property stamp consists of this prefix and of the context `attributes`, which can be passed in the second argument.

If the object is to be created, the second argument can carry parameters for its constructor under the property named by the factory's property prefix; usually `attributes` and `options` for a model or `models` and `options` for a collection. Instead of constructor parameters, this property can point to an already created object, so that the factory just stores it as-is.

The second argument can contain boolean flags to control how the context will handle the object: `detached`, `permanent`, `temporary` and `unique`.

```
// Create a favorite node collection pre-initialized with some nodes
// until it gets fetched with the context
var favorites = context.getCollection(FavoriteCollectionFactory, {
  favorites: {
    models: [{type: 141}, {type: 142}]
  }
});
```

getCollection(factory, options): object

Behaves just like `getObject`, but looks more intuitive, if the expected result is Backbone.Collection.

getModel(factory, options): object

Behaves just like `getObject`, but looks more intuitive, if the expected result is Backbone.Model.

hasObject(factory, options): boolean

Returns if there is an object maintained by the specified factory.

hasCollection(factory, options): boolean

Behaves just like `hasObject`, but looks more intuitive, if the expected object is Backbone.Collection.

hasModel(factory, options): boolean

Behaves just like `hasObject`, but looks more intuitive, if the expected object is Backbone.Model.

clear(options): void

Discards all objects from the context, which are not permanent. When `options.all` is set to `true`, all objects will be discarded.

fetch(options): Promise

Fetches all objects in the context, which are not detached. Discards all temporary objects before that. The options will be passed to the `fetch` methods in factories that take care of the fetchable objects.

suppressFetch(): boolean

Aborts fetching started by the `fetch` method. You can interrupt a running `fetch` in order to start another one, because the earlier result has become irrelevant. (Because a navigation got interrupted by yet another navigation, for example.)

Error event on the context will be never triggered and the returned promise will be never resolved. Sync event will be triggered immediately as the `suppressFetch` method is called to balance the earlier triggered `request` event. Events on the models and and collection will be triggered eventually, as their AJAX calls will finish.

This method does not abort the operation. It only allows another call to `fetch` be made and replace the one in progress.

Properties

fetching: Promise

The promise returned by `fetch` during fetching or `null` if no fetching is in progress.

fetched: boolean

`true` if the most recent `fetch` succeeded, `false` if the context has not been fetched yet, or fetching is in progress, or it failed.

error: Error

`null` if the most recent `fetch` succeeded, or fetching is in progress, or the context has not been fetched yet, an instance of `Error` if the most recent `fetch` failed.

opentext™

Events

'before:clear', context

The context is going to be cleared.

'clear', context

The context has been cleared.

'request', context

The context is going to be fetched.

'sync', context

Fetching the context succeeded.

'error', error, context

Fetching the context failed.

'add:factory', context, propertyName, factory

A new factory has been added to the context.

'remove:factory', context, propertyName, factory

A factory has been destroyed and will be removed from the context.

opentext™

Context Fragment

The context fragment can be used to fetch data for a dynamically added widget, instead of fetching the whole context, which would re-fetch data for widget created earlier.

```
// Subscribe a context fragment to the context, before
// a new widget is constructed and rendered.
contextFragment = new ContextFragment(context);
// Create the widget and render it to get the new models
// added to the context and to the context fragment too.
...
// Fetch only the new models. The new widget will update
// the displayed information as it is needed.
contextFragment.fetch();
// Unsubscribe the context fragment from the context,
// when it is not needed any more.
contextFragment.destroy() // Unsubscribe the fragment.
```

Details

The context supports two scenarios for changing the page content:

- refresh - the page (views) will be reused, only the data will be reloaded
- exchange - the page will be rebuilt (current views will be destroyed and new ones will be created) and new data wil be loaded

There is one more scenario, which you may see on the page:

- grow - new content (views) will be added to the page, which needs to load a new data, but the old data do not need to be reloaded

New views usually load the new data by `ensureFetched` and the context does not need to be involved in fetching the data. However, shared components might be used to add the new content, which

depend in the context to load their data. Because only the owning view knows what part of the context will have to be fetched, it is responsible for collecting a fragment of factories for fetching:

```
// render a new page
context.getObject(...) // get objects from the context
context.fetch()        // fetch collected factories
// work with the page   <-------------------------------+
// introduce new content to the page                    |
new ContextFragment(context) // remember new objects    |
context.getObject(...)       // add other objects       |
contextFragment.fetch()      // load new data           |
contextFragment.destroy()    // stop context watching  --+
```

No factories are removed, when a context fragment is fetched and destroyed:

operation / flag	refresh (fetch)	exchange (clear + fetch)	grow (fragment fetch + destroy)
permanent	stay	stay	stay
normal	stay	drop	stay
temporary	drop	drop	stay

The fetchability of factories follows the rules which the context declared. In addition to the static rules below, the actual fetchability is checked by the `isFetchable` method of the context:

method / flag	fetch	fragment fetch
permanent	allowed	allowed
normal	allowed	allowed
temporary	N/A (*)	forbidden

opentext™

method / flag	fetch	fragment fetch
detached	forbidden	forbidden

(*) Temporary factories are removed from the context when the `fetch` method starts executing. It does not make sense to discuss their fetchability.

Methods

constructor(context)

Start watching the original context for new factories.

fetch(options): Promise

Fetches all objects in the context fragment, which are fetchable by their originating context. The options will be passed to the `fetch` methods in factories that take care of the fetchable objects.

clear(): void

Discards all objects from the context fragment. The context fragment remains subscribed to the context.

destroy(): void

Stops watching the original context for new factories. The context fragment will not be usable any more.

Properties

fetching: Promise?

The promise returned by `fetch` during fetching or `null` if no fetching is in progress.

fetched: boolean

`true` if the most recent `fetch` succeeded, `false` if the context has not been fetched yet, or fetching is in progress, or it failed.

error: Error

`null` if the most recent `fetch` succeeded, or fetching is in progress, or the context has not been fetched yet, an instance of `Error` if the most recent `fetch` failed.

Events

'request', context

The context fragment is going to be fetched. This event is triggered on the original context too. The `fetching`, `fetched` and `error` properties on the original context are not modified.

'sync', context

Fetching the context fragment succeeded. This event is triggered on the original context too. The `fetching`, `fetched` and `error` properties on the original context are not modified.

'error', error, context

Fetching the context fragment failed. This event is triggered on the original context too. The `fetching`, `fetched` and `error` properties on the original context are not modified.

'add:factory', context, propertyName, factory

A new factory has been added to the context fragment.

'before:clear', context

The context fragment is going to be cleared.

opentext™

'clear', context

The context fragment has been cleared.

'destroy', context

The context fragment has been destroyed.

opentext™

PageContext

The simplest context, which can include and fetch models and collections, but does not provide any other functionality. If you use it with widgets, which expect changes based on their context-changing models, you will have to handle these changes yourself.

```
csui.require([
  'nuc/widgets/shortcut/shortcut.view',
  'nuc/contexts/page/page.context',
  'nuc/contexts/factories/next.node',
  'nuc/lib/marionette'
], function (ShortcutView, PageContext, NextNodeModelFactory, Marionette) {
  'use strict';

  var context = new PageContext(),
      nextNode = context.getModel(NextNodeModelFactory),

      region = new Marionette.Region({
        el: '#content'
      }),
      view = new ShortcutView({
        context: context,
        data: {
          type: 141
        }
      });

  // Perform some action if the widget triggered contextual node change
  nextNode.on('change:id', function () {
    alert('Node ID:' + nextNode.get('id'));
  });

  region.show(view);
  context.fetch();

});
```

opentext™

Plugins

Plugins descended from `ContextPlugin` (nuc/contexts/context.plugin) can be registered. They will be constructed and stored with the context instance. They can override the constructor and the method `isFetchable(factory)`.

opentext™

SynchronizedContext

A specialized derivation of Context. SynchronizedContext synchronizes the process of setting (set/reset) the fetched data on the participating objects (models or collections).

The fetches are performed on cloned models or collections, to which no marionette views listen for changes and therefore render is not triggered when these fetch calls return with new data from the server. When all fetches are completed, the data from the cloned models or collections are "copied" to the original models or collections by calling 'set' or 'reset'. This happens in a synchronous loop for all models and collections. This also means that views bound to the participating models or collections are rendered in a single javascript thread (as long as none of the views do something asynchronously - marionette views don't) and when that is finished the browser gets back control and starts displaying the updated DOM.

Note: due to the fact that the original models or collections are not fetched, it must not expected that 'request' or 'sync' gets triggered. Views must rely only on set/change/reset events.

It is also important that the clone method of the participating models or collections clone all necessary attributes to be able to fetch the correct data from the server.

For backward compatibility reason it is possible to enable triggering 'request' and 'sync' on the source context, which makes it easier for example to enable and disable blocking view.

Example:

```
    var context = new PageContext();
    var container = context.getModel(NodeModelFactory, {node: {attributes: {id: 2000}}});
    var childrenCollection = context.getCollection(Children2CollectionFactory,
        {options: {node: container}}
    );

// Create a new synchronized context:
// Specify the source context, which has already the models or collections
// that are listed in the second parameter.
// options in the third parameter:
```

```
// triggerEventsOnSourceContext: if true, request and sync are triggered
// on the 'source' context (the context specified in the first parameter)
  synchronizedContext = new SynchronizedContext(
      context,
      [container, childrenCollection],
      {triggerEventsOnSourceContext: true});

  synchronizedContext.fetch()
      .then(function () {
        console.log("Context fetched - all models/collections are fetched");
      })
      .catch(function (reason) {
        console.log("Context fetch failed:");
        console.log(reason);
      })
```

A complete working example is `pages/debug/synchronized_context.html````.

opentext™

I18n

Carries language settings and loads language modules for the selected locale.

TODO: Write the documentation.

Accept-Language in AJAX Calls

Smart UI has always set the chosen UI language to the `Accept-Language` header, when making AJAX calls via `Connector`. It ensures a consistent language of static assets (language pack) and the data (REST API responses). The UI language is chosen by the `locale` setting:

```
require(['i18n'], function (i18n) {
  console.log('locale:', i18n.settings.locale);
});
```

If you want to use a different locale for the data, you can set the property `acceptLanguage`, which will be sent to the server instead of `locale`:

```
require.config({
  config: {
    i18n: {
      locale: 'en-US',
      acceptLanguage: 'en-AU'
    }
  }
});
```

The values of both `locale` and `acceptLanguage` have to comply with BCP 47. They are case-insensitive and Smart UI will normalize them to lower-case before using them for loading static assets or sending in the `Accept-Language` header.

opentext™

CS uses the property `i18n.settings.userMetadataLanguage` to support multi-lingual UI. Unfortunately, this property does not follow BCP 47. If `userMetadataLanguage` is set and `acceptLanguage` is unset, Smart UI will convert the value of `userMetadataLanguage` to `acceptLanguage`. If both `acceptLanguage` and `userMetadataLanguage` are unset, Smart UI will set the value of `locale` to `acceptLanguage` to stay compatible with previous versions.

Setting `acceptLanguage` to `null` will stop Smart UI from setting the `Accept-Language` header in AJAX calls:

```
require.config({
  config: {
    i18n: {
      locale: 'en',
      acceptLanguage: null
    }
  }
});
```

opentext™

RequireJS

This document describes changes to the original RequireJS. See the RequireJS website for the original documentation.

Changes

1. Make the `pkgs` configuration object mergeable.
2. Add an attribute `data-csui-required` to to every element added to document head.
3. Recognise `rename` in the configuration to implement module name mapping in addition to the `starMap` for an additional module compatibility layer.
4. Return module configuration merged from the mapped original names and new ones.
5. Introduce a method `moduleConfig(id)` on the local require function.

`pkgs` mergeable

Allows calling `require.config({ pkgs: ... })` multiple times to configure packages step-by-step. Kind of misused by the original version of the mobile app to remap modules.

Attribute `data-csui-required`

Allows detecting all scripts and links inserted by RequireJs and the `css` plugin to `document.head` to be able to remove them later. Used to wipe out all modules loaded from one server, before another version can be loaded from a different server.

`rename` map

The RequireJS `starMap` can be used for remapping modules to be able to load different functionality on different pages. It can be used to implement product-specific features, if a RequireJS library is reused in

multiple products.

Another need for module remapping comes from refactoring, which moves a module to a different library, with or without deprecating the original module name. If a module needs to be remapped for compatibility, which was earlier remapped for product adaptation, the two map entries will conflict.

A direct conflict means that either the compatibility mapping, or the product adaptation will not work, depending on the order of the configuration statements:

```
csui/original -> nuc/moved      // keep compatibility with a moved module
csui/original -> custom/adapted // adapt a module for a new product
```

An direct conflict means that dependencies on the original module will not be adapted, if the adaptation maps only the new module name, because the `starMap` is not processed recursively:

```
csui/original -> nuc/moved      // keep compatibility with a moved module
nuc/moved     -> custom/adapted // adapt a module for a new product
```

The `rename` map is separate from `starMap` and solves the direct conflict. Modules remapped for compatibility are called "renamed" and have to be added to the `rename` map. The `starMap` continues to support product adapting. The module name normalisation makes use of both maps. If the `rename` map is configured alone, it will work like `starMap` alone.

When this configuration is used:

```
rename:  csui/original -> nuc/moved
```

The module names will be normalised like this:

```
csui/original -> nuc/moved // using rename
nuc/moved                  // just loaded
```

When this configuration is used:

```
rename:  csui/original -> nuc/moved
starMap: csui/original -> custom/adapted
```

The module names will be normalised like this:

```
custom/adapted                  // just loaded
csui/original -> custom/adapted // using starMap
nuc/moved     -> custom/adapted // using rename backwards and starMap
```

When this configuration is used:

```
rename:  csui/original -> nuc/moved
starMap: nuc/original  -> custom/adapted
```

The module names will be normalised like this:

```
custom/adapted                  // just loaded
csui/original -> custom/adapted // using rename backwards and starMap
nuc/moved     -> custom/adapted // using starMap
```

Merged module configuration

This is a feature supporting module renaming and remapping as discussed in the previous chapter.

