Noonian Documentation

Release 0.9.1

Eugene Newall-Lockett

Jan 26, 2021

CONTENTS:

1 Development Guide **3**
- 1.1 Introduction . . . 3
- 1.2 Getting Started . . . 3
- 1.3 A Tour of the DBUI . . . 5
- 1.4 Data Definition . . . 10
- 1.5 Basics of Building a DBUI Application . . . 14
- 1.6 Adding Business Logic . . . 15
- 1.7 Adding Users and Roles . . . 16
- 1.8 Installing and Building Packages . . . 16
- 1.9 Documentation . . . 17
- 1.10 Application and Unit Testing . . . 17
- 1.11 Logging . . . 17
- 1.12 Extended DBUI Features . . . 17
- 1.13 Building a Non-DBUI Angular Application . . . 18
- 1.14 Advanced Customization . . . 18
- 1.15 Indices and tables . . . 18

2 Noonian Reference **21**
- 2.1 Admin, Configuration, and Deployment . . . 21
- 2.2 Development Reference . . . 24
- 2.3 Noonian Core API . . . 28

3 Indices and tables **43**
- 3.1 Change Log . . . 43
- 3.2 GITLAB README . . . 43

Index **45**

This documentation is divided into two parts:

Development Guide

A guide meant to be read in sequence to learn how to develop the system.

Reference Materials

The detailed technical details you'll probably need to refer back to while developing on the system.

Todo: We need a User Guide: general DBUI usage for non-developer end-users of applications built from the DBUI. Showing how to use features such as the query editor, Perspective/column editor, CSV export, ...

CHAPTER

ONE

DEVELOPMENT GUIDE

This section is will walk you through the key concepts and structures needed for developing an application with Noonian.

1.1 Introduction

1.1.1 What is it?

- It is a completely *Free (Libre) MEAN*-based integrated **platform** for rapid development of full-featured browser-based applications.
- It is a **framework** for defining data objects for persistence, and for creating business logic and user interface around those objects.
- It is its own browser-based **development environment** for full-stack javascript applications.

1.1.2 What can I do with it?

- Define your data objects using meaningful field types.
- Edit those objects within a rich, customizable UI that allows you to create a fully-functional *CRUD* interface with a few lines of JSON.
- Build your application's UI entirely based on that built-in UI by adding buttons, triggers, web services, user roles and permissions, and custom pages.
- AND/OR build your application's UI to be completely distinct, leveraging Noonian to organize the Angular components, and for its data APIs and web services.

1.2 Getting Started

This section will guide you through the process of installing Noonian and its dependencies, and setting up a fresh instance.

These instructions should generally apply to Linux, Windows, and MacOS hosts.

1.2.1 Installation

The dependencies to required by Noonian are:

1. Node.js
2. MongoDB
3. bower

Mongo DB

Follow the instructions in the MongoDB documentation to get MongoDB installed on your system.

Ideally, you will be able to perform a system-wide installation and run it as a service. However, if you do not have root access to the machine on which you are running, it is possible to run it from a directory under user home directory.

Node.js

Package managers provide the easiest way to get the latest version installed on your system. The nodejs website provides a comprehensive list of available packages for most operating system hosts.

The Node.js installation includes the Node Package Manger *npm <https://npmjs.com>*, which will be used for the rest of the installation process.

Bower

At the commandline, install bower globally using npm:

```
npm install -g bower
```

If you performed a system-wide node install, you'll need to perform the above command as the root or administrative user.

Noonian

At the commandline, install noonian globally with npm:

```
npm install -g noonian
```

NOTE: if you are installing to a system-wide node install (as opposed to a node installation that is owned by a non-root user), there may be issues installing dependencies compiled via node-gyp. To get around the issue, use the "unsafe-perm" parameter:

```
npm install --unsafe-perm -g noonian
```

This will be resovled in a future version that will have refactored out any dependencies that are not pure javascript.

1.2.2 Instance Setup

First, create a directory for your instance, and make it your current working directory.

```
mkdir my-noonian-instance
cd my-noonian-instance
```

Then use the Noonian CLI to initialize the directory as a noonian instance:

```
noonian init my-instance
```

This will create a new file **instance-config.js**, which may be edited to configure the instance.

Edit the configuration file, and then use the CLI to bootstrap the database for the instance:

```
noonian bootstrap -p adminPassword
```

This will initialize the configured database with the core system, setting the initial password for admin to "adminPassword".

Use the CLI to start the instance:

```
noonian start
```

This will start the instance as a background service (forked process), and direct stdout and stderr to stdout.log and stderr.log.

You may stop it with the CLI as well:

```
noonian stop my-instance
```

(if your current working directory is the instance directory, the "my-instance" can be omitted)

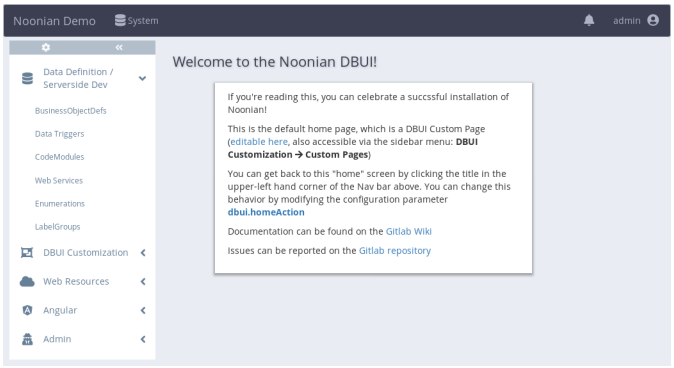
1.3 A Tour of the DBUI

Noonian's graphical user interface is referred to as **DBUI**: an initialism for "DataBase User Interface".

The DBUI is what you see when you log in, and provides all of the screens for viewing and editing Business Objects in the system. It provides a rich interface for managing your data, and is designed to serve as a starting point for your database application.

This section will provide a brief introduction to the overall layout and some basic features built in to the DBUI.

When you first log in to a fresh instance, you are greeted with the default home screen:



At the top we've got the **navbar**, on the left we've got the **sidebar**, and the majority of the screen is our main content area.

1.3.1 The Navbar

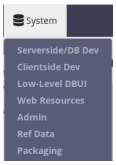
The top bar we'll refer to as the **navbar** (this is by convention; in actuality most navigation in a Noonian system occurs via the sidebar)

Starting from the left, the navbar components are:

Page Title

Displays the configured instance name. When clicked, takes user to the configurable "home" screen, which is the same screen displayed when the user first logs in.

System Menu



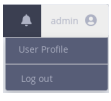
To the right of the title, the configured dropdown menus are displayed. Menus can be customized based on user role.

The **System** menu is the default navbar menu for the system admin role, and shows logical groupings of all *Business Object* classes in the system.

Notifications

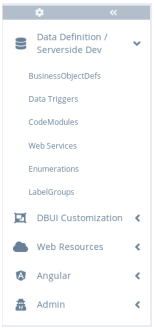
The bell icon on the right-hand side of the navbar contains a list of alert messages that the system has displayed to the user during their session.

User Menu



The rightmost icon on the navbar is for user profile management (password reset, etc.) and log-out.

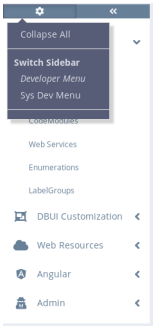
1.3.2 The Sidebar



The sidebar is the primary mechanism by which the user navigates a Noonian system. The content of the sidebar is customizable based on user role.

The default sidebar for the system admin role is the *Developer Menu*, providing access to the components used for developing a Noonian application.

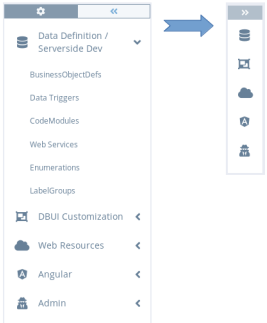
Gear Menu



The gear icon on the upper left-hand side of the sidebar allows the user to select which sidebar menu is displayed.

