HEDGEHOG (SUPER EARLY DRAFT) DOCS

Hedgehog helps you structure your organization's accumulation of software services into well-defined application and infrastructure packages that are easy to build, test, and audit. The web-based dashboard lets you deploy these packages to servers and environments using workflows that can accommodate anything from a fully-automated continuous deployment to a highly-regulated, compliance-driven process.

Packages

Hedgehog deploys packages to servers and environments. These packages can contain your application components, as well infrastructure configuration and deployment code.

Think of a package like a uniform-sized shipping box with uniform, machine- and human-readable labels describing the package. Inside the box are the things you want to deliver, and the box may even include assembly, installation, or delivery or other instructions on what to do with the contents.

Packages have become a unifying concept across a DevOps toolchain because they are built once and deployed consistently across environments. This means everyone can be certain that what goes to production is exactly what was built and tested.

Fortunately, there's not a whole lot to a package: it's a just a zip file containing the files you actually want to deploy, some code you may want to run, as well as some optional extra metadata. On top of the package file itself, a secure [cryptographic hash] is used to make sure that whichever package you first tested is *exactly* what gets deployed to each environment.

Creating Packages

There are a lot of options that developers and build servers alike can use to create and publish these packages:

- [Inedo ProGet Jenkins Plugin]
- [UPack.exe Commandline Tool]
- [Push to ProGet Visual Studio Extension]
- [Romp for Visual Studio Extension]
- Use a [drop folder] within ProGet
- Upload package files from the UI
- Simple HTTP Post with your own tool/script using [Universal Package API]

But you can also use Hedgehog's [advanced execution engine] to help you create packages by setting up a [release] and [pipeline] that imports builds artifacts from Jenkins, grabs files from a network drive, or even creates them directly using msbuild.

Romp Packages

When you want to also deploy infrastructure configuration, or embed your deployment code with your application code, this is where [Romp packages] come in. They are [universal packages], but with some additional metadata and scripts.

When you select a [romp package] to deploy, Hedgehog gives you the option to use the built-in installation and configuration options that the package creator bundled, such as:

- Installation and Configuration Scripts
- Required and Optional Variables
- Passwords and Secret Credentials

But unlike the [romp cli] tool, Hedgehog lets you target any number of servers to deploy your packages to.

Deploying Packages

Hedgehog offers three primary mechanisms for deploying a package: quick, package set, and release. While each of these methods ultimately will install your packages to a target set of servers, they progressively add more control and process for how these packages are deployed, and how they are visualized on dashboards.

This chart illustrates a few differences between the methods.

	Quick Deploy	Package Set Deploy	Release Deploy
Deployment Logs	✓	✓	✓
Scheduled deployments	✓	✓	✓
Any server, any environment, any time	✓	X	X
Predefined sequence of environments and targets	X	✓	✓
Approvals and gates	X	✓	✓
Deploy multiple packages	X	✓	✓
Project-level permissions	X	✓	✓
Templated variables	X	✓	✓
Notes and Calendar Planning	X	X	✓
Aggregate / multi-package Version Numbering	X	X	✓
Semantic version sequencing	X	X	✓

The quick deploy method offers the most flexibility, and is ideal for ad-hoc deployments and new applications. [Package set] and [release]-based deployments leverage [projects] to offer a whole range of additional settings, security, and help plan and visualize releases.

Quick Deploy

Hedgehog's Quick Deploy function is similar to running [romp install] in interactive mode, except you also select which servers to install the package on. Before using Quick Deploy, you will need to set up a [package source] and at least one [server] to deploy the package on.

1. **Package Selection**; after selecting package source from a list of configured sources, select the specific package and version to deploy from the autocomplete/search boxes
2. **Variables**; the selected package's metadata will be inspected, and the required and optional [package variables] will be displayed
   {{ Note: all values are treated as [OtterScript expressions]. This is mostly useful for entering [configuration variables] for package variables, and $, @, and % character literals should be escaped with `. }}
3. **Credentials**; the package's required credentials will be displayed, providing an ability to select credentials stored in Hedgehog
   {{ Note: we may support entering credentials directly in the future; for security purposes, you must enter them under Admin > Resource Credentials first }}
4. **Deployment Plan**; you will be prompted to select either the package's built-in `install.otter` plan or select a plan stored within Hedgehog

5. **Server targeting**; you can select either Servers by Name or Environments + Roles to target servers. Servers are deployed to in parallel.
   {{ note: when selecting Environments + Roles, the Roles act as an "or" while the Environments act as an "and"; that is, a server in any of the listed roles in all of the listed environments will be targeted }}.
6. **Schedule deployment;** you can select Execute Immediately or Schedule Deployment.
7. **Summary;** once all quick deploy options are configured, a summary of all options will be displayed.

Package Sources

Hedgehog was not designed to store packages, but instead to deliver/deploy them. After packages are created, they're published in a package repository like ProGet. Think of the repository like a warehouse that neatly stores, catalogs, and provides uniform access to all kinds of packages.

These repositories are referred to as "package sources" within Hedgehog, and consist of an endpoint URL (to either a specific feed or a feed manager) as well as an optional API key or username/password.

Thus, a "package" in Hedgehog is actually a reference to a specific package that's stored in an external package source. This way, when packages need to be deployed, each server can verify and then download the packages directly from the package source.

