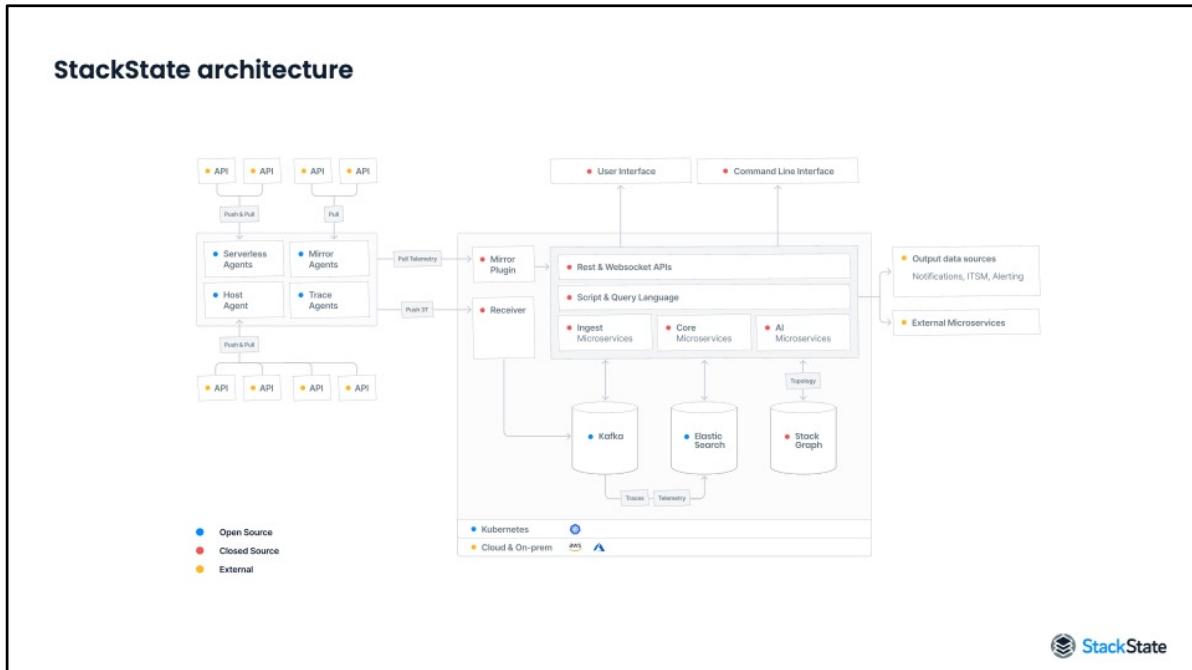


Welcome to the StackState administration training.
For this training we'll assume that you have finished the fundamentals training and/or have basic knowledge of StackState.

This course covers how to install, implement and run StackState.



The StackState architecture shows an overview of the StackState components and integration with external systems.

StackState runs on Kubernetes in the Cloud or data center. Linux packages are also available.

The left side of the architecture shows the input side, the data sources, providing information to StackState. Depending on the data source used, data retrieval can be push or pull based. Data is collected by Agents by interacting with the APIs of the different data sources. Collected data is forwarded to the Receiver, StackState's API, or pulled on-demand by the mirror plugin. The receiver is responsible for ingestion of topology, telemetry, and trace data. An example of a serverless agent might be an AWS lambda or Azure function that collects some data periodically from a data source and forwarding that information to the StackState Receiver. The Host Agent is an Agent that can be installed on a Linux server or Windows server to collect host level information. The Host Agent can also run in a Kubernetes/OpenShift cluster. The Host Agent supports various integrations to act as a data collector between various data sources and StackState. Host Agents periodically collect data.

Mirror Agents retrieve data on-demand when StackState, or the StackState user, requires the information. Trace agents are used to be integrated in applications to provide application level tracing.

The Receiver puts the received topology, telemetry, and trace data on Kafka. Different microservices read the data from Kafka. For example, topology information is processed by StackState's synchronization process to update topology information accordingly. Topology information is then stored in the versioned graph database StackGraph. Telemetry is processed by telemetry streams and health checks when the received data matches the stream's query. Traces and some telemetry data is temporarily stored in Elasticsearch for historical use in telemetry charts, for example. On top of the microservices, a script and query language can be used to query StackState that provides access to the topology, telemetry, traces, and time aspects.

The right and top side of the architecture show various methods of interacting with StackState. StackState can be accessed via the browser and via the command line interface (CLI). Depending on requirements, various functions can interact with external systems/sources to, for example; file tickets, send alerts, or invoke webhooks.

Installation



This section covers the StackState installation process.

Installation

- Software as a Service (SaaS)
- Kubernetes
- Linux packages



StackState is offered as a SaaS and is available for on-premise / data center installation. For on-premise, StackState can be deployed on Kubernetes and Linux packages are available. For Kubernetes, Helm charts are available. There are rpm and deb packages available for Linux. StackState will move away from Linux packages in the future.

We'll cover the Kubernetes and Linux packages installation process in the next sections. We also cover the installation process of the Host Agent and CLI.

Installation

Kubernetes



This section covers the StackState installation process on Kubernetes.

Installation – Kubernetes

- Required nodes:
 - High availability setup:
 - Recommended
 - Minimal
 - Non-high availability
- Requirements - <https://docs.stackstate.com/setup/requirements>
- Deployment using Helm chart
- Requires image pull credentials - <https://support.stackstate.com/>



The stackstate installation on Kubernetes requires several nodes
StackState can be deployed on an OpenShift, cloud or on-premise
Kubernetes cluster

There are two types of setup – high availability and non-high availability.
High availability can be deployed as either the recommended setup or a
minimal setup.

For test or development purposes you could opt for a non high
availability setup

The required nodes, virtual processors and memory for each setup can
be found on the StackState documentation website

StackState is deployed using a helm chart

We need credentials to pull in the StackState container images. These
credentials can be requested via the StackState support site.

Let's see how the installation process works.

7 October 2021

Installation – Kubernetes

```
helm repo add stackstate https://helm.stackstate.io
"stackstate" has been added to your repositories
  helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "stackstate" chart repository
Update Complete. Happy Helming!
```

```
kubectl create namespace stackstate
```



We're going to assume that Helm is already installed.

We'll add StackState to the Helm repository using the "repo add" Helm command.

We also need a Kubernetes namespace in where StackState is going to be deployed via kubectl. We're creating the namespace called "stackstate" in this example.

Installation – Kubernetes

```
./generate_values.sh
Please provide the license key (-l):
Please provide the username for pulling StackState Docker images (-u):
Please provide the password for pulling StackState Docker images (-p):
Please repeat the password for confirmation:
Please provide the base URL for StackState, for example https://my.stackstate.host (-b): http://localhost:8080
Please provide the password for the 'admin' user that will be set for StackState (-d):
Please repeat the password for confirmation:
Please provide the password that will be set for StackState's admin api (-a):
Please repeat the password for confirmation:
StackState has Kubernetes support for which an agent needs to be installed on the cluster.
See the Kubernetes Stackpack at https://docs.stackstate.com/ for the details.
Do you want to install the StackState Kubernetes Agent on the cluster? [yn] y

StackState and the agent use a cluster name to refer to this Kubernetes cluster.
For convenience we suggest to use the same name as in your kube context.
Please provide a name for the Kubernetes cluster : my-cluster
Generated 'values.yaml'.

Make sure to store this file in a safe place for usage during upgrades of
StackState. Generating a new file will generate a new API key which
would require updating all running agents and other clients.

Now for first time installation create a namespace for StackState first:
kubectl create namespace stackstate

Use the following command to install or upgrade StackState into
namespace 'stackstate' using helm release 'stackstate';

helm upgrade --install --namespace stackstate --values values.yaml stackstate stackstate/stackstate
```



Before deploying, we need a values.yaml file. The values.yaml file contains the StackState API key, credentials to pull in containers, etc.

We can use the generate_values.sh script to generate a values.yaml for us. The generate_values.sh script can be run in interactive mode and in non-interactive mode. Non-interactive mode allows you to pass arguments to the script. We're showing the interactive mode here.

The first question is about the StackState license key. The license key has been shared with your company via email.

The second and third questions are about the required credentials to be able to pull in StackState's container images. The password is not shown.

The fourth question is about the StackState base URL. The base URL is used by the user and agents to connect to StackState. You can change this value in the generated file, if you need to.

The fifth question is about the admin user's password. The admin user

can be used to access the StackState UI.

The sixth question is about whether you want to install the StackState Kubernetes Agent in the cluster. The StackState Kubernetes Agent reports on the Kubernetes cluster to StackState. You need to provide the name of the cluster.

After these questions, the values.yaml file is generated in your current path. All values provided to the interactive script is stored in the values.yaml file. Now we are ready to deploy StackState using helm. The values.yaml file is passed to the 'helm upgrade' command. Note that this deploys the high availability setup. If you want to deploy the non-high availability setup, you have to download the nonha-values.yaml and add that to the 'helm upgrade' command. We'll show you on the next slide.

Installation – Kubernetes

```
helm upgrade --install --namespace stackstate --values values.yaml stackstate stackstate/stackstate

helm upgrade --install --namespace stackstate --values values.yaml --values nonha-values.yaml stackstate stackstate/stackstate

helm list --namespace stackstate
NAME      NAMESPACE   REVISION   UPDATED             STATUS        CHART          APP VERSION
stackstate  stackstate    2          2021-09-06 09:44:19.094864767 +0000 UTC deployed  stackstate-4.4.1  4.4.1
```



Here is the ‘helm upgrade’ command to deploy StackState in your cluster. We are using the previously created namespace “stackstate” here. The values.yaml was generated by the generate_values.sh script.

The first statement deploys StackState as high availability setup and the second command shown here deploys StackState in a non-high availability setup. The nonha-values.yaml can be downloaded via the documentation website and specifies the number of replicas for certain deployments.

It may take a while before StackState is deployed.

The ‘helm list’ command shows the used chart and the app version. In this case, StackState release version 4.4.1 has been deployed.

Installation

Linux packages



This section covers the StackState installation process on Linux.

Installation – Linux packages

- RPM & deb Linux packages
- Setup type:
 - Development
 - Proof-of-Concept (PoC)
 - Production
 - Custom
- One or two nodes
- Requires Java 1.8

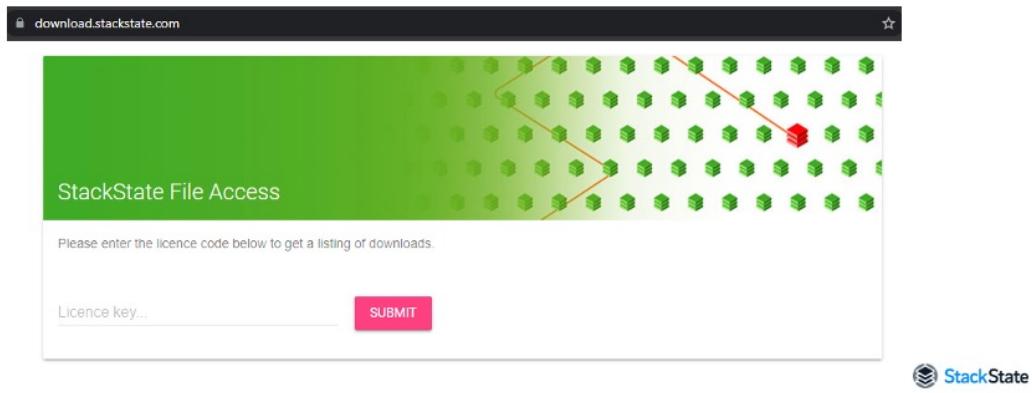


Linux packages are made available in RPM and DEB format. The Linux packages are distributed via a website. You need one or two nodes depending on the setup type you require. The development and proof-of-concept setups require only a single node. The custom setup by default assumes a single node. The production setup type requires two nodes. StackState is installed on one node and StackGraph, StackState's graph database, is installed on the other node. The StackState node will host the user interface and the receiver.

The node sizing requirements of each setup are described on the documentation website. StackState runs on the JVM. Before installing StackState we need to install Java on each node to meet that requirement.

Installation – Linux packages

- Packages available for download from <https://download.stackstate.com/>
- Requires a StackState license key
- Rpm & deb packages available



The StackState Linux packages can be downloaded from StackState's software distribution website, located at:
<https://download.stackstate.com>.

A StackState license key is required to be able to download the required files and to install StackState. The license key has been shared with your company via email.

Installation – Linux packages

The screenshot shows a web interface titled "StackState File Access". At the top, there is a green header with the StackState logo and a "SUBMIT" button. Below the header, a message says "Please enter the licence key below to get a listing of downloads." There is a text input field labeled "Licence key..." and a "SUBMIT" button. Below this, a table lists six download links:

File	Action
stackstate-4.4.0.x86_64.rpm	DOWNLOAD
stackstate-4.4.0.zip	DOWNLOAD
stackstate_4.4.0_amd64.deb	DOWNLOAD
sts-cli-4.4.0.zip	DOWNLOAD
sts-cli-4.4.0-windows.exe	DOWNLOAD
sts-cli-4.4.0-linux64	DOWNLOAD

In the bottom right corner of the interface, there is a "StackState" logo.

After entering the StackState license key, a list of files is shown. The files listed here are available for download. The filenames that are prefixed with 'stackstate' are the StackPack Linux packages. The filename also shows the StackState release version. You can download the package you require using the download button. The rpm and deb is available. The zip file is available because of historical reasons. You can copy the link from the download button if you want to use wget, cURL or any other download tool.

StackState has a release schedule for feature releases. Feature releases are major releases and will be released on a quarterly schedule. Maintenance releases are minor releases and will be released on an as-needed basis.

Installation – Linux packages

Install Java

```
[user@ip-172-30-0-179 ~]$ sudo yum install java-1.8.0-openjdk-headless.x86_64
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
Resolving Dependencies
--> Running transaction check
---> Package java-1.8.0-openjdk-headless.x86_64 1:1.8.0.302.b08-0.amzn2.0.1 will be installed

[user@ip-172-30-0-179 ~]$ java -version
openjdk version "1.8.0_302"
OpenJDK Runtime Environment (build 1.8.0_302-b08)
OpenJDK 64-Bit Server VM (build 25.302-b08, mixed mode)
```

Install StackState

```
[user@ip-172-30-0-179 ~]$ sudo rpm -i ./stackstate-4.4.0.x86_64.rpm
Creating system group: stackstate
Creating system user: stackstate in stackstate with stackstate user-daemon and shell /bin/false
SETUP environment not specified and no tty present, setup is not complete.
\ef3implease run /opt/stackstate/bin/setup.sh to complete installation\ef39m
```



Let's walk through the installation steps. Since we need Java, we install that first. We'll use OpenJDK headless version 1.8 here.

After installing Java, we can proceed to install StackState using your Linux distribution's rpm or dpkg commands. You need to install the Linux package on each node. After installation you are asked to run a setup script to configure StackState.

```
sudo yum install java-1.8.0-openjdk-headless.x86_64
sudo rpm -i ./stackstate-4.4.0.x86_64.rpm
```

Installation – Linux packages

Configuration

```
[user@ip-172-30-0-179 ~]$ sudo /opt/stackstate/bin/setup.sh
/opt/stackstate/etc/application_stackstate_1628533681353.tgz
Please specify the SETUP type for this StackState installation: DEVELOPMENT, POC, PRODUCTION-STACKSTATE, PRODUCTION-STACKGRAPH, CUSTOM. More information on: https://l.stackstate.com/installation/ DEVELOPMENT
Setting up StackState for development mode
Please specify the LICENSE_KEY for this StackState installation: [REDACTED]
Checking license key
Key successfully registered!
Please specify the STACKSTATE_BASE_URL for this StackState installation, the URL at which StackState is accessible. More information on: https://l.stackstate.com/installation/ http://[REDACTED]:7070
Please specify the RECEIVER_BASE_URL for this StackState installation, the URL at which the agent can reach StackState. More information on: http://l.stackstate.com/installation/ http://[REDACTED]:7077
Created symlink from /etc/systemd/system/stackgraph.service to /usr/lib/systemd/system/stackgraph.service.
Created symlink from /etc/systemd/system/multi-user.target.wants/stackgraph.service to /usr/lib/systemd/system/stackgraph.service.
Created symlink from /etc/systemd/system/stackstate.service to /usr/lib/systemd/system/stackstate.service.
Created symlink from /etc/systemd/system/multi-user.target.wants/stackstate.service to /usr/lib/systemd/system/stackstate.service.
```

Start the Systemd service:

```
[user@ip-172-30-0-179 ~]$ sudo systemctl start stackstate
```



The setup script will configure StackState for you. It generates a configuration file for you in the StackState installation directory.

The first question is about the setup type. You can choose development, poc, custom, production-stackstate, or production-stackgraph. Select production-stackstate on the production node that is to run StackState and production-stackgraph on the production node that is to run the StackState database. The production-stackstate setup type installs only the StackState systemd service on the node while production-stackgraph only installs the StackGraph systemd service on the node for a distributed setup. These two setup types are used for acceptance or production environments. The development, poc, and custom setup types are for single node installations. The development setup type limits the number of components and relations in a StackState view to 1000 and is useful as a development environment. The poc setup type almost gives the same power as the production setup, but is not suited for processing perpetual data streams. The development setup will require the least system resources of all setup types. The custom setup type will not set restrictions nor will it ask you for any input. All

setup types except for the custom type will check for available resources.

It's likely that you choose the development or production-stackstate/production-stackgraph setup type here. We will select the development setup type here.

The second question is about the StackState license key. You can enter StackState's license key here. The license key will be checked and registered. You don't need an internet connection for license key registration.

The third question is about the StackState base URL. The StackState base URL will be used by StackState to create user facing links. Set the StackState base URL to the URL you are planning to use in the browser to access StackState's user interface. Depending on your environment, the StackState base URL may be set to:

`http://stackstate.yourdomain.com`. In this case, we're using an IP-address, so, the StackState base URL will become `http://<ip>:7070`. TCP Port 7070 is the default port where the user interface listens on for the Linux packages. You can put a reverse proxy in front if you prefer. You can change the StackState base URL afterwards if need to.

The fourth, and final, question is about the receiver base URL. The receiver base URL is used to configure agents and external sources pushing data to StackState. Similar to the StackState base URL, the receiver base URL defines the URL on where the receiver is going to be accessible. Depending on your environment, the receiver base URL may be set to: `http://stackstateapi.yourdomain.com`. In this case, since we're using an IP address, the receiver base URL becomes `http://<ip>:7077`. TCP port 7077 is the default port on which the receiver listens on for the Linux packages. You can change the receiver base URL afterwards if need to.

After these questions you can see some symlinks being created to set up the systemd services. The output may differ between setup types. You can now start the StackState systemd service. The StackState systemd service depends on the StackGraph systemd service for all but one setup types. This means that the StackGraph service has to be started before the StackState service can be started. For single node setup types, the StackGraph service is started automatically when you start the StackState service. For the node where you selected setup type 'production-stackgraph' you have to start `stackgraph.service` instead of

stackstate.service.

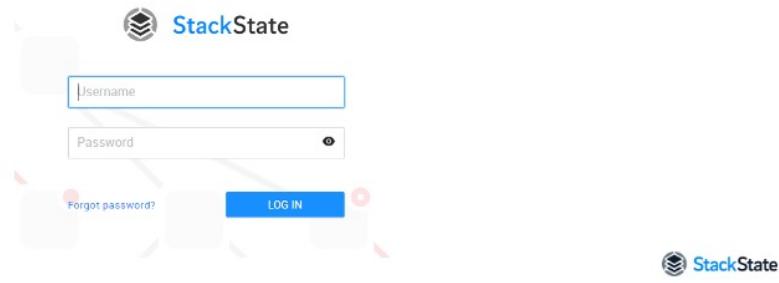
```
sudo /opt/stackstate/bin/setup.sh  
sudo systemctl start stackstate
```



After installation of StackState, StackState does not contain any data.
You can use the StackState User Interface to configure StackState.

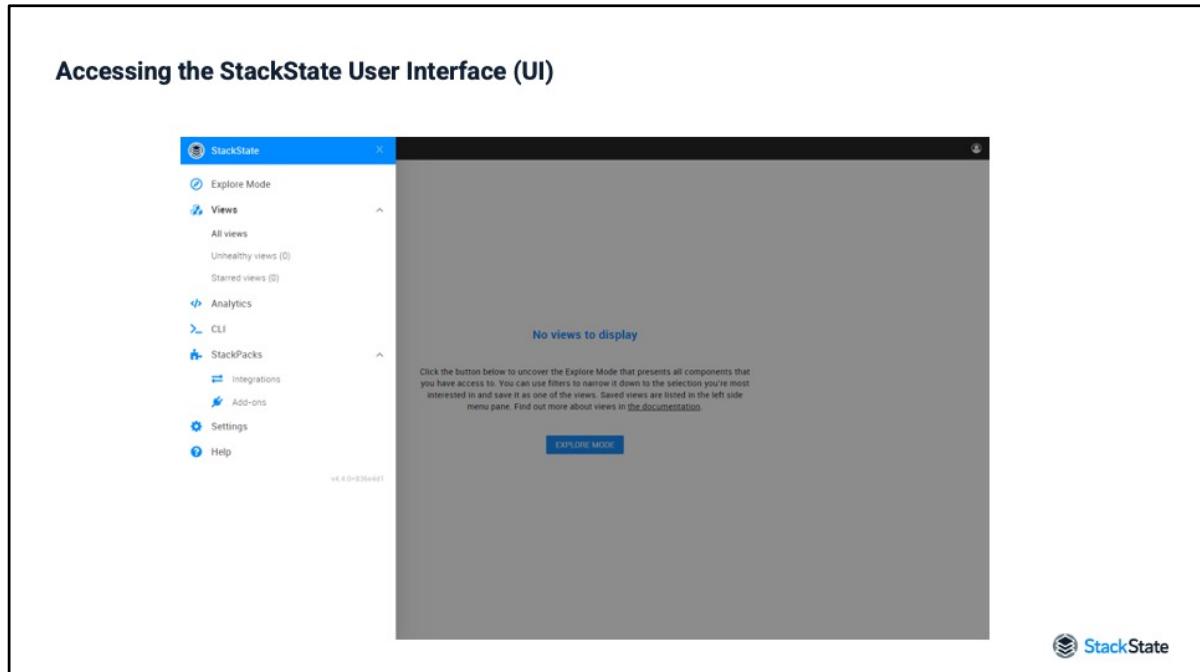
Accessing the StackState User Interface (UI)

- User interface is browser based
- Supported web browsers: Chrome & Firefox
- Linux packages: `http://<host/IP>:7070`
- Kubernetes: ingress specific, service `stackstate-router:8080`



StackState's user interface can be reached using web browsers Chrome or Firefox. For the Linux packages, StackState listens on TCP port 7070 and thus can be reached on `http://<ip/host>:7070`. For Kubernetes it depends on the Ingress configured, the Kubernetes Service `stackstate-router` listens on port TCP/8080. For SaaS, a link will be made available and shared with your organization.

First time opening the StackState user interface will give you a login screen. A default set of users defined after installation; admin, platformadmin, guest, and power. For the Linux packages, the default password for these users is `topology-telemetry-time`. For Kubernetes, the password is set in `values.yaml` during the installation process. It is possible to set up LDAP, Open ID Connect, or configure local user accounts. We'll cover users in more detail in the section about Security and RBAC.



After logging in into StackState for the first time, StackState is entirely empty. Next step is to integrate StackState with data sources. StackState ships with integrations that you can use to integrate with your data sources. These out of the box integrations are called StackPacks. You can proceed by installing one or more StackPacks to get your environment integrated.



Let's have a look at StackPacks. By default, StackState is shipped with StackPacks. More can be added and it is also possible to create your own StackPack.

Installation – StackPacks

- StackPack types:
 - Add-on
 - Integration
- Adding functionality to StackState or integrating with external data sources
- CLI commands

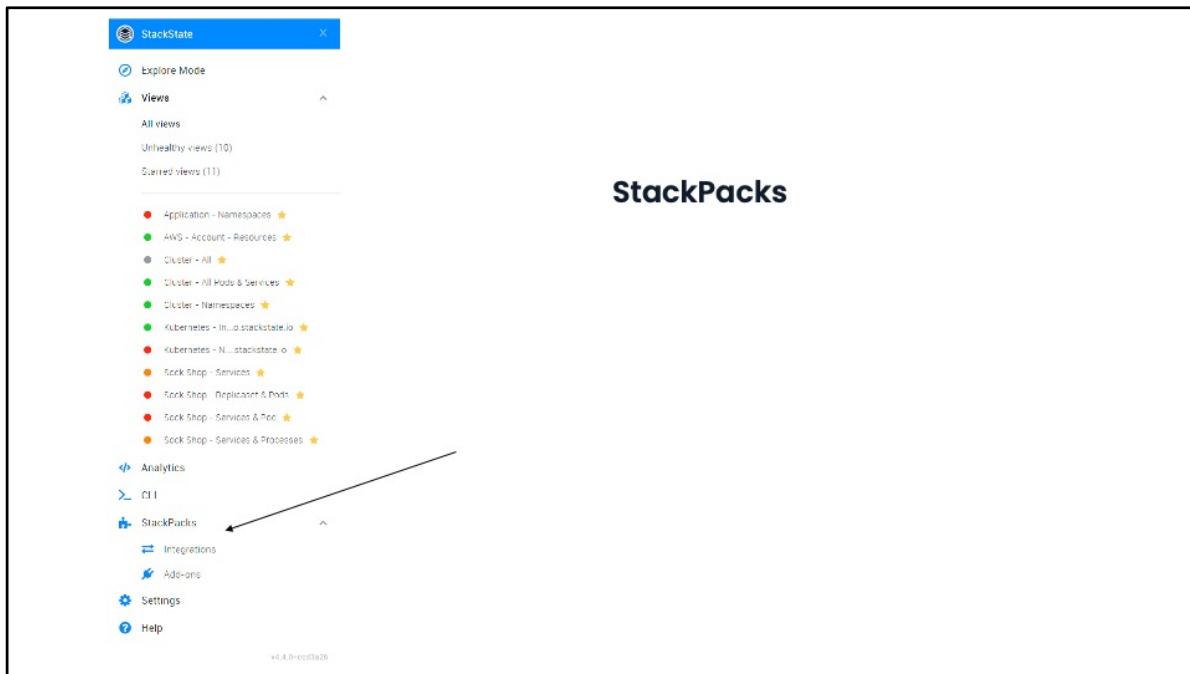


StackPacks are plugins for StackState that extend functionality and provide automated integration with external systems. Functionality of StackState can be extended with so called add-on StackPacks, like the Autonomous Anomaly Detector (AAD). Integration StackPacks can be used to make an integration with external data sources for your topology and/or telemetry data.

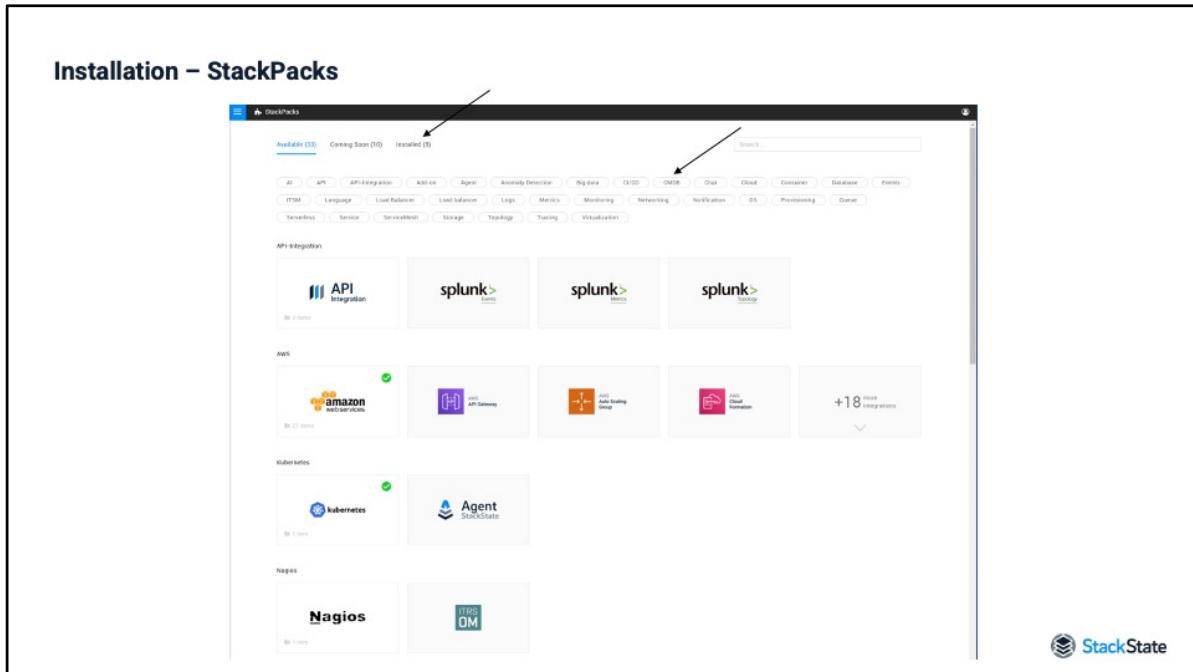
Each StackPack may have different prerequisites and installation instructions. Generally, integration typed StackPacks are two fold; it contains configuration of StackState and user instructions on how to get data flowing between data source to StackState. For example, some integrations require the StackState Agent to collect information from a data source. The StackState Agent collects data and forwards it to StackState. The configuration part of the StackPack tells StackState how to handle the received data for components and relations to materialize.

The lifecycle of a StackPack can be managed from the user interface and via the CLI. CLI commands are useful in deployment pipelines, for example. Let's have a look at the StackPack pages in the user interface.

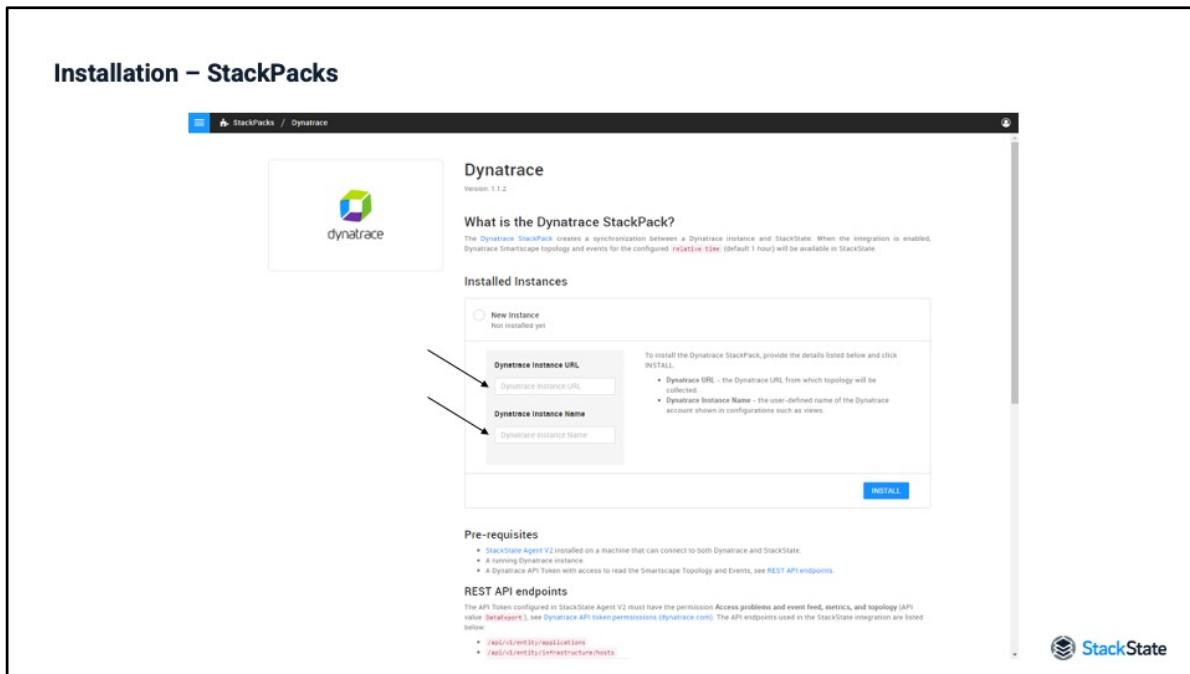
CLI commands can be found in the CLI section.



All available StackPacks can be found on the StackPack page which can be accessed from the main menu.

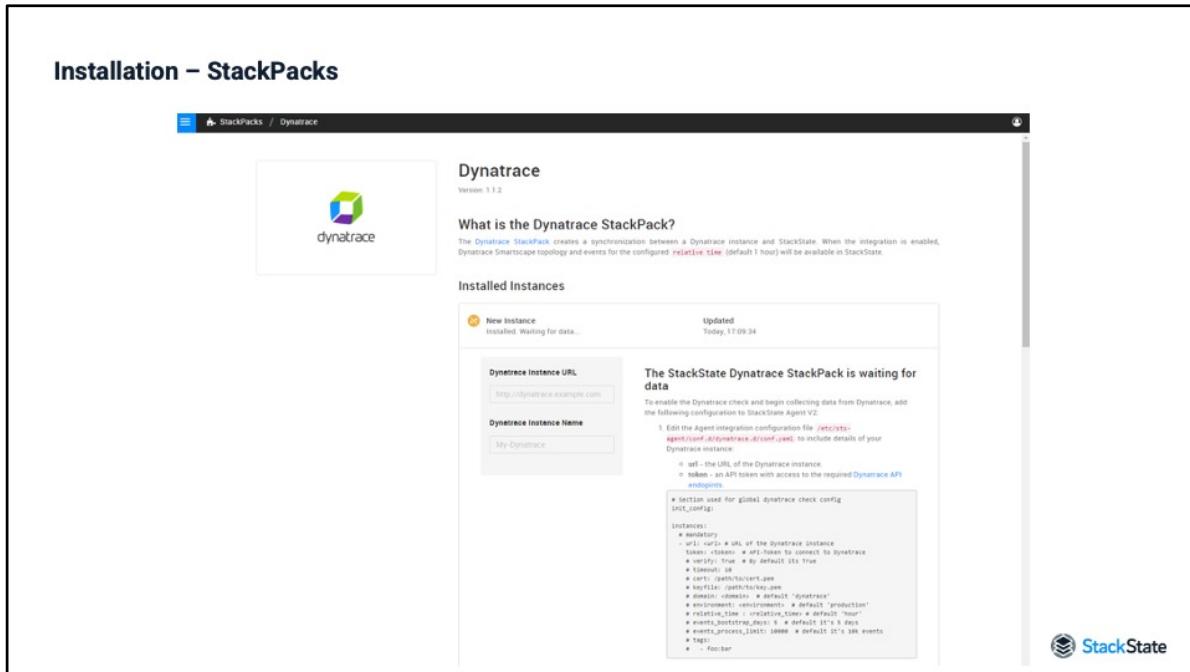


On the StackPack page you can see all available StackPacks. Installed StackPacks have a check mark next to them. You can also filter on installation status and category at the top of the page. It is also possible to find a StackPack by its name using the search box. You can click on any StackPack for information and follow the installation instructions.



Here is the Dynatrace StackPack. The StackPack page lists information about the integration with Dynatrace. All StackPack pages look similar. The StackPack page provides information about the integration, what can be expected, and the prerequisites.

There is a section that is called 'installed instances'. Here you can install the StackPack. The majority of StackPacks ask for some information that you can provide. Usually you need to tell the StackPack where the data source can be reached. This is usually a URL or endpoint of the data source's API. The URL is called the instance URL. This instance URL will be used by the StackState Agent, in case of Dynatrace, to collect information and send that to StackState. StackState knows using the instance URL how to process the data in topology synchronization. Some StackPacks also ask for an instance name. This name is usually used in the naming of the views such that you can differentiate between instances of the data source.



After providing the requested information, you can click on install. The StackPack will be installed and the status of the installed instance will change to installed state or, for most StackPacks, to waiting for data state. The input boxes have been grayed out at this stage. You could uninstall the StackPack again if you made a mistake in the input.

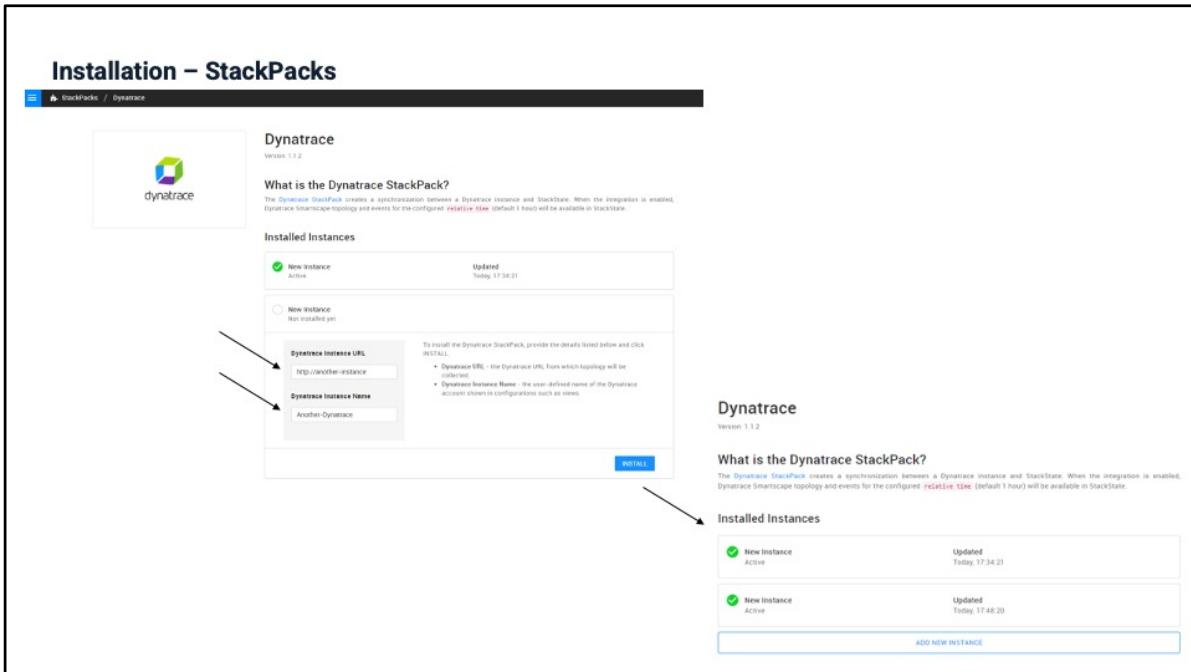
The waiting for data step shows information related to getting data into StackState. In case of Dynatrace, the integration is dependent on the StackState Agent for data collection. This step will show the steps necessary to configure the StackState Agent and getting data collected correctly.

The status will change from waiting for data to installed once StackState has received data from the data source.

Installation – StackPacks

The StackPack shown here is correctly installed, it shows that it is active. This page will show when the StackPack is in active state. The page will also show when you come back to the StackPack page. This page shows information about the next steps. The information shown here differs between StackPacks.

You can uninstall the StackPack from this page if you want to. Most StackPacks install StackState views, if you click on explore your data the page listing the views will be opened. You can find the StackPack related views there. In case of Dynatrace, you can find Dynatrace prefixed views listed there.



Installing a StackPack is actually installing an instance of the StackPack. It is possible to connect to multiple instances of a data source. For example, we have two Dynatrace instances installed in our environment. Maybe you have one instance per team or one for acceptance and one for your production environment. Regardless of use case, you can have multiple data source instances in your environment. Each StackPack instance refers to a single data source instance. You can install multiple instances of a StackPack if you have multiple instances of a data source. A StackPack instance configures StackState in such a way that StackState can handle data received from each data source instance without conflicting with another instance of the StackPack.

Installing another StackPack instance is exactly the same process of installing the very first instance. The only difference is that there are more instances listed under 'installed instances'. You can collapse/expand each instance by clicking on the instance.

Not all StackPacks support multiple instances, you can only install one instance of these StackPacks. The StackState Agent StackPack is an

example of a StackPack that supports only a single instance. The majority of StackPacks are multi-instance, this differs between StackPacks.

Uninstalling a StackPack instance is as simple as expanding the instance you want to uninstall and then click the uninstall button. You cannot uninstall all StackPack instances in one go, you have to uninstall one instance at a time. The CLI can be used to do this in one go by chaining some commands if you need to.



The StackState Agent is commonly used to act as data collector for StackState. Other StackPacks may depend on the StackState Agent for collecting data from a data source. There is a StackState Agent StackPack available to help with the installation of the StackState Agent. Additional steps will be provided to configure an integration correctly once the StackState Agent has been installed.

Installation – StackState Agent

- StackState Agent v2 StackPack
- Support for various operating systems;
 - Ubuntu
 - Debian
 - CentOS
 - RHEL
 - Fedora
 - Windows
- Docker container
- Networking requirements



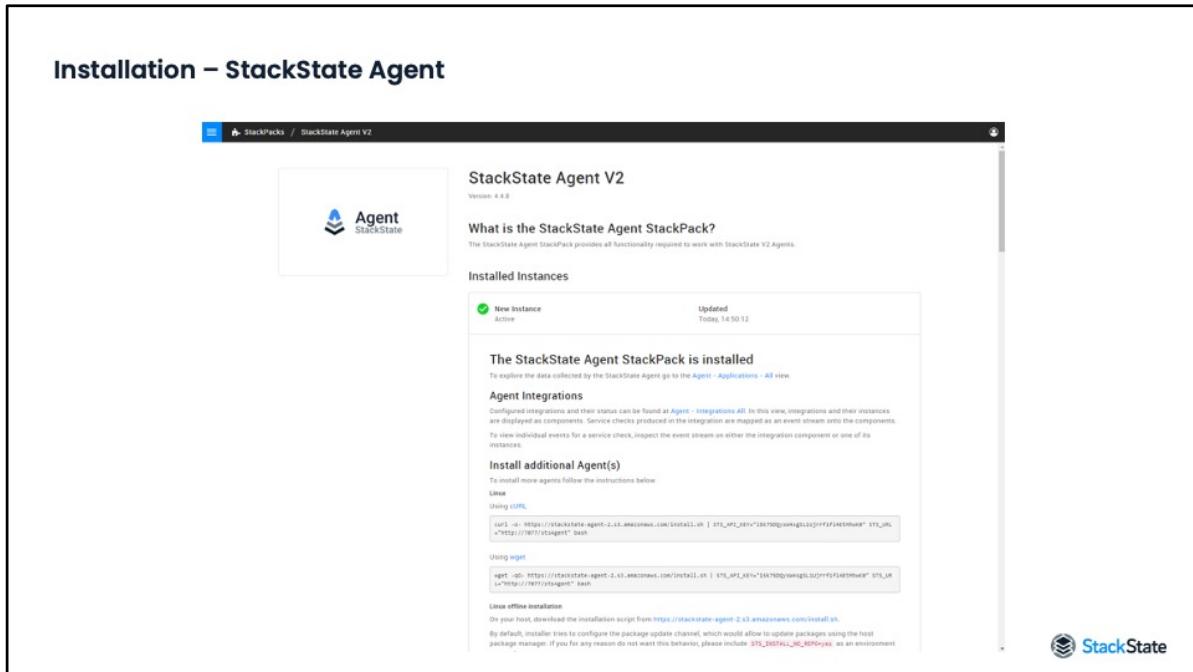
The StackState Agent is commonly used to report on a host or cluster and for integrations with data sources. The StackState Agent periodically collects data and sends it to the StackState receiver.

The StackState Agent v2 StackPack will help you to run the StackState Agent on Linux, Windows, or as Docker image. By default, the StackState Agent reports on the host it runs on. You will see components and relations based on what the StackState Agent has detected.

Configuring the StackState Agent to integrate with an external data source requires some additional steps. These steps are specific to the data source and are described on the data source specific StackPack pages as a prerequisite. Usually, enabling an integration in the StackState Agent is a matter of specifying a YAML configuration file. After the YAML file has been created or updated, the service stackstate-agent has to be restarted.

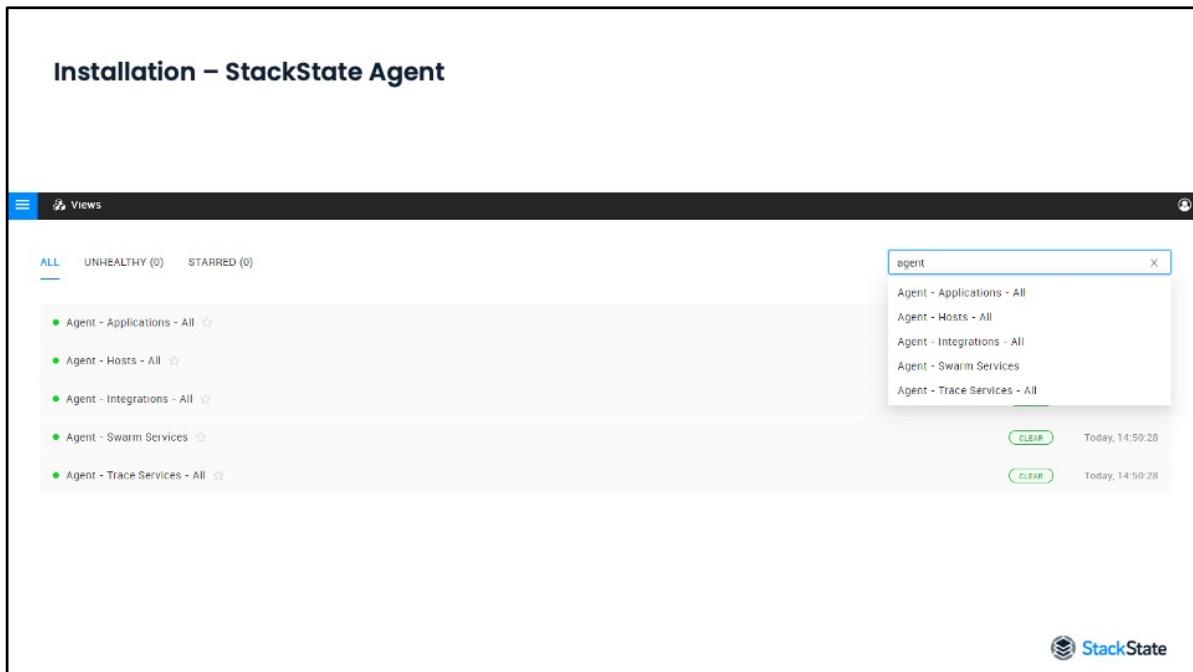
Be aware that networking requirements differ per data source connected. The StackState Agent needs to send data to the StackState

receiver on TCP port 7077 for Linux distributions and to TCP port 8080 for the stackstate-router Kubernetes service. The StackState Agent makes a connection to the data source, so the host running the Agent should be able to connect to the data source.

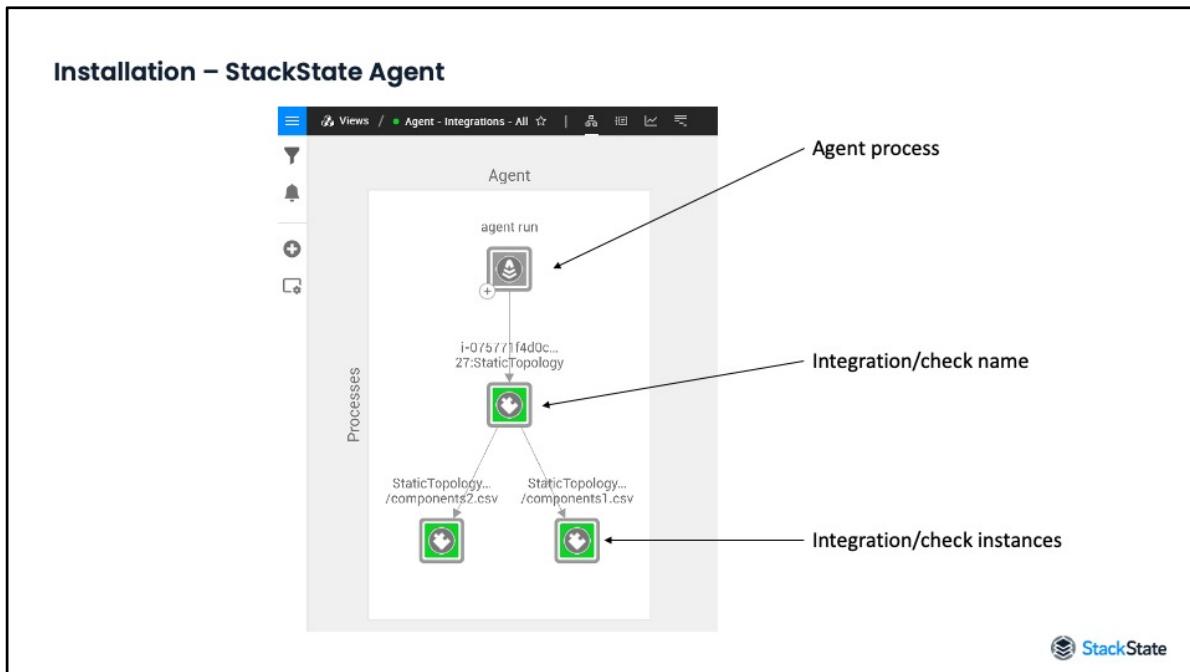


This is the StackState Agent v2 StackPack page. In this example, the StackPack is currently installed. The StackState Agent v2 StackPack only needs to be installed once to be able to receive data from one or more StackState Agents. You can go back to the StackState Agent StackPack at any time if you want to copy/paste the install command to deploy more StackState Agents.

For data source specific StackPacks that depend on the StackState Agent StackPack, you have a choice to make. You have to select one installed StackState Agent that will be responsible for data collection. It is possible to have a single StackState Agent that integrates with several data sources or one StackState Agent installed next to the data source. The choice is yours.



The StackState Agent StackPack installs several views in the StackState environment. These views show components that are reported from all StackState Agents that send data to StackState. There are views to show found hosts and applications. There is a view that reports on Docker Swarm services, if run in Swarm mode. There is a view that shows information related to traces. Lastly, there is a view that reports on Agent integrations. The integrations view lists all StackState Agent configured integrations and the status of these integrations. Let's have a look at an example.



Here is an example of the integrations view. Here you can find all integrations that have been enabled in the StackState Agent. Very useful for debugging integration related issues and for monitoring purposes.

This example shows that one StackState Agent is installed and its process is shown at the top. The component in the middle is of component type 'agent-integration' and shows the integrations that have been enabled in the StackState Agent. In this case the integration is called StaticTopology, because in this case the Static Topology StackPack has been installed.

Both components at the bottom represent the instances that are installed and are of component type 'agent-integration-instance'. In this case, there are two instances of the Static Topology StackPack installed, hence we see two components. If only one instance of any StackPack would have been installed, only one component would show.

These components are reported by the StackState Agent itself. Although there is a correlation to StackPack instances, these components reflect

what is configured on the StackState Agent and do not reflect StackPack instances. It's best to consider these components from a StackState Agent perspective. The status of a StackPack instance can be found on the StackPack pages.

Upgrading



Let's have a look at upgrading StackState.

Upgrading – StackState

- Version number format: <major>.<minor>.<maintenance>
- Minor/maintenance StackState upgrades
 - Backup
 - Upgrade StackState
 - Verify installation
 - Upgrade StackPacks if necessary
- Major StackState upgrades
 - Backup
 - Uninstall StackPacks
 - Upgrade StackState
 - Install StackPacks
 - Verify installation
- Version specific upgrade notes



StackState uses the following versioning scheme; a major version number is followed by a minor version number. There can also be maintenance releases for each minor version of StackState. Upgrading to a higher major version number is considered a major StackState upgrade and upgrading to a higher minor or maintenance version number is considered a minor upgrade.

The steps to take for a minor, or maintenance, upgrade is to create a backup of StackGraph, upgrade StackState, verify the installation, and upgrade any StackPacks if necessary. Backup options are discussed in the next slide. A full StackGraph backup is meant here, containing StackState configuration and topology. StackState upgrade for Linux you can use your package manager's upgrade command. For example, for RedHat based distributions the 'rpm --U' followed by the package's filename can be used. For Kubernetes, upgrading is the process of executing 'helm repo update' to update the Helm repository followed by the helm upgrade command. After the upgrade you can verify if the upgrade was successful by checking if all processes or pods are up and running. Don't forget to upgrade StackPacks when newer versions of a StackPack is available.

Upgrading to a higher major version of StackState follows the same steps as a minor upgrade. The difference is that for each major version of StackState, you must uninstall the installed StackPacks first before upgrading. This is required because the existing StackPack may not be compatible with the new version of StackState.

Before upgrading, don't forget to read the version specific upgrade notes on the documentation website. The version specific upgrade notes inform you about any changes that may impact your StackState environment. It provides information about possible settings that need to be adjusted before starting StackState. The upgrade may require you to adjust or create a new values.yaml, for example.

Ref: <https://docs.stackstate.com/setup/upgrade-stackstate/version-specific-upgrade-instructions>

Upgrading – backup options

- Configuration and topology data
- Telemetry data
- Configuration
- Manually created topology



StackState offers the backup options for the following; configuration and topology data, telemetry data, configuration, and manually created topology.

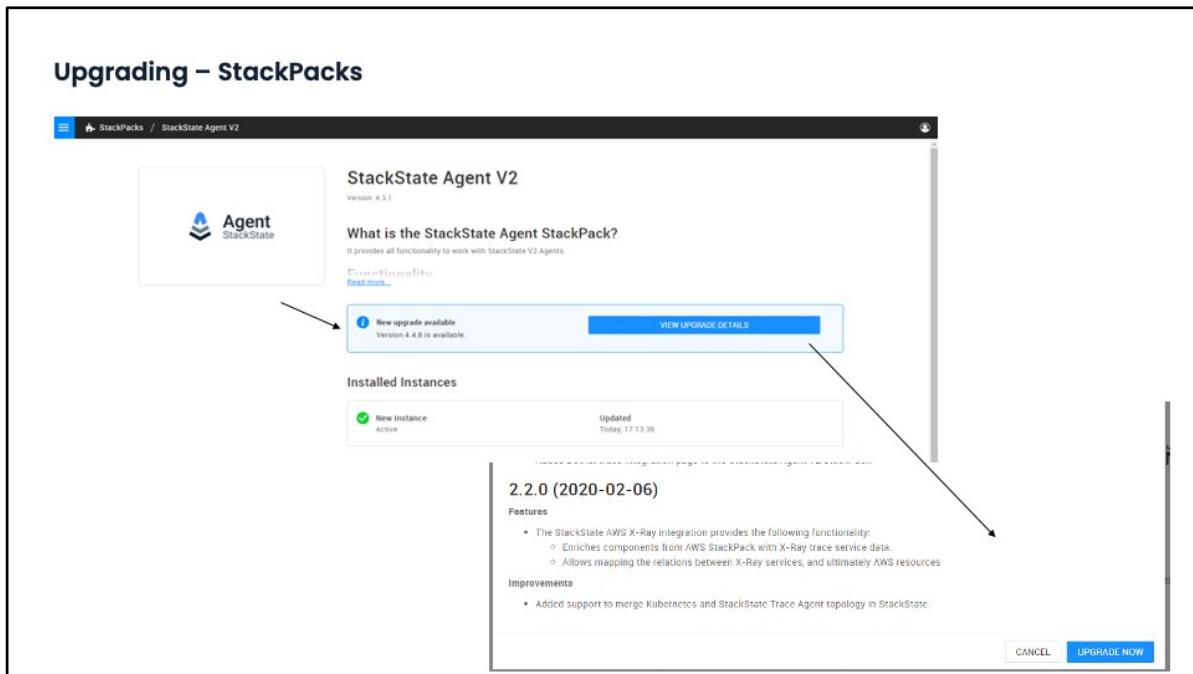
StackGraph contains configuration and topology data. A full backup can be created of StackGraph on Linux and Kubernetes. For Linux, the backup is written to disk asynchronously. For Kubernetes, you can choose from storage options AWS S3, Azure Blob Storage, or Kubernetes storage. This backup option is a full backup, there is no option for incremental backups. The backup operation for Linux is manual and for Kubernetes it is scheduled daily, by default. The resulting files have a '.graph' extension and can be moved, copied, or deleted as required. Restoring a backup is a manual action. Telemetry from Elasticsearch is not included in this backup.