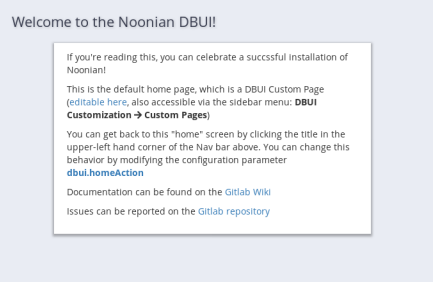
Additionally, it provides an option to collapse all submenus on the sidebar.

Minimize and Expand



The double chevron icon on the top right of the sidebar provides a way to reduce the size of the sidebar. Clicking it reduces the display to show only icons for each submenu title. In minimized mode, a submenu is exposed when the mouse hovers over its submenu icon.

1.3.3 Home Screen



When the user initially logs in, the main content area is populated with a configurable home screen. This is the same screen that is shown when the user clicks the title on top left of the navbar.

The default home screen on a fresh instance provides information on how to customize it.

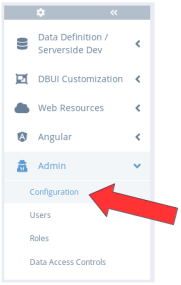
1.3.4 List, Edit, and View Business Objects

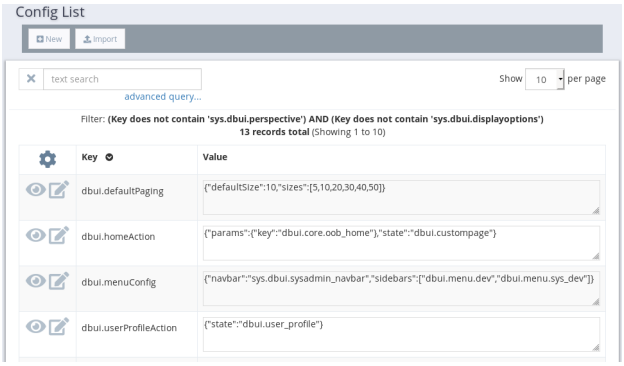
The core functionality of the Noonan DBUI is in editing, viewing and querying *Business Objects* in the database. There are 3 primary screens for doing so: **list**, **edit**, and **view**.

List

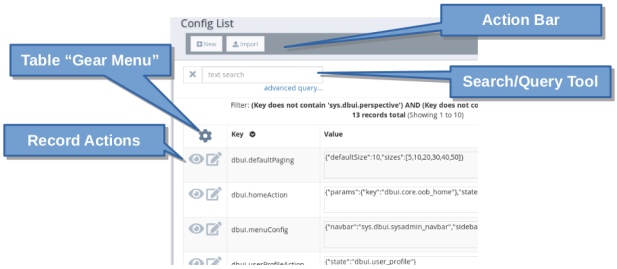
The **list** screen displays a list of Business Objects in the system. It includes search and query tools and configurable action buttons.

It happens that all of the links on the Developer Menu actually link to a **list** screen. As an example, let's take a look at the configuration list, since there are a number of objects to be seen on a fresh instance:





You see a filtered list of **Config** Business Objects in the system, and several buttons and tools.



Action Bar Shows buttons for configured actions for the current *perspective*.

Search/Query Tool Allows for a quick text search, or opening the Query Builder to perform more sophisticated query on the data.

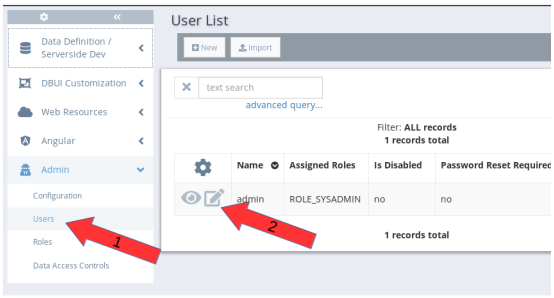
Gear Menu Shows a dropdown to perform table-related actions, such as editing the displayed columns, or refreshing or exporting the data.

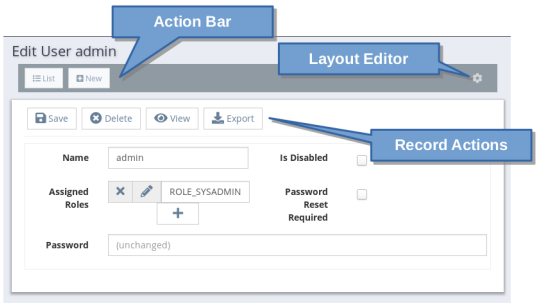
Record Actions Shows record-specific actions that can be invoked for a specific row. The **View** and **Edit** actions are generally present for the default list *perspectives*.

Edit

The **edit** screen displays a form for editing a single Business Object. It includes layout editing tools and configurable action buttons.

As an example, let's take a look at an **edit** screen for a User Business Object:





Action Bar Shows buttons for configured actions for the current *perspective*.

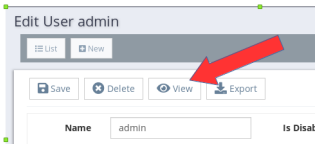
Layout Editor Shows a dialog that allows the user to change the fields displayed and the layout of the form.

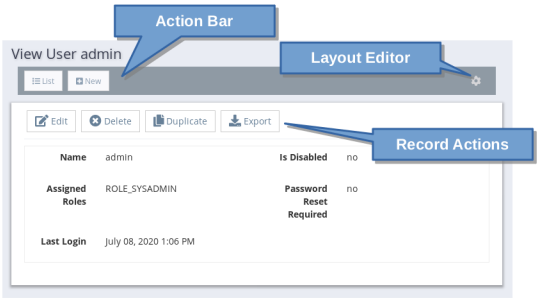
Record Actions Shows record-specific actions that can be invoked for the object being edited. The **Save**, **Delete**, **View**, and **Export** actions are generally present for the default edit *perspectives*.

View

The **view** screen is very similar to an edit screen, the obvious difference being the editability of the displayed Business Object.

As an example, let's look at the corresponding **view** for the User object we examined above. Use the **View** button from the edit screen:





Action Bar Shows buttons for configured actions for the current *perspective*.

Layout Editor Shows a dialog that allows the user to change the fields displayed and the layout of the form.

Record Actions Shows record-specific actions that can be invoked for the object being edited. The **Edit**, **Delete**, **Duplicate**, and **Export** actions are generally present for the default view *perspectives*.

1.4 Data Definition

1.4.1 The Business Object

Business Objects are the basic building blocks of any Noonian application. Defining a Business Object is akin to defining a database table in the SQL world, and resembles class definition in the Object Oriented Programming world.

In Noonian, you define the the data schema for your application by creating a set of **Business Object Definitions**.

When you create a Business Object Definition, you're defining structure for a particular class of Business Object that you want to persist.

That class of Business Object you defined translates to several components in a Noonian system:

1. A corresponding collection in MongoDB that contains the persisted objects
2. An API within the server-side Node.js environment for querying, reading, and updating the objects
3. An API provided by an Angular.js service that allows you to query, read and update from the client-side.
4. A set of screens for viewing, querying, and updating the objects within the Angular.js front-end, called the DBUI.

