

Walkthrough: Set up Continuous Integration And Release Management

In this tutorial, you'll set up database lifecycle management using Redgate tools, including the DLM Automation TeamCity plugin:



Version control



Continuous integration



Release management

In this stage, you'll use SQL Source Control, Subversion (SVN) and TeamCity to:

- [create the development database](#)
- [link the database to version control](#)
- [create a TeamCity project](#)
- [link the database to TeamCity](#)

When you've finished, TeamCity will monitor every change you check into version control.

In this stage, you'll use TeamCity and the DLM Automation TeamCity plugin build runners to:

- [build a database package](#)
- [test the package](#)
- [sync the package to a CI environment](#)
- [publish the package to a NuGet feed](#)

When you've finished, TeamCity will automatically trigger a continuous integration process every time a change is checked into version control.

In this stage, you'll use Octopus Deploy and DLM Automation step templates to:

- [create an Octopus Deploy project](#)
- [add a download and extract the package step](#)

- add [create](#), [review](#) and [deploy](#) steps
- [create a release](#)

When you've finished, you'll have set up an automated release process to deploy your database to production.

Before you start



We've described how to install tools on three different servers - development, build and deployment - which is a common configuration in the real world. If you can't do this, it's fine to install everything on one machine.

On your development server:

- Install SQL Server Management Studio (SSMS) version 2008 R2 or later.
If you want to run tSQLt tests against the database, you'll also need access to a SQL Server 2012 (or later) instance with common language runtime (CLR) integration enabled. For more information, see [Enabling CLR integration](#) (MSDN article).
- Install [TortoiseSVN](#).
- Install [Redgate SQL Source Control](#).
- Download the [WidgetDevelopmentDatabaseCIDemo.zip](#) file and extract the contents.

On your build server:

- Install [JetBrains TeamCity](#) version 7 or later. We've tested this example against version 9.1.1. This automatically installs a default build agent.
- From the [Redgate DLM Automation add-ons page](#), download the TeamCity plugin and copy to the TeamCity data directory plugin folder. See [Set up the TeamCity plugin](#).

On your deployment server:

- Install DLM Automation. See [Installing](#).
- Install [Octopus Deploy](#) and create a new environment called *Production*. See [Install Octopus Deploy and create a new environment](#).
- Install an Octopus Tentacle and assign the **db-server** role. See [Install an Octopus Tentacle](#).



Version control

In this section you'll:

- [create the development database](#) and use SQL Source Control to [link it to version control](#)
- [create a TeamCity project](#) and [link the development database to it](#)

Create the development database

Create the *WidgetDevelopment* database using the SQL script you extracted from the [WidgetDevelopmentDatabaseCIDemo.zip](#) file:

1. Open SQL Server Management Studio (SSMS).
2. From the **File** menu, select **Open > File**.
3. Browse to the *WidgetDevelopment.sql* file and click **Open**.
4. Click **Execute** to create the *WidgetDevelopment* database.



Don't worry about the warnings on missing objects. We'll create these once we configure the test step.

Link the database to version control

Use SQL Source Control to link the *WidgetDevelopment* database to a shared Subversion (SVN) repository. You'll use the dedicated developer model, where each developer in your team will work on their own local copy of the database.

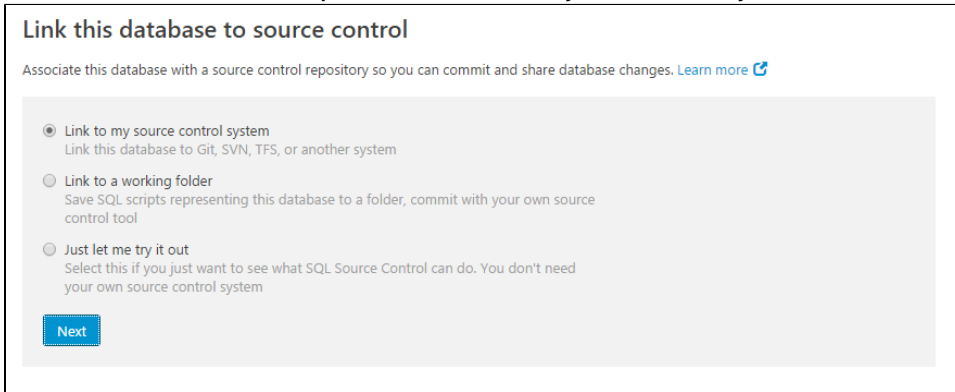
Use TortoiseSVN to create a Subversion repository. This must be a network folder that others in your team can access.

1. Using Windows Explorer, create an empty folder for the repository, for example, *Z:/WidgetShop*
2. Right-click the folder, and select **TortoiseSVN > Create repository here**.

Link the *WidgetDevelopment* database to Subversion using SQL Source Control:

1. In SSMS Object Explorer, select the *WidgetDevelopment* database you want to link to source control.

2. In SQL Source Control, on the **Setup** tab, make sure **Link to my source control system** is selected and click **Next**:



Link this database to source control

Associate this database with a source control repository so you can commit and share database changes. [Learn more](#)

- ☒ **Link to my source control system**
Link this database to Git, SVN, TFS, or another system
- ☐ **Link to a working folder**
Save SQL scripts representing this database to a folder, commit with your own source control tool
- ☐ **Just let me try it out**
Select this if you just want to see what SQL Source Control can do. You don't need your own source control system

Next

Select **Subversion (SVN)** and click **Next**:

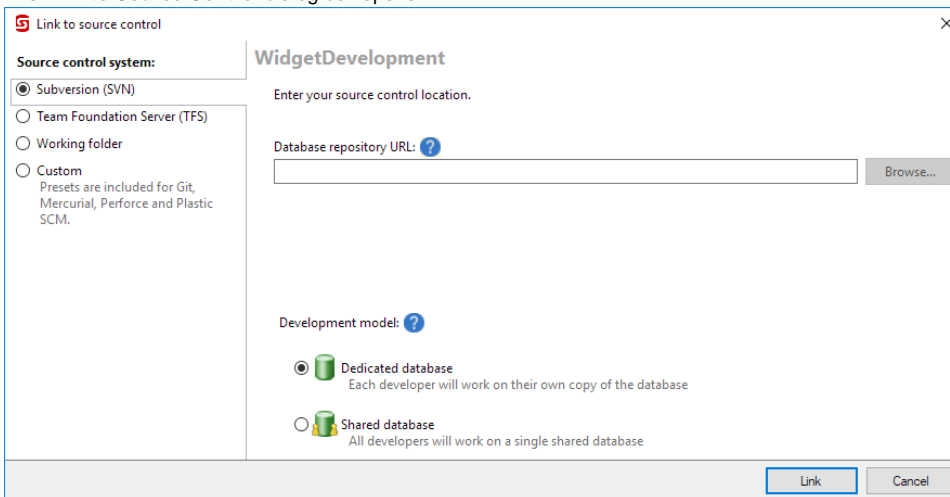


Choose your source control system

- ☐ **Git**
- ☒ **Subversion (SVN)**
- ☐ **Team Foundation Server (TFS)**
- ☐ **Other**
Link to any source control system with a command line, using a config file. Example files are provided for Mercurial, Perforce and Plastic SCM

Back **Next**

The **Link to Source Control** dialog box opens:



Link to source control

Source control system:

- ☒ **Subversion (SVN)**
- ☐ **Team Foundation Server (TFS)**
- ☐ **Working folder**
- ☐ **Custom**
Presets are included for Git, Mercurial, Perforce and Plastic SCM.