Telemetry data lives in Elasticsearch and can be backed up. For Linux, you can use the default backup/restore options Elasticsearch has to offer. For Kubernetes, scheduled Elasticsearch snapshots are enabled by default. By default, Elasticsearch snapshots are created daily. For

Kubernetes, you can choose from storage options AWS S3, Azure Blob Storage, or Kubernetes storage.

StackState configuration can be exported via the StackState CLI. You can export all or a subset of configuration using the CLI. The backup can be stored to file and has the '.stj' extension. Note that StackPacks import configuration into StackState. If your environment entirely relies on installed StackPacks then a configuration backup may be superfluous.

Topology created manually can be exported and imported separately. The StackState CLI can be used to export and import manually created components and relations. Please note that importing previously exported topology information is not idempotent.



StackPacks have their own version number. The StackPack version numbers adhere to the same version scheme as StackState. You can upgrade a StackPack separately from StackState. By default, a StackPack is shipped together with StackState. You can upload a StackPack via the StackState CLI. Upgrading StackState can include a newer version of an installed StackPack. A minor or major upgrade is then possible.

The StackPack page shows whether there is a new version available. Clicking on 'view upgrade details' will show the release notes of the StackPack. For a minor version upgrade there will be a 'upgrade now' button available at the bottom of the dialog. Clicking this button will trigger the upgrade the StackPack.

A major version upgrade will not show the 'upgrade now' button. In case of a major version upgrade, you must uninstall all StackPack instances before you can use the new StackPack version. Do note that uninstalling a StackPack instance causes all topology related to that StackPack instance to be removed. Uninstalling a StackPack instance may take a

while depending on the topology information available in StackState. You can uninstall or install StackPack instances via the user interface or via the StackState CLI.

Settings pages



Let's have a look at StackState's settings pages.

Settings pages

The screenshot shows the StackState interface. On the left, the main navigation bar includes 'Explore Mode', 'Views', 'Analytics', 'CLI', 'StackPacks' (with 'Integrations' and 'Add-ons' sub-options), 'Settings' (which is highlighted in blue), and 'Help'. Below this is the version information 'v8.4.0rc0StackState'. An arrow points from the 'Settings' link in the main menu to the detailed 'Settings' page on the right.

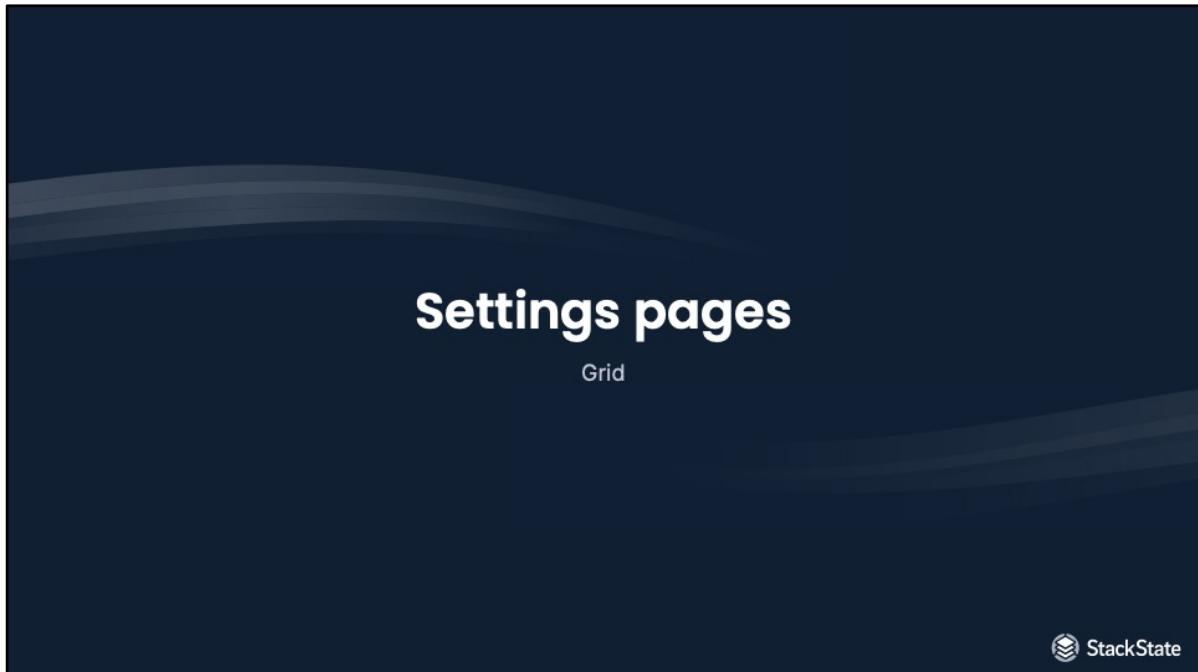
Settings

- Grid**: Describes a topology component located in a grid through one layer, one domain and at least one environment. Includes links for 'Layers', 'Environments', and 'Actions'.
- Types**: Components and relations have configurable types. Types can be used to provide specific implementations. Includes links for 'Component Types' and 'Relation Types'.
- Functions**: Components and relations have configurable functions. Functions can be used to provide specific implementations. Includes links for 'Object Functions', 'Propagation Functions', 'New Health State Configuration Functions', and 'Event Handler Functions'.
- Topology Synchronization**: Topology from multiple data sources are automatically merged into a single topology using a process called topology synchronization. Includes links for 'Sync Providers', 'Id Entity', 'Component Mapper Functions', 'Relation Mapper Functions', 'Component Templates', 'Relation Templates', and 'File Sources'.
- Telemetry Sources**: Sources for collecting data. Includes links for 'CloudWatch Metrics', 'CloudWatch Metrics', 'Logstash', 'Logstash', 'Metrics', 'Metrics', 'Metrics', 'Metrics', and 'Metrics'.
- Import/Export**: All telemetry can be imported and exported. Includes links for 'Export Telemetry' and 'Import Telemetry'.
- Actions**: Actions are scripts that can be used to show a report, change the stack interface navigation or trigger a fire and forget action. Includes a link for 'Component Actions'.
- Functions**: StackState reacts to changes in the GTL data model by configurable triggers and listeners. Includes links for 'Object Functions', 'Propagation Functions', 'New Health State Configuration Functions', and 'Event Handler Functions'.
- Telemetry Sources**: Telemetry sources can be used to add telemetry to the GTL data model. Includes links for 'File Sources', and 'File Sources'.

Please check the documentation to learn how to configure StackState. If you have questions, we're here to help. You can get in touch with our support team via the support site.

StackState

StackState's settings can be found under the main menu by clicking on settings. On the left, different sections are listed. We'll cover each of these sections in more detail. Installed StackPacks add settings to StackState and can be found in these pages.



Let's have a look at StackState's the grid section in settings pages.



The grid section defines all layers, domains, and environments that are available in StackState. A component is located in one layer and in one domain, and in one or more environments. The layer, domain, environments of a component can be found the component properties dialog.

Name	Order	Description	...
Containers (deprecated)	1.4	This function is deprecated and will be removed in a future release of StackState	...
Containers (deprecated)	1.45	This function is deprecated and will be removed in a future release of StackState	...
Containers	9000		...

The layers settings page will show all layers that are present in StackState. Each layer has a name, order, description, and a menu shown by three dots. Next to the layer's name a lock icon is shown. A lock icon illustrates that the layer is provided by a StackPack. We'll look at locking in more detail in a separate section. The layer ordering in StackState views is defined by the decimal value in the order column. The layers are ordered from top to bottom, where the top most domain has the lowest order set.

The menu offers the option to edit, delete, or export the layer. By editing the layer you're able to change the layer's name, order, and description. Delete a layer if you want to remove the layer. It may be possible that you cannot delete a layer because components still depend on it, there are still components present that can be found in the layer. You have to remove those components first before you're able to delete the layer. The last menu item is export. Exporting a layer results in a file being downloaded. The downloaded file is a StackState Template JSON (STJ) file and contains the layer's definition. STJ files can be imported again at a later moment in time. StackPacks also make use of STJ files to import

settings. The menu is quite similar for all settings you find in StackState.

Settings pages – domains

Name	Order	Description
Agent	1	

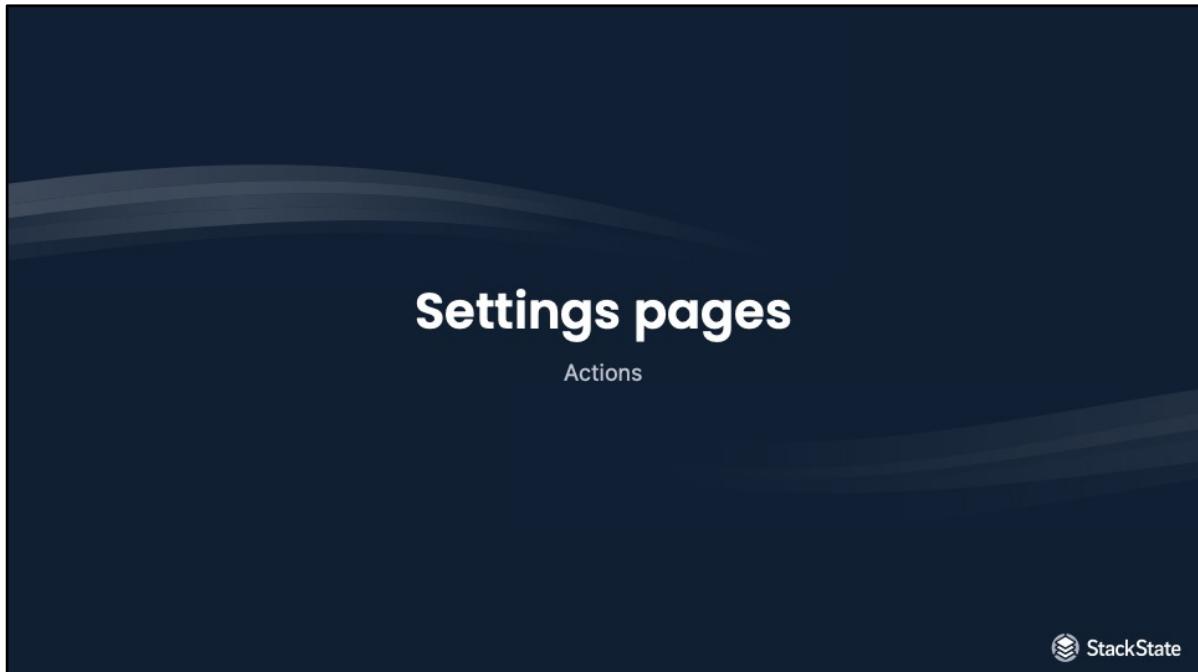
StackState

The domains settings page will show all domains that are present in StackState. The setting page for domains is very similar to the layers setting page. Similar to layers, domains have a name, ordering, and description. The domain ordering in StackState views is defined by the decimal value in the order column. The domains are ordered from left to right, where the left most domain has the lowest order set.

Settings pages – environments

The screenshot shows the 'Environments' section of the StackState Settings page. On the left, there is a sidebar with a 'Grid' section containing 'Layers', 'Domains', and 'Environments'. An arrow points from the 'Environments' link in the sidebar to the main 'Environments' table on the right. The main table has columns for 'Name' and 'Description'. A single row is visible, labeled 'Production'. To the right of the table is a context menu with options: 'Edit', 'Delete', and 'Export'. At the bottom right of the page is the StackState logo.

The environments settings page will show all environments that are present in StackState. A table is shown where each row represents a single environment. Similar to layers and domains, each environment has a name and description. An environment does not have an ordering because environments are not visualized in a StackState view.



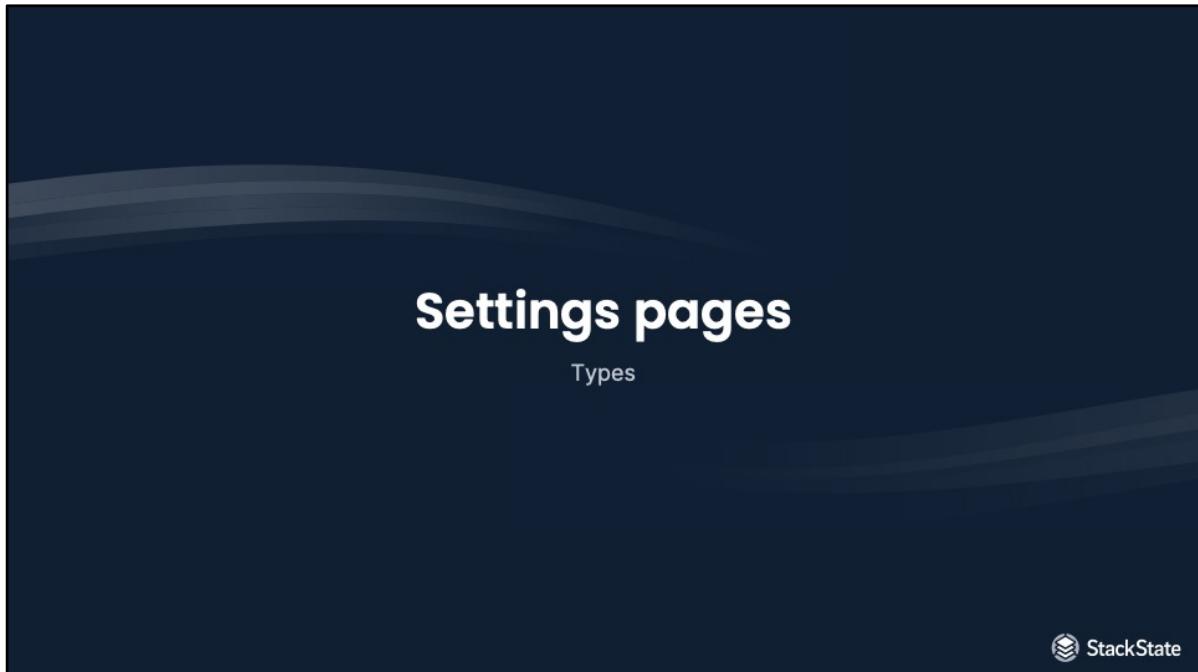
Let's have a look at StackState's the actions section in the settings page.

Settings pages – component actions

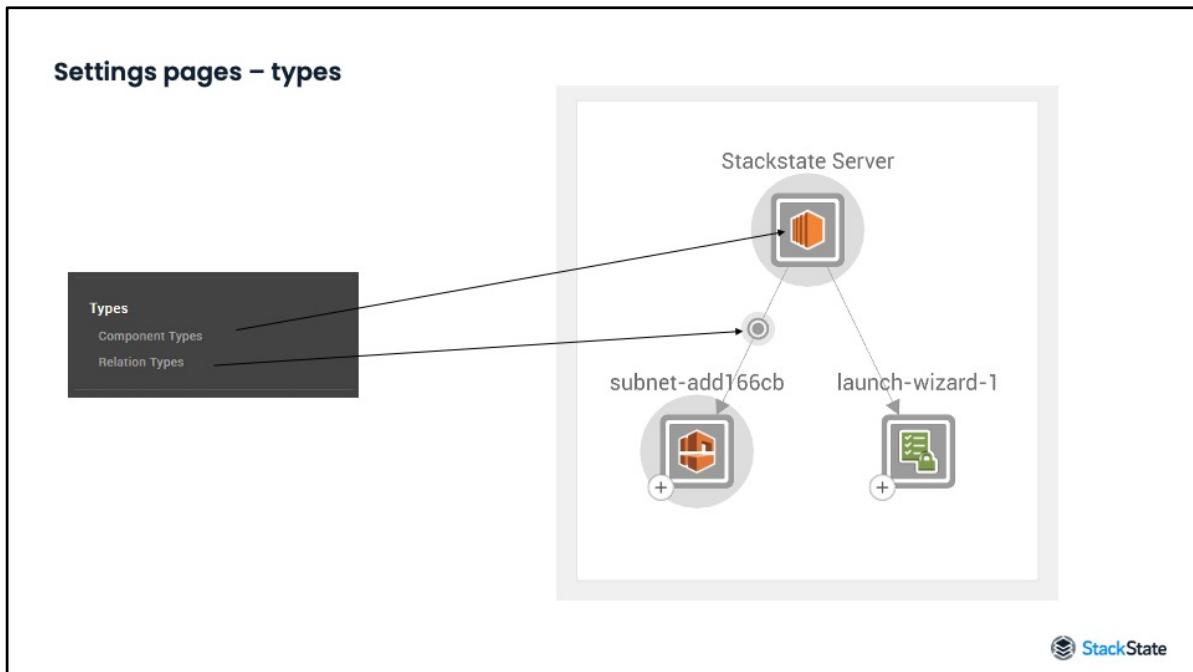
The screenshot shows two parts of the StackState interface. The top part is a 'Component Actions' settings page with a search bar, a table for managing component actions, and a sidebar with 'Actions' and 'Component Actions'. The bottom part is a 'Machines' view for an 'AWS Stackstate Server' component, showing its health status, dependencies, and a list of actions like 'Go to EC2 Console' and 'Health Forecast Report (DB)'. Arrows point from the 'Actions' and 'Component Actions' sections of the sidebar to their respective counterparts in the component details view.

The component actions settings page lists all component actions that are present in StackState. Component actions can be executed in the context of a component. For example, a component action can create, and open, a link where a part of the link contains component specific information.

A component action has a name, description, STQL query, and a script. The STQL query is used to determine which components the component action is applicable to. The component action will only be available on components where the component matched the STQL query. If you invoke a component action, the script will be executed asynchronously. The script is able to use the component's data on which it was executed on.

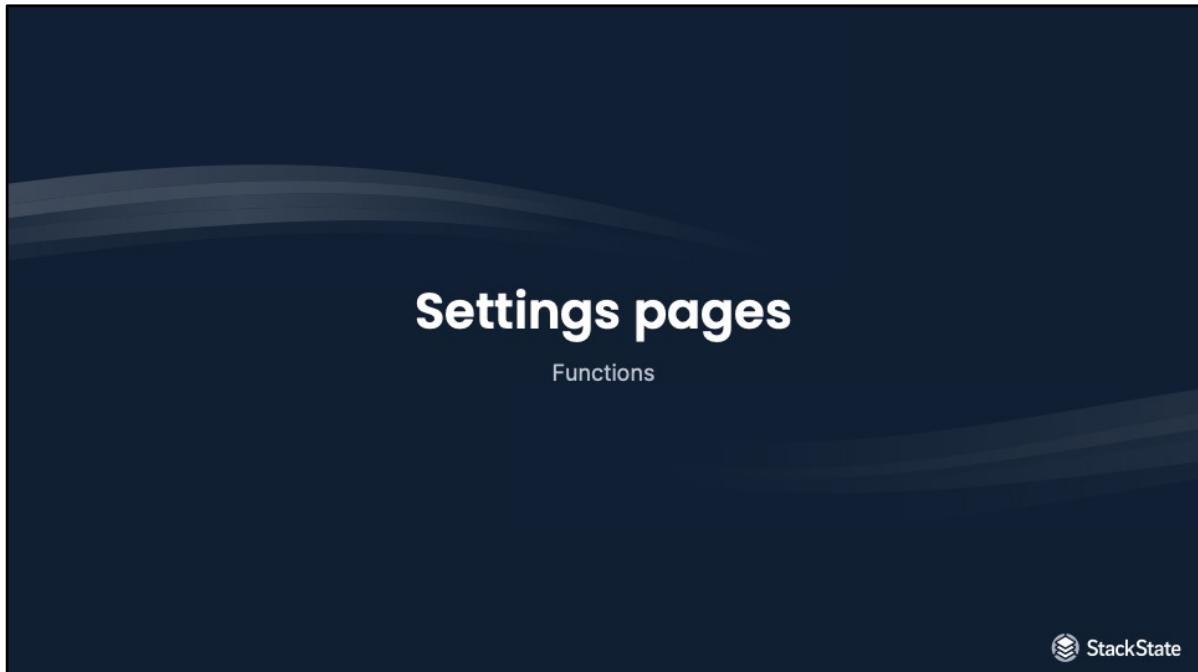


Let's have a look at StackState's the types section in the settings page.



The component types and relation types settings pages lists all component- and relation types that are present In StackState. Essentially, a component is of a certain type. A component can be a business service, a server, or something else entirely. A component type can have an icon. The icon is represented on each component of that specific type.

Similar to component types are relation types. A relation between two components is of a certain type. Relation types have a dependency direction. There are three dependency directions; one way, both directions, or none. One way relations, for short, are most common. One way relations have an arrow pointing from a source component to a target component. Both direction relations have an arrow on both sides of the relation. The none relation does not have any arrows.



Let's have a look at StackState's the functions section in the settings page.

Settings pages – functions / check functions

The screenshot shows the StackState UI for managing check functions. On the left, a sidebar titled 'Functions' has 'Check Functions' selected. The main content area is titled 'Check Functions' and contains a table with columns: Name, Description, ReturnType, and actions (Edit, Delete, Export). One row is shown with the name 'Metrics maximum last value', description 'Calculate the health state only by comparing the last value in the time window against the configured maximum values.', and returnType 'HEALTH_STATE'. Below the table is a list of telemetry streams: 'Health' (selected), 'Node Disk Space State', '/dev/root Disk Usage', 'Memory usage', and 'overlay Disk Usage'. The StackState logo is in the bottom right corner.

The check functions settings page lists all check functions that are currently available in StackState. Check functions have a name, description, parameters, return type, and a script. A check on a component, or relation, is based on one or more telemetry streams. A check executes the configured check function reactively when the underlying telemetry stream has received a new data point.

The check function's script is executed and returns a health- and/or a run state, as defined by the check function's return type. Check functions are by default stateless. There are some cases where state needs to be kept among check invocations, a memory HashMap can be leveraged to keep state between invocations.

Propagation Functions

Propagation defines how a propagated state flows from one component to the next. Propagation always flows from dependencies to dependent components and relations. Note that this is the opposite direction of the relation arrows in the graph. [Read more](#)

Name	Description	...
Active/active failover	Propagated health state is one lower than the calculated health state, i.e. CRITICAL becomes DEVIATING and DEVIATING becomes CLEAR	...
Decreasing propagated health state	Propagated health state is one higher than the calculated health state, i.e. CLEAR becomes DEVIATING and DEVIATING becomes CRITICAL	...
Increasing propagated health state	Propagates the state transparency when the running state of the component is set to RUNNING	...
Propagate when running	Quorum based cluster propagation	...
Stop propagation	Stop propagation for relation type	...

Edit **Delete** **Export**

StackState

Diagram: A grid of 9 squares representing components. The top row has three squares: UNKNOWN (red), CLEAR (green), UNKNOWN (grey). The middle row has three squares: CLEAR (green), UNKNOWN (grey), UNKNOWN (grey). The bottom row has three squares: CRITICAL (red), DEVIATING (orange), CLEAR (green).

The propagation functions settings page lists all propagation functions that are currently available in StackState. The default propagation function is not shown in this settings page but is available by default. The default propagation function set on any component in StackState is 'auto propagation'. Auto propagation determines the propagated health state by determining the maximum, or most severe, state of the component's own state and the propagated state of all dependencies. For example, the propagated health state will become red when the component is green but one of the dependencies has a red propagated health state. The same holds true when all dependencies are green and the component itself is red. Other available propagation functions are shown here.

Settings pages – functions / view health state configuration functions

View Health State Configuration Functions

Every team has a different definition of when the part of the IT landscape they are watching over is in danger. So the View health state can be used to indicate when the whole, as defined in a view, is in danger. [Read more...](#)

Name	Description
Minimum Health States	Change current view health state when the number of components with a (non-propagated) critical/deviating health state in the view is higher than the configured minCriticalHealthStates/minDeviatingHealthStates.

[Edit](#) [Delete](#) [Export](#)

Views / Agent - Hosts - All

StackState

The view health state functions settings page lists all view health state functions that are currently available in StackState. View health state functions determine the color of the view.

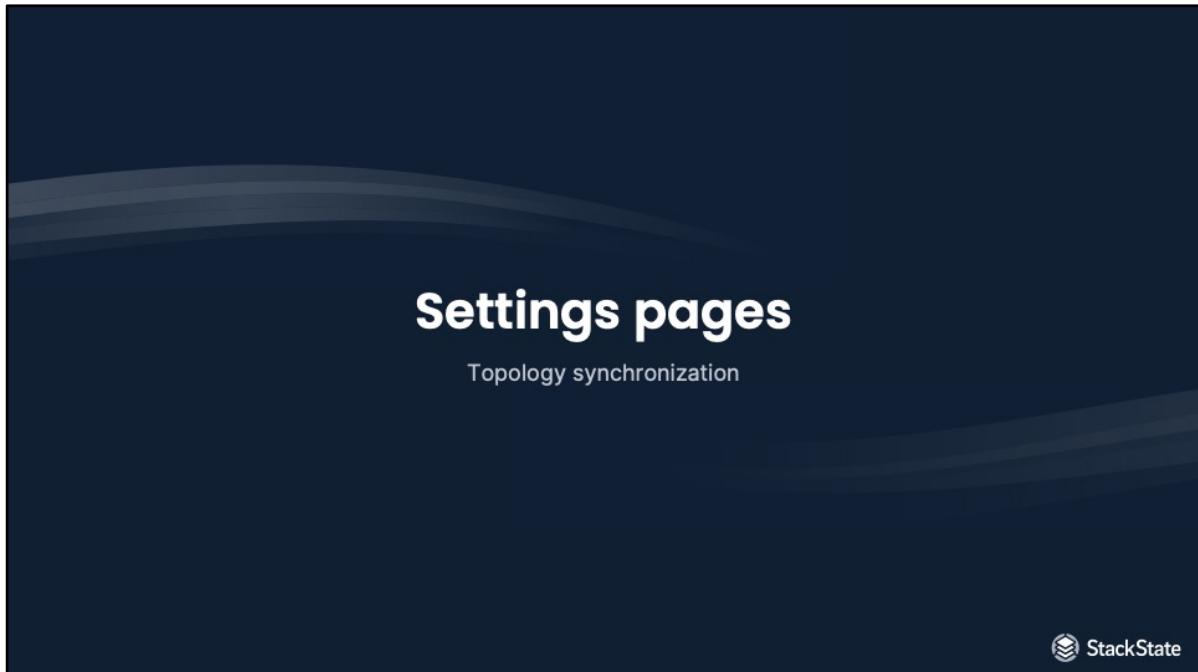
Settings pages – functions / event handler functions

Name	Description	Actions
HTTP Webhook POST request for component health changes	Send a notification message to Slack	... Edit Delete Export
HTTP Webhook POST request for view health state changes	This function is deprecated and will be removed in a future release of StackState	... Edit Delete Export
Notify via slack for component health state change (n/a Slack)	Send SMS when the view health state changes via the MessageBird SMS Gateway. Please get your authorisation-token via the MessageBird API settings	... Edit Delete Export
Notify via slack for component health state change. (deprecated)	This function is deprecated and will be removed in a future release of StackState	... Edit Delete Export
Notify via slack for view health state change. (deprecated)	This function is deprecated and will be removed in a future release of StackState	... Edit Delete Export
Send view health state changed notification via SMS (MessageBird)	Send email when the view health state changes	... Edit Delete Export
Send view health state changed notification via email.		... Edit Delete Export

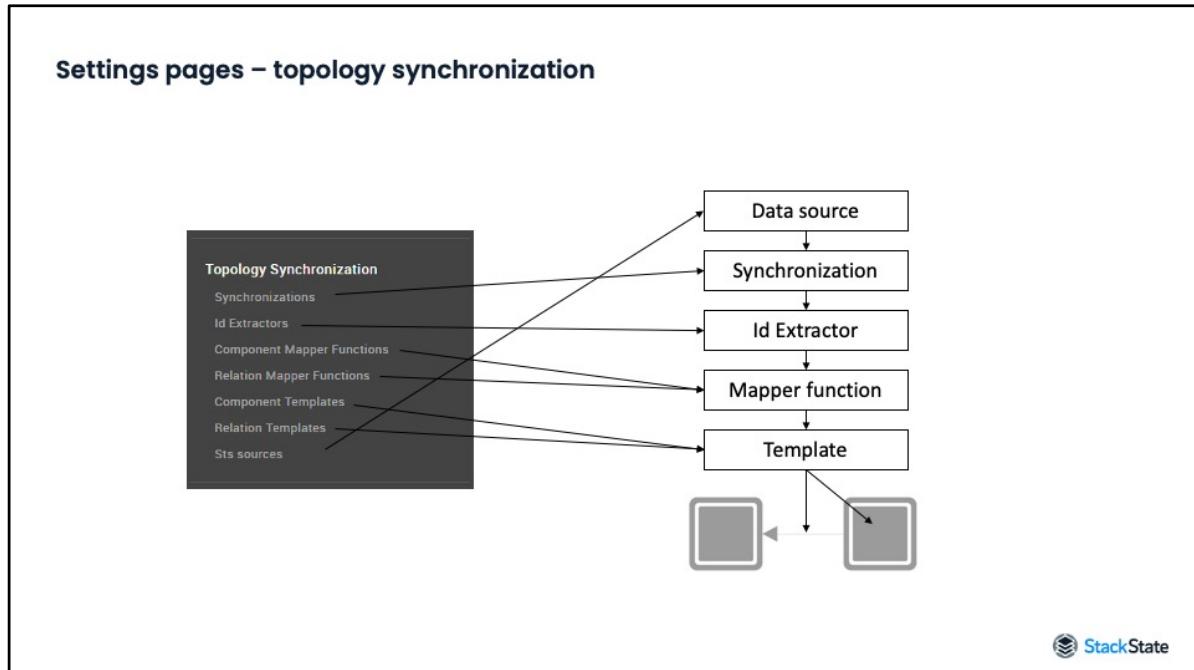
StackState

The event handler functions settings page lists all event handler functions that are currently available in StackState. Event handler functions can be used to send a message to an external system. The StackState Common StackPack comes with a number of event handler functions you can use on specific views. The Slack StackPack adds the Slack event handler.

You can set an event handler on a view via the 'manage event handlers' button in the menu on the left of the user interface.



Let's have a look at StackState's the topology synchronization section in the settings page.



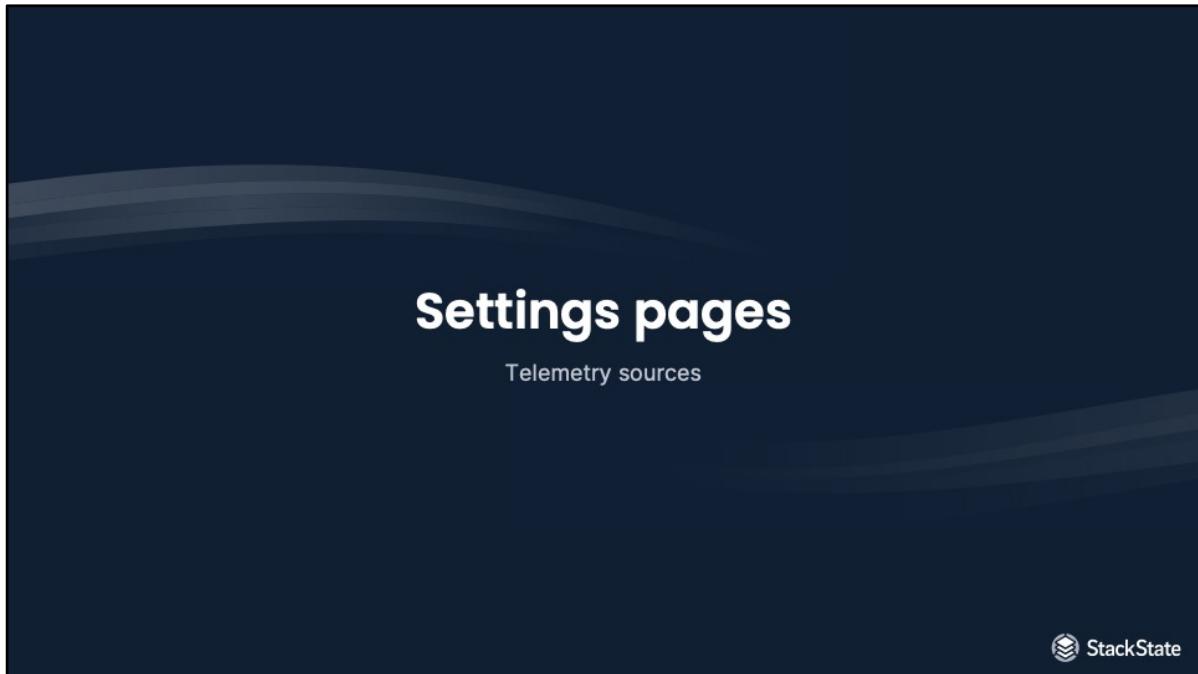
The topology section of the settings page contains the settings needed for StackState's topology synchronization process.

The data source provides StackState topology information that has to be processed by StackState's topology synchronization. The received topology information is available via a data source. The data source used for topology synchronization is called an 'Sts data source'. The Sts data source reads topology information from Kafka. The synchronization process makes use of id extractors, optional mapper functions, and templates to form topology as you see it. Synchronizations define how input data is mapped to the topology you see in the visualizer. StackState's synchronization process makes use of the other settings listed under topology synchronization. A synchronization defines what to use, and when.

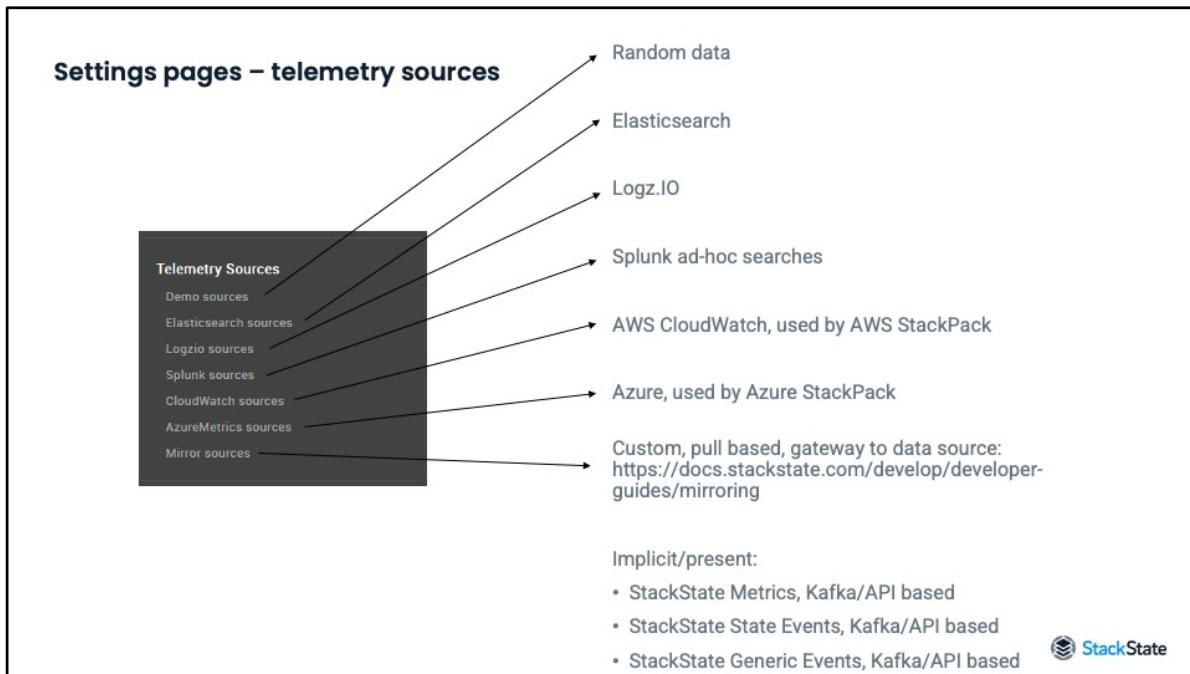
After receiving the data from a data source, synchronization executes the configured id extractor on the topology element. The id extractor is responsible for extracting a unique id, the component or relation type, and other identifiers from the input data. The unique id is used to identify

a component or relation within the synchronization such that future payloads are handled correctly. The other identifiers that can be extracted are used for merging between synchronizations. More on identifiers and merging in the section about topology synchronization. The component or relation type is used to determine which mapper function and which component or relation template to use. The component or relation materializes using the component or relation template.

We'll cover topology synchronization in more detail in the section about topology synchronization.



Let's have a look at StackState's the telemetry sources section in the settings page.



The charts you see on components or relations are backed by a telemetry data source. The telemetry data source knows how to communicate with the external data source. The telemetry data source is queried such that telemetry streams can receive telemetry data in a pull or push based fashion. The telemetry data sources are often referred to as telemetry plugins. Let's go over each of these telemetry data sources.

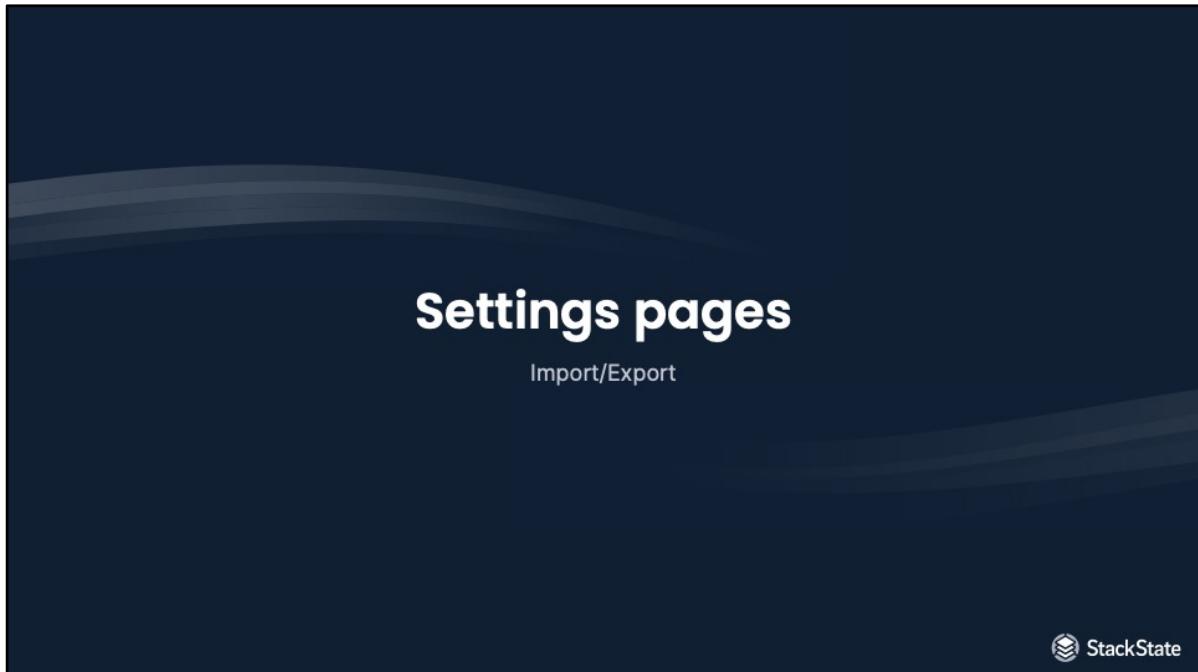
The demo telemetry data source can be used for demo data. The demo data source is a metrics only data source. Telemetry streams created using a demo data source will generate random metric charts. A very useful data source for testing a metrics based check function, for example.

The following telemetry data sources are pull based; Elasticsearch, Logz.io, Splunk, AWS CloudWatch, Azure, and mirror sources. Pull based means that the telemetry data is collected from the telemetry data source on-demand. sources. Pull based has advantages and disadvantages. One advantage is that there is no need for data replication. A disadvantage may be a potential performance impact on

the data source when there are many components and/or checks, each wanting data periodically. When you open a telemetry stream on a component or relation only then the data is retrieved from the data source. Checks require data periodically, checks will pull in data periodically. The time between each pull is defined by the “minimal live stream polling interval” of each telemetry data source.

Mirror sources is a way to connect StackState to third-party telemetry data sources. Mirror sources are a pull based method for pulling in telemetry data when the telemetry is needed. The mirror plugin talks to a mirror. The mirror is a custom webserver that implements a small REST API. The mirror will act then as a gateway between the mirror plugin and the telemetry data source. The mirror is intended to be stateless, it translates StackState telemetry requests to the third-party data source and returns data accordingly. The requests are made to the mirror when StackState requires it, pull based. StackState provides a mirror for Prometheus. You can create your own for your own telemetry data source.

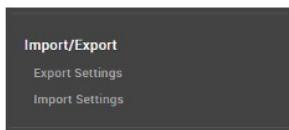
StackState has its own telemetry data sources that are implicitly present. These data sources are; StackState Metrics, StackState State Events, StackState Generic Events. These data sources are there by default and receive data that is send to the StackState receiver. These telemetry data sources are push based. The StackState receiver puts the received messages on a Kafka topic. Each of these data sources read from the Kafka topic belonging to the telemetry data source. Telemetry streams that are backed by these telemetry data sources will show new telemetry appear once they arrive. The StackState Metrics telemetry data source is used for metric data. The StackState Generic Events telemetry data source is used for log data. The StackState State Events telemetry data source is used for events that report on a certain health states. The StackState Agent reports to all three telemetry data sources. StackState Agent based integrations report to all three StackState telemetry data sources.



Let's have a look at StackState's the telemetry sources section in the settings page.

Settings pages – import/export

- Export and import StackState run-time settings.
- Backup and restore purposes
- Does not include topology nor telemetry data



Example STJ:

```
{
  "_version": "1.0.30",
  "nodes": [
    {
      "_type": "Layer",
      "id": -40,
      "identifier": "urn:stackpack:common:layer:containers",
      "name": "Containers",
      "order": 9000.0
    }
  ]
}
```



All settings in StackState can be exported and imported. Essentially, that is what StackPacks do, import settings. We've already seen that individual settings can be exported. Using the export settings functionality will download a STJ file containing all settings that is currently in StackState. Exporting settings does not include any topology nor telemetry data. The import functionality can be used to import StackState settings that are read from a local STJ file. The export functionality is useful for backup purposes and StackPack development.

The example here shows exported STJ file contains containing a version and a nodes array. There is only a single node in the nodes array. The node shows a layer's definition that was exported previously. The node contains all information needed to create a layer once it is imported.

Each node has an negative id added. This id is in certain scenarios used to form dependencies in a single STJ file. An example of this may be the link between synchronization and the Sts data source. In most cases these negative ids are not used, but present. In some cases these dependencies are handled by referring to the URN identifiers. On import,

be aware that the node definitions in the STJ do not contain duplicate negative ids. Having duplicate negative ids present may result in errors upon import.

Settings & StackPacks

Relation between them



Alright, those were the sections in the settings page. Let's have a look at the relation between the StackState settings we've just seen and StackPacks.

Settings & StackPacks

- A package that contains StackState configuration for a single integration/add-on
- Easy & dynamic configuration of new data source
- Can contain:
 - Check functions
 - Propagation functions
 - Component & relation templates
 - Synchronizations
 - ...
- SDK available

A StackPack can be seen as a package that contains StackState configuration and instructions. Upon installation, the configuration is imported and shows up in the settings pages. The StackPack page in the StackState UI shows instructions with the steps required to install the add-on or how to integrate with an external data source.

A StackPack can contain functions, templates, synchronizations, etc. Whatever is required to integrate an external data source or install the add-on. There is a software development kit available to create your own StackPack.

Settings & StackPacks

- StackPack can contain:
 - Views,
 - Layers, domains, environments,
 - Component actions
 - Component/relation types
 - Check functions
 - Propagation functions
 - View health state functions
 - Event handler functions
 - Synchronizations
 - Id extractors
 - Mapper functions
 - Templates
 - Sts data sources
 - Telemetry data sources

The diagram illustrates the StackPack structure. It shows a central box labeled 'StackPack' which contains four sub-components: 'Layer', 'Domain', 'Sync', and 'View'. The 'Sync' component is highlighted with a red border.

Each StackPack has STJ files included which hold the configuration. Since every setting from the settings pages can be exported to STJ format, it can be included in a StackPack. Upon installation of a StackPack this configuration is imported and can be seen in the settings pages. Also, views are not shown as a separate settings page, however, views can be exported and included in a StackPack.

Settings & StackPacks – identifiers

- Configuration can belong together
 - Example: configuration bundled in a StackPack
- Namespaces are formed by identifiers
- Namespaces 'group' configurations together
- Identifiers adhere to a certain scheme, Uniform Resource Name / URN
- Examples:
 - AWS StackPack: urn:stackpack:aws
 - Zabbix StackPack: urn:stackpack:zabbix
 - vSphere StackPack: urn:stackpack:vsphere



The configuration imported by a StackPack belongs together for it to function properly. StackState needs to be able to keep apart which configuration belongs to which StackPack. The configuration you find on the settings pages all have an identifier. This identifier is formatted as a uniform resource name, or URN. Essentially, configuration belongs to a certain namespace as defined by the URN identifier.

By placing configuration into the same namespace, configuration is grouped together. The identifiers adhere to a certain scheme. For example, the AWS StackPack will import its configuration into the urn:stackpack:aws namespace. Similarly, the Zabbix and vSphere StackPacks use urn:stackpack:zabbix and urn:stackpack:vsphere, respectively. Each StackPack makes use of their own namespace.

Settings & StackPacks – identifiers

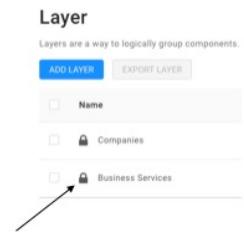
The screenshot shows the StackState interface for managing StackPacks. On the left is a sidebar with options like Grid, Layers, Domains, Environments, Actions, Types, and Functions. The main area is titled 'Edit layer' and contains fields for 'Name' (set to 'Containers'), 'Description' (with placeholder 'e.g. The virtualized servers'), 'Order' (set to '9000'), and 'Identifier' (set to 'urn:stackpack:common:layer:containers'). At the bottom right of the dialog are 'CANCEL' and 'UPDATE' buttons. In the bottom right corner of the entire screenshot is the StackState logo.

Here's an example of the layer Containers. The identifier of this layer is set to "urn:stackpack:common:layer:containers". You can read from the identifier that the namespace is "urn:stackpack:command", which tells us that the layer comes from the Common StackPack. After the StackPack name, the type of configuration is added. The last part of the URN identifier is set to the name of the configuration, in this case it is containers.

Adhering to the URN scheme, each configuration in StackState will have a unique identifier. StackPack life cycle management operates on these namespaces to create, update, or delete configuration.

Settings & StackPacks – locked/unlocked settings

- StackPack configuration is locked by default, indicated by the lock icon
- Protects against unwanted changes in configuration
- User may make changes to configuration which results in unlocked configuration
- Provides options to detect changes



 StackState

We have seen lock icons in front of configuration in the settings pages. The lock icons indicates that the configuration is locked. After installing a StackPack, the imported configuration is locked by default. The lock is used to determine whether configuration has been changed in any way.

As you can imagine, upgrading a StackPack that has configuration that is locked is much easier to upgrade than if some StackPack managed configuration is unlocked. Upgrading a StackPack will check the configuration in the namespace for unlocked configuration and will ask the user what to do in that case. You can either overwrite the unlocked configuration or to skip upgrading that configuration. Skipping will keep the local changes intact. Be aware that skipping will leave the configuration unlocked omitting the potential updated configuration that the new StackPack version might provide. Overriding the configuration might result in a loss of the manually made changes. Ideally, in a production environment we want to see only configuration that is locked. StackPacks can be deployed in deployment pipelines to facilitate this using the StackState CLI.

Settings & StackPacks – locked/unlocked configuration

Locked nodes are shown with the lock icon
 Changing locked nodes requires user's confirmation
 After changing the lock icon disappears

The screenshot shows the StackState interface for managing configuration layers. On the left, there's a list of layers: 'Name' (unchecked), 'Companies' (locked, checked), and 'Business Services' (locked, checked). A central modal window is open, stating that the layer is locked because it was created by a StackPack and asking if the user wants to proceed anyway. Arrows from the 'Companies' and 'Business Services' entries in the list point to this modal. On the right, another view of the layer list is shown, where the 'Companies' and 'Business Services' entries now have their lock icons removed, indicating they are no longer locked after the confirmation.

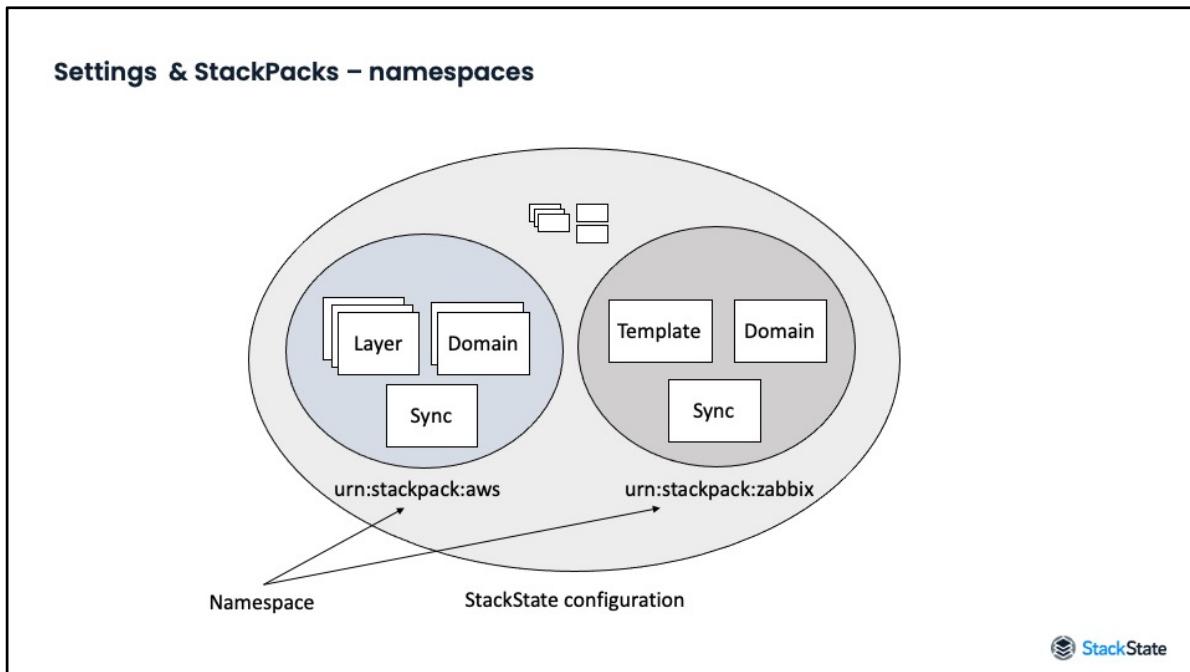
Here's an example of how unlocking configuration looks like.

In this example we unlock a layer called 'Business Service'. At the bottom left you see the lock icon in front of the business service layer. Let's say the user is going to update the layer order. After the value is changed you can click on save to persist the changes. Upon clicking save, StackState will ask for confirmation since the configuration node is locked.

After confirmation, the layer is updated and the lock is removed. Canceling results in no changes made to the configuration. The lock icon is a visible useful tool to determine whether configuration has been changed. There is also a StackState CLI command available to find configuration that is unlocked.

Do note that manually created configuration, via the settings page, does not have a lock, it is unlocked as it is not managed by a StackPack, yet. It's possible to create or edit configuration in a development or test environment before persisting the changes in your own custom

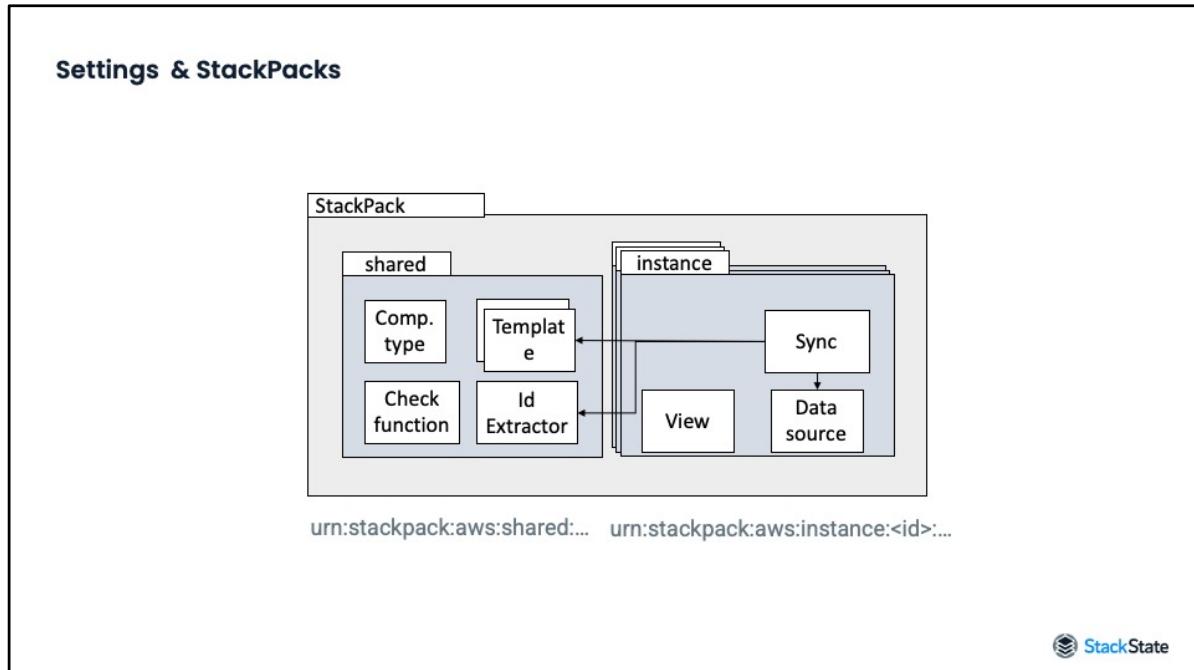
StackPack. It is not possible to lock unlocked configuration.



Here's an example visualization of the namespaces.

You have the entire configuration that lives in StackState. It's possible that some configuration does not have an identifier set, shown here on the top. Manual created configuration, for example, does not require to have an identifier in order for the functionality to work. If that configuration is going to be used in a StackPack, then an identifier is required. It is a good habit to add an identifier in new work.

StackPack managed configuration is stored in namespaces. The example here shows grouped together configuration for the AWS and Zabbix StackPacks. The namespaces are shown here. The URN identifier of the configurations contain the namespace. Of course, this illustration is not exhaustive in the number of configuration that can be present in a namespace, there is more configuration present like component types, views, etc. in each StackPack. The illustration shows that configuration can live together as a whole in StackState while parts are managed by StackPacks.



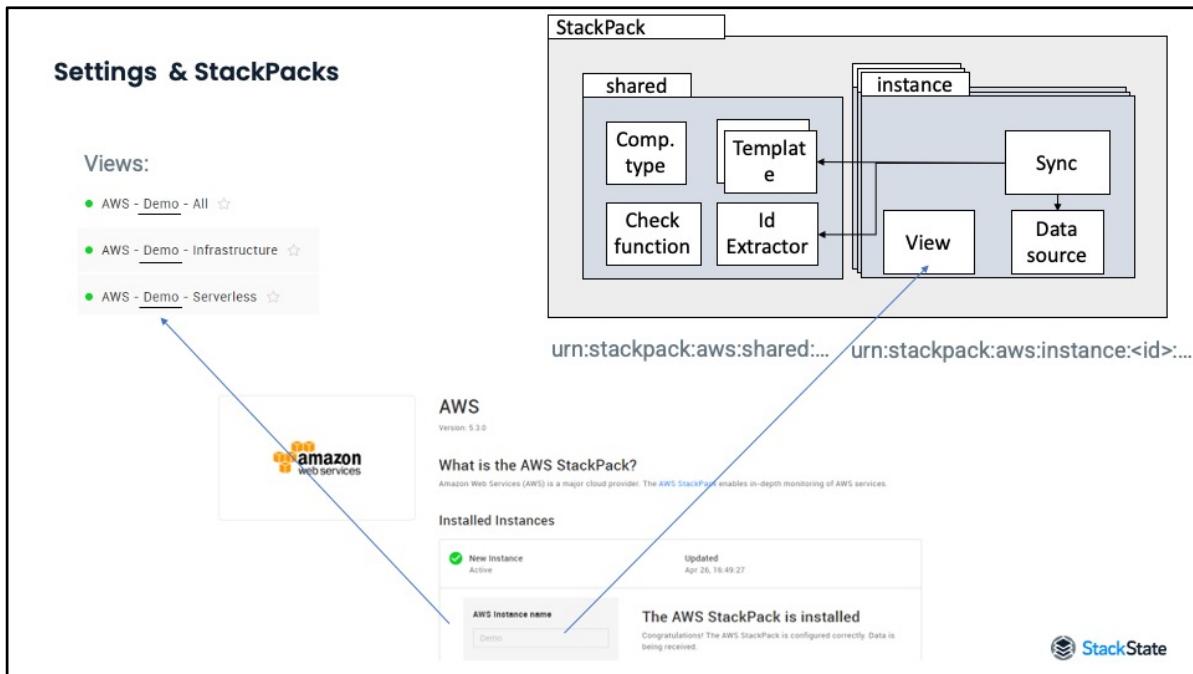
We have talked about configuration, URN identifiers, and namespaces. The majority of StackPacks support multiple instances. Having multiple instances of a single StackPack, what about duplication of configuration upon import you might say?

