

---

# **Stork**

***Release 0.14.0***

**Dec 08, 2020**



<b>1</b>	<b>Overview</b>	<b>3</b>
1.1	Goals	3
1.2	Architecture	3
<b>2</b>	<b>Installation</b>	<b>5</b>
2.1	Supported Systems	5
2.2	Installation Prerequisites	5
2.3	Database Migration Tool (optional)	6
2.4	Installing from Packages	7
2.4.1	Installing on Debian/Ubuntu	7
2.4.2	Installing on CentOS/RHEL/Fedora	7
2.4.3	Initial Setup of the Stork Server	8
2.4.4	Initial Setup of the Stork Agent	8
2.5	Installing from Sources	9
2.5.1	Compilation Prerequisites	9
2.5.2	Download Sources	9
2.5.3	Building	9
2.6	Integration with Prometheus and Grafana	10
2.6.1	Prometheus Integration	10
2.6.2	Grafana Integration	10
<b>3</b>	<b>Using Stork</b>	<b>13</b>
3.1	Managing Users	13
3.2	Changing a User Password	14
3.3	Configuration Settings	14
3.4	Deploying Stork Agent	14
3.5	Connecting and Monitoring Machines	15
3.5.1	Registering a New Machine	15
3.5.2	Monitoring a Machine	15
3.5.3	Deleting a Machine	15
3.6	Monitoring Applications	16
3.6.1	Application Status	16
3.6.2	IPv4 and IPv6 Subnets per Kea Application	16
3.6.3	IPv4 and IPv6 Subnets in the Whole Network	17
3.6.4	IPv4 and IPv6 Networks	17
3.6.5	Host Reservations	17
3.6.6	Sources of Host Reservations	18

3.6.7	Kea High Availability Status . . . . .	18
3.6.8	Viewing Kea Log . . . . .	19
3.7	Dashboard . . . . .	20
3.7.1	DHCP Panel . . . . .	20
3.7.2	Events Panel . . . . .	20
3.8	Events Page . . . . .	21
<b>4</b>	<b>Backend API</b>	<b>23</b>
<b>5</b>	<b>Developer's Guide</b>	<b>25</b>
5.1	Rakefile . . . . .	25
5.2	Generating Documentation . . . . .	25
5.3	Setting Up the Development Environment . . . . .	25
5.3.1	Installing Git Hooks . . . . .	26
5.4	Agent API . . . . .	26
5.5	ReST API . . . . .	27
5.6	Backend Unit Tests . . . . .	27
5.6.1	Unit Tests Database . . . . .	27
5.6.2	Unit Tests Coverage . . . . .	28
5.7	Backend Benchmarks . . . . .	28
5.8	WebUI Unit Tests . . . . .	28
5.9	System Tests . . . . .	29
5.9.1	Dependencies . . . . .	29
5.9.2	LXD Installation . . . . .	29
5.9.3	Running System Tests . . . . .	30
5.9.4	Developing System Tests . . . . .	31
5.10	Docker Containers for Development . . . . .	33
5.11	Packaging . . . . .	34
<b>6</b>	<b>Demo</b>	<b>35</b>
6.1	Requirements . . . . .	35
6.2	Setup Steps . . . . .	36
6.2.1	Premium Features . . . . .	36
6.3	Demo Containers . . . . .	36
6.4	Initialization . . . . .	37
6.5	Stork Environment Simulator . . . . .	37
6.6	Prometheus . . . . .	38
6.7	Grafana . . . . .	38
<b>7</b>	<b>Manual Pages</b>	<b>39</b>
7.1	stork-server - The central Stork server . . . . .	39
7.1.1	Synopsis . . . . .	39
7.1.2	Description . . . . .	39
7.1.3	Arguments . . . . .	39
7.1.4	Mailing Lists and Support . . . . .	40
7.1.5	History . . . . .	40
7.1.6	See Also . . . . .	40
7.2	stork-agent - Stork agent that monitors BIND 9 and Kea services . . . . .	40
7.2.1	Synopsis . . . . .	40
7.2.2	Description . . . . .	41
7.2.3	Arguments . . . . .	41
7.2.4	Mailing Lists and Support . . . . .	41
7.2.5	History . . . . .	42
7.2.6	See Also . . . . .	42
7.3	stork-db-migrate - The Stork database migration tool . . . . .	42

7.3.1	Synopsis . . . . .	42
7.3.2	Description . . . . .	42
7.3.3	Arguments . . . . .	42
7.3.4	Mailing Lists and Support . . . . .	43
7.3.5	History . . . . .	43
7.3.6	See Also . . . . .	43

**8 Indices and tables** **45**



Stork is a new project led by ISC with the aim of delivering an *ISC BIND 9* and *ISC Kea DHCP* use and monitoring dashboard. It is intended to be a spiritual successor of the earlier attempts *Kittiwake* and *Anterius*.



This is the reference guide for Stork version 0.14.0. Links to the most up-to-date version of this document, along with other documents for Stork, can be found on ISC's [Stork project homepage](#) or at [readthedocs](#).





### 1.1 Goals

The goals of the Stork project are as follows:

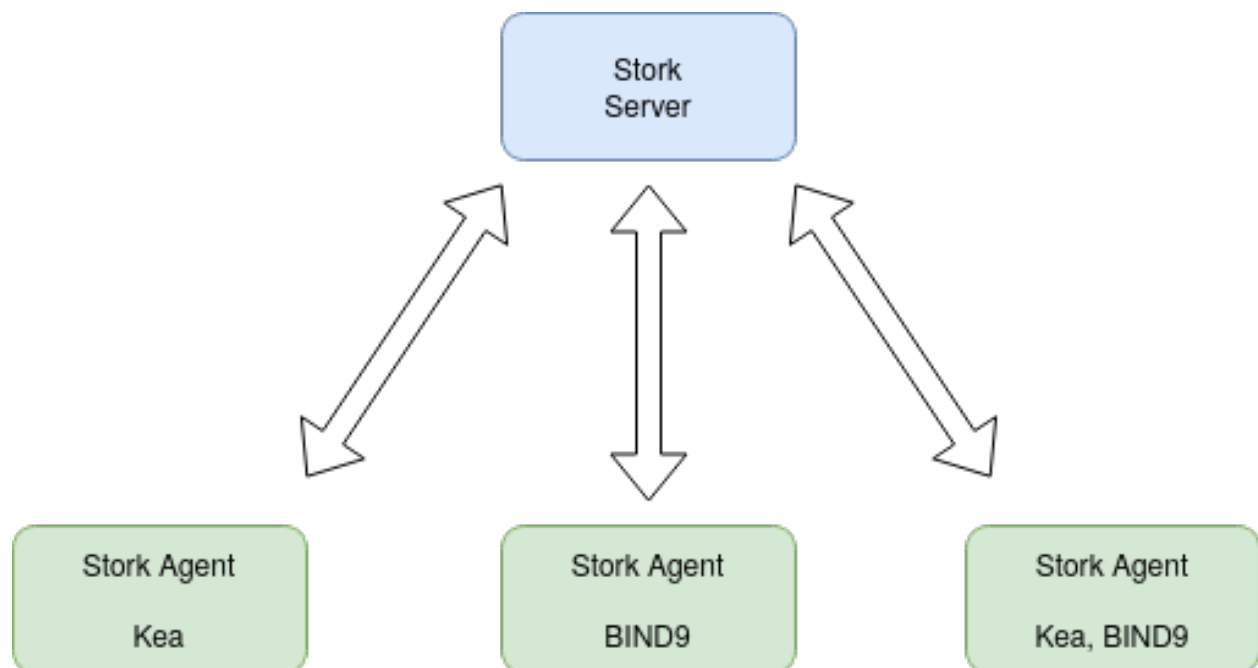
- to provide monitoring and insight into *ISC Kea DHCP* and *ISC BIND 9* operations
- to provide alerting mechanisms that indicate failures, fault conditions, and other unwanted events in *ISC Kea DHCP* and *ISC BIND 9* services
- to permit easier troubleshooting of these services

### 1.2 Architecture

Stork is comprised of two components: `Stork Server` and `Stork Agent`.

`Stork Agent` is installed along with *Kea DHCP* or *BIND 9* and interacts directly with those services. There may be many agents deployed in a network, one per machine.

`Stork Server` is installed on a stand-alone machine. It connects to any indicated agents and indirectly (via those agents) interacts with the *Kea DHCP* and *BIND 9* services. It provides an integrated, centralized front end for interacting with these services. Only one `Stork Server` is deployed in a network.



Stork can be installed from pre-built packages or from sources. The following sections describe both methods. Unless there's a good reason to compile from sources, installing from native DEB or RPM packages is easier and faster.

## 2.1 Supported Systems

Currently Stork is being tested on the following systems:

- Ubuntu 18.04 and 20.04
- Fedora 31 and 32
- CentOS 7
- MacOS 10.15\*

Note that MacOS is not and will not be officially supported. Many developers in our team use Macs, so we're trying to keep Stork buildable on this platform.

Stork server and agents are written in Go language. The server uses PostgreSQL database. In principle, the software could be run on any POSIX system that has Go compiler and PostgreSQL. It is likely the software can be built on many other modern systems, but for the time being our testing capabilities are modest. If your favourite OS is not on this list, please do try running Stork and report your findings.

## 2.2 Installation Prerequisites

The `Stork Agent` does not require any specific dependencies to run. It can be run immediately after installation.

Stork uses the `status-get` command to communicate with Kea, and therefore will only work with a version of Kea that supports `status-get`, which was introduced in Kea 1.7.3 and backported to 1.6.3.

