Stellar-Supercluster

an automation tool for stellar-core

First things first!

- `git clone https://github.com/stellar/stellar-supercluster`
- Install an F# mode in your editor if you like
- Follow along!

Talk Overview

- What SSC is intended for vs. not-intended for
- What it depends on to build and run
- Crash course in Kubernetes
- How SSC uses Kubernetes
- How the SSC codebase is organized
- How to run it and debug it
- How to extend it

Part 1:
What is it intended for
(and not-intended for)

Primary purpose: replace SCC (with SSC, they're different!)

- Long-standing automation tool: `stellar_core_commander`
  - Written in Ruby, runs `stellar-core` locally or via Docker passing lots of environment variables
  - `confd` inside Docker image expands config templates using environment variables
  - Shell script entry-point in Docker image to wrap core tasks

What was wrong with SCC?

- Very hard to debug, extend, or even keep working
  - Too many steps to edit, debug, version, upgrade, make mistakes in: Ruby DSL → Ruby library → Env vars → Dockerfile → Shell script → Confd → stellar-core
  - No typechecking or even typo-checking. Easy to modify an hours-long test and have it fail right at the end because of a string-pasting config error
- Limited to small set of Docker hosts, hard to scale or redeploy
- No plan for how to extend to work with Kubernetes

How does SSC improve on things?

- Types: more coding errors caught before runtime, can use IDE / autocomplete
  - Stellar SDK is strongly typed
  - Kubernetes API is strongly typed
  - Internal abstractions in SSC (configs, HTTP JSON responses) all typed
- Kubernetes has more visibility and debugging tools, robust controls
  - All configs, logs, status of all components viewable by `kubectl` commands
  - Kubernetes reports metrics to Prometheus, easy to observe
  - CPU / RAM quotas enforced, much harder to crash the cluster
- Can scale cluster arbitrarily by adding more workers, scheduling is dynamic
  - Single developer gets remote access to $LOTS_OF_CORES

Ok but what was SCC intended for?

- Acceptance tests
  - Run by Jenkins "as often as possible", typically ~daily
  - Bigger & slower than merge-gating CI unit tests
- Performance / load tests
  - Run by hand on developer workstation
  - Evaluate overall scalability of core

Acceptance Tests

- Usually several `stellar-core` instances, communicating
- Either simulated private networks or the real live networks
- Version mixtures, protocol and database upgrades
- Generation of history archives, replay of history archives
- Most thorough test: replay all of pubnet's history

What is not intended

- Not for managing live operational deployments
  - If you hit ctrl-c while it's running, it has no good way of recovering aside from "delete all the Kubernetes objects in the namespace"
- Not for small iterative unit tests while developing
  - You have to have a Docker image to run, which only happens after CI passes

Part 2:
What it depends on to build and run

Software Dependencies

- Stellar-Supercluster (SSC) is written in F#
  - This is a .NET language, so requires .NET Core which is available on Linux, MacOS and Windows
  - Build in root dir with `dotnet build`
  - The package is split in two pieces, one C#, one F#, so you can add C# parts if you really hate F#, but I like it

Software Dependencies

- Uses several libraries, all auto-installed when you build
  - The C# Kubernetes client
  - The C# Stellar SDK
  - The FSharp.Data JSON and CSV data-access library
  - The Nett TOML library
  - The Serilog logging library
  - The CommandLineParser library

Runtime Dependencies

- SSC runs against a Kubernetes cluster <u>only</u>
  - You need a **`kubeconfig`** file that lets you talk to one
- It runs a Docker image on the cluster
  - You need one that contains **`stellar-core`**
  - The default is **`stellar/stellar-core`** which is ok if you are happy tracking master
- Of course, F#/C# means you need the **`dotnet`** runtime installed on the client you run SSC from

Aside: why F#?

- It's strongly-typed but type-inferred and relatively terse.
  - "Missions" (a.k.a. "recipes" in SCC a.k.a. "tests") can be written directly in it without much boilerplate; it's "terse as a DSL".
- Immutable-by-default, polymorphic, functional style: relatively high expressivity and safety.
- .NET ecosystem: lots of well-typed libraries, cross-platform, decent JIT'ed performance when generating load / ingesting / scanning large output.