1.4.2 Creating a Business Object Definition

Expand the sidebar menu **Data Definition / Serverside Dev**; and select **BusinessObjectDefs**

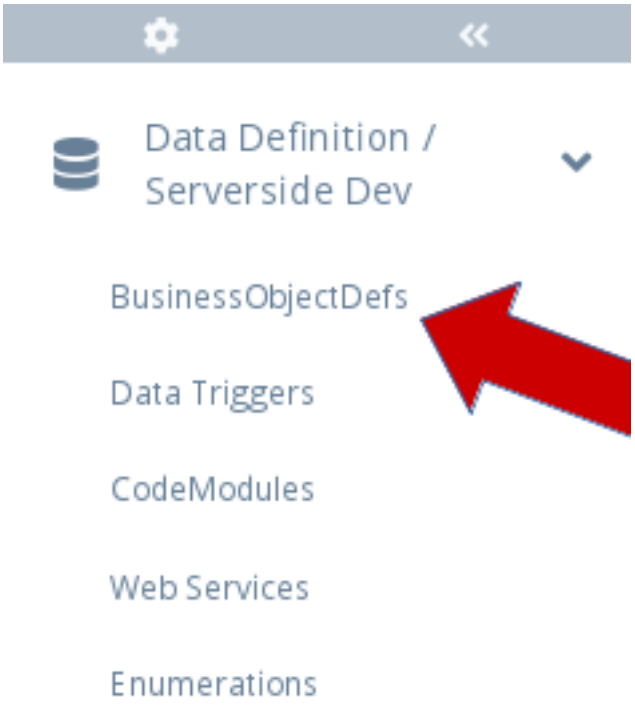


Fig. 1: BusinessObjectDefs Menu

You are shown the list of BusinessObjectDef's in the system. Select the **New** button at the top to create a new one.

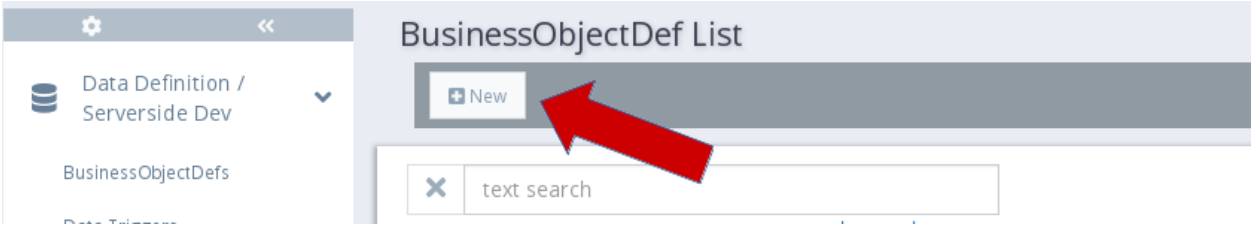


Fig. 2: BusinessObjectDef List -> New

You are shown a form that allows you to specify the fields for your new Business Object Definition. Let's demonstrate with a basic Person class:

The definition is simply a JSON object mapping each field name to its respective **Type Descriptor** :

```
{
    "name":"string",
    "birthday":"date",
    "phone_number":"phone",
    "address":"physical_address",
    "profile_picture":"image",
    "wears_glasses":"boolean",
    "number_of_children":"integer",
    "misc_notes":"text",
```

(continues on next page)

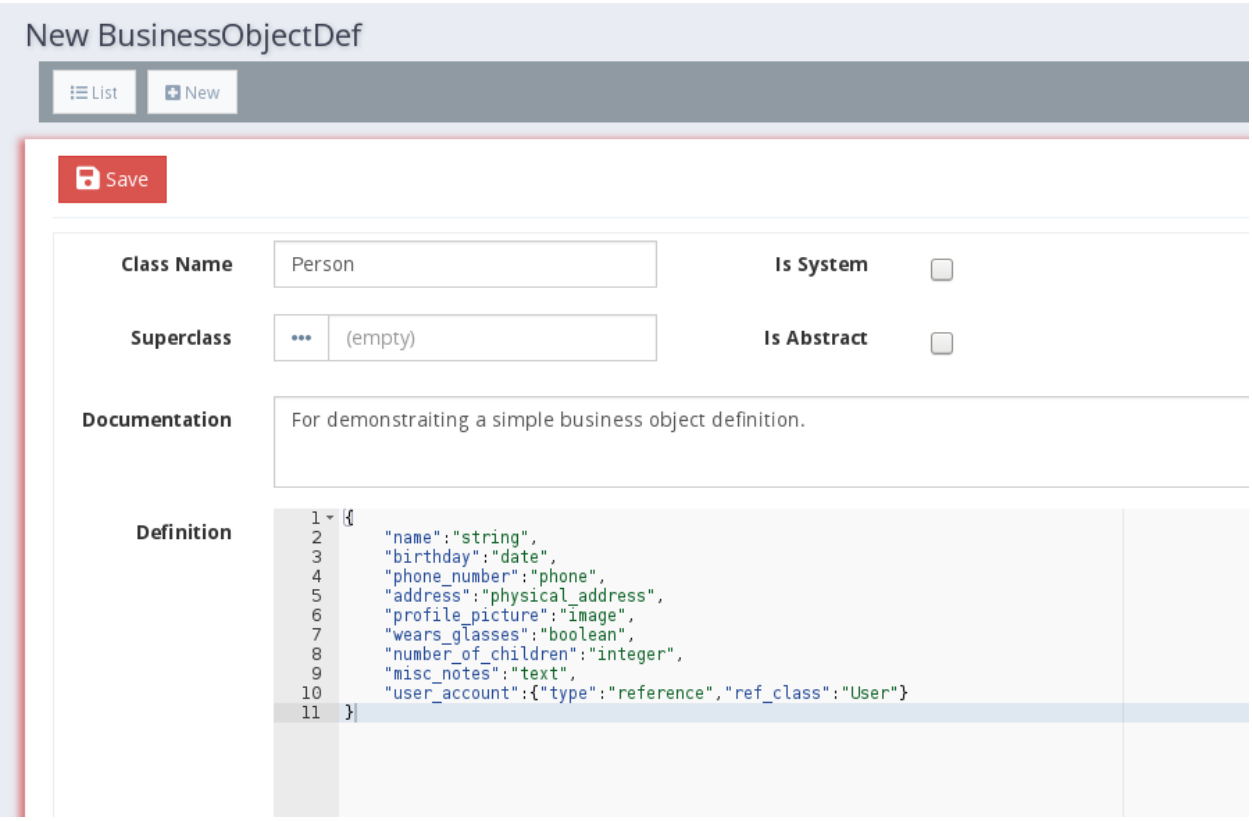


Fig. 3: New BusinessObjectDef

(continued from previous page)

```
    "user_account":{"type":"reference","ref_class":"User"}
}
```

Notice in the example, most of the types are described simple strings: "string", "date", "boolean", etc. This is a shorthand for those types that require no more than a single string. The user_account field is the exception: its type is "reference", so we need to specify what class of objects it should reference.

Also perhaps you've noticed how the fields "name" and "misc_notes", have the types "string" and "text", and you're wondering the purpose of the distinction - isn't a "text" field is just a long string? Indeed it is! However, by calling it a "text" field, we are 1) creating a more precise description, 2) telling the display logic to use the larger text block for editing it, and 3) telling the system to index it differently.

Let's go ahead and save the BusinessObjectDef. Notice how those types get expanded after you save:

```
{
  "name": {
    "type": "string"
  },
  "birthday": {
    "type": "date"
  },
  "phone_number": {
    "type": "phone"
  },
  "address": {
    "type": "physical_address"
  },
  "profile_picture": {
    "type": "image"
  },
  "wears_glasses": {
    "type": "boolean"
  },
  "number_of_children": {
    "type": "integer"
  },
  "misc_notes": {
    "type": "text"
  },
  "user_account": {
    "type": "reference",
    "ref_class": "User"
  }
}
```

When you save the a BusinessObjectDef, any single-string values get replaced by a full **Type Descriptor**. The **Type Descriptor** fully describes a field's type, and at its simplest consists of a single "type" property. Other properties on a Type Descriptor add neccessary detail in describing the fields type. Some are mandatory (e.g. "ref_class" for a reference field) , and others are optional (e.g. you may specify a minimum or maximum value of an integer field with a min or max property.)