Stork requires the premium `Host Commands (host_cmds)` hook library to be loaded by the Kea instance to retrieve host reservations stored in an external database. Stork does work without the Host Commands hook library,

but is not able to display host reservations. Stork can retrieve host reservations stored locally in the Kea configuration without any additional hook libraries.

Stork requires the open source Stat Commands (`stat_cmds`) hook library to be loaded by the Kea instance to retrieve lease statistics. Stork does work without the Stat Commands hook library, but will not be able to show pool utilization and other statistics.

For the Stork Server, a PostgreSQL database (<https://www.postgresql.org/>) version 11 or later is required. It may work with PostgreSQL 10, but this was not tested. The general installation procedure for PostgreSQL is OS-specific and is not included here. However, please keep in mind that Stork uses `pgcrypto` extensions, which are often come in a separate package. For example, you need `postgresql-crypto` package on Fedora and `postgresql12-contrib` on RHEL and CentOS.

These instructions prepare a database for use with the Stork Server, with the `stork` database user and `stork` password. Next, a database called `stork` is created and the `pgcrypto` extension is enabled in the database.

First, connect to PostgreSQL using `psql` and the `postgres` administration user. Depending on your system configuration, this may require switching to `postgres` user, using `su postgres` command first.

```
$ psql postgres
psql (11.5)
Type "help" for help.
postgres=#
```

Then, prepare the database:

```
postgres=# CREATE USER stork WITH PASSWORD 'stork';
CREATE ROLE
postgres=# CREATE DATABASE stork;
CREATE DATABASE
postgres=# GRANT ALL PRIVILEGES ON DATABASE stork TO stork;
GRANT
postgres=# \c stork
You are now connected to database "stork" as user "thomson".
stork=# create extension pgcrypto;
CREATE EXTENSION
```

---

**Note:** Make sure the actual password is stronger than ‘stork’ which is trivial to guess. Using default passwords is a security risk. Stork puts no restrictions on the characters used in the database passwords nor on their length. In particular, it accepts passwords containing spaces, quotes, double quotes and other special characters.

---

## 2.3 Database Migration Tool (optional)

Optional step: to initialize the database directly, the migrations tool must be built and used to initialize and upgrade the database to the latest schema. However, this is completely optional, as the database migration is triggered automatically upon server startup. This is only useful if for some reason it is desirable to set up the database but not yet run the server. In most cases this step can be skipped.

```
$ rake build_migrations
$ backend/cmd/stork-db-migrate/stork-db-migrate init
$ backend/cmd/stork-db-migrate/stork-db-migrate up
```

The up and down command has an optional `-t` parameter that specifies desired schema version. This is only useful when debugging database migrations.

```
$ # migrate up version 25
$ backend/cmd/stork-db-migrate/stork-db-migrate up -t 25
$ # migrate down back to version 17
$ backend/cmd/stork-db-migrate/stork-db-migrate down -t 17
```

Note the server requires the latest database version to run, will always run the migration on its own and will refuse to start if migration fails for whatever reason. The migration tool is mostly useful for debugging problems with migration or migrating the database without actually running the service. For complete reference, see manual page here: *stork-db-migrate - The Stork database migration tool*.

To debug migrations, another useful feature is SQL tracing using the `-db-trace-queries` parameter. It takes either “all” (trace all SQL operations, including migrations and run-time) or “run” (just run-time operations, skip migrations). If specified without parameter, “all” is assumed. With it enabled, *stork-db-migrate* will print out all its SQL queries on stderr. For example, you can use these commands to generate an SQL script that will update your schema. Note that for some migrations, the steps are dependent on the contents of your database, so this will not be an universal Stork schema. This parameter is also supported by the Stork server.

```
$ backend/cmd/stork-db-migrate/stork-db-migrate down -t 0
$ backend/cmd/stork-db-migrate/stork-db-migrate up --db-trace-queries 2> stork-schema.
↪txt
```

## 2.4 Installing from Packages

Stork packages are stored in repositories located on the Cloudsmith service: <https://cloudsmith.io/~isc/repos/stork/packages/>. Both Debian/Ubuntu and RPM packages may be found there.

Detailed instructions for setting up the operating system to use this repository are available under the *Set Me Up* button on the Cloudsmith repository page.

### 2.4.1 Installing on Debian/Ubuntu

The first step for both Debian and Ubuntu is:

```
$ curl -sLf 'https://dl.cloudsmith.io/public/isc/stork/cfg/setup/bash.deb.sh' | sudo_
↪bash
```

Next, install the package with Stork Server:

```
$ sudo apt install isc-stork-server
```

Then, install Stork Agent:

```
$ sudo apt install isc-stork-agent
```

It is possible to install both agent and server on the same machine.

### 2.4.2 Installing on CentOS/RHEL/Fedora

The first step for RPM-based distributions is:

```
$ curl -sLf 'https://dl.cloudsmith.io/public/isc/stork/cfg/setup/bash.rpm.sh' | sudo_
↪bash
```

Next, install the package with `Stork Server`:

```
$ sudo dnf install isc-stork-server
```

Then, install `Stork Agent`:

```
$ sudo dnf install isc-stork-agent
```

It is possible to install both agent and server on the same machine. If `dnf` is not available, `yum` can be used in similar fashion.

### 2.4.3 Initial Setup of the Stork Server

These steps are the same for both Debian-based and RPM-based distributions that use *SystemD*.

After installing `Stork Server` from the package, the basic settings must be configured. They are stored in `/etc/stork/server.env`.

These are the required settings to connect with the database:

- `STORK_DATABASE_HOST` - the address of a PostgreSQL database; default is *localhost*
- `STORK_DATABASE_PORT` - the port of a PostgreSQL database; default is *5432*
- `STORK_DATABASE_NAME` - the name of a database; default is *stork*
- `STORK_DATABASE_USER_NAME` - the username for connecting to the database; default is *stork*
- `STORK_DATABASE_PASSWORD` - the password for the username connecting to the database

With those settings in place, the `Stork Server` service can be enabled and started:

```
$ sudo systemctl enable isc-stork-server
$ sudo systemctl start isc-stork-server
```

To check the status:

```
$ sudo systemctl status isc-stork-server
```

By default, the `Stork Server` web service is exposed on port 8080, so it can be visited in a web browser at <http://localhost:8080>.

It is possible to put `Stork Server` behind an HTTP reverse proxy using *Nginx* or *Apache*. In the `Stork Server` package an example configuration file is provided for *Nginx*, in `/usr/share/stork/examples/nginx-stork.conf`.

### 2.4.4 Initial Setup of the Stork Agent

These steps are the same for both Debian-based and RPM-based distributions that use *SystemD*.

After installing `Stork Agent` from the package, the basic settings must be configured. They are stored in `/etc/stork/agent.env`.

These are the required settings to connect with the database:

- `STORK_AGENT_ADDRESS` - the IP address of the network interface which `Stork Agent` should use for listening for `Stork Server` incoming connections; default is *0.0.0.0* (i.e. listen on all interfaces)
- `STORK_AGENT_PORT` - the port that should be used for listening; default is *8080*

With those settings in place, the `Stork Agent` service can be enabled and started:

```
$ sudo systemctl enable isc-stork-agent
$ sudo systemctl start isc-stork-agent
```

To check the status:

```
$ sudo systemctl status isc-stork-agent
```

After starting, the agent periodically tries to detect installed Kea DHCP or BIND 9 services on the system. If it finds them, they are reported to the `Stork Server` when it connects to the agent.

Further configuration and usage of the `Stork Server` and the `Stork Agent` are described in the *Using Stork* chapter.

## 2.5 Installing from Sources

### 2.5.1 Compilation Prerequisites

Usually it's more convenient to install Stork using native packages. See *Supported Systems* and *Installing from Packages* for details regarding supported systems. However, you can build the sources on your own.

The dependencies needed to be installed to build `Stork` sources are:

- Rake
- Java Runtime Environment (only if building natively, not using Docker)
- Docker (only if running in containers, this is needed to build the demo)

Other dependencies are installed automatically in a local directory by Rake tasks. This does not require root privileges. If you intend to run the demo environment, you need Docker and don't need Java (Docker will install Java within a container).

For details about the environment, please see the Stork wiki at <https://gitlab.isc.org/isc-projects/stork/-/wikis/Install>.

### 2.5.2 Download Sources

The `Stork` sources are available on the ISC GitLab instance: <https://gitlab.isc.org/isc-projects/stork>.

To get the latest sources invoke:

```
$ git clone https://gitlab.isc.org/isc-projects/stork
```

### 2.5.3 Building

There are several components of `Stork`:

- `Stork Agent` - this is the binary *stork-agent*, written in Go
- `Stork Server` - this is comprised of two parts: - *backend service* - written in Go - *frontend* - an *Angular* application written in Typescript

All components can be built using the following command:

```
$ rake build_all
```

The agent component is installed using this command:

```
$ rake install_agent
```

and the server component with this command:

```
$ rake install_server
```

By default, all components are installed to the *root* folder in the current directory; however, this is not useful for installation in a production environment. It can be customized via the `DESTDIR` variable, e.g.:

```
$ sudo rake install_server DESTDIR=/usr
```

## 2.6 Integration with Prometheus and Grafana

Stork can optionally be integrated with [Prometheus](#), an open source monitoring and alerting toolkit and [Grafana](#), an easy-to-view analytics platform for querying, visualization and alerting. Grafana requires external data storage. Prometheus is currently the only environment supported by both Stork and Grafana. It is possible to use Prometheus only without Grafana, but using Grafana requires Prometheus.