We put configuration that is reused among StackPack instances in a shared namespace. This requires the configuration to be imported only once and reused multiple times, as needed by each instance. An example is that of component types. There is no need to import a component type per StackPack instance. Component types are imported in a shared namespace and each StackPack instance imported configuration can refer to the shared configuration. Instance specific configuration is usually generated by the StackPack during installation and depends on the user input. This configuration is imported in the instance namespace. Each instance has its own, autogenerated, id, and is added to the URN identifier. The StackPack instance id separates StackPack instance configuration from another instance of the same StackPack.

There are dependencies between configuration. Either dependencies

between instance and shared configuration or shared to shared configuration. Shared configuration cannot have a dependency on instance configuration. For example, a component template has a dependency on a component type. The component template refers to the component type using the URN identifier of the component type. This is similar for all other dependencies between configurations, references are based on configuration's URN identifier. A synchronization has many dependencies, it refers to the URN identifiers of an Sts data source, id extractors, mapper functions, and templates.

Shared configuration is imported in a shared namespace and StackPack instance specific configuration is imported in an instance specific namespace. Essentially, can configuration be shared? If yes, then it goes to the shared namespace for reuse. If not, then the configuration should be imported in an instance specific namespace. An Sts data source is usually placed in instance namespaces since the underlying Kafka topic is different per external data source. Since the Synchronization is dependent on the Sts data source, the synchronization is also included in the instance namespace. Id extractors, templates, etc. are usually reusable between StackPack instances and are put in the shared namespace of a StackPack.

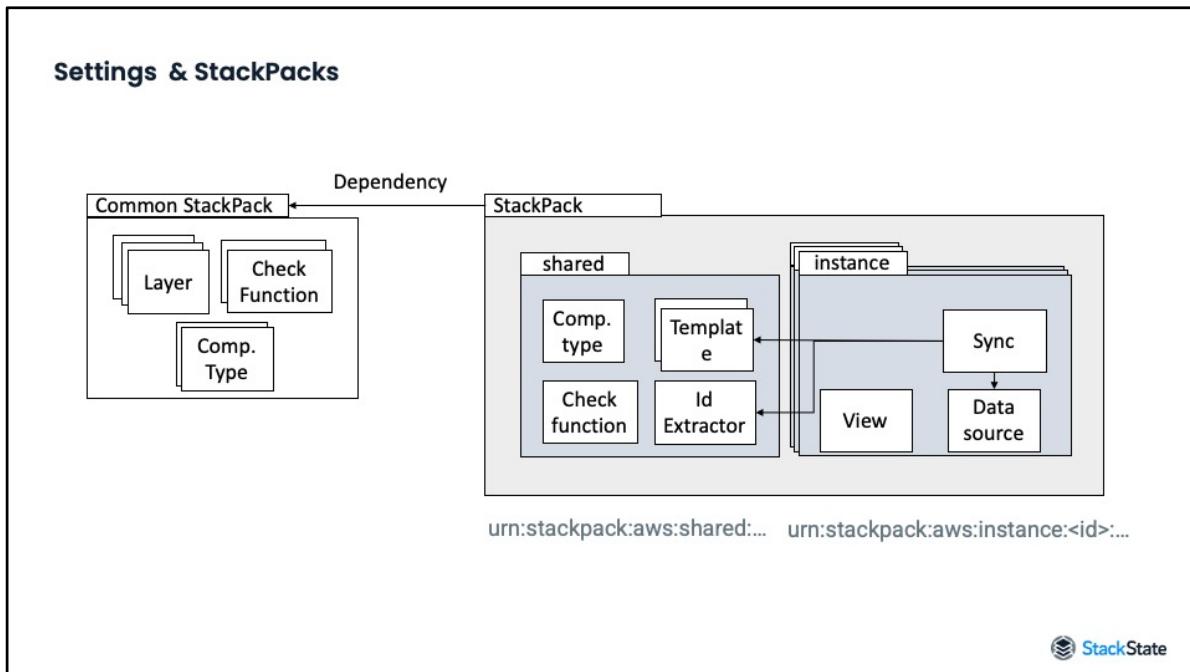


The majority of StackPacks asks the user for some information before installation. This input can be used to automatically make changes to the configuration before it is imported to the StackPack namespace.

Importing a view is a good example to illustrate the difference. Some StackPacks ask the user to provide an ‘instance name’. An instance name is a user recognizable name that is part of, for example, the name of a view. During installation, the user provides an instance name. The provided instance name would become part of the view’s name, in this example. Upon installation, StackPack takes the view definition and dynamically updates the definition to include the provided instance name before importing the view in StackState. In this case, the view must be imported into the instance namespace. This way, each installed StackPack instance will create a view including the ‘instance name’ in the view’s name. If the view were to be imported in the shared namespace you would end up in a situation where only the view remains of the last installed StackPack instance since the URN identifier is the same.

Importing the view in the instance namespace adds the instance id to the URN identifier making each view’s URN identifier unique. Importing a view into the shared namespace would have been no problem in the case

when the view's name is static. Inversely, importing a statically named view in an instance specific namespace would result in having duplicate views in StackState, one view per installed StackPack instance.



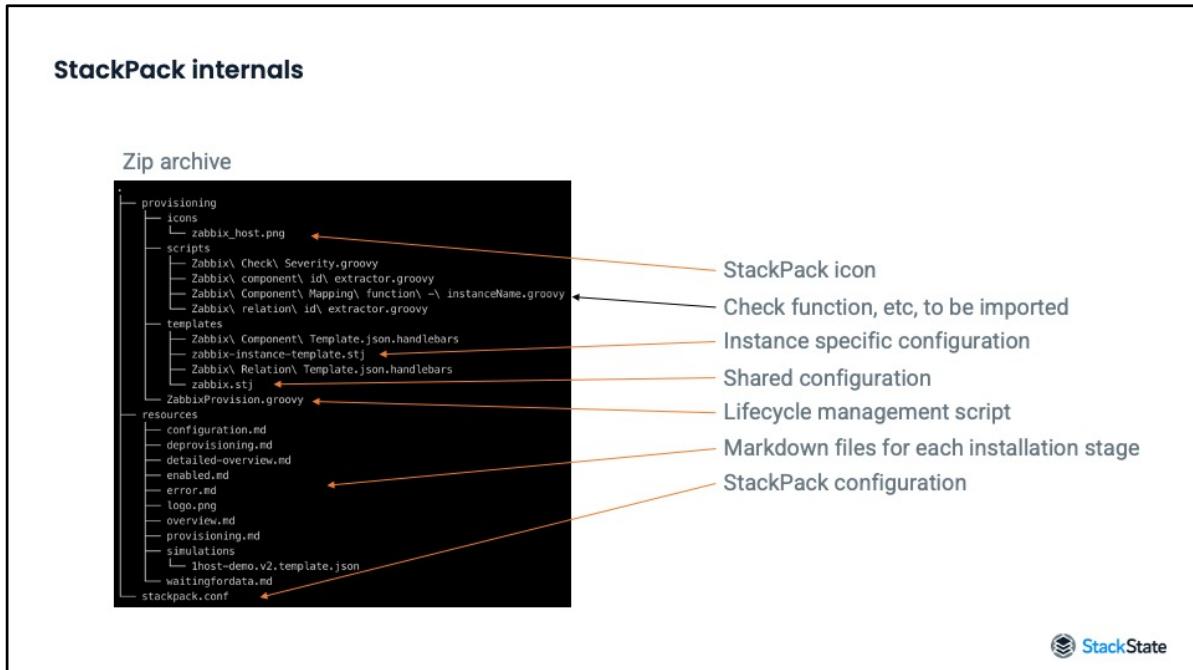
Lastly, StackPacks can have dependencies on other StackPacks. StackPack configuration may refer to common configuration that is shipped in another StackPack. Multiple StackPacks may reuse common configuration to avoid the need for duplicates. If no explicit dependency, then you may run into run time problems when the common StackPack was not installed.

For example, StackState ships a default set of component types in the StackState Common StackPack. Other StackPacks may refer to these common component types such that StackPacks don't have to import component types themselves. Upon installation of a StackPack, the StackPack it depends on is automatically installed if it was not already installed. Dependencies can only be made to StackPacks that do not require user input, like the StackState Common StackPack. These common StackPacks are very useful to reuse configuration among multiple StackPacks.

StackPack internals



Let's have a brief look inside a StackPack.



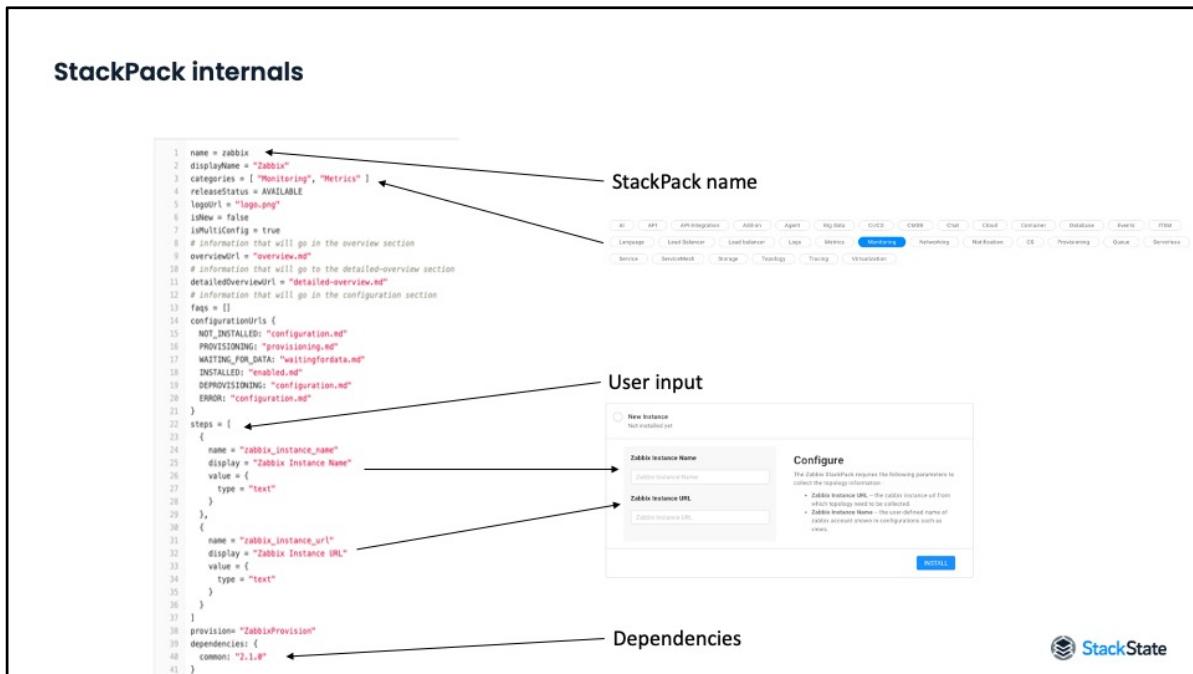
A StackPack file has the .sts extension and is essentially a zip archive containing several files and directories. The contents of the Zabbix StackPack are shown here. The contents of this StackPack is similar for all StackPacks.

At the root level, there is a directory called provisioning, a directory called resources, and a file called stackpack.conf. The stackpack.conf file contains the definition of the StackPack. It contains the StackPack's name, it defines the expected user input, etc. We'll have a more detailed look into this file on the next slide.

The provisioning directory contains STJ and support files that can be imported into StackState upon installation of a StackPack instance. There is one Groovy file present in this directory. The groovy file contains code that is invoked by StackState upon installation, upgrade, or uninstallation of the StackPack. Essentially, the groovy code defines what happens in each lifecycle of the StackPack. The majority of StackPacks import shared and instance specific configuration during installation of the StackPack. Shared and instance configuration is stored in the STJ files

located in this directory. In this case there are also some handlebars files present in the templates directory. These files are included in the STJ files to keep templates, etc., more readable in the StackPack. The same holds true for the scripts and icons directory contents.

The resources directory stores Markdown formatted text files. The contents of the Markdown files are shown on the StackPack pages in the user interface. The StackPack page as you see it in the user interface is entirely constructed out of Markdown formatted text files.



The stackpack.conf file contents of the Zabbix StackPack is shown here. The stackpack.conf format is the same for all StackPacks.

We'll go over the most important fields. The name field defines the StackPack's internal name. The displayName field defines the name of the StackPack as you see it in StackState. The categories are defined in which the StackPack can be found on the StackPack page. There are quite some references to the Markdown files located in the 'resources' directory.

The steps key defines the user input. The majority of StackPacks require user input as shown on the StackPack page. Each required input has a corresponding definition in the steps field. After the user provides the information and clicks on install, the information is used in the StackPack to alter the to be imported configuration before the configuration is actually imported.

The provision field defines the Groovy file to use for the StackPack's lifecycle management.

The dependencies field defines the StackPack's dependencies on other StackPacks. It also includes a version of the dependency. The dependency is automatically installed, if it is not installed already, before the StackPack is installed. The version defined here acts as a minimal version dependency. Major version differences will be rejected.

Check functions



Let's have a look at check functions.

Check functions

- Logic to determine the health and/or run state of a component/relation
- Groovy script
- Different functions/scripts can be used each with their own parameters
- Reactive to input, one or more metric or log stream(s)
- Health synchronization
 - Data source determines states

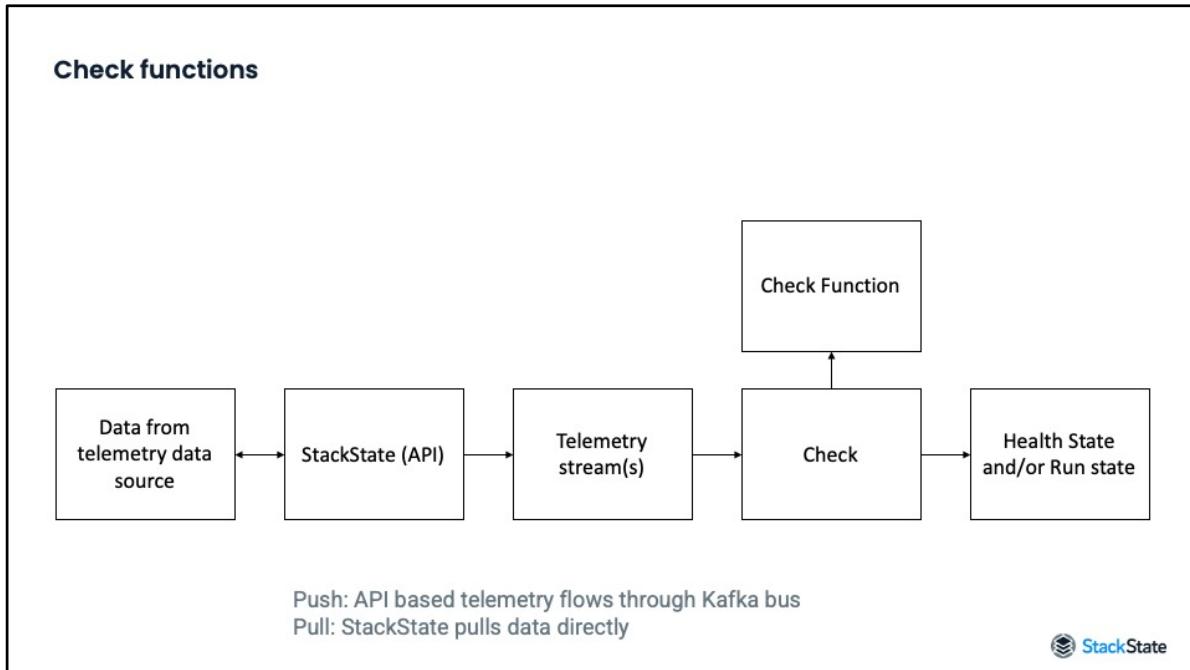


A check function contains logic to determine a health and/or run state of a component or relation. The logic is coded as a script, using the Groovy scripting language. A check function can use parameters, based on the inputs the logic can decide the output. One of these parameters is either a log or metric stream. A check function requires at least one log or metric stream.

Since you can have more than one parameter, you can have check functions that operate on more than one telemetry streams. An example of a check function using more telemetry streams is where one metric stream reports a value representing an upper limit and another metric stream reports the actual value. The check function decides whether the actual value is within acceptable bounds compared to the upper limit. Other parameters may be set to define where the bounds lie. One parameter may be used to specify when the check function should return the orange color. Similarly, another parameter can specify when the check function should return the red color. Such generic check function offers flexibility and supports multiple use cases.

The check function is reactive to the input it receives from a telemetry stream. The check function's script is executed only if the underlying telemetry stream receives new data.

A check function is different from health synchronization. If you have a data source that already determines and stores health states, then you should use health synchronization. It is easier to synchronize the health states from the data source than to have a check function to translate payloads on a telemetry stream. Check functions are useful when you have telemetry streams on which you want to derive a health and/or run state from. Of course, you can have telemetry streams that are not used by any check function.



Check functions operate on telemetry streams. These telemetry streams are getting the data from the data sources. The telemetry data is either push or pull based. An integration is pushing telemetry data to the StackState receiver or a telemetry plugin is pulling in data on-demand.

A check is defined on one or more topology elements either via topology synchronization or added manually. A check uses a check function and at least one parameter specifying the telemetry stream to use. The telemetry stream is also available on the same topology element. Additional parameters may be required depending on the check function.

When data arrives on the telemetry stream, the check invokes the check function's script. The script is executed and the logic determines the return value. The check may either return a health and/or run state. The check function also defines the optional short or detailed message.

Check functions

Log stream



Let's have a look at an example check function that uses a log stream.

Check functions – log stream

Edit check function

Name *: Zabbix Check Severity
Description: Check the severity of Zabbix host

Parameters:

- User: events (Event stream), Required: On
- Off

Returns health state
Returns run state
Script *:

```

1 def event = events[events.size() - 1]
2 def severityOpt = event.getParameter("severity")
3 if(severityOpt == null) {
4     getStruct("empty").getString("severity")
5 }
6 if (severity == "0" || severity == "1") return CLEAR
7 else if (severity == "2" || severity == "3") return DEVIATING
8 else if (severity == "4" || severity == "5") return CRITICAL
9
10 def triggermsg = event.getParameter("triggermsg")
11 if(triggermsg == null) {
12     getStruct("empty").getString("triggermsg")
13 } else {
14     "No Problems"
15 }
16 def msg = triggermsg.tokenize(",").join(",")
17
18 return [
19     healthState: severityOpt,
20     detailedMessage: msg,
21     shortMessage: "Click for more details"
22 ]

```

Identifier: um.stackpack.zabbix.shared.check-function.zabbix-check-severity

StackState

Here's an example of a check that uses the uses a log stream. In the component or relation details pane, the check function shows the charts of all telemetry streams it requires. Next to the charts, the check function's name is shown along with any parameters it requires. In this example, the check function's name is "Zabbix Check Severity" and requires no additional parameters set, as can be seen on the left.

The check function is shown on the right. The check function requires one parameter, the log telemetry stream. The parameter's name is 'events' in this case. The Groovy code is shown at the bottom. In the code, the parameter events is referenced. Upon each entry received from the log stream, the code is invoked. The code can use the entry's values to determine the resulting health state. In this case, the check function returns a health state.

Check functions – log stream

Check function:

```

1 def event = events[events.size() - 1]
2 def severityOpt = event.point
3           .getStructOrEmpty("tags")
4           .getString("severity")
5           .map{ severity ->
6     if (severity == "0" || severity == "1") return CLEAR
7     else if (severity == "2" || severity == "3") return DEVIATING
8     else if (severity == "4" || severity == "5") return CRITICAL
9   }
10 def triggermsg = event.point
11           .getStructOrEmpty("tags")
12           .getString("triggers")
13           .orElse("No Problems")
14
15 def msg = triggermsg.tokenize(',').join(" ")
16 return [
17   healthState: severityOpt,
18   detailedMessage: msg,
19   shortMessage: "Click for more details"
20 ]

```

Stream's log entry:

Timestamp	Value
Aug 18, 2020, 11:10:12.853	<pre> { "apiKey": "none", "eventType": "Zabbix", "host": "docker-desktop", "labels": {}, "message": "none", "Zabbix tags": { "host": "MCARE-M-SFC002", "host_id": "10006", "host_name": "Zabbix server", "severity": "3", "source_type_none": "Zabbix", "triggers": "[My Trigger fired]" }, "timeReceived": 1597741811856, "timestamp": 1597741812853, "title": "" } </pre>

StackState

Here's a more detailed look at the check function's code, as shown on the left. On the right an example log stream entry is shown. The check function receives this log stream entry.

Line 1 retrieves the entry from the log stream parameter called 'events', as we've seen on the previous slide. Since the log stream entries have a certain format, the code can retrieve the values needed. Line 2 retrieves the tags payload. From the 'tags' payload, the value from key 'severity' is retrieved. In this example, the severity is set to value "3".

Lines 6 until 7 translates the retrieved severity value into a StackState health state based on some logic. Line 16 defines the return value of the check function. The health state is returned along with a detailed and short message. The detailed message is dynamically constructed on lines 10 until 13 based on the log stream entry.

Check functions – log stream

Check function:

```

1 def event = events[events.size() - 1]
2 def severityOpt = event.point
3         .getStructOrDefault("tags")
4         .getString("severity")
5         .map{ severity ->
6             if (severity == "0" || severity == "1") return CLEAR
7             else if (severity == "2" || severity == "3") return DEVIATING
8             else if (severity == "4" || severity == "5") return CRITICAL
9         }
10    def triggermsg = event.point
11        .getStructOrDefault("tags")
12        .getString("triggers")
13        .orElse("No Problems")
14
15    def msg = triggermsg.tokenize(',[]).join(", ")
16    return [
17        healthState: severityOpt,
18        detailedMessage: msg,
19        shortMessage: "Click for more details"
20    ]

```

zabbix host status check

Health check is DEVIATING

6 days ago

Click for more details

Zabbix Host EventStream

1.0

0.8

0.6

0.4

0.2

0.0

10:11

10:31

10:51

11:11

fm Zabbix Check Severity

amazing live stream

Click for more details

My Trigger fired

OK

StackState

The result of the example's log stream entry processed by the check function is shown here. Input severity value "3" translated to the orange, or deviating, health state value. The resulting short and detailed messages are shown on the right.

Check functions

Metric stream



Let's have a look at an example check function that uses a metric stream.

Check functions – metric stream

Edit check function

Name*: Metrics maximum last value

Description: Calculate the health state only by comparing the last value in the time window against the configured maximum value.

Parameters:

- criticalValue: float, Required: On, Type: Float
- deviatingValue: float, Required: On, Type: Float
- metrics: Metric stream, Required: On, Type: Metric stream

User:

- On: On
- Off: Off

Script*:

```

1 IF (Metrics[-1].point >= c(criticalValue)) return CRITICAL;
2 IF (Metrics[-1].point >= d(deviatingValue)) return DEVIATING;
3 return CLEAR;

```

Here's an example of a check that uses the uses a metric stream. In the component or relation details pane, the check function shows the charts of all telemetry streams it requires. Next to the charts, the check function's name is shown along with any parameters it requires. In this example, the check function's name is "Metrics maximum last value" and has two additional parameters set, as can be seen on the left.

The check function is shown on the right. The check function requires three parameter in total; the metrics telemetry stream, a critical value, and a deviating value. The parameter for the metric telemetry stream is called 'metrics'. The check function determines based on the received metric which health state to return. If the received metric's value is above the deviating value set, then the deviating health state is returned. Similarly for the critical value, the critical health state is returned when the metric value is higher than the critical value set.

Check functions – metric stream

```

1 |
2 if (metrics[-1].point >= criticalValue) return CRITICAL;
3 if (metrics[-1].point >= deviatingValue) return DEVIATING;
4 return CLEAR;
5

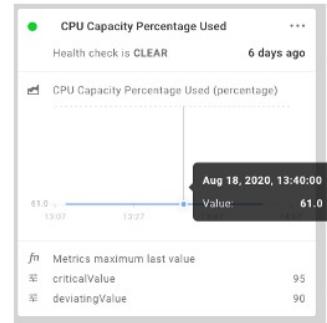
```

Translation:

```

If (61 >= 95) return CRITICAL;
If (61 >= 90) return DEVIATING;
return CLEAR

```



StackState

Here's a metric value to illustrate the metric stream-based check function. As you can see on the right, in the chart, the metric value is 61.

The check function's code is shown on the left. The value is retrieved and compared against the critical and deviating values provided to determine the returned health state.

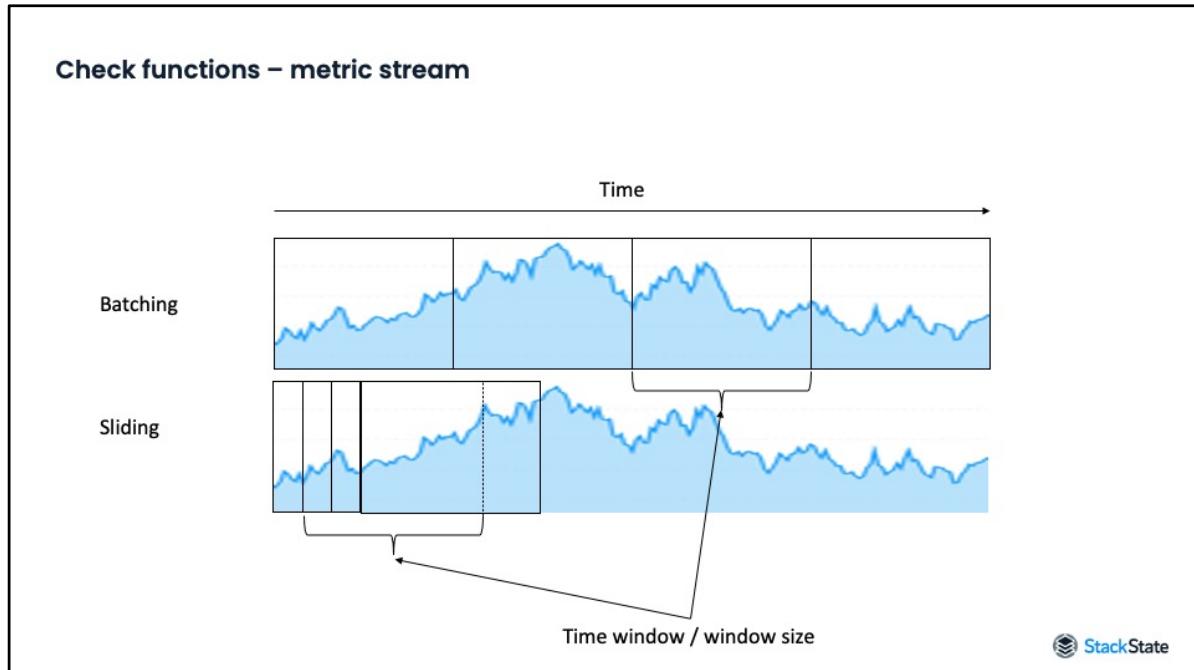
In this example, the metric value is 61. The value for critical is set to 95 and the value for deviating is set to 90. Since value 61 is lower than either 90 or 95, the CLEAR health state is returned by this check for this metric value.

Check functions – metric stream

The screenshot illustrates the configuration of a check function for a metric stream. On the left, a component details pane shows a check named 'CPU Capacity Percentage Used' with a status of 'OK'. In the center, an 'Edit check' dialog box is open, displaying the configuration for this specific check. The 'Check function' is set to 'METRICS MAXIMUM LAST VALUE', with arguments for 'criticalValue' (95) and 'deviatingValue' (90). To the right of the dialog, a 'Windowing method' dropdown menu is shown, with 'Sliding' selected. The StackState logo is visible in the bottom right corner.

Here's the full definition of the check. You can see the definition by editing the check in the component or relation details pane.

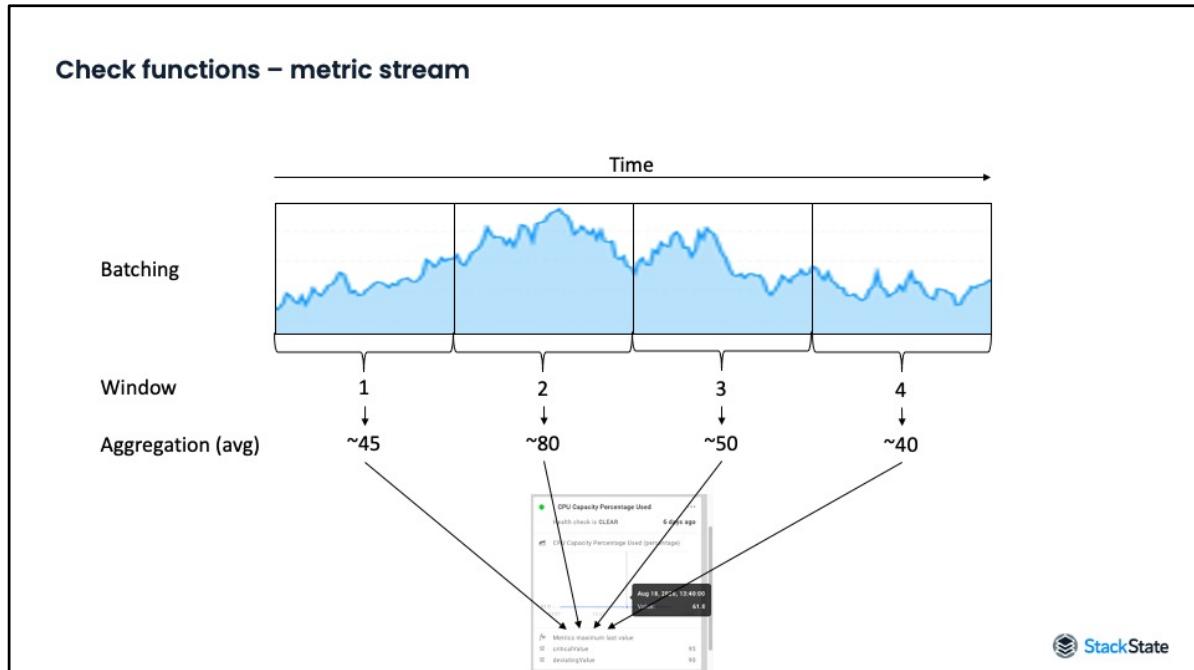
The set check function and arguments are shown here. There is also a windowing method, an aggregation method, and time window defined. The windowing method defines how to handle the metric values, to either process metrics in batches or to slide over values. These values are then processed by a set aggregation method. The aggregation method is set on the telemetry stream and is shown here as reference information. Depending on data sources, either batching and/or sliding windowing methods are supported. The time window is a boundary determining which metrics are to be included in the window used for aggregation. The window containing one or more metrics are processed by the aggregation method. The resulting value is passed to the check function for further processing.



The difference between the batching and sliding windowing methods is how the received metrics are used.

The batching windowing method closes the window when the time window has exceeded. The collected metrics in this window are processed further. A new window is started that will contain the metrics received after the previous window was closed. It essentially batches metrics based on time. The sliding windowing method slides forwards a window once new metrics are received. The older metrics are dropped, and new metrics are added to the window. Essentially, the window is sliding forward.

The timestamp of received metrics are used to determine when a time window is full. It is possible that some windows are longer than the time window specified since it needs metrics to determine whether the time window has exceeded.



Here's an example to illustrate the windowing method. A purely fictional metric stream is shown at the top together with batched windows. Four windows are shown, and the aggregation method is set to average. The fictional average is shown as well. The average is calculated and passed on to a check function.

In this example, the average of the first window is calculated to approximately the value 45. This value is passed on to the check function to determine the health state. The check returns the health state for each window over time.

Topology synchronization



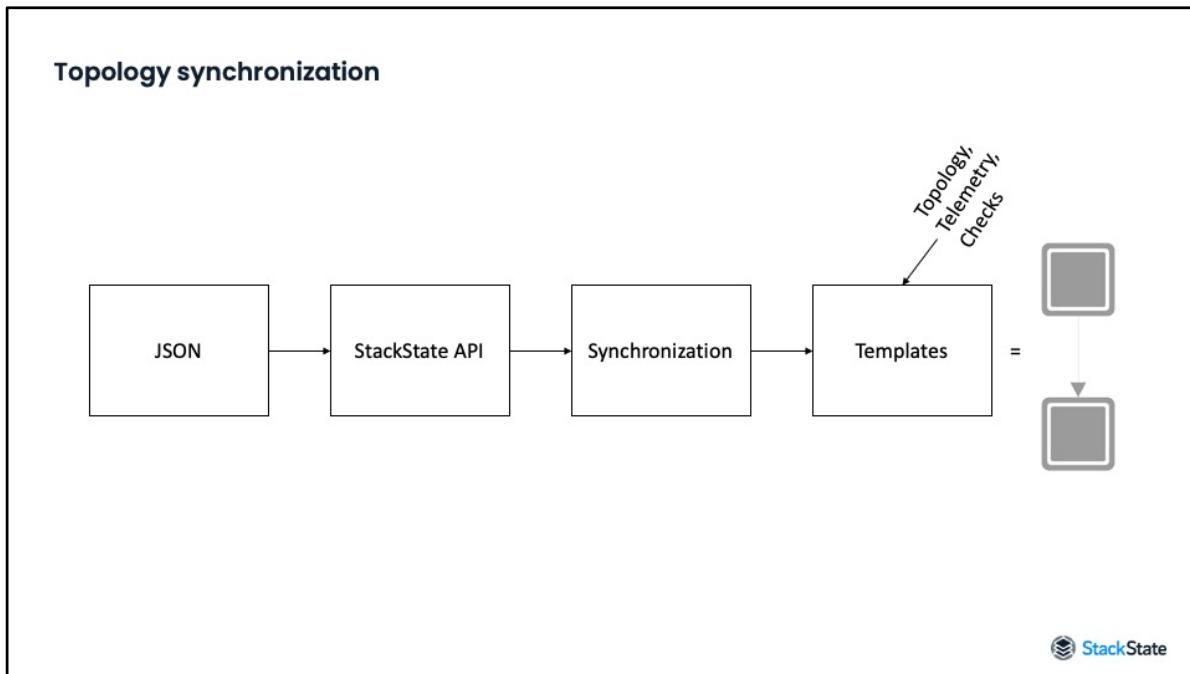
Let's have a look into how StackState synchronizes topology information from different sources.

Topology synchronization

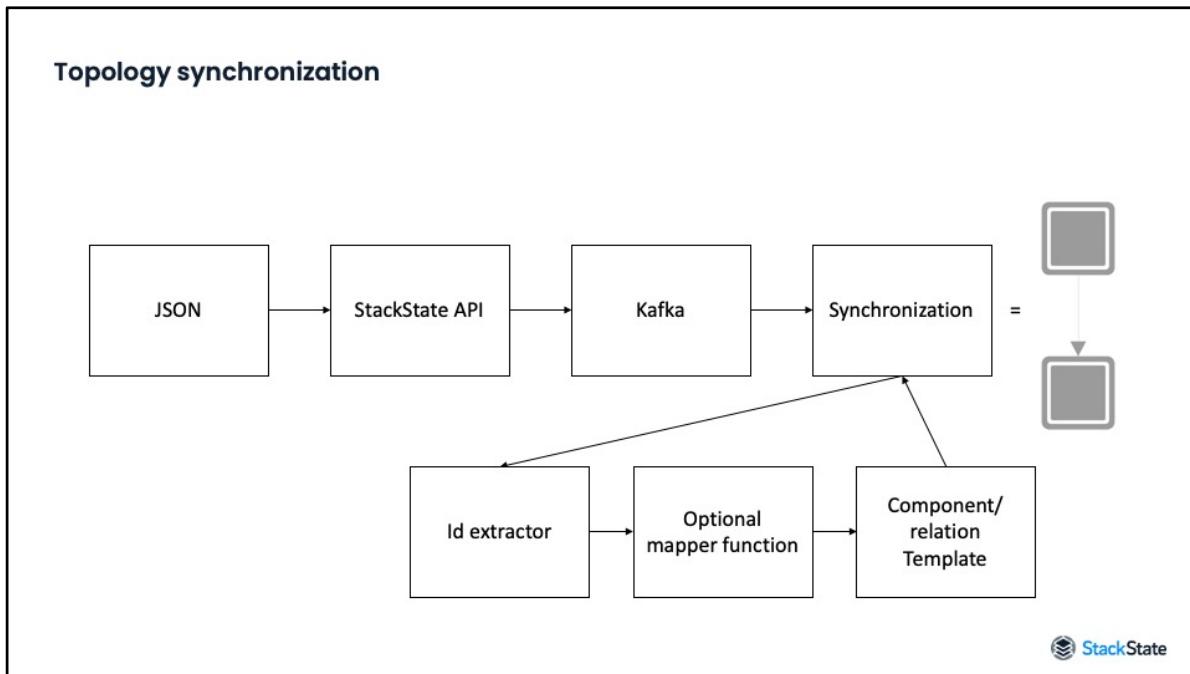
- Process of synchronizing topology information from different sources
- Process, overview:
 - Input from data source
 - Map input to component/relation template
 - Visualize
- Template:
 - Defines name, type, description, version, etc.
 - Defines telemetry streams (telemetry data sources)
 - Defines checks



Topology synchronization is the process of synchronizing a topology data source with StackState. StackState can have multiple synchronizations, one per data source, to form a combined topology. Depending on the data source, there are different ways of collecting the topology information. The collected topology is sent to the StackState receiver for processing. The synchronization process will take the input components and relations and map these to a template. The template defines what the component will look like. The template essentially maps the input data to what the name of the component will be, for example. The template also defines what telemetry streams and checks the component or relation will get.



Simply put, a JSON payload is send to StackState's API, the StackState receiver, containing information about topology extracted from a single data source. The synchronization process reads the received topology. It maps each component and each relation to a template. The templates form, or materialize, the actual components and relations that you can see in the topology visualization.



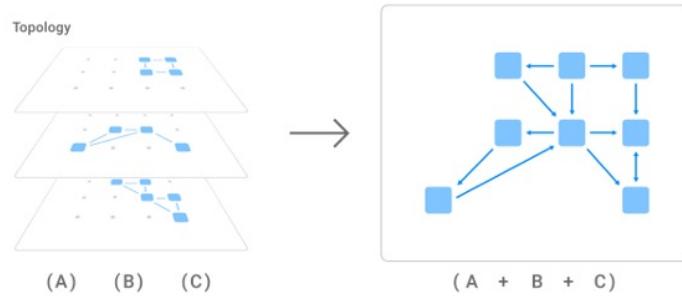
Adding a little bit more detail here. A JSON payload is send to the StackState receiver, containing information about topology extracted from a single data source. The receiver puts each component and relation on the Kafka bus that is specific to one data source. The synchronization reads the received component and relation messages from the Kafka bus in batches. The synchronization processes each topology element, components and relations, individually. One or more identifiers are extracted by the id extractor to unique identify a topology element in the topology. Optionally, data is transformed if need to. After that, the data is passed on to a component or relation template. After these steps topology will appear in the topology visualizer and the defined streams and checks are also available and running.

Topology synchronization

Merging

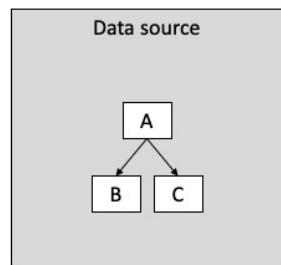


Before diving into what each step in the synchronization exactly does, let's have a look how components merge together in StackState, bringing multiple data sources together.

Topology synchronization – merging

Consider having three data sources that each provide components and relations, each forming a topology. In the picture the three data sources are labeled (a), (b), and (c). Between the three data sources there is overlap in topology. StackState merges together the overlapping components and relations to form a single topology, as seen on the right. In essence, topology merging happens when there is an overlap in components and relations originating from one or more topology data sources.

Topology synchronization – merging



- External ID defines the uniqueness in one topology.
- Relations use external ID of components
- Example data source defines:
 - Components:
 - A,
 - B,
 - C
 - Relations:
 - Source id A and target id B,
 - Source id A and target id C



Within a single topology data source, each component and relation has an identifier called the external ID. The external ID is used in a data source for unique identification of the component or relation. Next to an external ID, a relation defines a source and target external ID of the components it connects. The external ID is extracted by the ID extractor in synchronization.

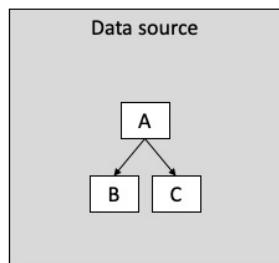
The example shows a single data source consisting of three components and two relations. The external id of each component is shown inside the box. The component at the top of the example topology has external ID 'A', or component A for short. There is a relation between component A and component B, and a relation between component A and component C.

Having two or more data sources using the exact same identifiers for any component or relation will not interfere with each other. The external ids are local to one data source. The external ID is not used for merging, we'll get to merging in a minute. When StackState receives a component payload from a data source, it will check its external ID. If the component

is not yet there, it will be created. If the component is already present, StackState will determine if the component needs an update. When a component's external ID changes, due to some reason, a new component is created. The component with the previous seen external ID will be removed or will stay behind, depending on the data source. We'll cover this scenario in the synchronization section about topology snapshotting.

One data source may report multiple components or relations with the same external ID in one data collection run, the same behavior is applied; most likely the last definition will be used assuming that the payload differed from the previous seen payload.

Topology synchronization – merging



- Zero, one, or more identifiers can be applied to each component
- Two components with the same identifier will be merged into one component
- externalId != merge identifier, can be included via the id extractor
- Universal resource name (URN) schema is used, e.g. urn:host:<fqdn>



So, how does the merging of components and relations work?

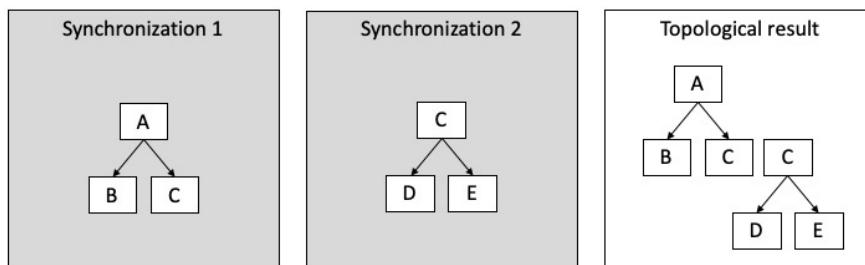
Next to the external ID, a component or relation can have an identifier. This identifier works on a global level, independent of individual data sources. The identifiers are extracted by the id extractor in synchronization. A component can have zero, one, or more identifiers set. An identifier might be the same as the component or relation's external ID. An identifier can be either a hostname, fully qualified domain name, IP-address, or any other id as long as the component or relation is uniquely identifiable among all data sources. It is good practice to use a standard formatting for the identifiers used. StackState has a URN schema that the StackPacks use. For example, a host uses schema urn:host followed by the hostname or FQDN. A schema helps in distinguishing values that may be the same but have different meanings.

A component or relation's identifiers are used for merging and topology events. When two data sources both report a single component and have the same identifier, the component is merged together. A single component can be found in StackState's visualization instead of two. When identifiers don't match, two components will be shown in

StackState's visualization. A merged component will contain information from all data sources it was merged from.

Topology synchronization – merging

External ID: shown in the box
Identifiers list: empty



Here's an example to illustrate the difference between external ID and identifiers used for topology merging.

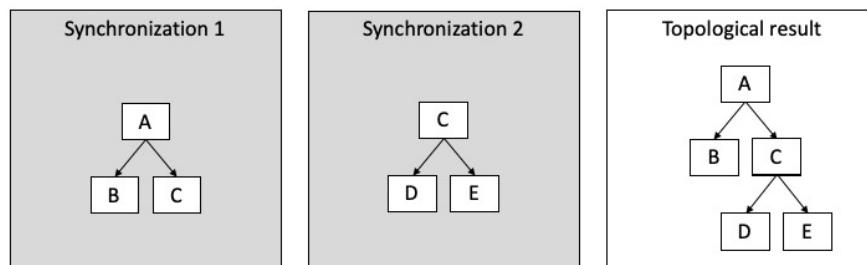
We have two synchronizations set up, synchronization 1 and 2. Each synchronization is representing a data source that both report three components and two relations. The external IDs are represented inside the boxes. Both synchronizations have a component C. You may assume that this is the same component, but reported from two different data sources. No component nor relation has an identifier set. The resulting topology would look like as shown on the right. From the topological result, no components have been merged together. Two component C's are shown since external IDs are local to a data source and no identifiers were used.

The letters in the topological result are not external IDs, they are put there for clarity to indicate which component we are talking about. Let's say these are the component names. In StackState components and relations have an database identifier or number after the data sources have been processed by the synchronization. A component or relation in

StackState can be merged from one or multiple components or relations, therefor a external ID representation is incorrect since these are local to the data source.

Topology synchronization – merging

External ID: shown in the box
Identifiers list: set to equal externalId



Merging topology happens based on overlap in identifiers, not external ID

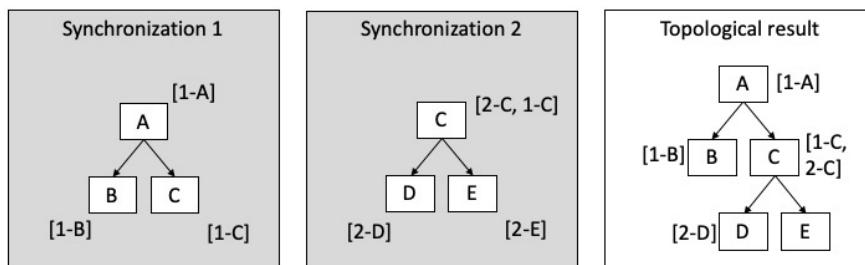


Let's have a look at the same example. This time an identifier is set for each component. In this case, the identifier is set to be the same as the component's external ID. The topological result is shown on the right. Component C, reported from both synchronizations, will have an overlap in identifiers causing a merge to happen.

Identifiers are used by StackState to merge together subsets of topology originating from multiple data sources to form one, ideally, complete topology.

Topology synchronization – merging

External ID: shown in the box
Identifiers list: displayed in []



Merging topology happens based on overlap in identifiers



Here's another example. In this example the identifiers are different from the external IDs used. The topology is the same as previous examples, three components and two relations from two synchronizations. As before, the external IDs are shown in the boxes. The identifiers of the components are shown in square brackets. Component A has identifier '1-A', and so on. The number prefix indicates the synchronization for clarity. In practice the identifiers would have been in URN format. Component C from Synchronization 2 has two identifiers set, instead of one, causing an overlap with Component C's identifier from the first synchronization. An overlap in identifiers will cause the components to merge. The resulting topology shows that component C has two identifiers set and the topology is merged together.

In this case, Component C acts a glue between synchronization 1 and synchronization 2 to form one topology. In this case component C from synchronization 2 had enough data to support an overlap. If the '1-C' identifiers was missing, then no merge would have happened. The same situations occur in actual data sources. One data source reports a certain set of component and relations while another data source reports their

own set. If there is any overlap in identifiers, a merge will happen. There has to be overlap, either one or the other data source needs to report the overlap. A CMDB may be able report a virtual machine and report a connection to a business service. A hypervisor might report the same virtual machine and information about storage. Both reported virtual machines have overlapping identifiers, resulting in a merge. Now you have a topology reporting a business service with relations to virtual machines and storage, all from two data sources.

The data sources provide the necessary identifiers for merging to happen. Without overlapping identifiers, no merge will happen. The URN format helps to facilitate this by standardizing the format of the identifiers. Hostnames will be placed in the urn:host: namespace, for example. Identifiers are case sensitive.

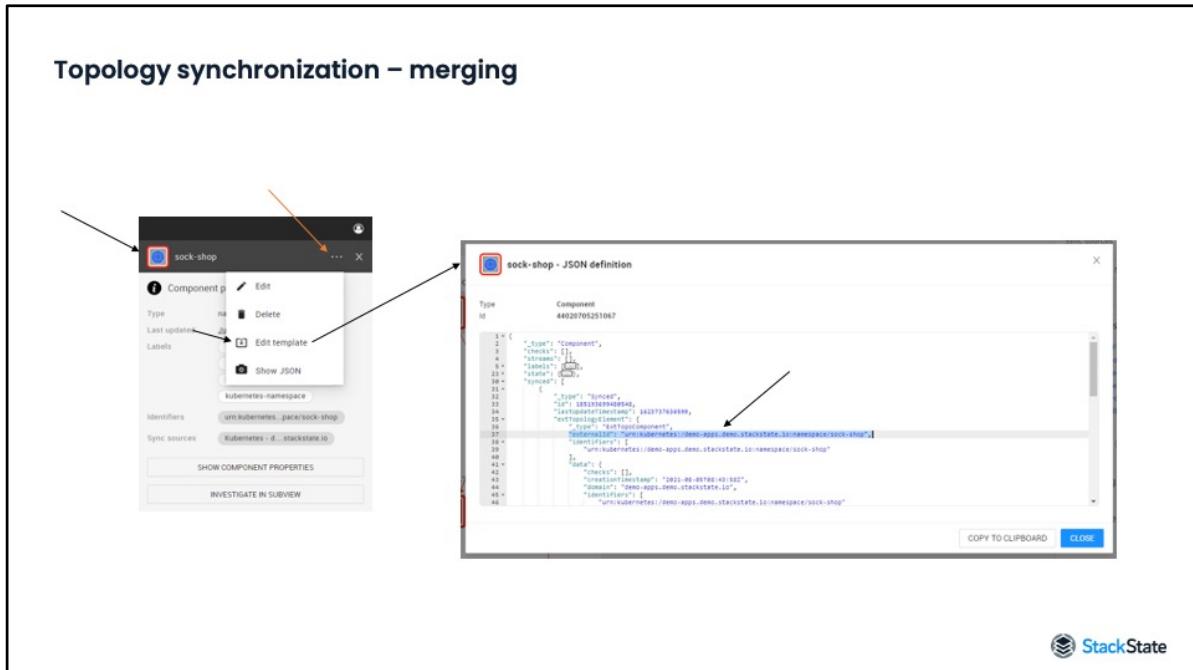
Topology synchronization – merging

The screenshot shows two panels from the StackState application. On the left is a 'Component properties' card for a component named 'sock-shop'. It displays basic information like type (namespace), last updated (Jun 15, 08:13:56), and labels (name=sock-shop). A callout arrow points from the 'Identifiers' section to the right panel. This section contains the identifier 'urn:kubernetes:demo-apps:demo.stackstate.io:namespace/sock-shop'. Below this is a 'Sync sources' section with 'Kubernetes <...> stackstate.io'. At the bottom is a 'SHOW COMPONENT PROPERTIES' button.

On the right is a detailed 'Component properties' dialog for the same 'sock-shop' component. The 'General' tab is selected, showing the component's name ('sock-shop'), type ('namespace'), labels ('cluster-name=demo-apps:demo.stackstate.io, kubernetes-namespace, name=sock-shop, stackpack=kubernetes'), version ('-'), last updated ('Jun 15, 2021, 08:13:56'), and description ('-'). The 'Identifiers' section lists the same identifier 'urn:kubernetes:demo-apps:demo.stackstate.io:namespace/sock-shop'. The 'Sync sources' section also lists 'Kubernetes - demo-apps:demo.stackstate.io'. The dialog includes tabs for 'General', 'Logs', 'Metrics', 'Traces', and 'OK' at the bottom right.

StackState

The identifiers of a component or relation is shown in the properties pane of the component or relation. The identifiers are shown near the top of the pane. The identifiers are also listed on the general tab after clicking 'show component properties'. The shown component here has one identifier set.



The external ID of a component or relation is a bit more hidden. For debugging it is useful to know where to find a component or relation's external ID. If you open the properties pane of the component or relation, on the top right you have a menu shown by the three dots. The last option in the menu is 'show json'. A new pane opens up showing the JSON definition of that component or relation. There is a key called 'synced' which is a list of components, each entry comes from a data source. Each entry has an 'extTopologyElement' block. In this block the 'externalId' can be found. If a component has been merged together from two sources, two 'synced' entries are to be found. It can also be possible that a component has been merged together from one data source, then two entries can also be found in the 'synced' block. The 'extTopologyElement', internally, stands for external topology element, an element from an external data source where element refers to a component or relation.



Let's have a look at the synchronization settings page and walk through one topology synchronization.

Topology synchronization – settings

Synchronizations

StackState can synchronize topology information from different sources. This includes your own sources. This way StackState enables you to create a model of your complete landscape. [Read more...](#)

	Name	Description	Created Components	Deleted Components	Created Relations	Deleted Relations	Status	Errors	...
<input type="checkbox"/>	AWS Sync for Account - Demo		622	301	1591	1204	Running	0	Edit Delete Reset data Export

StackState

The topology synchronization settings page shows all synchronizations currently active in StackState. The synchronization has a name and optionally a description. Each synchronization have some counters and a status. The status indicates the operation state of the synchronization, this should be Running. Running means that the synchronization is processing topology information from the Sts data source. The error count relates to topology related errors after processing, this may be template problems and such. There is a section on troubleshooting synchronization errors for more information about the error counter.

The created components counter indicates how many components were created over time and deleted components indicates how many components were removed over time. Subtracting the deleted component count and the created component count gives you a good idea of how many components there currently are in StackState. This number does not represent the actual number of components. Depending on component merges, the number may be skewed. One example of that happening is because of unmerges, a synced component was merged with another component, however that other component

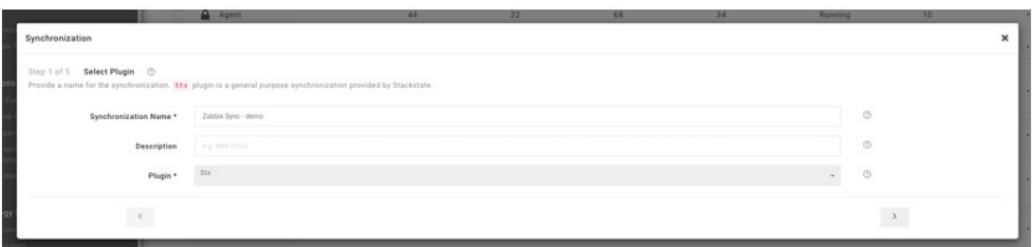
may have been removed. The deleted components counter will increase when a component is removed. The delete counter will not be increased in the unmerge case since the component still exists, however one of the two synchronizations did not report on it anymore. The created relations and deleted relations counter act similar to the component counters.

On the right, each synchronization has a menu. From the menu you can edit, delete, reset, or export the synchronization. We are going to look at edit synchronization in a minute. Delete will remove the synchronization from StackState. Reset data resets the synchronization and all previously create components and relations will be removed or unmerged. All counters will be reset to zero. Resetting may take a while depending on the number of components and relations that were synced by the synchronization. The synchronization status will jump to Resetting during the reset operation. Once the reset is done, the status will change to Running again. Under normal operations, resetting is not necessary. Be aware that resetting a sync may have impact on view definitions, the view's STQL queries may refer to a database id that is not there anymore after a sync reset. The settings can be exported via the export button, this will download the synchronization's definition in STJ file format.

At the top there are three buttons, add sync, export sync, and refresh. You can add a topology synchronization via the add sync button. Usually, synchronizations are created by StackPacks. Export sync button can be used if you tick one or more check boxes in front of each synchronization's name. You can export multiple synchronization definitions in one go. Refresh forces the counters to refresh. The counters are automatically refreshed periodically, for the impatient among us.