Please see the reference page for Field Types for a complete list of available field types and their respective type descriptor properties.

1.4.3 Indexing

coming soon

Todo: pending implementation of BusinessObjectDef index specification

1.5 Basics of Building a DBUI Application

This section will guide you through the key concepts needed to build an application in the DBUI.

1.5.1 Perspectives and Layouts

The primary screens in the DBUI are the List, Edit, and View screens. When we are looking at one of these screens, we are in fact looking at Business Object data from a particular *perspective*.

A ***perspective*** has a name, and contains the information about what fields to show and how to lay them out, what actions and buttons to display, any filtering and sorting to apply, and so on.

You can always tell the name of the perspective you're looking at by looking at the end of the URL:

/list/BusinessObjectClass/PERSPECTIVE_NAME

/view/BusinessObjectClass/OBJECT_ID/PERSPECTIVE_NAME

/edit/BusinessObjectClass/OBJECT_ID/PERSPECTIVE_NAME

List Perspectives

View and Edit Perspectives

Editing Perspective Data Directly

Perspectives are stored in the database as ***Config*** Business Objects, with keys named as follows:

sys.dbui.perspective.PERSPECTIVE_NAME.BusinessObjectClass

Perspective data can be viewed and edited directly by going through the DBUI Developer Menu, under ***DBUI Customization*** -> ***Perspectives***

Wildcard Perspectives

1.5.2 Menus

TODO

1.5.3 Custom Pages

How to create one

How to navigate via menu item

How to navigate via uiRouter state change

1.5.4 UI Actions

1.5.5 Configuration

Todo: Configuration relevant to development-guide; focus should be on using it within your code, not what specfic keys are for and how user/role customization works (link to sys and dbui2 package reference for that)

Customization

Todo: sys pacakge reference - dev guide should link where needed

Role-based

User-based

Important Configuration Items

Todo: The detailed reference for DBUI configuration items belongs w/in the doc for the actual sys.dbui2 package

Instance Name

Home Action

Navbar Menu

Sidebar Menu

Default Paging Settings

1.6 Adding Business Logic

This section is will walk you through the structures available to you when developing business logic in your Noonian application.

1.6.1 Fundamentals

Developing the server-side business logic in a Noonian application involves creating business objects containing your code.

1.6.2 Data Triggers

TODO

1.6.3 Web Services

TODO

1.6.4 Code Modules

TODO

1.6.5 Member Functions

TODO

1.6.6 Schedule Triggers

TODO

1.7 Adding Users and Roles

1.7.1 Menus and Perspectives

TODO

1.7.2 Data Access Controls

TODO

1.8 Installing and Building Packages

TODO

1.9 Documentation

Most of the Business Objects you create in the process of building an application have a **Documentation** field you can use to include relevant documentation for the respective components. These fields support JSDoc and reStructuredText, and can be used to generate API and development documenation for your Noonian applications.

TODO

Todo: pending implementation of Noonian Doc Generation

..todo:: Application Diagrams add-on

1.10 Application and Unit Testing

Noonian integrates Jasmine, Selenium, and Protractor for creating unit tests for your applications.

TODO

Todo: Pending implementation of Jasmine/Selenium/Protractor integration

---

1.11 Logging

TODO

1.12 Extended DBUI Features

This section takes you through several useful DBUI features not yet mentioned in this guide.

1.12.1 DBUI Pivot Page

TODO

1.12.2 Reporting

TODO

1.12.3 Internationalization

TODO

1.13 Building a Non-DBUI Angular Application

The guide thus far has show how it is possible to build a full-featured application by extending and customizing the DBUI. This section describes how you can use Noonian to build an Angular.js application that is not based on DBUI. In this scenario, it is still possible to leverage the DBUI and Noonian's data layer and APIs for your application.

A practical example of a use case might be a lightweight Angular-based front-end that leverages Noonian for the data layer and business logic, and the DBUI as an administrative back-end.

Note: A basic understanding of Angular.js compoenents is assumed for this section

1.13.1 Angular Components

1.14 Advanced Customization

This section is describes how to extend the fundamental objects of Noonian and the DBUI.

1.14.1 Custom Field Types

TODO

1.14.2 Custom Query Operations

TODO

1.15 Indices and tables

- *Glossary*
- genindex
- modindex
- search

1.15.1 Glossary

Noonian Terminology

Business Object A persisted data object.

DBUI The graphical user interface of Noonian: "DataBase User Interface"

Perspective A configuration object that describes how to render a DBUI screen: layout, table columns, action buttons, etc.

RecordAction An action within the DBUI that is tied to a Business Object when invoked.

Related Jargon

CRUD The fundamental database operations: Create Read Update Delete

Free Software Free as in "Free Speech" (liberty), not as in "Free Beer" (price)

MEAN A stack of technologies for web development that consists of Javascript and JSON for both front and back-end: M = MongoDB E = Express A = AngularJS N = Node.js

CHAPTER

TWO

NOONIAN REFERENCE

Detailed technical documentation for Noonian development and administration.

2.1 Admin, Configuration, and Deployment

2.1.1 Instance Configuration

Configuration for a Noonian instance is stored in a javascript file in the base of the instance directory: **instance-config.js**. A default is generated from a template when an instance is initialized using the CLI.

Configuration File

Required Parameters

At instance initialization, all required parameters are populated/generated with reasonable defaults.

instanceName

A name used for referring to the instance via the CLI. Should be unique across instances on a server for the CLI to work properly.

instanceId

An identifier used in versioning the persisted business objects. It should be short; by default it matches the instance name.

serverListen

TCP Port and address on which to bind. On initialization, the Noonian CLI examines the config files for all configured instances on the machine and chooses a port number by adding 1 to the highest one found.

```
serverListen: {
  port: 11100,
  host: '127.0.0.1'
}
```

mongo

MongoDB connection string and options. Initialization defaults to localhost

```
mongo: {
  uri: 'mongodb://localhost/noonian-myinstance'
}
```

secrets

Secret string used for signing the auth tokens for authentication. Initialization generates a random UUID.

```
secrets: {
  session: '1f6885e7-36a8-4dd9-909e-d585e5bd0879'
}
```

Logging

By default, log level is set to **debug**, writing logs to instance.log (all log messages) and error.log (only error messages).

See *Configuration* for more details on logging configuration.

Optional Parameters

urlBase (string)

A path under which the noonian instance will be served.

Use urlBase if noonian will be served to a path other than the root of the domain. For example, one domain can host multiple noonian instances as such: mydomain.com/instance1, mydomain.com/instance2

enableBackRefs

Enable "back reference" processing on reference fields.

enableHistory

Instance Directory Structure

The files and directories in the instance directory are described below.

- **instance-config.js**
- **client** - base directory for static content served from the filesystem by the webserver (this directory is checked BEFORE client directory of the core app listed above) * **bower_components** - holds bower dependencies of packages installed on this instance
- **node_modules** - holds any npm dependencies of packages installed on this instance
- *[pkg-key]* - if package building with filesystem sync is enabled, data updates are synced to this directory

2.1.2 Command Line Interface

```
Usage: noonian [options] [command]

Options:
  -V, --version                        output the version number
  -h, --help                           output usage information

Commands:
  start [options] [instanceName]       launch an instance
  stop [options] [instanceName]        stop a running instance
  restart [options] [instanceName]     restart a running instance
  env [options] [instanceName]         show environment variables for launching an␣
↪instance (returns source-able list of variable export lines)
```

(continues on next page)

(continued from previous page)

```
  list                                  list configured instances
  status                                list the currently-running Noonian instances
  startall                              Start all of the configured Noonian instances
  init [options] <instanceName>         initialize directory for a instance
  bootstrap [options] [instanceName]    bootstrap database for an instance
  add [options]                         add an existing instance to the index
  remove <instanceName>                 remove an instance from the instance index
  dbdump [options] [instanceName]       call mongodump to create a dump of the database
  dbshell [options] [instanceName]      start mongo shell for the instance's database
  open [options] [instanceName]         launch browser window
  watch [options] [instanceName]        watch stdout and stderr of an instance
  pm2-eco                               generate pm2 ecosystem JSON (print to stdout)
```

Instance Set-up

init, bootstrap, add, remove

Process Management

The CLI contains basic functionality for managing Noonian instances on the local machine. These features are meant for use in a development envrionment. For a production deployment, either a Docker container or a more sophisticated process management tool such as PM2 is recommended.