### 2.6.1 Prometheus Integration

Stork agent by default makes the BIND 9 and Kea statistics available in a format understandable by Prometheus (works as a Prometheus exporter, in Prometheus nomenclature). If Prometheus server is available, it can be configured to monitor Stork Agents. To enable Stork Agent monitoring, you need to edit `prometheus.yml` (typically stored in `/etc/prometheus/`, but this may vary depending on your installation) and add the following entries there:

```
# statistics from Kea
- job_name: 'kea'
  static_configs:
    - targets: ['agent-kea.example.org:9547', 'agent-kea6.example.org:9547', ... ]

# statistics from bind9
- job_name: 'bind9'
  static_configs:
    - targets: ['agent-bind9.example.org:9119', 'another-bind9.example.org:9119', ... ]
↵ ]
```

By default, Stork agent exports BIND 9 data on TCP port 9119 and Kea data on TCP port 9547. This can be configured using command line parameters (or the Prometheus export can be disabled altogether). For details, see the `stork-agent` manual page.

After restarting, the Prometheus web interface can be used to inspect whether statistics are exported properly. BIND 9 statistics use `bind_` prefix (e.g. `bind_incoming_queries_tcp`), while Kea statistics use `kea_` prefix (e.g. `kea_dhcp4_addresses_assigned_total`).

### 2.6.2 Grafana Integration

Stork provides several Grafana templates that can easily be imported. Those are available in the `grafana/` directory of the Stork source codes. Currently the available templates are `bind9-resolver.json` and `kea-dhcp4.json`. More are expected in the future. Grafana integration requires three steps.

1. Prometheus has to be added as a data source. This can be done in several ways, including UI interface and editing Grafana configuration files. For details, see Grafana documentation about Prometheus integration; here we simply



indicate the easiest method. Using the Grafana UI interface, select Configuration, select Data Sources, click “Add data source”, and choose Prometheus, then specify necessary parameters to connect to your Prometheus instance. In test environments, the only really necessary parameter is URL, but most production deployments also want authentication.

2. Import existing dashboard. In the Grafana UI click Dashboards, then Manage, then Import and select one of the templates, e.g. *kea-dhcp4.json*. Make sure to select your Prometheus data source that you added in the previous step. Once imported, the dashboard can be tweaked as needed.

3. Once Grafana is configured, go to Stork UI interface, log in as super-admin, click Settings in the Configuration menu and then fill URLs to Grafana and Prometheus that point to your installations. Once this is done, Stork will be able to show links for subnets leading to specific subnets. More integrations like this are expected in the future.

Alternatively, a Prometheus data source can be added by editing *datasource.yaml* (typically stored in */etc/grafana*, but this may vary depending on your installation) and adding entries similar to this one:

```
datasources:  
- name: Stork-Prometheus instance  
  type: prometheus  
  access: proxy  
  url: http://prometheus.example.org:9090  
  isDefault: true  
  editable: false
```

Also, the Grafana dashboard files can be copied to */var/lib/grafana/dashboards/* (again, this may vary depending on your installation).

Example dashboards with some live data can be seen in the [Stork screenshots gallery](#) .



This section describes how to use the features available in `Stork`. To connect to `Stork`, use a web browser and connect to port 8080. If `Stork` is running on a localhost, it can be reached by navigating to <http://localhost:8080>.

### 3.1 Managing Users

A default administrator account is created upon initial installation of `Stork`. It can be used to sign in to the system via the web UI, using the username `admin` and password `admin`.

To manage users, click on the `Configuration` menu and choose `Users` to see a list of existing users. There will be at least one user, `admin`.

To add a new user, click `Create User Account`. A new tab opens to specify the new account parameters. Some fields have specific restrictions:

- Username can consist of only letters, numbers, and an underscore (`_`).
- The e-mail field is optional, but if specified, it must be a well-formed e-mail.
- The `firstname` and `lastname` fields are mandatory.
- The password must only contain letters, digits, `@`, `,`, `!`, `+`, or `-`, and must be at least eight characters long.

Currently, users are associated with one of the two predefined groups (roles), i.e. `super-admin` or `admin`, which must be selected when the user account is created. Users belonging to the `super-admin` group are granted full privileges in the system, including creation and management of user accounts. The `admin` group has similar privileges, except that the users in this group are not allowed to manage other users' accounts.

Once the new user account information has been specified and all requirements are met, the `Save` button becomes active and the new account can be enabled.

## 3.2 Changing a User Password

An initial password is assigned by the administrator when a user account is created. Each user should change the password when first logging into the system. To change the password, click on the `Profile` menu and choose `Settings` to display the user profile information. Click on `Change password` in the menu bar on the left and specify the current password in the first input box. The new password must be specified and confirmed in the second and third input boxes, and must meet the password requirements specified in the previous section. When all entered data is valid, the `Save` button is activated for changing the password.

## 3.3 Configuration Settings

It is possible to control some of the Stork configuration settings from the web UI. Click on the `Configuration` menu and choose `Settings`. There are two classes of settings available: `Intervals` and `Grafana & Prometheus`.

`Intervals` settings specify configuration of “pullers”. A puller is a mechanism in Stork which triggers specific action according to the specified interval. Each puller has its own specific action and interval. Puller interval is specified in seconds and designates a time period between the completion of the previously invoked action and the beginning of the next invocation of this action.

For example, if the `Kea Hosts Puller Interval` is set to 10 seconds and it takes 5 seconds to pull the hosts information, the time period between the beginnings of the two consecutive attempts to pull the hosts information will be equal to 15 seconds. The pull time varies between deployments and depends on the amount of information pulled, network congestion and other factors. The interval setting guarantees that there is a constant idle time between any consecutive attempts.

The `Grafana & Prometheus` settings currently allow for specifying URLs of the `Prometheus` and `Grafana` instances used with Stork.

## 3.4 Deploying Stork Agent

The Stork system uses agents to monitor services. `Stork Agent` is a daemon that must be deployed and run on each machine to be monitored. Currently, there are no automated deployment routines and `Stork Agent` must be installed manually. This can be done in one of two ways: from `RPM` or `deb` packages (described in the [Installation](#) chapter), or by simply copying the `Stork Agent` binary to the destination machine manually. The packages are usually far more convenient.

Assuming you choose to not use the packages, the `Stork Agent` binary can be copied manually. Assuming services will be monitored on a machine with the IP `192.0.2.1`, enter the following on the Stork server command line:

```
$ cd <stork-dir>
$ scp backend/cmd/stork-agent login@192.0.2.1:/path
```

On the machine to be monitored, start the agent by running:

```
$ ./stork-agent
```

It is possible to set the `--host=` or `STORK_AGENT_ADDRESS` environment variables to specify which address the agent listens on. The `--port` or `STORK_AGENT_PORT` environment variables specify which TCP port the agent listens on.

Normally, the agent will create a TCP socket on which to listen for commands from a `stork-server` and create exporters which export data to `Prometheus`. There are two command line flags which may be used to alter this behavior. The `--listen-stork-only` flag instructs the agent to listen for commands from the Stork Server but

---

not for Prometheus requests. Conversely, the `--listen-prometheus-only` flag instructs the agent to listen for Prometheus requests but not for commands from the Stork Server.

---

**Note:** Unless explicitly specified, the agent listens on all addresses on port 8080. There are no authentication mechanisms implemented in the agent yet. Use with care!

---

## 3.5 Connecting and Monitoring Machines

### 3.5.1 Registering a New Machine

Once the agent is deployed and running on the machine to be monitored, the `Stork Server` must be instructed to start monitoring it. This can be done via the `Services` menu, under `Machines`, to see a list of currently registered machines.

To add a new machine, click `Add New Machine` and specify the machine address (IP address, hostname, or FQDN) and a port.

After the `Add` button is clicked, the server attempts to establish a connection to the agent. Make sure that any active firewalls will allow incoming connections to the TCP port specified.

Once a machine is added, a number of parameters are displayed, including hostname, address, agent version, number of CPU cores, CPU load, available total memory, current memory utilization, uptime, OS, platform family, platform name, OS version, kernel, virtualization details (if any), and host ID.

If any applications, i.e. *Kea DHCP* and/or *BIND 9*, are detected on this machine, the status of those applications is displayed and the link allows navigation to the application details.

Navigation to the discovered applications is also possible through the `Services` menu.

### 3.5.2 Monitoring a Machine

Monitoring of registered machines is accomplished via the `Services` menu, under `Machines`. A list of currently registered machines is displayed, with multiple pages available if needed.

A filtering mechanism that acts as an omnibox is available. Via a typed string, Stork can search for an address, agent version, hostname, OS, platform, OS version, kernel version, kernel architecture, virtualization system, or host-id fields.

The state of a machine can be inspected by clicking its hostname; a new tab opens with the machine's details. Multiple tabs can be open at the same time, and clicking `Refresh` updates the available information.

The machine state can also be refreshed via the `Action` menu. On the `Machines` list, each machine has its own menu; click on the triple-lines button at the right side and choose the `Refresh` option.

### 3.5.3 Deleting a Machine

To stop monitoring a machine, go to the `Machines` list, find the machine to stop monitoring, click on the triple-lines button at the right side, and choose `Delete`. This will terminate the connection between the Stork server and the agent running on the machine, and the server will no longer monitor it. However, the Stork agent process will continue running on the machine. Complete shutdown of a Stork agent process must be done manually, e.g. by connecting to the machine using `ssh` and stopping the agent there. One way to achieve that is to issue the `killall stork-agent` command.

## 3.6 Monitoring Applications

### 3.6.1 Application Status

Kea DHCP and BIND 9 applications discovered on connected machines are listed via the top-level menu bar, under *Services*. Both the Kea and BIND 9 applications can be selected; the list view includes the application version, application status, and some machine details. The *Action* button is also available, to refresh the information about the application.

The application status displays a list of daemons belonging to the application. For BIND 9, it is always only one daemon, `named`. In the case of Kea, several daemons may be presented in the application status column, typically: DHCPv4, DHCPv6, DDNS, and CA (Kea Control Agent).