Let's have a look what's inside a synchronization.

Topology synchronization – settings



The screenshot shows a modal dialog titled "Synchronization" with the sub-tittle "Step 1 of 5 Select Plugin". It asks for a name for the synchronization, with "sts" plugin provided by Stackstate. The "Synchronization Name" field is filled with "Zabbix Sync - demo". The "Description" field contains "e.g. Web Client". The "Plugin" dropdown is set to "Sts", which is highlighted in blue. There are navigation arrows at the bottom of the dialog.

StackState

After opening a synchronization's settings page, via edit, you get a dialog that acts like a wizard. There are five steps per synchronization. The first step is shown here.

The first step of the synchronization defines the name, optional description, and the plugin to use. There is only one plugin available, that's why the plugin box is grayed out. The only plugin available is the Sts data source. Essentially, the sync will listen to a Kafka data source.

Topology synchronization – settings

The screenshot shows the 'Topology synchronization – settings' interface. A modal window titled 'Synchronization' is open, showing 'Step 2 of 5 Data Source And Query'. It contains the following configuration:

- Choose source:** ZABBIX INSTANCE - DEMO
- Component id Extractor:** ZABBIX COMPONENT ID EXTRACTOR
- Relation id Extractor:** ZABBIX RELATION ID EXTRACTOR
- Start from earliest available topology data:** Off
- Identifier:** umustackpack.zabbix.instance:89999051267471:sync:zabbix

Below the modal, there is a progress bar with the status 'Running' and a value of 0. The StackState logo is visible at the bottom right.

The second step in the synchronization defines the Sts data source to use. The Sts data source has to be defined first. The Sts data sources can be found in the settings pages. The Sts data source defines the actual Kafka topic. The Sts data source can be chosen here. The data source cannot be changed after the synchronization is created.

The id extractors for components and relations are defined here. The id extractors are the first step in the synchronization processing pipeline to extract the external ID, type, and identifiers from the input data.

The switch 'start from earliest available topology data' is usually set to off. This switch controls the Kafka topic offset. When it is set to off, the Kafka consumer will be set to the end of the topic. When the switch is set to on, the Kafka consumer will be set to the very first available offset, essentially processing historical data first before catching up to more recent data.

The synchronization's identifier is also defined in this step.

Topology synchronization – settings

The screenshot shows the 'Synchronization' interface in StackState. The current step is 'Mapping Components'. The interface has a header with the step number and title, followed by a detailed description of what the step entails. Below this is a table with columns: Source Type, Action, Mapping function, Template function, and Merge Strategy. A blue button labeled '+ ADD NEW MAPPING' is visible. At the bottom, there's a section titled 'Other Sources' with dropdown menus for 'Create', 'ZABBIX COMPONENT...', 'ZABBIX COMPONENT...', and 'Merge, prefer theirs'.



The third step of the synchronization defines the component mappings. A component mapping is about selecting the component template given the input data such that a component can be visualized. Essentially, a mapping defines how input data is processed to become a component in StackState.

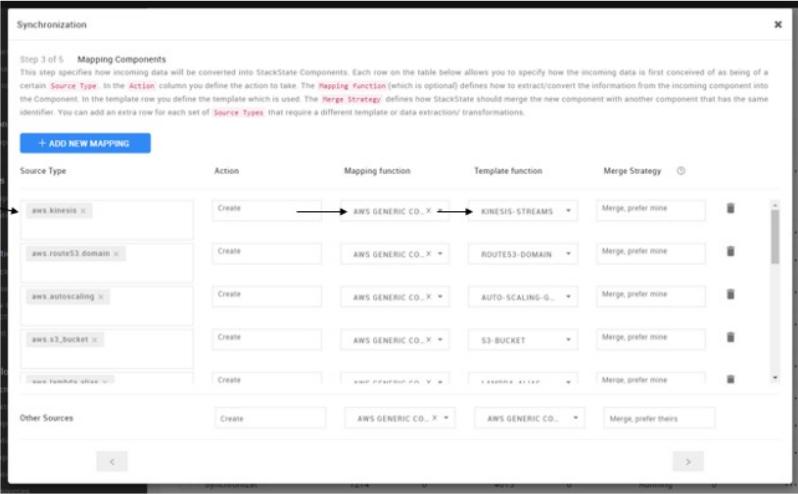
The mapping starts with the component type. The previously defined id extractor extracts the component type from the input data. The component type is matched against the defined mappings in step 3. If there is an explicit mapping available for a given component type, that mapping is used by the synchronization process. If there is no explicit mapping available for a given component type, a default mapping will be used by the synchronization process. At the bottom of step 3, Other sources defines the default mapping.

A maximum of one mapping can be defined for any component type. Component types are case sensitively matched. A mapping defines an action, mapper function, a template function, and merge strategy. The action defines if you want to create a component or a placeholder. The

mapper function is optional and can be used to transform input data. Data transformation is useful when there is no control over the data source provided data or some custom logic is required. A template function defines how the component will look like in StackState. Merge strategies define how components will be merged together when identifiers match between different synchronizations. We'll cover these topics separately in another section.

In the example shown here, there are no explicit mappings defined. This means that the synchronization process will use the default mapping for any topology data that is received from the data source. In this case, the default mapping has a mapper function defined and a template selected. All input data from the data source will be processed by the default mapping to form a StackState component.

Topology synchronization – settings



The screenshot shows the 'Synchronization' interface in the StackState tool. It's titled 'Step 3 of 5 Mapping Components'. The interface is a table with columns: Source Type, Action, Mapping Function, Template function, and Merge Strategy. There are two sections: 'Source Type' and 'Other Sources'. Under 'Source Type', there are five rows corresponding to AWS services: 'aws.kinesis', 'aws.route53.domain', 'aws.autoscaling', 'aws.s3.bucket', and 'aws.lambda.alien'. Each row has a 'Create' button under 'Action', a 'AWS GENERIC CO...' dropdown under 'Mapping Function', and a 'KINESIS-STREAMS' dropdown under 'Template function'. The 'Merge Strategy' for all rows is set to 'Merge, prefer mine'. Under 'Other Sources', there is one row with a 'Create' button, 'AWS GENERIC CO...' mapping function, and 'AWS GENERIC CO...' template function. The merge strategy is also 'Merge, prefer mine'. A note at the top explains the purpose of the table: 'This step specifies how incoming data will be converted into StackState Components. Each row on the table below allows you to specify how the incoming data is first conceived of as being of a certain Source Type. In the Action column you define the action to take. The Mapping Function (which is optional) defines how to extract/convert the information from the incoming component into the Component. In the template row you define the template which is used. The Merge Strategy defines how StackState should merge the new component with another component that has the same identifier. You can add an extra row for each set of Source Types that require a different template or data extractions/transformations.'

Here's another example of step 3, showing component mappings from the AWS StackPack. You can see here that there are several mappings defined.

If a component is reported from AWS, a kinesis stream, is processed by this synchronization, the id extractor will extract the component type. The AWS integration logic provides the component type to StackState based on the resource type. Each supported AWS resource type will have its own component type. The component type is matched against the mappings defined in the third step. Let's say the component type is 'aws.kinesis'. The first mapping in this example is matched. The reported component is processed by the mapper function 'AWS generic component mapper'. Subsequently, the output of the mapper function is passed to the 'kinesis-streams' template function. The template function is the last step in materializing the component that you can see on the visualizer. If the extracted component type was set to upper case letters or the component type had something appended to it, then the mapping would not have been matched and the default mapping would have been used.

Topology synchronization – settings



The screenshot shows the StackState Synchronization interface. The main title is "Synchronization" and the sub-step is "Step 4 of 5: Mapping Relations". A detailed description explains that this step defines how incoming data will be converted into StackState Relations. It describes the columns: "Source Type" (dropdown), "Action" (dropdown), "Mapping function" (dropdown), and "Template function" (dropdown). Below the table, there is a section for "Other Sources" with dropdown menus for "Create", "Action", and "Template function". At the bottom right of the interface is the StackState logo.

The fourth step 4 of the synchronization defines the relation mappings. The mapping process for relations works similar to that of components. The relation type is extracted by the relation id extractor and that is matched against the defined mappings in step 4. If there is no mapping explicitly defined, then the default mapping is used.

Topology synchronization – settings

The screenshot shows the 'Summary' step of the synchronization configuration. The configuration includes:

- Name:** Zabbix Sync - demo
- Description:** (empty)
- Plugin:** Sts
- Data source:** Zabbix instance - demo
- Component Actions:**
 - External Type:** Create
 - Action:** Create
 - Mapping Function:** Zabbix Component Mapping function - demo
 - Template Function:** Zabbix Component Template
 - Merge Strategy:** Merge, prefer theirs
- Relation Actions:**
 - External Type:** Create
 - Action:** Create
 - Mapping Function:** None
 - Template Function:** Zabbix Relation Template

A large blue 'SAVE' button is located at the bottom right of the dialog.

The fifth, and last, step shows a summary. It shows all the settings from previous steps. You can save any changes made to the synchronization by clicking on the save button on the summary page. You can only save changes on this page. You can close the dialog on the top right at every step without saving.

There is no need reset the sync after this point. The changes are picked up automatically. Changes to the id extractor, mapper functions, template functions, etc. are also picked up automatically.

Topology synchronization

Id extractor



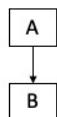
Synchronization reads the topological information from the Kafka bus, as defined by the Sts data source. The first step is to extract basic information such as the external id and type of the component or relation.

Topology synchronization – id extractor

From a received topology element, Synchronization extracts;

- External ID,
- Component/relation type,
- (optional) identifiers
- (optional) data transformation

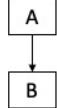
Example:



During processing of topology in the topology synchronization process, the id extractor extracts the external ID and the component or relation type. Additionally, identifiers can be extracted and it offers a data transformation capability. The id extractor phase of Synchronization is driven by an id extractor function. This function is powered by a Groovy script which is executed on each to be processed topology element. The function is therefore stateless. There is a separate id extractor for components and relations.

Let's illustrate how the id extractor function operates on the following example. Consider that a topology of two components and a relation between them is sent to the receiver. Synchronization will process each topology element in order. First Synchronization will process the components and after, all components have been processed, the relations are processed.

Topology synchronization – id extractor



```

Simplified component JSON:          Simplified component JSON:
{
  externalId: componentA
  type: process
  data: {
    name: Component A
    layer: processes
    domain: department B
    environment: Production
  }
}
{
  externalId: componentB
  type: VM
  data: {
    name: Component B
    layer: machines
    domain: department B
    environment: Production
  }
}

```



Given the example of two components and one relation, this is a simplified JSON representation of both components the receiver might receive. Component A is defined on the left and component B is defined on the right. Component A is of type process and component B is of type VM. In each data payload you can find a name, layer, domain, and environment. The data payload may differ between component payloads.

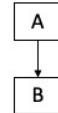
The JSON the receiver will always have the keys; external ID, type, and data. Most id extractors are quite straight-forward. The external ID and type are mandatory, it's usually just read unchanged. Occasionally, you might see the types being transformed to lower case. This is to avoid case sensitivity issues in the next step of the Synchronization process.

Next to extraction of the external id and type, identifiers can be extracted. Usually this is a process of parsing through the data payload and adding the identities to a HashSet. The external ID can also be added to the identities HashSet if need to.

At the end of the id extractor function a Groovy method is called to pass

the collected information to the next step in the synchronization.

Topology synchronization – id extractor



Simplified relation JSON:

```

{
  externalId: componentA-uses-componentB
  type: uses
  sourceld: componentA
  targetId: componentB
  data: {}
}
  
```

External IDs of components



Here's the simplified JSON representation of the relation that the receiver might receive. The relation payload is quite similar to the payload of a component. A relation has two additional mandatory keys, namely sourceld and targetId. The sourceld and targetId refer to a component's externalId. The sourceld refers to the component from which the relation should start, the source. The targetId refers to the component to which the relation should point, the target. If no component could be found whose externalId matches the sourceld or targetId, the relation is not created and a sync error is generated.

Commonly, the external ID of a relation is the concatenation of the sourceld, type and targetId. You can then read the externalId, in this example, like component A uses component B.

Topology synchronization – id extractor

Edit id extractor function

Name * Auto sync component id extractor

Description A generic component id extractor function for topology elements.

Parameters

Name	Type	Required	Multiple
topologyElement	JSON	true	false

ID Extractor *

```

1 element = topologyElement.asReadOnlyMap()
2
3 externalId = element['externalId']
4 type = element['typeName'].toLowerCase()
5 data = element['data']
6
7 identifiers = new HashSet<String>
8
9 if(data.containsKey("identifiers") && data["identifiers"] instanceof List<String>) {
10     data["identifiers"].each{id -
11         identifiers.add(id)
12     }
13 }
14
15 return Sts.createId(externalId, identifiers, type)
16

```

externalId: componentA
type: VM
data: {
name: Component A
layer: machines
domain: department B
environment: Production}

externalId, merge identifiers, component type

Here's an actual id extractor function. This is a component id extractor included in the Custom Synchronization StackPack. Let's walk through the code and see what happens. On the right an example payload is shown.

The id extractor function has parameters. Only one parameter is defined which the function requires to function properly. You can read it as an input to the code for execution. In this case, parameter topologyElement is of type JSON and is a system parameter. In Synchronization, the topologyElement is going to contain the component payload as shown on the right, in this example.

The first line of code grabs the JSON payload and converts it to a Groovy Map. Lines three, four, and five extract the externalId, type, and data payloads. The type is lower cased after extraction. Line 7 initializes an empty Groovy HashSet to store the component's identifiers in. Line 9 till 13 is logic to extract identifiers from the data payload. First it is checked whether the identifiers key exists in the data payload. If the identifiers key is present and the identifiers payload is an JSON array of Strings,

then the identifiers array is iterated through and added to the identifiers HashMap we defined previously on line 7. In the example no identifiers key is present in the data payload, therefor no component identifiers are added. Line 15 defines the return value of the id extractor function. Method 'Sts.createId' is called to which we pass the externalId, identifiers, and the type.

After the id extractor function has been executed, the next step is the optional mapper function. If no mapper function is defined for a sync mapping then the next step is a component/relation template.

Topology synchronization

Mapper function



Between the id extractor and the templates we can have an optional step to do some data transformation. We can do this with a so called mapper function. Let's see what a mapper function can do.

Topology synchronization – mapper function

- Optional
- Data transformation
- Examples:
 - Name to lower/upper case
 - Add labels
 - Dynamically set checksstreams
- Outcome is used as input to a template



A mapper function is an optional step between the id extractor and template. The id extractor extracts the type on which the synchronization determines which mapping to use. The mapping may optionally include a mapper function to do data transformation, if needed. A mapper function is useful when data transformation needs to take place when there is no control over the input source. Another option for mapper functions is to dynamically add labels, streams, or checks for the template to use based on input data assuming that the template supports it.

Essentially, the mapper function accepts the JSON input, does transformation, and outputs JSON. This JSON output is passed to the template for further Synchronization processing.

Topology synchronization – mapper function

Source Type	Action	Mapping Function	Template Function	Merge Strategy
vSphere-hostsystem	Create	VSPHERE HOST SYS...	VSPHERE HOST SYS...	Merge, prefer theirs
vSphere-virtualmachine	Create	VSPHERE VIRTUAL...	VSPHERE VIRTUAL...	Merge, prefer theirs

Other Sources

StackState

In a Synchronization, the mapping functions are defined on the third page for components, and on the fourth page for relations. Per component or relation type you can have a maximum of one mapper function set. The other sources, or types, can also have one mapper function set.

Topology synchronization – mapper function

```

17 labels <- "zabbix-instance-name:demo"
18
19
20 element.data.put("labels", labels)
21 element

```

The screenshot shows the StackState interface. On the left, there's a card for a 'Zabbix server' component. The component card displays several properties: Type (zabbix.host), Last updated (Aug 11, 2020, 13:51:47), Labels (zabbix-stackpack-zabbix, instance_art_localhost, zabbix-instance-localhost, root_group-Zabbix servers, zabbix-instance-name demo, zabbix), Identifiers (uri-host:/Zabbix server Zabbix server), Sync sources (Zabbix sync - demo), and SHOW COMPONENT PROPERTIES. Below the component card, there's a 'Machines' section showing a single 'Zabbix server' node with a status of 'Live'. To the right of the component card, there's a 'Health' section showing three items: 'zabbix host status check' (status: DEVIATING), 'Propagated health' (status: DEVIATING), and 'Telemetry streams' (Zabbix Host EventStream).

The JSON payload returned by the id extractor will be passed to the mapper function as a parameter. The mapper function's parameter is named `element`. The script can be used to transform the input JSON and yield an output JSON. The output is passed to the next step in the Synchronization process, templates.

The mapper function shown here is part of the Zabbix StackPack. The first line of code defines a list for labels. Lines 2 to 16 contains code to account for variables the template expects. It might be that occasionally the data source does not report a specific key, to avoid template errors, empty variables are set. On line 18 a label is added to the labels list. The label value seems hardcoded here, which is true. Actually, in this case the value was generated based on user input during installation of the Zabbix StackPack. The list is added to the element, or JSON payload, on line 20. Line 21 returns the transformed element. The resulting component shows that the label was added.

Topology synchronization – mapper function

Edit component mapping function

Name * Name to upper case

Description e.g.

Parameters System

Name	Type
element	JSON

Script *

```
1 def name = element.data.name
2 element.data.name = name.toUpperCase()
3 return element;
```



Here is another very simple example of a mapper function. From the input JSON, the name is extracted from the data payload. The name is transformed to use upper casing and the previous name is overwritten in the JSON. Lastly, the altered input JSON is returned to be passed on to the template.

Topology synchronization – mapper function

The screenshot shows a component properties pane for 'HOSTA / SYNC 1 - Component properties'. The 'Sync1' tab is selected. The pane displays the following fields:

Field	Value
name	HostA / sync 1
description	myDescription with link to [Google](https://google.com)
domain	Default

At the top of the pane, there is an 'Advanced' link. Arrows from this link point to both the 'Sync1' tab and the transformed 'HostA / sync 1' value in the 'name' field.

StackState

The result of the simple mapper function is as follows. In this example the component's data payload contained a name, description, domain, and possibly some more fields. The pane shown here is the component properties pane, accessible from the component details panel. This pane shows the component's unprocessed information as it was received by the receiver, the external topology element's data payload. At the top of the pane and on the component in the visualizer you will find the upper case name as transformed by the mapper function.

Topology synchronization

Component/relation templates



After the id extractor and optional mapper function has done their processing, it is time for the last step in topology synchronization. The last step is to materialize a component or relation. Given the input data, templates are StackState's way of creating a component or relation you see on the visualizer.

Topology synchronization – component/relation templates

- Materializes a component or relations
- Defines component / relation properties
- Can include telemetry from various sources
- Can include health/run checks based on one or more telemetry streams
- Handlebars templating
- Template materialization should result in a valid JSON
- One template can be applied to a class of components



The last processing step in topology synchronization is to materialize components and relations such that they become visible on in views. This is done by templates. Input JSON data is fed into a template which in turn yields a component or relation definition. There are two types of templates, you have component templates for components and relation templates for relations.

The template defines all the properties of a component or relation. Properties like name, description, version, layer, domain, environment, etc. Next to these properties, templates also defines which telemetry streams, check, and labels a component or relation will get.

The templating language used is Handlebars. A template can have loops, if statements, and other constructs. During run time, the template must generate valid JSON. When the template yields an invalid JSON, the component or relation is not created and a sync error is generated. The synchronization's errors counter will increase on each invalid template result. The template errors are logged in a synchronization specific log file in case of Linux and to standard out of the synchronization's pod in

case of Kubernetes. Troubleshooting templates is covered in more detail in the Troubleshooting section.

A template can be reused for multiple synchronization mappings. In other words, a single template can be used for multiple component types. This is especially useful for the ‘Other sources’ mapping in a Synchronization. One generic template defined as the ‘default’ template will materialize the component or relation for any type that was not explicitly mapped. This way you will always see the component end up in StackState. Similarly, you may define a single template that is used for multiple mappings.

Let’s have a look at some example templates.

```

1 - {
2   "type": "Component",
3   "checks": [],
4   "streams": [],
5   "name": {{element.data.name}}{{element.data.name}}[{{else}}{{element.externalId}}]{{/if}}
6   {{#element.data.description}}
7     description: {{#if element.data.description}}{{element.data.description}}{{/if}}
8   {{/if}}
9   type: {{#resolvedCreate}}"Component"{{/resolvedCreate}} element.type.name "Auto-synced Component" {{}},
10  version: {{#element.data.version}}{{/element.data.version}}{{/if}},
11  layer: {{#resolvedCreate}}"layer"{{/resolvedCreate}} element.data.layer "Auto-synced Components" {{}},
12  domain: {{#resolvedCreate}}"domain"{{/resolvedCreate}} element.data.domain "Auto-synced Domain" {{}},
13  environments: [
14    {{#resolvedCreate}}"Environment"{{/resolvedCreate}} element.data.environment "Auto-synced Environment" {{}},
15  ]
16 }
17

```

Similar to id extractors and mapper functions, templates have an input parameter. This input parameter is the JSON that was processed by either the id extractor or mapper function. The input parameter can be used to extract information from the input to use in the template.

Let's have a look at a component template. An example is shown here. In the template you see here has the following keys; _type, checks, streams, name, description, type, version, layer, domain, environments. The value for '_type' is hardcoded here, it's set to "Component" since the result is a component.

The value for checks is an empty JSON array. This means that the resulting component will not have any checks defined. We'll show you an example of a check definition in a minute.

The value for streams is also an empty JSON array. As you can guess, there will be no telemetry streams on this component. Both checks and streams are arrays of JSON objects, we'll see an example in a minute.

The name's value is a more complex statement. The value for name should result in a single JSON string. The value for 'name' will be used as the component's name as seen in the visualizer. Inside the double quotes an handlebars if statement can be found, between two curly brackets. Curly brackets are handlebars statements. What the if statement says is the following; when the input data payload contains a name key, use the value from the data payload's name. Otherwise, the component's external ID is used as its name. The if statement acts as a fallback mechanism. If the input JSON does not contain a name in the data object, then the name of the component will be the external ID of the component.

The description for components is optional. If you include a description, the value must be a JSON string. If you omit the key 'description' from the final JSON then no description is shown on the component. In this example a handlebars if-helper statement is put around the description definition on line 7. The if statement essentially checks if there is a description value is present in the data payload. Line 7 is only added in the final JSON if the statement on line 6 returns true.

The type determines the component type, and thus icon, of the component. Type expects a number, an id of the type that already exists in the database. We can get the database node id of the component type by referencing the name of the component type or by URN identifier. A handlebars helper does the resolving for you and yields the id required for the resulting JSON given. Usually information is retrieved from the input data and fed to a helper as its arguments. In this case, the value at location type.name is obtained from the input data before that value is being passed to the resolveOrCreate helper as its second argument. We'll cover these helpers individually since they are quite common in templates.

The version for components is optional. A JSON string is expected when the version is added to the template. The version will be shown on the component details pane and a topology event is generated when the version changes over time. The version here is set to the value retrieved from the input data's version payload.

Keys layer, domain, and environments are quite similar. They all expect a database node id and need thus need some resolving. Environments expect a JSON list of ids since a component can be in one or more environments. In this example there is only one environment being resolved. The layer, domain, and environment are set to what is supplied

in the input data. The integration supplies the values in this case. As an example, the AWS StackPack provided templates support setting custom AWS resource tags such that the layer, domain, or environment can be controlled from AWS dynamically. You define a resource tag on an AWS resource in the AWS console called 'stackstate-environment', for example. The integration will pass the resource tag to StackState where the component template will set the environment of the component to the resource tag's value.

As already seen here, a template may contain logic to form the resulting JSON that will represent the component. If-statements and loops are examples of logic that can be used in a template. We've seen if statements already. A loop can be used for the transformation of labels available from the input JSON to StackState labels. The input labels may have been generated by a mapper function or send directly to the receiver by the integration.

There is also a key for labels, which is not shown here. The labels key accepts a JSON array of label objects. The resulting labels will be the labels you see on the components in the visualization and can be used in view queries. Often you see some dynamic logic applied to the labels. Usually, a loop is used to dynamically transform an input string array to an array of label objects where the array is constructed in the mapper function or the data is received from the data source. We'll have an example of this in a minute.

The screenshot shows the StackState interface for managing topology synchronization. The top navigation bar has 'Topology' selected. Below it, a sub-navigation bar shows 'Component' and 'Relation' selected. The main content area is titled 'Topology synchronization – component/relation templates'. A specific relation template is being edited, with the title 'Edit relation template function' and the identifier 'demo-demo-relation-template'. The template details section shows parameters for 'System' and 'User', both labeled 'element' with type 'JSON'. The 'Required' column is checked for 'System', while 'Multiple' is checked for 'User'. The 'Template' section contains a JSON code block:

```

1<v [
2   "_type": "Relation",
3   "streams": [],
4   "labels": [],
5   "name": {{ element.sourceExternalId }} -> {{ element.targetExternalId }},
6   "description": "[{{ element.data.description }}]",
7   "component": "[{{ element.data.description }}]",
8   "({{!v}}"
9   "type": {{ relationType }} element.type.name "Auto-synced Relation",
10  "source": {{ element.sourceExternalId }},
11  "target": {{ element.targetExternalId }}
12 }
13
14 ]

```

The bottom right corner of the interface features the StackState logo.

Here's an example of a relation template.

The `_type`'s value here is fixed to "Relation" since the resulting topology element is going to be a relation.

Keys checks, streams, labels, name, and description have the same behavior as what we discussed for component templates.

The type determines the relation type. The relation type defines the directionality, the arrow you see in the visualization. The relation type is resolved based on the input data making the template quite generic.

A relation has source and target fields that grab the source and target external ID from the JSON input to set the relation's source and target components by their external IDs.

Topology synchronization – component/relation templates

```

"streams": [
    {
        "_type": "MetricStream",
        "dataSource": {{ resolve "DataSource" "Demo data source" }},
        "dataType": "METRICS",
        "id": -11,
        "name": "Demo stream",
        "query": {
            "type": "MetricTelemetryQuery",
            "aggregation": "MEAN",
            "conditions": [
                {
                    "key": "name",
                    "value": "server0.processor0."
                }
            ],
            "id": -20,
            "metricField": "value"
        }
    }
],
}

```

Each stream you see on the component or relation has a definition like you see here. Essentially, a stream is just a representation of a JSON definition. The template can be used to define a stream.

The example shown here is JSON definition of the metric stream you see on the right. The streams block accepts multiple JSON objects. So, you can define zero, one, or more streams in the template. In this example, one stream is defined.

The stream defined here is a metric stream. The '`_type`' and `dataType` fields defines the stream type. The stream type can either be a metric or log stream. This value is fixed.

The `dataSource` key defines which telemetry data source to use for the telemetry stream. This can be the any telemetry source defined in StackState settings, under telemetry sources. The `dataSource` expects a number as its value. Actually, it expects an internal node id. This id we don't have to find ourselves, we can use a helper to resolve this for us. In this case, a data source called "Demo data source" is resolved. The

helper returns the id that is required assuming a data source is defined with the provided name.

Each block has an id field. This value set in the id field is auto generated and is local to the current template. Each JSON object will have a negative id set. This is similar to the exported STJ files, which also contain negative ids. Most negative ids are superfluous. The negative ids are used to form dependencies between two JSON objects. We'll see later that health checks use negative ids to create a dependency on a telemetry stream. Remember the stream id, -11, as we are going to use that in the definition of a check.

Be aware that no duplicate negative ids are defined in the same template for any id field. The negative ids are resolved run time to newly created database node identifiers, having duplicates might result in unexpected and unintended configuration being referenced resulting in run-time errors in the synchronization.

The value of the key name defines the name of the stream as shown on the component/relation details pane. It is also reflected in the edit dialog as shown here on the right.

The query JSON block defines the filter of the telemetry stream. It has a type which defines whether the query is either a metric or log stream query. A metric stream query defines an aggregation, metricField and a set of conditions. The aggregation and metricField values correspond to what is set on the filter, as can be seen here on the right. The field conditions is a JSON array accepting JSON objects in the form of key/value pairs. These are the filters that are defined in the stream. Only equality matching is currently supported. In this example, there is only one stream filter set. The key is set to 'name' and the value is set to "server0.processor0". The telemetry data source is queried to show the telemetry that matches the defined filter conditions. This block is similar for an event stream. The difference is that an event stream only has conditions.

This template block defines a metric stream that is very static. If we put this JSON block in a component template, then all components that are created with this template end up having the exact same telemetry stream. All these streams would show the exact same data too since the filters are static. In this case, the filter is set to match a name to the value "server0.processor0". The used data source will return telemetry data matching the given filter.

You probably want to make the stream filter more dynamic. For instance, on the topology data you have information available that you can use to dynamically set the stream filter, the condition where value is set to "server0.processor0" can be changed in such a way to the filter value is obtained from the input topology data. Replacing the "server0.processor0" value to 'element.data.<something>' encapsulated in double curly brackets will make the stream filter dynamic based on the topology data received. A common use case is that the telemetry data source needs a specific filter to determine which data to retrieve for a specific component. The component or relation may already have this information available, which can be used by the telemetry data source to query the correct data. An example may be a hostname of a server. The component will have the host name available. The hostname is available in the template using the double curly brackets. The hostname can then be passed to the telemetry stream filter such that telemetry specific to that host is queried per server using the template.

We did not cover a log stream definition here. The concept is the same. We will generate the JSON as much as possible anyway.

Topology synchronization – component/relation templates

```

"checks": [
    {
        "_type": "Check",
        "arguments": [
            {
                "_type": "ArgumentMetricStreamRef",
                "downsamplingMethod": "MEAN",
                "maxWindow": 5000,
                "parameter": {{ get "urn:stackpack:common:check-function:metric-fixed-state" "Type=Parameter;Name=metrics" }},
                "stream": -11,
                "windowingMethod": "SLIDING"
            },
            {
                "_type": "ArgumentStateVal",
                "parameter": {{ get "urn:stackpack:common:check-function:metric-fixed-state" "Type=Parameter;Name=state" }},
                "value": "CLEAR"
            }
        ],
        "function": {{ get "urn:stackpack:common:check-function:metric-fixed-state" }},
        "name": "Demo check"
    }
],

```

The checks JSON array defines all check for the template. Zero, one, or more checks can be defined. Here's a definition of one health check.

The value for the `_type` key is statically set to “Check”. Defining a check requires the keys ‘arguments’, ‘function’, and ‘name’ to be present.

The value for the key ‘name’ defines the name you see on the check itself, in the component or relations details pane.

The function specifies the check function being used for this health check. The function’s value needs to be an internal node id. This id we don’t have to find ourselves, we can use a helper to resolve this for us. In this case, a ‘get’ helper is used. A ‘get’ helper resolves a node id using a URN identifier. The helper returns the id that is required assuming a check function. You can see the check function’s name shown on the check, as you can see here at the bottom right.

Topology synchronization – component/relation templates

```

"checks": [
    {
        "_type": "Check",
        "arguments": [
            {
                "_type": "ArgumentMetricStreamRef",
                "downsamplingMethod": "MEAN",
                "maxWindow": 5000,
                "parameter": {{ get "urn:stackpack:common:check-function:metric-fixed-state" "Type=Parameter;Name=metrics" }},
                "stream": -11,
                "windowingMethod": "SLIDING"
            },
            {
                "_type": "ArgumentStateVal",
                "parameter": {{ get "urn:stackpack:common:check-function:metric-fixed-state" "Type=Parameter;Name=state" }},
                "value": "CLEAR"
            }
        ],
        "function": {{ get "urn:stackpack:common:check-function:metric-fixed-state" }},
        "name": "Demo check"
    }
],

```

Check's parameters:

Name	Type	Required
metrics	Metric stream	<input checked="" type="checkbox"/>
state	State	<input checked="" type="checkbox"/>

Let's have a look at the check definition's arguments JSON array. The 'arguments' array defines all arguments that need to be passed to the check function.

In this example, the check function used is call 'Metric fixed state'. In a nutshell, this check function returns a fixed health state regardless of metric input. The fixed health state returned is the health state that is provided to the check function as an argument.

The check function's parameters are shown here at the bottom of the slide. It requires arguments being passed called 'metrics' and 'state'. The 'state' parameter defines the health state was is returned by the check function, as simple pass-through. The 'metrics' parameter specifies the check function's telemetry stream. In this case, the check function works on a metric stream therefor a metric stream parameter is present. Since the check function requires two arguments, we need to define two JSON blocks in the 'arguments' array, one for each parameter.

The first argument in the arguments JSON array is to set a value for the

check function's 'metrics' parameter. The JSON object has the _type set to 'ArgumentMetricsStreamRef'. This '_type' indicates that a metric stream argument is defined in this block. Essentially, a value is set to the metrics parameter of the check function. We need to tell StackState which telemetry stream to bind to which parameter of the check function. The 'parameter' defines the check function's parameter and 'stream' defines the telemetry stream. Both are internal node ids. We need to resolve these ids. In this case, the check function's parameter 'metrics' is resolved using the 'get' helper. Getting the stream node id works a bit different. We've seen the stream definition a couple of slides ago. We talked about the id key, that they define a negative number. The stream defined there had id -11 set. Since the stream is not created in run time yet, we cannot resolve any id for the telemetry stream. The stream is only created after the template has been processed. However, the stream and the check are defined in the same template. We can use negative ids to satisfy this dependency. Therefor, the value for the stream key is set to -11, to match the negative id of the telemetry stream. Run time these ids will be stored as an internal node id. Values for 'maxWindow' and 'windowingMethod' are specific to metric streams, and are set on the check, as can be seen if you edit the check on the component or relation details pane.

Some check functions require multiple telemetry streams, in these cases you just have more arguments defined in the JSON array, one per parameter.

The second argument in the arguments JSON array defines a value for the check function's 'state' parameter. Since the 'state' argument of the check function expects a health state value, the _type is set to 'ArgumentStateVal'. The 'value' defines the state as a JSON string. In this case the value "CLEAR" is passed. This causes the green color to appear on the check. You can also see the arguments of the check function on the check, as shown on the bottom right.

We did not cover a check function that require a log stream, multiple streams, or have a different set of required parameters. For each, the concept remains the same. We will generate the JSON as much as possible anyway. Similarly to what we've seen for stream filters, we can use handlebars to set arguments to values that are provided by topology information.

Topology synchronization – component/relation templates

The screenshot shows the StackState interface. On the left, a JSON code snippet defines a component's labels. On the right, the 'General' tab of the 'subnet-123 - Component properties' pane is displayed, showing the name 'subnet-123', type 'aws-subnet', and a 'Labels' section containing 'stackpack.aws'.

```

12 *
13 [
14   {
15     "labels": [
16       {
17         "_type": "Label",
18         "name": "stackpack:aws"
19       }
20     ]
21   }
22 ]
  
```

General AWS Sync for Account - Demo

Name	subnet-123
Type	aws-subnet
Labels	stackpack.aws

StackState

The labels JSON array defines the labels you see on a component or relation.

Each label is of type “Label” and require to have a name. The name defines what you see as the label on the component or relation details pane.

This is a very simple and static example taken from the AWS StackPack. It defines the label “stackpack:aws”. In this case, all components materialized by the template get this label. The user can filter in a view only on this label to get all components provided by the AWS StackPack, for example. We'll see a more complex example in a minute.

Topology synchronization – component/relation templates

Handlebars helpers/functions:

- add
- concat
- get
- getFirstExisting
- getOrCreate
- identifier
- join
- resolve
- resolveOrCreate



Next to the default Handlebar helpers, like the if statement, StackState has support for several helpers. We talked about resolving internal node ids, the if statement, and the join helper a bit already. Let's have a look at each of the helpers available.

The 'add', 'concat', and 'identifier' helpers perform some kind of transformation based on the input. Helpers 'get', 'getFirstExisting', 'getOrCreate', 'resolve', and 'resolveOrCreate' are used to resolve internal node ids. Lastly, the 'join' helper is used for repetitions.

Reference:

https://docs.stackstate.com/develop/reference/stj/stj_reference#functions

Topology synchronization – component/relation templates

```
# add <numeric value> <numeric value> ...
```

Topology data:

```

11  "data": [
12    "SubnetId": "subnet-123",
13
14    "number1": 1,
15    "number2": 2

```

Component properties

Type	aws-subnet
Labels	number:3

Template:

```

12  "labels": [
13    {
14      "_type": "Label",
15      "name": "number:{{ add element.data.number1 element.data.number2 }}"
16    }

```

StackState

The add helper adds together all numeric values that are passed as arguments to the helper.

In this example, we have two numeric values in the topology data. We want to add these values together and put the resulting value on a label. The template is shown at the bottom. We pass the references to the data as arguments to the add helper. Values for number1 and number2 are passed here. The result would be “number:3”, as can be seen on the right.

Topology synchronization – component/relation templates

```
# concat <string value> <string value> ...

Template:
5   "name": "{{ concat element.data.SubnetId " - " element.data.Location.AwsRegion }}",
```

Topology data:

```
11  "data": [
12    "SubnetId": "subnet-123",
13    "Location": [
14      "AwsAccount": "123456",
15      "AwsRegion": "eu-west-1"
16    ],

```

The screenshot shows the StackState interface with a component named "subnet-123 - eu-west-1" under the "Component properties" tab. The component has a type of "aws-subnet". Arrows from the code snippets point to the "SubnetId" and "AwsRegion" fields in the topology data, and another arrow points to the resulting component name.



The concat helper concatenates together the values of all arguments to return a single string.

The example shown here concatenates three values together to construct the name of the resulting component. The first value is obtained from the available topology data, a field called 'SubnetId'. The second argument passes a string literal, to format the output a bit. The last argument also provides information from the topology information, an AWS region.

On the left we see an example of the available topology data. We see the, in the concat example, referenced keys SubnetId and AwsRegion being available here. The values for these keys are obtained before the concat helper is executed. The outputted component's name can be seen on the right.

You could construct the name without the use of the concat helper since the name key expects a string value, however this demonstrates nicely what the helper does. We'll see a more complex example in a minute

using a combination of helpers.

Topology synchronization – component/relation templates

```
# get <identifier>
# get <identifier> Type=<type>;Name=<name>
```

Check function:

Name	Type	Required
metrics	Metric stream	<input checked="" type="checkbox"/>
state	State	<input checked="" type="checkbox"/>

Identifier

urn:stackpack:common:check-function:metric-fixed-state

```
"checks": [
  "function": {{ get "urn:stackpack:common:check-function:metric-fixed-state" }},
  "arguments": [
    "parameter": {{ get "urn:stackpack:common:check-function:metric-fixed-state" "Type=Parameter;Name=state" }},
  ],
  ...
]
```

 StackState

The get helper resolves internal node ids by providing an URN identifier as argument.

The argument passed to the get helper is an URN identifier. This can be an URN identifier of a component type, layer, domain, or some other configuration that is available in StackState. The helper return the internal node id required by certain keys in a component or relation template.

We've already seen the get helper being used in the example to explain the check JSON block. The example show here is the same example. We can see the get helper being used to reference the check function by the check function's URN identifier. We also needed to reference the parameters of the check function in the checks JSON block. The get helper has a variant that can be used to reference the required information in a nested way. In other words, the get helper will resolve the URN identifier first. Then, the argument's type and name are resolved within the scope of the found configuration with the URN identifier. In this case, the check function's parameter named state is resolved in the

context of the check function with the shown URN identifier.

Topology synchronization – component/relation templates

```
# getFirstExisting <identifier> <identifier> ...
```

Component template:

```

7   "domain": {{ getFirstExisting
8     (concat "urn:stackpack:common:domain:" element.data.my-domain)
9     "urn:stackpack:aws:shared:domain:aws"
10    )}},

```

The diagram illustrates the process of topology synchronization. It starts with a code snippet of a component template. An arrow points from this template to an 'Edit domain' form. This form has fields for Name (AWS), Description (e.g. Infrastructure for region 1), Order (0), and Identifier (urn:stackpack:aws:shared:domain:aws). Another arrow points from the 'Edit domain' form to a 'Resulting component' table. The table shows a single row with columns: Layer (Domain), Environment(s) (AWS), Networking (Production), and AWS.

Layer	Environment(s)	Networking
Domain	AWS	Production

StackState

The `getFirstExisting` helper is similar to the `get` helper. Out of a provided list of URN identifiers, it returns the first internal node id that can be resolved. This helper is useful when you need to generate an URN identifier out of topology data but you don't know beforehand if the generated URN identifier exists. You can use the second, or more, arguments as fallbacks.

Consider the fallback scenario where we want to set a component's domain based on the topology data. We provide a part of the URN identifier in the topology data. We concatenate the input together with a string such that the resulting URN identifier is a valid identifier, as shown on line 8 of the example component template shown on the top. The `getFirstExisting` helper will try to resolve the concatenated URN identifier first. Assume that there is no domain available that matches the provided URN identifier. The second argument is then tried. In this case, the second argument's URN identifier can be resolved. The resulting domain of the component will be set to what was resolved as the second argument to the `getFirstExisting` helper. The domain and its identifier is shown in the middle and the resulting component's domain is shown on

the right.

Topology synchronization – component/relation templates

```
# identifier <namespace> <type> <name>
```

```
identifier "urn:stackpack:common" "Environment" element.data.Tags.[stackstate-environment]
```

```
Example result: "urn:stackpack:common:environment:production"
```



The identifier helper can be used to construct valid URN identifiers.

The first argument defines the URN identifier namespace, the second argument defines the configuration, or node, type, and the last argument defines the name of the configuration. The result of the identifier helper is a valid URN identifier constructed by its inputs.

The example shown here constructs an identifier as used in the StackPack. The URN identifier is going to be a URN reference of a StackState Environment in the stackpack:common namespace. The last part of the identifier is set to the name of the environment and is provided by the value that is available as topology data.

Topology synchronization – component/relation templates

```
# getOrCreate <identifier> <create identifier> Type=<type>;Name=<name>
```

Component template:

```
3   "environments": [
4     {{#if element.data.Tags.[stackstate-environment]}}
5       {{getOrCreate
6         (identifier "urn:stackpack:aws:shared" "Environment" element.data.Tags.[stackstate-environment])
7         (identifier "urn:system:auto" "Environment" element.data.Tags.[stackstate-environment])
8         (concat "Type=Environment;Name=" element.data.Tags.[stackstate-environment])
9       )}}
10      {{else}}
11        {{get "urn:stackpack:common:environment:production" }}
12      {{/if}}
13    ],
]
```



The `getOrCreate` helper resolves internal node ids by providing an URN identifier as argument, just like the `get` helper. The `getOrCreate` helper can create simple types when the provided URN identifier is not available in StackState. It tries to resolve the provided URN identifier. If this URN identifier is found, the node id is returned, just like the `get` helper. If the URN is not found, a new configuration is created dynamically using the provided arguments. The newly created node id is then returned. Creating configuration only works for simple types, like layers, domains, environments.

The first argument is the URN identifier that is tried to resolve. The second argument defines the URN identifier that needs to be created if the first argument's URN identifier was not found. The third argument defines what type of configuration needs to be created and the name it should use. The created configuration will not have a lock.

Consider the following example. In this example we define a component's environment based on the received topology data. On line 4 we check

whether the key 'stackstate-environment' is present in the Tags map. If the key is present, the line 5 until line 9 is executed. If the key is not present, line 11 will be executed. Line 11 acts as a fallback, where the common StackPack's production environment is resolved using the get helper. Line 5 defines the getOrCreate helper. The first argument, on line 6, uses the identifier helper to define the URN identifier using the value of the 'stackstate-environment' variable. If the URN identifier is present in StackState, the Environment is resolved and processing stops. If the URN identifier constructed on line 6 is not present, a URN identifier is generated for the to create Environment on line 7. Line 8 defines the type and name of the to be created configuration. In this case, the config is an Environment and its name is going to be set the value of the 'stackstate-environment' variable. The Environment is created and the new internal node id is returned. The parentheses around line 6, 7, and 8 are there to group together the statements to help StackState determine which argument belongs to which helper.

Topology synchronization – component/relation templates

```
# join <iteratee> "<separator>" "<prefix>" "<suffix>"
```

Template:

```

12   "labels": [
13     {
14       "_type": "Label",
15       "name": "stackpack:aws"
16     }
17     {{# if element.data.labels }}
18       {{# join element.data.labels ", " ","}}
19       {
20         "_type": "Label",
21         "name": "{{this}}"
22       }
23     {{/ join}}
24   {{/if}}
25 ],

```

Topology data:

```

3 def element = StructType.wrapMap([
4   "type": [
5     "id": 88638427519583,
6     "lastUpdateTimestamp": 1619448593965,
7     "name": "aws.subnet",
8     "model": 75174337221170,
9     "_type": "ExtTopoComponentType"
10   ],
11   "data": [
12     "SubnetId": "subnet-123",
13     "labels": ["my-label", "another label"]
14   ],
15   "externalId": "subnet-123",
16   "identifiers": [
17     "subnet-123"
18   ]
19 ])

```

Result:

Component properties	
Type	aws-subnet
Labels	my-label stackpack:aws another label

```

"labels": [
  {
    "_type": "Label",
    "name": "my-label"
  },
  {
    "_type": "Label",
    "name": "stackpack:aws"
  },
  {
    "_type": "Label",
    "name": "another label"
  }
],

```

The join helper loops over a list or a map and joins together the elements. It prints the contents you provide. It prints a separator between each element of the array. It also offers to print a prefix before the first element and a suffix after the last element. The separator is mandatory. The prefix and suffix are optional. You can set the separator to an empty string if you need to. Let's explain the join helper in more detail using the labels template block.

This example builds on the previous example where we had a static label set, where the label was set to "stackpack:aws". The addition to the example is that labels are added dynamically based on the input.

On the left, labels block of the template is shown. Lines 13 until line 16 show creates the static label "stackpack:aws". Lines 17 until 24 make it more interesting. Line 17 defines an if statement that checks whether "element.data.labels" is present on the topology element's data. The if statement ends on line 24. If this statement is true, then line 18 is executed. Line 18 defines a join helper. It iterates the provided list and makes the value available within the join's context as a variable called

“this” wrapped in double curly brackets. The join helper just prints the text between lines 19 until 22, which holds the definition of a label. The name of the label is dynamically set the value of the “this” variable for each iteration. Essentially, we are looping over each value in the labels array and converting that to Label definitions.

The join helper prints the text separated by a comma, as defined by the second argument to the join helper. The separator between each printed label definition ensures the resulting JSON is valid. The last argument defines a prefix. The prefix, a comma, is printed before the first entry is printed. We need the prefix because line 16 does not contain a comma. We cannot put a comma on line 16 as we need to account for the case where there are no labels defined in the topology data.

There is also a fourth argument. The fourth argument defines a suffix and is optional. The suffix is printed after the last entry in the array has been processed.

The topology data available is shown on the top right. As you can see that line three defines the element. Line 11 defines the data payload and line 13 defines the labels payload. So, ‘element.data.labels’ refers to line 13 here. Since there is a labels payload, the if statement is passed and the join helper does its processing on each string in the labels array. The result is shown on the bottom right. The labels are shown on the component details pane in this example. Also, the resulting JSON is shown on the far right. Valid JSON was generated.

Topology synchronization – component/relation templates

```
# resolve <type> <name>
```

Template:
 "dataSource": {{ resolve "DataSource" "Demo data source" }},

StackState

The `resolve` helper resolves internal node ids based on the configuration's name. The first argument defines the type of configuration that needs to be resolved. The second argument defines the name of the configuration that needs to be resolved.

The example given here resolves the a telemetry data source. The telemetry data source is resolved on its name. In this case, there is a `DataSource` with the name 'Demo data source'.

StackState allows configuration names to be duplicate. It's also fine if two different types of configuration has the same name. There may be duplicate configurations found with the same name, even if the configuration is of the same type. The `resolve` helper will arbitrary select one and will print a warning in the StackState log file.

The `resolve` helper is similar to the `get` helper in that it resolves configuration. It is advised to use the `get` helper in future work since the majority of configuration has a URN identifier set.

Topology synchronization – component/relation templates

```
# resolveOrCreate <type> <name>
# resolveOrCreate <type> <name> <default>
```

Component template:

```
56   "layer": {{ resolveOrCreate "Layer" element.data.layer "Auto-synced Components" }},
57   "domain": {{ resolveOrCreate "Domain" element.data.domain "Auto-synced Domain" }},
58   "environments": [
59     {{ resolveOrCreate "Environment" element.data.environment "Auto-synced Environment" }}
60   ]
```



Similar to the resolve helper, the resolveOrCreate helper resolves internal node id based on the configuration's name. If no configuration is found with the given name, configuration will be added.

The resolveOrCreate's first argument defines the type of configuration it needs to resolve. The second argument accepts a String value, the name of the configuration. The third, and last argument accepts a String value and defines the name of the configuration. If the requested configuration was not found, the configuration is created where the name will set to what was provided in the second argument. This only works for simple types, like component types, layers, domains, etc. Usually the second argument is dynamic, based on topology data. If the key defined by the second argument is not present, a fallback can be used as defined by the third, optional, argument.

In the example, lines 56, 57, and 59 define a component's layer, domain, and environment, respectively. All three use the resolveOrCreate helper to resolve internal node ids. Let's examine line 56, the layer case. The first argument defines that a StackState Layer is to be resolved. The

second argument is dynamically set to the value of the data.layer key coming from the topology information. If this key exists on the topology information, a layer with this value is resolved. If the key is not present, then the default layer will be resolved, called “Auto-synced Components”. It works similarly for the domain and environment.

The resolveOrCreate helper is similar to the getOrCreate helper in that it resolves configuration and creates new configuration if necessary. It is advised to use the getOrCreate helper in future work since the majority of configuration has a URN identifier set.

Topology synchronization – component/relation templates

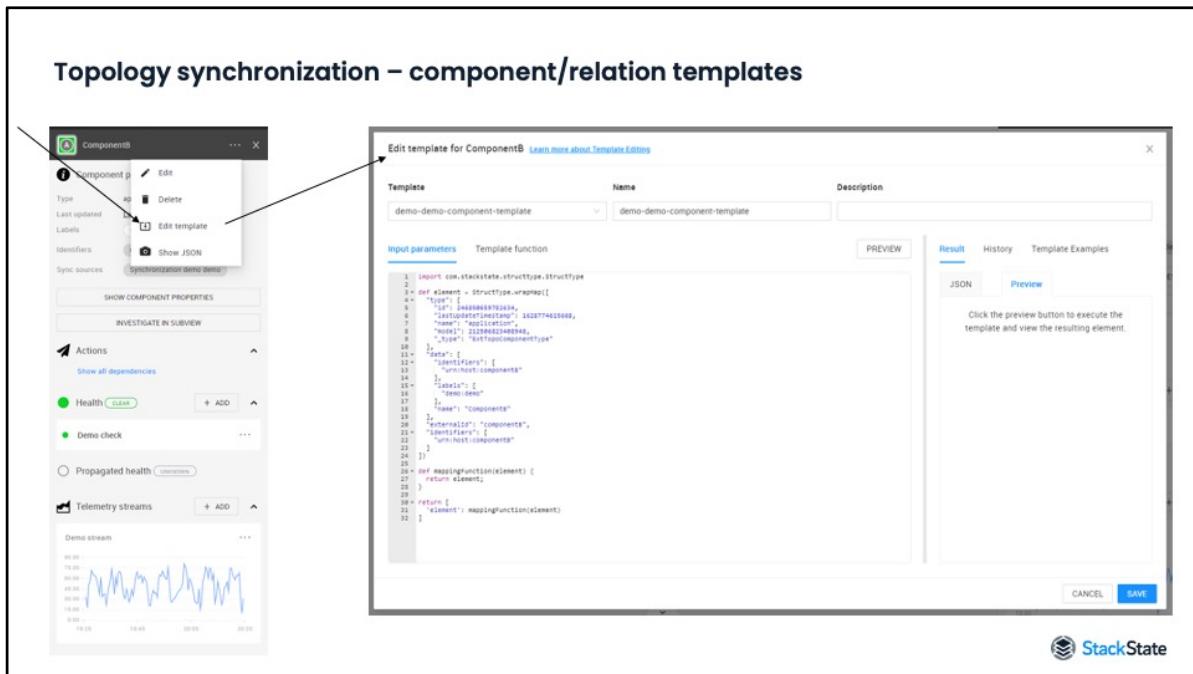
Synced component

Manually added stream and check to get the JSON definitions needed.

StackState

Now that we covered templates and helpers to construct valid JSON, how do we obtain the JSON definitions to begin with. Surely, we don't want to type everything ourselves.

You can make alterations to synchronized or unsynchronized component or relation to obtain the definitions you need in your template. In this example, we created a telemetry stream and a health check using that stream. This is an already synced component. The story here is to add the stream and check to the already in use component template.



On the top right of the component or relation's details pane, you see three dots, shown on the left here. The pop-up menu will show you the menu option 'edit template'. This will open a dialog. The dialog shows the in-use template at the top left. This shows the template that was used to create the component. Below input parameters, the topology data is shown as received and processed by topology synchronization, the information available to the template. If a mapper function was used in the creation of this component, the code is also included here. The tab 'Template function' will show the actual template contents of the selected template. You can press on the button preview to generate a preview of the resulting component or relation.

The topology data of the selected component can be used to test the template. The preview function is very useful here. If you want to change the template, you could alter the input data and/or the template, press preview, and see the results of the changes without the need to synchronize the topology data. Preview enables you to make rapid iterations until you are finished. If you click on 'save', the template is updated. There is no need to reset the synchronization. The

synchronization's cache is cleared and the updated template will be used to update the components or relations once new data is received. Data will not be replayed; you must wait for new data to see the changes made to the template to reflect on the components or relations affected.

Let's continue with our adding of a stream and check scenario.

Topology synchronization – component/relation templates

```

1: {
  "type": "component",
  "checks": [],
  "streams": []
}
3: [
  {
    "name": "{{element.data.name}}"
  }
]
5: [
  {
    "label": "{{element.data.labels}}"
  }
]
7: [
  {
    "type": "label",
    "name": "{{this}}"
  }
]
9: [
  {
    "type": "check"
  }
]
11: [
  {
    "type": "stream"
  }
]
13: [
  {
    "type": "label",
    "name": "demo-item"
  }
]
15: [
  {
    "tags": "{{element.data.tags}}"
  }
]
17: [
  {
    "type": "label",
    "name": "{{this}}"
  }
]
19: [
  {
    "type": "stream"
  }
]
21: [
  {
    "type": "label",
    "name": "{{this}}"
  }
]
23: [
  {
    "type": "stream"
  }
]
25: [
  {
    "type": "label",
    "name": "{{element.data.name}}{{element.data.name}}{{element.externalId}}{{element.type.name}}"
  }
]
27: [
  {
    "type": "label",
    "name": "{{element.data.description}}{{element.type.name}}"
  }
]
29: [
  {
    "type": "label",
    "name": "{{element.data.description}}{{element.type.name}}"
  }
]
31: [
  {
    "type": "label",
    "name": "{{element.data.name}}{{element.type.name}}"
  }
]
33: [
  {
    "type": "label",
    "name": "{{element.data.name}}{{element.type.name}}"
  }
]
35: [
  {
    "type": "label",
    "name": "{{element.data.name}}{{element.type.name}}"
  }
]
37: [
  {
    "type": "label",
    "name": "{{element.data.name}}{{element.type.name}}"
  }
]

```

Here's the template that was used to create this component.

The template has empty JSON arrays for both checks and streams. Even while we added the stream and health check manually to the component, the blocks remain empty. The template shown here is the template that was used to create the component, so, there were no checks and streams to begin with. If you click on preview, you won't see the manually created stream and check. We need to add the definitions of the stream and the check here to persist the changes. So, where to find the definitions of the stream and check?

