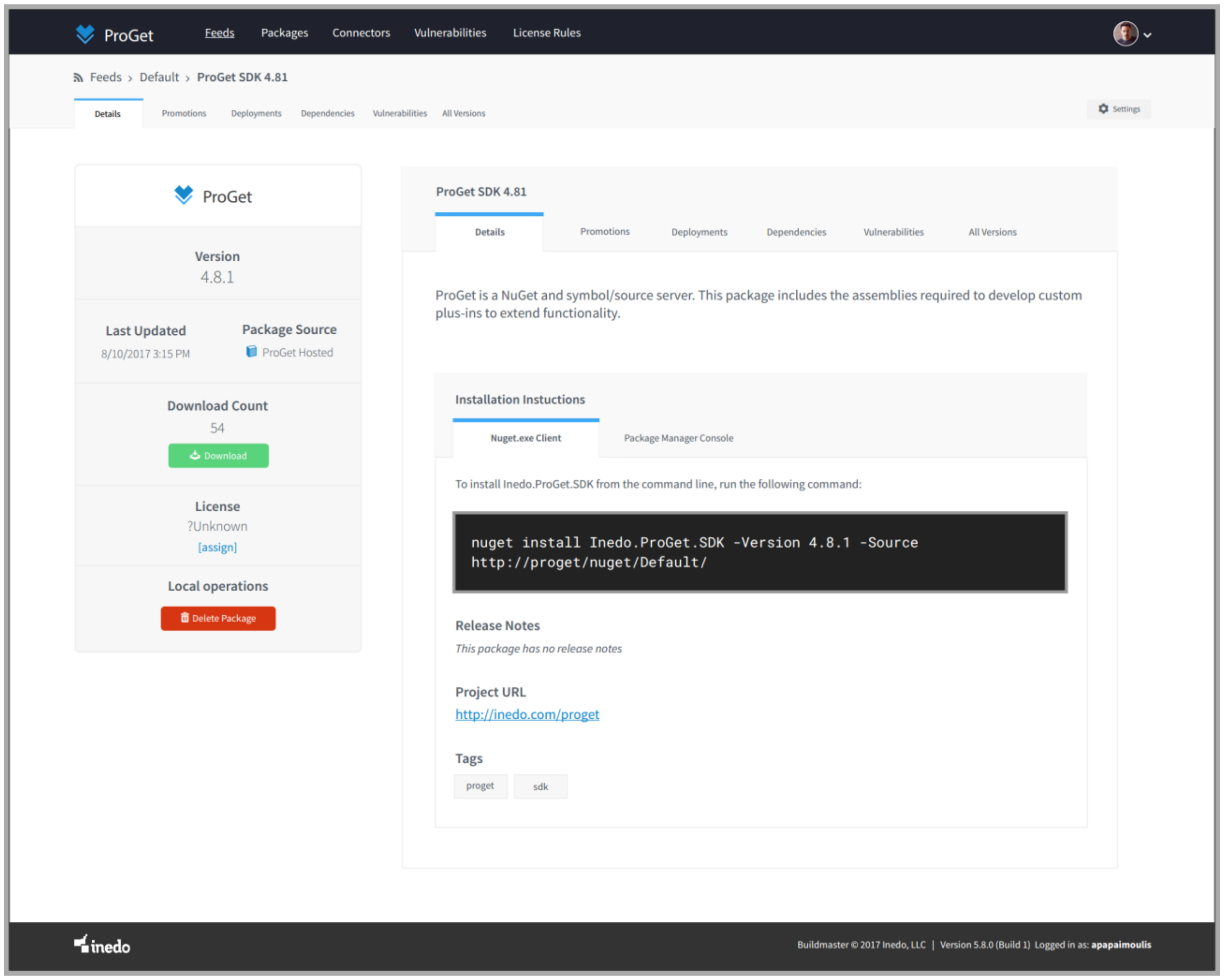
PROGET V5 PREVIEW

Although V5 is considered a “major” release of ProGet, most of the changes will be additive and have a minimal functional impact. This document details the planned changes for Version 5.0:

- New UI including a new logo
- Universal & Romp Package Improvements
- Webhooks (new feature)
- Inedo Execution Engine
- New NuGet Feed Type
- Inedo SDK & SDK Changes

User Interface

Certainly, the most notable change will be in the look and feel of ProGet. Although this is still in progress, we expect it to be derivative of this:



ProGet

Version
4.8.1

Last Updated
8/10/2017 3:15 PM

Package Source
ProGet Hosted

Download Count
54

Download

License
?Unknown
[assign]

Local operations
Delete Package

ProGet SDK 4.81

Details | Promotions | Deployments | Dependencies | Vulnerabilities | All Versions

ProGet is a NuGet and symbol/source server. This package includes the assemblies required to develop custom plus-ins to extend functionality.