Example mission

```
let simplePayment (context: MissionContext) =
    let coreSet = MakeLiveCoreSet "core" CoreSetOptions.Default
    context.Execute [coreSet] None (fun (formation: StellarFormation) ->
        formation.WaitUntilSynced [coreSet]
        formation.UpgradeProtocolToLatest [coreSet]

        formation.CreateAccount coreSet UserAlice
        formation.CreateAccount coreSet UserBob
        formation.Pay coreSet UserAlice UserBob
    )
```

Part 3:
Crash course in Kubernetes

Kubernetes Basics

- Resources: things client gets to talk about: get, put, list, patch
  - Volumes, Containers, Pods (groups-of-containers), Jobs, StatefulSets (groups-of-pods), Services, Ingresses ...
  - Each resource has a State and a Spec
  - Client can observe State, but can only modify Spec
- Controllers: server-side processes that modify resource States
  - DNS Controller, Ingress Controller, Job Controller, PersistentVolume Controller ...
  - Observe changes to resource Specs made by clients
  - Try to make State match Spec, if possible
    - Possibly instantiating more Resources, contacting other Controllers

Kubernetes Basics

- All resources have uniform YAML representation
    - `kubectl list` / `kubectl get` / `kubectl describe`
- Can dump to YAML file, edit, patch back into server
    - We do <u>not</u> do this except while debugging!
- Also exposed as uniform REST interface, mapped to strongly typed client SDKs for navigating & updating
    - We do this!

Kubernetes Basics

- SSC interaction pattern:
  - Connect to Kubernetes API server
  - Build client-side big pile of Resource Spec objects
  - Send REST requests to `create` all such Resources
  - Make long-poll `watch` requests until States match
  - Send HTTP commands to `stellar-core` to observe & inject traffic
    - Ingress Resource routes URL prefix to each container
  - Possibly `replace` Spec to reflect next test phase, re-`watch`

Kubernetes Unique Terms

- Pod: bundle of two sets-of-resources
  - Set of Containers running together on a node
  - Set of shared Volumes visible to the Containers
- Example (single Pod):
  - `stellar-core` container + `nginx` history-serving container
  - ConfigMap Volume + EmptyDir or EBS data Volume

Kubernetes Unique Terms

- StatefulSet: group of similar pods
  - PodTemplate that gets replicated for i in 0..N-1
  - Internal DNS names assigned
    - `$podname-$i.$service.$namespace.svc.cluster.local`
  - EBS volumes acquired from provisioner, attached
- Example (single StatefulSet):
  - Quorum of N `stellar-core` + `nginx` pods
  - Each configured to see each other in own `/cfg/$podname-$i.cfg`:
    - `PREFERRED_PEERS=["$podname-0.$service...", "$podname-1.service...", ...]`
    -
      ```
      [HISTORY.$podname-0]
      get="curl -sf http://$podname-0.$service.../{0} -o {1}"

      [HISTORY.$podname-1]
      get="curl -sf http://$podname-1.$service.../{0} -o {1}"
      ```

Kubernetes Namespaces

- All SSC Resources created in a Namespace
  - Umbrella Resource for holding other Resources
  - Access control, name isolation, resource quotas
  - SSC Resource names further randomly prefixed
- Namespace typically cleared pre-run by Jenkins
  - Get your own (ask admins) if you want isolated workspace for personal use!

Kubernetes Quotas

- RAM and CPU subject to Quotas
  - Admins set the quotas, we obey.
  - Namespace has overall Quotas; Containers have sub-quotas within.
- Quotas contain both a Request (guaranteed) and Limit (opportunistic) value. Must not allocate more of either than Quota allows, or will not be scheduled to run.
- Calculation of the "right" amount is complicated!
  - As much as possible to make room; not so much that Quota exceeded.
  - Depends on how many containers you run, how many SSC instances, etc.
  - SSC does these calculations, has many knobs to adjust.

Part 4:
How SSC uses Kubernetes

Resources Used

- Containers
  - To run `stellar-core`, `nginx`, `sqlite3`, maybe `postgresql`
- Volumes
  - ConfigMaps to store TOML .cfg files
  - EBS or EmptyDirs to store data (buckets, DB, history)
  - EBS Volumes use PersistentVolumeClaims dynamically provisioned by admin-configured "storage class"

Resources Used

- Pods and PodTemplates
  - Combine Volumes and Containers
  - Pod can have 1 or more Initialization commands, eg.
    `stellar-core new-db`
  - PodTemplate instantiated into multiple Pods by StatefulSet

Resources Used

- StatefulSets and Jobs
  - StatefulSet for group of stellar-core pods with ongoing overlay connections among themselves, eg. a quorum
  - Job for isolated run-then-exit commands such as parallel catchup tests that run `stellar-core catchup`