Topology synchronization – component/relation templates

The screenshot shows two overlapping dialog boxes. The top dialog is titled 'Edit template for ComponentB' and has tabs for 'Template', 'Name', and 'Description'. The 'Template' tab is selected, showing a dropdown menu with 'New template' and 'demo-demo-component-template' selected. The 'Name' tab shows 'ComponentB' and the 'Description' tab is empty. Below this is a preview area with the heading 'Check definition' containing JSON code for a check definition. Arrows point from the 'Template' dropdown in the top dialog to the 'Template' tab in the preview area, and from the 'Template' tab in the preview area to the 'Check definition' code. The bottom dialog is also titled 'Edit template for ComponentB' and has tabs for 'Input parameters' and 'Template function'. The 'Template function' tab is selected, showing a large block of JSON code. Arrows point from the 'Template function' tab in the bottom dialog to the 'Template function' tab in the preview area, and from the 'Template function' tab in the preview area to the 'Stream definition' code in the bottom dialog. The bottom dialog also has tabs for 'PREVIEW' and 'StackState'.

On the top left of the dialog you have the drop-down selector to select the template. There is also the option called 'new template'. By clicking on this option, the definition of the current component is shown in the 'template function' tab.

You'll notice that the streams and checks JSON blocks are now populated with the definitions of the stream and check we created manually. You can copy the definitions from here and paste the definitions in the correct JSON blocks of the intended template. You can go back to the intended template by clicking on the drop-down again and paste the definitions into the correct locations. You can also edit the template via the settings pages, however, the edit template dialog offers a preview function to reduce templating errors. After pasting the definitions, you can make the stream filters and check dynamic by using handlebars. Be aware of the negative ids after copy/pasting from the new and/or other templates. After you're done editing and testing, you can update the template by pressing save to persist the changes.

The example we gave used an already existing template. You can use the 'new template' option from the drop-down menu to create a new template. The new template option generates the template contents based on the topology data it has available. You may find that some blocks are sub optimal, maybe handlebars templating was there before but is gone. Usually, it is easier to clone an existing template and make changes to reflect your use case. You can create or clone a template via the settings pages. Alternatively, you can alter the existing template and copy/paste the altered contents to the 'new template' option, and after giving the new template a meaningful name, press save to create a new template.

After creating a new template, don't forget to add a component or relation mapping in the synchronization. Otherwise, the template will not be used. At least one explicit mapping must be created.

Topology synchronization

Merge strategies



Let's talk a bit about merge strategies.

Topology synchronization – merge strategies

- Merge strategies handles merging between components
- Defined per synchronization mapping
- Strategies available:
 - Use mine only
 - Use theirs always
 - Merge, prefer mine
 - Merge, prefer theirs



Merge strategies in topology synchronization define how topology elements merge together.

Synchronizing a component or relation from a data source seems simple in concept. What happens to a component or relation when another component or relation is synchronized that has an overlapping identifier set. The two components or relations are merged. The merge process needs to make several decisions. As you can imagine, in case of a component, what should the name of merged component become? Should the name of the component already in StackState be used, or should the name provided by the newly received component be used? What happens to the telemetry streams on both components? What happens to the health checks on both components? All valid questions. Merge strategies define how to handle these cases. A merge strategy is defined per synchronization mapping.

StackState offers four merge strategies. The ‘use mine only’ strategy discards the information that is currently present in StackState, the current mapping is used. The ‘use theirs always’ merge strategy is the

opposite of the ‘use mine only’ merge strategy and discards all information from the current mapping leaving the current topology untouched. Merge strategies ‘merge, prefer mine’ and ‘merge, prefer theirs’ try to merge as much information as possible. In case of conflict, data of the current mapping is used when using the ‘merge, prefer mine’ merge strategy to resolve conflict. Data of what’s currently in StackState is used to resolve conflict when using ‘merge, prefer theirs’. Merge strategies ‘merge, prefer mine’ and ‘merge, prefer theirs’ are most used because these strategies try to preserve information.

We’ll cover each merge strategy in more detail.

Topology synchronization – merge strategies

Source Type	Actions	Mapping function	Template function	Merge Strategy
aws.kinesis	Create	AWS GENERIC COMP_X	KINESIS-STREAMS	Merge, prefer mine
aws.route53.domain	Create	AWS GENERIC COMP_X	ROUTE53-DOMAIN	Merge, prefer mine
aws.autoscaling	Create	AWS GENERIC COMP_X	AUTO-SCALING-GRO...	Merge, prefer mine
aws.s3.bucket	Create	AWS GENERIC COMP_X	S3-BUCKET	Merge, prefer mine
aws.lambda.alias	Create	AWS GENERIC COMP_X	LAMBDA-ALIAS	Merge, prefer mine
Other Sources	Create	AWS GENERIC COMP_X	AWS GENERIC COMP_X	Merge, prefer theirs

StackState

The merge strategies are defined on the component and relation mappings of a topology synchronization. Each mapping has a merge strategy defined.

Here's an example of several component mappings from an AWS synchronization. Merge strategy 'merge, prefer mine' is defined for the mappings. The default, 'Other sources', mapping has merge strategy 'merge, prefer theirs' defined. In case of AWS, the mappings have 'merge, prefer mine' defined since the information from is considered "more relevant" than information reported about the same component from another data source. This should not be a surprise; we are reporting on an AWS resource here. The contribution of another data source is not less important, it can add additional information. The merge strategy is set because of that reason. For example, the name of the AWS resource should be preferred over the name that was reported by another data source. However, if both report checks and telemetry streams, you want to have both on the merged component.

Topology synchronization – merge strategies

Topology synchronization A:

Component (from sync A)

- Type: server
- Version: 1.0
- Last updated: Today_18.11.21
- Labels: sync A
- Identifiers: unhost component
- Sync sources: Synchronization demo syncA

Actions

- Health: CLEAR
- Check (from sync A)
- Propagated health: UNKNOWN

Telemetry streams

- stream (from sync A)

Topology synchronization B:

Component (from sync B)

- Type: server
- Last updated: Today_18.11.21
- Labels: sync B
- Identifiers: unhost component
- Sync sources: Synchronization demo syncB

Actions

- Health: CLEAR
- Check (from sync B)
- Propagated health: UNKNOWN

Telemetry streams

- stream (from sync B)

StackState

Let's use the following example to illustrate each merge strategy.

In this example, we will use two topology synchronizations named "sync A" and "sync B". Each synchronization will synchronize only one component. These components will merge because the same identifier is used on both components. The component is shown here, in an unmerged state, from each synchronization. The unmerged state is shown here to show what properties each synchronization brings.

Synchronization "sync A" reports a component that is named "Component (from sync A)" and synchronization "sync B" reports on a component that is named "Component (from sync B)". Both report a similarly named label, telemetry stream, and health check. Synchronization "sync A"'s component reports a version and a description while "sync B"'s component does not have a version nor a description. For simplicity, synchronization "sync A" will synchronize before "sync B".

In the next slides we will be going over the different merge strategies

using the two synchronizations.

Topology synchronization – merge strategies

Merge strategy “use mine only”

The screenshot shows a component card for "Component (from sync B)" within an "Auto-sy... Domain". The component icon is a green square with a white square inside. The component properties dialog is open, showing the following details:

- Type: server
- Last updated: Today 10:12:20
- Labels: sync.B
- Identifiers: urn:host:component
- Sync sources: Synchronization demo syncA, Synchronization demo syncB
- Actions: Health (CLEAR), Check (from sync B)
- Telemetry streams: stream (from sync B)

A vertical sidebar on the left lists "Auto-sy... mponents". A legend on the right indicates that green arrows point to "use mine only" merge strategy details.

In this scenario, the merge strategy “use mine only” is used.

Since synchronization “sync A” synchronizes before “sync B”, the data from “sync B” is used for the merged component. The component from “sync A” is discarded. The “use mine only” merge strategy on the mapping discards what was already available in StackState. You can see here that the merged component only has one label, one telemetry stream, and one health check, all coming from “sync B”.

You can still see that the component comes from “sync A” and “sync B” by looking at the “Sync sources”, as seen on the top right. The information from both synchronizations is still available under “show component properties”.

Topology synchronization – merge strategies

Merge strategy “use theirs only”

The screenshot shows the StackState interface for managing component synchronization. On the left, a component named "Component (from sync A)" is listed under an "Auto-sy... Domain". On the right, a detailed view of this component is shown in a modal window. The modal includes fields for Type (server), Version (1.0), Last updated (Today, 19:08:42), Labels (sync A), Description (Description (from sync A)), and Sync sources (Synchronization demo syncA). In the Actions section, there are buttons for Health (green), Check (from sync A) (green), and Propagated health (yellow). The Telemetry streams section shows a single stream (from sync A). Arrows from the text "Merge strategy ‘use theirs only’" point to the "Labels", "Description", "Sync sources", "Actions", and "Telemetry streams" sections of the modal.

In this scenario, the merge strategy “use theirs only” is used.

Since synchronization “sync A” synchronizes before “sync B”, the data from “sync A” is used for the merged component. The component from “sync B” is discarded. The “use mine theirs” merge strategy on the mapping discards what comes from “sync B”. You can see here that the name of the component comes from “sync A”. Next to the component’s name, there is only one label, one telemetry stream, and one health check available on the component, all coming from “sync A”.

Topology synchronization – merge strategies

Merge strategy “merge, prefer mine”

The screenshot shows a component card in the StackState interface. The component is named "Component (from sync B)" and is located in an "Auto-sy... Domain". The component properties section shows "Type: server", "Last updated: Today, 19:39:11", and "Labels: sync B, sync A". The "Description" field contains "Description (from sync A)". The "Sync sources" field lists "Synchronization demo syncA" and "Synchronization demo syncB". The "Actions" section shows three items: "Health" (green), "Check (from sync A)" (green), and "Check (from sync B)" (green). The "Telemetry streams" section shows two streams: "stream (from sync A)" and "stream (from sync B)". The StackState logo is visible in the bottom right corner.

In this scenario, the merge strategy “merge, prefer mine” is used.

Since synchronization “sync A” synchronizes before “sync B”, the name of the component is set to the name coming from “sync B”. Since both synchronizations provide the name of the component, there is a conflict. The merge strategy prefers the data from “sync B” over “sync A”’s data.

There is no need to prefer one sync over another in case of labels, telemetry streams, and checks. The labels, telemetry streams, and checks are merged such that you have both sets on the resulting, merged, component.

Topology synchronization – merge strategies

Merge strategy “merge, prefer theirs”

The screenshot shows a component card in the StackState interface. The component is named "Component (from sync A)". It has a "server" type, version 1.0, and was last updated at 19:41:55. The labels include "sync B" and "sync A". The description is "Description (from sync A)" and the identifiers are "Sync host component". The sync source is "Synchronization demo syncA". In the "Actions" section, there are three green status indicators: "Health (from sync A)", "Check (from sync B)", and "Check (from sync A)". Below these are sections for "Telemetry streams" and "Streams", each containing two entries: "stream (from sync A)" and "stream (from sync B)". Arrows from the left margin point to the component name, the type field, the last updated field, the description, the identifiers, the sync source, the health status, the check status, the telemetry streams, and the streams.

In this scenario, the merge strategy “merge, prefer theirs” is used.

Since synchronization “sync A” synchronizes before “sync B”, the name of the component is set to the name coming from “sync A”. Since both synchronizations provide the name of the component, there is a conflict. The merge strategy prefers the data from “sync A” over “sync B”’s data.

Similar to “prefer, merge mine”, the labels, telemetry streams, and checks are merged such that you have both sets on the resulting, merged, component.

Topology synchronization

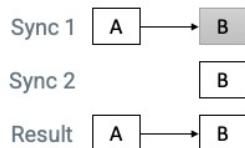
Mapping actions



Let's have a look at a synchronization's mapping actions.

Topology synchronization – mapping actions

- Defines how the sync should materialize the component, to show or not
- Actions:
 - Create
 - Create placeholder
- Create a placeholder component to merge in the future with a component from another data source with overlapping identifier(s)
- Use case examples:
 - Network connections
 - Relation to a component from another data source



 StackState

In synchronization, there are mappings for components and relations. These mappings are there to tell synchronization which template to use when a new topology element is processed. In the end, the templates create the component, or relation, you see on the visualizer. Each mapping has an action defined. By default this is set to 'Create'. The mapping action 'Create' is the default action and uses a template to materialize a component or relation. The other option is 'Create placeholder'. This option does not use a template. Actually, the action Create placeholder does not let the input topology element materialize into a component or relation. For all topology elements the id extractor is still run, causing the external ID and identifiers to be extracted and stored. The topology element is still stored, but locally in the so called external topology. The external topology is local to the data source only, just like the external ID.

The create placeholder action offers some option for certain use cases. If another synchronization happens to be materialized a component and this component has an overlap in the identifiers with a placeholder component then these components will merge together. The placeholder

component's information is then shown on the materialized component.

A use case might be that you have two data sources, two synchronizations, and you want to have a relation between components in the two data sources. However, you don't want to show the relation when the source or target component is not available in the other data source. We cannot create a relation based on external IDs only because external IDs are not shared among different synchronizations. In order to make this work, we need to merge together either the source or target component. Here, One data source needs to define the relation and report both components. The other data source can then define only the source or target component.

One data source should know both sides of the relation, otherwise how can you create a relation?

The example is shown here.

- Sync 1 reports component A and component B with a relation between the two components.
- Sync 2 reports target component B only.

If component B from sync 1 uses the mapping action 'Create' , then the component and relation will always be materialized.

In some use cases this is not desirable, we may only want to show the relation when component B is reported by Sync 2.

- If the mapping action for Sync 1 uses 'Create placeholder' for Component B, Component B will not materialize and the relation between component A and component B will not be visible.
- If the mapping action for Sync 2 is set to 'Create' for Component B, when component B is synchronized in Sync 2 it will materialize and component B from sync 1 and sync 2 will be merged together, causing the relation reported by Sync 1 to become visible.

Topology synchronization – mapping actions

In a synchronization, the actions are listed on the components and relations mapping pages. Each mapping has a mapping action.

In a synchronization, the actions are listed on the components and relations mapping pages. Each mapping has a mapping action.

Topology synchronization – mapping actions



 StackState

As you can see here, the mapping for “host” components has the mapping action “Create placeholder” set. Consequently, the mapper function, template function, and merge strategy fields are disabled for that mapping. The mapping for ‘other sources’ is set to “Create”, here the fields are not locked.

Topology synchronization

Topology snapshotting



Next topic is topology snapshotting. Topology snapshotting controls how StackState handles the topology payloads it receives.

Topology synchronization – topology snapshots

- Incremental building/sending topology information or, provide snapshot of whole topology
- Incremental relies on expiry, as can be specified in the data source, to be removed after expiry
- Use case: StackState Agent host process reporting

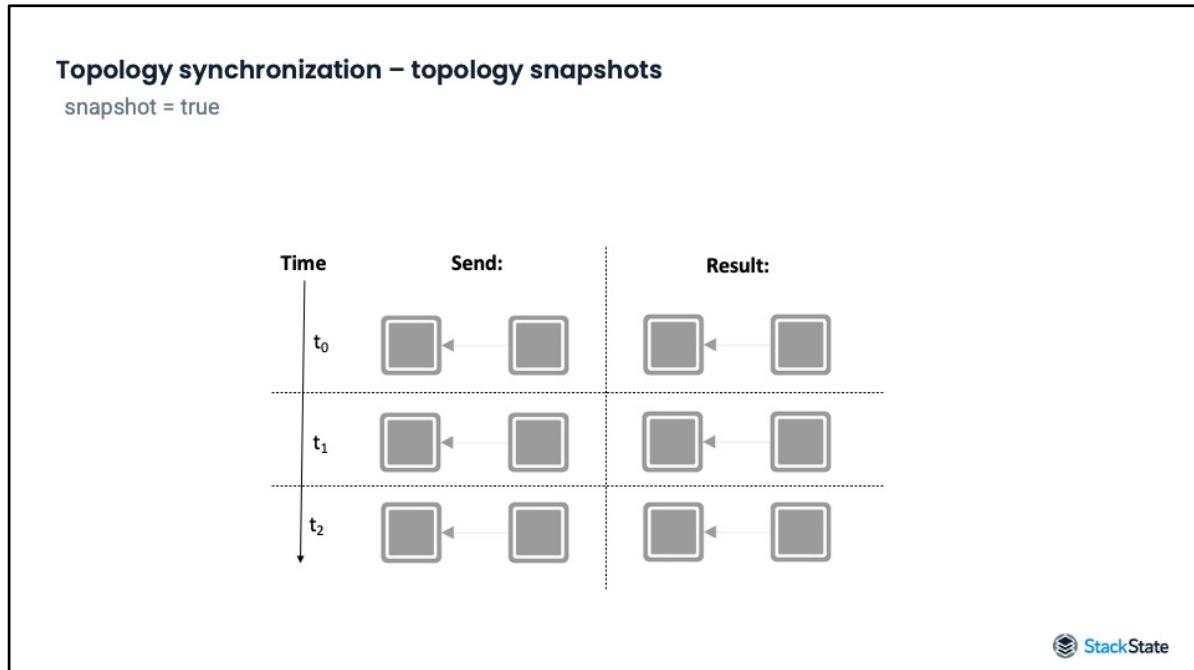


Topological information is send to the StackState receiver. The nature of collecting information may be different for use cases and data sources. You might be able to collect topology information in one go. This may be considered a snapshot of the entire topology. The actual state of the topology data source is collected in one payload. After a period of time the next collection run may be executed. The collected information is then send again to StackState. StackState will update the topology accordingly, since it is told that the payload is a topology snapshot. Any component or relation that was there in the first batch, but not in the second, will be removed from the visualization. The majority of integrations use topology snapshots.

There are some use cases where the snapshot cannot be formed, but has to rely on sending topology updates to StackState. StackState will update components accordingly to the updates it receives. The StackState Agent is one example, the agent collects information about host processes. Host processes come and go. The agent sends updates to StackState. Relying on topology updates, it is hard to determine when and if a component or relation should be removed. For these situations,

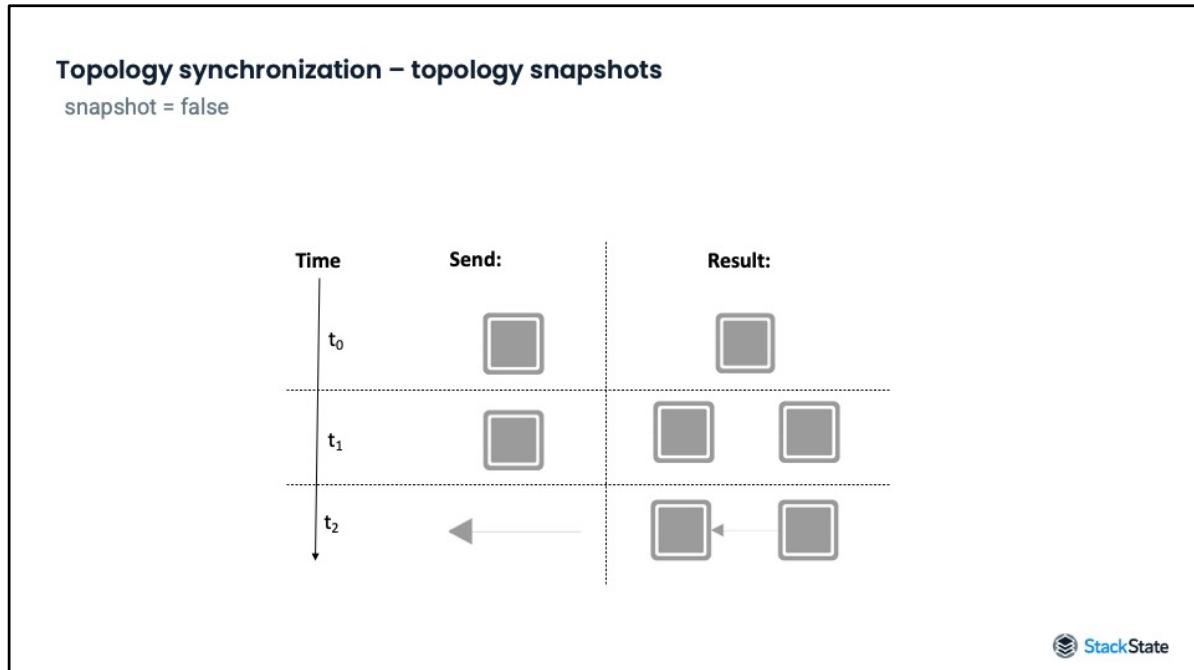
StackState can expire and clean up topology elements. The expire is set on the topology data source. After a certain amount of time, the component or relation is expired and is removed from the visualization. You can still find the component when you go back in time, of course. The expiry is reset if the component or relation every time an update is received. Next to expiry, it is also possible to explicitly send external IDs of components that have to removed from a synchronization.

Snapshotting is local to a single topology synchronization. One snapshot on one synchronization will not interfere with a snapshot processed on another synchronization.



Here's an example of topology snapshotting behavior over time. Time is shown vertically here. The send column shows what topology information is send to StackState and the result column shows the materialized result in StackState.

At time t_0 , two components and a relation is send to StackState. Since snapshotting is set, this is also the result. The same goes for time t_1 and t_2 ; the same payloads are sent. StackState handles these payloads as updates to the components. If a component was added, a third component would show up without removing the existing topology. If the relation was removed from the payload, only two components would have been sent to StackState and the result would reflect just that.



Here's an example where snapshotting is set to false in the payloads send to StackState. At t_0 a component is send to StackState. At t_1 another, different, component is send to StackState resulting in two components being shown. Finally, at t_2 , only a relation is send to StackState in the payload.

Topology synchronization – topology snapshots

Edit Sts data source

Name: Agent
Description: e.g. mesos cluster price for dc1

Use StackState's default: On

Instance Type: process

Topic: sts_topo_process_agents

Maximum batch size: 200

Expire elements: On
Expire after (minutes): 15

Identifier: um_stackpack.stackstate-agent-v2-shared-data-source-agent

REFRESH TEST CONNECTION CANCEL UPDATE

Edit Sts data source

Name: dataSource1
Description: DataSource that was generated for it.s.

Use StackState's default: On

Instance Type: d

Topic: sts_topo_d_a

Maximum batch size: 200

Expire elements: Off

Identifier: um_stackpack.ds-topo-instance-2441ab0-1bf2-4fe9-91a2-a7739fc997cc-data-source-datasource-0-s

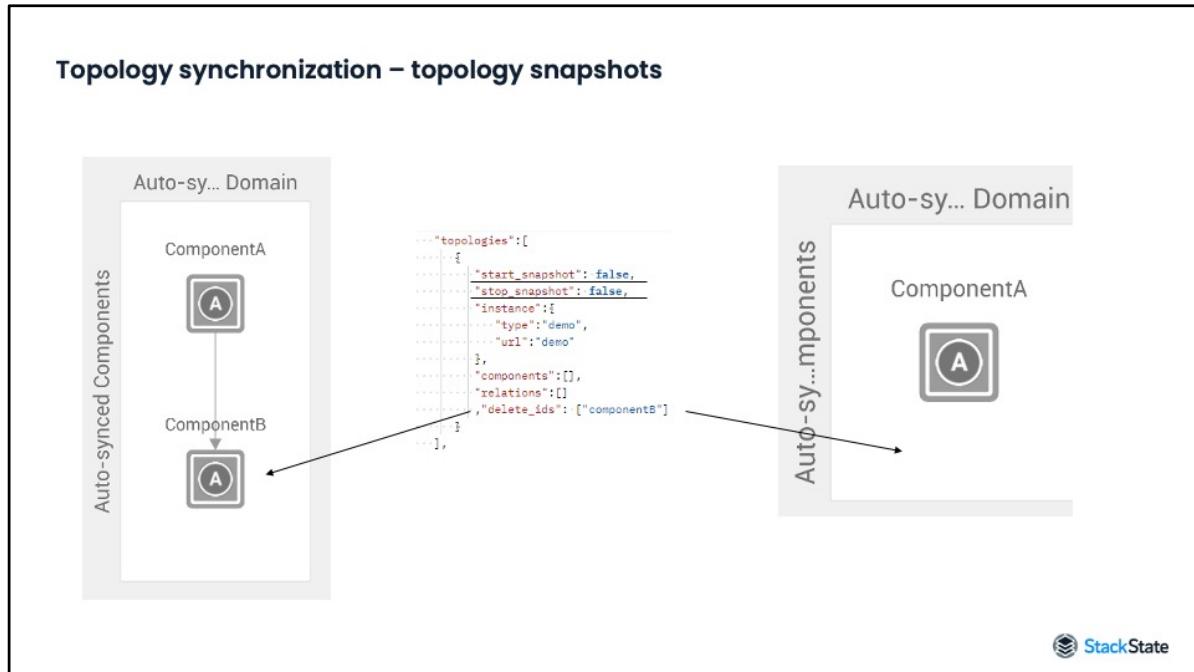
REFRESH TEST CONNECTION CANCEL UPDATE



Here we show two topology data sources to illustrate where expiry is configured.

On the left, the topology data source is shown that is part of the StackState Agent StackPack, which receives information about processes. In this case, the expiry is enabled and topology will expire after 15 minutes. After expiry, the topology is removed. It is possible to have stale data for a maximum of 15 minutes in this case.

On the right, the topology data source is shown that is part of the custom synchronization StackPack. Like most StackPacks, the topology collection is based on snapshots. Using snapshots removes the need of expiry since a new topology snapshot will remove old topology. You may have stale topology between collection runs. Having the combination of expiry and snapshotting may lead to flaky topology.



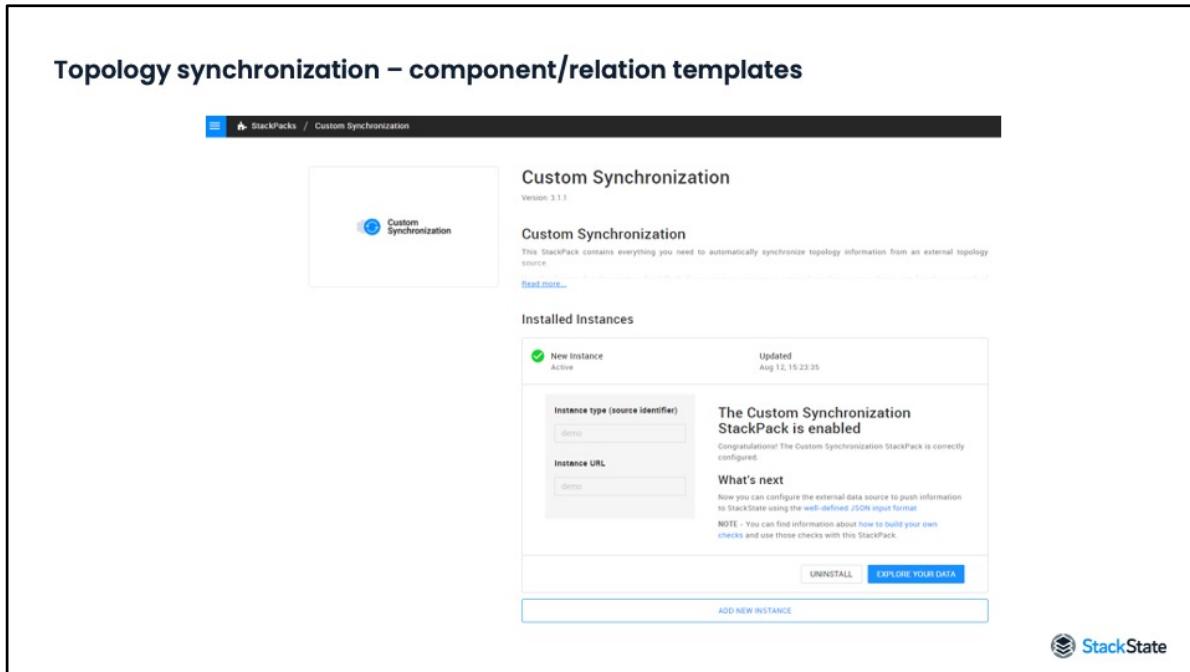
Next to expiry of components, it is possible to explicitly send external IDs to the StackState receiver for removal.

In the example shown here, there are two components and a single relation that were formed where snapshotting was set to false, as shown on the left. Instead of relying on expiry, we know that component B can be removed. We can send a payload to the StackState receiver, as shown in the middle, to explicitly remove component B. The result is shown on the right. The relation is deleted automatically after we explicitly removed component B.

Topology synchronization

Custom Synchronization StackPack walkthrough



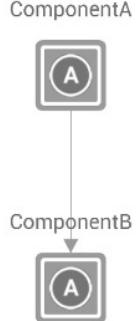


Let's have a look at a real-life example. A StackPack is available called Custom Synchronization StackPack. This StackPack can be used as a basis if you want to create your own StackPack. The Custom Synchronization StackPack contains a basic synchronization we can use in our example. We'll be sending JSON to the StackState receiver to show how the synchronization works.

As you can see here, the custom synchronization StackPack is already configured. The instance type and instance URL is set to demo. In any other StackPack, the instance type is usually the name of the integration. The instance type for the AWS StackPack is set to 'aws' for example. That removes the need to ask the user the instance type. A StackPack and the integration both know the instance type beforehand. The instance URL is the unique identification of the data source. In case of AWS, you have two accounts, the instance type would be set to 'aws', and the instance URL could well be a URL or the AWS account number. These values are used in the Sts data source, defining the name of the Kafka topic. In this example we set both to 'demo'. The Sts data source is used by the synchronization the StackPack creates.

We'll focus a bit more on components in this example. It is a similar process for relations.

Topology synchronization – component/relation templates



JSON:

```

    "start_snapshot": true,
    "stop_snapshot": true,
    "instance":{
        "type":"demo",
        "url":"demo"
    },
  
```

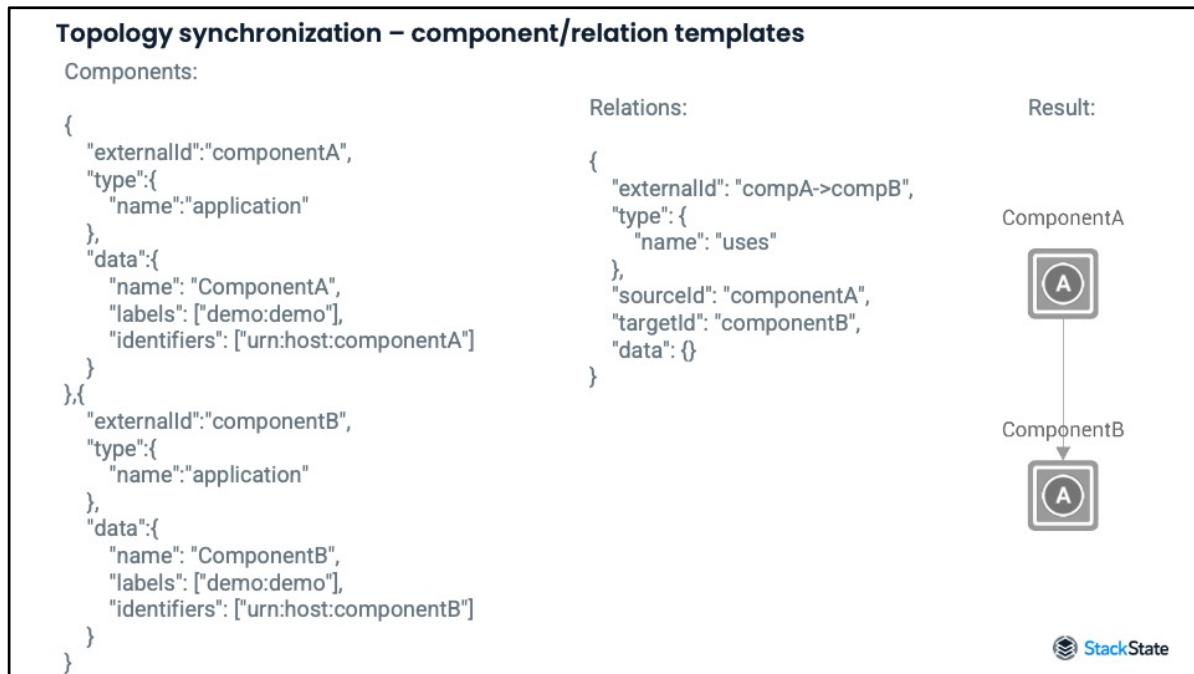


A part of the JSON that we are going to send to StackState is shown here. The idea is to send two components and a single relation. For topology synchronization, we define that we send a snapshot of topology. The start_snapshot and stop_snapshot boolean values take care of that. Setting both to false indicates that you are sending incremental updates only. It is possible send multiple JSON messages while still maintaining a single snapshot. This can be done by setting the start/stop snapshot values in the first message to true/false, the second and subsequent messages to false/false, and ending with false>true.

There is JSON object defining the instance the topology belongs to. This is the same instance type and instance URL we used to install the StackPack. The StackState receiver receives this JSON payload, reads the instance type and instance URL values such that it knows on which Kafka topic to put the topology information. The StackPack already created the topic. This is the link between integration and the Sts data source, the Kafka topic. The synchronization uses the Sts data source. Integration is a big word in this example, it is a JSON payload send to the StackState receiver. An integration is then software that collects data,

generates a JSON payload, and send that payload to StackState.

Let's have a look at the JSON format for the topology elements.



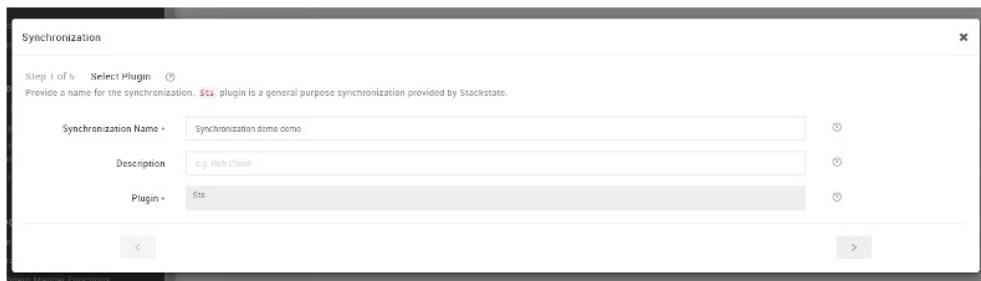
The JSON payloads for components and relations are shown here. These JSON payloads could have been generated by an integration.

For components, the JSON shown on the left defines two components. For each component, the externalId, type, and data keys are required to define a component. The external IDs “componentA” and “componentB” are defined here. The type of both components is set to “application”. The data block contains meta data of the components. There are no mandatory keys you have to set in the data JSON block. As you can see here, there is already a clear structure that the StackState synchronization can use. The name is already set in the data JSON block. There is also a list of labels and identifiers provided.

For relations, the JSON shown in the center of the slide defines a single relation. Similar to a component, a relation has an external ID, type, and data JSON blocks. Next to that, a relation defines a source Id and target Id, to reference two components to create a relation between. The source Id and target id references a component’s external ID.

Sending these JSON blocks, as part of the JSON StackState accepts, to the StackState receiver, you will see the result, as shown on the right.

Topology synchronization – component/relation templates



 StackState

The first step of the synchronization is shown here. The name is generated based on the input provided on during the installation of the StackPack. The plugin is set to the fixed Sts value, indicating that a Sts data source is configured for this synchronization.

Topology synchronization – component/relation templates

The screenshot shows the StackState Synchronization configuration interface. It is on Step 2 of 5, titled "Data Source And Query". The interface includes fields for "Choose source" (set to "DATASOURCE DEMO DEMO"), "Component Id Extractor" (set to "AUTO SYNC COMPONENT ID EXTRACTOR"), "Relation Id Extractor" (set to "AUTO SYNC RELATION ID EXTRACTOR"), and "Identifier" (set to "urn:stackpack:autosync:instance:1223e650-41e0-4c4f-0ee1-42d11d7c7f62:sync:synchronisation-demo-demo"). There are also "Start from earliest available topology data" and "Off" buttons. A "Next >" button is visible at the bottom right.

The second step in the synchronization is shown here.

The Sts data source used for this synchronization is defined here. Let's have a look at the Sts data source first.

Topology synchronization – component/relation templates

StackPack:

- New Instance Active
- Instance type (source identifier): demo
- Instance URL: demo

Edit Sts data source

- Name: DataSource demo demo
- Description: DataSource that was generated for demo demo
- Use StackState's default Kafka
- Instance Type: demo
- Topic: sts_topo_demo_demo
- Maximum batch size: 200
- Expire elements off
- Identifier: umcstackpack-autosync:instance:1323a650-41e0-4c4f-8ee1-42d11d7c7f62:data-source:datasource-demo-demo

JSON:

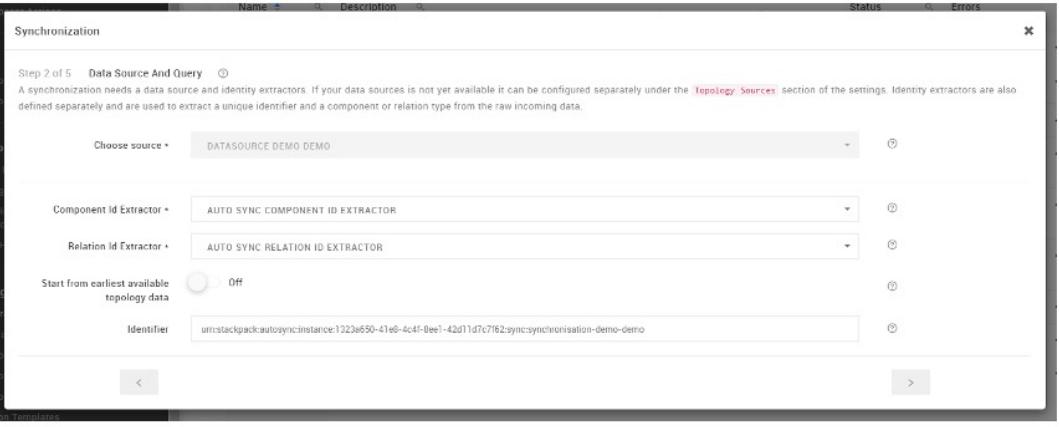
```
{
  "start_snapshot": true,
  "stop_snapshot": true,
  "instance": {
    "type": "demo",
    "url": "demo"
  }
}
```

Buttons: REFRESH, TEST CONNECTION, CANCEL, UPDATE, StackState logo

The configuration of the in-use Sts data source is shown here.

The most important part is the defined instance type and topic. You can see here that the topic is created by the StackPack has the format "sts_topo_demo_demo". The topic name is generated by the StackPack based on the input during installation of the StackPack. The topic is generated using format, which is the same for all StackPacks, sts_topo_<instance type>_<instance url>. The JSON also defines a type and URL. These are the exact same things. Essentially, the JSON instance and the topic name tell StackState which synchronization must process the received JSON.

Topology synchronization – component/relation templates



The screenshot shows the StackState topology synchronization configuration dialog. It is on Step 2 of 5, titled "Data Source And Query". The dialog includes fields for "Choose source" (set to "DATASOURCE DEMO DEMO"), "Component Id Extractor" (set to "AUTO SYNC COMPONENT ID EXTRACTOR"), "Relation Id Extractor" (set to "AUTO SYNC RELATION ID EXTRACTOR"), and an "Identifier" field containing the value "urn:stackpack:autosync:instance:1223e650-41e0-4c4f-0ee1-42d11d7c7f62:sync:synchronisation-demo-demo". There are also buttons for "Start from earliest available topology data" (set to "Off") and navigation arrows.

StackState

Back to the second step in the synchronization.

Next to the Sts data source, this step in the synchronization dialog also defines the id extractors to use. The id extractors extract the component and relation's external ID, type, and optional identifiers. Let's have a look at the component id extractor using the example JSON.

Topology synchronization – component/relation templates

JSON:

```
{
  "externalId": "componentA",
  "type": {
    "name": "application"
  },
  "data": {
    "name": "ComponentA",
    "labels": ["demo:demo"],
    "identifiers": ["urn:host:componentA"]
  }
}
```

Component ID extractor:

```

1 element = topologyElement.asReadOnlyMap()
2 externalId = element["externalId"]
3 type = element["typeName"].toLowerCase()
4 data = element["data"]
5
6 identifiers = new HashSet()
7
8 if(data.containsKey("identifiers") && data["identifiers"] instanceof List<String>) {
9   data["identifiers"].each{ id -
10     identifiers.add(id)
11   }
12 }
13
14
15 return Sts.createId(externalId, identifiers, type)
16

```

StackState

The component's JSON is shown on the left and the component id extractor is shown on the right.

Line three of the id extractor extracts the component's external ID. Line 4 extracts the component's type and lower cases the value it retrieved. Lower casing the type is quite common as it circumvents mapping mismatches.

Line 7 defines an empty Groovy HashSet that will temporary hold the component's identifiers. Line 9 defines an if-statement checking whether the component's data JSON payload contains a key called 'identifiers'. It also checks whether the value of key 'identifiers' is a list of strings. If this is the case, and it is, the 'identifiers' JSON array is iterated, one by one, adding the value to the identifiers HashSet that was defined on line 7. If the key 'identifiers' was not a JSON array of strings, no identifiers would have been added to the identifiers HashSet.

Line 15 defines the result of the id extractor; passing the external id, identifiers, and the component type to StackState.

Topology synchronization – component/relation templates

JSON:

```
{
  "externalId": "componentA",
  "type": {
    "name": "application"
  },
  "data": {
    "name": "ComponentA",
    "labels": ["demo:demo"],
    "identifiers": ["urn:host:componentA"]
  }
}
```

Here's the third step in the synchronization, the component mappings, together with the example JSON.

As you can see, there is no explicit mapping defined. Regardless of component type, the default component template will be used for the example component.

Imagine that there was an explicit mapping defined for the component type "application". The id extractor would have extracted the component type "application" from the JSON payload and would have matched against that mapping. The component template belonging to that mapping would be used.

Topology synchronization – component/relation templates

The screenshot illustrates the StackState topology synchronization process. On the left, a JSON payload for 'ComponentA' is shown, defining its name, type, and data. An orange arrow points from this payload to a portion of the component template on the right. The template uses Jinja2-style syntax to define the component's name based on its data. A blue arrow points from the template to the resulting component properties on the far right. These properties show the component is of type 'application', was last updated today at 20:26:40, has the label 'demo:demo', and has the identifier 'urn:host:componentA'. The StackState logo is visible in the bottom right corner.

```

39   "name": "demo:demo"
40   {{#if element.data.tags }}
41     {{# join element.data.tags ", " " "}}
42   {
43     "type": "Label",
44     "name": "{{ this }}"
45   }
46   {{</join>}}
47   {{</if>}}
48 },
49
50 "name": "{{#if element.data.name }}{{ element.data.name }}{{else}}{{ element.externalId }}{{/if}}
51 {{#if element.data.description}}
52   "description": "{{ element.data.description }}"
53 {{/if}}
54   "type" : {{ resolveOrCreate "ComponentType" element.type.name "Auto-synced Component" }},
55   "version": "{{ element.data.version }}",
56   "layer": {{ resolveOrCreate "Layer" element.data.layer "Auto-synced Components" }},
57   "domain": {{ resolveOrCreate "Domain" element.data.domain "Auto-synced Domain" }},
58   "environments": [
59     {{ resolveOrCreate "Environment" element.data.environment "Auto-synced Environment" }}
60   ]
61 }
62

```

ComponentA
Component properties

Type	application
Last updated	Today 20:26:40
Labels	demo:demo
Identifiers	urn:host:componentA
Sync sources	Synchronization demo demo

StackState

Let's have a look at a part of the component template, next to the component's JSON payload. On the left, the input JSON is shown. On the top right a part of the component template is shown. The properties of the resulting component is shown at the bottom right.

Line 50 of the component template defines the resulting component's name. An if-statement checks whether the key 'name' is present in the data JSON block. If there is, and it is, the name is used as the component's name.

Line 51 of the component template defines an if statement checking the presence of a 'description' field in the data JSON block. There is no 'description' key available in the example JSON. In that case, the description is not added to the resulting component. Similarly, there is no explicitly defined layer, domain, nor environment present in the JSON. The 'resolveOrCreate' helper on lines 56, 57, and 59 tries to find this key but is unable to. The helper will have to default to its last argument, the fallback.

Topology synchronization – component/relation templates

The screenshot shows a code editor with a component template and a StackState component properties interface.

Component Template (Left):

```

{
  "externalId": "componentA",
  "type": {
    "name": "application"
  },
  "data": {
    "name": "ComponentA",
    "labels": ["demo:demo"],
    "identifiers": ["urn:host:componentA"]
  }
}
  
```

StackState Component Properties (Right):

Component A (application)

Component properties	
Type	application
Last updated	Today 20:26:40
Labels	demo:demo
Identifiers	urn:host:componentA
Sync sources	Synchronization demo demo

Code Editor (Top Right):

```

28 +
29
30
31 +
32
33
34
35
36
37 +
38
39
40
41
42
43 +
44
45
46
47
48
49
  
```

Labels: [
 {{#if element.data.labels }}
 ({# join element.data.labels "," "" "," })
 {#_type: "Label",
 "name": "{{ this }}"
 }
 {{# join }}
 {{/if}}
 {
 "type": "Label",
 "name": "demo:demo"
 }
 {{#if element.data.tags }}
 ({# join element.data.tags "," "" "," })
 {#_type: "Label",
 "name": "{{ this }}"
 }
 {{# join }}
 {{/if}}
],
]
],
]

Here's another part of the component template, to illustrate how labels are created. Line 28 until 49 of the component template defines the labels of the resulting component. Multiple constructs are used here to define the component's labels.

Line 29 checks if there is a 'labels' present in the JSON data. In this case there is a 'labels' key present. Line 30 defines a join helper which iterates through the labels JSON array. All array elements will be separated by a comma, as defined by the first argument to the join helper. The prefix is set to an empty string, as defined by the second argument. After processing all elements, a comma is printed, as defined by the last argument to the join helper. Each element will be printed in format that is defined on lines 31 until 34.

Line 37 until 40 defines a single, fixed, label that is added to all components using this component template. This label was added to the component template during installation of the StackPack. The template was altered before the import was done using the input given during installation of the StackPack. The join helper needs to print a comma

before line 37 because otherwise you would end up with a template that yields invalid JSON.

The if-statement of line 41 will not find a 'tags' key in the JSON payload. No labels will be added from a 'tags' payload.

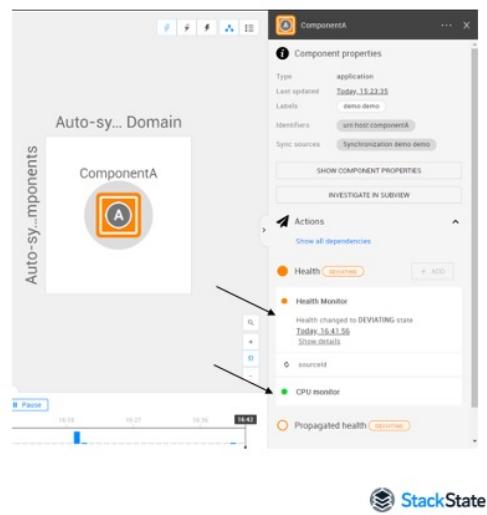
Health synchronization



Let's have a look into how StackState synchronizes health state information from different sources.

Health synchronization

- External monitoring systems calculate health states
- Synchronize health states from external monitoring systems
- StackState Agent integrations, CLI, or plain JSON
- Health streams are bound to topology elements using identifiers
- Difference between health synchronization and health checks is where the state is calculated.



Health synchronization synchronizes health states from external data sources. External monitoring systems derive health state information from the data they have available. External monitoring systems are usually already configured using a set of rules to determine health states based on the data available to them. There is no need to provide StackState with the same raw data and then have StackState derive the health state while the external data source has the health states readily available. For such data sources, StackState can synchronize the available health states.

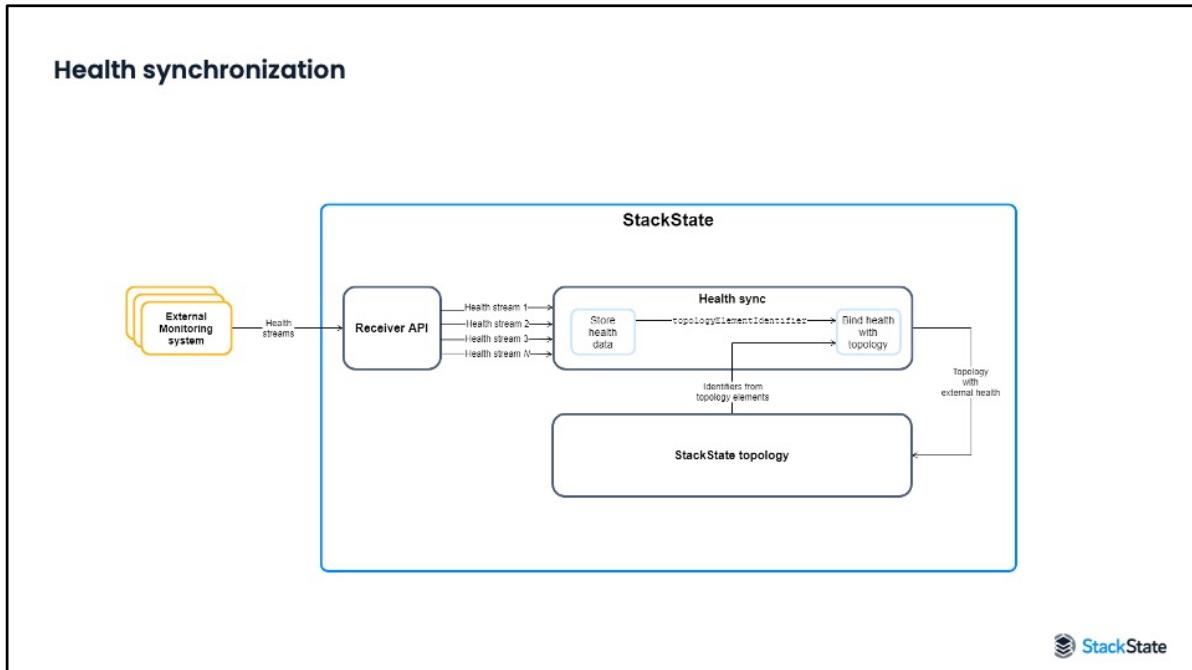
The health synchronization works with health streams. Health streams uniquely identifies the health synchronization and defines the boundaries within which the health states should be processed together. In other words, a single data source may report health states on a health stream to StackState. StackState processes the health stream such that the topology is updated accordingly and continues to do so after receiving updates from the data source without interfering other health streams.

Synchronized health states are bound to components or relations using

identifiers. These identifiers can be found on the detail panes, as shown here on the right. We'll have a look at how this works in a second.

The difference between health synchronization and health checks is where the health state is calculated. In health synchronization, the external data source does the health state calculation. Health synchronization has no calculation of health states in StackState, the processing is already done by the external system. The health state is merely synchronized. Compared to StackState health checks, health checks perform state calculation in StackState. A health check requires one or more telemetry streams to be defined. These telemetry streams receive data periodically from an external data source. Once telemetry data is received, the health check calculates the health state based on a set of rules defined in the check function

Certain StackState Agent integrations use health synchronization. You could also send health states directly via the StackState CLI or by sending JSON payloads to the StackState receiver. The CLI can also be used to manage health streams and get statistics on each health stream.



Here is an overview of how the health synchronization process works.

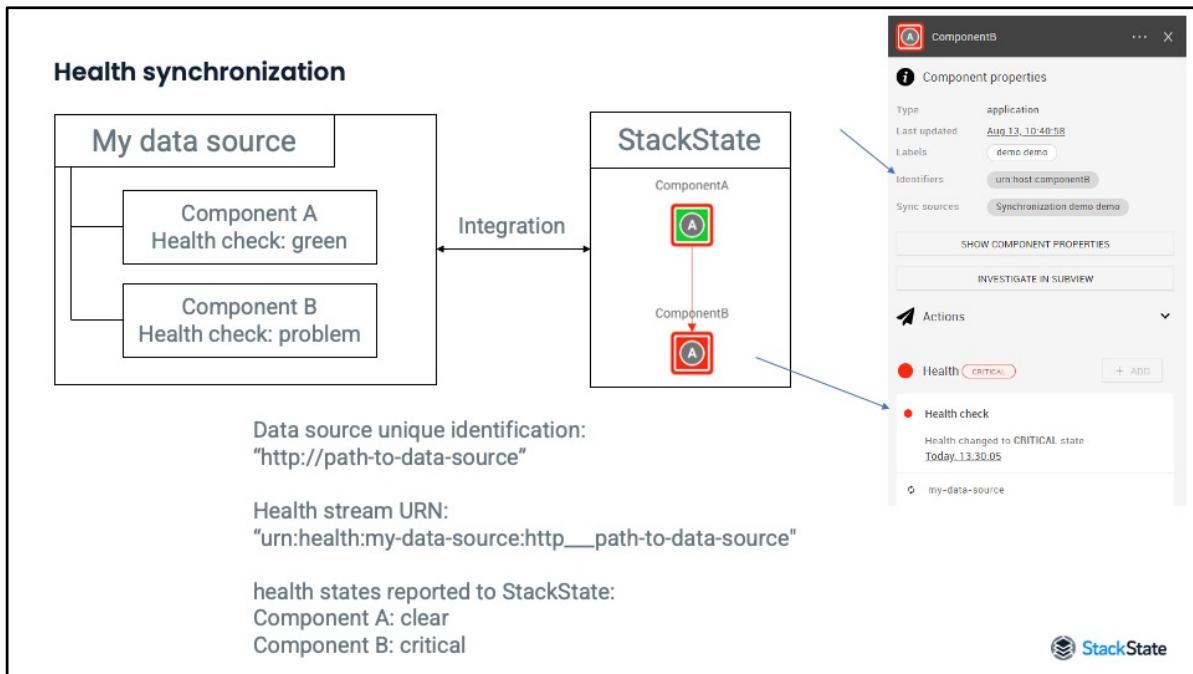
On the left we have external monitoring systems. These systems calculate the health states based on the data available to them. The health state information is shared with the StackState receiver in so called health streams.

The StackState receiver processes the received health states such that each stream of health states is processed separately by health synchronization. The health stream is defined in the received payload such that a distinction is made between health states and their meaning. You can see it as a logical grouping, it is a namespace.

Each health state in a health stream has a topology element identifier set. This identifier is matched against the identifiers available on components and relations provided by topology synchronization. You can find these identifiers on the details pane if you click on a component or relation in a view. Once a topology element is matched against the received identifier, the health state is bound to the topology element such that the health

state becomes visible on the matched component or relation. The synchronized health states will be used in the overall health state calculation of the topology element automatically.

Let's go over some examples to illustrate how the process works.



Consider the following example to illustrate how health streams work.

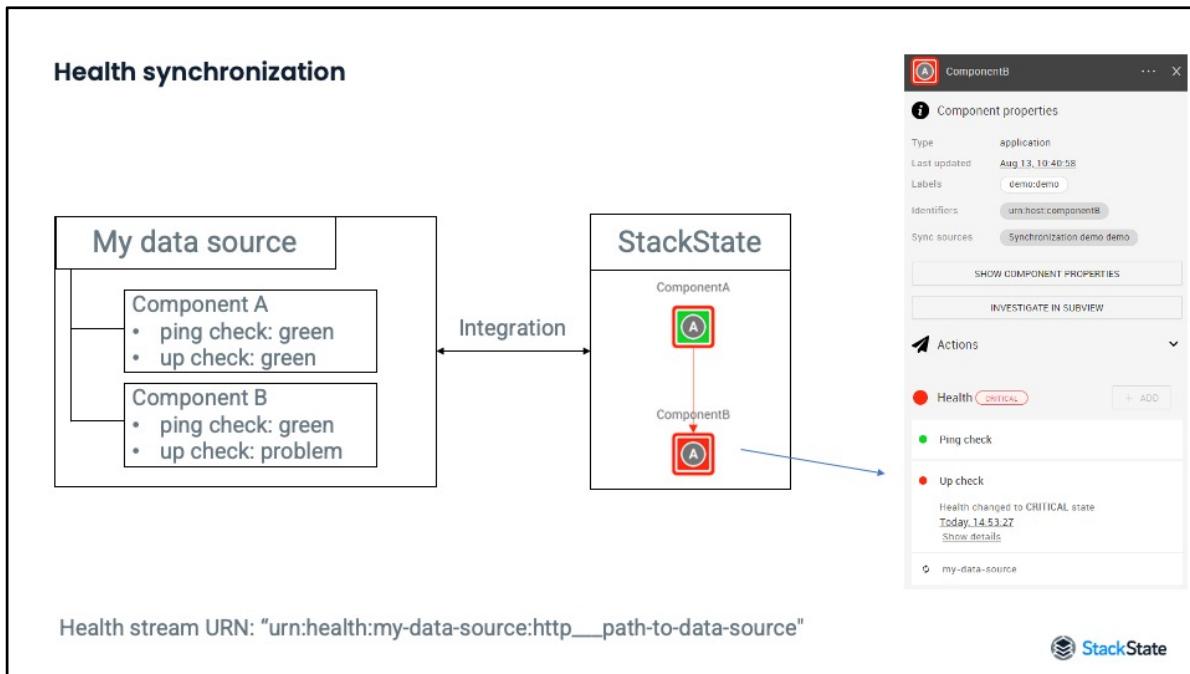
We have a data source one the left. This data source could be any monitoring system that calculates and stores health states. In this example, the data source is called 'My data source'. My data source contains health state information about two components, component A and component B. This may have been components such as applications, hosts, or business processes. The gist is that both components have a health check defined where the health check for component A reports green and the health check for component B reports a problem.