For BIND 9, the Stork Agent is looking for the `named` in the process list and parses the configuration file that is given with `-c` argument. If the `named` process is started without a specific configuration file, the Stork Agent will default to `/etc/bind/named.conf`.

Stork uses `rndc` to retrieve the application status. It looks for the `controls` statement in the configuration file, and uses the first listed control point for monitoring the application.

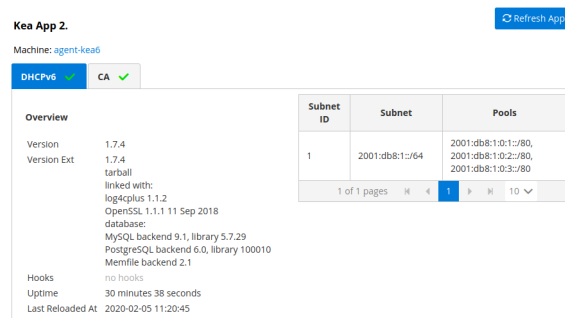
Furthermore, the Stork Agent can be used as a Prometheus exporter. Stork is able to do so if `named` is built with `json-c` because it will gather statistics via the JSON statistics API. The `named.conf` file must have a `statistics-channel` configured and the exporter will query the first listed channel. Stork is able to export the most metrics if `zone-statistics` is set to `full` in the `named.conf` configuration.

For Kea, the listed daemons are those that Stork finds in the CA configuration file. A warning sign is displayed for any daemons from the CA configuration file that are not running. In cases when the Kea installation is simply using the default CA configuration file, which includes configuration of daemons that are never intended to be launched, it is recommended to remove (or comment out) those configurations to eliminate unwanted warnings from Stork about inactive daemons.

### 3.6.2 IPv4 and IPv6 Subnets per Kea Application

One of the primary configuration aspects of any network is the layout of IP addressing. This is represented in Kea with IPv4 and IPv6 subnets. Each subnet represents addresses used on a physical link. Typically, certain parts of each subnet (“pools”) are delegated to the DHCP server to manage. Stork is able to display this information.

One way to inspect the subnets and pools within Kea is by looking at each Kea application to get an overview of what configurations a specific Kea application is serving. A list of configured subnets on that specific Kea application is displayed. The following picture shows a simple view of the Kea DHCPv6 server running with a single subnet, with three pools configured in it.



The screenshot shows the Stork interface for 'Kea App 2' on machine 'agent-kead'. The 'DHCPv6' application is selected, and the 'CA' status is indicated as active with a green checkmark. The overview section lists various details including version (1.7.4), version extension (1.7.4), tarball, linked with (log4cplus 1.1.2), OpenSSL (1.1.1 11 Sep 2018), database (MySQL backend 9.1, library 5.7.29; PostgreSQL backend 6.0, library 100010; Memfile backend 2.1), hooks (no hooks), uptime (30 minutes 38 seconds), and last reloaded at (2020-02-05 11:20:45). A table displays the configured subnets and pools:

Subnet ID	Subnet	Pools
1	2001:db8:1::/64	2001:db8:1:0:1::/80, 2001:db8:1:0:2::/80, 2001:db8:1:0:3::/80

The table includes pagination controls showing '1 of 1 pages' and navigation arrows.

### 3.6.3 IPv4 and IPv6 Subnets in the Whole Network

It is convenient to see the complete overview of all subnets configured in the network being monitored by Stork. Once at least one machine with the Kea application running is added to Stork, click on the DHCP menu and choose Subnets to see all available subnets. The view shows all IPv4 and IPv6 subnets with the address pools and links to the applications that are providing them. An example view of all subnets in the network is presented in the figure below.

**DHCP Subnets**

Q Filter subnets:  Protocol:

Subnet ID	Subnet	Pools	App ID
1	192.0.2.0/24	192.0.2.1-192.0.2.50   192.0.2.51-192.0.2.100 192.0.2.101-192.0.2.150   192.0.2.151-192.0.2.200	3
1	192.0.3.0/24	192.0.3.1-192.0.3.200	4
1	192.0.3.0/24	192.0.3.1-192.0.3.200	5
1	2001:db8:1::/64	2001:db8:1:0:1::/80   2001:db8:1:0:2::/80   2001:db8:1:0:3::/80	2

1 of 1 pages

There are filtering capabilities available in Stork; it is possible to choose whether to see IPv4 only, IPv6 only, or both. There is also an omniseach box available where users can type a search string. Note that for strings of four characters or more, the filtering takes place automatically, while shorter strings require the user to hit Enter. For example, in the above situation it is possible to show only the first (192.0.2.0/24) subnet by searching for the 0.2 string. One can also search for specific pools, and easily filter the subnet with a specific pool, by searching for part of the pool ranges, e.g. 3.200.

Stork is able to display pool utilization for each subnet, and displays the absolute number of addresses allocated and percentage of usage. There are two thresholds: 80% (warning; the pool utilization bar becomes orange) and 90% (critical; the pool utilization bar becomes red).

---

**Note:** As of Stork 0.5.0, if two or more servers are handling the same subnet (e.g. a HA pair), the same subnet is listed multiple times. This limitation will be addressed in future releases.

---

### 3.6.4 IPv4 and IPv6 Networks

Kea uses the concept of a shared network, which is essentially a stack of subnets deployed on the same physical link. Stork is able to retrieve information about shared networks and aggregate it across all configured Kea servers. The Shared Networks view allows for the inspection of networks and the subnets that belong in them. Pool utilization is shown for each subnet.

### 3.6.5 Host Reservations

Kea DHCP servers can be configured to assign static resources or parameters to the DHCP clients communicating with the servers. Most commonly these resources are the IP addresses or delegated prefixes. However, Kea also allows for assigning hostnames, PXE boot parameters, client classes, DHCP options, and others. The mechanism by which a given set of resources and/or parameters is associated with a given DHCP client is called “host reservations.”

A host reservation consists of one or more DHCP identifiers used to associate the reservation with a client, e.g. MAC address, DUID, or client identifier; and a collection of resources and/or parameters to be returned to the client if the client’s DHCP message is associated with the host reservation by one of the identifiers. Stork can detect existing host reservations specified both in the configuration files of the monitored Kea servers and in the host database backends accessed via the Kea host\_cmds premium hooks library. At present, Stork provides no means to update or delete host reservations.

All reservations detected by Stork can be listed by selecting the DHCP menu option and then selecting Hosts.

The first column in the presented view displays one or more DHCP identifiers for each host in the format `hw-address=0a:1b:bd:43:5f:99`, where `hw-address` is the identifier type. In this case, the identifier type is the MAC address of the DHCP client for which the reservation has been specified. Supported identifier types are described in the following sections of the Kea ARM: [Host Reservation in DHCPv4](#) and [Host Reservation in DHCPv6](#). If multiple identifiers are present for a reservation, the reservation will be assigned when at least one of the identifiers matches the received DHCP packet.

The second column, `IP Reservations`, includes the static assignments of the IP addresses and/or delegated prefixes to the clients. There may be one or more IP reservations for each host.

The `Hostname` column contains an optional hostname reservation, i.e. the hostname assigned to the particular client by the DHCP servers via the `Hostname` or `Client FQDN` option.

The `Global/Subnet` column contains the prefixes of the subnets to which the reserved IP addresses and prefixes belong. If the reservation is global, i.e. is valid for all configured subnets of the given server, the word “global” is shown instead of the subnet prefix.

Finally, the `AppID @ Machine` column includes one or more links to Kea applications configured to assign each reservation to the client. The number of applications will typically be greater than one when Kea servers operate in the High Availability setup. In this case, each of the HA peers uses the same configuration and may allocate IP addresses and delegated prefixes to the same set of clients, including static assignments via host reservations. If HA peers are configured correctly, the reservations they share will have two links in `AppID @ Machine` column. Next to each link there is a little label indicating whether the host reservation for the given server has been specified in its configuration file or a host database (via `host_cmds` premium hooks library).

The `Filter hosts` input box is located above the `Hosts` table. It allows for filtering the hosts by identifier types, identifier values, IP reservations, hostnames and by globality i.e. `is:global` and `not:global`. When filtering by DHCP identifier values, it is not necessary to use colons between the pairs of hexadecimal digits. For example, the reservation `hw-address=0a:1b:bd:43:5f:99` will be found regardless of whether the filtering text is `1b:bd:43` or `1bbd43`.

### 3.6.6 Sources of Host Reservations

There are two ways to configure the Kea servers to use host reservations. First, the host reservations can be specified within the Kea configuration files; see [Host Reservation in DHCPv4](#) for details. The other way is to use a host database backend, as described in [Storing Host Reservations in MySQL, PostgreSQL, or Cassandra](#). The second solution requires the given Kea server to be configured to use the `host_cmds` premium hooks library. This library implements control commands used to store and fetch the host reservations from the host database which the Kea server is connected to. If the `host_cmds` hooks library is not loaded, Stork will only present the reservations specified within the Kea configuration files.

Stork periodically fetches the reservations from the host database backends and updates them in the local database. The default interval at which Stork refreshes host reservation information is set to 60 seconds. This means that an update in the host reservation database will not be visible in Stork until up to 60 seconds after it was applied. This interval is currently not configurable.

---

**Note:** As of the Stork 0.7.0 release, the list of host reservations must be manually refreshed by reloading the browser page to observe the most recent updates fetched from the Kea servers.

---

### 3.6.7 Kea High Availability Status

When viewing the details of the Kea application for which High Availability is enabled (via the `libdhcp_ha.so` hooks library), the High Availability live status is presented and periodically refreshed for the DHCPv4 and/or DHCPv6 daemon configured as primary or secondary/standby server. The status is not displayed for the server configured as an



HA backup. See the [High Availability section in the Kea ARM](#) for details about the roles of the servers within the HA setup.