Of course, externalizing packages isn't an absolute requirement of using Hedgehog. But package management and package deployment are entirely separate problem spaces, and keeping them separate offers numerous advantages:

- **Security through buffering**; a package source acts as a security buffer between the groups that create the packages, and those that deploy them; package verification acts as an additional measure to ensure that the exact same packages are always deployed
- **Scalable package distribution**; a package source is primarily a url, which could point to anything from a basic universal feed to a load-balanced, globally-distributed ProGet instance.
- **Package-centric view**; a package source like ProGet organizes packages in a manner that is intuitive to engineers and other package creators; because Hedgehog adds [deployment records] to packages, it's easy to see exactly how packages were deployed

{{ Package Caching; the [universal package repository] already supports a package caching mechanism, and we may implement that in Hedgehog as needed. Please [let us know] if this would be of interest. }}

Package Sets

A package set is like an empty shipping pallet with a manifest form attached that's filled out in pencil.

They're used to to bundle a set of related packages – such as a PHP-based web frontend and a NodeJS-based API middle-tier – so that they can be delivered and deployed together. Not all of the packages in a set need to be deployed to the same targets at the same time – for example, your web frontend might go to different group of testing servers than your API middle-tier – but the entire package set is logically "delivered" to the same testing stage.

A package set also contains its own metadata and variables, which can used by the packages' deployment plans. These same deployment plans can also write variables back to the package set with the Set-PackagesetVariable operation, offering a convenient way to share metadata gathered during deployment.

Package sets can be simple, and reference just a single package from a [package source]. Or they can contain various packages from different packages sources. A package set can even start with zero packages, and have the first stage in the pipeline reference packages or create local packages programmatically.

Working with Package Sets

When shipping goods on pallets, it's generally best to add packages to your pallet, wrap it up, and then send it along its way. Of course, you can add or remove packages to the pallet en route, or change the manifest form at any time, but that makes things more complicated.

Package sets are similar. Generally speaking, you should set-up your package set early on in the process, but you have the flexibility to modify the package set as needed, either by editing metadata, or adding/removing packages, or variables as needed.

You can use a [package set template] to greatly simplify the process of complex package set creation.

Status and Lifecycle

A package set can be in one of three statuses that signify that where the package set is in its lifecycle:

- **Active**; progressing through its pipeline with the possibility of being deployed to the final stage
- **Deployed**; successfully deployed to the final stage of its pipeline
- **Rejected**; not deployed to its final stage, and instead determined to be inadequate or otherwise inappropriate for the final stage

When a package set is successfully deployed to the final stage of its pipeline, the status is automatically changed to Deployed. Users may reject package sets and, depending on pipeline configuration, package sets may automatically be rejected if another package set enters the same stage.

Administrators may also change the status of a package set at any time. This should only be done in exceptional situations, such as to append additional metadata to deployed packages, to "unreject" a package set, or correct other mistakes.

Variables

Package sets have a set of [configuration variables] associated with them that can be used by deployment plans at runtime; these can not only be used as input for required package installation variables, but for control flow within OtterScript, or anything else.

Properties and Metdata

Package sets have the following metadata properties; they may be accessed from within a deployment plan by using [variable functions], and can be changed from the UI or API

- **Id;** this is an immutable system-assigned identifier, and is used both internally and by the API to reference a specific package set; it cannot be changed and is unique across all package sets in an instance of Hedgehog
- **Name**; this is a human-friendly name used within the UI to help differentiate one package set from another; it is not unique across package sets
- **Template;** the name of the template used to create the package set; this is also used to determine deployment-time variables
- **Pipeline**; this references the pipeline that is used to deploy the package set through a sequence of stages
- **Project or Release**; a package set must be associated with either a [project] or [release]; you can change this on active package sets

You can use [raft item syntax] for Template and Pipeline names.

{{ Warning: changing a package set's pipeline after that package has been deployed to stages within another pipeline may yield to unpredictable behavior and inaccurate metadata. }}

Package Set Templates

Coming soon… like release templates.

Advanced Topic: Local Packages

There are some cases where you may want to "attach" files to a package set without creating and publishing an entire package to an external source. For example, one of the stages in your pipeline may generate a "release notes" HTML file and a later stage may email that file to customers after a release engineer verified the file looks acceptable. This is exactly a local package can come in.

Local packages are essentially zip files that are stored in a directory that Hedgehog manages, which can be on a local disk or a network share. Each zip file is associated with a package set, and these zip files can be uploaded or downloaded from the web user interface, as well as created and deployed using the [Create-LocalPackage] and [Deploy-LocalPackage] operations.

Unlike universal packages, local packages have no metadata associated with them, or even version numbers. They are merely a zip file that is attached to local package. In addition to the "release notes" example, you can do all sorts of things with them, such as importing build output into local packages directly from CI server, or even create an end-to-end continuous delivery workflow, where Hedgehog actually retrieves and compiles your source code.

Deployment Plans and OtterScript

Deployment plans are the instructions that tell you exactly what you should with a package. They could be more specific delivery instructions (put in the mailbox on the third floor), installation instructions (wire the lighting fixture into a 110v powerline after shutting off the power), or configuration instructions (once your speaker is plugged in, connect to an open WiFi network).

Hedgehog uses OtterScript and [Inedo's Execution Engine] to create these deployment plans. The OtterScript file can be stored within the package itself (see [Romp packages]), or you can store them in Hedgehog.

{{TODO – seems like there's lots of copy we can work from existing plans docs }}

http://inedo.com/support/documentation/buildmaster/core-concepts/deployment-plans

http://inedo.com/support/documentation/otter/core-concepts/plans

Pipelines

Pipeline are like the pre-determined route that package will follow to get to its destination; this route has various stops along the way (called stages) where packages are delivered/deployed, and then tested. After a successful deployment, the package set continues on its way to the next stop.