An example of a situation:

1. The original module csui/original supported configuration.
2. The original module was adapted in a new product and the new module might need additional configuration.
3. The original module was renamed in the library and parts of the configuration started to be set using the new name.

An example of RequireJS configuration:

```
rename:  csui/original -> nuc/moved
starMap: csui/original -> custom/adapted
```

The result of `module.config()` called in custom/adapted will contain an object merged from configurations set for all three module names. Forward and backward `rename` map and `starMap` are used to discover the other module names.

`moduleConfig` method

The configuration of a RequireJS module may be needed in another module. It can be used to keep compatibility after refactoring the module tree. The functionality of `require.moduleConfig` is similar to `module.config`, the difference is that you have to supply the module name:

```
define(['require', 'module'], function (require, module) {
  // merge the old and new module configurations
  var config = _.extend({},
    require.moduleConfig('other-module'), // configuration of other module
    module.config()                       // configuration of this module
  );
});
```

opentext™

Browsable Support for Backbone.Collections

Browsing means paging, sorting and filtering through a collection of items. These can be large and thus the Backbone.Collection might contain only a part of it in memory at a time. The Backbone.View showing the collection is supposed to use an extended interface to get the right "window" on the full collection.

Paging can be *discreet*, where the window is limited by a starting item index and an item count. The view usually presents a pagination control to set up the browsing state, dealing either in items or pages - "batches" of items with the same size.

Paging can also be *continuous*, which starts at the beginning and continues by loading a specified item count at a time and appending the items to the growing in-memory collection. The view usually calls the next fetch when the scrollbar hits the item threshold, performing an apparent "infinite scrolling".

The collection browsing support provided by the modules below works like this:

1. Set up the browsing state
2. Listen for the standard Backbone Collection and Model events to get notified when the (part of the) collection has been fetched
3. Fetch the collection; the specified part will be placed in the collection
4. Adjust the browsing state and repeat the step 3

This browsing support is based on the standard `fetch` method and the standard events (`add`, `remove`, `reset`, `sync`). Depending on the server capabilities, the fetched items may or may not be up-to-date when the fetch finishes. The `reload: true` option can be passed to the `fetch` call if fresh data are needed and the possible performance penalty is acceptable.

Samples of discreet fetching; if your collection uses concept of pages, you have to convert them to item indexes for the collection interface:

```
// Fetch the first page of 10 items
collection.setSkip(0, false);
```

```
collection.setTop(10, false);
collection.fetch();

// Fetch only the second page of 10 items
collection.setSkip(10, false);
collection.fetch();

// Fetch only another page of 10 items
collection.setSkip(collection.skipCount + 10, false);
collection.fetch();

// Fetch only another page of top items
collection.setSkip(collection.skipCount + collection.topCount, false);
collection.fetch();

// Check if more pages are available for fetching
if (collection.length < collection.totalCount) {
  ...
}

// Reload the collection and start browsing from the beginning
collection.setSkip(0, false);
collection.fetch({reload: true});
```

Samples of continuous fetching:

```
// Fetch the first 10 items
collection.setSkip(0, false);
collection.setTop(10, false);
collection.fetch();

// Fetch and append the second 10 items
collection.setSkip(10, false);
collection.fetch({
  remove: false,
  merge: false
});

// Fetch and append another 10 items
collection.setSkip(collection.skipCount + 10, false);
collection.fetch({
  remove: false,
```

```
  merge: false
});

// Fetch and append another top items
collection.setSkip(collection.skipCount + collection.topCount, false);
collection.fetch({
  remove: false,
  merge: false
});

// Check if more items are available for fetching
if (collection.length < collection.totalCount) {
  ...
}

// Reload the collection and start from the first 10 items again
collection.setSkip(0, false);
collection.fetch({reload: true});
```

Browsing Support Modules

nuc/models/browsable/browsable.mixin : extends the interface of Backbone.Collection with properties and methods supporting the browsing functionality

nuc/models/browsable/client-side.mixin : overrides the interface of Backbone.Collection, which can be fetched only completely, to support the browsing functionality on the client side

nuc/models/browsable/v1.request.mixin : provides serialization of the URL parameters for the browsing functionality using the concepts of the `/api/v1/nodes/:id/nodes`

nuc/models/browsable/v1.response.mixin : provides deserialization of the collection state and collection items using the concepts of the `/api/v1/nodes/:id/nodes`

nuc/models/browsable/v2.response.mixin : provides deserialization of the collection state and collection items using the concepts of the `/api/v2/members/favorites`

Examples

opentext™

Collection of node (container) children returned by `/api/v1/nodes/:id/nodes`, which supports paging, sorting and filtering:

```
var NodeChildrenCollection = Backbone.Collection.extend(_.defaults({

  model: NodeModel,

  constructor: function NodeChildrenCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);

    this.makeNodeResource(options)
        .makeBrowsable(options)
        .makeBrowsableV1Request(options)
        .makeBrowsableV1Response(options);
  },

  url: function () {
    var query = this.getBrowsableUrlQuery();
    return Url.combine(this.node.urlBase(),
        query ? '/nodes?' + query : '/nodes');
  },

  parse: function (response, options) {
    this.parseBrowsedState(response, options);
    return response.data;
  }

}, NodeResourceModel(Backbone.Collection)));

BrowsableMixin.mixin(NodeChildrenCollection.prototype);
BrowsableV1RequestMixin.mixin(NodeChildrenCollection.prototype);
BrowsableV1ResponseMixin.mixin(NodeChildrenCollection.prototype);
```

Collection of favourite nodes returned by `/api/v2/members/favorites`, which does not support paging, sorting and filtering:

```
var FavoritesCollection = Backbone.Collection.extend(_.defaults({

  model: NodeModel,
```

```
  constructor: function FavoritesCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);

    this.makeConnectable(options)
        .makeFetchable(options)
        .makeClientSideBrowsable(options);
  },

  url: function () {
    var url = this.connector.connection.url.replace('/v1', '/v2');
    return Url.combine(url, 'members/favorites');
  },

  parse: function (response, options) {
    return response.results;
  }

}, ConnectableModel(Backbone.Collection), FetchableModel(Backbone.Collection)));

ClientSideBrowsableMixin.mixin(NodeChildrenCollection.prototype);
BrowsableV2ResponseMixin.mixin(NodeChildrenCollection.prototype);
```

Continuous paging through favorites by "batches" of 10 items and logging the current collection length:

```
var connector = ...,
    favorites = new FavoriteNodeCollection(undefined, {
      connector: connector,
      top: 10
    });

// The top parameter can be set by `favorites.setTop(10, false)` too

favorites
  .fetch()
  .then(fetchNext);

function fetchNext() {
  console.log(favorites.length);
  if (favorites.length < favorites.totalCount) {
    return favorites
      .fetch({
        remove: false,
```

```
        merge: false
      })
      .then(fetchNext);
  }
}
```

Loading 10 children from the position 20 in their parent container and logging the actual collection length:

```
var node = ...,
    children = new NodeChildrenCollection(undefined, {
      node: node,
      skip: 20,
      top: 10
    });

// The skip and top parameters can be set by `children.setSkip(20, false)`
// and `children.setTop(10, false)` too

children
  .fetch()
  .done(function () {
    console.log(children.length);
  });

// Other "window" to the collection can be fetched by adjusting the limit
// using `children.setSkip(..., false)` and `children.setTop(..., false)`
// and repeating the `fetch` call.
```

Getting the most recent 5 documents from a folder, which means filtering the node (container) children by type including only documents and e-mails, sorting them by the last modification time in the descending direction (and by name, ascending if multiple were modified at the same time), limiting them just to the first 5 and logging their names:

```
var node = ...,
    children = new NodeChildrenCollection(undefined, {
      node: node,
      filter: {
        type: [144, 748]
```

```
      },
      orderBy: 'modify_date desc, name'
      top: 5
    });

// The parameters can be set by `setFilter, `setOrder` and `setTop` too

children
  .fetch()
  .done(function () {
    console.log(children.pluck('name').join(', '));
  });

// Another 5 items can be fetched by `fetch({remove: false, merge: false})`.  The
// item count to fetch can be adjusted by `children.setTop(..., false)` before it.
```

Remarks

Collections extended by either `BrowsableMixin` or `ClientSideBrowsableMixin` offer the same interface for both discreet and continuous paging. You can use them interchangingly without knowing which one you have.

`ClientSideBrowsableMixin` fetches the complete collection to an internal buffer at first, then it populates the collection according to the browsing state properties. Following fetches are served from the internal buffer. `BrowsableMixin` needs to be supported by the REST API on the server side. If you need to refresh the collection from the server independently on the client-side and server-side collection, pass the `reload: true` option to the `fetch` method. You usually start from the beginning too, by calling `setSkip(0, false)` right before the `fetch({reload: true})`.

opentext™

BrowsableMixin

Defines the interface of collection paging, sorting and filtering and provides an implementation of the collection state supporting them.

Using the browsing state properties to populate the collection is supposed to be added on your own or provided by other mixins.

Example

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);

    this.makeBrowsable(options);
  }

  // use the browsing properties to maintain the collection content

});

BrowsableMixin.mixin(MyCollection.prototype);
```

makeBrowsable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Collection constructor options passed in.

Browsing State Properties

skipCount : how many items should be skipped from the beginning of the collection; zero-based index of the first item (integer, 0 by default)

topCount : how many items should be fetched from top of the collection, starting from the `skipCount`; all the rest of the collection will be returned if not specified (integer, undefined by default)

orderBy : how to sort the fetched items; comma-delimited list of space-delimited sort criteria consisting of the property name and an optional sort direction, which defaults to ascending: `<name> [asc|desc], ...` (string, undefined by default)

filters : how to filter the fetched items; map of filer criteria with property names as keys and filert operands as values (object literal; empty by default) name: 'pro', type: [144, 848])

actualSkipCount : how many items were actually skipped when the server handled the request the caller should check this value and optionally correct the `skipCount` for future calls (integer, output only, set after fetching)

totalCount : how many total items are available for paging through (integer, output only, set after fetching)

totalCountComputed : set to `true` if the exact total count is not available, if the `totalCount` was computed only to allow random pagination and rounded to the page size boundary (may not be supported by all server adaptors)

Browsing State Changing Methods

```
setSkip(skip, fetch)            : boolean
setTop(top, fetch)              : boolean
setLimit(skip, top, fetch)      : boolean
resetLimit(fetch)               : boolean
setOrder(orderBy, fetch)        : boolean
resetOrder(fetch)               : boolean
setFilter(name, value, options) : boolean
clearFilter(fetch)              : boolean
```

Set the browsing state properties to the specified value or reset it to their default. The last `fetch` argument makes the collection fetch automatically if not set to `false` (`true` is the default). If the new browsing state requires calling a `fetch` to update the collection, `true` is returned, otherwise `false`.

The `setLimit` with `resetLimit` and `setOrder` with `resetOrder` work as pairs - the first sets (or overwrites) the current value, the second resets it to the default one. The `setFilter` with `clearFilter` work as a pair - the first sets (adds or overwrites) a property to the filter, the second clears all filters. The `options` parameter in `setFilter` can be either a `boolean` with the meaning of the `fetch` parameter of the other methods, or an object literal with `fetch` and `clear` properties. The `clear` property clears the existing filters beforeit sets the new.

Browsing Methods

These methods are supposed to be reused from Backbone, implemented on your own or by other mixin, like `ClientSideBrowsableMixin`.

fetch(options) : promise

Populates the collection according to the current browsing state. If the `reload: true` option is set, the fresh content is ensured by forcing a server call, if the method did not need to perform one.

A special scenario - continuously fetching the next page - can be implemented by incrementing the `skipCount` and fetching the items for appending them to the collection only:

```
collection.setSkip(collection.skipCount + collection.topCount, false);
collection.fetch({
  remove: false,
  merge: false
});
```

If the continuous fetching reached its end should be checked by testing the collection length, for example:

```
if (collection.length < collection.totalCount) {
  ... // fetch the next page
}
```

The `totalCount` is available first after the very first `fetch` call.

If you want to start fresh from the beginning in the middle of continuous fetching, change paging, sorting and filtering parameters and force removing the existing models:

```
collection.setSkip(0, false);
collection.fetch({
  remove: true,
  merge: false
});
```

You could add `reset: true` to optimize UI refresh; instead of multiple `add` events you would get a single `reset` event. You would drop the `remove` and `merge` parameters then, because the collection would be emptied.

See Also

`ClientSideBrowsableMixin`

opentext™

ClientSideBrowsableMixin

Implements paging, sorting and filtering on the client side using the collection state of the `BrowsableMixin`. The `BrowsableMixin` is included in this mixin and applied too.

The first `fetch` fills an internal buffer at first, then it populates the collection according to the browsing state properties. Following fetches are served from the internal buffer.

Request URL formatting and response parsing is supposed to be added by other mixins, according to the specifics of the server resource.

Example

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);

    this.makeClientSideBrowsable(options);
  },

  url: function () {
    // format the request URL fetching the complete collection
  },

  parse: function (response, options) {
    // return the response containing the complete collection
  }

});

ClientSideBrowsableMixin.mixin(MyCollection.prototype);
```

makeClientSideBrowsable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Collection constructor options passed in. It calls `makeBrowsable` from `BrowsableMixin` too.

fetch(options) : promise

Behaves like the original method, just populating the collection from an internal buffer, filled on the very first fetch, according to the current browsing state. If the `reload: true` option is set, the internal buffer is refreshed by a server call.

A special scenario - continuously fetching the next page - can be implemented by incrementing the `skipCount` and fetching the items for appending them to the collection only:

```
collection.setSkip(collection.skipCount + collection.topCount, false);
collection.fetch({
  remove: false,
  merge: false
});
```

If the continuous fetching reached its end should be checked by testing the collection length, for example:

```
if (collection.length < collection.totalCount) {
  ... // fetch the next page
}
```

The `totalCount` is available first after the very first `fetch` call.