The 'My data source' monitoring solution may be installed multiple times in your organization depending on use case. We connect to this data source via connection string `http://path-to-data-source`. This uniquely identifies this monitoring system from another instance in your organization. We define a health stream URN using this information. The namespace we use is "`urn:health:my-data-source:http__path-to-data-source`". The `urn:health` prefix in the URN is mandatory. After that we define a source identifier, in this case the name of the data source,

hence my-data-source. The last part is the stream id, we set that to the URL we used to connect to, for unique identification between multiple instances of the same system. We altered the URL a bit for readability and because a colon has special meaning in a URN. It is okay to alter the URN as long as the chosen format is used consistently.

In the middle we have StackState. The two components are shown here. The topology synchronization synchronized these two components and a relation from another data source or from the monitoring data source itself. It does not matter where the topology elements were synchronized from. The health states will be bound via topology element identifiers. These topology element identifiers are shown on the details pane, as shown here on the right. In this example, an integration reads the health checks from the data source periodically and sends it to StackState. The health states are transformed to clear and critical for components A and B, respectively, to comply with the health states StackState supports. StackState receives the payload and binds the received health states to both components. The health states of the components change accordingly. In the component details pane you can see the 'Health check' present.

```
--- reference: used CLI commands for this example ---  
sts health send start "urn:health:my-data-source:http__path-to-data-  
source"  
sts health send check-state "urn:health:my-data-source:http__path-to-  
data-source" "componentA health check" "Health check"  
"urn:host:componentA" clear  
sts health send check-state "urn:health:my-data-source:http__path-to-  
data-source" "componentB health check" "Health check"  
"urn:host:componentB" critical  
sts health send stop "urn:health:my-data-source:http__path-to-data-  
source"
```



Consider the same example, however, this time we have more states available per component in our data source. In this case, we have a ping and up check health states available in the data source. Let's say the ping check indicates whether the component is reachable and the up check indicates whether a service is running. The integration will report both components periodically and the health states will be updated accordingly in StackState. The ping and up checks are now visible in StackState, on both components, to reflect the health states received.

It is possible that an optional message is included with the health state. You can see on the right that the 'up check' has a message set by clicking on 'Show details'.

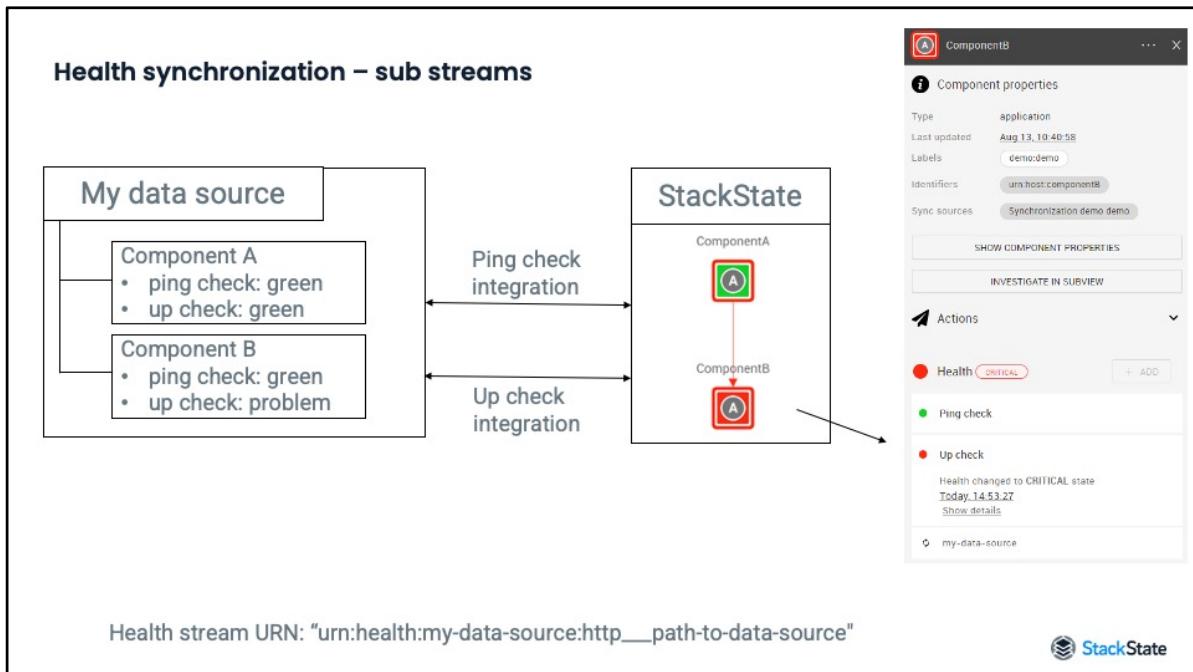
It is okay if not all components have the same health checks defined in the data source. A dynamic set of health checks in the source system will be reflected in StackState. Like topology synchronization, health synchronization is processing the health states as snapshots. This means that the latest processed health states are reflected in StackState. If one component has more checks defined compared to other components

provided on the same health stream, that's totally fine. If at any point in time a health state is removed in the data source, the health check would also be automatically removed from StackState.

--- reference: used CLI commands for this example ---

```
sts health send start -s "ping check" -e 12000 "urn:health:my-data-source:http__path-to-data-source"
sts health send check-state "urn:health:my-data-source:http__path-to-data-source" "componentA ping check" "Ping check"
"urn:host:componentA" clear -s "ping check"
sts health send check-state "urn:health:my-data-source:http__path-to-data-source" "componentB ping check" "Ping check"
"urn:host:componentB" clear -s "ping check"
sts health send stop -s "ping check" "urn:health:my-data-source:http__path-to-data-source"
```

```
sts health send start -s "up check" -e 12000 "urn:health:my-data-source:http__path-to-data-source"
sts health send check-state "urn:health:my-data-source:http__path-to-data-source" "componentA up check" "Up check" "urn:host:componentA"
clear -s "up check"
sts health send check-state "urn:health:my-data-source:http__path-to-data-source" "componentB up check" "Up check" "urn:host:componentB"
critical -s "up check" -m "Ping check failed, please have a look."
sts health send stop -s "up check" "urn:health:my-data-source:http__path-to-data-source"
```



One last variation on the example. Consider a scenario where not all health checks can be processed by only one integration. It might be that the data source is distributed in a certain way or that the integration code is able to collect a subset of the complete health stream's data. In the example, we need one integration for the ping check and one integration for the up check. The health stream URN stays the same since we are talking about a single data source.

We have now two integrations reporting on the same health stream. How does StackState handle that? Just sending two separate payloads using the same health stream URN will cause a race condition. The last received result will overwrite the previous processed result because health state when snapshots are used. If the 'ping check' health states are processed before the 'up check' health states are processed, the 'ping check' health states will be removed once the 'up check' health states have been processed. We can solve that using sub streams.

Each health stream has at least one sub stream in the same health stream URN. If you don't specify a sub stream explicitly, a default sub

stream is used. Each sub stream processes health states. A sub stream will not interfere with another sub streams, it's semi-independent. The sub stream does contribute to the health check states of the complete health stream.

For the example, that means that the 'ping check' integration reports on the sub stream called "ping" and the 'up check' integration reports on the sub stream called "up". If the ping sub stream health states are processed, you get to see the ping check in StackState. After processing the up sub stream health states, you get the up check in StackState as well. After processing both sub streams, the result is similar to previous example, as shown on the right.

It is possible to explicitly delete a health stream from StackState. Deleting a health stream will remove all health checks shown in StackState related to the provided health stream URN. In this example, where you have multiple sub streams, deleting a health stream will cause all sub streams to be deleted as well.

```
--- reference: used CLI commands for this example ---
sts health send start -e 12000 "urn:health:my-data-source:http__path-
to-data-source" -s "ping"
sts health send check-state "urn:health:my-data-source:http__path-to-
data-source" "componentA ping check" "Ping check"
"urn:host:componentA" clear -s "ping"
sts health send check-state "urn:health:my-data-source:http__path-to-
data-source" "componentB ping check" "Ping check"
"urn:host:componentB" clear -s "ping"
sts health send stop "urn:health:my-data-source:http__path-to-data-
source" -s "ping"

sts health send start -e 12000 "urn:health:my-data-source:http__path-
to-data-source" -s "up"
sts health send check-state "urn:health:my-data-source:http__path-to-
data-source" "componentA ping check" "Up check"
"urn:host:componentA" clear -s "up"
sts health send check-state "urn:health:my-data-source:http__path-to-
data-source" "componentB ping check" "Up check"
"urn:host:componentB" critical -m "Ping check failed, please have a look."
-s "up"
sts health send stop "urn:health:my-data-source:http__path-to-data-
```

source" -s "up"

```
[user@ip-172-30-0-151 ~]$ sts health list-streams
stream urn                                              sub stream count
-----
urn:health:my-data-source:http__path-to-data-source      2

[user@ip-172-30-0-151 ~]$ sts health list-sub-streams "urn:health:my-data-source:http__path-to-data-source"
sub stream id    check state count    repeat interval (Seconds)    expiry (Seconds)
-----
ping            2                  600                12000
up              2                  600                12000
```



Health streams and sub streams can be inspected. We can inspect the streams using the StackState CLI. For more information about the CLI health synchronization commands please refer to the CLI section.

The CLI command ‘health list-streams’ will list all health streams active in StackState. Using the previous example, we see one health stream. This health stream has two sub streams. The command ‘health list-sub-streams’ can be used to inspect the sub streams of a health stream by passing the health stream URN as its argument. You can see the ping and up sub streams shown at the bottom. You get some statistics on the count of check states and the configuration of each sub stream. More statistics can be obtained using the following command.

```
[user@ip-172-30-0-151 ~]$ sts health show "urn:health:my-data-source:http__path-to-data-source"
Aggregate metrics for the stream and all substreams:
metric          value between now and 300 seconds ago    value between 300 and 600 seconds ago    value between 600 and 900 seconds ago
latency (Seconds) 0.193                                -                                     -
messages processed (per second) -                      -                                     -
check states created (per second) -                      -                                     -
check states updated (per second) -                      -                                     -
check states deleted (per second) -                      -                                     -

Errors for non-existing sub streams:
error message   error occurrence count

[user@ip-172-30-0-151 ~]$ sts health show "urn:health:my-data-source:http__path-to-data-source" -s "ping"
Synchronized check state count: 2
repeat interval (Seconds): 600
Expiry (Seconds): 12000

Synchronization errors:
error message   error occurrence count

Synchronization metrics:
metric          value between now and 300 seconds ago    value between 300 and 600 seconds ago    value between 600 and 900 seconds ago
latency (Seconds) 0.26                                 0.193                                -
messages processed (per second) 0.006666666666666667 -                                     -
check states created (per second) 0.006666666666666667 -                                     -
check states updated (per second) -                      -                                     -
check states deleted (per second) -                      -                                     -
```

The 'health show' command offers more insights into a health stream's metrics by passing a health stream URN as its argument. You can also pass a sub stream to the same command to information about an individual sub stream. You get statistics on latency, messages processed, and the number of check states created, updated, and deleted per second.

Also, if any error related to a stream pops up, the details can be found here. The user interface will show you that an error has occurred and will point you to this command for further inspection.

Health synchronization

- Repeat interval
- Expiry interval



Some closing remarks about health stream snapshots.

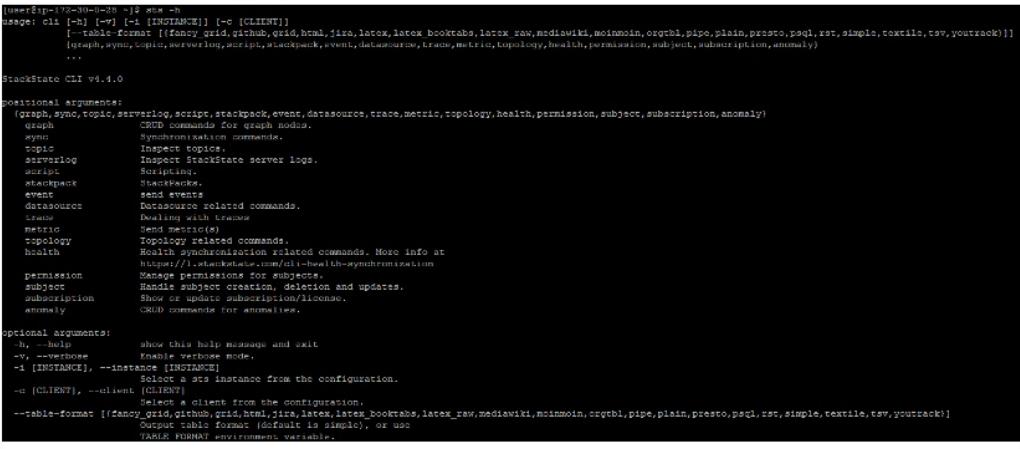
A health sub stream snapshot has a repeat interval in seconds defined. This repeat interval is a commitment from the external monitoring system to send complete snapshots over and over in order to keep the data up to date in StackState. StackState can then inform the user on how up to date the health synchronization is running.

A health stream can have an optional expiry interval defined. The expiry is set in seconds. An expiry is optional for streams using only the default sub stream. The expiry interval is mandatory when using more sub streams. The expiry interval is used to delete health state data after the defined number of seconds have elapsed. It is useful to delete data when the external system does not send data anymore. Data will not be deleted when data is received within the bounds of the expiry interval. Without an expire interval, the previously synchronized data would be left permanently hanging when StackState does not receive any data for a particular stream.

StackState CLI



In this section we will cover the StackState command line interface, or CLI for short. The CLI is a useful tool for the StackState admin to manage one or more StackState installations.



```

StackState CLI

usage: cli [-h] [-v] [-i [INSTANCE]] [-c [CLIENT]]
           [--table-format {[fancy_grid,github,grid,html,jira,lotus,latex,booktabs,latex_raw,mediawiki,mcimmo,orgtbl,pipe,plain,presto,pgsql,rst,simple,textile,tsv,yourtrack]}}
           ...
           ...

StackState CLI v4.1.0

positional arguments:
  (graph sync,topic,serverlog,script,stackpack,events,datasource,trace,metric,topology,health,permission,subject,subscription,anomaly)
  graph
    CMD commands for graph nodes.
  sync
    Syncronization commands.
  topic
    Inspect topics.
  serverlog
    Inspect StackState server logs.
  datasource
    Scripting.
  metricspec
    Metricspec.
  eventspec
    send events.
  datasource
    Datasource related commands.
  trace
    Dealing with traces.
  metric
    Send metric(s).
  topology
    Topology related commands.
  health
    Handle health-related commands. More info at
    https://stackstate.com/cli-health-commands
  permission
    Manage permissions for subjects.
  subject
    Handle subject creation, deletion and updates.
  subscription
    Show or update subscription/license.
  anomaly
    CMD commands for anomalies.

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Enable verbose mode.
  -i [INSTANCE], --instance [INSTANCE]
    Select an instance from the configuration.
  -c [CLIENT], --client [CLIENT]
    Select a client from the configuration.
  --table-format {[fancy_grid,github,grid,html,jira,lotus,latex,booktabs,latex_raw,mediawiki,mcimmo,orgtbl,pipe,plain,presto,pgsql,rst,simple,textile,tsv,yourtrack]}
    Output table format (default is simple), or use
    TABLE FORMAT environment variable.

```

StackState

The StackState CLI offers different commands. The commands cover different parts of StackState; from managing configuration, RBAC, sending data, to managing license keys.

The main help text is shown here. All available commands are listed here. Each command has its own help text, just append the `-h` argument to each command to get help.

We'll cover each command in this section.



Before we dive into each CLI command, let's install and configure the StackState CLI for use.

StackState CLI – installation

- Available for: Linux, Windows, and as Docker container
- Download from <https://download.stackstate.com/>

The screenshot shows the StackState CLI download page. At the top, there's a green header bar with the StackState logo and the text "StackState File Access". Below it, a message says "Please enter the license code below to get a list of downloads". There's a "License key:" input field and a "SUBMIT" button. Below this, there's a table of download links:

File	Action
stackstate-4.4.0.x86_64.rpm	DOWNLOAD
stackstate-4.4.0.zip	DOWNLOAD
stackstate_4.4.0_amd64.deb	DOWNLOAD
sts-cli-4.4.0.zip	DOWNLOAD
sts-cli-4.4.0-windows.exe	DOWNLOAD
sts-cli-4.4.0-linux64	DOWNLOAD

On the right side of the table, three download links are highlighted with arrows pointing to them:

- Docker (points to the zip file)
- Windows (points to the exe file)
- Linux (points to the deb file)

In the bottom right corner of the screenshot, there's a small StackState logo.

The StackState CLI can be downloaded from download.stackstate.com. A standalone executable is available for Linux and Windows. Look for the filenames prefixed `sts-cli`. The version is also shown in the filename. There is a zip archive available that contains a bash/PowerShell script to run the CLI as a Docker container. We'll walk you through the CLI installation process for Linux using the standalone executable. It is quite similar for the others.

You can download the StackState CLI executable using the download button. The downloaded file can be placed anywhere as long as there is a network connection to StackState. Hint: you can copy the link from the download button to use with `wget` or `CURL`.

After download you might want to rename the executable to `sts`, and/or create a symlink, give it executable permissions, and/or place it in your `$PATH` environment variable for easy access.

```
[user@ip-172-30-0-28 ~]$ ./sts -h
usage: sts [-h] [-v] [-i [INSTANCE]] [-c [CLIENT]]
           [-t [table-format]] [-o [FORMAT]]
           [-g [graph], -s [sync], -t [topic], -e [event], -d [datasource], -m [metric], -p [permission], -u [subject], -r [subscription], -a [anomaly]]
           ...
StackState CLI v9.4.0

positional arguments:
  graph, sync,topic,serverlog,script,stackpack,event,datasource,trace,metric,topology,health,permission,subject,subscription,anomaly
  graph          CRUD command for graph nodes.
  sync          Sync related commands.
  topic         Inspect topics.
  serverlog    Inspect StackState server logs.
  script        Scripting.
  stackpack    Stackpacks.
  event        Define events.
  datasource   Define source related commands.
  trace         Dealing with traces.
  metric        Send metric(s).
  topology     Topology related commands.
  health        Health synchronization related commands. More info at
                https://github.com/stackstate/stackstate/tree/main/health-synchronization
  permission   Namespace related commands for subjects.
  subject      Handle subject creation, deletion and updates.
  subscription Show or update subscription/licensing.
  anomaly     CRUD commands for anomalies.

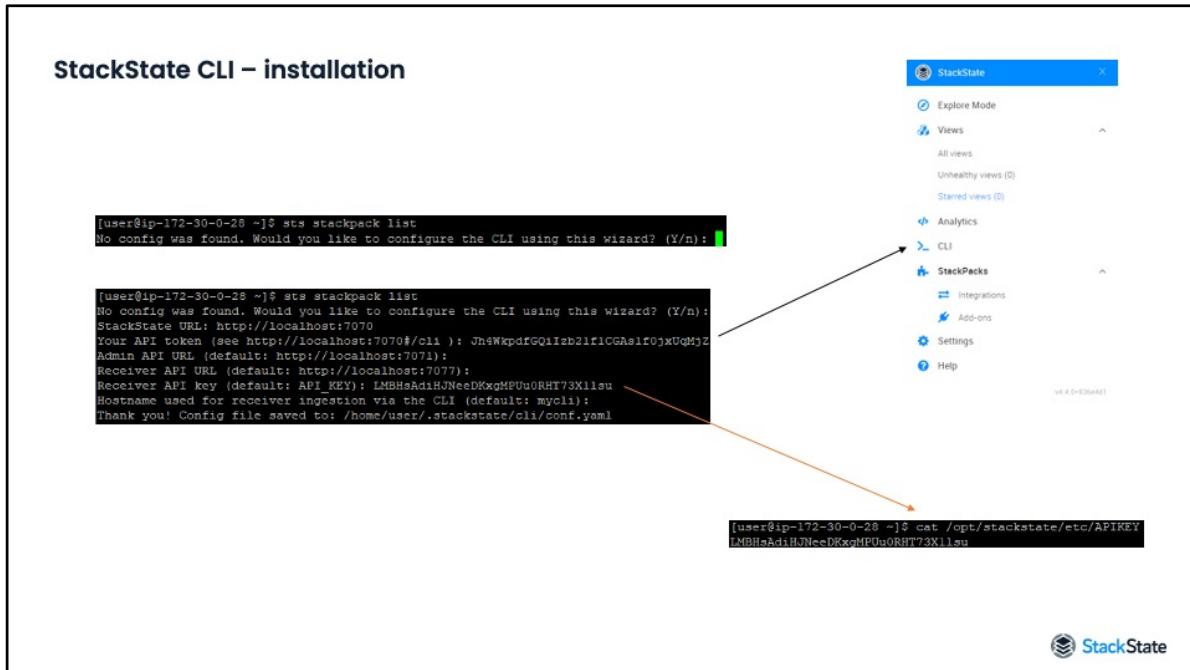
optional arguments:
  -h, --help       show this help message and exit
  -v, --verbose    Enable verbose mode.
  -i [INSTANCE], --instance [INSTANCE]
                  Select a std instance from the configuration.
  -c [CLIENT], --client [CLIENT]
                  Select a client from the configuration.
  --table-format [[fancy_grid,github,grid,html,jira,latex,latex_booktabs,latex_raw,mediawiki,mediawiki,orgtbl,pipe,plain,presto,pgsql,xst,simple,textile,tsv,yourtrack]]
  --format [FORMAT]
                  Output table format (default is simple), or use
                  YAML FORMAT environment variable.

[User@ip-172-30-0-28 ~]$ sts stackpack list
No config was found. Would you like to configure the CLI using this wizard? (Y/n): 
```



After download of the CLI you are ready to configure the CLI. We need to tell the CLI which StackState instance to use. You can execute command `./sts -h` to get help and see if the CLI works as expected.

The CLI configuration is read from a YAML configuration file in the home directory of the current user. The configuration file is located at `'/<home-directory>/stackstate/cli/conf.yaml'`. You can either place a YAML file directly, an example is available on the StackState documentation website, or have the CLI generate one. We'll go for the latter option.



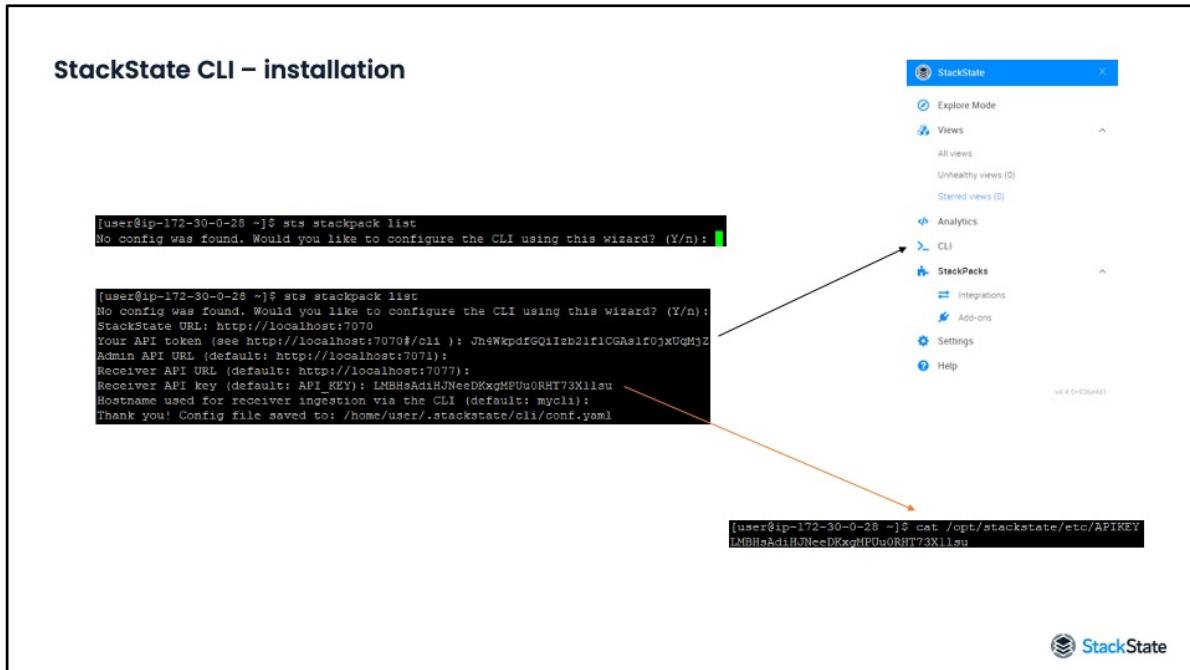
To generate the CLI's configuration YAML file we have to execute any command that will connect to StackState. For example, let's use the StackPack CLI command to list all StackPacks available. The command to get all StackPacks is `sts stackpack list`. The first time this command will ask you if you want to create a CLI configuration file. Press enter to generate a configuration file. The CLI will ask the following questions to configure the CLI correctly.

The first question is about the StackState URL. The StackState URL is the URL which you use to open the StackState user interface in the browser. The StackState URL is also called StackState base URL during installation. In this example, we're using localhost since the CLI is installed next to StackState, for demonstration purposes.

The second question is about your personal API token. Each user in StackState has a personal API token. This API token is specifically for the CLI to authenticate with the configured StackState instance. Your personal API token can be found in the user interface, by opening the CLI page from the main menu.

The screenshot shows the StackState user interface with a sidebar on the left containing navigation links: Explore Mode, Views, Analytics, CLI (which is expanded), StackPacks, Integrations, Add-ons, Settings, and Help. The main content area is titled "StackState CLI – installation". It features a large icon of a terminal window with the text "CLI" and the sub-instruction "Manage StackState using the CLI". Below this is a text input field labeled "Your API token" containing a long string of characters: "3h4ukpdHqQ111921f1C64a3fBjwQWjZ". A tooltip next to the input field says "Use this token to access the StackState API via the CLI.". Underneath the input field is a "Get started" section with two columns: "Installing and configuring the CLI" (describing the CLI executable) and "Using the CLI" (listing options like Import and export StackState configuration, Inspect data, Send data, Manage StackPacks, Scripting, and License). A "LEARN MORE" button is located at the bottom of the "Get started" section. At the very bottom of the page, there is a footer note: "If you are having trouble installing, configuring or using the StackState CLI, our SUPPORT TEAM will be glad to provide further assistance." The StackState logo is in the bottom right corner.

The API token can be found in the StackState user interface on the CLI page. The API token is specific to the user you log in with. You can copy the token from this page.



The third question is about the admin API URL. Some CLI commands use an admin API endpoint for StackState admin tasks. An example of an admin command that uses the admin API is `sts graph delete --all` to purge the database. Another example is the graph 'sts retention' command. The TCP port for the admin API is 7071 for Linux packages. For Kubernetes this is '/admin' on the StackState-router service.

The forth question is about the receiver API URL. The receiver API URL is where the StackState API can be accessed. This is a different endpoint than the StackState user interface. For the Linux packages, this is TCP port 7077. For Kubernetes, this usually is '/receiver' appended to the StackState URL. This URL is used when CLI commands generate and send topology, telemetry, trace, or anomaly data to StackState.

The fifth question is about the Receiver API key. The CLI is able to send data to the receiver API. The CLI needs to authenticate against the receiver API to do so. The receiver API key can be found in file `/opt/stackstate/etc/APIKEY` for Linux packages and in `values.yaml` for

Kubernetes.

The sixth and last question is about the hostname for receiver ingestion. The hostname is used in CLI commands that generate and send topology, telemetry, trace, or anomaly data to tell StackState from which host the data originated from. For example, the StackState Agent uses the hostname of the machine where it is installed on.

After providing the information, the configuration is written to /<home-directory>/.stackstate/cli/conf.yaml. You can change CLI configuration afterwards if need to.

StackState CLI – installation

```
[user@ip-172-30-0-28 cli]$ cat ~/.stackstate/cli/conf.yaml
instances:
  default:
    admin_api:
      apitoken_auth:
        token: Jn4WkpdfGQilzb2lfICGAslf0jxUqbijZ
        url: http://localhost:7071
    base_api:
      apitoken_auth:
        token: Jn4WkpdfGQilzb2lfICGAslf0jxUqbijZ
        url: http://localhost:7070
    clients:
      default:
        api_key: LMBHsAdiHJNeeDKxgMPUu0RHT73Xllsu
        hostname: mycli
        internal_hostname: mycli
    receiver_api:
      url: http://localhost:7077
```

name	display name	categories	installed version	next version	latest version	instance count
aad	Autonomous Anomaly Detector	Anomaly Detection, AI, Add-on	-	-	0.5.2	0
autosync	Custom Synchronization	-	-	-	3.1.1	0
aws	AWS (Legacy)	Cloud, Topology, Metrics	-	-	5.3.1	0
aws-v2	Amazon Web Services	Cloud, Topology, Metrics, Events	-	-	1.0.1	0



After using the wizard, the resulting configuration file looks like here, on the top. You can change this file if you need to.

Notice that under instances there is one key, called ‘default’. This is the instance the CLI is going to use per default. You can define more instances if needed. This is especially useful if you want to use the CLI for multiple StackState instances. For example, use the CLI on your local machine to connect to a development or production StackState environment. You can use the `sts -i` argument to use a non-default CLI instance. Within an instance, you can also specify more ‘clients’ and select the one you need using the sts –c argument. Specifying another client will cause commands that send data to StackState to use another API key and hostname in the payload.



Let's have a look at the StackState CLI's graph command.

StackState CLI – graph command

```
[user@ip-172-30-0-28 cli]$ sts graph -h
usage: cli graph [-h]
                  (list-types,list,show-node,update-node,export,import,delete,retention,unlock)
                  ...
positional arguments:
  command_class
    (list-types,list,show-node,update-node,export,import,delete,retention,unlock)
      list-types          List all readable node types.
      list               List graph nodes by type.
      show-node          Show node as json.
      update-node        Updates a node from std input on json format.
      export             Export graph nodes.
      import             Import graph nodes via stdin. Usage: `sts graph import < myfile.stj`'
      delete             Delete graph nodes.
      retention          Configure how long the StackState data graph retains data. More info can be found at http://docs.stackstate.com/setup/retention/
      unlock             Unlock graph nodes which are locked because of a dependency within StackState. Unlocking nodes might block future upgrades of StackPacks

optional arguments:
  -h, --help            show this help message and exit
```



The graph command can be used to perform create, read, update, or delete actions on graph nodes. Graph nodes here can be StackState settings, topology information, etc. Essentially accessing what is stored in StackGraph.

The list-types command returns all node types that are accessible. You can list all Graph nodes of a certain type using the list command. The list command returns basic information about the graph node. Each graph node has an id. This id you can use to get the full definition of the setting using the show-node command. You can import a node's definition again using the update node command. Updating a locked node will yield an error.

The export and import commands can be used to export or import a set of nodes. This is similar to the export/import functionality provided on the StackState settings page. For the export and import commands you can also specify a namespace. This allows you to export or import settings that belong together. A StackPack does exactly this, during installation it imports graph nodes in the StackPack specific's URN namespace. The

import command also has unlocked-strategies to either fail on, skip, or overwrite unlocked graph nodes. The unlocked-strategy is used when you upgrade a StackPack in the user interface and you have unlocked settings; StackState asks you how you want to handle this. The default unlock strategy is set to fail.

Next to importing and exporting graph nodes you can also delete them. The delete command accepts one or more ids of the nodes you want to delete. An error is given when you try to delete locked nodes, you have to unlock them first. The delete command also accepts a --all argument which purges the whole database. To unlock nodes, you can use the unlock command.

The retention command gives you access to StackGraph data retention. You can get and set how long StackGraph retains topology data. By default the retention is set to 8 days. The retention command needs an admin API token that has the `access-admin-api` permission.

StackState CLI – graph command

id	type	name	description	owned by	manual	last updated
135543269041042	Layer	Users	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
2497753651580877	Layer	Applications	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
280269384301050	Layer	Containers	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
241707484705467	Layer	Hardware	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
174728320219847	Layer	Storage	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
80504516544897	Layer	Processes	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
220882480884488	Layer	Machines	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
55332376645965	Layer	Blueprints	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
551225247133977	Layer	Serverless	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
116619679315050	Layer	Services	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
256578365885852	Layer	Locations	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
226755106560400	Layer	Uncategorized	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
94674272646728	Layer	Application Load Balancers	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
76251806274885	Layer	Business Processes	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
117736911243256	Layer	Messaging	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
160722674300226	Layer	Databases	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021
59289624466582	Layer	Networking	-	urn:stackpack:common	False	Tue Aug 10 13:35:51 2021



Here's an example where all the layers present in StackState are returned. Each layer has an id and name. The owned by column specifies in which URN namespace the nodes belong. In this case all the layers present in this specific StackState instance come from the Common StackPack. The manual column refers to the fact if the node is unlocked or not.

StackState CLI – graph command

```
[user@ip-172-30-0-28 cli]$ sts graph list --namespace urn:stackpack:common --ids Layer | xargs sts graph export --ids
{
  "_version": "1.0.33",
  "nodes": [
    {
      "_type": "Layer",
      "id": -13,
      "identifier": "urn:stackpack:common:layer:application-load-balancers",
      "name": "Application Load Balancers",
      "order": 5000.0
    },
    {
      "_type": "Layer",
      "id": -2,
      "identifier": "urn:stackpack:common:layer:applications",
      "name": "Applications",
      "order": 4000.0
    },
    {
      "_type": "Layer",
      "id": -8,
      "identifier": "urn:stackpack:common:layer:blueprints",
      "name": "Blueprints",
      "order": 1000.0
    }
  ]
}
```



Here's another example. This time we list all Layer nodes that are from the Common StackPack and print their ids. We pipe these ids as arguments to the export command. The command exports the requested layers. You can redirect the command's output to file as a next step. The output is in STJ format, similar to the export functionality offered in the StackState user interface. This format can also be imported again.



Let's have a look at the StackState CLI's sync command. This command can be used to list StackState synchronizations.

StackState CLI – sync command

```
[user@ip-172-30-0-28 cli]$ sts sync -h
usage: cli sync [-h] {list,list-topics} ...
positional arguments:
  command class
    {list,list-topics}
      list           List all available syncs.
      list-topics   List all available topics.
```



There are two arguments to the sync command. You can list all synchronizations and you can list all topology specific Kafka topics that are available in StackState. Let's have a look at an example.

StackState CLI – sync command

```
[user@ip-172-30-0-28 cli]$ sts sync list
  id  type   name           description  owned by
-----  -----
155310402965470 Sync  Synchronization demo demo - urn:stackpack:autosync:instance:81b2a443-c704-40ef-a474-a7eacec36875 False  Tue Aug 10 14:39:21 2021

-----
```



```
[user@ip-172-30-0-28 cli]$ sts sync list-topics
type      topic
-----  -----
demo     sts_topo_demo_demo
process  sts_topo_process_agents
trace    sts_topo_trace_agents
```



The first command, shows all Synchronizations in StackState. In this case it is only one synchronization. This synchronization comes from the Custom Synchronization StackPack as the namespace urn:stackpack:autosync belongs to that StackPack.

The second command shows all Kafka topics that are present for Synchronizations to use. The Sts data sources are essentially listeners to Kafka topics. Since there is a synchronization, there is also a Sts data source backing that synchronization. In this case, the topic sts_topo_demo_demo is used by the Sts data source backing the Synchronization. You can verify which topic is used by opening up the Sts data source settings page. A topic name is constructed to uniquely identify a data source. There is a prefix sts_topo to StackState that the topic contains topology information. Separated by underscores, you have this prefix, a type, and lastly, a name. In the case of the Custom Synchronization StackPack, the type is the instance type and the name is the instance URL provided during installation of the StackPack. This may differ between StackPacks, or integrations.



Let's have a look at the StackState CLI's topic command. This command can be used to look at Kafka topics.

StackState CLI – topic command

```
[user@ip-172-30-0-28 cli]$ sts topic -h
usage: cli topic [-h] {list,show,export} ...
positional arguments:
  command_class
    {list,show,export}
      list            List all topics.
      show           Show messages on a topic.
      export         Export topic messages in json format.
optional arguments:
  -h, --help        show this help message and exit
```



The topic command 'list' shows all the StackState Kafka topics available. Where CLI's sync list-topics command only shows Kafka topics related to topology synchronization, this command shows all StackState related topics. The show argument lets you inspect the contents of a Kafka topic. The export command is similar to the show command, the export command saves messages to a local file. Let's have a look at some examples.

StackState CLI – topic command

The screenshot shows two terminal windows side-by-side. The left window displays a list of Kafka topics:

```
[user@ip-172-30-0-28 cli]$ sts topic list
name
-----
sts_topo_trace_agents
sts_intake_health
sts_trace_events
sts_multi_metrics
sts_generic_events
sts_internal_events
sts_health_sync
sts_state_events
sts_correlate_endpoints
sts_topo_demo_demo
sts_topo_process_agents
sts_connection_beat_events
sts_topology_events
sts_internal_topology
```

An orange arrow points from the 'sts_topo_demo_demo' topic in the list to the right window. The right window shows the output of the 'topic show' command for that specific topic:

```
[user@ip-172-30-0-28 cli]$ sts topic show sts_topo_demo_demo
{
  "messages": [
    {
      "message": {
        "TopologyElement": {
          "collectionTimestamp": 1628608797264000,
          "ingestionTimestamp": 1628608796827,
          "payload": {
            "TopologyStopSnapshot": {}
          }
        }
      },
      "offset": 5,
      "partition": 0
    },
    {
      "message": {
        "TopologyElement": {
          "collectionTimestamp": 1628608797264000,
          "ingestionTimestamp": 1628608796827,
          "payload": {
            "TopologyComponent": {
              "date": "{\"labels\":(\"demo:demo\"),\"name\":\"ComponentA\"}",
              "externalId": "ComponentA",
              "typeName": "application"
            }
          }
        }
      },
      "offset": 4,
      "partition": 0
    }
  ]
}
```



Consider the following example. A topic list is returned by the command on the left. We will use the 'topic show' command to inspect the topology messages on the provided Kafka topic. Several messages are shown, including an Kafka's offset and partition. In this example, you see two messages. The bottom message contains a TopologyComponent payload, this payload will be used by StackState synchronization to create a component. You also see a TopologyStartSnapshot payload. The TopologyStartSnapshot and TopologyStopSnapshot payloads are to signal synchronization that all messages between the start and stop messages are to be considered a topology snapshot. The topic show command offers control over the number of messages to return, it is possible to select a starting offset, partition, and a limit.

StackState CLI – topic command

```
[user@ip-172-30-0-28 cli]$ sts topic export sts_topo_demo_demo -o 4 -b -m 1
Total downloaded count: 1 messages.
[user@ip-172-30-0-28 cli]$ cat sts_topo_demo_demo.json
[{
    "partition": 0,
    "offset": 4,
    "message": {
        "TopologyElement": {
            "collectionTimestamp": 1628608797264000,
            "payload": {
                "TopologyComponent": {
                    "externalId": "componentA",
                    "typeName": "application",
                    "data": "{\"labels\": [\"demo:demo\"], \"name\": \"ComponentA\"}"
                }
            },
            "ingestionTimestamp": 1628608796827
        }
    }
}] [user@ip-172-30-0-28 cli]$
```



Similar to the topic show command, we can export the messages to file using the 'topic export' command. From previous example, we want to extract the TopologyComponent payload and save that to file. The **-o** argument defines which Kafka offset to start, the **-m** argument specifies the number of messages to return, and the **-b** argument ensures that the output is be human readable.



Let's have a look at the StackState CLI's serverlog command.

StackState CLI – serverlog command

```
[user@ip-172-30-0-122 ~]$ sts serverlog setlevel -h
usage: cli serverlog setlevel [-h] id {OFF,ERROR,WARNING,INFO,DEBUG,DISABLE}
positional arguments:
  id                  The function's instance id to set the log level for.
  {OFF,ERROR,WARNING,INFO,DEBUG,DISABLE}
                        The log level to set.

optional arguments:
  -h, --help            show this help message and exit
```

```
[user@ip-172-30-0-122 ~]$ sts serverlog setlevel 183296670111843 INFO
OK
```



To enable, or disable, logging on a user-defined function you can use the ‘serverlog setlevel’ command. You need to obtain the database id of the function instantiation. You cannot enable logging on the function id itself because you may end up with a lot of log messages from all kinds of objects using the same function. In most cases you want to look at specific instantiations. The command expects the id and the log level you want to have as arguments. After enabling the logging, the log messages will appear in stackstate.log. The logger logs the id in each log message generated by the function. This means you can filter the log file to easily find the function’s log messages.

Don’t forget to disable the function logging after you are done. You can disable the function logging for the specific instance by setting the log level to OFF.

Do note that log messages will only appear when the check is actually run.

More information on function logging can be found in the troubleshooting

section.



Let's have a look at the StackState CLI's script command. This command can be used to execute scripts against StackState data model.

StackState CLI – script command

```
[user@ip-172-30-0-122 ~]$ sts script -h
usage: cli script [-h] {execute} ...

positional arguments:
  command_class
    {execute}
      execute      Execute a script string via stdin.

optional arguments:
  -h, --help       show this help message and exit
```

```
[user@ip-172-30-0-122 ~]$ sts script execute -h
usage: cli script execute [-h] [-a [ARGUMENTS]] [-s [{groovy,handlebars}]] [-t [TIMEOUT]]

optional arguments:
  -h, --help           show this help message and exit
  -a [ARGUMENTS], --arguments [ARGUMENTS]
                        Arguments script string.
  -s [{groovy,handlebars}], --script_type [{groovy,handlebars}]
                        Script type. Defaults to "groovy"
  -t [TIMEOUT], --timeout [TIMEOUT]
                        Timeout in seconds. Defaults to 15
```



The only command currently available is the ‘script execute’ command. This command accepts either groovy or handlebars script from the standard input. StackState will execute this command and returns the result to standard out. You can also pass arguments to your script if you need something in the script to be dynamic.

The screenshot shows the StackState CLI interface. On the left, there is a 'Query' panel containing the command: `1: Topology.query('name = "Demo component"')`. Below this are 'CLEAR' and 'EXECUTE' buttons. On the right, under the 'Result' tab, the output is displayed in 'JSON' format. The JSON response includes a 'result' object with a 'queryResults' array containing one item. This item has a '_type' of 'TopologyScriptApiQueryResult', a 'query' object with a '_type' of 'TopologyScriptApiQuery' and a 'query' value of 'name = "Demo component"', and a 'result' object with a '_type' of 'ViewSnapshot'. The 'Components' field of the result contains a single component object with details such as '_type': 'ViewComponent', 'description': '', 'domain': 149433314143860, 'environments': [247063716872328], and 'id': 190026691810914.

Consider the following example. In this example an analytical script is used to retrieve one component. The same script can be executed with the 'script execute' command. Since there are some quotes in the query, it needs proper escaping. Alternatively, you can put the script into a file and pass the contents of the file to the CLI.



Let's have a look at the StackState CLI's stackpack command. This command can be used to manage StackPacks.

StackState CLI – stackpack command

```
[user@ip-172-30-0-122 ~]$ sts stackpack -h
usage: cli stackpack [-h]
                      (list,upload,list-instances,list-parameters,install,confirm-manual-steps,uninstall,upgrade)
                      ...
positional arguments:
  command_class
    (list,upload,list-instances,list-parameters,install,confirm-manual-steps,uninstall,upgrade)
      list           List StackPacks available in StackState.
      upload         Add an '.sts' file to StackState.
      list-instances List instances of a particular StackPack.
      list-parameters List installation parameters of a particular
                        StackPack.
      install        Trigger installation of a new instance of a particular
                        StackPack.
      confirm-manual-steps
                        Trigger confirmation of manual steps for an instance
                        in WAITING_FOR_MANUAL_STEPS status of a particular
                        StackPack.
      uninstall     Trigger uninstallation of an instance of a particular
                        StackPack.
      upgrade       Trigger upgrade of a particular StackPack.

optional arguments:
  -h, --help        show this help message and exit
```



The StackPack command can be used to list available StackPacks, install, uninstall, or upgrade a StackPack. If you have developed your own StackPack, you can use the `stackpack upload` command to upload your StackPack to StackState. You can list available StackPacks or list installed StackPack instances. Upgrade of StackPacks can only be done for minor releases. Major releases of a StackPack may require a different set of parameters.

Installation of StackPacks may require user input, the parameters of a StackPack can be found using the '`stackpack list-parameters`' command. Some StackPacks have a state after installation where the user has to perform some action. The '`stackpack confirm-manual-steps`' command can be used to confirm StackPacks that have a manual step before it is marked active.

StackState CLI – stackpack command							
sts stackpack list							
name	display name	categories	installed version	next version	latest version	instance count	
aad	Autonomous Anomaly Detector	Anomaly Detection, AI, Add-on	-	-	0.9.2	0	
autosync	Custom Synchronization	-	-	3.1.1	0		
aws	AWS (Legacy)	Cloud, Topology, Metrics	-	-	5.3.1	0	
aws-v2	Amazon Web Services	Cloud, Topology, Metrics, Events	-	-	1.0.1	0	
azure	Azure	Cloud	-	-	4.1.1	0	
basic	Manual Topology	-	2.1.1	-	2.1.1	1	
cloudera	Cloudera	Big data, Topology, Metrics	-	-	1.3.1	0	
discovery	Discovery	CMDB	-	-	0.0.1	0	
dynatrace	Dynatrace	Topology, Monitoring	-	-	1.1.2	0	
gcp	Google Cloud Platform	Cloud	-	-	0.0.1	0	
graphite	Graphite	Monitoring, Metrics	-	-	0.0.2	0	
health-forecast	Health Forecast	AI, Add-on	-	-	1.0.3	0	
humio	Humio	Monitoring, Logs	-	-	1.0.0	0	
kubernetes	Kubernetes	Container, Topology, Metrics	-	-	3.9.5	0	
logzio	Logz.io	Monitoring, Logs	-	-	0.0.2	0	
nagios	Nagios	Monitoring, Events	-	-	2.6.3	0	
omnibus	Omnibus	Monitoring, Events	-	-	0.0.2	0	
openshift	OpenShift	Container, Topology, Metrics	-	-	3.7.6	0	
opsgenie	OpsGenie	ITSM, Events, Notification	-	-	0.0.2	0	
pager-duty	PagerDuty	ITSM, Events, Notification	-	-	0.0.2	0	
sap	SAP	Service, Monitoring	-	-	1.3.1	0	
scorm	Scorm	Monitoring, Topology, Events	-	-	2.1.1	0	
service-now	ServiceNow	ITSM, Topology	-	-	5.3.1	0	
slack	Slack	Chat, Notification	-	-	0.0.4	0	
solarwinds	SolarWinds	Topology	-	-	1.0.0	0	
splunk-topology	Splunk	Logs	-	-	1.2.0	0	
stackstate-agent-v2	StackState Agent V2	Topology, Events, Metrics, Agent	-	-	4.4.8	0	
stackstate-self-health	StackState Self Health	-	-	-	4.4.0-rc.11	0	
static-topology	Static Topology	Topology	-	-	2.3.2	0	
ucmdb	HP UCMDB	CMDB, Topology	-	-	0.0.2	0	
vsphere	VMWare vSphere	Virtualization, Topology, Metrics	-	-	2.3.1	0	
xl-deploy	XL Deploy	CI/CD, Topology	-	-	0.0.2	0	
zabbix	Zabbix	Monitoring, Metrics	-	-	0.0.1	0	

The ‘stackpack list’ command will show all the StackPacks that are available in StackState.

The name column shows the internal name of the StackPack. The internal name is used in other StackPack CLI commands. The display name column shows the name of the StackPack as shown to the user in the user interface. The categories column shows the categories in which the StackPack is available in the user interface. The latest version column shows the latest version of the StackPack currently available. The installed version column shows the version of the currently installed StackPack. These versions may differ. The installed version shown will differ from latest version when the installed version is not upgraded yet to the latest available version. The next version column will show the version you can upgrade the StackPack to. The instance count indicates the number of instances a StackPack is installed. Most StackPacks have multi-instance support.

The ‘stackpack list’ command also has an option to only show installed StackPacks.

StackState CLI – stackpack command

sts stackpack list-instances autosync

```
[user@ip-172-30-0-151 ~]$ sts stackpack list-instances -h
usage: cli stackpack list-instances [-h]
                                     [-s [{NOT_INSTALLED,PROVISIONING,WAITING_FOR_DATA,INSTALLED,DEPROVISIONING,UPGRADING,ERROR,WAITING_FOR_MANUAL_STEPS}])
                                     <stackpack name>

positional arguments:
  <stackpack name>      StackPack name.

optional arguments:
  -h, --help            show this help message and exit
  -s [{NOT_INSTALLED,PROVISIONING,WAITING_FOR_DATA,INSTALLED,DEPROVISIONING,UPGRADING,ERROR,WAITING_FOR_MANUAL_STEPS}], --status [{NOT_INSTALLED,PROVISIONING,WAITING FOR DATA,INSTALLED,DEPROVISIONING,UPGRADING,ERROR,WAITING FOR MANUAL STEPS}]
                        List instances with specified status only.
```

```
[root@ip-172-31-11-245 cli]# ./bin/sts stackpack list-instances autosync
id      name      status      version      last updated
-----
```



The ‘stackpack list-instances’ command lists all instances of a StackPack. It is possible to filter on status, which is useful if you have many StackPack instances installed.

The command requires the internal StackPack name as its argument. It returns a list of StackPack instances for the given StackPack. If nothing is returned, then no StackPack instance is installed. Otherwise, the id, name, status, version, and last updated date is returned for each instance. The status column refers to the status you see on the StackPack page in the user interface. The last updated date refers to the most recent update made to the StackPack instance.

StackState CLI – stackpack command

```
sts stackpack list-parameters autosync
```

[user@ip-172-30-0-122 ~]\$ sts stackpack list

name	display name	type
ad	Autonomous Anomaly Detector	
autosync	Custom Synchronization	
aws	AWS (Legacy)	
aws-v2	Amazon Web Services	

[user@ip-172-30-0-122 ~]\$ sts stackpack list-parameters autosync

name	display name	type
sts_instance_type	Instance type (source identifier)	Text
sts_instance_url	Instance URL	Text

Custom Synchronization
Version 3.1.1

Custom Synchronization
This StackPack contains everything you need to automatically synchronize topology information from an external topology source.
[Read more...](#)

Installed Instances

New Instance	Not installed yet
<input type="radio"/> Instance type (source identifier)	Instance type (source identifier)
<input type="radio"/> Instance URL	Instance URL

Installation
Click the install button to start the installation process of the StackPack. During this process, StackState provides instructions for steps required to finish the configuration of this StackPack.

INSTALL

StackState

Let's walk through the installation process of a StackPack. The process is similar for all StackPacks. In this scenario we're going to install the custom synchronization StackPack using only CLI commands. From the 'stackpack list' command we can see that the internal StackPack name of the Custom Synchronization command is 'autosync'.

The custom synchronization StackPack is a multi-instance StackPack and requires some user input. Before we can install the StackPack using the CLI, we need to know the names of the parameters such that we can provide values for these. The 'stackpack list-parameters' command is used to get the StackPack parameters. The name column will show the internal name of the parameter. The display name column shows the parameter's name as shown on the StackPack page. The type column defines what type of input the parameter expects. Usually the parameter is a text box, a password is another option.

StackState CLI – stackpack command

```
sts stackpack install autosync \
-p sts_instance_type demo \
-p sts_instance_url demo
```

The screenshot shows the terminal output of the command:

```
[User@ip-172-30-0-122 ~]$ sts stackpack install autosync -p sts_instance_type demo -p sts_instance_url demo
Successfully triggered installation of StackPack autosync. Check the status of the process using the command 'stackpack list-instances autosync' or debug any errors using 'serverlog show stackpack/autosync'
id name status version last updated
79233792710522 -- PROVISIONING 3.1.1 Wed Aug 11 15:49:15 2021
```

Below the terminal, there is a screenshot of the StackState web interface showing the 'Custom Synchronization' StackPack details. Arrows point from the 'Instance type (source identifier)' and 'Instance URL' fields in the interface to the corresponding parameters in the terminal command.

Now we have the name of the StackPack and its parameters, we are ready for the next step. The next step is to install the StackPack. The 'stackpack install' command is used to install a StackPack. The first argument is the StackPack's internal name. For a StackPack that does not have parameters, you are good to go. For a StackPack that does have parameters, we supply parameter values using the -p argument. You need to define one -p argument per StackPack parameter as key/value pair. The key is parameter's name followed by the value, you want to pass, separated by a space.

In the scenario, we are installing the Custom Synchronization StackPack and we define two parameters. The internal StackPack name of the Custom Synchronization StackPack's is autosync, which will be the first argument. We provide the value 'demo' to both parameters, for demonstrational purposes. After executing the command, StackState will provision the StackPack. The installation process is asynchronous. The output of the 'stackpack install' command states that the StackPack instance is in PROVISIONING state. Notice that the StackPack instance received a database id and it shows the StackPack version number. You

can monitor the installation status using the command ‘stackpack list-instances’.

StackState CLI – stackpack command

```
sts stackpack list-instances autosync
```

The screenshot shows the StackState CLI output and the corresponding StackState web interface for the 'Custom Synchronization' StackPack.

CLI Output:

```
[user@ip-172-30-0-122 ~]$ sts stackpack list-instances autosync
  id   name    status      version  last updated
179233792710522 - WAITING FOR DATA  3.1.1   Wed Aug 11 15:49:16 2021
```

Web Interface:

- Custom Synchronization:** Shows the StackPack details, including its version (3.1.1) and a note that it contains everything needed for automatic synchronization from an external topology source. A link to "Read more..." is present.
- Installed Instances:** A table showing one instance of the StackPack. The instance has a status of "INSTALLED. Waiting for data..." and was updated "Today, 17:48:15". It includes fields for "Instance type (source identifier)" (set to "demo") and "Instance URL" (set to "demo"). To the right of the table, a message says: "The Autosync StackPack is waiting for your action, please send some topology!"

The 'stackpack list-instances' command will return all installed instances of a certain StackPack. If you have a lot of instances of a certain StackPack installed, you can add a flag to the command to filter on status. In our scenario, we want to inspect the installation status of the custom synchronization StackPack using the internal StackPack name 'autosync'.

As can be seen here, the status of the StackPack is 'WAITING_FOR_DATA'. This means that the installation process succeeded. Most StackPacks wait for data to arrive to be processed by StackState Synchronization before the status of the StackPack instance flips to INSTALLED. Essentially, the StackPack is listening to a Kafka topic. If any message is received, the status will flip to INSTALLED.

StackState CLI – stackpack command

```
sts stackpack list-instances autosync
```

The StackState CLI command `sts stackpack list-instances autosync` is run, resulting in the following output:

```
[user@ip-172-30-0-122 ~]$ sts stackpack list-instances autosync
      id   name    status   version   last updated
-----+-----+-----+-----+-----+
 179233792710522 -  INSTALLED  3.1.1  Wed Aug 11 16:26:26 2021
```

An arrow points from the CLI output to the StackState UI interface.

Custom Synchronization
Version 3.1.1

This StackPack contains everything you need to automatically synchronize topology information from an external topology source.
[Read more...](#)

Installed Instances

New Instance	Updated
Active demo	Today, 17:49:16

Instance type (source identifier)
demo

Instance URL
demo

The Custom Synchronization StackPack is enabled
Congratulations! Your new Synchronization StackPack is correctly configured.