Essentially, pipelines will model the process your organization uses to deploy packages and applications to various servers and environments. They can be as simple or as complex as needed.

A basic website might use a pipeline with only two stages (testing and production) and deploy a package to a different folder on the same server. Another application may require a dozen stages, each with multiple targets that go to different environments, and all sorts of automatic and human approvals to meet compliance requirements.

Working with Pipelines

Not everything in a pipeline has to be automatic. A pipeline can be configured to require that a package set waits at a checkpoint until certain requirements are met: automatic verification, human approval, or if it's within a certain datetime window. Pipelines can also require that a human (or script) triggers that actual package deployment.

Properties and Metadata

Pipelines have three properties that help distinguish it: a Name, Description, and a Color.

{ something needed}

Guidelines and Best Practices

{much more needed}

Start Simple {TODO}

This level of flexibility can make it feel overwhelming to design the "right" pipeline.

Keep Pipelines Reusable {TODO}

TODO

Stages vs Targets vs Plans {TODO}

TODO

Using and Abusing Pre- and Post-Deployment Steps {TODO}

The Pipeline Model

Pipelines are used to model and automate a software delivery processes. As such, the pipeline "meta-model" has a lot of components and attributes, many of which are even more thoroughly documented in the [Pipelines as JSON] section.

You don't need to use all of the features at first, but it will be helpful to get a high-level of understanding of what the pipeline meta-model.

{{ diagram of a pipeline meta-model }}

Example: Simple Pipeline

Diagram showing [Int] > [Test] > [Prod] simple pipeline

Example: Complex Pipeline

Diagram showing [Int] > [Test] > [Prod] simple pipeline

Model Components

Pipeline

A pipeline consists of a sequence of [stages], variables, and several other properties and options.

{ copy information from JSON reference }Stage

A stage consists of a gate, variables, pre-deployment steps, targets, post-deployment steps, and several properties and options.

{ copy information from JSON reference }

Target Execution Mode

Talk about sequential vs parallel.

Gate

A gate is a mix of manual and automated approval requirements that a package set must have before being deployed to a stage, and are used to ensure the quality and acceptability of a package set as it passes through the pipeline.

{ copy information from JSON reference: User, Group, Automatic, Windows }

Target

A target is comprised of a deployment plan, an environment name, a list of servers or roles, and a list of packages to be deployed. All of these fields support variable expressions, which you can even set in the pre-deployment steps.

Deployment Plan

This is the actual OtterScript that will be run in order to deploy the package set. It can reference a plan stored within Hedgehog, or it can be the plan embedded in a Romp package. When referencing a plan stored within Hedgehog, you can use the [raft item syntax] to specify plans stored in other rafts and other projects.

Note that, depending on how server and package targeting is configured, it may be run multiple times across servers and packages.

Environment Targeting

Deployments can be targeted to an environment, which effect it in a few ways:

- **Permissions**; users must have permission to deploy to that environment in order to deploy to that stage
- **Role selection filtering**; if you also target by role, then the server must also be in that environment to be included
- **Runtime server protection**; a server must be in the target environment, or a runtime error will occur
- **Visualization**; this helps you see which packages have been deployed to which environment

Server Targeting

You can target servers either directly (by name) or by role (servers with that role). At deployment time, the [Pipeline Stage Deployment Execution] will build a list of servers using the server names or role names specified, and then run the specified deployment plan with each of those servers in context.

Note that when targeting servers by role names, only servers that are associated with the targeted environment will be used.

{{ infobox **Multiple Role Targeting Note**: When you target multiple role names, the same deployment plan will be executed once for each role selected, against all servers with that role and in the target environment. This means that, if you target two roles – and one server happens to have both of those roles – the deployment plan will be executed against that server at least twice. }}

Package Targeting

You can target server packages as well. At deployment time, the [Pipeline Stage Deployment Execution] will run the specified deployment plan with each targeted package in context. If the package set does not contain a package with that name, the plan will not be executed.

Note that this targeting does not apply to [local packages].

{{ infobox **Multiple Package Targeting Note**: When you target multiple package, the same deployment plan will be executed once for each package that's in the package set, against all targeted servers. This means that, if you target three packages that are in the set, the deployment plan will be executed against each server at least three times. }}

Pre- and Post-Deployment Steps

Before the pipeline stage's targets are evaluated, the [Pipeline Stage Deployment Execution] will run the pre-deployment steps associated with the stage.

Pre-deployment steps are executed in the sequence specified, and a failure in execution will cause the target evaluation to not be performed. Only the "run on fail" post-deployment steps will run when this happens.

Post-deployment steps work nearly the same way, except they are run after all of the targets have finished executing. Post-deployment steps will not if

Available Steps

The following steps may be selected as pre- and post-deployment steps.

- **Send Email**; this sends an email to the specified addresses with the specified priority, subject, and body
- **Set Stage Variable Value**; this creates or updates an ephemeral variable that is used throughout pipeline stage execution, and is made available to all plans as a configuration variable
- **Execute PowerShell Statement;** this runs the specified text using PowerShell
- **Execute OtterScript Script**; this runs the specified text using the OtterScript runtime

OtterScript and Steps

Each of available steps are stored as separate OtterScript scripts, and executed by the OtterScript runtime. They are simply displayed in a more user-friendly manner in the UI.

The [Send-Email] and [PSExec] operations are used to perform the send email and PowerShell steps. You can use a "pseudo operation" called `Set-PipelineStageVariableValue` to set a variable value on the stage in the same manner. This operation has two required properties (`Name` and `Value`).