WidgetDevelopment

Enter your source control location.

Database repository URL: **Browse...**

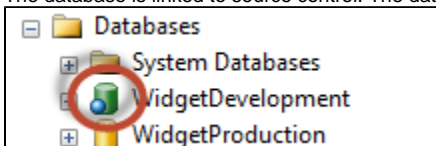
Development model:

- ☒ **Dedicated database**
Each developer will work on their own copy of the database
- ☐ **Shared database**
All developers will work on a single shared database

Link **Cancel**

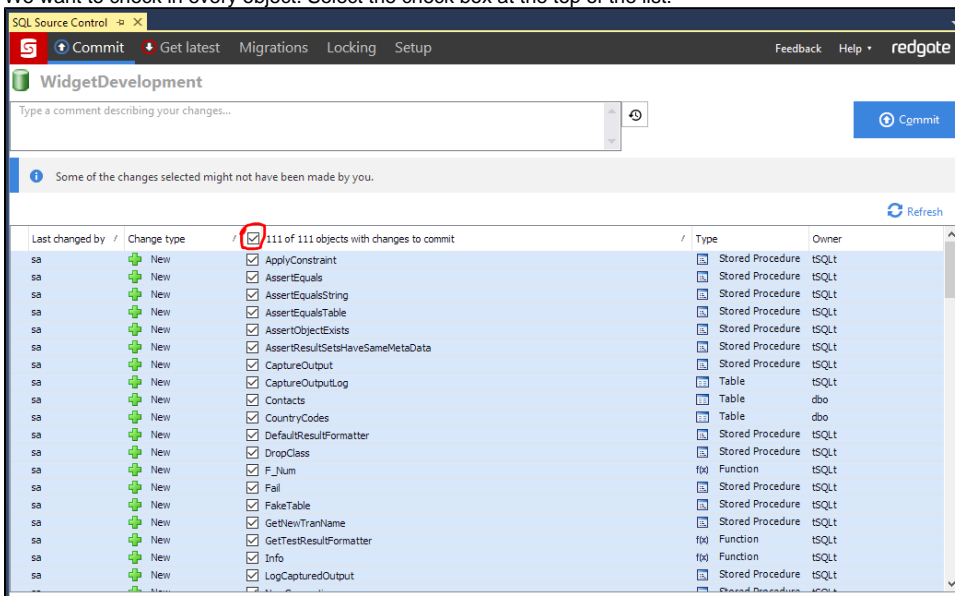
- On the left under Source control system, select **Subversion (SVN)**.
- In the **Database repository URL** field, specify the `file:///Z:/WidgetShop/` folder you created in your SVN repository.
This is where SQL Source Control will save your SQL scripts.
- Under Development model, select **Dedicated database**.
- Click **Link**.

The database is linked to source control. The database icon in the Object Explorer changes to show that the database is linked:



- Once the link to source control has been confirmed, click **OK**.
- To add the database objects to source control, click the **Commit** tab.

- We want to check in every object. Select the check box at the top of the list:



- Add a comment that describes the changes, for example, *new database files added*.
- Click **Commit**.
A progress dialog box is displayed while SQL Source Control commits the changes to your source control system.
- Click **OK** to close the dialog box.

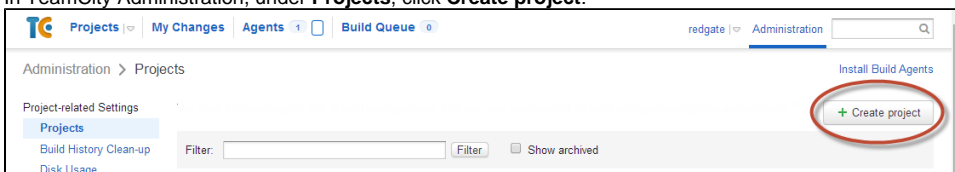
Source control is updated with your changes.

The SVN repository you've created is the shared directory that the team will use to check in code changes. Map it to the local folder that'll contain your working copy of the database. To do this:

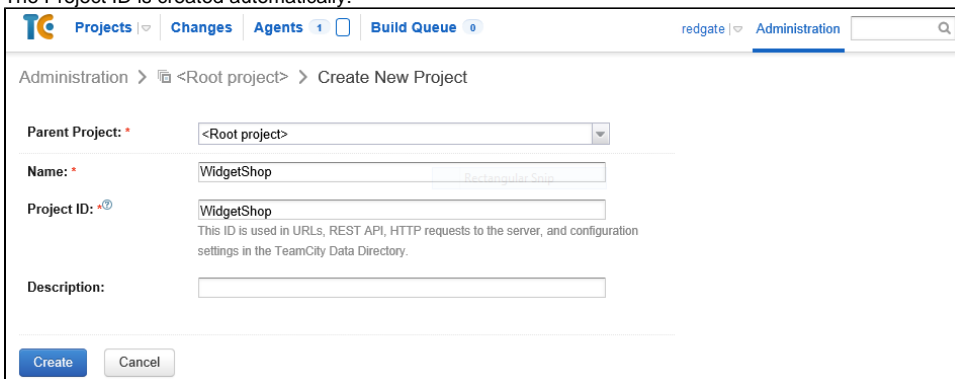
- Right-click on your Desktop and select **TortoiseSVN > Repo-browser**.
- Paste the URL for the SVN repository, *file:///Z:/WidgetShop/*. Click **OK**.
- At the **Repository Browser - TortoiseSVN** dialog, in the left panel, right-click on the *WidgetShop* folder and select **Checkout**.
- In the **Checkout directory** field, enter your local working folder, for example *C:\WidgetShop*.
- Click **OK**. A local copy of the WidgetShop folder is created.

