Two Ways to Extend the K8s API

Add resources to a Kubernetes API with TPR or AA

Eric Tune (github: @erictune, etune@google.com), Anirudh Ramanathan (github: @foxish, foxish@google.com)

Public

This document speaks to 4 audiences:

- *To extension developers, to explain that we have two different extension mechanisms, and how to chose which one to use for a specific task.*
- *To cluster administrator, to explain what they need to consider before they install an extension in their cluster.*
- *To Kubernetes API machinery developers, to encourage them to view what they are building from the perspective of those first two audiences.*
- *To the Kubernetes leadership, to convince them of the importance of investing in both AAes and TPRs*

It is written like User Documentation, except that it describes a desired future state. It is intended to complement, not replace, the existing proposals and issues on AA and TPR. I

Contents

Why Extend Kubernetes?

Developers may want to extend the API provided by their Kubernetes cluster. Here are some concrete use cases:

- **K-V Store**: I just need to store some key-value pairs in a REST API for a simple internal application.
- **SparkJobs**: When Spark Jobs are run on Kubernetes, they report their status via a SparkJob resource. CLIs and UIs can show this status without the need to actually compile Spark awareness into Kubernetes.
- **Custom Controller**: None of the built-in controllers do exactly what I need for my application. I write a MySpecialSet controller, and extend Kubernetes API to have a MySpecialSet resource, which I can use similarly to Deployment, etc.
- **Operator**: A Operator for etcd which exposes a Status and Spec for an etcd instance via a Kubernetes object. Anyone can quickly start using this controller from a single command (e.g. Helm Chart) without needing complex cluster setup.
- **PAAS**: I want to install a complete PAAS on top of Kubernetes, which is tightly integrated with Kubernetes, using all the same API conventions.

Two ways to Extend Kubernetes

The use cases above range from very simple (the first bullet) to very complex (the last bullet). Because a single mechanism can't cater to the full range of use cases, Kubernetes provides two different mechanisms to extend the API. They are **Third Party Resources (TPR)** and **User API Servers (AA)**.

TPRs require learning less and need not add "moving parts" to your cluster. AAes provide a deeper integration with Kubernetes, more features, and more options for dealing with performance, policy and customization. AAes require that you write some code in go language, and that you build, deploy and monitor a moderately complex piece of software.

For Developers: If your use case is simple, you may want to go directly to the detailed documentation on ThirdPartyResources. There is a relatively low cost to prototyping using TPRs, and then switching to AA later if you need more features. If your use case is complex, or if you are not sure, keep reading this document, and then, if you chose them, read the detailed documentation on Aggregated API Servers. Note: We use the term AA in this document to distinguish between the Core API Server, which serves e.g. /api/v1/pods, and a User-written API Server (AA). Both of these will sit behind a proxy which "aggregates" them.

For Cluster Administrators: You may be considering installing an extension on your Kubernetes cluster. The next section summarizes what it means to install a TPR or an AA on your cluster.

Before Installing Extensions on Your Cluster

Installing TPRs is safe and simple:

- A single command is needed to install a TPR on your cluster. TODO: xref detailed TPR docs.
- Installing a TPR just adds a rest endpoint to your apiserver. It does not cause any "third party code" to run on your apiserver.
- TPRs do not install any new "code" in your api server. Sometimes a TPR may have a corresponding controller that needs to be run. This is not covered here.
- TPRs consume storage space and API "requests per second" of your API server in the same way that ConfigMaps or Secrets do. This is typically not a concern, but watch out for extensions that encourage storing large amounts of data in your API server.
- Access to TPRs always uses the authentication (e.g. basic) and authorization system (e.g. RBAC) you have configured for your primary API server. The same audit log is used. You may need to add new RBAC roles with permissions for the new resources and bind users to the appropriate new roles, or edit your existing roles.

Installing AAes is significantly more complex than using TPRs:

- Installing an AA means taking on the need to monitor and upgrade a new thing.
- The AA itself is third-party code. Trust the source or understand what role it runs as and what permissions it requires when operating against your primary API server.
- Some AA require running their own storage backend, others use TPR as their storage.
- An AA may be written so that its resources use your primary authentication and authorization, (like TPR) or they may use their own. Consult the docs of the AA you are considering installing.
- An AA uses its own audit log.

Choosing between TPR and AA

Short Version

Chose TPR if:

- ease of building your extension is the most important factor.
- you control the client for this resource (its behavior and when it is updated)
- you are building a prototype and don't mind switching to AA later
- you just need a little storage and a way to surface status to kubernetes CLIs/UIs

Otherwise, choose AA.

Medium Version

TPRs give you:

- **Low Development Effort**.
  - Just write a handful of lines of YAML and the endpoint is installed.
  - No framework to learn.
  - No requirement to use go language.
  - Fewer concepts. All the advantages of AA (below) require some time to understand even if not needed.
- **Low Install Effort**: "kubectl create -f tprschema.yaml" and you are done.
- **Low Maintenance Effort**.
  - Cluster Admin doesn't need to monitor/maintain a new service with TPR.
  - Extension Dev does need to make a new software release if there is an API server bug fixed.
  - Cluster Admin does not need to watch for and install updates.