Automatic and Human Approvals

As a package set progresses through a pipeline, both people and automatic processes and can add an "approval" to indicate that a package is eligible to enter to be deployed to a particular stage. The approvals required for that stage are defined in the [gate], and the actual approvals received become part of the package set's history and metadata.

Human Approvals

There are two types of human approvals: User and Group. A user-based can only be entered by the user specified in the gate, whereas a group-based approval can be entered by any members of the group specified in the requirement. A gate can also specify that more than one member of a particular group must approve a package set for a stage.

In either case, when a user approves a package set, the following is recorded:

- User's name
- Group name (if group-based)
- Stage name
- Approval description
- User comments
- Date/time

Users may revoke an approval they've given, provided that the package set wasn't already deployed to the target stage, and the package set is still active.

To determine if all human approval requirements have been met, approvals received is simply compared against the pipeline gate. This is nearly instantaneous, and can be performed at any time.

Automatic Approvals

There are two types of automatic approvals: variable test and OtterScript template.

{{ EXAMPLES of VARIABLES ./ SCRIPTS? }}

Because these automatic approvals can change over time (and are generally nearly instantaneous to run), they are checked whenever a user visits the package set page, and immediately before a non-forced deployment occurs.

Background Checking

Determining when automatic approval requirements have been met is a bit trickier. Because these require running arbitrary code (even the variable test, which may involve invoking a [variable function]), it must be performed by the Hedgehog service as a background task.

There are three service components that perform these checks:

- [Automatic Approval TaskRunner] - triggered by visiting the web UI
- [Automatic Approval Execution] - dispatched by the web UI for diagnostic purposes

- [Promotion Stage Execution] - run immediately prior to pre-deployment steps

Each of these components add or update an approval record on the package set:

- Approval description
- State (Met, Not Met, Not Required)
- Date/time

Once all of the checks have been run, the state (Met or Not Required) can be compared against the pipeline gate.

Pipelines as JSON

Unlike [Package Sets], [Releases], and [Projects], pipelines are not persisted as data in the Hedgehog database. Instead, they are JSON-formatted file stored in an abstract file system called a [raft]. You can edit pipelines directly in a raft, or in the web interface using the visual or text mode.

The pipeline's name is simply the name of the file (less the `.json` file extension), and the pipeline file contains a single [pipeline object]

Pipeline Object

This is the entire contents of a pipeline's json file.

`Color`	a *string* in the format #RRGGBB used to represent the color used in the UI
`Description`	a *string* to document the pipelines intended usage
`EnforceStageSequence`	a *boolean* to indicate whether the stage sequence should be enforced (requiring a Force to override), or if any stage can be deployed to at any time
`PostDeploymentOptions`	An *object* with the following property/value pairs. Each value is a *Boolean*. • `CancelReleases`; cancel earlier (lower-sequenced) releases that are still active and have not yet been deployed. • `CreateRelease`; creates a new release by incrementing the final part after a release has been deployed. • `DeployRelease`; mark the release and package as deployed once it reaches the final stage.
`Variables`	An *object* with property/values representing variable names and values ○ a variable name is a *string* of no more than fifty characters: numbers (0-9), upper- and lower-case letters (a-Z), dashes (-), spaces ( ), and underscores (_) and must start with a letter, and may not start or end with a dash, underscore, or space; a variable ○ a variable value is a *string* of any number of characters
`Stages`	An *array* of [stage] objects

Stage Object

This object is used Stages property of a Pipeline object.

`Name`	a *string* representing the name of the stage
`Description`	a *string* to document the stage's intended usage; this is not displayed anywhere in the ui
`AutoPromote`	a *boolean* indicating whether a deployment to the next stage will automatically occur if the target is next stage's gate is met and
`TargetExecutionMode`	a *string* with either `Parallel` or `Sequential`
`Gate`	a [Gate] *object*
`Variables`	an *object* with property/values representing variable names and values ○ a variable name is a *string* of no more than fifty characters: numbers (0-9), upper- and lower-case letters (a-Z), dashes (-), spaces ( ), and underscores (_) and must start with a letter, and may not start or end with a dash, underscore, or space; a variable ○ a variable value is a *string* of any number of characters
`PreDeploymentSteps`	an *array* of strings containing OtterScript scripts that are run prior to target evaluation
`Targets`	an *array* of [target] objects
`PostDeploymentSteps`	an *array* of objects or strings, representing steps that are run after target execution Each object has two properties: • `Script`; a *string* containing an OtterScript • `RunOnFail`; a *boolean* indicating if the step should run even if one of the targets failed A *string* in this array is an OtterScript that is run only when all targets succeeded

Gate Object

This object is used to describe the Gate property of a Stage object.

`UserApprovals`	an *array* of objects representing people that must approve deployment into the stage; each object has two properties: • `Name`; a *string* of the username of the approver • `Description`; a *string* of what will be displayed on the approvals required and received
`GroupApprovals`	an *array* of objects representing people in groups that must approve deployment into the stage; each object has two properties: • `Name`; a *string* of the group name that an approver belongs to • `Description`; a *string* of what will be displayed on the approvals required and received • `Count`; an *integer* of how many approvers in that group must approve the packageset before its met
`AutomaticApprovals`	an *array* of *strings* representing an OtterScript statement fragment: • Predicate expression, as used by an "if" statement (such as `$variable == value`)

	• Call Template Statement with two output variables:
	  ○ `$State`, which must return a value of `NotMet`, `NotApplicable`, or `Met`
	  ○ `$Description`, which must return an arbitrary string value describing the state

`DeploymentWindows`	an *array* of objects representing times in which a deployment may occur • `Days`; an array of *strings* representing the names of the days of the week (in the current culture) that a deployment StartTime occurs [Current Behavior] • `Days`; an array of *integers* representing the day of week numbers (with 0 being Sunday) that a deployment StartTime occurs [Current Behavior] • `StartTime`; a *string* in a ISO 8601 24-hour time format with an optional time zone offset: hh:mm[:ss.sss±hh:mm] • `EndTime`; a *string* in a ISO 8601 24-hour time format with an optional time zone offset: hh:mm[:ss.sss±hh:mm]

Target Object

Target objects are expected in the Targets array property on the Stage object.