Create a TeamCity project

- In TeamCity Administration, under **Projects**, click **Create project**:



- On the Create New Project page, add *WidgetShop* as the name for the project.
The Project ID is created automatically:



- Click **Create**.

4. Under **Build Configurations**, click **Create build configuration**:

The screenshot shows the 'Create Build Configuration' page in TeamCity. The breadcrumb navigation is 'Administration > <Root project> > WidgetShop'. The left sidebar shows 'Project Settings' with 'General Settings' selected. A yellow message bar at the top states: 'Project "WidgetShop" has been successfully created. You can now create a build configuration.' The form fields are: 'Name' (WidgetShop), 'Project ID' (WidgetShop, with a 'Regenerate ID' link), and 'Description' (empty). Below the form are 'Save' and 'Cancel' buttons. A section titled 'Build Configurations (19 left)' contains a message: 'Build configurations define how to retrieve and build sources of a project. There are no build configurations in this project.' At the bottom of this section are two buttons: '+ Create build configuration' (highlighted with a red circle) and '+ Create build configuration from URL'.

5. On the **Create Build Configuration** page, add *WidgetDevelopment* as the build name:

The screenshot shows the 'Create Build Configuration' page with 'WidgetDevelopment' entered in the 'Name' field. The 'Build configuration ID' field now contains 'WidgetShop_WidgetDevelopment'. The 'Description' field is empty. The 'Create' and 'Cancel' buttons are at the bottom.

6. Click **Create**.

Link the database to TeamCity

Configure the TeamCity VCS settings so it knows which network folder to monitor for changes:

1. On the **VCS Roots** page, from the **Type of VCS** drop-down list, select **Subversion**.
2. Under **VCS Root Name and ID**, in the **VCS root name** field, enter *SVN WidgetShop* as the unique name for this root. Under **SVN Connection Settings**, in the **URL** field, enter *file:///Z:/WidgetShop/*.

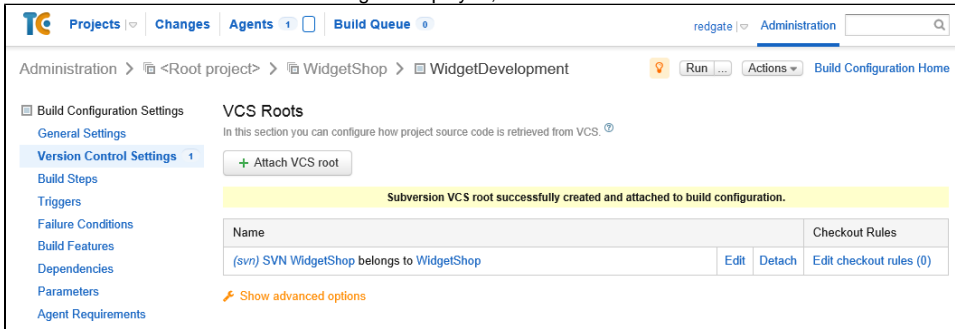
This was the Database repository URL you entered when you linked the database to source control.

The screenshot shows the 'New VCS Root' page in TeamCity. The breadcrumb navigation is 'Administration > <Root project> > WidgetShop > Version Control Settings > New VCS Root'. A yellow message bar at the top states: 'Build configuration successfully created. You can now configure VCS roots.' The form fields are: 'Type of VCS' (Subversion), 'VCS root name' (SVN WidgetShop, with a note: 'A unique name to distinguish this VCS root from other roots.'), 'VCS root ID' (WidgetShop_SvnWidgetShop, with a note: 'VCS root ID must be unique across all VCS roots. VCS root ID can be used in parameter references to VCS root parameters and REST API.'), 'URL' (file:///Z:/WidgetShop/), 'Username' (empty, with a note: 'The username specified here overrides the username from the URL.'), and 'Password' (empty). At the bottom are 'Create', 'Test connection', and 'Skip' buttons. A link 'Show advanced options' is also present.



If authentication is required for your source control server, you must specify a username and password.

3. Click **Test connection**. TeamCity checks that it can connect to the source control location.
4. Once the 'connection successful' message is displayed, close it and click **Create**.



The VCS settings are complete.



Continuous integration

These sections explain how to add automated steps to:

- [build a package](#) every time a change is checked into source control
- [run tSQLt tests](#) against the package
- [sync the package to a CI environment](#)
- [publish the package to a NuGet feed](#)

Build a package

In this section you'll use the DLM Automation TeamCity plugin to:

- [add a build step](#) to validate the SQL creation script and create a NuGet package
- [add a VCS trigger](#) to force a build every time a change is checked into version control
- [trigger a build](#) to test this step

Add a build step

You'll add the build step using one of Redgate's build runners for TeamCity.



A build runner allows a specific third party build tool, such as DLM Automation, to integrate with TeamCity. This example uses the Build runner that's part of Redgate's DLM Automation TeamCity plugin. It defines how to run the build and handle the output.

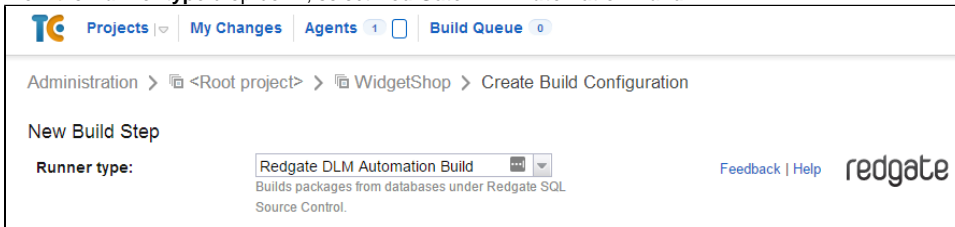
For more information on build runners, see [Configuring Build Steps](#) (JetBrains documentation).

When the build step validates the creation script, DLM Automation creates a temporary version of the database using LocalDB. This database is dropped once the build is complete.


1. From the **Build Configuration Settings** menu, select **Build Steps**.
2. Click **Add build step**:



- From the **Runner type** drop-down, select **Red Gate DLM Automation Build**:






- Under **Source-controlled database**, leave the **Database folder is my build VCS root** option selected. We've already configured the VCS root as our database location.
- Under **Output package**, at the **Package ID** field, enter *WidgetShopLatest*. This is the name of the NuGet package you'll create. The name must be unique and can't contain spaces.

 When you're thinking of a package name to use in your own environment, remember that it's going to be deployed to other databases. A generic name that describes what you're deploying is better than one that's specific to the build step or the database itself. For example, if your database is called *Development*, this wouldn't make sense as a package name deployed to your production database.