What's next

StackState

When StackState received data on the Kafka topic, the status of the StackPack instance will switch to INSTALLED. The StackPack instance is installed and can be used by users.

StackState CLI – stackpack command

```
sts stackpack uninstall autosync <instance id>
```

```
[user@ip-172-30-0-122 ~]$ sts stackpack list-instances autosync
id      name      status      version      last updated
-----  -----  -----
179233792710522 - INSTALLED  3.1.1       Wed Aug 11 16:26:26 2021
```

```
[user@ip-172-30-0-122 ~]$ sts stackpack uninstall autosync 179233792710522
Successfully triggered uninstallation of instance 179233792710522 of StackPack autosync. Check the status of the process using 'stackpack list-instances autosync' or debug any errors using 'serverlog show stackpack/autosync'
```

```
[user@ip-172-30-0-122 ~]$ sts stackpack list-instances autosync
id      name      status      version      last updated
-----  -----  -----
```



The uninstallation process is quite simple. The ‘stackpack uninstall’ command is used to uninstall StackPack instances. The stackpack uninstall command accepts two arguments. The first argument of the command is the internal name of the StackPack you want to uninstall. The second, and last, argument is the StackPack instance id. A StackPack’s internal name can be found using the ‘stackpack list’ command and the StackPack instance id can be found using the ‘stackpack list-instances’ command.



Let's have a look at the StackState CLI's event command. This command can be used to send events to StackState

StackState CLI – event command

```
[user@ip-172-30-0-122 ~]$ sts event -h
usage: cli event [-h] {send} ...
positional arguments:
  command_class
    send      Send an event to StackState
optional arguments:
  -h, --help   show this help message and exit
[sts@ip-172-30-0-122 ~]$ sts event send -h
usage: cli event send [-h] [-t TITLE] [-x TEXT] [-s timestamp TIMESTAMP]
  [-s SOURCE] [-i ELEMENT_ID] [-l LINKS]
  [-c (Changes,Anomalies,Activities,Alerts,Others)]
  [-d DATA] [-t TAGS]
  type
positional arguments:
  type            Event type
optional arguments:
  -h, --help       show this help message and exit
  -t TITLE        event title
  -x TEXT         event description
  --timestamp TIMESTAMP
  The timestamp set on the message. A UTC timestamp in
  seconds. Defaults to current time
  -s SOURCE, --source SOURCE
  a source of the event, e.g. PagerDuty
  --source-id SOURCE_ID
  an identifier of an event from the source system, e.g.
  incident number for PagerDuty source
  -i (ELEMENT_IDS [ELEMENT_IDS ...]), --element-ids (ELEMENT_IDS [ELEMENT_IDS ...])
  identifiers of the stack element this event is related
  to separated by space: -i urn:host:/example.org
  urn:service-
  instance://myservice@example.org:2579;1288784265. If
  not specified, identifier for hostname from CLI
  configuration will be used
  --links [LINKS [LINKS ...]]
  a list of links with titles divided by colon with
  space, e.g.--links 'PageDuty Incident:
  https://example.pagerduty.com/incidents/9812371961956'
  'Conference: https://stackstate.zoom.us/j/69255663067?p
  wd=V1L2kjdl8jw0s94dhbb'
  -c (Changes,Anomalies,Activities,Alerts,Others) --category (Changes,Anomalies,Activities,Alerts,Others)
  -d DATA, --data DATA JSON with event extra data, e.g. -d
  '{"impacted_service": "payments", "urgency": "high"}'
  -t TAGS, --tags TAGS A list of tags in the format
  tag key:tag value,tag key:tag value
[sts@ip-172-30-0-122 ~]$
```

The ‘event send’ command can send an event to StackState. You have several options to construct the contents of the event. You can give the event a title, body/text, a timestamp, source name, source identifier, links, key/value tags, and other data.

An event’s source is used to identify the data source. This can be the name of the data source. The source-id relates to the data source, to identify the event related to what happened in the data source. If the event has an identifier in the data source, you could use the same identifier as source-id.

The element_ids argument specifies the components or relations the event belongs to. An element id need to match one or more identifiers of a component or relation. You can find the identifiers of a component or relation in the details pane. One event can have one or more identifiers set that match one or more topology elements. If no element identifier is set on the event, or not topology element is matched, the event will not show up in the event perspective. The event can still be used as an event on a log stream when no element identifiers are set.

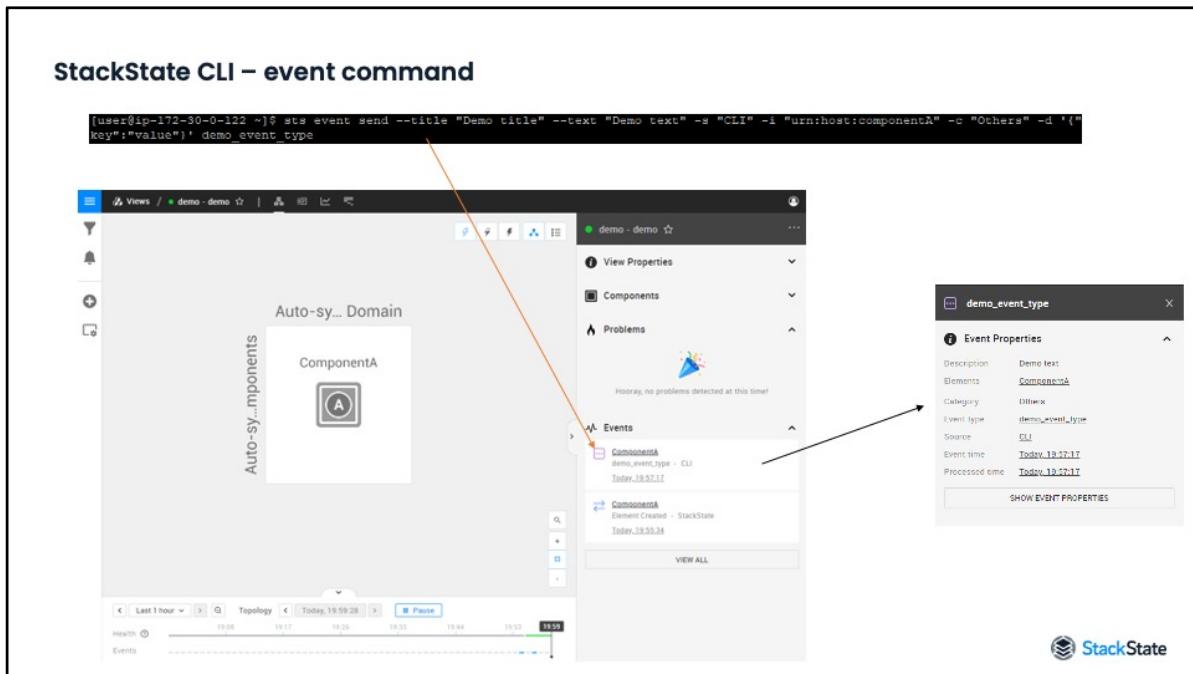
A link can be added to the event to provide a backlink to the data source, if relevant.

Events in StackState have a category assigned to it, you can set the category of the event to changes, anomalies, activities, alerts, or other.

Lastly, metadata can be added to the event as data or tags. This information is shown in the user interface if you click on the event.

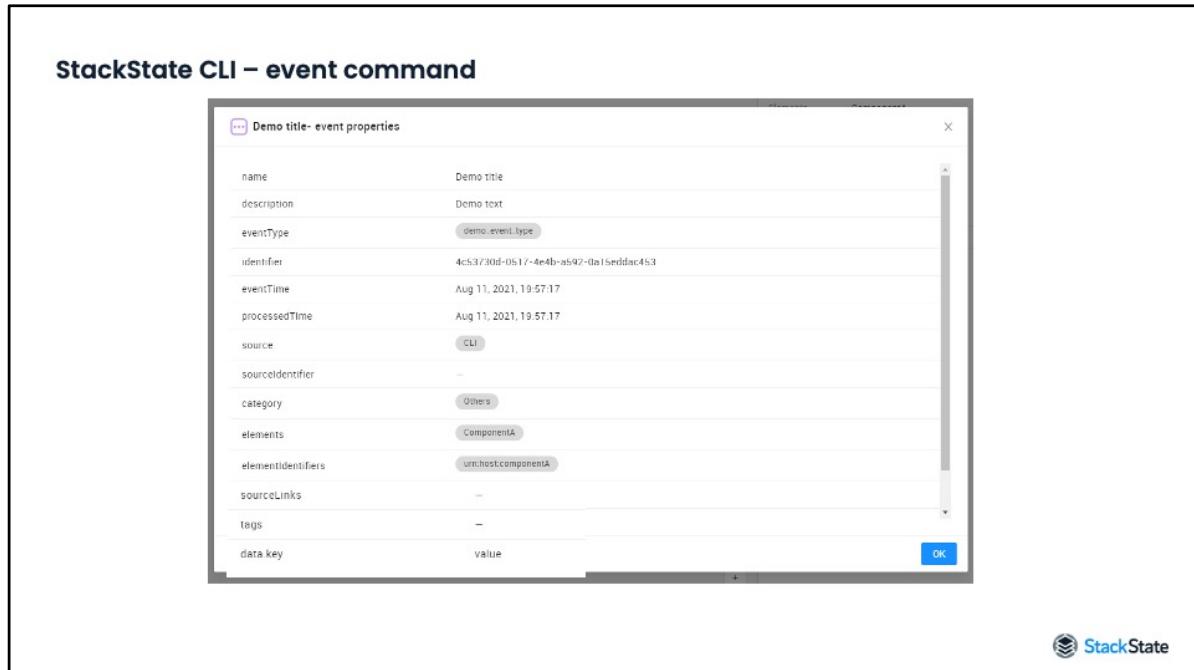
The only mandatory argument to the command is the event type. The event type specifies what the event is about, what it describes. For example, an event having the event type called 'host_status' might describe a host's health status.

The category, event type, event source, and tags can be filtered on in the StackState user interface.

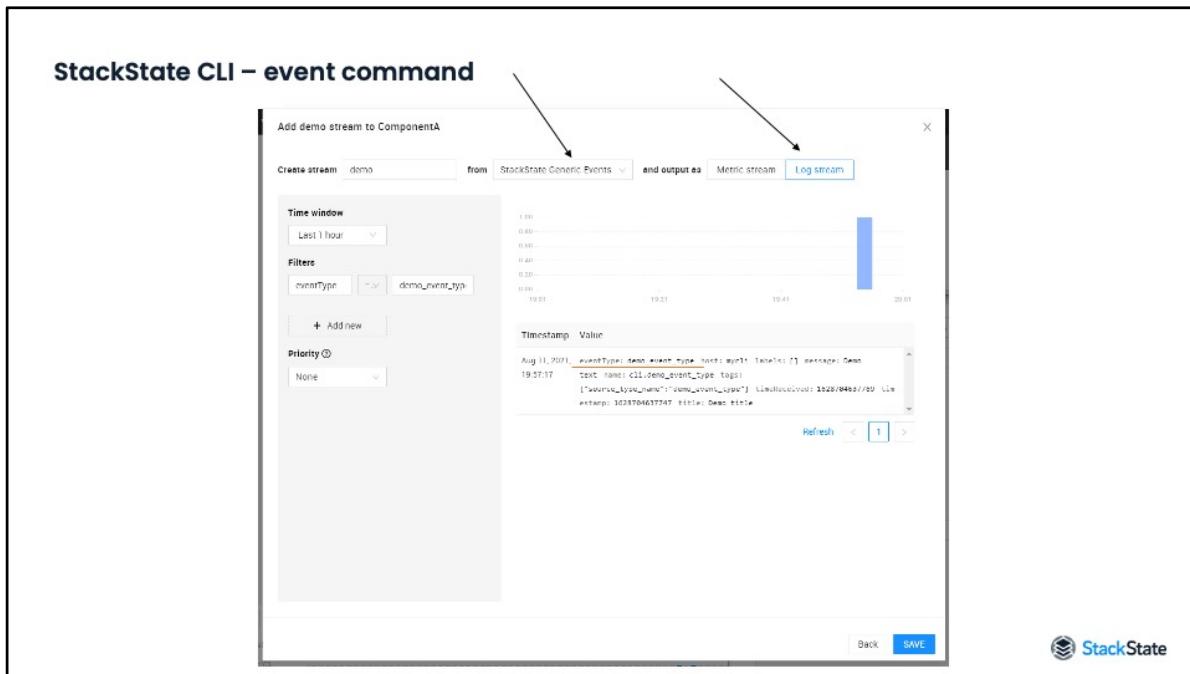


Consider the following example. The 'event send' command used in this example sends an event to StackState where the title is set to 'demo title' and the event text to 'demo text'. The source is set to "CLI" indicating that the event was generated by the CLI. In this case we have one element id set, using the -i flag. The component shown on the left has one identifier set and that matches exactly to what is provided in the CLI command. The category of the event is set to "Others". Additional key/value metadata is added too. Lastly, the event type is set.

After sending the event, StackState will show the event in the event perspective when it matches a topology element in that view. In this case there is a component found matching the identifier. Clicking on the event will show the event's details. Clicking on the 'show event properties' button will show more information about the event.



Here's the event properties pane for the example event. As you can see, the event's information is shown here.



Next to the event showing up in the event perspective when an element identifier is set, the event is also available as an event in the StackState Generic Events data source. This means that you can add an log stream on a component or relation that listens to these events.



Let's have a look at the StackState CLI's datasource command.

StackState CLI – datasource command

```
[user@ip-172-30-0-122 ~]$ sts datasource -h
usage: cli datasource [-h] {list} ...

positional arguments:
  command_class
  {list}
  list      List all datasources

optional arguments:
  -h, --help    show this help message and exit
```



```
[user@ip-172-30-0-122 ~]$ sts datasource list
   id type name           description          comes by          manual last updated
-----+-----+-----+-----+-----+-----+-----+-----+
2738839208651467 DataSource StackState Metrics -              urn:stackpack:common False  Wed Aug 11 18:41:16 2021
100122427675605 DataSource demo -                  urn:stackpack:common True   Wed Aug 11 13:49:16 2021
139594492328865 DataSource StackState State Events -             urn:stackpack:common False  Wed Aug 11 13:51:16 2021
101144950575705 DataSource StackState Audit Events -           urn:stackpack:common False  Wed Aug 11 13:51:16 2021
357047941575172 DataSource StackState Generic Events -         urn:stackpack:common False  Wed Aug 11 13:41:16 2021
239083564276411 DataSource StackState Multi Metrics -        urn:stackpack:common False  Wed Aug 11 18:41:16 2021
```

 StackState

The ‘datasource list’ command lists all the data sources that are available in StackState.

The example at the bottom shows a list of data sources returned by the ‘datasource list’ command. The id column shows the database id of the data source. The name column shows the data source’s name, as can be found in the StackState settings pages. A data source can have a description. The identifier is shown. By looking at the identifier you can see that a couple data sources belong to the StackPacks and one does not have an identifier. The demo data source was manually created, and no identifier was set. The manual column also indicates that the data source was created manually. Lastly, a last updated timestamp is shown.

This may be a useful command if you want to export a data source’s configuration for backup or StackPack development purposes.



Let's have a look at the StackState CLI's trace command.

StackState CLI – trace command

```
luser@ip-172-30-0-122 ~]$ sts trace -h
usage: cli trace [-h] {send,example} ...
positional arguments:
  command_class
    {send,example}
    send           Send traces
    example        Show example input file for 'sts trace send' command
optional arguments:
  -h, --help       show this help message and exit
```



The 'trace' command can be used to send traces to StackState. You need the StackState Agent StackPack installed. The Synchronization included in the StackPack Agent StackPack deals with the topology information derived from traces. An example is provided to get you going.

StackState CLI – trace command

```
[user@ip-172-30-0-122 ~]$ sts trace example -h
usage: cli trace example [-h] -f {plain,template}

optional arguments:
-h, --help            show this help message and exit
-f {plain,template}
      Desired format: "plain" - VAML file with a simple flat
      list of trace spans, no magic happens "template" -
      YAML file with traces list. Where every trace is
      defined by a span tree. Trace & span IDs are generated
      automatically. Start times of spans are defined as an
      offset to basetime (current time by default). This
      format is useful for scenarios where you want simply
      define a structure and get it to Trace Perspective
      asap (some demo or bug report)

[...]
```

(user@ip-172-30-0-122 ~]\$ sts trace example -f plain

```
spans:
- trace_id: 83434272453424
  span_id: 10
  service: trace_api
  name: Traces API
  resource: search
  type: web
  start: 2020-09-18T15:00:00.000000+0200
  duration: 1000 # ms
  meta:
    http.method: GET
    http.url: https://trace.stackstate.infra.company.com/list
    http.host: trace.stackstate.infra.company.com
    span.serviceName: traceAPI
    span.serviceURN: urn:service:/traceAPI
    span.serviceInstanceURN: urn:service-instance:/traceAPI:/feawj2
    # Traefik agent meta
    sta.origin: traefik
    span.kind: client
- trace_id: 83434272453424
  span_id: 20
  parent_id: 10
  service: elasticsearch
  name: Stackstate Elastic
  resource: index.list
  type: db
```



Let's have a brief look at the trace example provided by the CLI. The example is returned to standard output, to your terminal. You can either format the example as plain text or a template. Formatting the example as plain text will show you the example with absolute values, values like timestamps. If you use the template format, timestamps will be returned differently such that upon execution the timestamps are relative to wall clock time.

StackState CLI – trace command

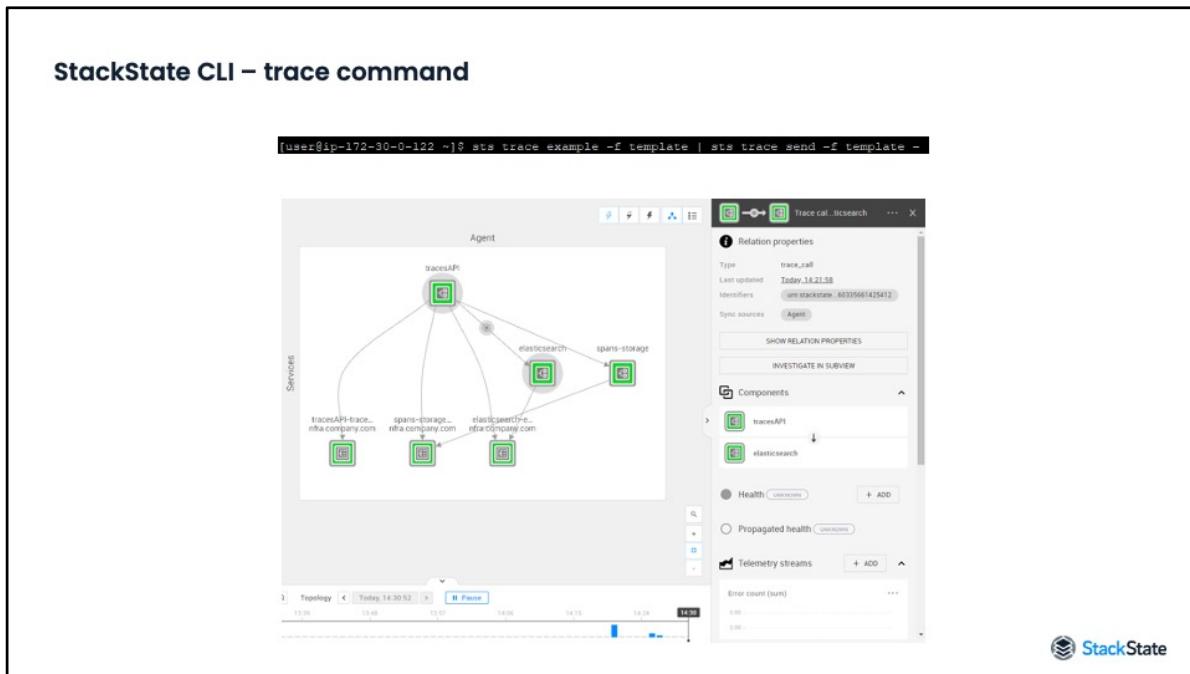
```
[user@ip-172-30-0-122 ~]$ sts trace send -h
usage: cli trace send [-h] [-f {plain,template}] file

positional arguments:
  file                Input file with traces definition (see 'sts trace
                      example' command). Use '-' for standard input

optional arguments:
  -h, --help           show this help message and exit
  -f {plain,template}, --format {plain,template}
                      Format of input file: *plain* - YAML file with a
                      simple flat list of trace spans, no magic happens
                      *template* - YAML file with traces list. Where every
                      trace is defined by a span tree. Trace & span IDs are
                      generated automatically. Start times of spans are
                      defined as an offset to basetime (current time by
                      default). This format is useful for scenarios where
                      you want simply define a structure and get it to Trace
                      Perspective asap (some demo or bug report)
```



Sending trace information to StackState can be done with the ‘trace send’ command. Similar to the command that generates the example, the sending command also expects a format. By specifying the format, the send command knows whether it should do processing or just send out the trace data. The command can read trace data from file or from standard in.



Let's send out the trace example to StackState. The StackState Agent StackPack is already installed. We use the trace example command, using template format, to return the example trace data to standard out. The output is then redirected to the 'trace send' command for sending to StackState.

The result is shown in StackState, here's the 'Agent – Trace Services – All' view. The components and relations shown here were extracted from the example trace data. You can switch to the trace of the same view perspectives to view the trace information.



Let's have a look at the StackState CLI's metric command.

StackState CLI – metric command

```
[user@ip-172-30-0-122 ~]$ sts metric -h
usage: cli metric [-h] {send,get} ...
positional arguments:
  command class
    {send,get}
      send      Send metrics.
      get       Get metric.
optional arguments:
  -h, --help  show this help message and exit.
```



The metric command can be used to send metrics to StackState or to query a StackState metric data source. We'll look at both options.

StackState CLI – metric send command

```
[user@ip-172-30-0-122 ~]$ oss metric send -h
usage: oss metric send [-h] [-p PERIOD] [-b BANDWIDTH] [-r SAMPLERATE]
                        [-y TYPE] [-t TAGS] [--timestamp TIMESTAMP]
                        [-n noise NOISE]
                        [--linear | --wave | --csv | --baseline]
                        name [metric_value]

positional arguments:
  name                  Metric name,
  metric value          Metric value.

optional arguments:
  -h, --help             show this help message and exit
  -p PERIOD             The period for which metrics will be generated.
                        -sampleRate will determine how many datapoints will be
                        created within this period.
                        supported with any combination of w (weeks), d (days),
                        h (hours), m (minutes), s (seconds). examples: "1w4d"
                        "40m 20s" 40 minutes and 20 seconds "1h4m" 1
                        hour, 4 minutes and 40 seconds. "1x 2d 3h 4m 5s" 1 week, 2 days,
                        3 hours, 4 minutes and 5 seconds defaults to 0s
  -b BANDWIDTH          The maximum value used when generating
                        values. In the format min-max. defaults to 10-90
  -r SAMPLERATE          The sampling rate used in seconds between samples.
                        Defaults to 15
  -y TYPE, --type TYPE  the type of metric
  -t TAGS, --tags TAGS  A list of tags in the format
                        key, key,tag_value, tag_key,tag_value
  --timestamp TIMESTAMP  The timestamp act on the message. For a range of
                        metrics it is the time of the most recent metric. A
                        UTC timestamp in seconds. Defaults to current time
  --noise NOISE          Add noise to the generated metric values. A value
                        between -noise and noise will be added to all values
                        generated. defaults to 0
  --linear               A linear progression from a min value to a max value
                        set by the --bandwidth.
  --wave                 Create one wave per day between a min value and max
                        value. Set the --bandwidth to a value less than 1.
  --csv                 Load data from a csv file --input
  --baseline             Create a baseline pattern. Values are at a minimum at
                        night and maximum during the day. these values are set
                        with --bandwidth. saturday and sunday have reduced
                        values during the day.
```



The ‘metric send’ command metrics to StackState. The command has several options to tweak the metrics you want to send. You can send a couple of metrics or a single value to the StackState Metrics data source. You can even add noise to the metric values.

Sending a single value requires you to set the metric name and provide a value. Sending multiple metrics can be done by selecting a period, bandwidth, sample rate, etc. Don’t forget to add some meta data to be able to filter the metric in a metric stream.

There are quite some options, let’s look at some examples.

StackState CLI – metric send command

```
[user@ip-172-30-0-122 ~]$ sts metric send --period "1w" --bandwidth 0-100 --type cli_metric_type --tags "key:value" --linear cli_metric
[user@ip-172-30-0-122 ~]$
```

The screenshot shows the StackState Metrics interface with the following details:

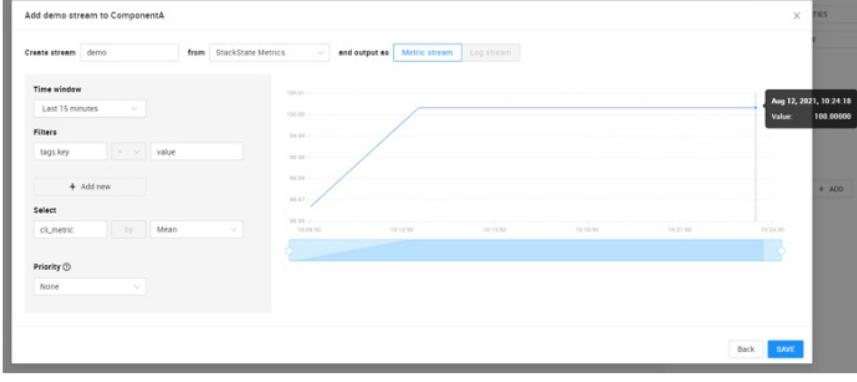
- Create stream:** demo
- From:** StackState Metrics
- And output as:** Metric stream
- Time window:** Last 7 days
- Filters:** tags key: value
- Select:** cli_metric by Mean
- Priority:** None
- Graph:** A line chart showing a linear increase from 0 to 100 over the last 7 days. A specific data point is highlighted: Aug 5, 2021, 10:30:00 with a value of 0.31374.
- Buttons:** Back, SAVE, StackState logo

Consider the following example. In this example the CLI will send one week's worth of metrics. The metric values will be between 0 and 100. The value will increase linearly from 0 to 100 over the span of the week. There is a metric type set and the metric has a name. The type might be something like cpu_percentage and the name might be the cpu or host name. A key/value pair is added as a tag to be able to filter when creating a metric stream.

In the stream below, you can see the linear line going from 0 to 100 when the time window is set to last 7 days. The metric name is used in the 'Select' input box.

StackState CLI – metric send command

```
[user@ip-172-30-0-122 ~]$ sts metric send --tags "key:value" cli_metric 100
```



The screenshot shows the StackState Metrics interface. A modal window titled 'Add demo stream to Component' is open, showing a line chart for a stream named 'demo'. The chart displays a single data point at the time 'Aug 12, 2021, 10:24:18' with a value of '100.00000'. The interface includes various configuration options such as 'Time window' (Last 15 minutes), 'Filters' (tags.key), 'Select' (e.g., 'cli_metric' by Mean), and 'Priority' (None). The StackState logo is visible in the bottom right corner.

In this example we are sending a single metric to StackState. The stream filters are the same compared to the previous example. The 'metric send' command is able to send out the metric using only the metric name and the metric value, the tags are optional. Omitting the tags here would result in the metric not being shown, because of a filter mismatch.

StackState CLI – metric get command

```
(user@ip-172-30-0-122 ~)$ sts metric get -h
usage: cli metric get [-h] [-q {QUERY}] [-s {START}) (-e {END})
                      [-p {PAGESIZE}] [-f {OUTPUTFILE}] [-c {(csv,json)}]
                      [-t {METRICFIELD}] [-m {MAXCOUNT}] [-b]
                      datasource

positional arguments:
  datasource           Telemetry data source id.

optional arguments:
  -h, --help            show this help message and exit
  -q {QUERY}, --query {QUERY}
                        Metric Query, e.g "host="localhost",tag.team="".
  -s {START}, --start {START}
                        Start timestamp
  -e {END}, --end {END}
                        End timestamp
  -p {PAGESIZE}, --pagesize {PAGESIZE}
                        Page size
  -f {OUTPUTFILE}, --outputfile {OUTPUTFILE}
                        Output file, metric-(datasource)-(query).csv by
                        default.
  -c {(csv,json)}, --outputfile-format {(csv,json)}
                        Output file format.
  -t {METRICFIELD}, --metricfield {METRICFIELD}
                        Value field name.
  -m {MAXCOUNT}, --maxcount {MAXCOUNT}
                        Maximum amount of messages to download.
  -b, --prettyprint    Pretty print output json.
```



The metric get command can be used to query StackState metric data sources. This may be the StackState metrics data source, or any external data source that is configured in StackState. The results are written to either a csv or json file. By default, the results are written to a csv file in the current working directory.

The query argument sets the filter you want to use, this is the exact same key/value pair as you would use on a metric stream in the StackState user interface, formatted differently. The value from the metric stream's 'Select' box goes into the metric field argument.

You can select a time range and a maximum number of metrics returned. The start and end timestamp arguments are expecting values in epoch milliseconds. The maximum number of metrics returned is by default limited to 1024 metrics.

The 'metric get' command accepts a data source that is already configured in StackState. The data source id can be found using the CLI's 'datasource list' command.

StackState CLI – metric get command

```
[user@ip-172-30-0-122 ~]$ sts datasource list
id type name
173839208651467 DataSource StackState Metrics
100122427675685 DataSource demo
```

```
[user@ip-172-30-0-122 ~]$ sts metric get -q "tags.key-value" --metricfield cli_metric --start `date --date="1 days ago" "+%s$ms`' -m 1000000 273839208651467
total downloaded count: 5514 messages.
```

Epoch ms

```
[user@ip-172-30-0-122 ~]$ head metric-273839208651467-tags_key_value.csv
timestamp,metric
2021-08-11 09:15:26,86.32936507941172
2021-08-11 09:15:41,86.33184523814188
2021-08-11 09:15:56,86.33432539697205
2021-08-11 09:16:11,86.33680555560221
2021-08-11 09:16:26,86.33928571433238
2021-08-11 09:16:41,86.34176587306254
```

```
[user@ip-172-30-0-122 ~]$ tail metric-273839208651467-tags_key_value.csv
2021-08-12 08:11:26,99.98015873024137
2021-08-12 08:11:41,99.98263888897154
2021-08-12 08:11:56,99.9851190477017
2021-08-12 08:12:11,99.98759920643187
2021-08-12 08:12:26,99.99007936516203
2021-08-12 08:12:41,99.9925595238922
2021-08-12 08:12:56,99.99503966262236
2021-08-12 08:13:11,99.99751984135253
2021-08-12 08:13:26,100.0000000000627
2021-08-12 08:24:19,100.0
```

StackState

Consider the following example. The examples from `metric send` did sent a week's worth of metric data and a single metric value. With the 'metric get' command we can query this data again.

Since these metrics were sent to the StackState Metrics data source, we need to get the data source id of the StackState Metrics data source using the 'datasource list' command. We use the same metric stream filter as the previous examples, a key/value pair as filter on tags. The metric field is set to the metric's name used previously. For demonstrational purposes we only get the last day worth of metric data. We pass a command between backticks to be evaluated before the CLI command is evaluated, the command gets the timestamp of a day ago, in milliseconds. The -m argument specifies the maximum number of metrics to return. The last argument is the data source id.

The output is written to a csv file. The top and bottom of the file is shown at the bottom on the slide. The csv has a header contains the timestamp and the actual metric value. Other information is put into the filename.



Let's have a look at the StackState CLI's topology command.

StackState CLI – topology command

```
[user@ip-172-30-0-122 ~]$ sts topology -h
usage: cli topology [-h] {send} ...

positional arguments:
  command_class
    send      Send topology using templates. Send topology using stdin.
              Templates may be placed in a separate directory. Please refer
              to documentation for full explanation.

optional arguments:
  -h, --help   show this help message and exit
```

```
[user@ip-172-30-0-122 ~]$ sts topology send -h
usage: cli topology send [-h] [--timestamp TIMESTAMP] {-d} {-f} [-raw]

optional arguments:
  -h, --help            show this help message and exit
  --timestamp TIMESTAMP
                        The timestamp set on the message. A UTC timestamp in
                        seconds. Defaults to current time
  -d, --dry-run          Show instead of send HTTP body.
  -f, --force            Send the topology without checking.
  -raw, --raw             Send the topology directly in the intake API format
```



The topology send command can be used to send components and relations to StackState. The topology send command works with local files that can be passed to the command via standard input. These files are templates for specific component and relation types defining a component or relation's structure. Another file defines the actual data and defines which template to use. The idea is that you define a component or relation template once such that the topology specification file is very clean. An example is provided in the CLI's zip package you can download from the download website. The example templates can be found in the templates directory.

There is no StackPack available, you can use the Custom synchronization StackPack to get started with the example provided in the zip package.

The topology send command by defaults uses templates. You can also send plain JSON to the StackState receiver using the --raw argument. The --raw argument does not use templates, it accepts JSON from standard input.



Let's have a look at the StackState CLI's health command.

StackState CLI – health command

```
[user@ip-172-30-0-151 ~]$ sts health -h
usage: cli health [-h] {list-streams,list-sub-streams,show,delete,send} ...

positional arguments:
  command_class
    {list-streams,list-sub-streams,show,delete,send}
      list-streams           List active health synchronization streams.
      list-sub-streams       List active health synchronization sub streams of a
                             stream.
      show                  Show the status of an active health synchronization
                             stream.
      delete                Delete a health synchronization stream.
      send                  Send health synchronization data to stackstate.

optional arguments:
  -h, --help              show this help message and exit
```



The ‘health’ command can be used to handle health synchronization streams and sub streams.

The ‘health list-streams’ command shows all the health synchronization streams currently active. The ‘health list-sub-streams’ command shows all health stream related sub streams currently active. The ‘health show’ command provides insights in a health synchronization stream. It is also possible to delete health synchronization streams are send data to one of the streams.

StackState CLI – health command

The screenshot shows two parts: the StackState CLI output and the StackState UI interface.

StackState CLI Output:

```
[user@ip-172-30-0-151 ~]$ sts health list-streams
stream urn                                sub stream count
-----
urn:health:sourceId:streamId               2
```



```
[user@ip-172-30-0-151 ~]$ sts health list-sub-streams urn:health:sourceId:streamId
sub stream id      check state count   repeat interval (Seconds)    expiry (Seconds)
-----
subStreamId        2                      50                         200
```

StackState UI (ComponentA details):

The UI displays the following details for ComponentA:

- Component properties:**
 - Type: application
 - Last updated: Today, 15.23.35
 - Labels: demo demo
 - Identifiers: urn:host:componentA
 - Sync sources: Synchronization demo demo
- Actions:**
 - Show all dependencies
 - Health: DEVIATING (with + ADD button)
 - Health Monitor: DEVIATING (with Show details link)
 - sourced
 - CPU monitor
 - Propagated health: DEVIATING

A timeline at the bottom shows time points: 16.18, 16.27, 16.36, and 16.42.

Here's an example of the 'health list-streams' command. In this case, it shows a single health synchronization stream. The stream shown here has one sub stream. The stream itself is also counts towards the sub stream count in the 'health list-streams' command as it is considered to be a default sub stream. Listing all sub streams of a stream shows the number of check states and the stream's repeat interval and expiry settings.

StackState CLI – health command

```
[user@ip-172-30-0-151 ~]$ sts health show urn:health:sourceId:streamId
Aggregate metrics for the stream and all substreams:

metric          value between now and 300 seconds ago  value between 300 and 600 seconds ago  value between 600 and 900 seconds ago
-----          -----
latency (Seconds)          0.178          0.178          0.106
messages processed (per second) 0.0233333 0.004666666666666667 0.003333333333333335
check states created (per second) 0.006 0.003333333333333335 0.003333333333333335
check states updated (per second) 0.01 -
check states deleted (per second) 0.00333333 0.00333333333333335 -

Errors for non-existing sub streams:

error message          error occurrence count
-----          -----
substream with id 'subStreamId' expired          2
main stream expired          1

[user@ip-172-30-0-151 ~]$ sts health show urn:health:sourceId:streamId -s "subStreamId"
Synchronized check state count: 2
Repeat interval (Seconds): 50
Expiry (Seconds): 200
Synchronization errors:
error message          error occurrence count
-----          -----
Synchronization warning:
metric          value between now and 300 seconds ago  value between 300 and 600 seconds ago  value between 600 and 900 seconds ago
-----          -----
latency (Seconds)          0.179          0.179          0.106
messages processed (per second) 0.0233333 0.004666666666666675 0.003333333333333335
check states created (per second) 0.006 0.003333333333333335 0.003333333333333335
check states updated (per second) 0.01 -
check states deleted (per second) 0.00333333 0.00333333333333335
[user@ip-172-30-0-151 ~]$ [user@ip-172-30-0-151 ~]$ sts health show urn:health:sourceId:streamId --> "subStreamId" --k
Check states with identifier matching exactly 1 topology element: X
Check states with identifier which has no matching topology element:
check state id  topology element identifier
Check states with identifier which has multiple matching topology elements:
check state id  topology element identifier  number of matched topology elements

```



The `health show` command is useful to find out more information about a stream, or a stream's sub stream. Information for a given substream can be obtained by passing the stream id to the `-s` argument. The 'health show' command provides insights into latency, number of messages processed, check state updates, errors, etc. It also can provide a list of topology elements that did not match the identifiers received in a health synchronization stream.

StackState CLI – health command

```
[user@ip-172-30-0-151 ~]$ sts health send start -r 50 urn:health:sourceId:streamId
[user@ip-172-30-0-151 ~]$ sts health send check-state urn:health:sourceId:streamId checkStateId "Health Monitor" urn:host:componentA deviating
[user@ip-172-30-0-151 ~]$ sts health send stop urn:health:sourceId:streamId
```

Here's an example of sending a check state using a health synchronization stream with the 'health send' command. We're sending a health snapshot in this case.

We need to start the snapshot first, by providing the start argument. We're setting the repeat interval to 50 seconds and we set a URN identifier for the stream. Next command we use is the 'health send check-state' command to send a check state to the stream. The first argument given is the URN identifier of the stream we are writing the check state to. The second argument is the check state id, which can represent an identifier in the data source. The third argument is the name of the check state that will be displayed as can be seen on this component. The forth argument is a topology identifier that matches the component's identifier in order for the check state to be shown. The last argument describes the health state value, or the color. Finally, we need to stop the stream snapshot by providing the stream's URN to the 'health send stop' command.

StackState CLI – health command

```
[user@ip-172-30-0-151 ~]$ sts health list-streams
stream urn                                sub stream count
-----
urn:health:postman:streamId                1
urn:health:sourceId:streamId               1
urn:health:sourceId:streamId2              1
```

```
[user@ip-172-30-0-151 ~]$ sts health delete urn:health:sourceId:streamId1
[user@ip-172-30-0-151 ~]$ sts health delete urn:health:postman:streamId
[user@ip-172-30-0-151 ~]$ sts health list-streams
stream urn                                sub stream count
-----
urn:health:sourceId:streamId               1
```



The ‘health delete’ command can be used to delete a health synchronization stream. In this example we delete two out of three streams present. Deleting a stream will also delete the sub streams of that stream. Deleting a stream might cause the check states to disappear from certain topology element. The health states will be updated automatically.



Let's have a look at the StackState CLI's permission command.

StackState CLI – permission command

```
[user@ip-172-30-0-151 ~]$ sts permission -h
usage: cli permission [-h] {show,list,grant,revoke} ...
positional arguments:
  command_class
    {show,list,grant,revoke}
      show           Show permissions of a subject.
      list           List all types of permissions.
      grant          Grant a permission to a subject.
      revoke         Revoke a permission from a subject.
optional arguments:
  -h, --help        show this help message and exit
```



The permission command is used for roll-based access control, or RBAC. We'll cover RBAC in more detail in the section on Security. Permissions are set to subjects in StackState. A combination of a subject and a set of permissions are called a role. A role describes a group of users that can access specific data set.

The 'permission show' command shows all permissions granted to a specific subject. The 'permission list' command shows all permissions available in StackState that can be used on subjects. Finally, permissions can be granted to and revoked from subjects. Granting permissions require a resource to scope the permission to. For system-wide permissions, the resource can be set to 'system'. For permissions like 'access-view', we need to specify a view's name as the resource to specifically grant access to only that view. If you want access to all views, you can set the resource to 'everything'.

StackState CLI – permission command

```
(user@ip-172-30-0-151 ~]$ sts permission show stackstate-admin
subject          permission           resource
-----
stackstate-admin manage-annotations      system
stackstate-admin execute-scripts        system
stackstate-admin read-settings         system
stackstate-admin access-explore        system
stackstate-admin access-analytics       system
stackstate-admin access-synchronization-data system
stackstate-admin access-log-data       system
stackstate-admin execute-node-sync     system
stackstate-admin manage-event-handlers system
stackstate-admin access-topo-data      system
stackstate-admin manage-topology-elements system
stackstate-admin import-settings       system
stackstate-admin export-settings       system
stackstate-admin execute-restricted-scripts system
stackstate-admin perform-custom-query   system
stackstate-admin update-permissions     system
stackstate-admin read-permissions       system
stackstate-admin manage-telemetry-streams system
stackstate-admin execute-component-templates system
stackstate-admin update-visualization    system
stackstate-admin upload-stackpacks      system
stackstate-admin create-views          system
stackstate-admin update-settings        system
stackstate-admin manage-stackpacks     system
stackstate-admin execute-component-actions system
stackstate-admin access-view           everything
stackstate-admin save-view            everything
stackstate-admin delete-view          everything
```



The ‘permission show’ command shows all permissions granted to a subject. In this example, the defaults are listed for the stackstate-admin subject. You can see that most permissions are set to the stackstate-admin subject. Most permissions are set to the system resource, these are global permissions. Permissions access-view, save-view, delete-view can be set to specific views. In the stackstate-admin subject’s case, the user can access, save, or delete all views.



Let's have a look at the StackState CLI's subject command.

StackState CLI – subject command

```
[user@ip-172-30-0-151 ~]$ sts subject -h
usage: cli subject [-h] {show-all,show,delete,save} ...

positional arguments:
  command class
    {show-all,show,delete,save}
  show-all           Displays a list of all the existing subjects.
  show              Show a specific subject
  delete            Removes the given subject
  save              Creates a new subject or updates an existing one
                    (resolved by <handle>). Subjects are security entities
                    (such as users or groups) to which all permissions are
                    tied to (see help on 'permissions' for further
                    information).

optional arguments:
  -h, --help          show this help message and exit
```



The subject command is used for roll-based access control, or RBAC. We'll cover RBAC in more detail in the section on Security.

The 'subject show-all' command shows all subjects that are currently configured in StackState. Similar to the 'subject show-all' command, the 'subject show' commands shows information about a single subject. Lastly, the CLI has to be used to create or delete subjects in StackState.

StackState CLI – subject command

```
[user@ip-172-30-0-151 ~]$ sts subject show-all
Handle          Scope Query
-----
stackstate-guest
stackstate-platform-admin
stackstate-admin
stackstate-power-user
```



Here's an example of the 'subject show-all' command. It contains the default subjects after installing StackState. These match the predefined roles for administrators, platform administrators, power users, and guests. A scope is defined as STQL query and is prefixed before every view's query upon opening the view. No scope is defined for these subjects. No scope set means that these users have no restrictions on the components and relations they can see in views.

StackState CLI – subject command

```
[user@ip-172-30-0-151 ~]$ sts subject save -h
usage: cli subject save [-h] handle scope [version]

positional arguments:
  handle      The subject handle.
  scope       The query in STQL that will be prepended to every topology
             element retrieved in StackState. For example, if your scope is
             "label = \"A\"", then all STQL executed in StackState (e.g.
             Retrieving topology) will only return elements that have the
             label A.
  version     The STQL version of the scope query. (default: 0.0.1)

optional arguments:
  -h, --help  show this help message and exit

[User@ip-172-30-0-151 ~]$ sts subject save agent-only-subject 'domain = "Agent"'
[User@ip-172-30-0-151 ~]$
[User@ip-172-30-0-151 ~]$
[User@ip-172-30-0-151 ~]$ sts subject show agent-only-subject
Handle          Scope Query
-----
agent-only-subject domain = "Agent"

[User@ip-172-30-0-151 ~]$ sts permission show agent-only-subject
subject    permission    resource
-----  -----  -----

```



Here is an example of adding a subject. In this example we will create a subject that restricts the user to the domain called Agent. Components in other domains cannot be viewed.

The first argument is a handle, or the subject's name. We'll set the handle to 'agent-only-subject'. The second argument is the scope. The scope is a STQL query that limits the viewed topology whenever a view is opened. Essentially, a view's query is prepended with the subject's scope before showing the results in a view. This results in no components to be visible that do not match the subject's scope.

Deleting a subject is a matter of passing the scope's handle to the 'subject delete' command.

Making changes to scopes or permissions will cause affected users to be logged out.



Let's have a look at the StackState CLI's subscription command.

StackState CLI – subscription command

```
[user@ip-172-30-0-151 ~]$ sts subscription -h
usage: cli subscription [-h] {show,update} ...

positional arguments:
  command_class
  {show,update}
    show            Show current subscription
    update          Change the current subscription by providing a new license
                    key

optional arguments:
  -h, --help        show this help message and exit
```

```
[user@ip-172-30-0-151 ~]$ sts subscription update -h
usage: cli subscription update [-h] <license-key>

positional arguments:
  <license-key>  The new license key

optional arguments:
  -h, --help        show this help message and exit
```



The subscription command is used for managing the StackState license key. The 'subscription show' command shows the validity of the current StackState license key. The license key can be updated using the 'subscription update' command.



Let's have a look at the StackState CLI's anomaly command.

StackState CLI – anomaly command

```
[user@ip-172-30-0-122 ~]$ sts anomaly -h
usage: cli anomaly [-h] {send} ...
positional arguments:
  command_class
    (send)
      send      Sends anomaly annotation.
optional arguments:
  -h, --help      show this help message and exit

[User@ip-172-30-0-122 ~]$ sts anomaly send -h
usage: cli anomaly send [-h] --component-name COMPONENT_NAME --stream-name STREAM_NAME --start-time START_TIME
                           [--anomaly-direction {RISE,DROP}] [--duration DURATION]
                           [--severity {HIGH,MEDIUM,LOW}]
                           [--severity-score SEVERITY_SCORE] [--name NAME]
                           [--description DESCRIPTION]
optional arguments:
  -h, --help      show this help message and exit
  --component-name COMPONENT_NAME
                  Component name.
  --stream-name STREAM_NAME
                  Stream name.
  --start-time START_TIME
                  Anomaly start time. e.g. absolute timestamp value or
                  relative, e.g. -10m - 10 minutes ago, possible time
                  units: m - minutes, h - hours. (To overcome parsing
                  issues please specify relative time in the format
                  --start-time=-10m)
  --anomaly-direction {RISE,DROP}
                  Anomaly direction.
  --duration DURATION
                  Anomaly duration (seconds).
  --severity {HIGH,MEDIUM,LOW}
                  Anomaly severity.
  --severity-score SEVERITY_SCORE
                  Anomaly severity score.
  --name NAME
                  Anomaly name field contents.
  --description DESCRIPTION
                  Anomaly description field contents.
```

StackState

The ‘anomaly send’ command can be used to send simulated anomalies to StackState. The anomalies are shown on a component’s stream and in the event perspective. The command needs a component and stream name. The start time and duration arguments form the time window in which the event has occurred. You can set an anomaly direction, severity, and a severity score. The default severity is HIGH when omitted. The severity score is a value between 0 and 1, where 1 equals HIGH. The name and description will be shown on the event in the event perspective.

Let’s have a look at an example.

StackState CLI – anomaly command

```
[user@ip-172-30-0-122 ~]$ sts anomaly send --component-name "ComponentA" \
> --stream-name "demo" \
> --start-time=-10m \
> --anomaly-direction RISE \
> --duration "300" \
> --severity-score 1 \
> --name cli_anomaly \
> --description "demo anomaly"
Getting component for 'ComponentA'...
Getting stream 'demo' ...
The request has been accepted for processing, but the processing has not been completed.
```

Consider the following example where we have a single component, called ComponentA, which has a metric stream. The metric stream is named 'demo'. We pass the component's name and the name of the stream to the 'anomaly send' command. The start time of the anomaly is relative 10 minutes ago and the duration of the anomaly is set to 300 seconds, or 5 minutes. The severity score is set to 1, or 100%.

The results can be seen in the metric stream's chart and on the event perspective. The event from the event perspective is shown here on the right.

Security



Let's have a look at the security aspects of StackState.

Security

- Networking
- HTTPS/TLS
- RBAC
- Agent v2 password management



In this section, we'll be covering networking, HTTPS, Roll-based access control, and Agent v2's password management.

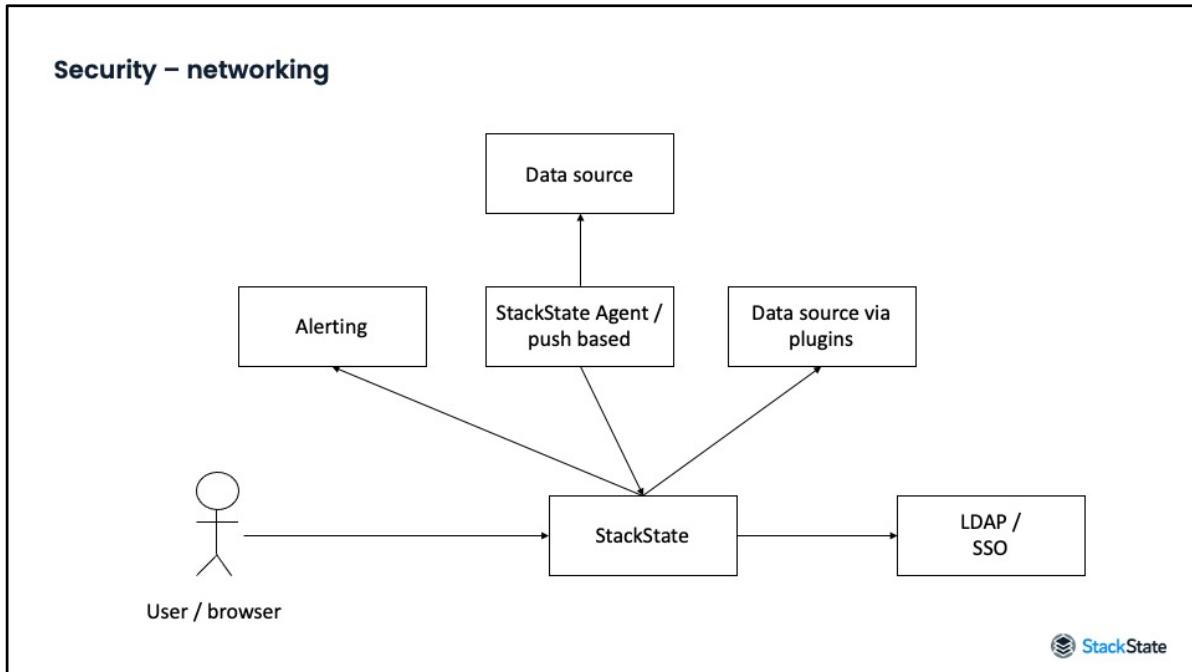
In the section about networking we'll discuss the networking requirements such that StackState can connect to data sources. In the HTTPS/TLS section we'll cover secure connections for the user and data sources. We talk about roll-based access control in the RBAC section. Finally, we'll have a section on secrets in the StackState Agent.

Security

Networking



Let's have a look at networking.



The user connects to the StackState user interface using a web browser. There needs to be a network path between the user and the user interface.

When a LDAP or single sign-on provider is configured, there needs to be a network path between StackState and the id provider.

Depending on the data sources connected to StackState, a networking connection is needed between the data source and StackState. For the StackState telemetry plugins, StackState initiates a connection to the data source directly. For integrations using the StackState Agent, there needs to be an open networking connection between the StackState Agent and StackState. The StackState agent will initiate the connection to StackState. Depending on integration, the StackState Agent initiates connections to the data sources it supports. The specific requirements for those connections can be found in the StackPacks.

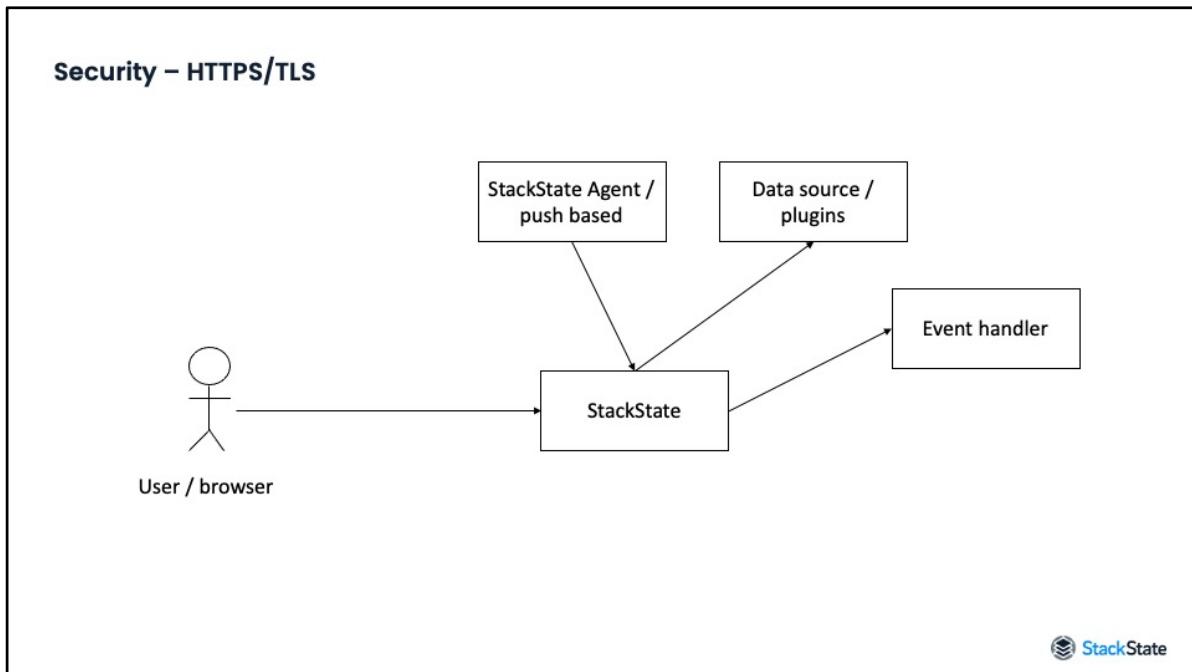
Most event handler functions require network communication. Network connections for configured event handlers need to be available. Which

networking connections that are required needs to be determined based on the event handler used.

In case your organization makes use of a proxy server, the StackState Agent and/or StackState must be configured to use that proxy server.



Let's have a look at secured connections.



By default, StackState does have a TLS certificate configured. You can supply your own to secure the communication between the browser and StackState. It is possible to add a reverse proxy that terminates TLS between the browser and StackState. The reverse proxy has to support web sockets. Alternatively, you can configure StackState and the StackState receiver to use a certificate.

Also, data sources and event handlers may use certificates that are either self-signed or a certificate that is not by default trusted by the JVM. These certificates need to be imported in StackState.

Ref:

https://docs.stackstate.com/setup/installation/linux_install/reverse_proxy

Ref:

https://docs.stackstate.com/setup/installation/linux_install/how_to_setup_tls_without_reverse_proxy

Ref: <https://docs.stackstate.com/configure/security/self-signed-cert-1>

Security

RBAC



Let's have a look at StackState's RBAC mechanism.

Security – RBAC

- Authentication methods
 - File based
 - LDAP
 - OpenID Connect (OIDC)
 - Keycloak
- Default roles:
 - Administrator
 - Platform Administrator
 - Power user
 - Guest
- Default users:
 - admin
 - platformAdmin
 - power-user
 - guest



StackState supports authentication via local accounts, via LDAP, via OpenId Connect or via Keycloak. Local accounts are defined in a file and is called ‘file based’ authentication and is enabled by default. LDAP can be used to connect to your organization’s Active Directory, for example. OpenID connect can be used if you want to use single sign-on.