`PlanName`	a *string* of the plan name that will be executed at the target, or null to use the plan embedded in the packages
`EnvironmentName`	a *string* of the name of the environment that is targeted
`RoleNames`	an *array* of *strings* representing the roles that are targeted
`ServerNames`	an *array* of *strings* representing the names of the servers targeted
`PackageNames`	an *array* of *strings* representing the names of packages to iterate over null indicates all packages an empty array indicates no packages

Releases

A release is an event where a planned set of changes are tested and delivered to production (or another final stage) using a set of packages. However, because packages are immutable by design, if a defect is found during testing process, a new package set must be built and tested in the same manner. This means any given release may require that multiple package sets are created and tested in order to find a suitable package set to deliver.

Hedgehog uses releases as a means of organizing and visualizing these related package sets. Releases also provide a set of configuration variables that all package sets can access, target dates that can assist with release planning, and notes that can facilitate communication.

Working with Releases

Think of release like a rough plan that guide your team in delivering a required set changes through testing and on to their final destination. You'll probably have a desired production delivery date, as well as other key dates like when internal and client testers will schedule time to review the changes. You also may know in advance which pipeline and template package sets in the release should use.

While releases primarily serve as a visualization, they also can help define and facilitate better self-service processes. For example, a team lead may create multiple releases at once – one planned for next week, another for next month, and an emergency hotfix – and different engineers will be able to deliver different changes of the same application to different testers, all without stepping on each other's toes.

Status and Lifecycle

A release can be in one of three statuses that reflect the state of development;

- **Active**; packages sets are being created and deployed through pipelines
- **Deployed**; a package set was deployed to the final stage of its pipeline
- **Cancelled**; no package set was deployed to the final stage, and package sets are no longer being created or deployed; all package sets are also rejected

When a package set is successfully deployed to the final stage of its pipeline, the release status is automatically changed to Deployed. This is controlled by the pipeline, and be configured to behave differently.

Administrators may also change the status of a release set at any time. This should only be done in exceptional situations, such as to append additional metadata to a release, to "unreject" a package set, or correct other mistakes.

Variables

Releases have a set of [configuration variables] associated with them that can be used by deployment plans at runtime; these can not only be used as input for required package installation variables, but for control flow within OtterScript, or anything else.

Properties and Metdata

Releases have the following metadata properties; they may be accessed from within a deployment plan by using [variable functions], and can be changed from the UI or API

- **Id;** this is an immutable system-assigned identifier, and is used both internally and by the API to reference a release; it cannot be changed and is unique across all releases in an instance of Hedgehog
- **Number**; this is a human-friendly name in a “SemVer2” format; it is unique across releases in a project
- **Name**; this is a human-friendly name used within the UI to help differentiate one release from another; it is not unique across releases
- **Template;** the name of the [package set template] used to create package sets; this is also used to determine deployment-time variables
- **Project**; a release must be associated with a [project]; you can change this on active releases

You can use [raft item syntax] for Template names.

Projects

Project are used to organize and visualize your package sets, pipelines, releases, templates, and plans. You can also use them for security access control by granting and denying tasks to different users across projects. They can also be nested, which means you can model complex applications or organizational units as needed.

Working with Projects & Sub-projects

Think of a project like a file system folder/directory, and the project's contents (pipelines, releases, templates, etc.) like the files within that folder. Just like folders, projects can be nested and their contents can reference other projects' contents, such as having a pipeline in one project reference a plan in another project.

Variables

[Configuration variables] may be scoped to a project, and can be seen and created on the project's settings page. These variables can be used by anything within the project: pipelines, plan, and even sub-projects.

Properties and Metdata

Projects sets have the following metadata properties; they may be accessed from within a deployment plan by using [variable functions].

- **ProjectId;** this is an immutable system-assigned identifier, and is used both internally and by the API to reference a specific project; it cannot be changed and is unique across all projects in an instance of Hedgehog
- **Name**; this is a human-friendly name used within the UI to help differentiate one project from another; it is unique to its parent project (or the global scope, if no parent project)
- **Description**; a markdown-formatted field that can contain any user-defined text; it is displayed on the project's home page
- **Parent ProjectId**; an optional reference to a parent project; this must be set to another project that that would not cause a circular reference
- **Raft name**; the [raft] that the project will use for its contents; this an advanced option, and won't be available to change until you've created multiple rafts
- **Active/Inactive**; you can deactivate a project, which will mostly hide it from the user interface and in most API calls. Unless you explicitly purge a project, a project will be deactivated

Project Content Name Resolution

Most content items reference things by name. For example, a pipeline named "Hotfix" might reference a plan named "Emergency Deploy" in a stage named "Production". While you can the [fully-qualified syntax](#->raft asset resolution), it's best to let Hedgehog resolve simple names for you, especially when content items are contained within the same project.

When searching for content with an unqualified name, Hedgehog will first look in current project for content with that name and type. If no content is found, then the parent project is searched, and then the ancestors, all the way to the global (i.e. no project) level.