If you want to start fresh from the beginning in the middle of continuous fetching, change paging, sorting and filtering parameters and force removing the existing models:

```
collection.setSkip(0, false);
collection.fetch({
  remove: true,
  merge: false
});
```

You could add `reset: true` to optimize UI refresh; instead of multiple `add` events you would get a single `reset` event. You would drop the `remove` and `merge` parameters then, because the collection would be emptied.

populate(models, options) : promise

Behaves like the `fetch` method, but populates the collection from an explicit array of objects or models. All models will be pushed to the internal buffer and the collection will be populated according to its filtering, sorting and paging parameters.

You could add `reset: true` to optimize UI refresh; instead of multiple `add` events you would get a single `reset` event.

add(models, options) : models

remove(models, options) : models

reset(models, options) : models

set(models, options) : models

Behave like original methods, just affecting both the collection and the internal buffer. The `reset` method can be used to fill the collection without a server-side `fetch`. Subsequential `fetch` will use the internal buffer to populate the collection.

A special scenario - emptying the collection, but not the internal buffer - can be performed by calling `reset` with the `populate` set to `false`.

You could add `reset: true` to optimize UI refresh; instead of multiple `add` events you would get a single `reset` event.

opentext™

compareObjectsForSort(property, left, right) : -1|0|1

Can be overridden to modify the default ordering implementation, which handles JavaScript primitive values by applying the comparison operators to them. Strings are compared using grammar rules of the selected locale (i18n).

The `left` and `right` parameters contain `Backbone.Model`s to compare and the `property` contains the attribute name from the models. The result has to be implemented according to `strcmp`:

- -1 if left < right
- 1 if left > right
- 0 if left = right

The overridden method can handle only special attributes and can rely on the original implementation for the rest. For example:

```
var originalCompare = MyCollection.prototype.compareObjectsForSort;
MyCollection.prototype.compareObjectsForSort = function (property, left, right) {
  if (property === 'name') {
    left = left.get(property);
    right = right.get(property);
    if (left < right) {
      return -1;
    } else if (left > right) {
      return 1;
    }
    return 0;
  }
  return originalCompare.call(this, property, left, right);
};
```

See Also

`BrowsableMixin`

BrowsableV1RequestMixin

Serializes URL and parses response according to the V1 of the REST API (`/api/v1/nodes/:id/nodes`), for example) using the collection state maintained by `BrowsableMixin`. The `BrowsableMixin` or a mixin including it must be applied with this mixin together.

Browsing implementation and response parsing is supposed to be added by other mixins, according to the specifics of the server resource.

Example

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);

    this
      .makeBrowsable(options)
      .makeBrowsableV1Request(options);
  },

  url: function () {
    // use `getBrowsableUrlQuery` to format the URL query or its part
  }

});

BrowsableMixin.mixin(MyCollection.prototype);
BrowsableV1RequestMixin.mixin(MyCollection.prototype);
```

makeBrowsableV1Request(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Collection constructor options passed in. It calls `makeBrowsable` from `BrowsableMixin` too.

opentext™

getBrowsableUrlQuery(options) : object literal or string

Formats the URL query parameters for paging, sorting and filtering. They can be concatenated with other URL query parts (both object lierals and strings) by `Url.combineQueryString`.

```
query = Url.combineQueryString(
  ...,
  this.getBrowsableUrlQuery()
);
```

See Also

`BrowsableMixin`, `BrowsableV1ResponseMixin`

opentext™

BrowsableV1ResponseMixin

Deserializes the collection state maintained by `BrowsableMixin` and the collection of items from the server response according to the V1 of the REST API (`/api/v1/nodes/:id/nodes`), for example). The `BrowsableMixin` or a mixin including it must be applied with this mixin together.

Browsing implementation and response parsing is supposed to be added by other mixins, according to the specifics of the server resource.

Example

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);

    this
      .makeBrowsable(options)
      .makeBrowsableV1Request(options)
      .makeBrowsableV1Response(options);
  },

  url: function () {
    // use `getBrowsableUrlQuery` to format the URL query or its part
  },

  parse: function (response, options) {
    // use `parseBrowsedState` to update the browsing properties
    // according to the server response and return the received part
    // of the collection by calling `parseBrowsedItems`
  }

});

BrowsableMixin.mixin(MyCollection.prototype);
BrowsableV1RequestMixin.mixin(MyCollection.prototype);
BrowsableV1ResponseMixin.mixin(MyCollection.prototype);
```

opentext™

makeBrowsableV1Response(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Collection constructor options passed in. It calls `makeBrowsable` from `BrowsableMixin` too.

parseBrowsedState(response, options) : nothing

Updates the browsing state from the server response. It expects the arguments passed into the `parse` method, where this method should be called.

parseBrowsedItems(response, options) : array of object literals

Gathers the items from the server response and returns them as an array, which can be used initialized the collection of models with.

See Also

`BrowsableMixin`, `BrowsableV1RequestMixin`

opentext™

BrowsableV2ResponseMixin

Deserializes the collection state maintained by `BrowsableMixin` and the collection of items from the server response according to the V1 of the REST API (`/api/v2/members/favorites`), for example). The `BrowsableMixin` or a mixin including it must be applied with this mixin together.

Browsing implementation and response parsing is supposed to be added by other mixins, according to the specifics of the server resource.

Example

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);

    this
      .makeClientSideBrowsable(options)
      .makeBrowsableV1Request(options)
      .makeBrowsableV2Response(options);
  },

  url: function () {
    // use `getBrowsableUrlQuery` to format the URL query or its part
  },

  parse: function (response, options) {
    // use `parseBrowsedState` to update the browsing properties
    // according to the server response and return the received part
    // of the collection by calling `parseBrowsedItems`
  }

});

ClientSideBrowsableMixin.mixin(MyCollection.prototype);
BrowsableV1RequestMixin.mixin(MyCollection.prototype);
BrowsableV2ResponseMixin.mixin(MyCollection.prototype);
```

opentext™

makeBrowsableV2Response(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Collection constructor options passed in. It calls `makeBrowsable` from `BrowsableMixin` too.

parseBrowsedState(response, options) : nothing

Updates the browsing state from the server response. It expects the arguments passed into the `parse` method, where this method should be called.

parseBrowsedItems(response, options) : array of object literals

Gathers the items from the server response and returns them as an array, which can be used initialized the collection of models with.

See Also

`ClientSideBrowsableMixin`, `BrowsableV1RequestMixin`

opentext™

AutoFetchableMixin

Makes use of the module identifier to:

- check if the model can be fetched; the default property to check is 'id'
- fetch the model automatically when the identifier changes (if requested); the default event to listen to is 'change:id'

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeFetchable(options)
      .makeAutoFetchable(options);
  }

});

ConnectableMixin.mixin(MyModel.prototype);
FetchableMixin.mixin(MyModel.prototype);
AutoFetchableMixin.mixin(MyModel.prototype);
```

This mixin us usually comined together with the `ConnectableMixin` or with another cumulated mixin which includes it and also with the with the `FetchableMixin` to prevent parallel fetches. If you need all these three mixins, have a look at the `ResourceMixin`, which combines these three together.

How use the mixin

```
// Enable watching for the model identifier changes
var model = new MyModel(undefined, {
```

```
      connector: connector,
      autofetch: true
    });

// A fetch will take place, notifying the event listeners about its progress
model.set('id', 2000);
```

makeAutoFetchable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model constructor options passed in.

Recognized option properies:

autofetch : If set to `true`, it executes the `fetch` method whenever the model identifier changes (boolean, `undefined` is the default)

autofetchEvent : Can override the event to listen to for the automatic fetching (string, 'change:id' by default)

automateFetch(boolean) : void

Truns on or off the automatic fetching depending if you pass `true` or `false` in. It can be used to change the behaviour set up by the constructor.

isFetchable() : boolean

Returns `true` if the model is fetchable, otherwise `false`. If the model is not fetchable, the automatic fetching will not take place.

See Also

`ConnectableMixin`, `FetchableMixin`, `ResourceMixin`

opentext™

CommandableMixin

Provides support for the setting `commands` URL query parameter as introduced by the `api/v1/nodes/:id/nodes` (V1) resource.

Server responses can contain *permitted actions* to be able to support enabling and disabling in the corresponding UI; how many and which ones should be checked by the server can be specified.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeCommandable(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getRequestedCommandsUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});

ConnectableMixin.mixin(MyModel.prototype);
CommandableMixin.mixin(MyModel.prototype);
```

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How to use the mixin

opentext™

Set up the URL parameters by calling `setCommands` and `resetCommands` and fetch the model:

```
// Set the commands for requesting when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      commands: ['delete', 'reserve']
    });
model.fetch();

// Set the commands for requesting after creating the model
model.setCommands(['delete', 'reserve']);
model.fetch();
```

makeCommandable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

commands : One or more command signatures to be requested for being checked. The value is handled the same way as the `setCommands` method does it. An empty array is the default.

commands

Command signatures to be requested for being checked (array of strings, empty by default, read-only).

setCommands(names) : void

Asks for one or more commands to be checked. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have two commands checked, option 1
model.setCommands(['delete', 'reserve']);
```

```
// Have two commands checked, option 2
model.setCommands('delete');
model.setCommands('reserve');
// Have two commands checked, option 3
model.setCommands('delete,reserve');
```

resetCommands(names) : void

Prevents one or more commands from being checked. The `names` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all commands will be removed (not to be checked).

```
// Cancel all command checks and fetch the fresh data
model.resetCommands();
model.fetch();
```

getRequestedCommandsUrlQuery() : string

Formats the URL query parameters for the command investigation. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getRequestedCommandsUrlQuery()
    );
if (query) {
  url = Url.appendQuery(url, query);
}
```

See Also

`ConnectableMixin`

opentext™

ConnectableMixin

Provides support for the server connection using the `Connector` to update the target model or collection.

Remarks

This mixin overrides the `_prepareModel` method and calls the original implementation afterwards. If you supply your own custom implementation of this method, or use another mixin which overrides it, you should apply this mixin after yours.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this.makeConnectable(options);
  },

  url: function () {
    return Url.combine(this.connector.connection.url, 'myresource');
  }

});

ConnectableMixin.mixin(MyModel.prototype);
```

How use the mixin

Specify the connector and fetch the model:

```
// Specify the connector when creating the model
var connector = new Connector(...),
    model = new MyModel(undefined, {connector: connector});
model.fetch();

// Set the connector after creating the model
connector.assignTo(model);
model.fetch();
```

makeConnectable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properies:

connector : The `Connector` instance to use. If it is not provided, the connector has to be assigned at latest before the server is accessed or resource URL is computed. (object, `undefined` by default)

connector

The `Connector` instance assigned to this model or collection (object, read-only)

See Also

`ResourceMixin`

opentext™

DelayedCommandableMixin

Provides support for fetching permitted actions by an extra call after the primary node collection has been fetched. It depends on setting the `commands` URL query parameter as introduced by the `api/v1/nodes/:id/nodes` (V1) resource to specify just the default actions. The it issues an additional `api/v2/actions/nodes` call to enquire about the rest of the actions.

Server responses can contain *permitted actions* to be able to support enabling and disabling in the corresponding UI; how many and which ones should be checked by the server can be specified.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeDelayedCommandable(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getRequestedCommandsUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});

ConnectableMixin.mixin(MyModel.prototype);
DelayedCommandableMixin.mixin(MyModel.prototype);
```

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

opentext™

How to use the mixin

Set up the URL parameters by calling `setDefaultActionCommands`, `resetDefaultActionCommands`, `setCommands` and `resetCommands` and fetch the model:

```
// Set the commands for requesting when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      commands: ['download', 'delete', 'reserve']
      defaultActionCommands: ['download'],
      delayRestCommands: true
    });
model.fetch();

// Set the commands for requesting after creating the model
model.setCommands(['download', 'delete', 'reserve']);
model.setDefaultActionCommands(['download']);
model.fetch();
```

makeDelayedCommandable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

commands : One or more command signatures to be requested for being checked. The value is handled the same way as the `setCommands` method does it. An empty array is the default.

defaultActionCommands : One or more command signatures to be requested for being checked immediately by the first call. The rest of commands specified by the `commands` parameter will be fetched later by the second call. The value is handled the same way as the `setDefaultActionCommands` method does it. An empty array is the default.

delayRestCommands : Enables delayed fetching of the non-default actions by a second server call. `False` is the default.

delayRestCommandsForModels : Enables delayed fetching of the non-default actions by a second server call for child models, which are created automatically by adding attribute objects to the collection; if the mixin is aplied to a collection, the parameter `delayRestCommands` does not apply to the child models; the `delayRestCommandsForModels` does. `False` is the default.

commands

Command signatures to be requested for being checked (array of strings, empty by default, read-only).

defaultActionCommands

Command signatures for being checked to be requested immediately by the first server call (array of strings, empty by default, read-only).

delayRestCommands

: Says, if delayed fetching of the non-default actions by a second server call is enabled (boolean, `false` by default, read-only).

delayRestCommandsForModels

: Says, if delayed fetching of the non-default actions by a second server call is enabled for child models, which are created automatically by adding attribute objects to the collection; if the mixin is aplied to a collection, the parameter `delayRestCommands` does not apply to the child models; the `delayRestCommandsForModels` does (boolean, `false` by default, read-only).

delayedActions

A collection fetching the rest of non-default actions. If you need to wait until all permitted actions are received, you need to check the `fetched` status of this collection, or listen to the 'sync' even of this

collection. The target collection, which you applied this mixin too will report that it is fetched immediately after the first server call is finished with the default actions only.