AA gives you:

- **Typed fields, validation, and defaults.**   These help users prevent errors and allow you to evolve your API independently of your clients. TPRs don't have these features However, if you control the client, then this may matter less. For example, Spark on Kubernetes uses a SparkJobs third-party resource. These are short lived and typically created programmatically by a single client, `spark-submit`. So, these features are less valuable.
- **Versioning**: In Kubernetes core objects, we have the ability to convert between versions. We also have the ability to add fields to existing objects without version changes. This is easier to incorporate into AA where you can specify custom ways to handle schema changes (mirroring an annotation to a field for compatibility, conversion functions, etc). Again, if you control all clients, you can handle some clients updates.
- **Schema**: a swagger-schema helps clients prevent errors, and allows generation of forms in UIs, and automatic generation of REST clients.
- **Custom Storage**: If you need storage with a different performance mode (e.g. time-series database instead of key-value store) or isolation for security (e.g. encryption secrets or different trust level for admins), then AA gives you the ability to configure different storage.
- **Custom Business Logic**: A AA can have custom business logic that runs synchronously with CRUD of objects (special authorization policy, multi-object transaction, setting defaults from other parts of cluster state, hooks on create or delete, etc). Some of these goals can be achieved in a more round-about way with a TPR (such as using a finalizer for a delete hook), but it will be asynchronous.
- **Subresources:** This lets meta-controllers like HorizontalPodAutoscaler and PodDisruptionBudget interact with your new resource. *(Support for subresources may be added to TPR in future though*)

Feature-by-Feature Comparison

The following two tables compare in detail the requirements to create and features that can be accomplished in TPR and AA.

Developer Requirements	**What does it mean?**	**Required for AA?**	**Required for TPR?**
Go	Must be able to program in Go language, and have a go build system for your project? May limit ability to use within other OSS projects or companies which are not open to Go language.	✓	
Vendor Server Code	The extension developer needs to vendor libraries, and copy an example APIserver main program from the Kubernetes project, then add her own code. This may incur future need to do a rebase and security fix of server. rebases or need for bug fixes.	✓	

Feature	**What it is for**	**Works with AA?**	**Works with TPR?**
CRUD	The new endpoints support CRUD basic operations via HTTP	✓	✓ (any bugs will be fixed)
Watch	The new endpoints support Kubernetes Watch operations via HTTP	✓	Yes (any bugs will be fixed)
Discovery	Clients like kubectl and dashboard automatically offer list, display, and field edit operations on your resources.	✓	✓
Custom printers (kubectl)	kubectl describe; human friendly -o modes	In the future. That code is moving to apiserver.	No (may support via webhook in the future)
json-patch	The new endpoints support PATCH with `Content-Type: application/json-patch+json`.	✓	✓
merge-patch	The new endpoints support PATCH with `Content-Type: application/merge-patch+json`.	✓	✓
strategic-merge-patch	The new endpoints support PATCH with `Content-Type: application/strategic-merge-patch+json`.	✓	No, cannot support without an IDL.

Validation	Gives human clients immediate feedback if they make a mistake (e.g. negative value in field that requires positive value)	✓	No (might support via webhook in the future, but it would be optional)
HTTPS	The new endpoints use HTTPS	✓	✓
Protocol Buffers	The endpoints can use Protocol Buffers	✓	No
Custom storage	Allows replacing the primary storage (etcd) with another storage that has different performance characteristics (e.g. slower but cheaper) or to provide storage isolation from the main (perhaps hosted) apiserver.	✓, today: run your own etcd or write your own thing Tomorrow: can somehow use core apiserver storage	No
Version in REST path	Can I specify a version in the path of the endpoint, like "v1".	✓	No, but this is a bug which we will fix.
Conversion	Can my server convert between two versions with different but convertible formats?	✓	No (may support via webhook in the future)
Built-in Authentication	Can the extension reuse the core apiserver's authentication?	✓	✓
Custom Authentication	Can the extension do its own authentication	Maybe. But why?	No
Built-in Authorization	Can I reuse the authorization scheme of the Core API server (e.g. RBAC).	✓	✓
Custom Authorization	Can the extension do its own authorization scheme in place of or in addition to the cluster's built-in one.	✓	No
Built-in Admission Control	Can I apply the core API server's admission control to my objects? Quota, etc.	Maybe. Being designed.	No. But… the changes being considered for AA may effectively make this "yes".
Custom Admission Control	Can I write my own "admission controllers"?	✓	No
Finalizers	Can I have resource deletion block on another client doing something.	✓	✓
Non-admin Install	Can a non-"root" user install an extension?	Depends. On some clusters, admins may trust users to install AAs. On clusters where namespaces are used for multi tenancy (e.g. OpenShift), admin may not want to allow tenants to install AAs, but might take request.	Depends. On some clusters, admins may trust users to install TPRs. On clusters where namespaces are used for multi tenancy (e.g. OpenShift), admin may not want to allow tenants to install but might take requests.
UI/CLI Display	Kubectl, dashboard can display objects of your new type, without modification.	✓, that is the plan.	✓, that is the plan.
UI/CLI Guided Creation	Kubectl, dashboard can provide views to guide you in creating objects of your new type, without modification to the CLI	✓, that is possible in principle since there is OpenAPI spec.	No, but a schema is a potential future addition to TPR, althought that would loose some of the ease of use, since