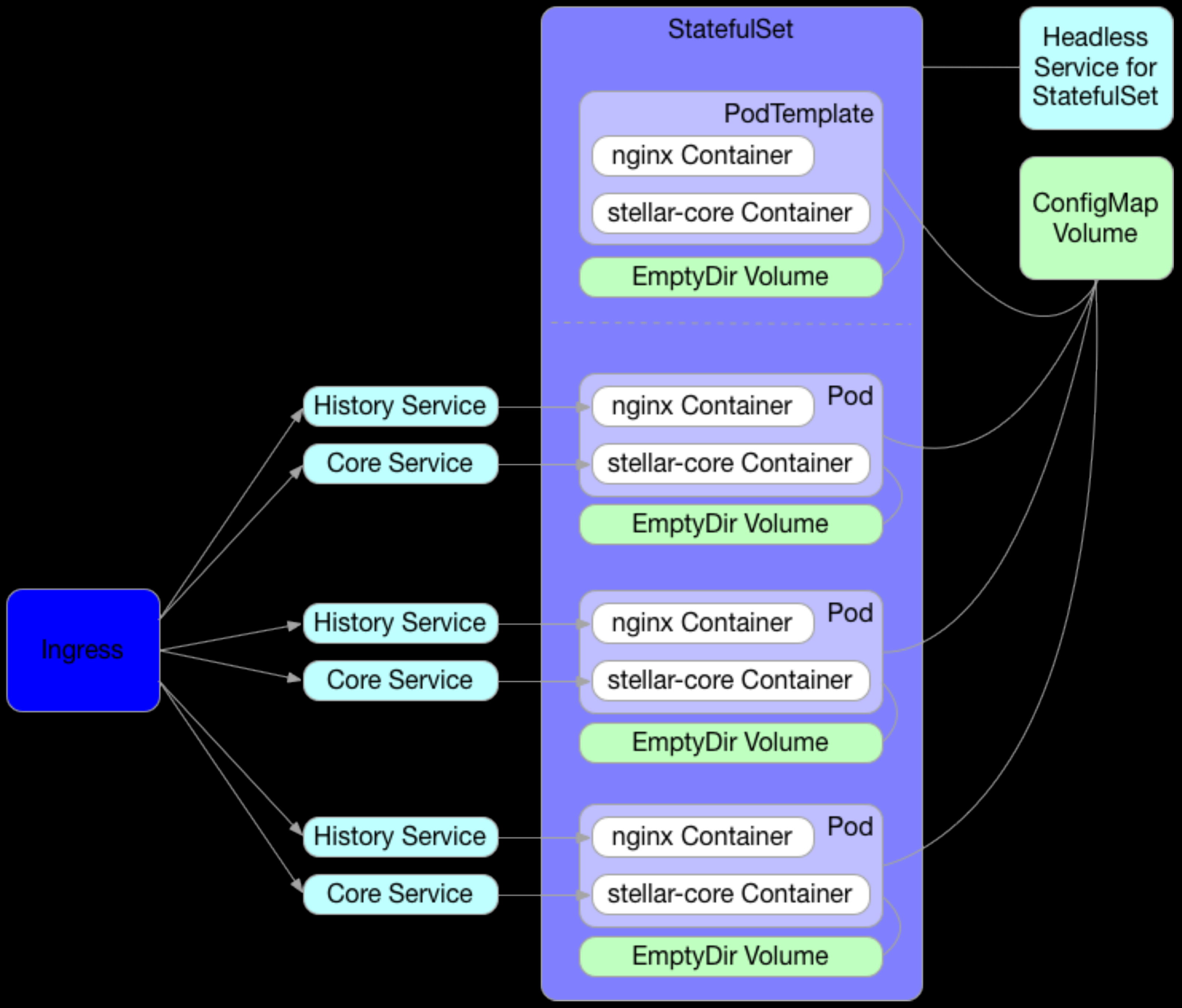
Resources Used

- Services
  - Specifies network port Ingress might route traffic to
  - Can be used for load balancing, but we do not
  - We make two Services per-Pod in each StatefulSet:
    - History Service to talk to the Pod's `nginx`
    - Core Service to talk to the Pod's `stellar-core`

Resources Used

- Ingress
  - Listens on port at cluster edge
  - Runs `traefik` "cloud-native edge router" (HTTP proxy)
  - Routes HTTP requests to (per-Pod) Services inside:
    - History: `http://$ingress/$peer-$i/history/...`
    - Core: `http://$ingress/$peer-$i/core/...`

Diagram Time!