```
var model = new MyModel(undefined, {
      connector: connector,
      commands: ['download', 'delete', 'reserve']
      defaultActionCommands: ['download'],
      delayRestCommands: true
    });
model
    .once('sync', function () {
      // nodes with permitted default actions are available
    })
    .delayedActions.once('sync', function () {
      // nodes with all permitted actions are available
    })
    .fetch();
```

setCommands(names) : void

Asks for one or more commands to be checked. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have two commands checked, option 1
model.setCommands(['delete', 'reserve']);
// Have two commands checked, option 2
model.setCommands('delete');
model.setCommands('reserve');
// Have two commands checked, option 3
model.setCommands('delete,reserve');
```

resetCommands(names) : void

Prevents one or more commands from being checked. The `names` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all commands will be removed (not to be checked).

```
// Cancel all command checks and fetch the fresh data
model.resetCommands();
model.fetch();
```

setDefaultActionCommands(names) : void

Asks for one or more commands to be checked immediately by the first server call. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array. The rest of commands specified by the `setCommands` method will be fetched later by the second call.

```
// Have two commands checked, option 1
model.setCommands(['browse', 'download']);
// Have two commands checked, option 2
model.setCommands('browse');
model.setCommands('download');
// Have two commands checked, option 3
model.setCommands('browse,download');
```

resetDefaultActionCommands(names) : void

Prevents one or more commands from being checked immediatley by the first server call. The `names` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all commands will be removed (not to be checked by the first server call).

```
// Have all commands fetched delayed by the second server call
model.setCommands(['browse', 'download']);
model.resetDefaultActionCommands();
model.fetch();
```

getRequestedCommandsUrlQuery() : string

opentext™

Formats the URL query parameters for the command investigation. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`. If delayed action fetching is enabled, only the default actions will be returned by this method; the rest of specified actions fill be fetched by an additional server call later.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getRequestedCommandsUrlQuery()
    );
if (query) {
  url = Url.appendQuery(url, query);
}
```

See Also

`ConnectableMixin`, `CommandableMixin`

opentext™

ExpandableMixin

Provides support for the setting `expand` URL query parameter as introduced by the `api/v1/nodes/:id` or `api/v1/nodes/:id/nodes` (V1) resources.

Server responses can contain references to other resources; typically IDs or URLs. The *expansion* means replacing them with object literals containing the resource information, so that the caller does not have to request every associated resource by an additional server call.

Expandable resource types:

node : nodes and volumes (`original_id`, `parent_id`, `volume_id` etc.)

user : users or user groups (`create_user_id`, `modify_user_id`, `owner_user_id` etc.)

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeExpandable(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getExpandableResourcesUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});
```

```
ConnectableMixin.mixin(MyModel.prototype);
ExpandableMixin.mixin(MyModel.prototype);
```

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How to use the mixin

Set up the URL parameters by calling `setExpand` and `resetExpand` and fetch the model:

```
// Set the expansion when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      expand: ['node', 'user']
    });
model.fetch();

// Set the expansion after creating the model
model.setExpand(['node', 'user']);
model.fetch();
```

makeExpandable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

expand : One or more resource types expanded. The value is handled the same way as the `setExpand` method does it. An empty array is the default.

expand

Resource types to get expanded in the response (array of strings, empty by default, read-only).

setExpand(name) : void

Makes one or more resource types expanded. The `name` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have two resource types expanded, option 1
model.setExpand(['node', 'user']);
// Have two resource types expanded, option 2
model.setExpand('node');
model.setExpand('user');
// Have two resource types expanded, option 3
model.setExpand('node,user');
```

resetExpand(name) : void

Prevents one or more resource types from being expanded. The `name` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all expansions will be removed (disabled).

```
// Cancel all expansions and fetch the fresh data
model.resetExpand();
model.fetch();
```

getExpandableResourcesUrlQuery() : string

Formats the URL query parameters for the resource expansion. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getExpandableResourcesUrlQuery()
    );
if (query) {
```

```
    url = Url.appendQuery(url, query);
  }
```

See Also

`ConnectableMixin`, `ResourceMixin`

opentext™

FetchableMixin

Makes the calls to the `fetch` method more robust by:

- preventing multiple server calls when the `fetch` is called quickly one call after another
- setting the `reset` option automatically to optimize collections loading (if requested)
- checking, if the model has already been fetched and ensuring, that a model has always been fetched before it is used

Remarks

This mixin overrides the `fetch` method and calls the original implementation from it. If you supply your own custom implementation of this method, or use another mixin which overrides it, you should apply this mixin after yours.

When the `fetch` method is called and the previous call has not ended yet, **the new call will not be made**. The promise returned by the previous call will be returned instead. If you specified different options than for the previous call, they will not be reflected.

How to apply the mixin to a collection

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeFetchable(options);
  }

});

ConnectableMixin.mixin(MyCollection.prototype);
FetchableMixin.mixin(MyCollection.prototype);
```

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How use the mixin

```
// Set the `reset` option for the future `fetch` calls automatically
var collection = new MyCollection(undefined, {
      connector: connector,
      autoreset: true
    });
collection.fetch();

// Ensure that the collection has been fetched and process it (I)
collection
  .ensureFetched()
  .done(function () {
    ...
  });
```

makeFetchable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properies:

autoreset : If set to `true`, the `reset` option for the future `fetch` calls will be set to `true` automatically, unless this option is not set by the caller (boolean, `undefined` by default)

fetching

Can be checked to test whether the model or collection is currently being fetched. When fetching is in progress, it contains the promise returned by the most recent `fetch` call. When no fetching is in proress it is `false`. (promise or boolean, read-only)

opentext™

fetched

Can be checked to test whether the most recent `fetch` call has succeeded or not on this model or collection; it is set to `true` on the first occurrence and never changed afterwards (boolean, read-only)

error

Contains an error if the most recent `fetch` call failed. It is `undefined` if the most recent `fetch` call succeeded (Error, read-only)

ensureFetched(options) : promise

Returns a promise resolved as soon as the model is fetched. It fetches the model, if it has not been fetched yet, or it returns immediately, when the model has already been fetched.

invalidateFetch

Invalidates the fetched state of the collection so that the next ensureFetched() call will fetch the data.

See Also

`ConnectableMixin`, `ResourceMixin`

opentext™

IncludingAdditionalResourcesMixin

Provides support for the setting URL query parameter flags as introduced by the `api/v1/nodes/:id` (V1) resource.

Server responses can contain associated resources to avoid requesting every associated resource by an additional server call, or other data, which may not be needed every time. For example:

actions : Include permitted actions.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeIncludingAdditionalResources(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getAdditionalResourcesUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});

ConnectableMixin.mixin(MyModel.prototype);
IncludingAdditionalResourcesMixin.mixin(MyModel.prototype);
```

opentext™

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How to use the mixin

Set up the URL parameters by calling `includeResources` and `excludeResources` and fetch the model:

```
// Set the inclusion when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      includeResources: ['perspective']
    });
model.fetch();

// Set the inclusion after creating the model
model.includeResources('perspective');
model.fetch();
```

makeIncludingAdditionalResources(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

includeResources : One or more resources to include. The value is handled the same way as the `includeResources` method does it. An empty array is the default.

_additionalResources

Resources to get included in the response in the response (array of strings, empty by default, read-only).

includeResources(names) : void

opentext™

Makes one or more resources included. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have a resources included, option 1
model.includeResources('perspective');
// Have a resource included, option 2
model.includeResources(['perspective']);
```

excludeResources(names) : void

Prevents one or more resources from being included. The `names` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all inclusions will be removed (disabled).

```
// Cancel all inclusions and fetch the fresh data
model.excludeResources();
model.fetch();
```

getAdditionalResourcesUrlQuery() : string

Formats the URL query parameters for the resource inclusion. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getAdditionalResourcesUrlQuery()
    );
if (query) {
  url = Url.appendQuery(url, query);
}
```

See Also

opentext™

`ConnectableMixin`, `ResourceMixin`

opentext™

Mixins for Models

Here you can find prototype partials to be "mixed in" to your model objects. You will get additional methods in your prototype coming from the *mixin*.

Mixins should not replace your existing methods or methods you inherit from a parent object. You are supposed to call the mixin methods explicitly to gain the added functionality. If they override a Backbone method, it should be documented and you will need to be careful about the order you apply the mixins in.

Mixins usually export an object literal to be merged with the prototype of the target model by calling its static `mixin` method:

```
MyMixin.mixin(MyModel.prototype);
```

opentext™

NodeAutoFetchableMixin

Makes use of the identifier of the related node to:

- check if the model or collection can be fetched; it checks only if the related node is fetchable by default
- fetch the model or collection automatically when the identifier of the related node changes (if requested); the default event to listen to is 'change:id'

How to apply the mixin to a model

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);
    this
      .makeNodeConnectable(options)
      .makeFetchable(options)
      .makeNodeAutoFetchable(options);
  }

});

NodeConnectableMixin.mixin(MyCollection.prototype);
FetchableMixin.mixin(MyCollection.prototype);
NodeAutoFetchableMixin.mixin(MyCollection.prototype);
```

This mixin us usually comined together with the `NodeConnectableMixin` or with another cumulated mixin which includes it and also with the with the `FetchableMixin` to prevent parallel fetches. If you need all these three mixins, have a look at the `NodeResourceMixin`, which combines these three together.

How use the mixin

```
// Enable watching for the model identifier changes
var connector = new Connector(...),
    node = new NodeModel({id: 2000}, {connector: connector}),
    collection = new MyCollection(undefined, {
      node: node,
      autofetch: true
    });

// A fetch of the collection will take place, notifying the event
// listeners about its progress
node.set('id', 2000);
```

makeNodeAutoFetchable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model constructor options passed in.

Recognized option properies:

autofetch : If set to `true`, it executes the `fetch` method whenever the node model identifier changes (boolean, `undefined` is the default)

autofetchEvent : Can override the event to listen to on the node model for the automatic fetching (string, 'change:id' by default)

automateFetch(boolean) : void

Truns on or off the automatic fetching depending if you pass `true` or `false` in. It can be used to change the behaviour set up by the constructor.

isFetchable() : boolean

Returns `true` if the model or collection is fetchable, otherwise `false`. It requires the related node not only fetchable, but also having the 'id' property set. (While nodes can be fetched using other properties,

like volumes by 'type', for example, other resources related to the node may not.) If the model or collection is not fetchable, the automatic fetching will not take place.

See Also

`NodeConnectableMixin`, `FetchableMixin`, `NodeResourceMixin`

opentext™

NodeConnectableMixin

Provides support for the server connection via a node, represented by `NodeModel`, its descendant or by other class with the `ConnectableMixin` applied.

Many resources are associated with a node, which can be a parent, for example, or in other relation. The node provides the `Connector` to update the target model or collection and its identifier usually takes part in the resource URL.

Remarks

This mixin overrides the `_prepareModel` method and calls the original implementation afterwards. If you supply your own custom implementation of this method, or use another mixin which overrides it, you should apply this mixin after yours.

How to apply the mixin to a model

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);
    this.makeNodeConnectable(options);
  },

  url: function () {
    return Url.combine(this.node.urlBase(), 'mysubresources');
  }

});

NodeConnectableMixin.mixin(MyCollection.prototype);
```

How use the mixin

```
// Specify the connector when creating the model
var connector = new Connector(...),
    node = new NodeModel({id: 2000}, {connector: connector}),
    collection = new MyCollection(undefined, {node: node});
collection.fetch();
```

makeNodeConnectable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properies:

node : The `NodeModel` instance to use (object, mandatory)

node

The `NodeModel` instance assigned to this model or collection in the constructor (object, read-only)

connector

The `Connector` instance assigned to this model or collection via the related node (object, read-only)

See Also

`NodeResourceMixin`, `NodeModel`

opentext™

NodeResourceMixin

Helps implementing a model or collection for a typical server resource, which is related to a node (represented by the `NodeModel`, its descendant or by other class with the `ConnectableMixin` applied), by combining the following three mixins: `NodeConnectableMixin`, `FetchableMixin` and `NodeAutoFetchableMixin`.

How to apply the mixin to a collection

```
var MyCollection = Backbone.Collection.extend({

  constructor: function MyCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);
    this.makeNodeResource(options);
  },

  urlRoot: function () {
    return Url.combine(this.node.urlBase(), 'mysubresources');
  }

});

NodeResourceMixin.mixin(MyCollection.prototype);
```

Remarks

The included `FetchableMixin` overrides the `fetch` method and calls the original implementation from it. If you supply your own custom implementation of this method, or use another mixin which overrides it, you should apply this mixin after yours.

How use the mixin

Specify the related node and fetch the collection:

```
// Specify the node attributes when creating the node
var connector = new Connector(...),
    model = new MyModel({
      id: 2000
    }, {
      connector: connector
    }),
    collection = new MyCollection(undefined, {
      node: node
    });
collection.fetch();

// Set the node attributes after creating it
node.set('id', 2000);
collection.fetch();
```

makeNodeResource(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model constructor options passed in.

See the `make` methods of `NodeConnectableMixin`, `FetchableMixin` and `NodeAutoFetchableMixin` for the properties recognized by this method.