Start / Stop / Restart

Status

PM2 Ecosystem Generation

Utilities

Database Dump

Database Shell

Get Environment Variables

Installing Bash Autocompletion

The Noonian CLI has a bash auto-completion script so commands and instance names can be auto-completed at the command prompt.

If you have root access to the system, you can create a symbolic link in **/etc/bash_completion.d** to the NOONIAN_HOME/template/bash_completion.d

```
ln -s /path/to/installation/node_modules/noonian/template/bash_completion.sh /etc/
↪bash_completion.d/noonian
```

If you do not have root access, you can simply source the bash_completion.sh in your .bashrc. (Alternatively, refer to this discussion for other options)

2.1.3 Deployment

This document describes various options for production deployment of a Noonian instance.

PM2

Run Noonian as a service using PM2

Proxy through Apache

Docker

There are multiple scenarios for deploying Noonian using docker:

- All-in-one, single instance
- All-in-one, multi-instance
- External MongoDB, single-instance Noonian container
- External MongoDB, multi-instance Noonian container

All-in-one Container

Full Node + MongoDB + Noonian stack in one image/container.

Docker-Compose Stack

Two separate containers: one MongoDB, one Node.js/Noonian

2.2 Development Reference

This documentation provides detailed technical information for Noonian development to supplement the API documentation.

2.2.1 Data Model

This section covers defining Business Objects as well as using the server-side persistence layer for accessing and manipulating those objects.

Business Object Definition

The data layer of a Noonian application is defined by creating a collection of BusinessObjectDef objects.

Persistence API

The API is modelled after the Mongo shell, wherein each collection in the database corresponds to a property on the **db** object, and that property contains the full *CRUD* interface for working with that collection:

```
//Mongo shell
db.SomeCollection.find(...)
```

The **db** API in Noonian functions in a similar fashion. Each Business Object Definition (BusinessObjectDef) in the system corresponds to a property on the **db** object, and that property contains the full *CRUD* interface for working with those Business Objects:

```
//Server-side Noonian code
db.SomeBusinessObject.find(...).then(resultList=>{...})
```

These properties on the **db** object are called the **Business Object Models**, and are in fact augmented instances of **Model** from the mongoose library. This documentation will cover the most important functions, but the full API can be referenced on the mongoose project website (the current version of Noonian utilizes mongoose v5.9).

Differences from Mongo and Mongoose

An understanding of the MongoDB shell operations and the mongoose API translates to an almost complete understanding of the Noonian persistence API. This section describes where Noonian differs from one or the other.

Noonian and Mongoose are Promise-Based

The objects returned when running operations in the mongo shell generally provide synchronous access to the data. For example:

```
> u = db.User.findOne({name:'admin'});
> print(u.name);
admin
>
```

Noonian/Mongoose run asynchronously utilizing promises:

```
db.User.findOne({name:admin}).then(u=>{
   console.log(u.name);
});
```

Noonian and Mongoose are Schema-Based

Construct objects instead of using "insert".

Metadata

Inheritance

Other Internals

- Data Triggers
- Field pre/post processing
- ID Generation

Creation/Insertion

Query/Lookup

Deletion

Aggregation Pipeline

2.2.2 Indexing

TODO

2.2.3 Query Ops

TODO

2.2.4 References

TODO

2.2.5 Config

Server/instance config specifications

Detailed description of all "sys." configuration items, but not the dbui ones

sys.urlConfig sys.two_factor_auth sys.password_complexity

Todo: clearly deliniate noonian core from the sys package: core should be refactored to put core-specific config (e.g. the above 3) into instance-config; config that drives functionality within the sys package should be in noonian config

---

2.2.6 Logging

Noonian integrates the winston package for javascript logging, and provides functionality for creating and configuring a hierarchy of named loggers.

Configuration

Basic Configuration

Basic logging configuration can be set in instance-config.js using either the key **logLevel** or **loggers**.

logLevel

Simplest configuration, specifying only the log level. Possible values are:

- *error*
- *warn*
- *info*
- *verbose*
- *debug*
- *silly*

All log messages are written to the file **instance.log**, and error messages are written to the file **error.log**.

loggers

Todo: Allows for more control over which levels go to which targes, and some customization of log message formatting.

Advanced Configuration

Todo: For more control over logging transports and formatting, the file logging-config.js in the instance directory may be used to incorporate arbitrary winston loggers into Noonian.

Logger Naming

Loggers can be named in a similar hierarchical fashion as is done with apache log4j.

The name of the logger is included in messages written by that logger.

Todo: Loggers can be enabled, disabled, and configured by name.

2.2.7 Auth

TODO

2.2.8 Data Export

TODO

2.2.9 GridFS

TODO

2.2.10 Packaging

2.2.11 Web Sockets

TODO

2.3 Noonian Core API

This section documents the API for the code belonging to the Noonian server-side **core**. The **core** is that code that is installed when you run *npm install*.

2.3.1 Authorization and Authentication

Injectable as: **auth**

Provides functionality around user authentication, role checking, and data access permissions.

getCurrentUser(*req*)
Gets the User BusinessObject for the user currently logged in

Arguments

- **req** – the request object (Express JS)

Returns **Promise.<User>** – resolving to user account associated with the request, or false if none

updateUserPassword(*req*, *newPassword*)
Update password for a user; checks configured password complexity requirements if applicable.

Arguments

- **req** – the request object (Express JS)
- **newPassword** (*string*) –

Returns **Promise.<{"success"}>** –

getCurrentUserRoles(*req*)
Get roles for current user.

Arguments

- **req** – the request object (Express JS)

Returns **Promise.<Array.<string>>** – array of Role ids

checkRolesForUser(*user*, *rolespec*, *noShortCircuit*)

Check roles of a user account to determine if fullfills a role specficiation. If user has ROLE_SYSADMIN, it will automatically succeeds unless noShortCircuit is enabled.

Arguments

- **user** (*BusinessObject.<User>*) – user Business Object
- **rolespec** (*Array.<string>*) – list of role ids
- **noShortCircuit** (*boolean*) – don't short-circut check for ROLE_SYSADMIN

Returns **boolean** – true if user passes role check

checkRoles(*req*, *rolespec*)

Check roles of a user account to determine if fullfills a role specficiation. If user has ROLE_SYSADMIN, it will automatically succeed.

Arguments

- **req** – the request object (Express JS)
- **rolespec** (*Array.<string>*) – list of role ids

Returns **boolean** – Promise<> resolves to true on pass; rejects on failure

aggregateReadDacs(*req*, *TargetBoModel*)

Pulls together **Read** DACs that apply to TargetBoModel and the current user's roles.

Arguments

- **req** – the request object (Express JS)
- **TargetBoModel** – the Business Object model

Returns **Promise.<{{condition, fieldRestrictions}}>** – A promise resolving to aggregated DataAccessControl

aggregateUpdateDacs(*req*, *TargetBoModel*)

Pulls together **Update** DACs that apply to TargetBoModel and the current user's roles.

Arguments

- **req** – the request object (Express JS)
- **TargetBoModel** – the Business Object model

Returns **Promise.<{{condition, fieldRestrictions}}>** – A promise resolving to aggregated DataAccessControl

aggregateCreateDacs(*req*, *TargetBoModel*)

Pulls together **Create** DACs that apply to TargetBoModel and the current user's roles.