- Under **Temporary database server**, select **SQL LocalDB**. DLM Automation uses LocalDB to recreate and validate a temporary version of your database. This database is dropped once the build is complete.
- If you're using [Redgate's DLM Dashboard](#) (version 1.4.4.313 or later), DLM Automation can send schema information to it every time the build step runs. Under **DLM Dashboard integration**:
 - Enable the **Send schema information to DLM Dashboard** check box.
 - Enter the name or IP address of the machine hosting the DLM Dashboard.
 - Enter the DLM Dashboard port number. The default port is 19528.

Once you deploy changes (for example, by running the [sync step](#)), DLM Dashboard:

- recognizes the deployed schema from the information DLM Automation sent during the build step
- adds the schema to its list of recognized schemas, with the name <packageId-packageVersion>, for example, *WidgetShop1.0*
- labels the schema with the DLM Automation icon 
- labels the schema as an update , not drift 

For more information about how DLM Automation works with DLM Dashboard, see [DLM Automation integration](#) (DLM Dashboard documentation). For help understanding SQL Doc documentation, see [What's in the documentation?](#)

- DLM Automation can include [SQL Doc](#) database documentation in the NuGet package that it builds. To select this option, under **Database documentation**, click **Include database documentation**.

 Including database documentation will increase the size of the NuGet package and the time it takes to build, particularly for large databases.

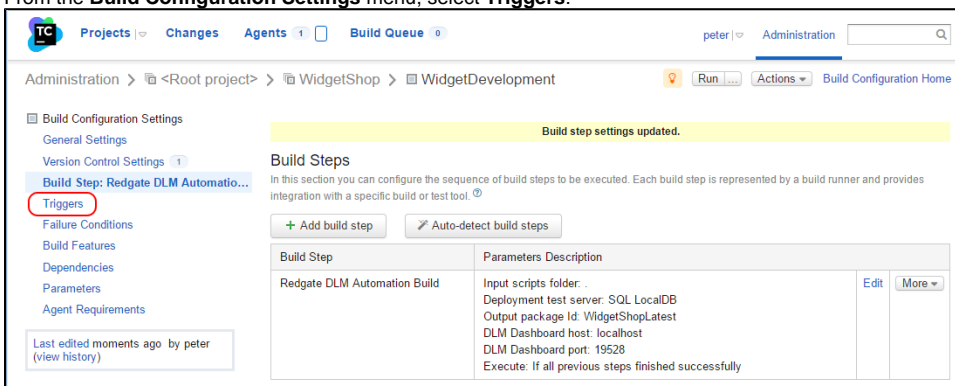
The documentation is stored in **db > docs > main.html** in the NuGet package. If you're using [DLM Dashboard](#) version 1.6.3 or later and have set up DLM Dashboard integration, you can also view the documentation directly from DLM Dashboard. See [DLM Automation integration](#) (DLM Dashboard documentation).

- Click **Save**.

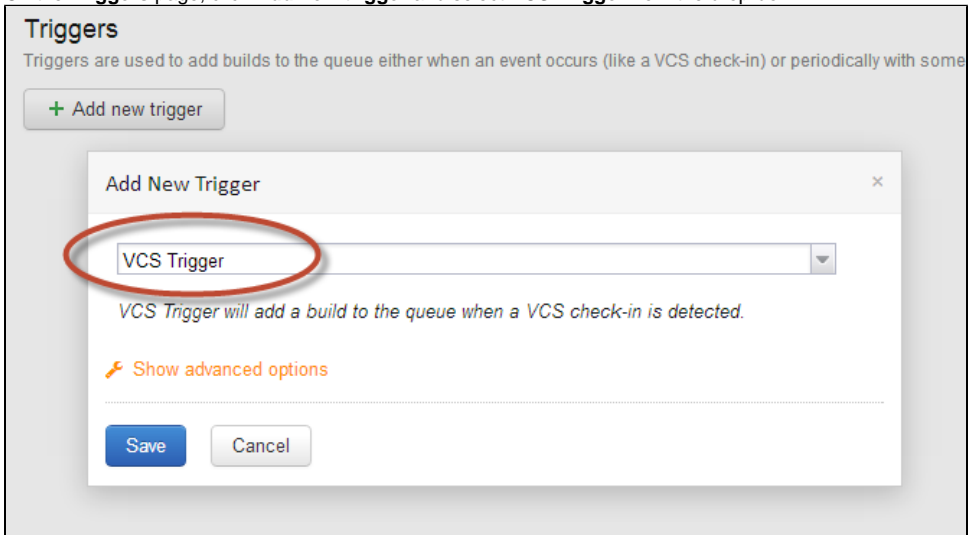
Add a VCS trigger

Add a trigger that'll force a build every time a change is checked into version control:

- From the **Build Configuration Settings** menu, select **Triggers**:



2. On the **Triggers** page, click **Add new trigger** and select **VCS Trigger** from the drop-down:



Triggers

Triggers are used to add builds to the queue either when an event occurs (like a VCS check-in) or periodically with some

+ Add new trigger

Add New Trigger

VCS Trigger

VCS Trigger will add a build to the queue when a VCS check-in is detected.

Show advanced options

Save Cancel

3. Click **Save**.
TeamCity will now run a build when you check in a change to your *WidgetDevelopment* database.

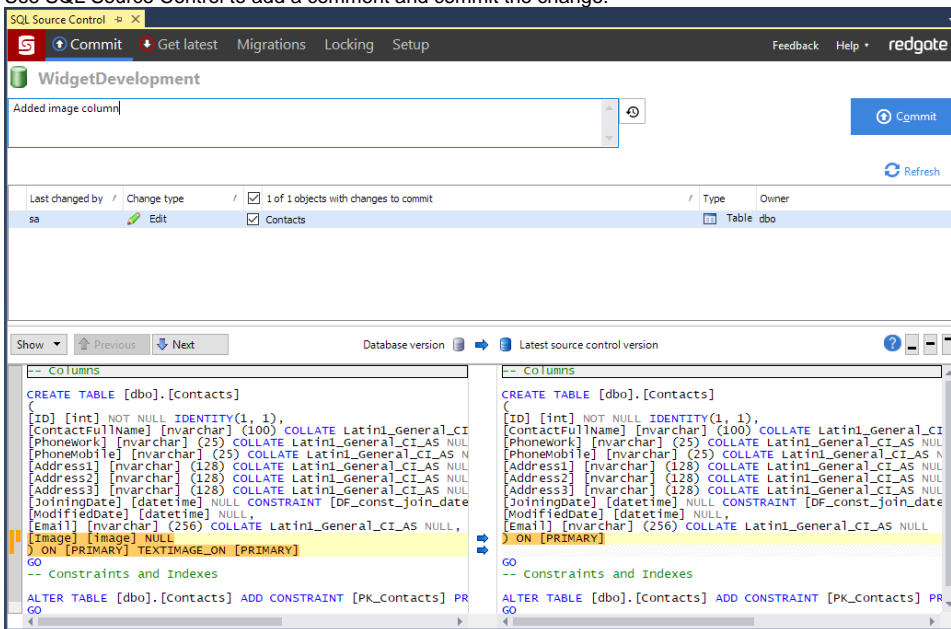
Trigger a build

Make a change to the *WidgetDevelopment* database to trigger a build automatically:

1. Add a new *Image* column to the *Contacts* table:

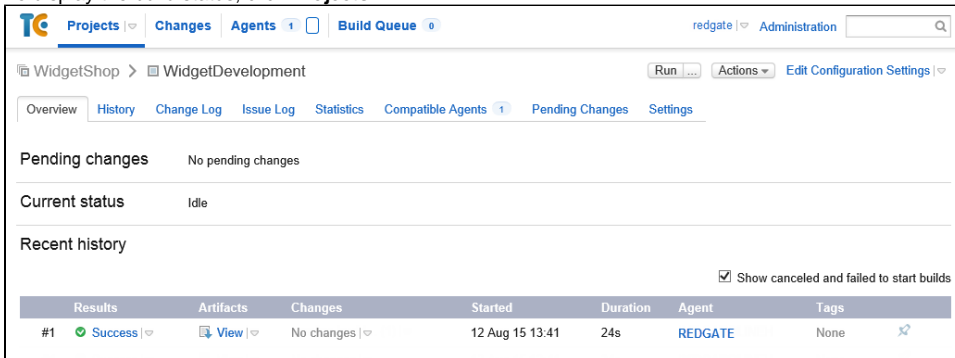
	Column Name	Data Type	Allow Nulls
🔑	ID	int	<input type="checkbox"/>
	ContactFullName	nvarchar(100)	<input type="checkbox"/>
	PhoneWork	nvarchar(25)	<input checked="" type="checkbox"/>
	PhoneMobile	nvarchar(25)	<input checked="" type="checkbox"/>
	Address1	nvarchar(128)	<input checked="" type="checkbox"/>
	Address2	nvarchar(128)	<input checked="" type="checkbox"/>
	Address3	nvarchar(128)	<input checked="" type="checkbox"/>
	JoiningDate	datetime	<input checked="" type="checkbox"/>
	ModifiedDate	datetime	<input checked="" type="checkbox"/>
	Email	nvarchar(256)	<input checked="" type="checkbox"/>
▶	Image	nchar(10)	<input checked="" type="checkbox"/>
		geography	<input type="checkbox"/>
		geometry	<input type="checkbox"/>
		hierarchyid	<input type="checkbox"/>
		image	<input type="checkbox"/>
		int	<input type="checkbox"/>
		money	<input type="checkbox"/>
		nchar(10)	<input type="checkbox"/>
		ntext	<input type="checkbox"/>

2. Use SQL Source Control to add a comment and commit the change:

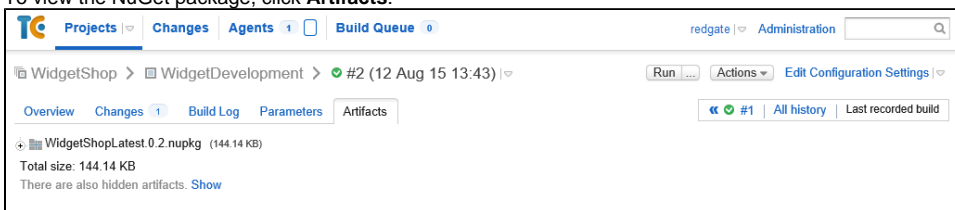


Once the change is committed to source control, TeamCity picks up the change and runs a build.

3. To display the build status, click **Projects**:



4. To show a summary of the log messages printed by the build, move your cursor over the **Success** drop-down button, and then click **Build log**.
5. To view the NuGet package, click **Artifacts**:



The build artifact is the name TeamCity gives to the output of a build step. In this example, the NuGet package is the artifact, and it's stored on TeamCity's server until we're ready to deploy it.

Test the package

In this section you'll:

- [check a SQL Data Generator file into version control](#). This will generate meaningful test data for the database
- [add a test step](#) so that every time there's a new build of your database, DLM Automation will create a temporary version and run tSQLt tests against it
- [trigger a build](#) to test this step

The WidgetDevelopment database already has tSQLt installed. When you're setting this up in your own environment, you'll need to install it. For more information, see [SQL Test](#).

Check the SQL Data Generator file into version control

We've provided a SQL Data Generator file that's already configured to generate data that's meaningful to the rows, columns and tables in the *WidgetDevelopment* database. You extracted this file from the [WidgetDevelopmentDatabaseCIDemo.zip](#) file.

Check the *WidgetTestData.sqlgen* file into SVN:

1. Using Windows Explorer, copy the *WidgetTestData.sqlgen* file from *WidgetDevelopmentDatabaseCIDemo > WidgetShop > Database* to *C:\WidgetShop*
2. Right-click on the *WidgetTestData.sqlgen* file and select **TortoiseSVN > Add**.
3. Right-click again and select **SVN Commit**.
4. At the **Commit** dialog, add a comment, for example, *sqlgen file added*. Click **OK**.

Add a test step

You can run different types of tSQLt tests, such as static analysis, unit or integration tests, against your temporary database.

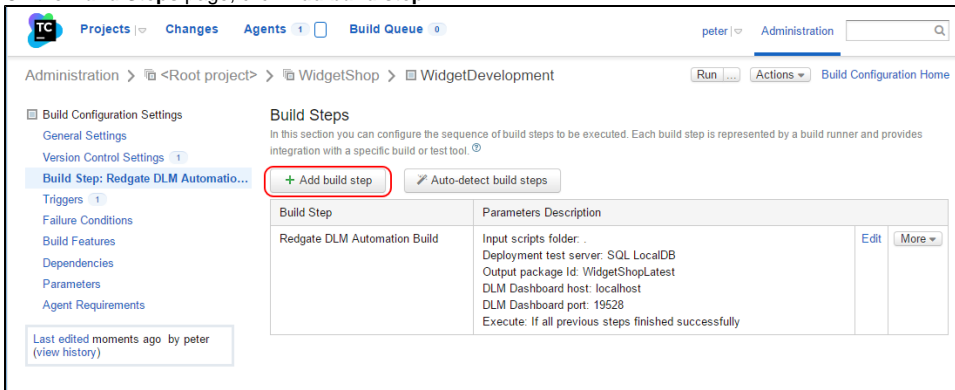
The database schema for *WidgetDevelopment* already includes four basic SQL Cop tests that will run against the temporary database and check for:

- procedures named SP_
- procedures using dynamic SQL without sp_executesql
- procedures with @@Identity
- procedures with SET ROWCOUNT

There's also a unit test in the schema that checks email addresses in the Contacts column. Once the tests are complete, DLM Automation generates test reports for review. The temporary database is then dropped.