The following picture shows a typical High Availability status view displayed in the Stork UI.

### High Availability

Local server	Remote server (4 seconds ago)
State: <i>load-balancing</i>	State: <i>load-balancing</i>
Role: <i>primary</i>	Role: <i>secondary</i>
Scopes served: <i>server1</i>	Scopes served: <i>(none)</i>
Note	
The local server responds to the entire DHCP traffic.	

The local server is the DHCP server (daemon) belonging to the application for which the status is displayed; the remote server is its active HA partner. The remote server belongs to a different application running on a different machine, and this machine may or may not be monitored by Stork. The statuses of both the local and the remote server are fetched by sending the `status-get` command to the Kea server whose details are displayed (the local server). In the load-balancing and hot-standby modes the local server periodically checks the status of its partner by sending the `ha-heartbeat` command to it. Therefore, this information is not always up-to-date; its age depends on the heartbeat command interval (typically 10 seconds). The status of the remote server includes the age of the data displayed.

The status information contains the role, state, and scopes served by each HA partner. In the usual HA case, both servers are in load-balancing state, which means that both are serving DHCP clients and there is no failure. If the remote server crashes, the local server transitions to the partner-down state, which will be reflected in this view. If the local server crashes, this will manifest itself as a communication problem between Stork and the server.

As of Stork 0.8.0 release, the High Availability view may also contain the information about the heartbeat status between the two servers and the information about the failover progress. This information is only available while monitoring Kea 1.7.8 versions and later.

The failover progress information is only presented when one of the active servers has been unable to communicate with the partner via the heartbeat exchange for a time exceeding the `max-heartbeat-delay` threshold. If the server is configured to monitor the DHCP traffic directed to the partner to verify that the partner is not responding to this traffic before transitioning to the partner-down state, the information about the number of unacked clients (clients which failed to get the lease), connecting clients (all clients currently trying to get the lease from the partner) and the number of analyzed packets are displayed. The system administrator may use this information to diagnose why the failover transition has not taken place or when such transition is likely to happen.

More about High Availability status information provided by Kea can be found in the [Kea ARM](#).

### 3.6.8 Viewing Kea Log

Stork offers a simple logs viewing mechanism to diagnose issues with monitored applications.

**Note:** As of Kea 0.10 release, this mechanism only supports viewing Kea log files. Viewing BIND9 logs is not supported yet. Monitoring other logging locations such as: stdout, stderr or syslog is also not supported.

---

Kea can be configured to log into multiple destinations. Different types of log messages may be output into different log files, syslog, stdout or stderr. The list of log destinations used by the Kea application is available on the Kea app page. Click on the Kea app whose logs you want to view. Next, select the Kea daemon by clicking on one of the tabs, e.g. DHCPv4 tab. Scroll down to the Loggers section.

This section contains a table with a list of configured loggers for the selected daemon. For each configured logger the logger's name, logging severity and output location are presented. The possible output locations are: log file, stdout, stderr or syslog. It is only possible to view the logs output to the log files. Therefore, for each log file there is a link which leads to the log viewer showing the selected file's contents. The loggers which output to the stdout, stderr and syslog are also listed but the links to the log viewer are not available for them.

Clicking on the selected log file navigates to the log viewer for this file. By default, the viewer displays the tail of the log file up to 4000 characters. Depending on the network latency and the size of the log file, it may take several seconds or more before the log contents are fetched and displayed.

The log viewer title bar comprises three buttons. The button with the refresh icon triggers log data fetch without modifying the size of the presented data. Clicking on the + button extends the size of the viewed log tail by 4000 characters and refreshes the data in the log viewer. Conversely, clicking on the – button reduces the amount of presented data by 4000 characters. Every time any of these buttons is clicked, the viewer discards currently presented data and displays the latest part of the log file tail.

Please keep in mind that extending the size of the viewed log tail may cause slowness of the log viewer and network congestion as you increase the amount of data fetched from the monitored machine.

## 3.7 Dashboard

The Main Stork page presents a dashboard. It contains a panel with information about DHCP and a panel with events observed or noticed by Stork server.

### 3.7.1 DHCP Panel

DHCP panel includes two sections: one for DHCPv4 and one for DHCPv6. Each section contains 3 kinds of information:

- list of up to 5 subnets with the highest pool utilization
- list of up to 5 shared networks with the highest pool utilization
- statistics about DHCP

### 3.7.2 Events Panel

Events panel presents the list of the most recent events captured by the Stork server. There are 3 urgency levels of the events: info, warning and error. Events pertaining to the particular entities, e.g. machines or applications, provide a link to a web page containing the information about the given object.

## 3.8 Events Page

Events page presents a list of all events. It allows for filtering events by:

- urgency level,
- machine,
- application type (Kea, BIND 9)
- daemon type (DHCPv4, DHCPv6, named, etc)
- user who caused given event (this is available to `super-admin` group only)



## CHAPTER 4

---

### Backend API

---

Stork Agent provides a REST API. The API is generated using [Swagger](#). Source YAML files are stored in the *api/* directory in the source files. To view the REST API documentation, open the Stork interface, click Help and choose Stork API Docs (SwaggerUI) or Stork API Docs (Redoc).



---

**Note:** We acknowledge that users and developers have different needs, so the user and developer documents should eventually be separated. However, since the project is still in its early stages, this section is kept in the Stork ARM for convenience.

---

### 5.1 Rakefile

Rakefile is a script for performing many development tasks like building source code, running linters, running unit tests, and running Stork services directly or in Docker containers.

There are several other Rake targets. For a complete list of available tasks, use `rake -T`. Also see the Stork [wiki](#) for detailed instructions.

### 5.2 Generating Documentation

To generate documentation, simply type `rake doc`. `Sphinx` and `rtd-theme` must be installed. The generated documentation will be available in the `doc/singlehtml` directory.

### 5.3 Setting Up the Development Environment

The following steps install Stork and its dependencies natively, i.e. on the host machine, rather than using Docker images.

First, PostgreSQL must be installed. This is OS-specific, so please follow the instructions from the [Installation](#) chapter.

Once the database environment is set up, the next step is to build all the tools. Note the first command below downloads some missing dependencies and installs them in a local directory. This is done only once and is not needed for future rebuilds, although it is safe to rerun the command.

```
$ rake build_backend
$ rake build_ui
```

The environment should be ready to run! Open three consoles and run the following three commands, one in each console:

```
$ rake run_server
```

```
$ rake serve_ui
```

```
$ rake run_agent
```

Once all three processes are running, connect to <http://localhost:8080> via a web browser. See *Using Stork* for initial password information or for adding new machines to the server.

The `run_agent` runs the agent directly on the current operating system, natively; the exposed port of the agent is 8888.

There are other Rake tasks for running preconfigured agents in Docker containers. They are exposed to the host on specific ports.

When these agents are added as machines in the `Stork Server UI`, both a localhost address and a port specific to a given container must be specified. This is a list of containers can be found in *Docker Containers for Development* section.

### 5.3.1 Installing Git Hooks

There is a simple git hook that inserts the issue number in the commit message automatically; to use it, go to the `utils` directory and run the `git-hooks-install` script. It will copy the necessary file to the `.git/hooks` directory.

## 5.4 Agent API

The connection between the server and the agents is established using gRPC over http/2. The agent API definition is kept in the `backend/api/agent.proto` file. For debugging purposes, it is possible to connect to the agent using the `grpcurl` tool. For example, a list of currently provided gRPC calls may be retrieved with this command:

```
$ grpcurl -plaintext -proto backend/api/agent.proto localhost:8888 describe
agentapi.Agent is a service:
service Agent {
  rpc detectServices ( .agentapi.DetectServicesReq ) returns ( .agentapi.
↪DetectServicesRsp );
  rpc getState ( .agentapi.GetStateReq ) returns ( .agentapi.GetStateRsp );
  rpc restartKea ( .agentapi.RestartKeaReq ) returns ( .agentapi.RestartKeaRsp );
}
```

Specific gRPC calls can also be made. For example, to get the machine state, the following command can be used:

```
$ grpcurl -plaintext -proto backend/api/agent.proto localhost:8888 agentapi.Agent.
↪getState
{
  "agentVersion": "0.1.0",
  "hostname": "copernicus",
  "cpus": "8",
```

(continues on next page)



(continued from previous page)

```
"cpusLoad": "1.68 1.46 1.28",
"memory": "16",
"usedMemory": "59",
"uptime": "2",
"os": "darwin",
"platform": "darwin",
"platformFamily": "Standalone Workstation",
"platformVersion": "10.14.6",
"kernelVersion": "18.7.0",
"kernelArch": "x86_64",
"hostID": "c41337a1-0ec3-3896-a954-a1f85e849d53"
}
```

## 5.5 ReST API

The primary user of the ReST API is the Stork UI in a web browser. The definition of the ReST API is located in the `api` folder and is described in Swagger 2.0 format.

The description in Swagger is split into multiple files. Two files comprise a tag group:

- `*-paths.yaml` - defines URLs
- `*-defs.yaml` - contains entity definitions

All these files are combined by the `yamllinc` tool into a single Swagger file `swagger.yaml`. Then, `swagger.yaml` generates code for:

- the UI fronted by `swagger-codegen`
- the backend in Go lang by `go-swagger`

All these steps are accomplished by Rakefile.

## 5.6 Backend Unit Tests

There are unit tests for backend part (agent and server) written in Go. They can be run using Rake:

```
$ rake unittest_backend
```

This requires preparing a database in PostgreSQL. One way to avoid doing this manually is by using a docker container with PostgreSQL which is automatically created when running the following Rake task:

```
$ rake unittest_backend_db
```

This one task spawns a container with PostgreSQL in the background and then it runs unit tests. When the tests are completed the database is shutdown and removed.