Arguments

- **req** – the request object (Express JS)
- **TargetBoModel** – the Business Object model

Returns **Promise.<{{condition, fieldRestrictions}}>** – A promise resolving to aggregated DataAccessControl

aggregateDeleteDacs(*req*, *TargetBoModel*)

Pulls together **Delete** DACs that apply to TargetBoModel and the current user's roles.

Arguments

- **req** – the request object (Express JS)
- **TargetBoModel** – the Business Object model

Returns **Promise.<{{condition, fieldRestrictions}}>** – A promise resolving to aggregated DataAccessControl

checkCondition(*condObj*, *targetObject*)
Utility to check a query condition against an object; used for conditional DACs

Arguments

- **condObj** (*object*) –
- **targetObject** (*object*) –

checkReadDacs(*req*, *TargetBoModel*, *query*)
Check **Read** DACs for a given user, BusinessObject class, and query. Returns a promise that either: a) Resolves to a query that has been modified to restrict access based on applicable DACs, or b) Rejects when DACs don't allow read access to the requested TargetBoModel

Arguments

- **req** – the request object (Express JS)
- **TargetBoModel** – the Business Object model
- **query** (*object*) – the original query

Returns **Promise** –

2.3.2 Data Source

Injectable as: **db**

The server-side api for retrieving and updating *Business Objects*.

db contains all of the Business Object Models for interfacing with the persistence layer, as described in *Data Model*

Todo: Document the Business Object Model's enhanced mongoose API in mongoose_intercept and add a bomodel.rst

---

generateId()
Generates a random UUID and returns in URL-safe base64.

Returns **string** – new v4 URL-safe base64-encoded UUID

_svc
Contains references to datasource submodules: field types, data triggers, query operations, gridFS access, reference processing, and package building/installation.

db._svc.FieldTypeService

Contains logic pertaining to FieldType objects: mongoose schema elements, calling to/fromDb function.

Todo: this is really low-level functionality probably not suitable for published API

fieldtypes.**init**()
Initialize FieldType cache from DB.

Returns **promise** – fullfilled upon completion of caching

augmentTypeDescMap(*tdm*)
Augments a type descriptor map's composites and arrays: flag composites, pull in definitions for named composites, and recursively process (Mutates the provided object; Used in initialization of BO Metadata.)

Arguments

- **tdm** (*object*) – type descriptor map to process

Returns **object** – the same object that was passed in

class MongooseSchemaWrapper()

Arguments

- **textIndex** (*boolean*) – indicates whether text index is configured for the FieldType
- **type** (*function*) – the constructor object passed to mongoose

getSchemaElem(*typeDescriptor*)
Convert typeDescriptor object from BOD into mongoose schema element.

Arguments

- **typeDescriptor** (*object*) – the type descriptor to convert

Returns **MongooseSchemaWrapper** – the object used by the Mongoose schema for a field w/ provided typeDescriptor

getFieldTypeHandler(*typeDescriptorOrId*)
Get the actual FieldType object by ID or by type descriptor

Arguments

- **typeDescriptorOrId** (*object | string*) – type descriptor object or field type name

Returns **FieldType** –

processToDb(*modelObj*)
Preprocess fields for persistence to mongo using the respective FieldType's to_db function

Arguments

- **modelObj** (*BusinessObject*) – object to process

processFromDb(*modelObj*)
Preprocess fields coming from mongo using the respective FieldType's from_db function

Arguments

- **modelObj** (*BusinessObject*) – object to process

db._svc.DataTriggerService

All functionality to respond to data-update events to DataTriggers.

Todo: this is really low-level functionality probably not suitable for published API

datatrigger.**init**()
Initialize datatrigger servce

Returns **Promise** – resolving when init is complete

registerDataTrigger(*key*, *bodId*, *beforeAfter*, *onCreate*, *onUpdate*, *onDelete*, *actionFn*, *priority*)
Register a system-level trigger(triggers that are not DataTrigger business objects)

Arguments

- **key** (*string*) – key under which to register
- **bodId** (*string*) – id of BusinessObjectDef to which trigger applies; if null, applies globally
- **beforeAfter** (*string*) – 'before' or 'after'
- **onCreate** (*boolean*) – trigger on create
- **onUpdate** (*boolean*) – trigger on update
- **onDelete** (*boolean*) – trigger on delete
- **actionFn** (*function*) – function to invoke
- **priority** (*number*) – relative priority

refreshDataTriggers()
Reload all DataTriggers from the database

Returns Promise resolving when reload is complete

processBeforeCreate(*modelObj*, *keyFilter*, *saveOptions*)
Process triggers for pre-create

Arguments

- **modelObj** (*BusinessObject*) – object for which to process triggers
- **keyFilter** (*string*) – regular expression string; process only those triggers whose key matches
- **saveOptions** (*object*) – object passed to save() function; injected into DataTriggers' action function

Returns **Promise** – resolves to: map of DataTrigger key to return/resolve value from respective action invication

processBeforeUpdate(*modelObj*, *keyFilter*, *saveOptions*)
Process triggers for pre-update

Arguments

- **modelObj** (*BusinessObject*) – object for which to process triggers
- **keyFilter** (*string*) – regular expression string; process only those triggers whose key matches

- **saveOptions** (*object*) – object passed to save() function; injected into DataTriggers' action function

Returns Promise – resolves to: map of DataTrigger key to return/resolve value from respective action invication

processBeforeDelete(*modelObj*, *keyFilter*, *saveOptions*)
Process triggers for pre-delete

Arguments

- **modelObj** (*BusinessObject*) – object for which to process triggers
- **keyFilter** (*string*) – regular expression string; process only those triggers whose key matches
- **saveOptions** (*object*) – object passed to save() function; injected into DataTriggers' action function

Returns Promise – resolves to: map of DataTrigger key to return/resolve value from respective action invication

processAfterCreate(*modelObj*, *keyFilter*, *saveOptions*)
Process triggers for post-create

Arguments

- **modelObj** (*BusinessObject*) – object for which to process triggers
- **keyFilter** (*string*) – regular expression string; process only those triggers whose key matches
- **saveOptions** (*object*) – object passed to save() function; injected into DataTriggers' action function

Returns Promise – resolves to: map of DataTrigger key to return/resolve value from respective action invication

processAfterUpdate(*modelObj*, *keyFilter*, *saveOptions*)
Process triggers for post-update

Arguments

- **modelObj** (*BusinessObject*) – object for which to process triggers
- **keyFilter** (*string*) – regular expression string; process only those triggers whose key matches
- **saveOptions** (*object*) – object passed to save() function; injected into DataTriggers' action function

Returns Promise – resolves to: map of DataTrigger key to return/resolve value from respective action invication

processAfterDelete(*modelObj*, *keyFilter*, *saveOptions*)
Process triggers for post-delete

Arguments

- **modelObj** (*BusinessObject*) – object for which to process triggers
- **keyFilter** (*string*) – regular expression string; process only those triggers whose key matches
- **saveOptions** (*object*) – object passed to save() function; injected into DataTriggers' action function

Returns **Promise** – resolves to: map of DataTrigger key to return/resolve value from respective action invication

db._svc.QueryOpService

Funcationality around queries, query clauses and conditions

`query.`**`init`**`()`
Initialize service from QueryOp objects in the DB

Returns **Promise** – resolving when init complete

`getQueryOpList`(*forType*)
Look up relevant QueryOps for a given type

Arguments

- **`forType`** (*`string`*) – field type

Returns **Array.<QueryOp>** –

`satisfiesCondition`(*modelObj*, *condition*)
Evaluate a query condition against a model object

Arguments

- **`modelObj`** (*`BusinessObject`*) – object to evaluate against
- **`condition`** (*`object`*) – the query condition

Returns true iff modelObj satisifes query condition

`applyNoonianContext`(*queryObj*, *context*)

Search for any $noonian_context objects within the queryObj, replace with values from actual context.
Keys mapping to an object such as {$noonian_context:'dotted.spec.string'} will be mapped to context.dotted.spec.string *Mutates queryObj!*

Arguments

- **`queryObj`** (*`object`*) –
- **`context`** (*`object`*) –

Returns **null** –

`queryToMongo`(*queryObj*, *boMetaData*)
Process any custom query operators to create a query for mongodb *Mutates queryObj!*

Arguments

- **`queryObj`** (*`object`*) –
- **`boMetaData`** (*`object`*) –

Returns **null** –

`stringifyQuery`(*queryObj*, *boMetaData*, *fieldLabels*, *noonianContext*)
Convert query object to human-readable string

Arguments

- **`queryObj`** (*`object`*) –
- **`boMetaData`** (*`object`*) – bo_meta_data of object to which queryObj applies

- **fieldLabels** (*object*) – map field name to label
- **noonianContext** (*object*) – context to apply to labels

Returns **string** –

db._svc.GridFsService

Functionality around dealing with reading and writing files to/from Mongo's gridfs. Wraps gridfs API adding noonian metadata, id's and logic.