	code.		users have to write in an IDL and generate OpenAPI.
Unset vs Empty	Can clients distinguish unset fields from set to 0 or empty-string fields.	✓	✓
OpenAPI Schema	Is there an OpenAPI (swagger) schema for the types that can be dynamically fetched from the server? Is the user protected from misspelling field names by ensuring only allowed fields are set? Are types enforced (e.g. don't put an int in a string field?)	✓	No, but a schema is a potential future addition to TPR
Works with Hosted master	Works with a master that even the cluster administrator cannot access (e.g. GKE)	✓. There is a plan in the proposal for it to work.	✓
Handles nested JSON and types (numeric string, array)		✓	✓
Subresources?	Are subresources, such as /scale supported? This is needed in order for PodDisruptionBudgets to manage your custom controller, or to allow kubectl scale to scale your custom controller, or for a dashboard to offer a "scale" dialog for your objects, or for a HPA to scale your custom controller.	✓	No, but it seems clear we need to add this to TPR.
Object Count Quota	As the cluster admin, can I prevent a client from making so many objects of the new kind that it fills up my storage.	Yes-ish. You can write your own Quota policy. And the storage can be isolated. But there is not a simple way to extend the ResourceQuota object to have object-count-quota on AA objects.	No, but adding would not harm TPR ease of use, so we should add to Quota.

Building for a Complete Extension

TODO: expand this section

1. Define a resource
   a. TODO: links to API convention docs, and say what parts are appropriate for TPR.
2. Generate a client for the resource
   a. Simplest: CURL
   b. Use the dynamic client.
   c. Use gengo to generate a client, and release it and so on.
   d. Use our go client generator
   e. Use open-api plus generator of your choice.
3. Write a controller for the resource

    a. Patterns: single controller per cluster vs controller per resource. What are the tradeoffs
4. Integration with common CLIs and UIs
    a. What can I do with TPR to make Kubectl and Dashboard show/edit my resource.
    b. Kubectl plugins
5. Diagrams.

API Endpoint Format

Neither AA nor TPR are intended to allow a user to create arbitrary APIs. If that is what you need, choose another tool.
There are certain common restrictions for both:

- Arbitrary resource paths are not possible Paths follow certain patterns, such as:
    - `/apis/<API_GROUP>/<VERSION>/namespace/<NS>/<KIND>/name`
    - This allows automatic application of namespacing and authorization policy to objects.
- Objects have Kubernetes metadata.

For example: /apis/example.com/v1/mynewkind/

Status

Aggregated API Servers are currently under active development with first version expected in 1.6 or 1.7.

ThirdPartyResources (TPRs) have been available in Kubernetes since 1.0 (?).   However, there are proposals to rewrite the implementation to harden it, and to change the API in breaking ways.

Plea

If you are a developer, let us know if this doc would help you decide which to use.

If you are a kubernetes leader, let us know that you are convinced that neither AA or TPR alone is sufficient.

- TPR is not flexible enough and does not offer enough client assistance.
- The bar for creating and maintaining an AA is too high to impose on all users that want to extend Kubernetes.

    - For example, an expert in a particular database technology should be able to write an operator for that database in her language of choice (for which there might not be any kubernetes client) and use TPR for storage and UI integration.
    - The go language requirement is onerous. For example, the Spark project is in scala and is not going to host go code (a SparkJob AA) just to do kubernetes integration.

No single solution can meet all needs. So, we need both.

Update: it was decided that we will pursue both tracks. See https://docs.google.com/document/d/1lU1SnVtEec2ilfYx5U3L5N0za2YhfEOik0uPen276Ks/edit#

Our opinion:

1. We (the kubernetes developers) should stay the course on AA.
2. We should harden TPR against obvious crashes and breakages and fix many of the known issues, some of which may require breaking API changes.
3. We should ensure that a user can start by implementing a type using TPR, and later upgrade to AA, and her clients should not be aware of the change (but not necessarily the other way).
    a. Users looking to experiment with extensibility in Kubernetes should not have to start by writing and rolling out a complex AA. TPR should serve as a starting point and chart a path to eventually having an AA supporting one’s resource type if there is a need for the advanced features provided by it.
4. We should keep TPR focused on ease of learning how to use it (to extend K8s), consistent with previous goals.