### 5.6.1 Unit Tests Database

When docker container with a database is not used for unit tests, the PostgreSQL server must be started and the following role must be created:

```
postgres=# CREATE USER storktest WITH PASSWORD 'storktest';
CREATE ROLE
postgres=# ALTER ROLE storktest SUPERUSER;
ALTER ROLE
```

To point unit tests to our specific database set `POSTGRES_ADDR` environment variable, e.g.:

```
$ rake unittest_backend POSTGRES_ADDR=host:port
```

By default it points to `localhost:5432`.

Similarly, if the db setup requires a password other than the default `storktest`, it's convenient to set up `PGPASSWORD` variable accordingly. This can be done the following way:

```
$ rake unittest_backend PGPASSWORD=secret123
```

Note there's no need to create the `storktest` database itself. It is created and destroyed by the Rakefile task.

## 5.6.2 Unit Tests Coverage

At the end of tests execution there is coverage report presented. If coverage of any module is below a threshold of 35% then an error is raised.

## 5.7 Backend Benchmarks

Benchmarks are part of the backend unit tests. They are implemented using the `golang` “testing” library and they test performance sensitive parts of the backend. Unlike unit tests, the benchmarks do not return pass/fail status. They measure average execution time of functions and print the results to the console.

In order to run unit tests with benchmarks the `benchmark` environment variable must be specified as follows:

```
$ rake unittest_backend benchmark=true
```

This command will run all unit tests and all benchmarks. Running benchmarks without unit tests is possible using the combination of `benchmark` and `test` environment variables:

```
$ rake unittest_backend benchmark=true test=Bench
```

Benchmarks are useful to test performance of complex functions and find bottlenecks. When working on improving performance of a function, examining a benchmark result before and after the changes is a good practice to ensure that the goals of the changes are achieved.

Similarly, adding a new logic to a function will often cause performance degradation and careful examination of the benchmark result drop for that function may be a driver for improving efficiency of the new code.

## 5.8 WebUI Unit Tests

We do have the WebUI tests. We take advantage of the unit-tests generated automatically by Angular. The simplest way to run these tests is by using rake tasks:

```
rake build_ui
rake ng_test
```

The tests require Chromium (on Linux) or Chrome (on Mac) browser. The `rake ng_test` task will attempt to locate the browser binary and launch it automatically. If the browser binary is not found in the default locations the rake task will return an error. It is possible to set the location manually by setting the `CHROME_BIN` environment variable. For example:

```
export CHROME_BIN=/usr/local/bin/chromium-browser
rake ng_test
```

By default, the tests launch the browser in the headless mode in which test results and any possible errors are printed in the console. Though, in some situations it is useful to run the browser in non headless mode because it provides debugging features in Chrome's graphical interface. It also allows for selectively running the tests. Run the tests in non headless mode using the `debug` variable appended to the rake command:

```
rake ng_test debug=true
```

The tests are being run in random order by default which makes it sometimes difficult to chase the individual errors. One convenient way to run them is to click Debug, click Options and unset the "run tests in random order". This will run the tests always in the same order.

You can run specific test by clicking on its name. For example, you can run one specific test by opening this link <http://localhost:9876/debug.html?spec=ProfilePageComponent>

When adding new component or service with `ng generate component/service ...`, the Angular framework will add `.spec.ts` file for you with a boilerplate code there. In most cases, the first step in running those tests is to add necessary Stork imports. If in doubt, take a look at commits on [https://gitlab.isc.org/isc-projects/stork/-/merge\\_requests/97](https://gitlab.isc.org/isc-projects/stork/-/merge_requests/97). There are many examples how to fix failing tests.

## 5.9 System Tests

System tests for *Stork* are designed to test *Stork* in distributed environment. They allow for testing several *Stork* servers and agents running at the same time in one test case. They are run inside LXD containers. It is possible to set up *Kea* or *BIND 9* services along *Stork* agents. The framework enables tinkering in containers so custom *Kea* configs can be deployed or specific *Kea* daemons can be stopped.

The tests can use:

- Stork server ReST API directly or
- Stork web UI via Selenium.

### 5.9.1 Dependencies

System tests require:

- Linux operating system (preferred Ubuntu or Fedora)
- Python 3
- LXD containers (<https://linuxcontainers.org/lxd/introduction>)

### 5.9.2 LXD Installation

The easiest way to install LXD is to use `snap`. So first let's install `snap`.

On Fedora:

```
$ sudo dnf install snapd
```

On Ubuntu:

```
$ sudo apt install snapd
```

Then install LXD:

```
$ sudo snap install lxd
```

And then add your user to lxd group:

```
$ sudo usermod -a -G lxd $USER
```

Now you need to relogin to make your presence in lxd group visible in the shell session.

After installing LXD it requires initialization. Run:

```
$ lxd init
```

and then for each question press **Enter** i.e. use default values:

```
Would you like to use LXD clustering? (yes/no) [default=no]: **Enter**
Do you want to configure a new storage pool? (yes/no) [default=yes]: **Enter**
Name of the new storage pool [default=default]: **Enter**
Name of the storage backend to use (dir, btrfs) [default=btrfs]: **Enter**
Would you like to create a new btrfs subvolume under /var/snap/lxd/common/lxd? (yes/
↵no) [default=yes]: **Enter**
Would you like to connect to a MAAS server? (yes/no) [default=no]: **Enter**
Would you like to create a new local network bridge? (yes/no) [default=yes]: ↵
↵**Enter**
What should the new bridge be called? [default=lxdbr0]: **Enter**
What IPv4 address should be used? (CIDR subnet notation, "auto" or "none") ↵
↵[default=auto]: **Enter**
What IPv6 address should be used? (CIDR subnet notation, "auto" or "none") ↵
↵[default=auto]: **Enter**
Would you like LXD to be available over the network? (yes/no) [default=no]: **Enter**
Would you like stale cached images to be updated automatically? (yes/no) ↵
↵[default=yes] **Enter**
Would you like a YAML "lxd init" preseed to be printed? (yes/no) [default=no]: ↵
↵**Enter**
```

More details can be found on: <https://linuxcontainers.org/lxd/getting-started-cli/>

One of the questions was about a subvolume. It is stored in `/var/snap/lxd/common/lxd`. This subvolume is used to store images and containers. If the space is exhausted then it is not possible to create new containers. This is not connected with your total disk space but the space in this subvolume. To free space you may remove some stale images or stopped containers. Basic usage of LXD is presented on: <https://linuxcontainers.org/lxd/getting-started-cli/#lxd-client>

### 5.9.3 Running System Tests

After preparing all dependencies now it is possible to start tests. But first RPM and deb Stork packages need to be prepared. This can be done with this Rake task:

```
$ rake build_pkgs_in_docker
```

When we have packages then the tests can be invoked by the following Rake task:

```
$ rake system_tests
```

This command beside running the tests first prepares Python virtual environment (`venv`) where `pytest` and other Python dependencies are installed. `pytest` is a Python testing framework that is used in Stork system tests.

At the bottom of logs there are listed test cases with their result status.

The tests can be invoked directly using `pytest` but first we need to change directory to `tests/system`:

```
$ cd tests/system
$ ./venv/bin/pytest --tb=long -l -r ap -s tests.py
```

The switches passed to `pytest` are:

- `--tb=long` in case of failures present long format of traceback
- `-l` show values of local variables in tracebacks
- `-r ap` at the end of execution print report that includes (p)assed and (a)ll except passed (p)

To run particular test case add it just after `test.py`:

```
$ ./venv/bin/pytest --tb=long -l -r ap -s tests.py::test_users_management[centos/7-
↳ubuntu/18.04]
```

To get a list of tests without actually running them, the following command can be used:

```
$ ./venv/bin/pytest --collect-only tests.py
```

The test names of available tests will be printed as `<Function name_of_the_test>`.

A single test case can be run using a `rake` task with the test variable set to the test name:

```
$ rake system_tests test=tests.py::test_users_management[centos/7-ubuntu/18.04]
```

## 5.9.4 Developing System Tests

System tests are defined in `tests.py` and other files that start from `test_`. There are two other files that are defining framework for Stork system tests:

- `conftest.py` - it defines hooks for `pytest`s
- `containers.py` - it handles LXD containers: starting/stopping, communication like invoking commands, uploading/downloading files, installing and preparing Stork Agent and Server, and Kea, and other dependencies that they requires.

Most of tests are constructed as follow:

```
@pytest.mark.parametrize("agent, server", SUPPORTED_DISTROS)
def test_machines(agent, server):
    # login to stork server
    r = server.api_post('/sessions',
                        json=dict(useremail='admin', userpassword='admin'),
                        expected_status=200)
    assert r.json()['login'] == 'admin'

    # add machine
    machine = dict(
        address=agent.mgmt_ip,
```

(continues on next page)

(continued from previous page)

```

    agentPort=8080)
    r = server.api_post('/machines', json=machine, expected_status=200)
    assert r.json()['address'] == agent.mgmt_ip

    # wait for application discovery by Stork Agent
    for i in range(20):
        r = server.api_get('/machines')
        data = r.json()
        if len(data['items']) == 1 and \
            len(data['items'][0]['apps'][0]['details']['daemons']) > 1:
            break
        time.sleep(2)

    # check discovered application by Stork Agent
    m = data['items'][0]
    assert m['apps'][0]['version'] == '1.7.3'

```

Let's dissect this code and explain each part.

```
@pytest.mark.parametrize("agent, server", SUPPORTED_DISTROS)
```

This indicates that we are parametrizing the test and there will be one or more instances of this test in execution for each set of parameters.