Part 5:
How the SSC codebase is organized

Root directory

- `Dockerfile.dotnet`
  - Makes a build environment containing `dotnet`, stores result as Docker image
- `Dockerfile`
  - Runs `dotnet build StellarSupercluster.sln`, stores result as Docker image
- `StellarSupercluster.sln`
  - "Solution" file for `dotnet`, references sub-projects in `src/`
- `src/**`
  - The code
- `Jenkinsfile`
  - Groovy code, builds Docker images above, runs acceptance tests

src directory

- `App`
  - Command-line parsing and sub-command definitions
- `CSLibrary`
  - C# sub-project, currently an empty stub
- `FSLibrary`
  - F# sub-project, currently almost all of the code

FSLibrary directory

- `Mission*.fs`
  - Specific acceptance-test scenarios ("missions"), one per file.
- `Stellar*.fs`
  - Logic to write stellar-core configs, build Kubernetes Specs, watch Kubernetes States, build Stellar SDK transactions, speak HTTP to Ingress
- `{csv,json}-type-samples/**`
  - Sample JSON and CSV data from which static types are derived

Primary Types

- `StellarCoreCfg.fs (type StellarCoreCfg)`
  - Immutable config for <u>single</u> stellar-core node (with TOML writing)
- `StellarCoreSet.fs (type CoreSet)`
  - Immutable <u>set</u> of related stellar-core config values (eg. a quorum)
- `StellarNetworkCfg.fs (type NetworkCfg)`
  - Immutable <u>set of CoreSets</u> and collective network config (Quota, Ingress, Namespace, nonce, network passphrase)
- `StellarFormation.fs (type StellarFormation)`
  - Mutable <u>NetworkCfg</u> and Kubernetes objects

Secondary Types

- `StellarCorePeer.fs (type Peer)`
  - Handle to a specific member of a CoreSet within a NetworkCfg
- `StellarDestination.fs (type Destination)`
  - Client-side directory in which logs & data dumps written
- `StellarPerformanceReporter.fs (type PerformanceReporter)`
  - Accumulates sequence of metrics from Peer, writes CSV
- `StellarNamespaceContent.fs (type NamespaceContent)`
  - Tracks live Kubernetes Resources, deletes them on shutdown

Type Extensions

- `StellarCoreHTTP.fs (extends type Peer)`
    - Adds methods to speak HTTP to stellar-core through Ingress
- `StellarTransaction.fs (extends type Peer)`
    - Adds methods to compose XDR / Stellar SDK-level objects to send over HTTP
- `StellarDataDump.fs (extends type StellarFormation)`
    - Adds methods to extract logs, SQL DB, etc. to Destination
- `StellarJobExec.fs (extends type StellarFormation)`
    - Adds methods to run one-off Jobs (including in parallel)
- `StellarKubeSpecs.fs (extends type NetworkCfg)`
    - Adds methods to translate NetworkCfg content to Kubernetes Specs

Part 6:
How to run it and debug it

Running

- This part is a little clunky. Fundamentally just:
  - `dotnet run ...`
- But in practice, lots of args:
  - ```
    dotnet run --project src/App/App.fsproj
               --configuration Release
               --
               mission SomeMissionName
               --kubeconfig path/to/kubeconfig
               --destination path/to/destination
               --image stellar/stellar-core:latest
               --old-image stellar/stellar-core:$vers
    ```
    ```

Debugging

- Mainly you want:
    - `kubectl logs <pod-name>`
    - `kubectl (get|list|describe) <resource-type> <name>`
    - `kubectl (exec|cp|port-forward) ...`
- Also:
    - Watch Prometheus metrics
    - Browse to the Kubernetes dashboard (if you're allowed)
    - Run with `--keep-data` and inspect Resources, then manually delete with `kubectl delete --all pod,svc,sts,ing,pvc,cm,job`

Part 7:
How to extend it

Extensions: missions

- Add new missions:
  - Duplicate and modify `src/FSLibrary/Mission*.fs` file
- Add any parameters needed
  - Add to `src/App/Program.fs` arg-parsing code
  - Thread through to `src/FSLibrary/MissionContext.fs`
  - From there to wherever else they're needed
- Wire in to `Jenkinsfile`, adjusting `--num-concurrent-missions`

Extensions: Resources

- Customize existing Kubernetes Resources:
  - Edit Spec-forming code in `StellarKubeSpecs.fs`
- Add new Kubernetes Resources:
  - Add methods to make Specs in `StellarKubeSpecs.fs`
  - Create Resources in `Kubernetes.MakeFormation` extension method in `StellarSupercluster.fs`

Extensions: protocol

- Add HTTP endpoint calls to stellar-core:
  - Add methods to `StellarCoreHTTP.fs`
- Add XDR / SDK-typed interactions with stellar-core:
  - Add methods to `StellarTransaction.fs`

Fini