To create the test step:

1. On the **Build Steps** page, click **Add build step**:

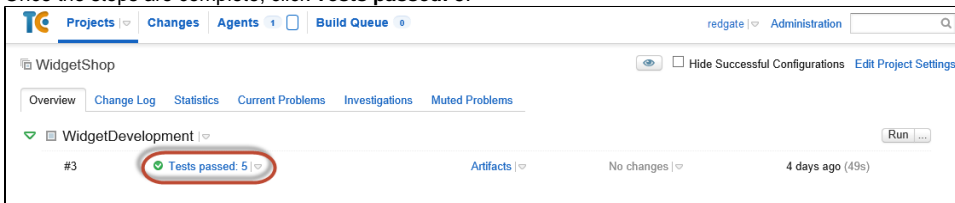


2. From the **Runner type** drop-down, select **Red Gate DLM Automation Test**.
3. Under **Output package**, at the **Package ID** field, enter *WidgetShopLatest*. This is the name of the NuGet package created by the build step.
4. Under **Temporary database server**, select **SQL LocalDB**. DLM Automation will use LocalDB to recreate the database for testing.
5. At the **Run tests** option, leave **Run every test** selected.
6. At the **Generate test data** option, select the **Populate the database with data before testing** check box, and enter the path to the *WidgetTestData.sqlgen* file. This must be relative to the VCS Root folder. In this example, the path is *file:///Z:/WidgetShop/*, so you just need to enter *WidgetTestData.sqlgen*.
7. Click **Save**.

Trigger a build

Make a change to the *WidgetDevelopment* database to trigger the build and test steps automatically:

1. Check in a change to the *WidgetDevelopment* database.
2. To display the build status, in TeamCity click **Projects > WidgetShop**. Once the build step is complete, the test step starts automatically.
3. Once the steps are complete, click **Tests passed: 5**:



4. Click the **Tests** tab. Details of the completed SQL Cop tests and unit test are displayed:

Status	Test	Duration	Order#
OK	SQL Cop.test Procedures Named SP_1 (SQL Cop)	< 1ms	1
OK	SQL Cop.test Procedures using dynamic SQL without sp_executesql (SQL Cop)	< 1ms	2
OK	SQL Cop.test Procedures with @@Identity (SQL Cop)	< 1ms	3
OK	SQL Cop.test Procedures With SET ROWCOUNT (SQL Cop)	< 1ms	4
OK	Unit Tests test Email in AddContact (Unit Tests)	< 1ms	5

To see more details about a specific test, or for troubleshooting a failed test, move your cursor over the drop-down button and select **Show in build log**.

Sync the package to a CI environment

Once your database has been through an initial test phase, it's good practice to run system, acceptance or smoke tests against it in an environment that simulates production. We'll call this the CI (Continuous Integration) database, although it's sometimes called staging or preproduction.

In this section you'll:

- [create a CI database](#)
- [add a sync step](#) so that every time your database has been built and tested, DLM Automation will synchronize it to your CI database
- [trigger a build](#) to test this step

Create a CI database

Create a blank database called *WidgetCI* as your CI database:

1. Open SQL Server Management Studio (SSMS).
2. Click **New Query**.
3. Execute the following SQL query to create the *WidgetCI* database:

```
CREATE DATABASE WidgetCI
GO
USE WidgetCI
GO
```

The database is now configured. It doesn't need to be checked into source control because it'll be updated automatically every time a change is made to the *WidgetDevelopment* database.

Add a sync step

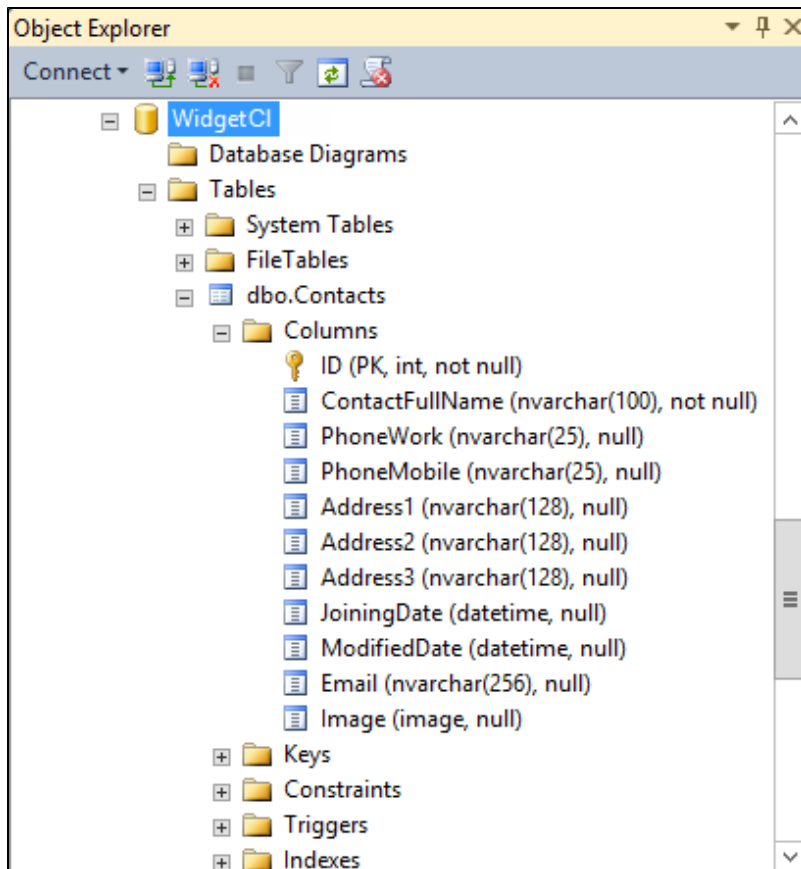
Add a sync step in TeamCity:

1. On the **Build Steps** page, click **Add build step**.
2. From the **Runner type** drop-down, select **Red Gate DLM Automation Sync**.
3. Under Output package, at the Package ID text box, enter *WidgetShopLatest*.
4. This is the name of the NuGet package you've already built and tested.
5. Under Database server, enter the name of the target server and database you want to update. In this example, we're updating the *WidgetCI* database using Windows authentication.
6. Click **Save**.