The constant `SUPPORTED_DISTROS` defines two sets of operating systems for testing:

```

SUPPORTED_DISTROS = [
    ('ubuntu/18.04', 'centos/7'),
    ('centos/7', 'ubuntu/18.04')
]

```

The first set indicates that for Stork agent a Ubuntu 18.04 should be used in LXD container and for Stork server Centos 7. The second sets is the opposite of the first one.

The next line:

```
def test_machines(agent, server):
```

defines the test function. Normally agent and server argument would get text values 'ubuntu/18.04' and 'centos/7' but there is prepared a hook in `conftest.py`, in `pytest_pyfunc_call()` function that intercepts these arguments and based on them it spins up LXD containers with indicated operating systems. This hook also at the end of the test collects Stork logs from these containers and stores them in `test-results` folder for later user analysis if needed.

So instead text values the hook replaces the arguments with references to actual LXC container objects. Then the test may interact directly with them. Beside substituting agent and server arguments the hook intercepts any argument that starts with agent and server. This way we may have several agents in the test, e.g. agent1 or agent\_kea, agent\_bind9.

Then we are logging into the Stork server using its ReST API:

```

# login to stork server
r = server.api_post('/sessions',
                    json=dict(useremail='admin', userpassword='admin'),
                    expected_status=200)
assert r.json()['login'] == 'admin'

```

And then we are adding a machine with Stork agent to Stork server:

```
# add machine
machine = dict(
    address=agent.mgmt_ip,
    agentPort=8080)
r = server.api_post('/machines', json=machine, expected_status=200)
assert r.json()['address'] == agent.mgmt_ip
```

Here we have a check that verifies returned address of the machine.

Then we need to wait until Stork agent detects Kea application and reports it to Stork server. We are pulling periodically the server if it received information about Kea app.

```
# wait for application discovery by Stork Agent
for i in range(20):
    r = server.api_get('/machines')
    data = r.json()
    if len(data['items']) == 1 and \
        len(data['items'][0]['apps'][0]['details']['daemons']) > 1:
        break
    time.sleep(2)
```

At the end we are verifying returned data about Kea application:

```
# check discovered application by Stork Agent
m = data['items'][0]
assert m['apps'][0]['version'] == '1.7.3'
```

## 5.10 Docker Containers for Development

To ease developemnt, there are several Docker containers available. These containers and several more are used in Stork Demo that is described in *Demo* chapter. The full description of each container can found in that chapter.

The following Rake tasks are starting these containers.

Table 1: Rake tasks for managing development containers.

Rake task	Description
rake build_kea_container	Build a container <i>agent-kea</i> with Stork Agent and Kea with DHCPv4.
rake run_kea_container	Start <i>agent-kea</i> container. Published port is 8888.
rake build_kea6_container	Build a <i>agent-kea6</i> container with Stork Agent and Kea with DHCPv6.
rake run_kea6_container	Start <i>agent-kea6</i> container. Published port is 8886.
rake build_kea_ha_container	Build two containers, <i>agent-kea-ha1</i> and <i>agent-kea-ha2</i> , with Stork Agent and Kea with DHCPv4 that are configured to work together in <i>High Availability</i> mode.
rake run_kea_ha_container	Start <i>agent-kea-ha1</i> and <i>agent-kea-ha2</i> containers. Published ports are 8881 and 8882.
rake build_kea_hosts_container	Build a <i>agent-kea-hosts</i> container with Stork Agent and Kea with DHCPv4 with host reservations stored in a database. It requires <b>premium</b> features.
rake run_kea_hosts_container	Start <i>agent-kea-hosts</i> container. It requires <b>premium</b> features.

Continued on next page

Table 1 – continued from previous page

Rake task	Description
<code>rake build_bind9_container</code>	Build a <i>agent-bind9</i> container with Stork Agent and BIND 9.
<code>rake run_bind9_container</code>	Start <i>agent-bind9</i> container. Published port is 9999.

## 5.11 Packaging

There are scripts for packaging the binary form of Stork. There are two supported formats:

- RPM
- deb

The RPM package is built on the latest CentOS version. The deb package is built on the latest Ubuntu LTS.

There are two packages built for each system: a server and an agent.

There are Rake tasks that perform the entire build procedure in a Docker container: *build\_rpms\_in\_docker* and *build\_debs\_in\_docker*. It is also possible to build packages directly in the current operating system; this is provided by the *deb\_agent*, *rpm\_agent*, *deb\_server*, and *rpm\_server* Rake tasks.

Internally, these packages are built by FPM (<https://fpm.readthedocs.io/>). The containers that are used to build packages are prebuilt with all dependencies required, using the *build\_fpm\_containers* Rake task. The definitions of these containers are placed in *docker/pkgs/centos-8.txt* and *docker/pkgs/ubuntu-18-04.txt*.



A demo installation of `Stork` can be used to demonstrate `Stork` capabilities but can be used for its development as well.

The demo installation uses *Docker* and *Docker Compose* to set up all *Stork* services. It contains:

- Stork Server
- Stork Agent with Kea DHCPv4
- Stork Agent with Kea DHCPv6
- Stork Agent with Kea HA-1 (high availability server 1)
- Stork Agent with Kea HA-2 (high availability server 2)
- Stork Agent with BIND 9
- Stork Environment Simulator
- PostgreSQL database
- Prometheus & Grafana

These services allow observation of many `Stork` features.

## 6.1 Requirements

Running the `Stork Demo` requires the same dependencies as building `Stork`, which is described in the *Installing from Sources* chapter.

Besides the standard dependencies, the `Stork Demo` requires:

- Docker
- Docker Compose

For details, please see the `Stork` wiki <https://gitlab.isc.org/isc-projects/stork/wikis/Development-Environment>.

## 6.2 Setup Steps

The following command retrieves all required software (go, goswagger, nodejs, Angular dependencies, etc.) to the local directory. No root password is necessary. Then it prepares Docker images and starts them up.

```
$ rake docker_up
```

Once the build process finishes, the Stork UI is available at <http://localhost:8080/>. Use any browser to connect.

### 6.2.1 Premium Features

It is possible to run the demo with premium features enabled in Kea apps. It requires starting the demo with an access token to the Kea premium repositories. Access tokens can be found on <https://cloudsmith.io/~isc/repos/kea-1-7-prv/setup/#formats-deb>. The token can be found inside this URL on that page: `https://dl.cloudsmith.io/<access token>/isc/kea-1-7-prv/cfg/setup/bash.deb.sh`. This web page and the token are available only to ISC employees and paid customers of ISC.

```
$ rake docker_up cs_repo_access_token=<access token>
```

## 6.3 Demo Containers

The setup procedure creates several Docker containers. Their definition is stored in `docker-compose.yaml` file in Stork source code repository.

These containers have Stork production services and components:

**server** This container is essential. It runs the Stork server, which interacts with all the agents and the database and exposes the API. Without it, Stork will not be able to function.

**webui** This container is essential in most circumstances. It provides the front-end web interface. It is potentially unnecessary with the custom development of a Stork API client.

**agent-bind9** This container runs a BIND 9 server. With this container, the agent can be added as a machine and Stork will begin monitoring its BIND 9 service.

**agent-bind9-2** This container also runs a BIND 9 server, for the purpose of experimenting with two different DNS servers.

**agent-kea** This container runs a Kea DHCPv4 server. With this container, the agent can be added as a machine and Stork will begin monitoring its Kea DHCPv4 service.

**agent-kea6** An agent with a Kea DHCPv6 server.

**agent-kea-ha1 and agent-kea-ha2** These two containers should, in general, be run together. They each have a Kea DHCPv4 server instance configured in a HA pair. With both running and registered as machines in Stork, users can observe certain HA mechanisms, such as one taking over the traffic if the partner becomes unavailable.

**agent-kea-many-subnets** An agent with a Kea DHCPv4 server that has many subnets defined in its config (about 7000)

These are containers with 3rd party services that are required by Stork:

**postgres** This container is essential. It runs the PostgreSQL database that is used by the Stork server. Without it, the Stork server will produce error messages about an unavailable database.

**prometheus** Prometheus, a monitoring solution (<https://prometheus.io/>). This container is used for monitoring applications. It is preconfigured to monitor Kea and BIND 9 containers.

**grafana** This is a container with Grafana (<https://grafana.com/>), a dashboard for Prometheus. It is preconfigured to pull data from a Prometheus container and show Stork dashboards.

These are supporting containers:

**simulator** Stork Environment Simulator, a web application that can run DHCP traffic using perfdhcp (useful to observe non-zero statistics coming from Kea), run DNS traffic using dig and flamethrower (useful to observe non-zero statistics coming from BIND 9), and start and stop any service in any other container (useful to simulate e.g. crash of Kea by stopping it).

---

**Note:** The containers running the Kea and BIND 9 applications are for demo purposes only. They allow users to quickly start experimenting with Stork without having to manually deploy Kea and/or BIND 9 instances.

---

The PostgreSQL database schema is automatically migrated to the latest version required by the Stork server process.

The setup procedure assumes those images are fully under Stork control. If there are existing images, they will be overwritten.

## 6.4 Initialization

Stork Server requires some initial information:

1. Go to <http://localhost:8080/machines/all>
2. Add new machines (leave the default port):
  1. agent-kea
  2. agent-kea6
  3. agent-kea-ha1
  4. agent-kea-ha2
  5. agent-bind9
  6. agent-bind9-2

## 6.5 Stork Environment Simulator

Stork Environment Simulator allows:

- sending DHCP traffic to Kea applications
- sending DNS requests to BIND 9 applications
- stopping and starting Stork Agents, Kea and BIND 9 daemons

Stork Environment Simulator allows DHCP traffic to be sent to selected subnets pre-configured in Kea instances, with a limitation: it is possible to send traffic to only one subnet from a given shared network.