StackState has four default roles; administrator, platform administrator, power user, and guest. The default users; admin, platformAdmin, power-user, and guest have the equivalently named role assigned. The administrator can see and change content of StackState. The administrator can install StackPacks, grant and revoke permissions, etc. The platform-administrator can perform management tasks on the StackState platform. The platform administrator can change data retention, clear the database, view logs, etc. The power user role is like the administrator role as it can change StackState configuration. However, the power user role cannot grant or revoke permissions, can install but not upload StackPacks, and have restrictive access in the analytics environment. The power user and guest roles can access all views but cannot make changes to them.

Ref:

https://docs.stackstate.com/configure/security/rbac/rbac_permissions

Security – RBAC

Linux: /opt/stackstate/etc/application_stackstate.conf

```
api {
    bindAddress = "0.0.0.0"
    port = 7070
    authentication {
        enabled = true
        authServer {
            authServerType = "stackstateAuthServer"
            stackstateAuthServer {
                defaultHashAlgorithm = "SHA256"
                defaultHashAlgorithm = "SHA256"
                # echo -n "topology-telemetry-time" | md5sum
                # Set the MD5 Hash into 'auth.password'
                # When changing users / passwords also change the admin api username / password
                logins = [
                    {username = "admin", password: ${stackstate.api.authentication.authServer.stackstateAuthServer.defaultPassword}, roles = ${stackstate.authorization.adminGroups} },
                    {username = "platformadmin", password: ${stackstate.api.authentication.authServer.stackstateAuthServer.defaultPassword}, roles = ${stackstate.authorization.platformAdminGroups} },
                    {username = "guest", password: ${stackstate.api.authentication.authServer.stackstateAuthServer.defaultPassword}, roles = ${stackstate.authorization.guestGroups} },
                    {username = "power", password: ${stackstate.api.authentication.authServer.stackstateAuthServer.defaultPassword}, roles = ${stackstate.authorization.powerUserGroups} }
                ]
            }
        }
    }
}
```

Kubernetes: --values authentication.yaml

```
stackstate:
  authentication:
    file:
      logins:
        - username: admin
          passwordHash: 5f4dcc3b5aa765d61d8327deb882cf99
          roles: [ stackstate-admin ]
        - username: platformadmin
          passwordHash: 5f4dcc3b5aa765d61d8327deb882cf99
          roles: [ stackstate-platform-admin ]
        - username: guest
          passwordHash: 5f4dcc3b5aa765d61d8327deb882cf99
          roles: [ stackstate-guest ]
        - username: power-user
          passwordHash: 5f4dcc3b5aa765d61d8327deb882cf99
          roles: [ stackstate-power-user ]
```



For the Linux distributions, file based authentication is configured in the application_stackstate.conf file in the etc/ directory of StackState's installation directory. By default, that is /opt/stackstate/etc/. For Kubernetes, an authentication.yaml file can be created and passed to the 'helm upgrade' command.

For Kubernetes, an example shown on the top right. You can see one entry per login. The username field defines the username that the user has to use to login with. The passwordHash field defines the password that is used for the user to login. Passwords are hashed, either MD5 or bcrypt can be used. The roles field defines which roles are assigned to the user. The default roles are used here. You can add roles here. Additional users can be added. This example shows file based authentication. The YAML key here is stackstate.authentication.file. To configure LDAP, OIDC, or Keycloak, you need to put the relevant configuration the correct location. For LDAP that's stackstate.authentication.ldap, for OIDC that's stackstate.authentication_oidc, and for Keycloak that's stackstate.authentication_keycloak. Please refer to the documentation

website for examples and how to configure the specific values required.

For Linux, the file based logins are defined in the stackstateAuthServer config block. Similar to Kubernetes, the username, password, and roles are defined here. The passwords shown here are are reference to another configuration entry containing the value. You can use the same or use a MD5 or bcrypt hash directly. Similar for roles, a reference is made to a list. You can add more entries there, for convenience. You can add more logins here.

Be aware of the authServerType setting. This setting defines which authentication method to use. The value corresponds to the key of the configuration. In this case the authServerType references the 'stackstateAuthServer' key. In this case StackState will use the configuration from that configuration block. There is also a IdapAuthServer block present, setting authServerType to IldapAuthServer results in StackState to use the LDAP configuration instead. To configure OIDC set authServerType to oidcAuthServer and change the configuration block accordingly. For Keycloak, you need configuration block 'keycloakAuthServer'.

Security – RBAC

- Roles:
 - Subject
 - User/group
 - Scope
 - Permission
 - Actions in StackState
 - System permissions
 - View permissions



A role in StackState is a combination of a subject and a set of permissions. A subject is a group of users or a username. A subject has a scope by design. A scope is a STQL query defining what topology the subject has access to. A subject per default has no permissions set.

A permissions defines what actions a user or group can perform inside StackState. There are two types of permissions; system permissions and view permissions. System permissions are scoped towards StackState configuration and view permissions are scoped towards what a user or group can do with specific views.

The StackState CLI must be used to configure RBAC in StackState. RBAC works on all supported authentication methods. For more information about the StackState CLI's RBAC command, please have a look at the StackState CLI section.

Security – RBAC

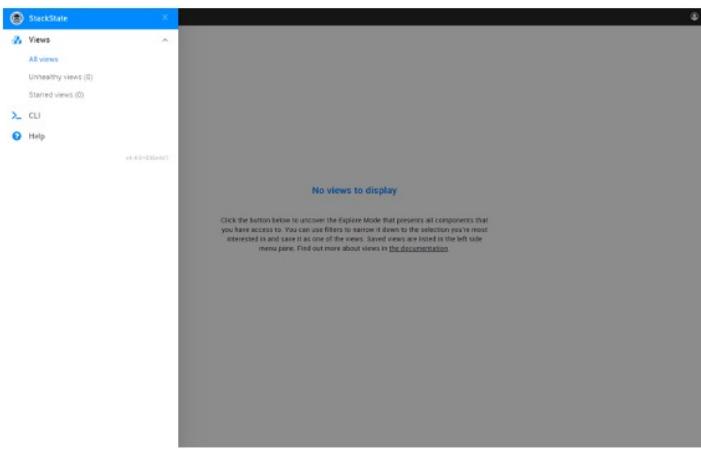
Example scenario:

- File based authentication
- Username = john
- Roles = [manager]
- Read-only access to a view called 'manager view'
- Scope limited to business layer only



Let's use an example to explain RBAC. Consider the following scenario. We are using file based authentication where we add a username for a person called John. The username is set to lower case "john". John is a manager. For managers we create a role called 'manager'. The idea is that John has access to a view called 'manager view'. There is no need to alter the view, read-only permissions on the view is enough. Additional limitation for John is that he doesn't need to view components that are lower in the dependency tree, a limit of components in the business layer is enough.

Security – RBAC



The screenshot shows the StackState web application's interface. On the left, there is a sidebar with the StackState logo and navigation links: 'Views' (which is currently selected), 'Unhealthy views (0)', 'Saved views (0)', 'CLI', and 'Help'. Below the sidebar, a message reads: 'No views to display'. At the bottom of the message, there is a note: 'Click the button below to enter the Express mode that presents all components that you have access to. You can use filters to narrow it down to the selection you're most interested in and save it as one of the views. Saved views are listed in the left sidebar menu pane. Find out more about views in [the documentation](#)'. At the bottom right of the main area, there is a small StackState logo.

If John logs in into StackState, he will see the following. Nothing much John can do here. We need to configure the subject and set permissions. Let's configure the subject first.

Security – RBAC

```
[user@ip-172-30-0-151 ~]$ sts subject show-all
Handle           Scope Query
-----
stackstate-guest
stackstate-power-user
agent-only-subject      domain = "Agent"
stackstate-platform-admin
stackstate-admin

sts subject save manager 'layer = "Business"'

[user@ip-172-30-0-151 ~]$ sts subject show-all
Handle           Scope Query
-----
stackstate-guest
stackstate-power-user
agent-only-subject      domain = "Agent"
stackstate-platform-admin
manager          layer = "Business"
stackstate-admin
```



Subjects can be managed with the ‘subject’ StackState CLI command. The ‘subject show-all’ command can be used to view all subjects present in StackState. As you can see on the top of this slide, there is no subject called ‘manager’. Let’s create the subject.

Creating a subject is done using the ‘subject save’ command. The command requires subject and scope arguments. We’ll pass the subject value ‘manager’ and scope ‘layer = “business”’ to the command.

Executing the ‘subject show-all’ again shows that the subject handle is created. The default subjects have no scope, these users are not limited in what they can see.

Now that we have a subject defined, we need to add permissions.

Security – RBAC

```
[user@ip-172-30-0-151 ~]$ sts permission show manager
subject      permission      resource
-----  -----  -----

```

```
sts permission grant manager access-view "manager view"
```

```
[user@ip-172-30-0-151 ~]$ sts permission show manager
subject      permission      resource
-----  -----  -----
manager      access-view     manager view

```



Permissions of a subject can be managed by the ‘permission show’ StackState CLI command. We pass the subject’s handle as an argument to the command. The result for the ‘manager’ subject is shown at the top of this slide. As you can see, there are no permissions defined for the ‘manager’ subject.

In our scenario we want let John access the view called “manager view”. We need to grant the access-view permission to John’s handle, to the ‘manager’ subject. This is shown in the middle here. We pass the name of the view as the last argument. The last argument specifies the resource for which you want to grant permission to. In this case, we want to grant the access-view permission only the view called “manager-view”. You could also specify “everything” here if you want to grant ‘access-view’ permissions to all views. Once a new view is added, permission is already granted.

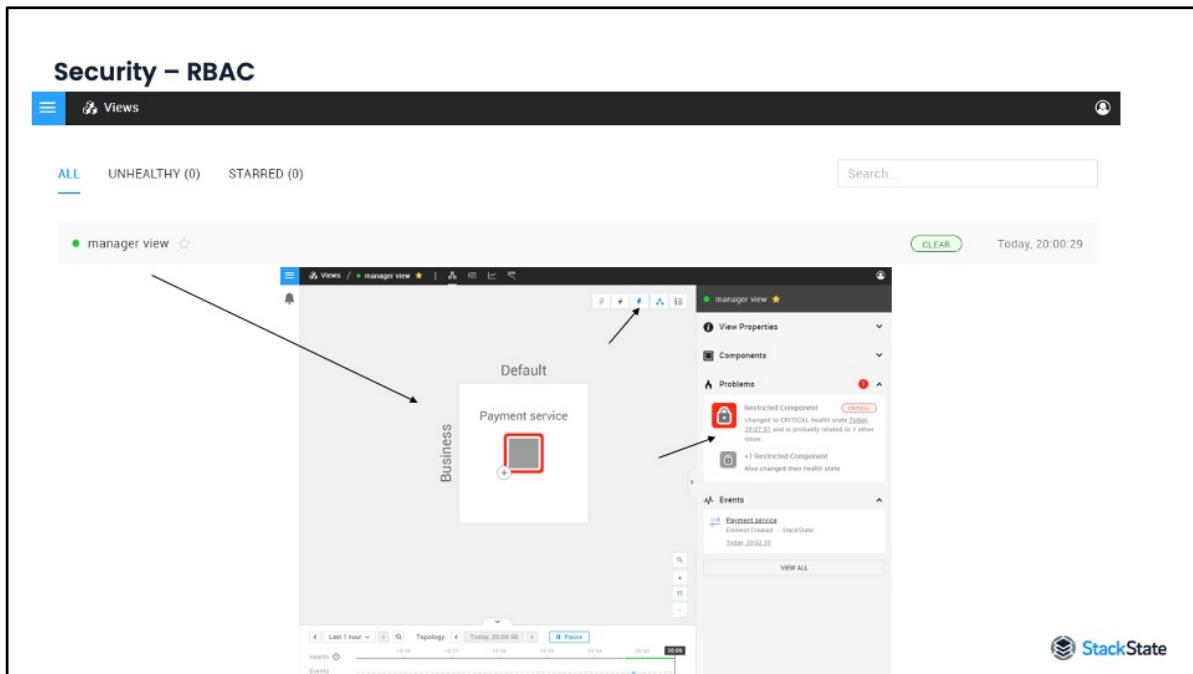
After granting the permission we can show the permissions of the subject using the ‘permission show’ command again. As we can see here, the manager subject now has the access-view permission on the

resource, a view called, 'manager view'.

Currently logged in users will be logged out when a permission is granted or revoked affecting the user's subject. Please consult the documentation website for an exhaustive list of permissions.

Ref:

https://docs.stackstate.com/configure/security/rbac/rbac_permissions



Here's what John sees after the permissions were granted. John now has the view called 'manager view' in his list of available views. Opening the view shows a single component, called "payment service".

The propagated health state of the "payment service" component is critical. The view has root cause analysis enabled. Even though the component's propagated health state is critical, the root cause is not shown. The problems pane also shows that the root cause belongs to a restricted component. This is because the subject has a scope set where components are only shown that are in the layer 'Business'.

Security – RBAC

The screenshot shows the StackState 'manager view' interface. On the left, there's a hierarchical topology diagram under 'Default' scope. It includes nodes for 'Payment service' (Business), 'Payment App' (Applications), and 'Payment Database' (Databases). The 'Payment Database' node is highlighted with a red border. To the right of the topology are several panels: 'View Properties', 'Components', 'Problems' (with a notification for a critical issue), and 'Events' (listing three events related to the Payment Database). At the bottom, there are 'Health' and 'Events' status bars.

Here's the same 'manager view', however, this time logged in as an administrator user. There is no limitation applied to the administrator about the data shown. John would have been able to see the full root cause when John's subject's scope was set to 'domain = Default', for example.

With the subject's scope, you can limit the exposure of certain components or parts of your topology. The scope's query is essentially prefixed to the view's STQL query.

Security

Agent v2 password management



Let's have a look at the StackState Agent's password management.

Security – Agent v2 password management

- StackState Agent v2
- Integration passwords

```
init_config:

instances:
  # collection interval for topology
  - min_collection_interval: 300

  # Base URL of Dynatrace instance
  # SaaS url example - https://[your-environment-id].live.dynatrace.com
  # Managed url example - https://[your-domain]/e/[your-environment-id]
  url: https://test.live.dynatrace.com

  # Api Token from Dynatrace platform which has access to read the topology API
  # Docs - https://www.dynatrace.com/support/help/dynatrace-api/basics/dynatrace-api-authentication/#generate-a-token
  # Token and Passwords can always be encrypted using Secret Management in Agent V2. Read more on docs link below
  # Docs - https://docs.stackstate.com/configure/security/secrets_management
  token: test_token
```

↓

token: ENC[dynatrace_token]



Integrations that require the StackState Agent usually require some form of authentication. Configuring a StackState Agent check comes down to editing the correct YAML file. Secrets, like username and password, need to be defined as plain text in this YAML file. This may not be desirable.

StackState Agent v2 has the option to call a user-provided executable to handle retrieval and decryption of secrets. This approach allows users to rely on any secrets management backend and select their preferred authentication method to establish initial trust with it. Using this option, plain text secrets can then be replaced in the YAML file for a notation that references a secret that the backend will provide during run time.

Here's an example of the Dynatrace integration's YAML file. As you can see, the token is stored as plain text. When a secrets backend has been configured, the token can be replaced. In this case, the value of the token is set to ENC[dynatrace_token]. This means that during run-time the value 'Dynatrace-token' is passed to the secrets backend. A password is returned by the secrets backend for use by the integration, in-memory.

Ref:

https://docs.stackstate.com/configure/security/secrets_management

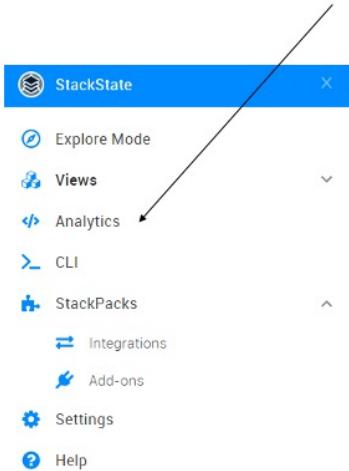
Analytics



StackState offers an analytical environment where you can write scripts and query StackState.

Analytics

- Scripting
- Query StackState's 4T data model
- StackState CLI
- Script APIs



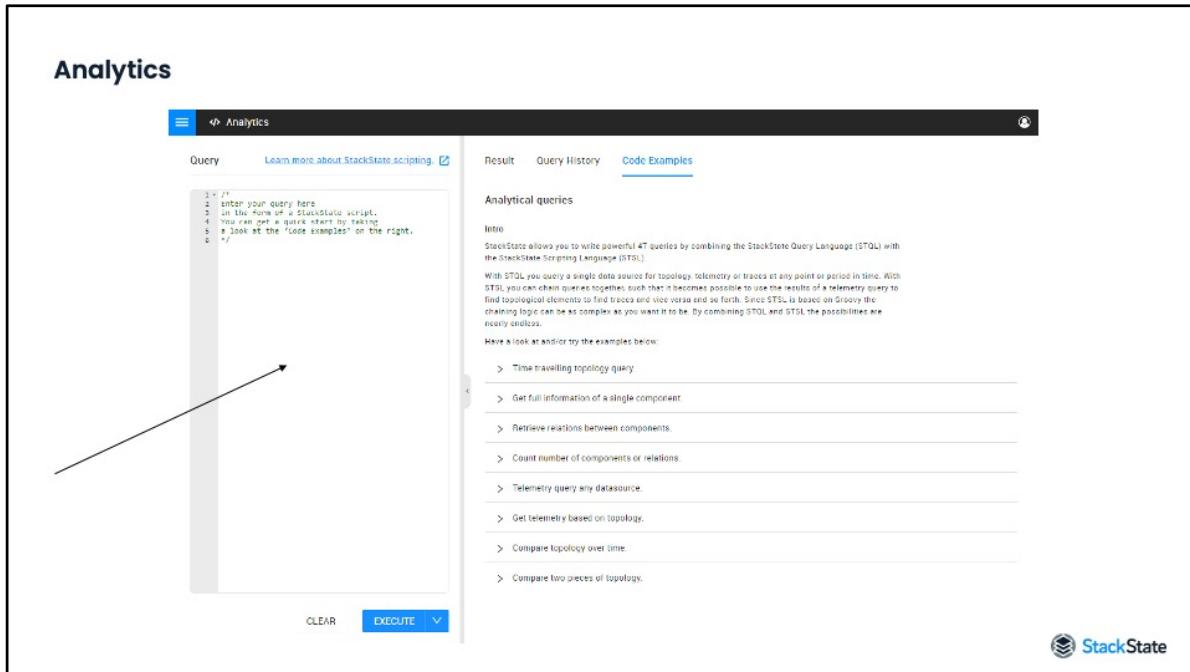
The screenshot shows the StackState user interface. At the top is a blue header bar with the StackState logo and an 'X' button. Below the header is a vertical sidebar with the following items:

- Explore Mode
- Views
- Analytics** (with an arrow pointing to it from the text above)
- CLI
- StackPacks
 - Integrations
 - Add-ons
- Settings
- Help

At the bottom right of the sidebar is the StackState logo and the text "v4.4.0+ccd3a26".

The analytics environment can be found in the main menu. The analytics environment can be used to create scripts in the Groovy programming language. There are script APIs available to query the StackState data model.

You can use the analytics environment in the StackState user interface to execute your script. Another option is to execute your scripts using the StackState CLI. This may be useful if you want to automate certain scripts.



This is the analytics environment. On the left you see the text area where you can develop your script. You can execute the script by pressing the button at the below. The result can be found on the right, under the 'Result' tab. A history of executed queries is kept under the 'Query History' tab, along with the query's result. The tab shown here is 'Code Examples'. Here you can find various examples to help you going.

The screenshot shows the Stackstate Analytics interface. On the left, there is a 'Query' section containing the following script:

```
topology.query('name = "Stackstate server"')
```

Below the query is a 'Result' section. A blue arrow points from the word 'Result' to the 'JSON' tab. The JSON output is a complex object representing a topology snapshot for a 'Stackstate server'. The object includes fields like '_type', 'scriptType', 'query', 'script', 'component', 'id', 'parent', 'children', 'relations', 'lastUpdateTimestamp', 'lastUpdateTime', 'lastUpdateTimestampMs', 'lastUpdateTimeMs', 'status', 'state', 'health', 'lastHealth', 'lastHealthTime', 'lastHealthTimeMs', and 'lastHealthStatus'. A 'Preview' tab is also visible in the Result section.

On the right side of the interface, there is a preview window titled 'Stackstate Server' which displays a small icon of a server with a stack of clouds above it.

The result of your script is by default returned in textual format. For some queries, the result can also be visualized. Where possible, the visualization can be found under the 'Preview'. A preview is available for script APIs Topology.query, Telemetry.query, Prediction.predictMetrics, and UI.showReport.

Script APIs



Let's have a look at the available script APIs.

Script APIs

- Available script APIs:
 - Async
 - Component
 - Http
 - Prediction
 - StackPack
 - Telemetry
 - Time
 - Topology
 - UI
- Available in:
 - Asynchronously executed functions
 - Component Actions
 - Analytics



In the analytical environment, you have access to the following script APIs. We'll cover each of them separately.

Certain script APIs can be used in functions, like event handler functions. These functions must be executed asynchronously, the script APIs are not available in synchronously executed function. Script APIs can also be used in component actions and analytics.

Ref: <https://docs.stackstate.com/develop/scripting/script-apis>

Script APIs

Async script API



Let's have a look at the Async script API.

Script APIs – Async

- `Async.sequence([ScriptApi.asyncFn2(), ScriptApi.asyncFn3(), ...])`

```

1  String host = "i-09ab417e69418525"
2
3
4  def topologyQuery = Topology.query("label IN ('host:$host')")
5
6  def telemetryQuery = Telemetry.query('stackState Generic Events',
7    "eventType='ec2_state' and host='$host'"
8    .aggregation("EVENT_COUNT", "1h")
9    .start("-1h")
10   .metricField("eventType")
11
12
13
14  Async.sequence([topologyQuery, telemetryQuery])
15  .then{ topologyResult, telemetryResult ->
16
17    // process results
18
19  }

```



Since the Script APIs are executed asynchronous, the responses of Script APIs are a promise to a future result. Eventually the promise is completed and an `AsyncScriptResult` is returned which holds the results of the request.

Usually, you execute a Script API and work directly with the results once these are available. There may be situations where you want to execute two, or more, script APIs and work with the results of these functions in one go. The `Async` script API offers the `sequence` function to do this. The `sequence` function accepts a list of future `AsyncScriptResults`, each returned by a script API. The `sequence` function returns a single promise to a future value. The future value will contain all the values that you expected to be returned by the different script APIs. Essentially, it flattens the results of different script API functions into one promise.

The example shown here queries topology and telemetry information for one host. Line 4 shows the topology query and line 6 until line 10 shows the telemetry query. The idea is to use data returned from both script APIs to do further processing. Maybe you want to list the host

information together with a certain metric in a script. Line 4 and 6 will hold a promise to an AsyncScriptResult, returned by both script APIs. We can sequence the return values using Async.sequence to be able to work with both return values at the same time. Line 14 shows the sequence function. We pass the variables to the function in an array. Line 15 will be executed once the values for the topology and telemetry queries are available. You can use the results in further processing.

```
String host = "i-09a1b417e69418525"

def topologyQuery = Topology.query("label IN ('host:$host')")

def telemetryQuery = Telemetry.query('StackState Generic Events',
"eventType='ec2_state' and host='$host'")
.aggregation("EVENT_COUNT", "1h").start("-1h")
.metricField("eventType")

Async.sequence([topologyQuery, telemetryQuery])
.then{ topologyResult, telemetryResult ->
// process results
}

// return [//
//      "name": topologyResult.queryResults.result.components[0].name,//      "event
//      count": telemetryResult.response.data[0][0]//      }]

```

Script APIs

Component script API



Let's have a look at the Component script API.

Script APIs – Component

- Component.withId(<id>
 - .get()
 - .checks()
 - .domain()
 - .streams()
 - .type()
 - .layer()
 - .environments()
 - .propagation()

```

1 Component.withId(100256617276128)
2   .layer()
3   // .then( it.name )
4

1 Component.withId(100256617276128)
2   .get()


```

JSON Preview

```

{
  "_type": "Layer",
  "id": 129928494886211,
  "identifier": "urn:stackpack:common:layer:machines",
  "lastupdateTimestamp": 1619182435919,
  "name": "Machines",
  "order": 13000,
  "ownedBy": "urn:stackpack:common"
}


```

JSON Preview

```

{
  "type": "ViewComponent",
  "description": "",
  "domain": 138706066156886,
  "environments": [
    198208936963099
  ],
  "failingChecks": [],
  "id": 100256617276128,
  "layer": "1-09a1b417e69418525",
  "urn:host": "/1-09a1b417e69418525",
  "urn:aws:ec2:eu-west-1:714565590525:instance/1-09a1b417e69418525",
  "incomings": [],
  "lastupdateTimestamp": 1638910993149,
  "layer": 129928494886211,
  "name": "StackState Server"
}


```

StackState

The Component script API can be used to get information about a single component.

The Component.withId function expects a component's node id as its argument. The id is available in component actions, return by the topology script API, or can be obtained by showing a component's JSON in the user interface. Some builder methods are available to use to get specific information about the component.

The get() builder returns all information about the component. The other builder methods can be used to get specific information about the component's checks, streams, domain, type, layer, environments, or propagation. Compared to the get builder method, the specific builder methods provide more information while the get method will only provide ids.

The example shown on the top-right here shows the layer builder method is used to get information about the component's layer. You can do further processing with the results, like obtaining only the name of the

layer.

The bottom-right example shows the get builder method being used. The result shows information about the component. The layer value returned here is the node id of the layer itself, as can be found in the result of the layer builder method too.

Script APIs

HTTP script API



Let's have a look at the HTTP script API.

Script APIs – Http

- `Http.get(<uri>)`
 - `timeout(<duration>)`
 - `.param(<name>, <value>)`
 - `.header(<name>, <value>)`
 - `.jsonResponse()`
 - `.jsonBody()`

```

1  Http.get("https://xkcd.com/info.0.json")
2  .jsonBody()
3  .then(it.img)
4
5
6

```

JSON Preview
["https://imgs.xkcd.com/comics/lumpers_and_splitters.png"](https://imgs.xkcd.com/comics/lumpers_and_splitters.png)

- `Http.put(<uri>)`
- `Http.post(<uri>)`
- `Http.delete(<uri>)`
- `Http.options(<uri>)`
- `Http.patch(<uri>)`
- `Http.head(<uri>)`



The Http script API can be used to send HTTP requests to external systems. There are functions available to send out get, put, post, delete, options, patch, or head HTTP requests. Each function has its own builder methods to set parameters, headers, a content type, etc. We haven't listed all builder methods for each function here, that would be too much. Text or JSON requests and responses are possible. StackState will deserialize JSON responses such that you can use the results for further processing.

The example shown here sends out a HTTP GET request to a specific URL. The response is JSON and we obtain the response by using builder method `.jsonBody()`. We could also use `jsonResponse()` if you are also interested in the return HTTP headers. Since the JSON is already deserialized we can do further processing on the response. In this case, we retrieve the 'img' tag from the response JSON payload.

Do keep in mind that StackState must be able to make a networking connection to these URLs. The StackState process on Linux distributions and the `stackstate-api` pod for Kubernetes will set up these connections.

Script APIs

Prediction script API



Let's have a look at the Prediction script API.

Script APIs – Prediction

- Prediction.predictMetrics(<predictorName>, <horizon>, <telemetry query>
 - .predictionPoints(<number>)
 - .includeHistory(<start>, <end>)

```

1 def telemetryMetricQuery = Telemetry.query("StackState Multi Metrics", "name"
2   .metricfield("connectionCount")
3   .start("-3h").aggregation("MEAN", "1m")
4   .compileQuery()
5
6
7 prediction.predictMetrics("fft", "1m", telemetryMetricQuery)
8   .predictionPoints(38)
9   .includeHistory(-1h)
10
11
12
13
14
15
16

```

Learn more about StackState scripting.

Result Query History Code Examples

JSON Preview

History Predicted

StackState

The prediction script API can be used to make predictions on metrics. The predictMetrics function requires a predictor, or algorithm, a horizon, and a telemetry query as its arguments. Depending on the use case, a predictor can be chosen. The predictor is either Fast Fourier Transformation (FFT), linear, or Harmonic Mean Normal (HMN). The horizon expects a Duration type and specifies how much future to predict. The telemetry query is used to obtain metrics for the prediction. Builder method predictionPoints specifies the number of metric points to be generated between the last seen metric and the defined horizon. Builder method includeHistory defines the number of metrics to include in the result for visualization, respecting the time bounds provided.

The example shown here defines a telemetry query on line 1. The query is not executed. Instead, the query is compiled as stated on line 4 such that it can be passed to predictMetrics as the query argument. The telemetry query is executed from the predictMetrics function to make predictions. In this case, metrics are queries over a period of the last three hours with 1 minute buckets. The query is compiled and passed to the predictMetrics function, as its third argument. The predictor used

here is Fast Fourier Transformation and the horizon to predict is 30 minutes. 30 metric points will be generated in the 30 minutes window. The function will return the last hour of metrics and the predicted points. This can be seen on the preview tab. The JSON tab will show a JSON array containing the actual metric values.

```
def telemetryMetricQuery = Telemetry.query("StackState Multi Metrics",
'name = "connection metric" and tags.externalId = "urn:endpoint:/demo-
apps.demo.stackstate.io:10.0.2.73"')
.metricField("connectionCount")
.start("-3h").aggregation("MEAN", "1m")
.compileQuery()

Prediction.predictMetrics("fft", "30m", telemetryMetricQuery)
.predictionPoints(30)
.includeHistory("-1h")
```



Let's have a look at the StackPack script API.

Script APIs – StackPack

- StackPack.isInstalled(<name>)

- StackPack.getResources(<namespace>, <node type>)


StackState

The StackPack script API provides handy operations to get the status of a StackPack or resources that are provided by a StackPack.

The `isInstalled` function returns a Boolean value indicating whether a StackPack is installed or not. The example shown here checks whether the StackState Agent StackPack is installed. We need to pass the StackPack's internal name as the argument to the `isInstalled` function.

The `getResources` function returns all configuration present in StackState in the given StackPack namespace and of a certain type. In this example, all views from the StackState Agent StackPack namespace are returned. Post-processing is applied to only show the view's name.

Script APIs

Telemetry script API



Let's have a look at the Telemetry script API.

Script APIs – Telemetry

- `Telemetry.query(<data source>, <query>)`
 - `.aggregation(<method>, <bucket size>)`
 - `.start(<time Instant>)`
 - `.end(<time Instant>)`
 - `.window(<start>, <end>)`
 - `.limit(<points>)`
 - `.metricField(<name>)`
 - `.compileQuery()`



The Telemetry script API can be used to query telemetry data sources for metrics. The arguments the query functions require are a data source name and a query. The data source name can be 'StackState Metrics' or any metric data source's name configured under telemetry data sources in the settings pages. The query is a string consisting of equality matches between keys and values, like what you see in the telemetry stream inspector. Only the 'and' construct is supported in the query if you need multiple query clauses.

The aggregation builder methods defines the aggregation method and bucket size to use. The aggregation method can be a mean, percentile, min, max, sum, or a count.

The start and end builder methods can be used to specify time bounds, to limit the number of metrics returned. Alternatively, you can use the window builder method to select a start and end time range.

A limit to the number of metric points returned can be set using the limit builder method. The limit only works on non-aggregated queries.

The metricField builder method specifies the field from where the metric is extracted. This is the same field as defined on a stream, as can be seen in the metric inspector dialog.

Optionally, the builder method compileQuery can be used to compile, but not execute, the query. The query is then compiled, additional builder methods cannot be called anymore. A compiled query can be passed to the predictMetrics function of the Prediction script API, for example.

Component named "ingress":

```
Telemetry.query("StackState Multi Metrics", 'name = "connection metric"  
and tags.externalId = "urn:endpoint:/demo-  
apps.demo.stackstate.io:10.0.2.73") .metricField("connectionCount")  
.window("-3h", "-1h") .aggregation("MEAN", "1m")
```

Script APIs

Time script API



Let's have a look at the Time script API.

Script APIs – Time

- TimeSlice:
→`Time.currentTimeSlice()`
- Instant
→`1570738241087`
→`"2019-09-18T17:34:02.666Z"`
→`"-523s"`
- Duration
→`"1w"` – weeks
→`"2d"` – days
→`"3h"` – hours
→`"4m"` – minutes
→`"5s"` – seconds



The Time script API represents certain data types that are required by other script APIs. There are data types TimeSlice, Instant, and Duration.

The TimeSlice type represents a moment in time. StackState will use this time slice to query the database. The current moment in time can be obtained by using `'Time.currentTimeSlice()'`. The Topology.query script API is an example of a function that accepts a time slice to query topology at a specific moment in time.

The Instant type represents a relative or absolute moment in time. For example, the Telemetry.query functions accept an Instant to place bounds on the number of telemetry points it needs to query. You can pass an epoch milliseconds value, an ISO 8601 formatted date/time string, or a relative time duration.

The Duration type represents a moment in time relative to the current wall clock time. StackState supports weeks, days, hours, minutes, and seconds. For example, the aggregation method in telemetry queries require a duration as bucket size.

Script APIs

Topology script API



Let's have a look at the Topology script API.

Script APIs – Topology

- Topology.query(<query>
 - .at(<time>)
 - .repeatAt(<time>)
 - .diff(<query result>)
 - .components()
 - .problems()
 - .relations()
 - ...

The screenshot shows the StackState Analytics interface. On the left, under the 'Query' tab, there is a code editor with the following STQL query:

```
String query = "name = \"Stackstate Server\"";
topology().query(query)
```

On the right, under the 'Result' tab, there is a visualization of a 'Stackstate Server' component, which is represented by a small orange cube icon inside a white box. Below the visualization, there is some descriptive text: 'Stackstate Server'.

The 'Topology' script API can be used to execute STQL queries. Further processing can be applied to the results. You can copy the STQL query from any view and execute it in your script or dynamically create your own STQL query.

In this example, a STQL query is defined that queries StackState to return all components with the name 'Stackstate Server'. The query function is used to query StackState's topology. A preview is supported for this type of query, a visualization is given. The JSON tab will provide more detailed information about the query result.

The builder method 'at' accepts an Instant or TimeSlice typed value and defines a moment in the past. The topology from the provided moment in time is queried. We'll see an example in the next slide.

The repeatAt builder method can be used to execute the same query at another moment in time. The result would then include the original query's result and the result of each added repeatAt builder method.

The diff builder method can be used to obtain a topology difference from two queries, optionally using different moments in time. We'll see an example in a minute.

Instead of returning all components, relations, and problems; specific builder methods can be used to only include the required components, relations, or problems in the result.

Script APIs – Topology

The screenshot shows the StackState interface. On the left, there is a code editor window containing Java-like pseudocode for querying topology components. On the right, there is a preview window showing four components named 'carts' arranged in a 2x2 grid. Each component is represented by a square icon with a circular arrow and a small symbol inside, such as a server or a database. The top-left component is orange, the top-right is red, the bottom-left is green, and the bottom-right is blue. The StackState logo is visible in the bottom right corner.

```
1 String stol = 'name = "carts"\n2 topology\n3     .query(stol)\n4     .at("-1h")\n5\n6\n7
```

JSON Preview

carts carts

carts carts

carts carts

StackState

Here's an example of a topology query at a specific moment in time. We are showing all components named 'carts' here that were in StackState one hour ago, as specific by passing the '-1h' Instant to the 'at' builder method.

Script APIs – Topology

Component result processing:

```

1 String stql = "name = 'carts'"
2
3 Topology
4   .query(stql)
5   .components()
6   .thenCollect{
7     return [
8       { "id": it.id,
9         "name": it.name
10      }
11    ]
12  }

```

JSON Preview

```
[{"id": 6678594672087, "name": "carts"}, {"id": 240157740662173, "name": "carts"}, {"id": 276434501809244, "name": "carts"}]
```

Component count:

```

1 String stql = "name = 'carts'"
2
3 Topology
4   .query(stql)
5   .components()
6   .count()
7

```

JSON Preview

```
3
```



Here are two other examples of the topology query function.

The example shown on top queries StackState for all components named 'carts'. Only components are requested from the query. Some post-processing is done; 'the component's id and name are returned.

The example shown on the bottom of the slide shows the number of component returned from the executed query.

String stql = "name = 'carts'"
Topology .query(stql) .components()
.thenCollect{ return [{ "id": it.id, "name": it.name }] }

Script APIs – Topology

Topology difference now versus one hour ago:

```

1 string stql = 'name = "carts"'
2
3 Topology
4   .query(stql)
5   .at("1h")
6   .diff(Topology.query(stql))

```

```

1 component
2   .withId(210664350027218)
3   .at(1632388129998)
4   .get()
5   .then {
6     it.name
7   }

```

StackState

Here's an example of the diff builder method. A STQL query is used over two moments in time to obtain the topology difference between them. The first query is set to obtain the topology as it was known an hour ago. The same query is passed to the diff builder method using the current moment in time.

The result is shown on the right. Both queries are returned next to the diff result. One query has the epoch milliseconds timestamp defined. The result section shows what was deleted, added, updated. As you can see here, in this example, a component was deleted in the last hour. If you want more information about this component, you could use the Component script API, pass the component id as the argument to the `withId` function, and set the correct moment in time.

Script APIs

UI script API



Let's have a look at the UI script API.

Script APIs – UI

- `UI.baseUrl()`
- `UI.createUrl()`
 - `.view(<name>)`
 - `.explore()`
 - `.topologyQuery(<query>)`
 - `.url()`
 - ...
- `UI.redirectToURL(<url>)`
- `UI.showReport(<name>, <STML content>, <data?>)`
- `UI.showTopologyByQuery(<query>)`

```

1 ~ UI.showReport("Report", """
2   To buy                                | Amount |
3   -----+-----+-----+
4   [apples](http://google.com/?q=apples) | 2      |
5   [pears](http://google.com/?q=pears)    | 5      |
6   """)
7
8

```

To buy	Amount
Apples	2
Pears	5

 StackState

The UI script API offers functions to control the user interface.

The `baseUrl` function returns the StackState base URL, as configured during installation. The base URL can be used to create a link to StackState, for example.

The `createUrl` function can be used to generate a link to a StackState. This link can be shared with others to offer deep linking. There are many builder methods available. You can use an existing view or explore as starting point. A STQL query can be added, a perspective can be selected, root cause analysis can be controlled, etc., etc. Once you are done constructing your view, you add the 'url' builder method which returns the actual URL as a string.

The `redirectToURL` function can be used to open the provided URL in a new browser tab. This function is commonly used in component actions to deep link into an external system.

The `showReport` function can be used to show a report. The content

can be generated. The report has a name, content, and optional data. The content is provided in StackState Markup Language, or STML for short. You can create pretty-formatted reports using StackState data. STML is a markdown-based language with extensions in the form of [HTML](#)-like tags. The data argument is a map with data elements that can be referenced by the STML. STML tags ‘auto-widget’ and ‘heatmap’ are available. You can find examples of a heatmap in the health forecast component action. At the bottom of this slide an example is shown of a simple report. The report shows a list of apples and pears including an amount for each. The report’s content is defined as Markdown. You can use this function in a component action, for example, the report would then be shown in a popup.

The `showTopologyByQuery` function opens the given STQL query in a new view. This function is commonly used by component actions to open a new view based on the component selected.



Let's have a look at the UI script API.

Script APIs – View

- View.getAll()

```
1 View.getAll()
2   .then { it.view.name }
3
4
```

JSONPreview

["cluster - Namespaces",
 "cluster - All",

 StackState

The View script API offers the getAll function to return all view definitions currently available to the current user. Post-processing is applied in this example to list all view names.

Troubleshooting



Let's talk about troubleshooting StackState.

Troubleshooting

- Log file locations
- Function logging
- No topology being synchronized
- Synchronization errors



In this section we'll cover where to find the StackState log files, how to increase the log level for individual functions, how to debug synchronization pipelines, And what to do when you have synchronization errors.

Troubleshooting

Log file locations



Let's have a look at important log file locations.

Troubleshooting – log file locations

Linux

- /opt/stackstate/var/log
 - stackstate.log
 - processmanager.log
 - stackgraph/
 - stackpack/
 - stackstate-receiver
 - kafka-to-es/
 - sync/

Kubernetes

- stackstate-api
- StackGraph - pods with prefix stackstate-hbase
- StackState receiver pod
- StackPacks - stackstate-api
- A pod for each Kafka to Elasticsearch process (e.g. stackstate-mm2es)
- Topology synchronization - stackstate-sync



For Linux distributions, the default installation directory is /opt/stackstate. The var/log directory inside /opt/stackstate contains the StackState log files.

For Kubernetes, there are pods with descriptive names and logging is outputted on standard out.

For LINUX

- The most important log file is the file stackstate.log. The majority of logs are recorded here.
In case of problems this is a good place to start.
- The processmanager.log log file contains log details from the process manager.
The systemd service starts the StackState process manager.
The process manager is responsible for starting and stopping StackState subprocesses. For example, the receiver or Elasticsearch.
If you have problems starting StackState, or the systemd service timed

out, have a look at the log file processmanager.log. The StackState process is the last process to start. If the process StackState failed to start, have a look at stackstate.log next.

- The StackGraph directory contains log files related to the StackGraph systemd service. Note that there is also a processmanager.log log file in this directory.
 - The StackPack directory contains log files about StackPacks. Each StackPack has a log file.
The name of the log file is set to the StackPack's internal name. Information about the StackPack lifecycle can be found in this file.
 - The StackState receiver logs to a file in the stackstate-receiver directory.
 - The kafka-to-es directory contains log files about each Kafka to Elasticsearch process.
These processes are responsible for getting telemetry data from Kafka to Elasticsearch. There are processes for metrics, events, and traces.
 - The sync directory contains all logs related to a topology synchronization. There is one log file per synchronization.
-

For KUBERNETES

- Logging for the stackstate-api pod is similar to the stackstate.log log file, however, some parts are split off to separate pods. In case of problems the stackstate-api pod is a good place to start.
- There are separate pods for StackGraph. These pods have a name with the prefix 'stackstate-hbase'.
- StackPack information is logged on the stackstate-api pod.
- There is a pod for the StackState receiver.
- There is a pod for each process. Kafka to Elasticsearch process.
These processes are responsible for getting telemetry data from Kafka

to Elasticsearch. There are processes for metrics, events, and traces. For example, the pod stackstate-mm2es is responsible for metrics.

- For topology synchronization, there is one stackstate-sync pod. The synchronization's name is shown in the log entries.

Troubleshooting

Function logging



Let's have a look at logging from in-use functions in StackState.

Troubleshooting – function logging

- Functions:
 - Check functions
 - Propagation functions
 - Event handler functions
 - ...
- Function scripts can contain (debug) log statements
- By default, log statements are not logged
- For debugging you can enable logging for one instance of the function



Functions in StackState consists of Groovy scripts that are executed run-time, when needed. Examples of function are; check functions, propagation functions, and event handler functions. Function scripts can contain log statements for you to use to debug certain functions in new or existing work.

By default, function logging is not enabled. Logging is done on a per instance level. This means, you can enable logging on a single instance of a function at a time. An instance, for example, is a single check on a specific component. As you can imagine, enabling logging on one function results in a fire hose of log messages if that function is instantiated a lot of times, which probably isn't meaningful anymore.

The log messages are logged to stackstate.log or to the stackstate-api pod, depending on how StackState is installed. Let's have a look at an example.

Troubleshooting – function logging

To enable function logging:

1. Add log statement in the function's code, e.g. `log.info(variable.toString())`
2. Obtain function's instance node id from StackState
3. CLI: `sts serverlog setlevel <id> <log level>`
4. Tail log:
 - `/opt/stackstate/var/log/stackstate.log`
 - Pod: `stackstate-api`



To enable logging on a function's script, we must follow the following steps.

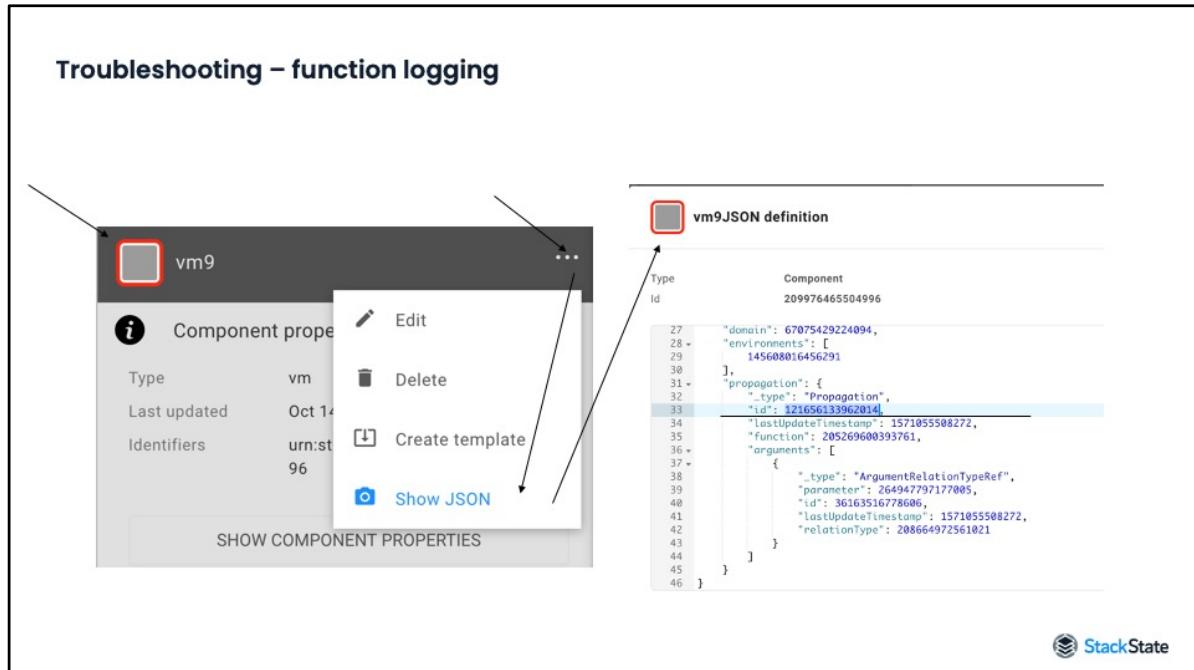
First, we need to include log statements in the script that we want to debug. The logger is available as 'log'. A function call like 'log.info' will log the message on INFO level.

To enable logging, you must find the function's instance id. Enabling logging requires a StackState internal id. This is an in-memory id and is volatile, it changes after each StackState restart for example. Obtaining the id works differently for the different classes of functions that are available. Ids can be found via the user interface and/or StackState CLI. Please consult the StackState documentation on how to find the id of each function class. We'll cover a propagation function in a minute.

The StackState CLI must be used to enable or disable function logging. The CLI's 'serverlog setlevel' command expects the id as its first argument and the desired log level as second argument.

After setting the log level for the specific id, you can now tail the log file stackstate-log or the stackstate-api pod for the log messages. Be aware that the log messages only appear once the function is invoked, reactively.

Ref: <https://docs.stackstate.com/configure/logging/enable-logging>



Here's an example on how to retrieve the id of the in-use propagation function on a single component. After opening a component's details pane, you can click on the three dots and the menu will show. Click on 'show JSON'. A dialog will open that will show the component's JSON, or database, definition.

There is a JSON key called 'propagation' present. It is shown on line 33 in this example, shown on the right. The id shown in this JSON block is the id of the propagation function instance we need. The id is shown on line 33. You can copy this id. There is also a function id, which is the id of the function itself, the node id. We can only set the log level on specific instances of the function. Check function ids can also be found in this JSON, under the 'checks' JSON block.

Troubleshooting – function logging

```
[root@ip-172-31-45-182 cli]# ./bin/sts serverlog setlevel 54877354957012 DEBUG
OK
[root@ip-172-31-45-182 cli]#
```

```
tail -f /opt/stackstate/var/log/stackstate.log | grep <id>
2019-10-14 12:31:54,187 [StackStateGlobalActorSystem-akka.actor.default-dispatcher-10] INFO com.stackstate.domain.stackelement.propagation.Propagation - [54877354957012] Number of components found: 4
2019-10-14 12:31:54,187 [StackStateGlobalActorSystem-akka.actor.default-dispatcher-10] INFO com.stackstate.domain.stackelement.propagation.Propagation - [54877354957012] component v3 has status 900
2019-10-14 12:31:54,188 [StackStateGlobalActorSystem-akka.actor.default-dispatcher-10] INFO com.stackstate.domain.stackelement.propagation.Propagation - [54877354957012] component v2 has status 100
2019-10-14 12:31:54,188 [StackStateGlobalActorSystem-akka.actor.default-dispatcher-10] INFO com.stackstate.domain.stackelement.propagation.Propagation - [54877354957012] component v4 has status 900
2019-10-14 12:31:54,188 [StackStateGlobalActorSystem-akka.actor.default-dispatcher-10] INFO com.stackstate.domain.stackelement.propagation.Propagation - [54877354957012] component v1 has status 900
2019-10-14 12:31:54,188 [StackStateGlobalActorSystem-akka.actor.default-dispatcher-10] INFO com.stackstate.domain.stackelement.propagation.Propagation - [54877354957012] Number of CRITICAL components found: 3
```



You can pass the copied id to the ‘serverlog setlevel’ CLI command together with a log level to enable function logging. In this example, we are setting the log level to DEBUG for this particular propagation function.

We open a tail of the stackstate log file, or stackstate-api pod, and wait until the function is invoked. In case of a propagation function, a state change need to happen. During debugging, you probably want to flip a health status manually to trigger an invocation. After the function is invoked, you can see the log statements appear in the log.

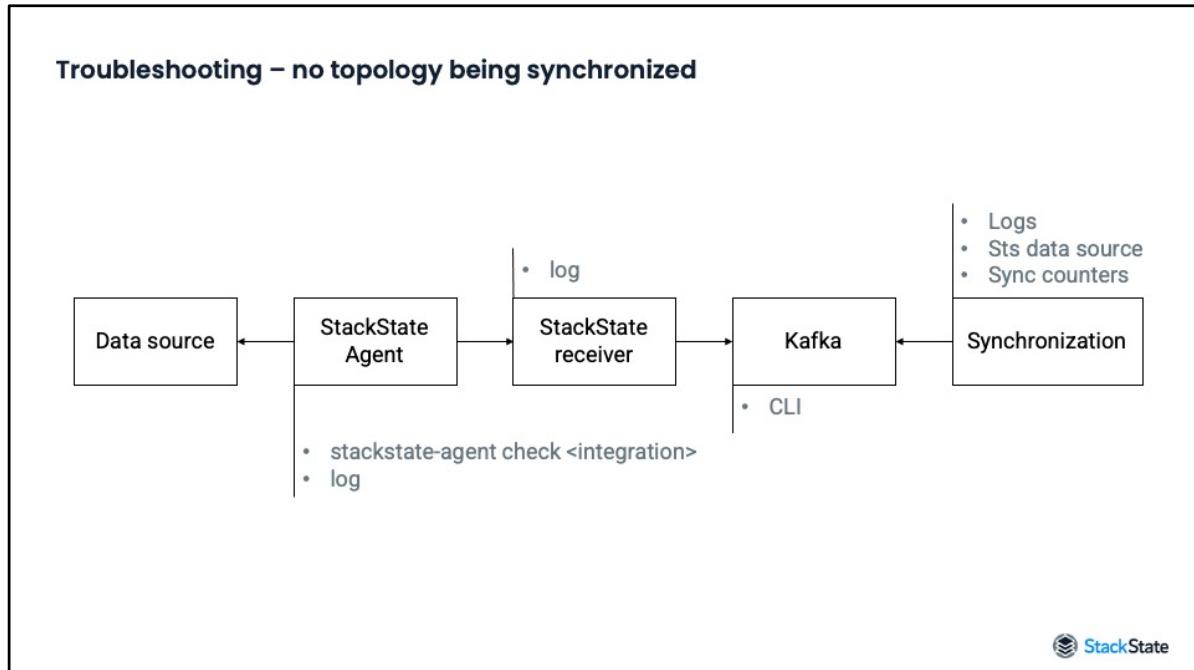
In this case we see 6 log lines appear on each invocation. The log statements were logged on INFO level. The set DEBUG level includes the INFO log level.

Troubleshooting

No topology being synchronized



Let's have a look at what you can do to debug a synchronization.



What do I need to do when no topology is synchronized? Let's walk through each step of the synchronization process by covering a typical case, an integration using the StackState Agent.

The StackState Agent is a good place to start your investigation. The StackState Agent's logs contains log messages from each integration. The logs will tell you if an integration was able to complete and errors will show up here. Also, check the StackState Agent's log for hints that it has problems connecting to StackState.

It is possible to trigger the integration via the 'stackstate-agent check' command on your terminal. This command will not send any data to StackState. Instead, it will return the topology and telemetry collected to standard output along with the generated log messages.

The StackState receiver receives data from the StackState Agent. You could check the logs for the StackState receiver. JSON deserialization errors would appear here. The StackState receiver extracts the topology and telemetry payloads from the received JSON. The StackState

Receiver puts messages on the Kafka bus.

Topology and telemetry are stored on Kafka, on separate topics. You could use the StackState CLI to list all topics present on Kafka. For topology synchronization, a topic should be present where the name has the following format; `sts_topo_<instance_type>_<instance url>`. The instance type is recognizable the name of an integration and the instance URL has to correspond to the Agent's integration YAML, usually the URL of the data source.

Most StackPacks already create this topic during installation of the StackPack. The 'waiting for data' state of a StackPack is nothing more than listening to any message on the Kafka bus before going to installed state. In this case, it is a good idea to check the messages on the Kafka topic using the StackState CLI. If there are, recent, messages on the Kafka bus then you know that the issue is not in the data collection.

StackState's topology synchronization reads data from a Kafka bus once it becomes available. The Kafka topic used is defined in the Sts data source. It is a good idea to check if the Sts data source defined topic name matches what is returned by the 'stackstate-agent check' command. Be aware that topic names are case sensitive.

Do check the synchronization counters in the synchronization settings page. Increasing numbers tell you whether data is synchronized over time. If only the error counter increases, that's more likely a processing problem. There might be an issue with a mapping or template. We'll discuss sync errors in the next section.

Troubleshooting

Synchronization errors



Let's discuss synchronization errors.

Troubleshooting – synchronization errors

Synchronizations

StackState can synchronize topology information from different sources. This includes your own sources. This way StackState enables you to create a model of your complete landscape. [Read more...](#)

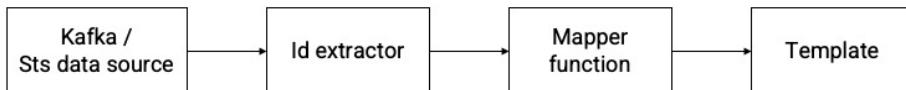
<input type="checkbox"/>	Name	Description	Created Components	Deleted Components	Created Relations	Deleted Relations	Status	Errors	...
<input type="checkbox"/>	Synchronization demo	demo	0	0	0	0	Running	2	

 StackState

The error counter of a synchronization indicates that processing received data resulted in an error. An error may occur in each step of a synchronization mapping.

It is not possible to view the errors in the user interface. Errors and warnings are logged to log files. You can retrieve by accessing logs directly. Stackstate.log and the synchronization's specific logs are the only files you need to check. In case of Kubernetes, the stackstate-api and stackstate-sync pods. Let's discover in which logs you need to access.

Troubleshooting – synchronization errors



If you have an issue in the id extractor, no topology will be synchronized. An exception is logged to stackstate.log, or on the stackstate-api pod, on each received topology element. The synchronization's error counter will not increase.

If a mapper function is defined for a synchronization mapping and there is an issue with the function, an exception is logged to the synchronization's specific log file. For Kubernetes, the exception is logged on the stackstate-sync pod where the synchronization's name is shown in the log message. The synchronization's error counter will increase.

Issues with templates are logged to the synchronization's specific log file. The synchronization's error counter will increase.

The log messages will help you in resolving the issue. The type is logged as well, helping you in determining which mapping to look at.

A note on relations. It is possible that relations reference to a source or

target component that does not exist. The components are always processed before relations, the components are simply not present in the synchronization's topology. In this case, the relation is not created and a warning is logged to the synchronization's specific log file. The component external ID and relation external ID are logged to help.