Installation Instuctions

Nuget.exe Client | Package Manager Console

To install Inedo.ProGet.SDK from the command line, run the following command:

```
nuget install Inedo.ProGet.SDK -Version 4.8.1 -Source
http://proget/nuget/Default/
```

Release Notes

This package has no release notes

Project URL

http://inedo.com/proget

Tags

proget sdk

inedo

Buildmaster © 2017 Inedo, LLC | Version 5.8.0 (Build 1) Logged in as: **apapaimoulis**

New Top-Level Navigation

With the introduction of Assets in v4.8, the top navigation has gotten a bit crowded. We plan to change it to something like the following: Feeds, Packages, Assets, Containers, Compliance.

This navigation change will drive two other changes as well:

- **Docker Container Registries Split from Feeds**; Although Docker Registries will still be internally represented as feeds, they will be presented in the UI as an entirely separate concept, similar to Asset Directories.
- **Licensing Filtering and Vulnerability as Compliance**; these will still be separate features, but contained under the same compliance feature set. This will allow us to continue to add compliance-driven features under this same section

Universal & Romp Package Improvements

Universal Feeds and Packages were introduced in ProGet v4, and we're happy to see that many users have found then quite useful. With the coming launch of Romp and Hedgehog, we plan to expand Universal and Romp Package functionality within ProGet. These will primarily be UI improvements (such as a file browser and a romp-specific feed type) as well as API improvements; for example, the ability to download single files from packages.

ProGet Webhooks

ProGet Webhooks help you connect ProGet with other services like deployment tools, chatbots, ITSM platforms, and more. This is done by creating webhooks that will send HTTP-based notifications (payloads) when certain events like package publishing, deployment, and deletion occur. These payloads can be adapted to fit into nearly any other tool's API, including workflow automation services.

The draft documentation for this feature is included below.

Inedo Execution Engine Integration

Currently, ProGet uses a Scheduled Tasks feature to run background jobs through the service. These will be migrated to the Inedo Execution Engine, which is used in our other products, and provides a consistent mechanism for viewing progress and logs of short- and long-running background jobs.

New NuGet Feed Type

NuGet has a lot of documented and undocumented versioning quirks, which has made package identification quite challenging. For example, 1.0 and 1.0.0 are considered to be separate versions, which means you can have two packages with basically the same version number.

ProGet implements some of these quirks for NuGet feeds, but not all; for example, NuGet had also considered "1.0.0" and "1.00.0" to be separate versions, but in ProGet they are the same.

Fortunately, Microsoft has been working towards eliminating these quirks over the years, and introduced several breaking changes in various versions (including v3.4) and on NuGet.org, and is continuing to move the product towards a well-defined versioning scheme (SemVer2). Unfortunately, many organizations rely on the quirks – which is why we have been reluctant to introduce these changes in our product.

With v5, we will be introducing a new feed type that supports only SemVer2 version numbers, and will be renaming the old NuGet feed type to "NuGet (Quirks)". The legacy type will not be creatable from the user interface, but we will provide a manual migration mechanism.

Migrating NuGet (Quirks) Feeds

ProGet v5 will include the Inedo Execution Engine, and an execution can be dispatched from the administration page to migrate a NuGet (Quirks) feed. This execution will have a single option (Move Packages), which is only available on disk-based package stores.

The execution will do the following:

- Scan all packages; if any have a quirky version, an error will be logged and the execution will terminate with an error
- Create a new feed with temporary (GUID) name and disabled state
- Add new metadata (Database) entries to the new feed
- Move or Copy Packages to the new Package Store (depending on option of the execution)
- Rename the legacy feed to <FeedName>-legacy and disable it
- Activate new feed feed

Inedo SDK & SDK Changes

We will be switching to a new, unified SDK that will be used by all of our products. This will greatly simplify development by Inedo, as well as for third-party developers, and will allow a lot of code changes between applications. This will have a very minimal impact for ProGet, as very few users have created a custom extension.

ProGet Webhooks Feature Documentation

ProGet Webhooks help you connect ProGet with other services like deployment tools, chatbots, ITSM platforms, and more. This is done by creating webhooks that will send HTTP-based notifications (payloads) when certain events like package publishing, deployment, and deletion occur. These payloads can be adapted to fit into nearly any other tool's API, including workflow automation services.