Example: Sharing Resources with a Hierarchy

One common use case for project nesting is to use a parent project as a sort of "container" for both other projects and the content that those projects will use. For example, consider a set of libraries that are all published in a consistent manner, using the same template ("Library"), which references a pipeline ("Standard") and a deployment plan ("Publish Library"):

```
/projects
  /CommonLibraries
    /templates
      Library.json
    /plans
      Publish-Library.otter
    /pipelines
       Standard.json
    /projects
      /Library1
      /Library2
      /Library3
```

With this layout, new library projects can easily be added by simply creating a project under the "CommonLibraries" project. All package sets in these libraries can use the same pipeline and plan. So long as the same "Library" template is used, everything will be built and deployed consistently. Of course, the packages or projects themselves can variables of to control how and where they are published.

Anti-example: Confusing Hierarchy

This following layout is pretty bad, and you should never set anything up like this, because it's quite confusing. But it demonstrates how the name resolution works.

```
/projects
 /ParentProject
  /plans

   Emergency-Deploy.otter
  /pipelines

       Hotfix.json
    /projects
      /ChildProject1
        /plans
          Emergency-Deploy.otter
      /ChildProject2

     /pipelines

      Emergency.json
```

When a package set in ChildProject1 specifies “Hotfix” as a pipeline, then ParentProject’s Hotfix.json will be used. If ParentProject’s HotFix specifies “Emergency Deploy” as a plan, then that package set will use ChildProject1’s Emergency-Deploy.json, and *not* the ParentProject’s version.

When a package set in ChildProject2 specifies “Emergency” as pipeline, then ChildProject2’s Emergency.json will be used. If this specifies “Emergency Deploy” as a plan, then the package set will use ParentProject’s Emergency-Deploy.otter.

Best Practices

Like folders, projects require a unique name within their containing project. But aside from that, there are no restrictions as to what kinds of contents a project can contain. This means you ought to be considerate when created your project structure, and specially with nested, sub-projects. You don’t want to end up with a project hierarchy that mimics that nightmare share drive with a mess of random files and folders scattered throughout.

Avoid Simple, Context-free Names

From a UI perspective, it’s not feasible to always display a project’s full path. Operating systems have the same challenges with folder hierarchies. As such, you should avoid create a project hierarchy that relies on short project names that rely entirely on the parent projects to understand context. For example, the name “Rel” is meaningless without the hierarchy like “HDARS > Current > Rel”. Instead, HdarsRel might be better.

Avoid Relying on Cascading Behavior

As the [anti-example] demonstrated, content name resolution allows for a “cascading” behavior of sorts. You can have a child project “override” a pipeline or plan in the parent project simply by creating an item with the same name.

This is a generally bad idea, because it will be hard for other users – and yourself, down the line – to distinguish when something is “overridden” or when it’s following a “standard”. This isn’t something we will try to “solve” in the UI, because this feature simply isn’t intended to be used for this purpose.

If you need to make an exception to an “inherited” behavior, then you should use a different plan name. For example, if one of your dozen libraries behaves differently than all the others, then you should use differently-named templates, plans, or pipelines. Or use variables to control the behavior. Just don’t hide it in a “cascaded” item.

Be Careful When Renaming and Modifying Hierarchy

Because content can be cross-referenced across projects, you should be careful when changing a project’s name or parent project. If any content is using a [fully-qualified syntax](#->raft asset resolution) to reference an item within a project you rename or “move”, then that resolution may fail.

Even though Hedgehog will detect and fix references when you move or rename a project, there are many cases when it’s simply not possible. For example, if you use a variable to determine the name of a plan to use in a pipeline, there’s no way for Hedgehog to know how that variable that gets set, or that it might get set to the “old” name of a content item.

Executions

[[ADVANCED TOPIC WARNING --- this documentation is technical, perhaps too technical for casual reading; it primarily was written to provide a very detailed specification for how it works behind the scenes]]

An execution represents a deployment, orchestration, routine task, synchronization, or any other type of job that is run by the service using the [Inedo Execution Engine].

All executions records are stored in the database, have [scoped logs], and the following properties:

- Execution ID
- Start/End Dates
- Run State: Pending, Executing, Completed
- Execution Status: Normal, Warn, Error
- Execution Type

Different types of executions will have additional properties as well.

Dispatching and Running Executions

The service uses a the [ExecutionDispatcher TaskRunner] to query the database for executions with a *Run State* of "Pending" a *Start Date* that is before or equal to the current datetime. For each suitable execution found, a new background task is used to run the execution.

The [Service] administration page will display the currently running background tasks that were created by the ExecutionDispatcher, and provide links to the appropriate [execution in progress] page.

When the execution is complete, the background task will terminate; you can view all executions (regardless of state) using the Executions administration page, although it will probably be easier to find the specific execution record using a more specific context (such as package deployment history).

Troubleshooting: "Stuck" or "Frozen" Executions

An execution's lifecycle is generally short, but can range from just a few seconds to several hours. Executions cannot be canceled per se, but some executions (namely, [deployment executions]) support cancellation requests.

Like all computer programs, it's certainly possible for an execution to "freeze up" (i.e. remain indefinitely in an Executing state while not logging any new information). Because execution lifecycles vary so much, there's no definitive rule for determining when an execution is "frozen", and thus there is no way to programmatically detect it. You'll simply have to use your judgement.

When an execution becomes "stuck", it's most certainly one of several error conditions:

- Invalid state with the EventDispatcher
- Execution engine halted without reporting
- Code that the execution engine is running
- Third-party code never returning or timing out properly

The only safe way to clear a "stuck" execution is to stop and then start the service. This will obviously also halt all other running executions, so be careful when doing that.