Stork Environment Simulator also allows sending DNS traffic to selected DNS servers.

Stork Environment Simulator can add all the machines available in the demo setup. It can stop and start selected Stork Agents, and Kea and BIND 9 applications. This is useful to simulate communication problems between applications, Stork Agents and the Stork Server.

Stork Environment Simulator can be found at: <http://localhost:5000/>

For development purposes simulator can be started directly by command:

```
$ rake run_sim
```

## 6.6 Prometheus

The Prometheus instance is preconfigured and pulls statistics from:

- node exporters: agent-kea:9100, agent-bind9:9100, agent-bind9:9100
- kea exporters embedded in stork-agent: agent-kea:9547, agent-kea6:9547, agent-kea-ha1:9547, agent-kea-ha2:9547
- bind exporters embedded in stork-agent: agent-bind9:9119, agent-bind9-2:9119

The Prometheus web page can be found at: <http://localhost:9090/>

## 6.7 Grafana

The Grafana instance is preconfigured as well. It pulls data from Prometheus and loads dashboards from the Stork repository, in the Grafana folder.

The Grafana web page can be found at: <http://localhost:3000/>

## 7.1 stork-server - The central Stork server

### 7.1.1 Synopsis

**stork-server**

### 7.1.2 Description

The `stork-server` provides the main Stork Server capabilities. In every Stork deployment, there should be exactly one `stork-server`.

### 7.1.3 Arguments

The Stork Server takes the following arguments:

**-h or --help** displays list of available parameters.

**-v or --version** returns `stork-server` version.

**-d or --db-name=** the name of the database to connect to (default: `stork`) [`$(STORK_DATABASE_NAME)`]

**-u or --db-user** the user name to be used for database connections (default: `stork`) [`$(STORK_DATABASE_USER_NAME)`]

**--db-host** the name of the host where database is available (default: `localhost`) [`$(STORK_DATABASE_HOST)`]

**-p or --db-port** the port on which the database is available (default: `5432`) [`$(STORK_DATABASE_PORT)`]

**--db-trace-queries=** enable tracing SQL queries: `run` (only runtime, without migrations), `all` (migrations and run-time), `all` is the default and covers both migrations and run-time. enable tracing SQL queries [`$(STORK_DATABASE_TRACE)`]

**--rest-cleanup-timeout** grace period for which to wait before killing idle connections (default: `10s`)

**--rest-graceful-timeout** grace period for which to wait before shutting down the server (default: 15s)

**--rest-max-header-size** controls the maximum number of bytes the server will read parsing the request header's keys and values, including the request line. It does not limit the size of the request body. (default: 1MiB)

**--rest-host** the IP to listen on for connections over ReST API [\${STORK\_REST\_HOST}]

**--rest-port** the port to listen on for connections over ReST API (default: 8080) [\${STORK\_REST\_PORT}]

**--rest-listen-limit** limit the number of outstanding requests

**--rest-keep-alive** set the TCP keep-alive timeouts on accepted connections. It prunes dead TCP connections (e.g. closing laptop mid-download) (default: 3m)

**--rest-read-timeout** maximum duration before timing out read of the request (default: 30s)

**--rest-write-timeout** maximum duration before timing out write of the response (default: 60s)

**--rest-tls-certificate** the certificate to use for secure connections [\${STORK\_REST\_TLS\_CERTIFICATE}]

**--rest-tls-key** the private key to use for secure connections [\${STORK\_REST\_TLS\_PRIVATE\_KEY}]

**--rest-tls-ca** the certificate authority file to be used with mutual tls auth [\${STORK\_REST\_TLS\_CA\_CERTIFICATE}]

**--rest-static-files-dir** directory with static files for UI [\${STORK\_REST\_STATIC\_FILES\_DIR}]

Note there is no argument for database password, as the command line arguments can sometimes be seen by other users. You can pass it using STORK\_DATABASE\_PASSWORD variable.

## 7.1.4 Mailing Lists and Support

There are public mailing lists available for the Stork project. **stork-users** (stork-users at lists.isc.org) is intended for Stork users. **stork-dev** (stork-dev at lists.isc.org) is intended for Stork developers, prospective contributors, and other advanced users. The lists are available at <https://lists.isc.org>. The community provides best-effort support on both of those lists.

Once stork will become more mature, ISC will be providing professional support for Stork services.

## 7.1.5 History

The `stork-server` was first coded in November 2019 by Michal Nowikowski and Marcin Siodelski.

## 7.1.6 See Also

*stork-agent* (8)

# 7.2 stork-agent - Stork agent that monitors BIND 9 and Kea services

## 7.2.1 Synopsis

**stork-agent** [-host] [-port]

## 7.2.2 Description

The `stork-agent` is a small tool that is being run on the systems that are running BIND 9 and Kea services. Stork server connects to the Stork Agent and uses it to monitor services remotely.

## 7.2.3 Arguments

Stork does not use explicit configuration file. Instead, its behavior can be controlled with command line switches and/or variables. The Stork Agent takes the following command line switches. Equivalent environment variables are listed in square brackets, where applicable.

**--listen-stork-only** listen for commands from the Stork Server only, but not for Prometheus requests. [`$STORK_AGENT_LISTEN_STORK_ONLY`]

**--listen-prometheus-only** listen for Prometheus requests only, but not for commands from the Stork Server. [`$STORK_AGENT_LISTEN_PROMETHEUS_ONLY`]

**-v** or **--version** show software version.

Stork Server flags:

**--host=** the IP or hostname to listen on for incoming Stork server connection. [`$STORK_AGENT_ADDRESS`]

**--port=** the TCP port to listen on for incoming Stork server connection. (default: 8080) [`$STORK_AGENT_PORT`]

Prometheus Kea Exporter flags:

**--prometheus-kea-exporter-host=** the IP or hostname to listen on for incoming Prometheus connection (default: 0.0.0.0) [`$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_ADDRESS`]

**--prometheus-kea-exporter-port=** the port to listen on for incoming Prometheus connection (default: 9547) [`$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_PORT`]

**--prometheus-kea-exporter-interval=** specifies how often the agent collects stats from Kea, in seconds (default: 10) [`$STORK_AGENT_PROMETHEUS_KEA_EXPORTER_INTERVAL`]

Prometheus BIND 9 Exporter flags:

**--prometheus-bind9-exporter-host=** the IP or hostname to listen on for incoming Prometheus connection (default: 0.0.0.0) [`$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_ADDRESS`]

**--prometheus-bind9-exporter-port=** the port to listen on for incoming Prometheus connection (default: 9119) [`$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_PORT`]

**--prometheus-bind9-exporter-interval=** specifies how often the agent collects stats from BIND 9, in seconds (default: 10) [`$STORK_AGENT_PROMETHEUS_BIND9_EXPORTER_INTERVAL`]

**-h** or **--help** displays list of available parameters.

## 7.2.4 Mailing Lists and Support

There are public mailing lists available for the Stork project. **stork-users** (stork-users at lists.isc.org) is intended for Stork users. **stork-dev** (stork-dev at lists.isc.org) is intended for Stork developers, prospective contributors, and other advanced users. The lists are available at <https://lists.isc.org>. The community provides best-effort support on both of those lists.

Once stork will become more mature, ISC will be providing professional support for Stork services.

## 7.2.5 History

The `stork-agent` was first coded in November 2019 by Michal Nowikowski.

## 7.2.6 See Also

`stork-server(8)`

# 7.3 stork-db-migrate - The Stork database migration tool

## 7.3.1 Synopsis

**stork-db-migrate** [*options*] *command*

## 7.3.2 Description

The `stork-db-migrate` is an optional tool that assists the database schema migrations. Usually, there is no need to use this tool, as Stork server always runs the migration scripts on startup. However, it may be useful for debugging and manual migrations.

## 7.3.3 Arguments

The Stork DB migration tools takes the following commands:

Available commands:

- `down` Revert last migration (or use `-t` to migrate to specific version)
- `init` Create schema versioning table in the database
- `reset` Revert all migrations
- `set_version` Set database version without running migrations
- `up` Run all available migrations (or use `-t` to migrate to specific version)
- `version` Print current migration version

Application Options:

- `-d, --db-name=` the name of the database to connect to (default: `stork`) [`STORK_DATABASE_NAME`]
- `-u, --db-user=` the user name to be used for database connections (default: `stork`) [`STORK_DATABASE_USER_NAME`]
- `--db-host=` the name of the host where database is available (default: `localhost`) [`STORK_DATABASE_HOST`]
- `-p, --db-port=` the port on which the database is available (default: `5432`) [`STORK_DATABASE_PORT`]
- `--db-trace-queries=` enable tracing SQL queries: `run` (only runtime, without migrations), `all` (migrations and run-time), `all` is the default and covers both migrations and run-time.enable tracing SQL queries [`STORK_DATABASE_TRACE`]
- `-h, --help` show help message

Note there is no argument for database password, as the command line arguments can sometimes be seen by other users. You can pass it using `STORK_DATABASE_PASSWORD` variable.



### 7.3.4 Mailing Lists and Support

There are public mailing lists available for the Stork project. **stork-users** (stork-users at lists.isc.org) is intended for Stork users. **stork-dev** (stork-dev at lists.isc.org) is intended for Stork developers, prospective contributors, and other advanced users. The lists are available at <https://lists.isc.org>. The community provides best-effort support on both of those lists.

Once stork will become more mature, ISC will be providing professional support for Stork services.

### 7.3.5 History

The `stork-db-migrate` was first coded in October 2019 by Marcin Siodelski.

### 7.3.6 See Also

*stork-agent (8)*, *stork-server (8)*



## CHAPTER 8

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`