Todo: need to refactor away from gridfs-stream: https://www.npmjs.com/package/mongoose-gridfs Updated mongo api supports streaming: https://mongodb.github.io/node-mongodb-native/3.1/tutorials/gridfs/streaming/ ancient bug report: https://github.com/aheckmann/gridfs-stream/issues/125

Todo: API is a little sloppy; saveFile creates a file by passing a readstream, writeFile creates a file by retreiving a writeStream

---

gridfs.**init**()
Initialize gridfs servce

Returns **Promise** – resolving when init is complete

saveFile(*readStream*, *metadata*)
Stream a file to gridfs

Arguments

- **readStream** (*stream.Readable*) – node readable stream containing contents of file to write
- **metadata** (*Object*) –

Returns **Promise.<string>** – file id that can later be used to getFile;

writeFile(*metadata*)
Open a write stream to a file; augments metadata with an attachment_id

Arguments

- **metadata** (*Object*) –

Returns **stream.Writable** – stream that receives file contents

getFile(*fileId*)
Retreive a file by id

Arguments

- **fileId** (*string*) –

Returns **Promise.<{{readstream:stream.Readable, metadata:object}}>** –

deleteFile(*fileId*)
Delete a file from gridfs

Arguments

- **fileId** (*string*) –

Returns **Promise.<{{result:'success', file:fileId}}>** –

annotateIncomingRef (*fileId*, *boClass*, *boId*, *field*)
Add metadata to track incoming reference to a file

Arguments

- **fileId** (*string*) –
- **boClass** (*string*) – class name of BusinessObject containing reference
- **boId** (*string*) – id of BusinessObject containing reference
- **field** (*string*) – field name containing reference

Returns null

getAllFileMetadata ()
Get all files in fs.files

Returns **Promise.<object>** – object mapping filenames to metadata.

exportFile (*attachmentId*, *outPath*)
Export an attachment from gridfs to the filesystem

Arguments

- **attachmentId** (*string*) –
- **outPath** (*string*) – fully-qualified filename of target file

Returns **Promise** – resolving to outPath when file write is completed

importFile (*inPath*, *contentType*)
Import a file from the filesystem

Arguments

- **inPath** (*string*) – fully-qualified path to file to import
- **contentType** (*string*) – ContentType of given file. if not provided, file extension is used to detect

Returns **Promise.<object>** – resolving to metadata object which can be assigned to a field of type "attachment"

cleanup ()
Scan the database for files in gridfs that do not have any references from attachment fields, and delete them.

Returns **Promise.<{{deleteCount:number, deleted:object}}>** –

db._svc.RefService

Functionality around processing reference fields

Todo: this is really low-level functionality probably not suitable for published API

references.**init** (*conf*)
Initialize reference service

Returns **Promsie** – resolving when initialization is complete

repair ()
Traverse all BusinessObjects in the system and catalog all references Rebuild IncomingRefModel table and update all reference fields so that _disp and denormalized fields are up-to-date

Returns **Promise** – resolving when repiar is complete

db._svc.PackagingService

Functionality around building, installing, versioning Noonian packages.

installBootstrapPackages()
Install packages identified in configuration "bootstrap"

Returns **Promise** – resolving when package is complete

Tracking Updates for Package Content

updateLogger(*isCreate*, *isUpdate*, *isDelete*)
Create an UpdateLog entry for an update. If filesystem sync is enabled, write files for the object

Arguments

- **isCreate** (*boolean*) –
- **isUpdate** (*boolean*) –
- **isDelete** (*boolean*) –

Returns **Promise** – resolving when operations are complete

importObject(*className*, *obj*, *pacakgeRef*)
Import a plain json object (originating from a package or file) into db, with special handling of BusinessObject-Def's and versioning. (used by both fs_sync and pkg_stream)

Arguments

- **className** (*string*) –
- **obj** (*object*) –
- **pacakgeRef** – reference to BusinessObjectPackage on who's behalf this object is being imported; if not present, no version checking will occur

Returns **Promise.<BusinessObject>** – the created/updated object

packageToFs(*bopId*)
Exports a package's objects to filesystem, to begin filesystem sync and allow for collaboration/source control in git

Arguments

- **bopId** (*string*) – id of BusinessObjectPackage

Returns **Promise** – resolving when export is complete

Processing Package Streams

checkPackage (*metaObj*, *inProgress*)
Check the package metaObj against the current sytem.

How the noonian dependency summary is built: configured RemotePackageRepositories are queried to obtain metadata objects for all of the package's noonian dependencies. Then those objects are recursively checked, and the tree of results are flattened into a list, sorted in the order that they should be installed.

Arguments

- **metaObj** (*object*) – metadata object from the BusinessObjectPackage
- **inProgress** – used for tracking recursive calls

Returns **object** – summary describing the package - basic package info (key, name, desc, version) - installed version of that pkg (if applicable) - parameters requested by package to be collected from user on install - list of dependencies for npm, bower, and noonian - list of check results of noonian depenencies

getPackageMetadataFromStream (*pkgReadStream*)
Pull metadata element from Noonian package JSON stream; ignoring all other stream content

Arguments

- **pkgReadStream** (*stream.Readable*) – Noonian package JSON stream

Returns **Promise.<object>** – the metadata object

checkPackageStream (*pkgReadStream*)

Read package metadata from pkg stream and:

1) check its dependencies against what is installed
2) resolve to the user_parameters

Arguments

- **pkgReadStream** (*stream.Readable*) – Noonian package JSON stream

Returns **Promise.<object>** – the metadata object

installPackage (*pkgReadStream*, *userParams*, *skipDep*)
Install the package (and its dependencies) to this instance from json stream

Arguments

- **pkgReadStream** – node readable stream of package json
- **userParams** – object containing parameters to be passed to pacakges' install functions keyed by package key
- **skipDep** – skip dependency check/installation (mainly for use in recursive calls)

Returns **Promise.<{{result, metaObj, dependencyResults, recursiveResults, functionResults}}>** –

buildPackage (*bopId*, *majorMinorPatch*)

Run against a BusinessObjectPackage (BOP) record;

- builds the package file, incorporating all UpdateLog's associated w/ the BOP

- stores it in gridfs, sets as package_file attachment to BOP
- updates manifest and increments minor version on BOP record

Arguments

- **bopId** (*string*) – id of BusinessObjectPackage
- **majorMinorPatch** – which version component to increment

Returns Promise<string> gridfs file id of generated package

Internal Functions

Below are functions used internally for system initialization and package installation.

installBusinessObjectDef(*bodObj*)

Used internally for package installation. Creates or updates BOD in the database, and adds it to the Model cache. If the BOD indicates it is a child class whose parent model is not yet in the cache, it stores it to be installed on subsequent call for parent BOD

Todo: clean up return value to be more sensible

Arguments

- **bodObj** (*object*) – a plain-object representation of a BusinessObjectDef

Returns **Promise.<(null|true)>** – resolving to null when model is installed and initialization is complete or true if awaiting parent class installation

bootstrapDatabase(*adminPw*)

Used internally by CLI. Bootstrap a clean database

Arguments

- **adminPw** (*string*) – password for admin account. If null, password is set to a newly-generated UUID and printed to console.error

Returns **Promise.<User>** – business object for admin user

refreshModels()

Completely refresh data layer (called on package install). Delete all models in the cash, and reload from BusinessObjectDefs and MemberFunctions in the database

Returns **Promise** – resolves when reload is complete

db.**init**(*conf*)

Initialize mongo connection and the entire data layer.