Configuring Webhooks

Webhooks are configured at the feed level in ProGet on the Manage Feed page. Multiple hooks can be specified, but the order of invocation is not guaranteed.

Usage Note: Webhooks are intended for basic notification and triggering purposes only. Do not attempt to "chain" multiple webhooks to orchestrate a series of sequential events; it may just happen to work in testing, but the order of webhook invocation is not predictable and may occur in parallel.

A webhook may be associated with a feed, and has the following properties:

- Id; a system-generated unique identifier

- Name; a non-unique identifier used to distinguish webhooks in the UI and logs
- Event type; one of the event types
- URL; the URL to send the request to
- Conditional; an expression that is evaluated to see whether the webhook should run
- Custom payload; customize the request that is sent instead of the default payload
    - Method; one of GET, DELETE, POST, PUT, HEAD, OPTIONS, PATCH
    - Headers; a new-line separate list of HTTP headers
    - Body; the body of the request that will be sent

Events Types

Webhooks can be configured for the following events:

- added – a package is added to a feed using the API, web interface, or a drop-path
- promoted – a package is promoted using the API or web interface
- deployed – a deployment record is added; note that this occurs when certain HTTP headers are submitted during an API-based package download
- deleted – a package is deleted from a feed using the API or web interface
- purged – a package was deleted because of a retention policy

Conditional

The conditional is an optional expression that is evaluated prior to a webhook being invoked. This is takes the same format as an OtterScript predicate expression which has a standard unary operator (!),equality operators (== and !=), boolean comparison (&& and ||), and parenthesis .

Following are some example expressions and their results.

$FeedType == NuGet	If the feed type is NuGet, the webhook will execute; note that "nuget" will always evaluate to false; because it's a case sensitive comparison
true	The webhook will always execute
!true	The webhook will never execute
true && false	The webhook will never execute
$CustomVar == true	If $CustomVar is a defined variable, and it's "true", then the webhook will execute; otherwise, an error will occur if variable is not defined

Default Payload

By default, ProGet will POST a simple JSON-based object, with key/values containing package information and event type. Note that these use variables to send package-specific data.

Headers:

```
Content-Type=application/json
```

Content:

```
$ToJson(%(
    feed: $FeedName,
    package: $PackageId,
    version: $PackageVersion,
    hash: $PackageHash,
    packageType: $FeedType,
    event: $WebhookEvent,
    user: $UserName
))
```

Workflow Automation Services

When a basic HTTP request isn't enough to perform a particular task, you can use third-party workflow automation services to act as a sort of bridge between ProGet and the tools you want to automate.

There are lots of hosted and on-prem workflow automation services on the market – Zapier, IFTT, Microsoft Flow, elastic.io, and so on – and while they all function in a slightly different way, they'll all integrate with ProGet in just about the same way: ProGet will send an HTTP request to the automation service, and the automation service will trigger some workflow that may dispatch additional requests to other APIs.

Example: Zapier

In Zapier, automation workflows are called Zaps. When you create a new Zip with a webhook-enabled trigger, you'll receive a new URL that looks something like this:

```
https://zapier.com/hooks/catch/n/Lx2RH/
```

Zappier prides itself on making things "just work" when it comes to webhooks, and since ProGet's default payload is a simple JSON-based key/value pair, you can use that to get started.

After entering the Zap URL on a webhook for a feed, ProGet will start sending data to Zapier when the selected event occurs. Zappier will parse the fields (feed, group, name, version, etc) automatically.

From there, you can use those values for outbound integrations. See Zapier's Webhook Documentation for lots more details and troubleshooting.

Authentication to Other Services

While all APIs manage authentication a bit differently, you're usually provided with at least one of two options: using an API key, or basic access authentication. Both of these authentication methods can be done within a Webhook, though you might need to use a customize payload.

API keys are often sent on the querystring, message body, or in an HTTP header. You can add these wherever appropriate in a webhook.

Basic access authentication is actually just an HTTP header. You can include it in the headers section of the webhook, as a line like this:

```
Authorization: Basic $EncodeBasicAuth(username,password)
```

In both cases, it's a good practice to use variables to store keys, names, and passwords. Although this doesn't secure them (they are always sent as plain text over HTTP/S), it will often obscure them from causal viewing and make it easier to reuse in different webhooks.

Variables

You can use variables to combine static configuration information (such as API keys), context-specific data (like user name), and package metadata (such as version number) to create all sorts of custom payloads.

Configuration Variables

Configuration variables are static, key/value pairs that can be defined either globally or on a feed. If you define a feed-level variable with the same name as a global variable, then the feed-level variable will be used when events occur in the context of that feed.