Trigger a build

Check in a change to the *WidgetDevelopment* database to trigger a build.

Once the build and test steps are complete, open SSMS to verify that the *WidgetCI* database has been synchronized with the contents of *WidgetDevelopment:WidgetCI* is in sync with *WidgetDevelopment*.



The *WidgetCI* database is now up to date with the latest version in source control. At this point, you could run additional tests against it.

Publish the package to a NuGet feed

Now that you've finished testing, you can publish the package to a NuGet feed. You can then use Octopus Deploy with DLM Automation to manage deployments.

We'll use TeamCity as a NuGet server to save you having to set up your own repository. For more details, see [Using TeamCity as a NuGet Server](#) (TeamCity documentation).

In this section you'll:

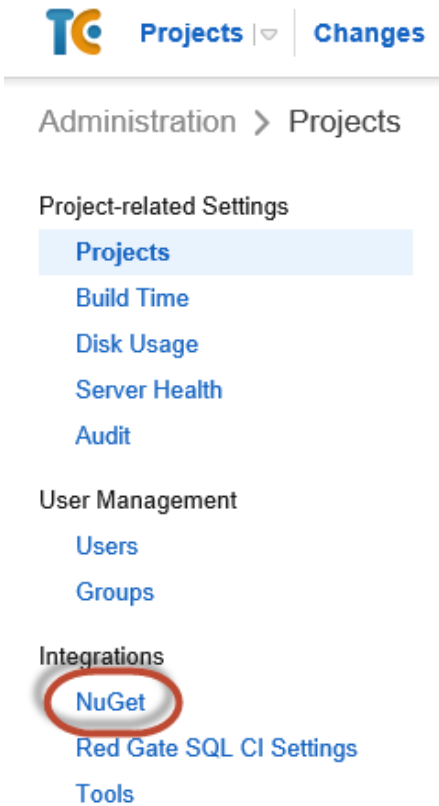
- [enable the TeamCity NuGet server](#) so you can publish the package to its NuGet feed
- [trigger a build](#) to test this step



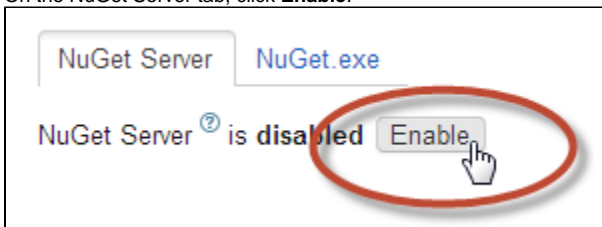
If you're using a different release management tool, check the help documentation provided by that tool vendor.

Enable the TeamCity NuGet server

1. In TeamCity Administration, under **Integrations**, click **NuGet**:



2. On the NuGet Server tab, click **Enable**:



Trigger a build

Trigger a build to publish the package to the NuGet feed automatically.



Release management

Now you can use Octopus Deploy and DLM Automation to deploy your database.

The easiest way to do this is using DLM Automation step templates for Octopus Deploy.

1. Copy the DLM Automation step templates to your Octopus Deploy library

After you've copied the DLM Automation step templates to your Octopus Deploy library, they're available whenever you add a process step in an Octopus Deploy project:

1. Go to the [DLM Automation step templates](#) on the Octopus Deploy library website.
2. In this example, we'll use the "Redgate - Create Database Release" and "Redgate - Deploy from Database Release" step templates. Click on the "Redgate - Create Database Release" template.

3. Click **Copy to clipboard**:

The screenshot shows the Octopus Deploy Library interface. At the top, there's a header with the Octopus Deploy Library logo. Below it, the template title 'Redgate - Create Database Release' is displayed, along with its export date and author. A description follows, explaining that the template creates resources for database deployment. A 'Requires' section specifies the DLM Automation version. A 'Version date' is also provided. The 'Parameters' section lists two parameters: 'Export path' and 'Delete files in export folder', each with a description and a default value. On the right side, there's a 'Copy to clipboard' button, which is highlighted with a red circle. Below this button, a 'Hide JSON' link is visible, followed by a JSON snippet representing the template's configuration.

You're now ready to paste the script from your clipboard into your Octopus Deploy library:

1. In Octopus Deploy, at the top of the page, click **Library**.
2. On the **Step templates** tab, click **Import**.
3. In the Import window, paste the copied template into the empty field.
4. Click **Import**.
5. Click **Save**.
6. Repeat steps 1 to 5 to copy the "Redgate - Deploy from Database Release" step template in the same way.

2. Create an Octopus Deploy project

1. In Octopus Deploy, click **Projects** and **All**.
2. Click **Add project**.
3. In the **Name** field, enter *Widget Deployment*.
4. Click **Save**.

You'll now add a series of deployment process steps to your Octopus Deploy project.

3. Add the "Download and extract database package" step

This step picks up the NuGet package of the database schema you're going to deploy.

1. Set up your NuGet package feed by doing one of the following:
 - register your existing external NuGet package feed with Octopus. For more details, see [Adding external package feeds](#).
 - configure your build server to push packages to the Octopus built-in repository. For more details, see [Using the built-in repository](#).
2. In the **Widget Deployment** project, on the **Process** tab, click **Add step** and select **Deploy a NuGet package**.
3. In the **Step name** field, enter *Download and extract database package*.
4. In the **Machine roles** field, enter *db-server* and press **Enter**.
This must match the role you assigned to the Tentacle.
5. In the **NuGet feed** field, select either the name of the external feed you registered when you [set up your NuGet feed](#), or the *Octopus Server (built-in)* repository.
6. In the **NuGet package ID** field, enter the name of the package without the version number. For example, if the package was called *WidgetShopLa test.0.1.nupkg*, you'd only enter *Widget*.
When the package is generated, NuGet package manager automatically adds a number. If we included it here, Octopus would only deploy the package that matched that name and version number. By removing the number, we're telling Octopus to always look for the latest package with that name.
7. In the **Environments** field, select *Production*.
If you leave this blank, the step will be accessible to all environments.
8. Click **Save**.

4. Add the "Create database release" step

This step creates the database deployment resources, including the Update.sql script.

1. On the project **Process** tab, click **Add step** and **Redgate - Create Database Release**.

2. In the **Machine roles** field, enter *db-server* and press **Enter**.
This must match the role you assigned to the Tentacle.
3. In the **Export path** field, enter the path the database deployment resources will be exported to.
This path will later be used in the "Deploy from Database Release" step. It must be accessible to all tentacles used in database deployment steps.
4. In the **Database package step** field, select *Download and extract database package*.
5. In the **Target SQL Server instance** field, enter the fully qualified SQL Server instance for the database you're deploying to.
6. In the **Target database name** field, enter the name of the database you're deploying to.
7. In the **Username (optional)** and **Password (optional)** fields, enter the SQL Server username and password used to connect to the target database.
If you leave these blank, Windows authentication will be used to connect to the target database.
8. In the **Environments** field, select *Production*.
If you leave this blank, the step will be accessible to all environments.
9. Click **Save**.