See also the properties and methods exposed by these three mixins to learn what this convenience mixin provides

See Also

`NodeConnectableMixin`, `FetchableMixin`, `NodeAutoFetchableMixin`

opentext™

ResourceMixin

Helps implementing a model for a typical server resource, which has an identifier (the property 'id' by default), by combining the following three mixins: `ConnectableMixin`, `FetchableMixin` and `AutoFetchableMixin`.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this.makeResource(options);
  },

  urlRoot: function () {
    return Url.combine(this.connector.connection.url, 'myresources');
  }

});

ResourceMixin.mixin(MyModel.prototype);
```

Remarks

The included `FetchableMixin` overrides the `fetch` method and calls the original implementation from it. If you supply your own custom implementation of this method, or use another mixin which overrides it, you should apply this mixin after yours.

How use the mixin

Specify the model attributes, the connector and fetch the model:

```
// Specify the attributes and the connector when creating the model
var connector = new Connector(...),
    model = new MyModel({
      id: 2000
    }, {
      connector: connector
    });
model.fetch();

// Set the attributes and the connector after creating the model
model.set('id', 2000);
connector.assignTo(model);
model.fetch();
```

makeResource(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model constructor options passed in.

See the `make` methods of `ConnectableMixin`, `FetchableMixin` and `AutoFetchableMixin` for the properties recognized by this method.

See also the properties and methods exposed by these three mixins to learn what this convenience mixin provides

See Also

`ConnectableMixin`, `FetchableMixin`, `AutoFetchableMixin`

opentext™

RulesMatchingMixin

Helps implementing a collection of rule models. which are supposed to select one model, which rules match the input object. For example, choose an icon or the default click-action for a particular node.

How to apply the mixin to a model

```
var NodeActionSelectingModel = Backbone.Model.extend({
  idAttribute: 'signature',

  defaults: {
    sequence: 100,
    signature: null
  },

  constructor: function NodeActionSelectingModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this.makeRulesMatching(options);
  },

  enabled: function (node, options) {
    return this.matchRules(node, this.attributes, options);
  }
});

RulesMatchingMixin.mixin(NodeActionSelectingModel.prototype);

var NodeActionSelectingCollection = Backbone.Collection.extend({
  model: NodeActionSelectingModel,

  comparator: 'sequence',

  constructor: function NodeActionSelectingCollection(models, options) {
    Backbone.Collection.prototype.constructor.apply(this, arguments);
  },

  findByNode: function (node, options) {
    return this.find(function (rule) {
```

```
      return rule.enabled(node, options);
    });
  }
});
```

The `options` parameter in `findByNode` and `enabled` methods is optional. You can use it for special cases. For example, if you expect the developers to use the `decides` rule, you can allow them access other data than just the subject of the decision (NodeModel in this example). For example, the `context` instance. You can set such data to an object and pass via that optional `options` parameter.

How use the mixin

Populate a collection of rule models. Whenever you need to find, which rule models matches the object of the selection, use the method exposed for this:

```
var nodeActions = new NodeActionSelectingCollection([
  {
    equals: {
      container: true
    },
    signature: 'Browse'
  },
  {
    equals: {
      type: [144, 749]
    },
    signature: 'Open'
  },
  {
    and: [
      equals: {
        type: 144
      },
      containsNoCase: {
        mime_type: [
          "application/msword",
          "application/vnd.ms-word",
          "application/vnd.msword",
          "application/vnd.openxmlformats-officedocument.wordprocessingml.document",
          "application/vnd.wordprocessing-openxml",
```

```
            "application/vnd.ces-quickword",
            "application/vnd.ms-word.document.macroEnabled.12",
            "application/vnd.ms-word.document.12",
            "application/mspowerpoint",
            "application/vnd.ms-powerpoint",
            "application/vnd.openxmlformats-officedocument.presentationml.presentation",
            "application/vnd.ces-quickpoint",
            "application/vnd.presentation-openxml",
            "application/vnd.presentation-openxmlm",
            "application/vnd.ms-powerpoint.presentation.macroEnabled.12",
            "application/msexcel",
            "application/vnd.ms-excel",
            "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
            "application/vnd.ces-quicksheet",
            "application/vnd.spreadsheet-openxml",
            "application/vnd.ms-excel.sheet.macroEnabled.12",
          ]
        }
      },
      signature: 'Edit',
      sequence: 50
    },
    {
      and: {
        equals: {
          type: 144
        },
        or: {
          startsWithNoCase: {
            mime_type: 'image/'
          },
          equalsNoCase: {
            mime_type: ['application/pdf', 'application/x-pdf']
          }
        }
      },
      signature: 'Convert',
      sequence: 50
    },
    {
      and: [
        {
          equals: {
            type: 144
```

```
        }
      },
      {
        equalsNoCase: {
          mime_type: 'text/plain'
        }
      }
    ],
    signature: 'Read',
    sequence: 50
  },
  {
    signature: 'Properties',
    sequence: 200
  }
]);

var node = new NodeModel(...),
    action = nodeActions.findByNode(node);
```

makeRulesMatching(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model constructor options passed in.

Conditions

Conditions from `equals` to `includes` below are case-sensitive, if their operands are strings. They have their case-insensitive couterparts ending with "NoCase". For example, `equalsNoCase` or `includesNoCase`.

equals - the value of the selected property has to equal at least one of the specified values. Strings comparisons are case-insensitive.

contains - the value of the selected property has to be a part of at least one of the specified values. String case-insensitive "indexOf" operation is applied.

startsWith - the value of the selected property has to be at the beginning of at least one of the specified values. String case-insensitive "startsWith" operation is applied.

endsWith - the value of the selected property has to be at the end of at least one of the specified values. String case-insensitive "endsWith" operation is applied.

matches - the regular expression has to match at least one of the specified values. The operation is case-insensitive.

includes - if the value of the selected property is an object, it has to include at least one the specified keys; if the value of the selected property is an array, it has to include at least one the specified items. The standard equal operator is applied when comparing keys and items.

has - at least one of the specified properties must exist and not be null.

decides - the function has to return `true` at least for one of the specified values.

or - at least one of the sub-conditions in the arry or object has to return `true`.

all - all of the sub-conditions in the arry or object has to return `true`.

not - negates the result of the sub-condition.

Properties

Conditions are performed on properties of the scenario-controlling object - on attributes of the controlling model. Property name is the key in the object, which is the value of the operation:

```
{
  equals: {
    type: 144
  },
  signature: 'Open'
}
```

The key can be a dot-separated expression, which would evaluate like when a nested object is accessed in JavaScript:

```
{
  equals: {
    'id_expand.type': 5574
  },
  signature: 'OpenWikiPageVersion'
}
```

The object passed to `matchRules` can be either an object literal, which will become the direct source of the properties to test, or a `Backbone.Model` instance, which `aatributes` will be used of.

opentext™

StateCarrierMixin

Provides support for maintaining the `state` properties as introduced by the `api/v2/nodes/:id` or `api/v2/nodes/:id/nodes` (V2) resources.

Server responses may contain not only data, but also state properties, which may be needed for evaluation on either client or server sides. The `metadata_token`, for example.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({
  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeFieldsV2(options)
      .makeStateCarrier(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource');
    var query = Url.combineQueryString(
      this.getResourceFieldsUrlQuery(),
      this.getStateEnablingUrlQuery()
    );
    return Url.appendQuery(url, query);
  },

  parse: function (response, options) {
    var results = response.results || response;
    this.parseState(results, 'properties');
    return results.data.properties;
  }
});

ConnectableMixin.mixin(MyModel.prototype);
FieldsV2Mixin.mixin(MyModel.prototype);
StateCarrierMixin.mixin(MyModel.prototype);
```

This mixin us usually combined together with the `ConnectableMixin` and `FieldsV2Mixin` or with another cumulated mixin which includes them. This mixin includes the `StateRequestorMixin` and applieas it automatically.

How to use the mixin

Get or set properties in te `state` object as you need them:

```
// Get the metadata toklen to pass it to a modification server call
var metadataToken = model.state.get('metadata_token');

// Set the metadatan token from a response of a server call
model.state.set('metadata_token', results.state.properties.metadata_token);
```

Simplify parsing the state properties from a standard V2 API response:

```
// Support retrieving the attributes when fetching either collection or model.
var results = response.results || response;
var data = results.data;
this.parseState(data, results, 'properties');
```

makeStateCarrier(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the `Backbone.Model` or `Backbone.Collection` constructor options passed in.

state : Backbone.Model

Maintains the object state properties (empty by default).

parseState(response, role) : void

opentext™

Parses a standard V2 response and sets the state properties to the `state` object on the instance. The `response` parameter is an object with a response object; either a complete response, or just the `results` sub-object, if it is present in the response as a wrapper. The `role` parameter is a `string` choosing the `fields` role, which the state should be received for; the default value is "properties".

See Also

`StateRequestorMixin`. `FieldsV2Mixin`

opentext™

StateRequestorMixin

Provides support for setting the `state` URL query parameter as introduced by the `api/v2/nodes/:id` or `api/v2/nodes/:id/nodes` (V2) resources.

Server responses may contain not only data, but also state properties, which would reduce the performce of the request, if they were always returned.

The state can be enabled or disabled. Enabling the state will include state properties for the enabled `fields` in the response.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({
  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeFieldsV2(options)
      .makeStateRequestor(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource');
    var query = Url.combineQueryString(
      this.getResourceFieldsUrlQuery(),
      this.getStateEnablingUrlQuery()
    );
    return Url.appendQuery(url, query);
  }
  });

ConnectableMixin.mixin(MyModel.prototype);
FieldsV2Mixin.mixin(MyModel.prototype);
StateRequestorMixin.mixin(MyModel.prototype);
```

opentext™

This mixin us usually combined together with the `ConnectableMixin` and `FieldsV2Mixin` or with another cumulated mixin which includes them.

How to use the mixin

Enabkle or disable retrieval of the state by calling `enableState` or `disableState` and fetch the model:

```
// Set the expansion when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      fields: {
        properties: ['parent_id', 'create_user_id']
      }
    });
model.fetch();

// Set the expansion after creating the model
model.setFields('properties', ['parent_id', 'create_user_id']);
model.fetch();
```

makeFieldsV2(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the `Backbone.Model` or `Backbone.Collection` constructor `options` passed in.

Recognized option properties:

stateEnabled : If the state retrieval should be enabled or not (boolean, `false` by default).

stateEnabled : boolean

If the state retrieval should be enabled or not (`false` by default, read-only).

enableState() : void

opentext™

Enables retrieval of the object state.

```
// Enable retrieval of the object state
model.enableState();
```

disableState() : void

Disables retrieval of the object state.

```
// Disable retrieval of the object state
model.enableState();
```

getStateEnablingUrlQuery() : object

Returns an object with URL query parameters for the request URL construction. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`.

```
var url = ...,;
var query = Url.combineQueryString(
  ...,
  this.getStateEnablingUrlQuery()
);
url = Url.appendQuery(url, query);
```

See Also

`StateCarrierMixin`. `FieldsV2Mixin`

opentext™

SyncableFromMultipleSourcesMixin

Simplifies implementing models and collections, which have to be fetched by issuing multiple asynchronous calls. Either by executing multiple `$.ajax` statements, or by feching multiple models or collections and merging their content.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this.makeSyncableFromMultipleSources(options);
  },

  sync: function (method, model, options) {
    return this.syncFromMultipleSources(
      [...promises], this._mergeSources,
      this._convertError, options);
  },

  _mergeSources: function (...results) {
    return merged response;
  },

  _convertError: function (result) {
    return {
      status: ...,
      statusText: '...',
      responseJSON: ...
    };
  }

});

SyncableFromMultipleSources.mixin(MyModel.prototype);
```

opentext™

How to combine multiple `$.ajax` calls

```
sync: function (method, model, options) {
  var first = $.ajax(this.connector.extendAjaxOptions({
                url: '...'
              })
              .then(function (response) {
                return response;
              }),
      second = $.ajax(this.connector.extendAjaxOptions({
                 url: '...'
               })
               .then(function (response) {
                 return response;
               });
  return this.syncFromMultipleSources(
      [first, second], this._mergeSources, options);
},

_mergeSources: function (first, second) {
  return merged response to be parsed;
}
```

How to combine multiple model/collection fetches

```
sync: function (method, model, options) {
  var first = new FirstModel(...),
      second = new SecondCollection(...);
  first = first.fetch(options)
              .then(function () {
                return first.toJSON()
              }),
  second = second.fetch(options)
               .then(function () {
                 return second.toJSON()
               }),
  options.parse = false;
  return this.syncFromMultipleSources(
      [first, second], this._mergeSources, options);
},
```

```
_mergeSources: function (first, second) {
  return already parsed merged response;
}
```

makeSyncableFromMultipleSources(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

syncFromMultipleSources(promises, mergeSources, convertError, options) : promise

Implements the interface (events, callbacks and promise) of `Backbone.sync` by waiting on the source promises and by resolving the target promise with the merged response returned by the caller's callback, which receives results of the source promises to merge them.

If one of the source promises fails, the rejected result will be passed to `convertError`, which is an optional parameter. (A function expecting rejected result from $.ajax will be used by default.) If specified, it has to return an object simulating the jQuery AJAX object:

```
{
  statusText: '...',
  responseJSON: {...}
}
```

opentext™

UploadableMixin

Enables creating and modifying the resource behind a `Backbone.Model`. Simplifies requests with multi-part content, enables mocking by mockjax and supports pasing the JSON body as form-encoded parameter "body". Also introduces the `prepare` method to "massage" request body similarly to the `parse` method for the response body.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({
  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this.makeConnectable(options)
        .makeUploadable(options);
  },

  url: function () {
    return Url.combine(this.connector.connection.url, 'myresource')
  }
});

ConnectableMixin.mixin(MyModel.prototype);
UploadableMixin.mixin(MyModel.prototype);
```

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How to use the mixin

It just works whenever you call the `save` method of the model.

makeUploadable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

prepare(data, options) : object

Can be overridden to "massage" the data, which are going to be sent to the server in the request body. If not overridden, the unchanged input of the `save` method, or `this.attributes`, will be sent to the server as-is. The `prepare` method converts model attributes to the request body object, so that the server accepts it, like the `parse` method converts the response body object to model attributes, so that the client understands them.

```
// For example, the server sends and receives resource attributes wrapped
// in an object with a `data` property.
var MyModel = Backbone.Model.extend({
  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this.makeUploadable(options);
  },

  prepare: function (data, options) {
    return {data: data};
  },

  parse: function (response, options) {
    return response.data;
  }
});

UploadableMixin.mixin(MyModel.prototype);
```

Calling the `prepare` method can be prevented by setting the `prepare` option to `false`, similarly to preventing the `parse` method from being called by setting the `parse` option to `false`.

See Also

`ConnectableMixin`, `ResourceMixin`

opentext™

Motivation

PATCH Semantics Support

The CS REST API implements neither PUT nor PATCH semantics correctly, which makes usage of high-level JavaScript frameworks like Backbone impossible. The resource modification request has to be built in a custom way, so that it follows the PATCH semantics, but uses the PUT verb. This mixin allows usage of the Backbone in the usual way - with patch-mode for modifications.