When the service starts up again, all executions in a “Executing” state will be changed to “Completed”, and marked as “Failed”.

If the same type of execution continues to get stuck in the same place, try to gather as much information as you can to help us isolate the problem and [submit a ticket] with details you’ve gathered.

Viewing Execution Status and Logs

Aside from using the [Executions API], there are two pages you can use to see the status of executions.

Execution in Progress Page

This is a “live” view of the execution, and displays a *very rough approximation* of the progress (it’s about as accurate as any installer’s progress bar that you’ve seen), as well as the currently executing steps.

The logs are displayed “as they come”, which can get a bit confusing when multiple things are happening in parallel.

Changing Pending Executions

If you have the appropriate permission, and the execution is in a Pending state, you can change the Start Date, set the Start Date when it’s null, or delete the execution.

Execution Details Page

This is a static view of the execution and displays the logs in a scoped view. For completed executions, this is by far the most useful view, and you will often be redirected to this page from the Execution In Progress page once the execution completes.

Changing Execution Status

If you have the appropriate permission, you can change the Execution Status of an execution in a Completed state. There is usually no good reason to do this, aside from having a part of the UI be green instead of red.

Execution Types

{ several types of executions?? }

Pipeline Stage Execution

This execution is initiated through [Pipeline-only] and [Release] deployment workflows, and deploys a specified Package Set to the specified Stage in its pipeline. It will also dispatch additional execution [Pipeline Targeted Deployment Executions].

A pipeline stage is comprised of several elements, including a pre-deployment steps, deployment targets, and post-deployment steps. Because pre- and post-deployment steps are effectively OtterScript operations (with the exception of Execute OtterScript Plan, which is a plan), a set of pre- or post-deployment steps are converted into a single OtterScript plan. This OtterScript plan runs in the localhost server context by default, but other servers may be target using the “for server …” OtterScript notation, as long as those servers are not restricted by environment.

After validating that the package set can be deployed to the specified stage and the pre- and post-deployment plans are valid, the pre-deployment plan is run. Each target is then evaluated, and a

[Pipeline Targeted Execution] is dispatched for that target. Once all targets have completed, the post-deployment plan is run.

Deployment Execution Types

A deployment execution runs a specified deployment plan against an optional Server Targeting while providing additional context (such as runtime variables and information about the project, release, package, etc).

There are two types of deployment executions: Package Deployment and Pipeline Deployment. Both types use Server Targeting.

Package Deployment Execution

This is a Deployment Execution that is initiated through the [Quick Deploy] workflow. A Server Target is always specified, and the deployment plan is either stored within Hedgehog or the package (`install.otter`).

Once loaded and compiled, the actual plan is wrapped in a Set Context Statement (with a ContextType of `package`, and ContextValue of the package name). The plan is then wrapped using the Server Targeting described below.

Pipeline Targeted Execution

A pipeline targeted execution is dispatched by a [Pipeline Stage Execution], and provides

Each package in the set is wrapped in a Set Context Statement (with a ContextType of `package`, and ContextValue of the package name), and that plan is then wrapped using the Server Targeting described below.

Server Targeting

If a Server Targeting is also specified, then the plan will be wrapped as follows before execution.

Direct Targeting

A single Context Iteration Statement will be created with the Source set to a literal expression of the server names (e.g. `@(Server1, Server2, Server3)`). The Body contain an Execution Directive Statement with an Asynchronous flag, and the Body of that will contain the actual plan.

Roles + Environment Targeting

For every role targeted, a Set Context Statement (with a ContextType of `role`, and ContextValue of the role name) will be created. The Body of those statements will comprised of a single Context Iteration Statement with the Source set to a literal expression of the servers in that role and environment. The Body contain an Execution Directive Statement with an Asynchronous flag, and the Body of that will contain the actual plan.

If no servers were in any of the role iterations, then a Log-Warning statement will be appended to the wrapped plan, as the actual plan will not execute.

Pre-execution Failure

This execution invokes the [OtterScript runtime] to execute either the specified plan directly, or a wrapped version of the plan.

If an error condition occurs before the runtime is invoked, such as a complication error or a pipeline resolution error, the error message will be written to logs, the Execution Status will be set to Error, and the Run State will be set to Completed.

Automatic Approval Gate Execution {TODO}

This execution is invoked by the

Infrastructure Import Execution {TODO}

Issue Import Execution {TODO}

If the package's `installer.otter` file is used, then it is loaded from the package

-based OtterScript is not specified, then the install.otter file within the package will be used

the package will be inspected

It ru

Deploying a Release

Releases

In notepad, enter the following.

```
{
    "WebsiteRoot": "C:\\Website\\Accounts"
}
```

Romp uses the standardized [local package registry specification], which allows romp and other tools to see which packages are installed on the machine. By default, a Machine-level registry is used, which is stored in `%ProgramData%\upack`.

Rafts

{{ Rafts are an advanced feature. If you are just getting started with Hedgehog, you should revisit this topic once you've familiarized yourself with the other components. }}

Plans, assets, pipelines, and templates are essentially just files, and to make sharing, versioning, branching, etc. possible, Hedgehog uses a **raft** to store these files. A raft is sort of abstracted file system, specifically designed for storing these types of items.

The Default Raft

Because you likely won't need multiple rafts right away, a raft named "Default" is automatically created when you install Hedgehog. This is a [Database] raft and is backed up when you do a regular [Back-up of Hedgehog].

If you only have a single raft configured, and that raft is named "Default", then the ability to filter or select rafts will not be exposed in on plan, project, asset, etc. pages.