Arguments

- **conf** (*object*) – instance configuration to use

Returns **Promise** – resolved upon completion

2.3.3 Configuration

Injectable as: **config**

Provides access to the Noonian configuration store, as well as the running instance configuration.

Todo: Move this to a CodeModule in the sys package; make instanceConf injectable by itself

instanceConf
A reference to the instance configuration (instance-config.js)

getValue (*key*, *defaultValue*)
Retrieve a value from the config store

Arguments

- **key** (*string*) – config key
- **defaultValue** – value to return if key is not present in store

saveValue (*key*, *value*, *userId*)
Save a value to the config store. If userId is not null, a user-specific config key will be saved.

Arguments

- **key** (*string*) – config key
- **value** – value to save
- **userId** (*string*) – ID of User account

2.3.4 Logger

Injectable as: **logger**

API for accessing the Noonian logger. See *Logging*

logger.**get** (*loggerName*)
Get a logger. If loggerName is specified, a child of the rootLogger is created with specified name. Otherwise, the system's default logger is returned.

Arguments

- **loggerName** (*string*) – name of logger to retreive

Returns **WinstonLogger** –

2.3.5 Invoker

Injectable as: **invoker**

Utility for invoking functions server-side with custom and standard injections.

getParameterNames (*fn*)
Extracts names of parameters from a js function

Arguments

- **fn** (*function*) –

Returns **Array.<string>** – parameter names

invokeInjected(*fnToInvoke*, *injectedParamMap*, *fnThis*)
invokes a function, injecting the arguments from injetedParamMap

Arguments

- **fnToInvoke** (*function*) – function to invoke
- **injectedParamMap** (*object*) – parameters to inject into invocation, keyed by name
- **fnThis** (*object*) – the **this** context provided on invocation

Returns the return value of the invoked function

invokeAndReturnPromise(*fnToInvoke*, *injectedParamMap*, *fnThis*)
invokes a function, injecting the arguments from injetedParamMap, wrapping return value of invocation in promise (if it isn't one already)

Arguments

- **fnToInvoke** (*function*) – function to invoke
- **injectedParamMap** (*object*) – parameters to inject into invocation, keyed by name
- **fnThis** (*object*) – the **this** context provided on invocation

Returns Promise resolving to return value of the invoked function

refreshCodeModules()
Refresh cache of CodeModules from database

Returns Promise resolving when refresh is complete

2.3.6 Business Object Metadata

All Business Object models and instances have a property **_bo_meta_data**, also accessible via the ES6 Symbol **metadata**. This property contains information about class name, field types, and inheritance.

BoMetaData()

An instance of BoMetaData is present on all Business Object models and instances keyed by _bo_meta_data, as well as ES6 Symbol metadata

Usually this is only instantiated internally

CHAPTER

THREE

INDICES AND TABLES

- *Change Log*
- genindex
- modindex
- search

3.1 Change Log

3.1.1 0.X.X | *release date*

Additions

- Item 1
- Item 2

Bug Fixes

- Item 1
- Item 2

3.2 GITLAB README

See noonian.readthedocs.io for Noonian documentation.

This directory contains the reStructuredText source for Noonian Documentation, and can be built using Sphinx.

This project is integrated with readthedocs.org, which automatically builds and deploys the documentation in this directory.

3.2.1 How to build

The following is required to build this documentation:

1. Python and sphinx must be installed
2. **pip** packages listed in doc/requirements.txt must be installed
3. The **npm** package **jsdoc** must be installed globally, or locally w/ the jsdoc CLI on the path.

This project's devDependencies includes **jsdoc**, so a full `npm install` will get it locally.

sphinx-build.sh adds jsdoc to the path and runs the sphinx makefile to build the HTML documentation.

INDEX

Symbols

_svc (*None attribute*), 30

A

aggregateCreateDacs() (*built-in function*), 29
aggregateDeleteDacs() (*built-in function*), 29
aggregateReadDacs() (*built-in function*), 29
aggregateUpdateDacs() (*built-in function*), 29
annotateIncomingRef() (*built-in function*), 35
applyNoonianContext() (*built-in function*), 34
augmentTypeDescMap() (*built-in function*), 31

B

BoMetaData() (*built-in function*), 41
bootstrapDatabase() (*built-in function*), 39
buildPackage() (*built-in function*), 38
Business Object, **18**

C

checkCondition() (*built-in function*), 30
checkPackage() (*built-in function*), 38
checkPackageStream() (*built-in function*), 38
checkReadDacs() (*built-in function*), 30
checkRoles() (*built-in function*), 29
checkRolesForUser() (*built-in function*), 29
cleanup() (*built-in function*), 36
CRUD, **19**

D

datatrigger.init() (*datatrigger method*), 32
db.init() (*db method*), 39
DBUI, **18**
deleteFile() (*built-in function*), 35

E

exportFile() (*built-in function*), 36

F

fieldtypes.init() (*fieldtypes method*), 31
Free Software, **19**

G

generateId() (*built-in function*), 30
getAllFileMetadata() (*built-in function*), 36
getCurrentUser() (*built-in function*), 28
getCurrentUserRoles() (*built-in function*), 28
getFieldTypeHandler() (*built-in function*), 31
getFile() (*built-in function*), 35
getPackageMetadataFromStream() (*built-in function*), 38
getParameterNames() (*built-in function*), 40
getQueryOpList() (*built-in function*), 34
getSchemaElem() (*built-in function*), 31
getValue() (*built-in function*), 40
gridfs.init() (*gridfs method*), 35

I

importFile() (*built-in function*), 36
importObject() (*built-in function*), 37
installBootstrapPackages() (*built-in function*), 37
installBusinessObjectDef() (*built-in function*), 39
installPackage() (*built-in function*), 38
instanceConf (*None attribute*), 40
invokeAndReturnPromise() (*built-in function*), 41
invokeInjected() (*built-in function*), 40

L

logger.get() (*logger method*), 40

M

MEAN, **19**
MongooseSchemaWrapper() (*class*), 31

P

packageToFs() (*built-in function*), 37
Perspective, **18**
processAfterCreate() (*built-in function*), 33
processAfterDelete() (*built-in function*), 33
processAfterUpdate() (*built-in function*), 33

processBeforeCreate() (*built-in function*), 32
processBeforeDelete() (*built-in function*), 33
processBeforeUpdate() (*built-in function*), 32
processFromDb() (*built-in function*), 31
processToDb() (*built-in function*), 31

Q

query.init() (*query method*), 34
queryToMongo() (*built-in function*), 34

R

RecordAction, **19**
references.init() (*references method*), 36
refreshCodeModules() (*built-in function*), 41
refreshDataTriggers() (*built-in function*), 32
refreshModels() (*built-in function*), 39
registerDataTrigger() (*built-in function*), 32
repair() (*built-in function*), 36

S

satisfiesCondition() (*built-in function*), 34
saveFile() (*built-in function*), 35
saveValue() (*built-in function*), 40
stringifyQuery() (*built-in function*), 34

U

updateLogger() (*built-in function*), 37
updateUserPassword() (*built-in function*), 28

W

writeFile() (*built-in function*), 35