```
// Rename a concrete node
var connector = new Connector({
      connection: {
        url: '//server/instance/cs/api/v1',
        supportPath: '/instancesupport'
      }
    }),
    node = new NodeModel({
      id: 12345
    }, {
      connector: connector
    });
node.save({
      // properties to change on the server and in the model
      name: 'New name'
    }, {
      patch: true, // send only properties specified above;
                   // not everything from this.attributes
      wait: true   // set the properties to this.attributes
                   // only if and after the request succeeds
    })
    .done(function () {
      console.log('New name:', node.get('name'));
    })
    .fail(function () {
      console.log('Renaming the node failed.',
        'Old name:', node.get('name'));
    });
```

Another missing feature in the CS REST API is returning the created and modified properties from the POST and PUT responses. If you need the model complete after a modification, you need to fetch it again, wasting a server call:

```
// Rename a concrete node and refresh other properties like `modify_date`
node.save({
      name: 'New name'
    }, {
      patch: true,
      wait: true
    })
    .then(function () {
      return node.fetch();
    })
    .done(function () {
      console.log('New name:', node.get('name'));
    })
    .fail(function () {
      console.log('Renaming the node failed.');
    });
```

File Upload Support

If the newly created resource needs a raw file content, you can pass the fields of the file type via options and let the mixin build the right request payload and set ist content type.

```
// Upload a new document
var connector = new Connector({
      connection: {
        url: '//server/instance/cs/api/v1',
        supportPath: '/instancesupport'
      }
    }),
    node = new NodeModel({
      type: 144
    }, {
      connector: connector
    }),
    file = ...; // a File or Blob object
```

```
node.save({
      name: 'New document',
      parent_id: 2000
    }, {
      files: {
        file: file
      }
    })
    .done(function () {
      console.log('New document ID:', node.get('id'));
    })
    .fail(function (request) {
      var error = new base.Error(request);
      console.log('Uploading document failed:', error);
    });
```

Because the CS REST API is not friendly to clients expecting RESTful APIs, you will need to fetch the newly created node to get all properties, wasting another server call:

```
// Upload a new document and get all common properties
var node = new NodeModel(undefined, {connector: connector}),
    file = ...; // a File or Blob object
node.save({
      type: 144,
      name: 'New document',
      parent_id: 2000
    }, {
      files: {
        file: file
      }
    })
    .then(function () {
      return node.fetch();
    })
    .done(function () {
      console.log('New document ID:', node.get('id'));
    })
    .fail(function (request) {
      var error = new base.Error(request);
      console.log('Uploading document failed:', error);
    });
```

opentext™

AdditionalResourcesV2Mixin

Provides support for the setting URL query parameter flags as introduced by the `api/v2/nodes/:id` or `api/v2/nodes/:id/nodes` (V2) resources.

Server responses can contain associated resources to avoid requesting every associated resource by an additional server call, or other data, which may not be needed every time. For example:

perspective : Include the perspective configuration, if the resource can carry one.

metadata : Include the definitions of object properties.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeAdditionalResourcesV2Mixin(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getAdditionalResourcesUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});

ConnectableMixin.mixin(MyModel.prototype);
AdditionalResourcesV2Mixin.mixin(MyModel.prototype);
```

opentext™

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How to use the mixin

Set up the URL parameters by calling `includeResources` and `excludeResources` and fetch the model:

```
// Set the inclusion when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      includeResources: ['perspective']
    });
model.fetch();

// Set the inclusion after creating the model
model.includeResources('perspective');
model.fetch();
```

makeAdditionalResourcesV2Mixin(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

includeResources : One or more resources to include. The value is handled the same way as the `includeResources` method does it. An empty array is the default.

_additionalResources

Resources to get included in the response in the response (array of strings, empty by default, read-only).

includeResources(names) : void

opentext™

Makes one or more resources included. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have a resources included, option 1
model.includeResources('perspective');
// Have a resource included, option 2
model.includeResources(['perspective']);
```

excludeResources(names) : void

Prevents one or more resources from being included. The `names` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all inclusions will be removed (disabled).

```
// Cancel all inclusions and fetch the fresh data
model.excludeResources();
model.fetch();
```

getAdditionalResourcesUrlQuery() : string

Formats the URL query parameters for the resource inclusion. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getAdditionalResourcesUrlQuery()
    );
if (query) {
  url = Url.appendQuery(url, query);
}
```

See Also

opentext™

`ConnectableMixin`, `ResourceMixin`

opentext™

CommandableMixin

Provides support for the setting `actions` URL query parameter as introduced by the `api/v2/members/favorites` or `api/v2/members/accessed` (V2) resources.

Server responses can contain *permitted actions* to be able to support enabling and disabling in the corresponding UI; how many and which ones should be checked by the server can be specified.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeCommandable(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getRequestedCommandsUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});

ConnectableMixin.mixin(MyModel.prototype);
CommandableMixin.mixin(MyModel.prototype);
```

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How to use the mixin

Set up the URL parameters by calling `setCommands` and `resetCommands` and fetch the model:

```
// Set the commands for requesting when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      commands: ['delete', 'reserve']
    });
model.fetch();

// Set the commands for requesting after creating the model
model.setCommands(['delete', 'reserve']);
model.fetch();
```

makeCommandable(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

commands : One or more command signatures to be requested for being checked. The value is handled the same way as the `setCommands` method does it. An empty array is the default.

commands

Command signatures to be requested for being checked (array of strings, empty by default, read-only).

setCommands(names) : void

Asks for one or more commands to be checked. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have two commands checked, option 1
model.setCommands(['delete', 'reserve']);
```

```
// Have two commands checked, option 2
model.setCommands('delete');
model.setCommands('reserve');
// Have two commands checked, option 3
model.setCommands('delete,reserve');
```

resetCommands(names) : void

Prevents one or more commands from being checked. The `names` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all commands will be removed (not to be checked).

```
// Cancel all command checks and fetch the fresh data
model.resetCommands();
model.fetch();
```

getRequestedCommandsUrlQuery() : string

Formats the URL query parameters for the command investigation. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getRequestedCommandsUrlQuery()
    );
if (query) {
  url = Url.appendQuery(url, query);
}
```

See Also

`ConnectableMixin`

opentext™

DelayedCommandableV2Mixin

Provides support for fetching permitted actions by an extra call after the primary node model or collection has been fetched. It depends on setting the `actions` URL query parameter as introduced by the `api/v2/nodes/:id` (V2) resource to specify just the default actions. After the main call has finished. an additional `api/v2/actions/nodes` call will be issued to enquire about the rest of the actions.

Server responses can contain *permitted actions* to be able to support enabling and disabling in the corresponding UI; how many and which ones should be checked by the server, it can be specified.

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeDelayedCommandableV2(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getRequestedCommandsUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});

ConnectableMixin.mixin(MyModel.prototype);
DelayedCommandableV2Mixin.mixin(MyModel.prototype);
```

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

opentext™

How to use the mixin

Set up the URL parameters by calling `setDefaultActionCommands`, `resetDefaultActionCommands`, `setCommands` and `resetCommands` and fetch the model:

```
// Set the commands for requesting when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      commands: ['download', 'delete', 'reserve']
      defaultActionCommands: ['download'],
      delayRestCommands: true
    });
model.fetch();

// Set the commands for requesting after creating the model
model.setCommands(['download', 'delete', 'reserve']);
model.setDefaultActionCommands(['download']);
model.fetch();
```

makeDelayedCommandableV2(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

commands : One or more command signatures to be requested for being checked. The value is handled the same way as the `setCommands` method does it. An empty array is the default.

defaultActionCommands : One or more command signatures to be requested for being checked immediately by the first call. The rest of commands specified by the `commands` parameter will be fetched later by the second call. The value is handled the same way as the `setDefaultActionCommands` method does it. An empty array is the default.

delayRestCommands : Enables delayed fetching of the non-default actions by a second server call. `False` is the default.

delayRestCommandsForModels : Enables delayed fetching of the non-default actions by a second server call for child models, which are created automatically by adding attribute objects to the collection; if the mixin is aplied to a collection, the parameter `delayRestCommands` does not apply to the child models; the `delayRestCommandsForModels` does. `False` is the default.

promoteSomeRestCommands : Forces delayed fetching of only some non-default actions listed by `nuc/utils/promoted.actionitems`, if `delayRestCommands` is `true`. `True` is the default.

commands

Command signatures to be requested for being checked (array of strings, empty by default, read-only).

defaultActionCommands

Command signatures for being checked to be requested immediately by the first server call (array of strings, empty by default, read-only).

delayRestCommands

: Says, if delayed fetching of the non-default actions by a second server call is enabled (boolean, `false` by default, read-only).

delayRestCommandsForModels

: Says, if delayed fetching of the non-default actions by a second server call is enabled for child models, which are created automatically by adding attribute objects to the collection; if the mixin is aplied to a collection, the parameter `delayRestCommands` does not apply to the child models; the `delayRestCommandsForModels` does (boolean, `false` by default, read-only).

promoteSomeRestCommands

opentext™

: Says, if only non-default actions listed by `nuc/utils/promoted.actionitems` will be fetche delayed, if `delayRestCommands` is `true` (boolean, `true` by default, read-only).

delayedActions

A collection fetching the rest of non-default actions. If you need to wait until all permitted actions are received, you need to check the `fetched` status of this collection, or listen to the 'sync' even of this collection. The target collection, which you applied this mixin too will report that it is fetched immediately after the first server call is finished with the default actions only.

```
var model = new MyModel(undefined, {
      connector: connector,
      commands: ['download', 'delete', 'reserve']
      defaultActionCommands: ['download'],
      delayRestCommands: true
    });
model
    .once('sync', function () {
      // nodes with permitted default actions are available
    })
    .delayedActions.once('sync', function () {
      // nodes with all permitted actions are available
    })
    .fetch();
```

setCommands(names) : void

Asks for one or more commands to be checked. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have two commands checked, option 1
model.setCommands(['delete', 'reserve']);
// Have two commands checked, option 2
model.setCommands('delete');
model.setCommands('reserve');
```

```
// Have two commands checked, option 3
model.setCommands('delete,reserve');
```

resetCommands(names) : void

Prevents one or more commands from being checked. The `names` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all commands will be removed (not to be checked).

```
// Cancel all command checks and fetch the fresh data
model.resetCommands();
model.fetch();
```

setDefaultActionCommands(names) : void

Asks for one or more commands to be checked immediately by the first server call. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array. The rest of commands specified by the `setCommands` method will be fetched later by the second call.

```
// Have two commands checked, option 1
model.setCommands(['browse', 'download']);
// Have two commands checked, option 2
model.setCommands('browse');
model.setCommands('download');
// Have two commands checked, option 3
model.setCommands('browse,download');
```

resetDefaultActionCommands(names) : void

Prevents one or more commands from being checked immediatley by the first server call. The `names` parameter can be either string, or an array of strings, or nothing. The string can contain a comma-

delimited list, in which case it will be split to an array. If nothing is specified, all commands will be removed (not to be checked by the first server call).

```
// Have all commands fetched delayed by the second server call
model.setCommands(['browse', 'download']);
model.resetDefaultActionCommands();
model.fetch();
```

getRequestedCommandsUrlQuery() : string

Formats the URL query parameters for the command investigation. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`. If delayed action fetching is enabled, only the default actions will be returned by this method; the rest of specified actions fill be fetched by an additional server call later.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getRequestedCommandsUrlQuery()
    );
if (query) {
  url = Url.appendQuery(url, query);
}
```

setEnabledDelayRestCommands(enabled, promoted) : void

Enables or disables the delayed command fetching and promoted actions. The current state can be seen by `delayRestCommands` and `promoteSomeRestCommands` properties.

See Also

`ConnectableMixin`, `CommandableV2Mixin`

opentext™

ExpandableV2Mixin

Provides support for the setting `expand` URL query parameter as introduced by the `api/v2/nodes/:id` or `api/v2/nodes/:id/nodes` (V2) resources.

Server responses can contain references to other resources; typically IDs or URLs. The *expansion* means replacing them with object literals containing the resource information, so that the caller does not have to request every associated resource by an additional server call.

Expanding needs the role to expand from (`properties`, `versions` etc.) and optionally the name or names of properties to expand (`parent_id`, `create_user_id` etc.).

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeExpandableV2(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getExpandableResourcesUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});

ConnectableMixin.mixin(MyModel.prototype);
ExpandableV2Mixin.mixin(MyModel.prototype);
```

opentext™

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How to use the mixin

Set up the URL parameters by calling `setExpand` and `resetExpand` and fetch the model:

```
// Set the expansion when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      expand: {
        properties: ['parent_id', 'create_user_id']
      }
    });
model.fetch();

// Set the expansion after creating the model
model.setExpand('properties', ['parent_id', 'create_user_id']);
model.fetch();
```

makeExpandableV2(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

expand : One or more resources to expand. Keys and values from the object literal are handled the same way as the `setExpand` method does the key and value (role and properties). An empty object literal is the default.

expand

Resources to get expanded in the response (object literal of strings pointing to arrays of strings, empty by default, read-only).

opentext™

hasExpand(role) : boolean

Checks if a specific or any resource will be expanded. The `role` parameter is a string.