Creating, Managing, and Downloading Rafts

You can create, manage, and download a raft as a zip file by going to Admin > Global Components > Rafts.

Rafts and Projects

When you specify a raft for a project, Hedgehog will always search within the associated raft for content, using the [Project Content Name Resolution] search. Only the associated raft will be searched, which means that if a parent project uses a different raft, content may never be located. If no raft is specified, then the "Default" raft (if one is named that) is used.

Raft Repository Types

Rafts rely on an [Extensible Raft Provider] to retrieve and store raft data. There are three built-in raft types:

- **Database** - this is the default raft provider, and persists all information in Hedgehog's database; while the contents are not versioned, it's the simplest to use and back-up as part of the Hedgehog database
- **Disk** - the raft is persisted as a simple file structure on disk; this may be appropriate for quick testing purposes
- **Git** - this raft is synchronized with a branch of a remote Git repository. See the [Storing Hedgehog Content and Configurations in Git] tutorial to learn how to use this type of raft.

Because rafts are extensible, you can build them yourself. See the [Inedo SDK Documentation] to learn more.

Disk Layout Conventions

When stored on disk, in source control, or exported as a zip file, rafts are laid-out as follows.

Deployment Plans	Files (.otter) within the /plans subdirectory
Variables	An OtterScript stored in the /variables file, consisting entirely of assign variable statements

Scripts	Files (.ps1, .js, .sh) within the /scripts subdirectory
Modules	Files (.otter) within the /lib subdirectory
Assets	Files within the /files subdirectory
Pipelines	Files (.json) within the /pipelines subdirectory
Package Set Templates	Files (.json) within the /package-set-templates subdirectory
Project-specific Content	A /projects subdirectory containing directories named by the project, and a layout identical to a raft

Project-specific Content

Project-specific content is stored in directory named after the project in a /projects subdirectory. While this may make for some long paths, it allows for projects to be named anything. For example, the "default" pipeline in the BuildMaster > SDK project would be here: /projects/BuildMaster/projects/SDK/pipelines/default.json

Raft Variables

Variables persisted within a raft are not currently displayed anywhere in the UI, and are intended to be used for storing default or fallback values for plans stored in a portable/reusable rafts. They are the lowest scope, and are only used if a [Configuration Variable] of the same name does not exist

Raft Asset Resolution

Hedgehog can automatically resolve names using the [Project Content Name Resolution] system, sometimes it's convenient to specific an explicit, fully-qualified path. You can do this with a combination of raft names and Project Paths.

- **AssetName**; this uses project content name resolution, and searches the folder of the current project (if exists), and then up the projects chain to the root
- **RaftName:: AssetName**; searches the root (only) of the specified raft
- **RaftName::ProjectPath::AssetName**; searches the project path (e.g. /buildmaster/sdk) only

Service { TODO }

Talk about task runners, and specifically what they do/will do.

- Agent Updater Task Runner (rename?)
  - Checks status of agent periodically
  - Updates to latest client agent
- Execution Dispatcher Task Runner
  - This is the old PlanActionExecutor
- Update Checker Task Runner
  - Checks product, extensions
- ~~Retention Policy Task Runner~~
- ~~IssueTrackerSync Task Runner~~
- Infrastructure Sync Task Runner
- AutomaticApproval Task Runner

Configuration Variables { TODO }

Each variable is comprised of the following:

- **Name**; this is standard variable name
- **Type**;
- **Value**;
- **Sensitive**; a flag that indicates the value is sensitive, and should be obscured from casual viewing in the UI
- **Evaluate**; a flag indicating whether the

Migrating from BuildMaster

Hedgehog was created forking BuildMaster 5.8, uninclusing the code for legacy features, tweaking the model a bit, and then renaming/refactoring things in the code. This makes migrating quite easy.

Migration Strategy

{{ three things to migrate }}

- **Infrastructure**; the servers, roles, and environments
- **Global components**; resource credentials and global assets (plans, pipelines, etc)
- **Projects**;

{{ project-by-project is a good way to start }}

Infrastructure Migration

BuildMaster, Otter, and Hedgehog all share the same infrastructure model, and have the same [infrastructure sync] feature that allows a consistent model across the tools. Because they all use the same [Inedo Agent], configuring another Inedo product to communicate with those servers is quite easy.

At first, we recommend manually importing your infrastructure from BuildMaster. You do this by export a JSON document describing this infrastructure from BuildMaster, and then import it into Hedgehog. Down the line, you can set up an infrastructure sync.

Using the Migration Tool

The migration tool has three modes:

- Global components
- All project
- Single project

There is no overwriting; only addition?

{{ The migration tool will not validate your usage of [BuildMaster Legacy Features]. For example, if you have a pipeline that is relying on a legacy plan, it's simply going to fail when you import the plan. Make sure to verify the Legacy Features Dashboard in BuildMaster to see what features you are using. }}

Database Permissions

The migration tool data directly from the BuildMaster Database and writes it in the Hedgehog Database, so at a minimum, you will need read permission from BuildMaster's database tables, and read/write permissions for Hedgehog's database.

Hedgehog vs BuildMaster Concepts

Because the product

BuildMaster v5	Hedgehog v1	
Application	Project	
Deployable	-	These are replaced with variables and, to some extent, packages

Application Group	-	Projects can have a parent project, which makes application groups unnecessary
Artifact	Local Package	
Release Package (Build*)	Package Set	
Promotion	-	These are no longer used; but a [Pipeline Stage Deployment Execution] effectively takes the place

* denotes v4/v3 terminology