Variable names must be no more than fifty characters: numbers (0-9), upper- and lower-case letters (a-Z), dashes (-), and underscores (_); must start with a letter, and may not start or end with a dash or underscore.

Variable Functions

You may have seen expressions like $JSEncode($PackageGroup) in the default payload; these are variable functions, which take in a number of parameters and return a value. All of the "built-in variables" like $FeedName, $PackageVersion, and $Date are implemented as variable functions.

You can create configuration variables with the same name of a variable function, but we don't recommend it because it can get quite confusing when things like $FeedName don't return the actual feed name. If there is a configuration variable of the same name, that value will be used instead, unless you explicitly reference the parameter list (such as $FeedName()).

Variable functions are extensible, so you can write your own with an Inedo Extension.

Reading Package Metadata

Variable functions are used to extract metadata from packages. There are several built-in functions that allow you to access common metadata across all packages: $PackageName, $PackageVersion, etc. You can also use the $PackageProperty() function.

For example, when $PackageProperty(_sourceTarget) is used on a universal package, the _sourceTarget property is returned. If there is no such property, then an error will be logged and the webhook execution will not occur.

However, if you specify the second argument (defaultText) to the PackageProperty function, then it will succeed. That is, $PackageProperty(_sourceTarget, unspecified) will return the value of *_sourceTarget* or *unspecified*.

Escaping $ and Unresolvable Variables

Anything that "looks" like a variable – i.e. text that starts with a $, then a character – will be parsed as a variable expression, and evaluated. If a configuration variable or variable function cannot be found, then an error will be logged and the request will not complete. This is usually what you'd want, and will help you track down a typo like $PackagName.

However, if you *actually* want a $-character somewhere in your payload, you'll need to escape it using a grave apostrophe (`) like this: `$. If you want to use a grave apostrophe, then you'll also need to escape it like this: ``. Whitespace character expansion is also available with `r, `n, and `t.

Built-in Variables

- $WebhookName – returns the name of the currently running webhook
- $WebhookId – returns the id of the currently running webhook
- $WebhookEvent – returns the name of the event
- $FeedId – returns the id of the feed in context
- $FeedName– returns the name of the feed in context
- $FeedType – returns the type of the feed in context; NuGet, Chocolatey, PowerShell, {…}
- $Date(format) – same description
- $PackageGroup – returns the package's group name, or empty
- $PackageName– returns the package's name or empty if it only has an Id
- $PackageId– returns the package's Id
- $PackageVersion – returns the package version
- $PackageHash – returns the package hash, if it had one computed already
- $PackageProperty(name, default) – returns an arbitrary property from the metadata; when default is specified, that text is returned in place of an error occurring when a property doesn't exist
- $Username – returns the name of the user who initiated the event; this will be SYSTEM if it was triggered by the service (a drop path, for example) or Anonymous
- $EncodeBasicAuth(username,password) – returns the base64 string used in basic auth requests
- $JSEncode(s) – returns a value that has been encoded for use in a javascript/JSON string
- $Coalesce(s, …) – returns the first non-whitespace value from the arguments

Error Handling and Debugging

ProGet will not wait for a response from the external server to continue. When an error does occur (either from bad HTTP status codes or network-level issues), it will be saved to ProGet's event log.

Note that, some errors -- such variables being unable to be replaced, or an invalid URL -- will cause a HTTP request to not be made.

Webhooks requests may be made from either the web or service, depending where the event occurred. For example, a package promote request will always come from the web node that actually received the request, whereas a package add request may come from the web node (web ui or API) or the service (drop path).

Webhook Management API Endpoints

Data specification

Webhook object

```
{
  "name": "EJH1000",
  "url": "https://hdarfs1000.zapier.com/hdar/asd",
  "event": "added",
  "payload": {
    "method": "POST",
    "headers: {"Content-Type": "secret", "HDirs": "123"},
    "content": "hello I am of the $PackageName"
  }
}
```

Webhook Listing

```
[
  { "id": 132, "feedId": 2, "name": "EHJ1000" }
]
```

List Webhooks Endpoint

```
GET /api/webhooks/list?feedId=<feedId>
 -> feedId is optional
 -> returns listing
```

Delete Webhook

```
POST or DELETE /api/webhooks/delete?id=<id>
```

Create Webhook

```
PUT or POST /api/webhooks/create
 -> webhook object
 -> returns ID as body of response
```

Edit Webhook

```
PUT or POST /api/webhooks/edit?id=<id>
 -> webhook object
```