```
// Checks if any resource is expanded
var anythingExpanded = model.hasExpand();
// Checks if any common properties are expancded
var propertiesExpanded = model.hasExpand('properties');
```

setExpand(role, names) : void

Makes one or more resources expanded. The `role` parameter is a string. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have two resources expanded, option 1
model.setExpand('properties', ['parent_id', 'create_user_id']);
// Have two resources expanded, option 2
model.setExpand('properties', 'parent_id');
model.setExpand('properties', 'create_user_id');
// Have two resource types expanded, option 3
model.setExpand('properties', 'parent_id,create_user_id');
```

resetExpand(role, names) : void

Prevents one or more resources from being expanded. The `role` parameter is a string. If nothing is specified, all roles will be removed (disabled). The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all properties from the role will be removed (disabled).

```
// Cancel all expansions and fetch the fresh data
model.resetExpand();
model.fetch();
```

opentext™

getExpandableResourcesUrlQuery() : string

Formats the URL query parameters for the resource expansion. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getExpandableResourcesUrlQuery()
    );
if (query) {
  url = Url.appendQuery(url, query);
}
```

See Also

`ConnectableMixin`, `ResourceMixin`

opentext™

FieldsV2Mixin

Provides support for the setting `fields` URL query parameter as introduced by the `api/v2/members/favorites` or `api/v2/members/accessed` (V2) resources.

Server responses can contain various properties, which can increase the response size, if all of them were always returned.

Adding a field needs the role, which contains them (`properties`, `versions` etc.), and optionally their names too (`parent_id`, `create_user_id` etc.).

How to apply the mixin to a model

```
var MyModel = Backbone.Model.extend({

  constructor: function MyModel(attributes, options) {
    Backbone.Model.prototype.constructor.apply(this, arguments);
    this
      .makeConnectable(options)
      .makeFieldsV2(options);
  },

  url: function () {
    var url = Url.combine(this.connector.connection.url, 'myresource'),
        query = Url.combineQueryString(
          this.getResourceFieldsUrlQuery()
        );
    return query ? url + '?' + query : url;
  }

});

ConnectableMixin.mixin(MyModel.prototype);
FieldsV2Mixin.mixin(MyModel.prototype);
```

opentext™

This mixin us usually combined together with the `ConnectableMixin` or with another cumulated mixin which includes it.

How to use the mixin

Set up the URL parameters by calling `setFields` and `resetFields` and fetch the model:

```
// Set the expansion when creating the model
var model = new MyModel(undefined, {
      connector: connector,
      fields: {
        properties: ['parent_id', 'create_user_id']
      }
    });
model.fetch();

// Set the expansion after creating the model
model.setFields('properties', ['parent_id', 'create_user_id']);
model.fetch();
```

makeFieldsV2(options) : this

Must be called in the constructor to initialize the mixin functionality. Expects the Backbone.Model or Backbone.Collection constructor options passed in.

Recognized option properties:

fields : One or more properties to include. Keys and values from the object literal are handled the same way as the `setFields` method does the key and value (role and properties). An empty object literal is the default.

fields

Fields to include in the response (object literal of strings pointing to arrays of strings, empty by default, read-only).

opentext™

hasFields(role) : boolean

Checks if a specific field or any fields will be included. The `role` parameter is a string.

```
// Checks if any fields are included
var anyFields = model.hasFields();
// Checks if any common property fields are included
var propertyFields = model.hasFields('properties');
```

setFields(role, names) : void

Adds one or more fields to the response. The `role` parameter is a string. The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array.

```
// Have two fields added, option 1
model.setFields('properties', ['parent_id', 'create_user_id']);
// Have two fields added, option 2
model.setFields('properties', 'parent_id');
model.setFields('properties', 'create_user_id');
// Have two fields added, option 3
model.setFields('properties', 'parent_id,create_user_id');
```

resetFields(role, names) : void

Removes one or more fields from the response. The `role` parameter is a string. If nothing is specified, all roles will be removed (not returned). The `names` parameter can be either string, or an array of strings. The string can contain a comma-delimited list, in which case it will be split to an array. If nothing is specified, all properties from the role will be removed (not returned).

```
// Cancel all expansions and fetch the fresh data
model.resetFields();
model.fetch();
```

opentext™

getResourceFieldsUrlQuery() : string

Formats the URL query parameters for the field addition. They can be concatenated with other URL query parts (both object literals and strings) by `Url.combineQueryString`.

```
var url = ...,
    query = Url.combineQueryString(
      ...,
      this.getResourceFieldsUrlQuery()
    );
if (query) {
  url = Url.appendQuery(url, query);
}
```

See Also

`ConnectableMixin`

opentext™

NodeAddableTypeCollection

Loads types of nodes, which can be added to the specified container by the authenticated user. When creating a new instance, you need to pass an instance of the `NodeModel to the constructor to specify the target container.

AddableTypeModel Attributes

type (integer) : Subtype number of the node type

type_name (string) : Displayable name of the node type

Examples

```
// Fetch node types which can be added to the specific container
var connector = new Connector({
      connection: {
        url: '//server/instance/cs/api/v1',
        supportPath: '/instancesupport'
      }
    }),
    parent = new NodeModel({
      id: 12345
    }, {
      connector: connector
    }),
    addableTypes = new NodeAddableTypeCollection(undefined, {
      node: parent
    });
addableTypes.fetch()
    .done(function () {
      console.log(addableTypes.pluck('type_name'));
    });
```

opentext™

See Also

Node Model,

opentext™

NodeModel

Provides read/write access to CS nodes. When creating a new instance, you need to pass an instance of the `Connector` to the constructor to connect it to the server.

Nodes can be fetched single or within a collection like container children, for example. The `NodeModel` supports initialization from the server responses, which have the same response format as the core CS REST API. finishes.

Examples

```
// Fetch a concrete node and print its name on the console
var connector = new Connector({
      connection: {
        url: '//server/instance/cs/api/v1',
        supportPath: '/instancesupport'
      }
    }),
    node = new NodeModel({
      id: 12345
    }, {
      connector: connector
    });
node.fetch()
    .done(function () {
      console.log(node.get('name'));
    });

// Create a model for the Enterprise Volume by its subtype
var enterpriseVolume = new NodeModel({
      id: 'volume',
      type: 141
    }, {
      connector: connector
    });
```

opentext™

NodeChildren2Collection

Loads children of a CS node container using the V2 REST API. When creating a new instance, you need to pass an instance of the NodeModel to the constructor to specify the parent node.

Examples

```
// Fetch children of a concrete node and print their count on the console
var connector = new Connector({
      connection: {
        url: '//server/instance/cs/api/v1',
        supportPath: '/instancesupport'
      }
    }),
    parent = new NodeModel({
      id: 12345
    }, {
      connector: connector
    }),
    children = new NodeChildren2ollection(undefined, {
      node: parent
    });
children.fetch()
        .done(function () {
          console.log(children.length);
        });
```

See Also

Node Model, Browsable Interface

NodeColumn2Collection

Collects schemas of node properties:

```
{
  "column_key": "...",
  "name": "...",
  "type" ...,
  "sort": ...
}
```

or keys of custom columns:

```
{
  "column_key": "..."
}
```

See `node.definitions` and `node.columns` in `NodeModel`.

ToolItemMaskCollection

Lists all registered tool item masks. The models are initialized with the `id` attribute pointing to the module name, which provides the tool item mask and accepts Require.js configuration to populate the mask.

```
var toolItemMasks = new ToolItemMaskCollection();
toolItemMasks.each(function (mask) {
  console.log(mask.id);
});
```

The collection is populated by Require.js configuration from the extension configuratoin (`prefix-extension.json` files).

```
{
  "nuc/models/tool.item.mask/tool.item.mask.collection": {
    "masks": {
      "csui": [
        "csui/widgets/nodestable/toolbaritems.masks",
        "csui/widgets/nodestable/headermenuitems.mask"
      ],
      "conws": [
        "conws/widgets/related.workspaces/toolbaritems.masks"
      ]
    }
  }
}
```

opentext™

Authenticators

Common Methods

unauthenticate(options) : void

Discards authentication parameters by removing the authentication ticket or other means of authentication from the authenticator instance. The next AJAX call will require a new authentication request.

The `options` parameter should contain the property `reason` pointing to a string with the reason for unauthenticating:

- "logged-out" - an explicit logout has been initiated by the user. The window will be directed to the logout/login page automatically.
- "expired" - an invalid authentication ticket was detected, outgoing AJAX calls were suspended and a new authentication token is about to be requested, automatically or by a login form.
- "failed" - obtaining an authentication token has failed unrecoverably and the page will remain unusable.

This method trigges an event "loggedOut" on the authenticator propagating the `reason` property in the event arguments.

If the reason is "expired", the "loggedOut" event may be followed by the "loggedIn" event, as soon as a fresh authentication ticket has been obtained.

Base Classes

These authenticators are meant to inherit from. They can be used to implement custom retrieval of authentication headers; either the built-in ticket or a custom one.

opentext™

Authenticator

Gives the freedom to implement any means of authentication.

RequestAuthenticator

Provides functionality for authenticators, which make an initial AJAX call and obtain some authentication information. Usually a header value.

Final Classes

These authenticators are meant to be instantiated and passed to a new connector instance. They can be chosen automatically by `Connector` depending on the connection parameters.

BasicAuthenticator

Uses Basic Authentication to authenticate all AJAX calls.

```
"credentials": {
  "username": "...",
  "password": "..."
}
```

CredentialsAuthenticator

Uses user credentials for the initial call obtaining OTCSTicket and then continues using the ticket.

```
"credentials": {
  "username": "...",
  "password": "..."
}
```

InitialHeaderAuthenticator

opentext™

Uses a custom authentication header for the initial call obtaining OTCSTicket and then continues using the ticket.

```
"authenticationHeaders": {
  ...
}
```

InteractiveCredentialsAuthenticator

Opens a modal dialog, as soon as a request is issued, which needs authentication. The dialog will ask for user credentials and perform the same login as `CredentialsAuthenticator` to obtain the authentication ticket.

RegularHeaderAuthenticator

Uses a custom request header to authenticate all AJAX calls.

```
"headers": {
  ...
}
```

TicketAuthenticator

Uses the built-in OTCSTicket header to authenticate all AJAX calls.

```
"session": {
  "ticket": "..."
}
```

opentext™

Base

This module holds miscellaneous static functions.

autoAlignDropDowns(inputElement, dropdownContainer, applyWidth, view, callback)

Align drop-downs either top or bottom for the inputElement based on available space.

Parameters

inputElement: The element object relative to which the drop-dwon will be shown

dropdownContainer: The drop-down container element object

applyWidth: True for applying inputElement width to drop-down elements, false for applying custom specific width

view: inputElement view

callback: Is used to perform any actions in the view after form scroll / window scroll

opentext™

Module nuc/util/connector

This module contains a utility object for the Content Server connection - `Connector`.

Connector

Utility object for the Content Server connection.

Widgets create this object internally to establish a connection to the Content Server, but it can be created and passed to the widget from the client code too, if there are multiple widgets on the page and they should share the same server connection.

Example:

```
// Ensure initialization of the framework.
csui.onReady(function() {

  // Specify the server connection settings.
  var connection = {
        url: "//server/otcs/cs/api/v1",
        supportPath: "/otcssupport"
      },
      // Create the server connecting object.
      connector = new csui.util.Connector({
        connection: connection
      }),
      // Pass the connector to a widget.
      folderbrowser = new csui.widget.FolderBrowserWidget({
        connector: connector,
        start: {id: 2000}
      }),
  // Display the widget.
  folderbrowser.show({placeholder: '#target'});

});
```

opentext™

Constructor

Creates a new connector instance.

Parameters:

- options - object literal with initial settings

Properties of `options`:

- connection - object with the API url and other parameters of the server connection
- headers - object with HTTP headers to add to every request

Properties of `connection`:

- url - string pointing to the URL root of the REST API, for example: //server/otcs/cs/api/v1/
- supportPath - string with the URL path to the static resources, usually the CGI path with the "support" suffix: "/otcssupport" here //server/otcs/cs/api/v1/
- session - object with `ticket` property containing a string obtained from the server to authenticate the API requests
- credentials - object with `username`, `password` and `domain` properties to authenticate the API requests with
- authenticationHeaders - object with properties for HTTP headers which are provide the authentication information

The minimum connection settings include the `url` and `supportPath` properties. The authentication will take place interactively as soon as the first server call will take place:

```
connection: {
  url: '//server/otcs/cs/api/v1',
  supportPath: '/otcssupport'
}
```

The connection settings can contain an access token (ticket) to use an already pre-authenticated session. The ticket can be obtained by the `/auth` API request handler:

```
connection: {
  url: '//server/otcs/cs/api/v1',
  supportPath: '/otcssupport',
  session: {
    ticket: '...'
  }
}
```

The connection settings can also contain user credentials for the automatic authentication:

```
connection: {
  url: '//server/otcs/cs/api/v1',
  supportPath: '/otcssupport',
  credentials: {
    username: 'guest',
    password: 'opentext',
    domain: '' // optional
  }
}
```

At last, the connection settings can contain HTTP headers which are acepted by a server side login callback to authenticate the API request:

```
connection: {
  url: '//server/otcs/cs/api/v1',
  supportPath: '/otcssupport',
  authenticationHeaders: {
    OTDSTicket: '...'
  }
}
```

Returns:

The newly created object instance.

Example:

See the `Connector` object for an example.

makeAjaxCall(options) : promise

Replacement for the direct usage of `$.ajax` in Smart UI.

The best practice to exchange the data with the server is using Backbone and implementing the communication by overriding methods "url" and "parse". But sometimes it is difficult to find a CRUD mapping for a functional API request. In that case, use `connector.makeAjaxCall` instead of `$.ajax`. It will take care of the following scenarios, which `$.ajax` does not cover:

- Refresh the authentication ticket automatically, once it expires.
- Support mocking with mockjax.
- Enable running on other platforms, than CS.
  - Force GET requests to pass their data always as URL parameters.
  - Ensure content type "multipart/form-data" if `FormData` is used.
  - Change content type "application/json" to "application/x-www-form-urlencoded" and wrap request body to satisfy the non-standard CS REST API request format.

The `options` parameter and the returned promise are the same as `$.ajax` describes them. Structured request and respone data are supposed to be handled always as JSON objects.