5. Add the "Review database deployment resources" step

This step pauses deployment to let you review the database deployment resources, including the *Changes.html* report, before allowing deployment to go ahead.

1. On the project **Process** tab, click **Add step** and select **Manual intervention required**.
2. In the **Step name** field, enter *Review database deployment resources*.
3. In the **Instructions** field, copy and paste this text:

```
Please review the schema and static data changes, warnings and SQL change script in 'Changes.html'.
```

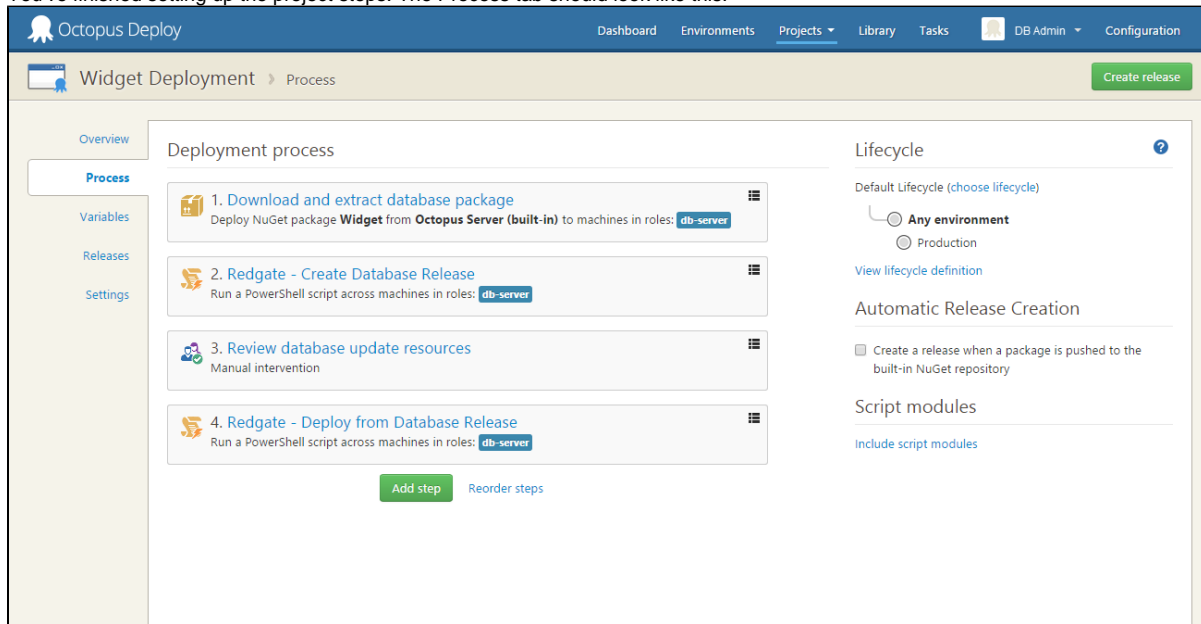
4. In the **Environments** field, select *Production*.
If you leave this blank, the step will be accessible to all environments.
5. Click **Save**.

6. Add the "Deploy from database release" step

This step uses the database deployment resources to deploy the database changes.

1. On the project **Process** tab, click **Add step** and select **Redgate - Deploy from Database Release**.
2. In the **Machine roles** field, enter *db-server* and press **Enter**.
This must match the role you assigned to the Tentacle.
3. In the **Export path** field, enter the path the database deployment resources will be exported to.
This must match the export path you entered in [4. Add the 'Create database release' step](#).
4. In the **Database package step** field, select *Download and extract database package*.
5. In the **Target SQL Server instance** field, enter the fully qualified SQL Server instance for the database you're deploying to.
6. In the **Target database name** field, enter the name of the database you're deploying to.
7. In the **Username (optional)** and **Password (optional)** fields, enter the SQL Server username and password used to connect to the target database.
If you leave these blank, Windows authentication will be used to connect to the target database.
8. In the **Environments** field, select *Production*.
If you leave this blank, the step will be accessible to all environments.
9. Click **Save**.

You've finished setting up the project steps. The Process tab should look like this:



7. Create a release

Now all the steps are set up, you can run your deployment process to create a release:

1. Create a blank database called *WidgetProduction*:
 - a. Open SQL Server Management Studio (SSMS).
 - b. Click **New Query**.
 - c. Execute the following SQL query to create the database:

```
CREATE DATABASE WidgetProduction
GO
USE WidgetProduction
GO
```

2. In the **Widget Deployment** project, on the **Process** tab, click **Create release**.
This page lets you add an optional release note.
3. Click **Save**.
4. Click **Deploy to Production** (or if there's more than one environment, click **Deploy** and select *Production*).

5. Click **Deploy Now**.

As the deployment process runs, Octopus Deploy shows the task progress list. The deployment pauses so you can review the database deployment resources:

The screenshot shows the Octopus Deploy interface for a deployment task named 'Widget Deployment' (Release 0.0.6). The task is currently in a 'Waiting...' state. The 'Task progress' section shows a list of steps: 'Deploy Widget Deployment release 0.0.6 to Production', 'Acquire packages', 'Step 1: Download and extract database package', 'Step 2: Redgate - Create Database Release', 'Step 3: Review database update resources' (which is the current step), and 'Step 4: Redgate - Deploy from Database Release'. The 'Artifacts' section on the right shows three files: 'Update.sql', 'Warnings.xml', and 'Changes.html' (which is circled in red). The 'History' section shows a table with columns 'When', 'Who', and 'What'. The 'Environment' section shows a 'Success' status for the deployment to Production.

6. Click **Changes.html** to download the Change report.

Use the report to review the update script, warnings, and details of what'll be added, removed or modified if you go ahead with deployment.

7. In Octopus Deploy, click **assign to me** and, in **Notes**, enter a comment to say you've reviewed the database deployment resources.

8. If you're happy with the report, click **Proceed**.

When the deployment is complete, the Task progress page looks like this:

The screenshot shows the Octopus Deploy interface for the same deployment task, now in a 'Success' state. The 'Task progress' section shows the same list of steps, all of which are now completed with green checkmarks. The 'Submitted by' field is set to 'db Admin'. The 'Info' field shows the time '16:20:21'. The 'Environment' section shows a 'Success' status for the deployment to Production.

You've now completed the deployment of the database package.