```
// Get basic node information
connector.makeAjaxCall({
  url: '.../api/v1/nodes/123',
  data: {
    fields: ['properties']
  }
}).then(function (response) {
  console.log(response);
}).catch(function (error) {
  console.error(error);
});

// Get current user information
connector.makeAjaxCall({
  url: '.../api/v1/auth'
```

```
});

// Get user picture
connector.makeAjaxCall({
  url: '.../api/v1/members/123/photo',
  dataType: 'binary'
});

// Rename a node
connector.makeAjaxCall({
  type: 'PUT'
  url: '.../api/v1/nodes/123',
  data: {name: 'New name'}
})

// Add a new version
var data = new FormData();
data.append('file', file);
connector.makeAjaxCall({
  type: 'POST'
  url: '.../api/v1/nodes/123/versions',
  data: data
})
```

opentext™

Load extensions

Require.js plugin to load extension modules listed in the module config of the specified module. The extension modules are supposed to perform self-registration as expected by the extensible module.

How to configure extra cell views to show custom columns in the table, which is supported by the extensible module `csui/controls/table/cells/cell.factory`:

```
require.config({
  config: {
    'csui/controls/table/cells/cell.factory': {
      extensions: {
        // Module array should be wrapped in a map key to allow merging
        // of extensions from multiple calls to require.config; arrays
        // are not merged
        'samples': [
          'samples/cells/reserved/reserved.view'
          'samples/cells/social/social.view'
        ]
      }
    }
  }
});
```

How the extensible module `csui/controls/table/cells/cell.factory` loads all configured extensions:

```
define([
  ..., 'csui-ext!csui/controls/table/cells/cell.factory'
], function (...) {
  // Before the callback is executed, 'samples/cells/reserved/reserved.view'
  // and 'samples/cells/social/social.view' are loaded and executed
  ...
});
```

opentext™

Log

Configurable logging.

Synopsis

The default logger uses the global configuration:

```
require(['nuc/utils/log'], function (log) {
  log.info('Module loaded.') && console.info(log.last);
});
```

Log factory allows configuring the log level on a module basis:

```
require(['module', 'nuc/utils/log'], function (module, log) {
  log = log(module.id);
  log.debug('Module {0} loaded.', module.id) && console.log(log.last);
});
```

The targets where to log to and the default log level threshold can be set globally:

```
require.config({
  config: {
    'nuc/utils/log': {
      page: true,
      level: 'DEBUG'
    }
  }
});
```

The log level threshold can be changed for any specific module:

```
require.config({
  config: {
    'nuc/utils/log': {
      modules: {
        'csui/controls/table/table.view': { level: 'DEBUG' }
      }
    }
  }
});
```

The log level, timing and deprecation logging flags, both global and module-specific, can be updated anytime after loading the page:

```
log = csui.require('nuc/utils/log');
log.set({
  deprecated: true,
  modules: {
    'csui/controls/table/table.view': { level: 'DEBUG' }
  }
});
```

Usage

```
error(message: string [, ...parameters]): boolean
warn(message: string [, ...parameters]): boolean
info(message: string [, ...parameters]): boolean
debug(message: string [, ...parameters]): boolean
```

Issues a log message with the log level according to the method name, which will be logged, if enabled by the configuration. The parameter `message` can be either a formatted message or a message format with parameter placeholders `{N}`, where `N` is a zero-based index of the parameter. Returns `true` is the message was logged and should be logged on the console too. The property `last` contains the formatted message.

```
deprecate(message: string [, ...parameters]): boolean
```

Issues a warning message if the `deprecated` flag is enabled in the configuration. Otherwise is the usage the same es for the `warn` method.

```
can(level: string): boolean
```

Check if the current log level is above the configured threshold and logging is enabled. It means that a message with that log level will be logged. The log levels are `'ERROR'`, `'WARN'`, `'INFO'` and `'DEBUG'`.

```
last: string
```

The last formatted log message.

```
time(scope: string): void
```

Starts measuring time in a named scope, if enabled by the configuration. Writes directly to the console.

```
timeEnd(scope: string): void
```

Ends measuring time in a named scope and prints the time duration on the console, if enabled by the configuration. Writes directly to the console.

```
getObjectName(instance: object): string
```

Returns the name of the function object (class) of the specified object instance.

```
getStackTrace([frameCountToSkip: number]): string
```

Returns the call stack at the place where it is called, without the call to `getStackTrace` itself. The parameter `frameCountToSkip` can be used to omit further frames from the top.

opentext™

Configuration Parameters

Most of the configuration parameters have to be set before the first module gets loaded. Later changes will have no effect to the most of them. See the next chapter for more information. The parameters and their default values:

```
console: true,               // enable logging on the console
consoleRe: false,            // enable logging using https://console.re
level: 'WARN',               // change the global log level threshold
timing: false,               // enable the timing log entries
deprecated: false,           // enable the deprecation warnings
page: false,                 // enable logging to a bottom page area
performanceTimeStamp: false, // prefix every message by the issued time
moduleNameStamp: false,      // prefix every message by the originating module name
server: false,               // enable posting log messages to the server
window: false,               // enable logging to a new browser window
modules: {}                  // change the log level threshold for specific modules
```

The log levels are `'ERROR'`, `'WARN'`, `'INFO'` and `'DEBUG'`. The log level threshold can be set from the highest `'ERROR'` to the lowest `'DEBUG'`. Log messages with the log level higher or equal to the theshold will be actually logged, the others will be skipped.

If a log level threshold is set for a specific module, it will override the global value for log messages issued from the specific module. The parameter `modules` expects an object with the following content:

```
{
  '<module name>': { level: string, timing: boolean, deprecated: boolean },
  ...
}
```

The parameter `server` can be either `false` or an object with the following parameters, including their default values:

```
url: undefined      // the URL to post log messages to
batchSize: 10,      // post the log messages in batches instead of single
timed: true,        // enables regular flushing incomplete log batches
```

```
timerInterval: 500, // set the interval of flushing incomplete log batches,
headers: {}         // set additional HTTP headers for the log requests
```

The `url` is mandatory. It will receive `'POST'` requests with objects, for example:

```
{
  "logger": "[default]",
  "timeStamp": 1201048234203,
  "level": "ERROR",
  "url": "http://server/otcs/cs/app",
  "message": "Object 12345 could not be accessed."
}
```

Configuration Updates

Only the following parameters can be modified after the this module got loaded:

```
level: 'WARN',              // change the global log level threshold
timing: false,              // enable the timing log entries
deprecated: false,          // enable the deprecation warnings
modules: {}                 // change the log level threshold for specific modules
```

The initial configuration can be changed by:

```
set({ level, timing, deprecated, modules }): void
reset({ level, timing, deprecated, modules }): void
```

Calling `set` will change the value of the particular parameter, either globally, or for the specified modules.

Calling `reset` will apply the defaults (as shown above) for the affected parameters at first and then it will apply the changed parameters on top of them.

The initial configuration can be re-applied by calling:

```
reload(): void
```

This method will discard changes made by `set` and `reset` and restore the configuration set by `require.config` only.

Beyond Console

If logging is enabled for other targets that `console`, it will be supported by log4javascript.

Setting `consoleRe` will be supplied by Console.Re. Setting `page` will be supplied by InPageAppender and PatternLayout. Setting `server` will be supplied by AjaxAppender and JsonLayout. Setting `window` will be supplied by PopUpAppender and PatternLayout.

The module `nuc/lib/log4javascript` will be loaded on demand, which means that first log messages may be delayed. Before the module `nuc/lib/log4javascript` gets loaded, the log messages will be logged only on the console, if it is enabled. Once the module `nuc/lib/log4javascript` gets loaded, earlier messages will be sent to the logger. They will not get lost. If all log messages have to be sent to the selected targets really right away, the module `nuc/lib/log4javascript` can be forced to be loaded ahead, for example:

```
require.config({
  paths: {
    smart: '/support/smart',
    csui: '/support/csui'
  },
  config: {
    'nuc/utils/log': {
      page: true,
      level: 'DEBUG'
    }
  },
  deps: [
    'require',
    'nuc/lib/require.config!nuc/nuc-extensions.json',
    'nuc/lib/require.config!smart/smart-extensions.json',
    'nuc/lib/require.config!csui/csui-extensions.json',
```

```
    'nuc/lib/log4javascript'
  ],
  callback: function (require) {
    require([...], function (...) {
      ...
    });
  }
})
```

opentext™

PageLeavingBlocker

Prevents the user from leaving the current page without being warned, that some running operation would be interrupted and pending items would not be processed. If the blocker is enabled, the web browser is supposed to show a confirmation dialog with the specified message and a question if the user is really sure, when the user tries to:

- navigate to other page.
- reload the current page.
- close the current window or tab.
- quit the web browser application.

Warning: Mobile browsers may not support the page-leave warning. Neither of Safari, Chrome and Firefox on iOS supports it today. The page will be left as soon as the user attempts it.

The blocker should be enabled providing the warning message before a long-running multiple-item processing operation starts and disabled, when all items have been processed. Calls to `enable` and `disable` methods have to be paired, otherwise the warning would be shown any time later, when the user would try to leave the page.

Examples

Processing multiple items *without* the page-leaving blocker:

```
require(['nuc/lib/jquery', 'nuc/utils/page.leaving.blocker',
  'nuc/lib/jquery.when.all'
], function ($, PageLeavingBlocker) {

  // Returns a promise for all items and works in the background
  function processItems(items) {
    var promises = items.map(processItem);
    return $.whenAll.apply($, promises);
  }
```

```
  // Returns a promise for the single item and works in the background
  function processItem(item) {
    ...
  }

  processItems(...);

});
```

Processing multiple items *with* the page-leaving blocker:

```
require(['nuc/lib/jquery', 'nuc/utils/page.leaving.blocker',
  'nuc/lib/jquery.when.all'
], function ($, PageLeavingBlocker) {

  // Returns a promise for all items and works in the background
  function processItems(items) {
    // Enable the blocker before processing of the first item begins
    PageLeavingBlocker.enable(
      'If you leave the page now, pending items will not be processed.');
    var promises = items.map(processItem);
    return $.whenAll
        .apply($, promises)
        // Disable the blocker after processing of the last item ends
        .always(PageLeavingBlocker.disable);
  }

  // Returns a promise for the single item and works in the background
  function processItem(item) {
  }

  processItems(...);

});
```

Methods

isEnabled() : boolean

Returns `true` if the page-leaving blocker is enabled, otherwise `false`.

enabled(message) : void

Enables the page-leaving blocker. The `message` string will be displayed, if the user tries to leave the current page. The message passed the the most recent call to `enable` will be displayed. Every call to `enable` has to be paired with a call to `disable`.

disable() : void

Cancels the previous `enable` call and if it was the last one, it isables the page-leaving blocker. Every call to `enable` has to be paired with a call to `disable`.

Date Parsing and Formatting

This module provides methods for parsing and formatting date and time values.

opentext™

Localizable Strings

This module holds functions for performing string operations with localized strings or multi-lingual objects. They use the current application's locale, which is accessible via the "nuc/lib/i18n" module.

getClosestLocalizedString(value, fallback) : string

Returns a localized text picked from the multilingual value for the closest language to the current locale settings.

The multilingual value is an object literal where keys are locales or languages and values are localized texts:

```
{
  "<locale or language>": "<localized text>",
  ...
}
```

For example:

```
{
  "en":    "carbon-fiber",
  "en-br": "carbon-fibre",
  "de":    "Kohlefaser"
}
```

Locale identifier has the format `<language>-<country>` where `language` is a two-letter code from ISO-639-1 and `country` is a two-letter code from ISO-3166-1. Locale matching is case-insensitive.

The CS REST API uses different format for locale identifiers, which occur in multilingual metadata - underscore server as the separator instead of hyphen (`<language>_<country>`). It is recognized by this method as well to allow easy integration, but do not use such locale identifiers in your code and

configugarion. They are invalid for components which parse locale identifiers using RFC-5646 and RFC-4647 standards. You would have to convert your identifiers all the time.

Parameters

value (object literal or any) : Multilingual value as a map, where keys are locales or languages and values are localized texts, or any value which woudl be converted to a string (optional; an empty string - `''` - is the default)

fallback (any) : A string to return if no fitting locale can be found in the `value` map.

Result

The function returns always a string; if you do not pass an object literal as the `value`, it will return `value.toString()` or `''` if the value is `null` or `undefined`. You do not need to check if the input `value` is multilingual; you can pass normal strings to the function as well.

Example

```
var multilingualLabel = {
      'en': 'carbon-fiber',
      'de': 'Kohlenfaser'
    },
    // Receives 'carbon-fiber' for the 'en-US' locale
    label = base.getClosestLocalizedString(multilingualLabel);
```

locale*String (...) : ...

These string operations work similarly to comparison operators and function in the `String` prototype, but they use rules for character comparing, changing case and transliterating as expected for the current application's locale.

localeCompareString(left, right, options): -1|0|1

opentext™

Compares two strings according to the current locale and returns the result of the following operation: `left < right ? -1 : left > right ? 1 : 0`.

String comparisons may use different rules, depending on the usage scenario; like searching or sorting, for example. Specify your intention using the `options` parameter:

```
{usage: 'search'} - you are searching or filtering
{usage: 'sort'}   - your are sorting
```

localeContainsString(hay, needle): boolean

Works like a locale-specific `String.prototype.contains`.

localeEndsWithString(full, end): boolean

Works like a locale-specific `String.prototype.endsWith`.

localeIndexOfString(hay, needle): number

Works like a locale-specific `String.prototype.indexOf`.

localeStartsWithString(full, start): boolean

Works like a locale-specific `String.prototype.startsWith`.

formatMessage (count, messages, ...) : string

Chooses a string format from `messages` according to the supplied `count` and formats a string with `sformat` from "nuc/lib/underscore.string" using the `count` and optionally other arguments. The argument `messages` is expected to have the following structure:

```
{
  formatForNone: 'expression for count == 0',
  formatForOne: 'expression for count == 1',
```

```
    formatForTwo: 'expression for 2 <= count <= 4',
    formatForFive: 'expression for count >= 5'
  }
```

If the key `formatForNone` is missing, `formatForFive` will be used instead of it.

Grammars of different languages use different rules how to declinate a substantive, which comes after a number representing a count. The count value usually changes the extension of the substantive, related adjectives, or sometimes the entire expression.

Example

```
English     Czech           Format
-----------------------------------------------------------
no file     žádný soubor    none (formatForNone, optional)
 1 file     1 soubor        one  (formatForOne)
 2 files    2 soubory       some (formatForTwo)
 3 files    3 soubory
 4 files    4 soubory
 5 files    5 souborů       many (formatForFive)
 6 files    6 souborů
 ...
 N files    N souborů       many (formatForFive)
 0 files    0 souborů       many (formatForFive, fallback)
```

Member Name Formatting (nuc/utils/types/member)

This module provides methods for formatting names of users and user groups.

opentext™

Number Formatting (nuc/utils/types/number)

This module provides methods for formatting numeric values.