Knative build for Apache OpenWhisk Runtimes

Plans for allowing OpenWhisk Functions to built so they may run on Knatve Serving

Matt Rutkowski, IBM, STSM Open Technologies, DEG
Priti Desai, IBM, Open Source Developer, DEG

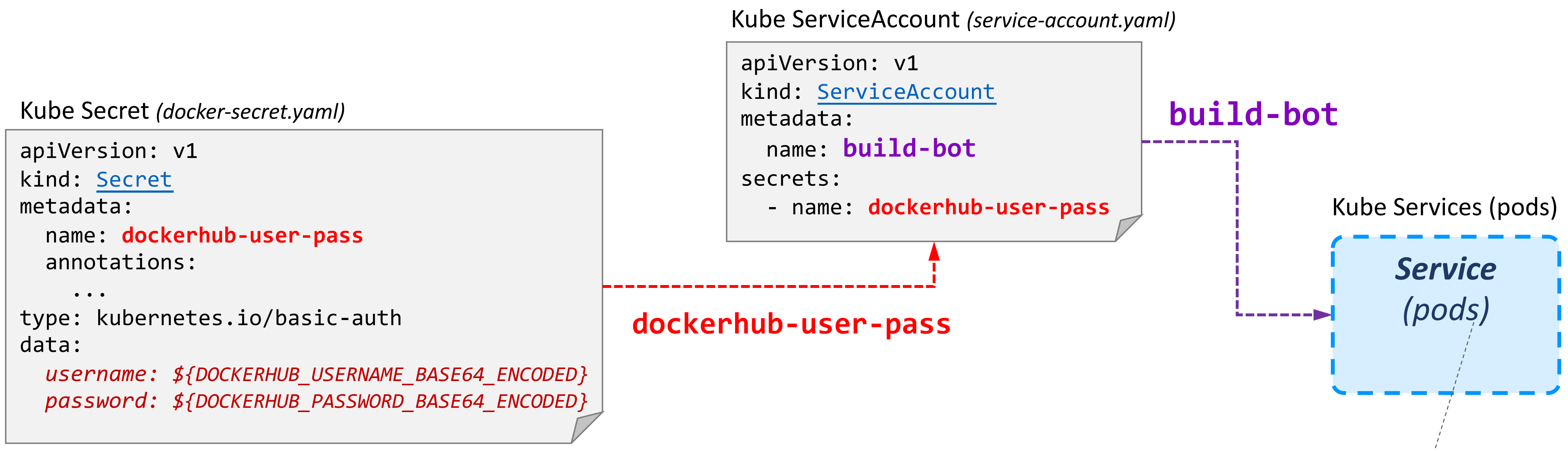
Mar 14, 2019

Goals

- Allow Apache OpenWhisk Actions (i.e., functions) to run on Kubernetes via Knative build methods
    - Initially target the Apache OpenWhisk NodeJS runtime (as it is the most popular and shows AWS Lambda equivalency)
    - Apply methodology to ActionLoop (Go proxy) and support most remaining
    - Apply methodology to Java Runtime
    - Explore NodeJS, Java runtimes using ActionLoop (some prototypes started for NodeJS)
- References
    - User experience:
        - Sample: https://github.com/knative/docs/tree/master/serving/samples/source-to-url-go
        - But not using the (google) kaniko build template (which performs a build-deploy as one step in a Kaniko container image)
- Results/Claims:
    - "Seamless function deployment using Knative (service) or native OpenWhisk API ("wsk") [or Lambda]
        - Akin to "TriggerMesh" announce for cross-running Lambda functions on Kube using Knative
            - https://hub.packtpub.com/triggermesh-announces-open-source-knative-lambda-runtime-aws-lambda-functions-can-now-be-deployed-on-knative/
            - https://www.zdnet.com/article/triggermesh-brings-aws-lambda-serverless-computing-to-kubernetes/
        - *TriggerMesh Lambda runtime: https://github.com/triggermesh/knative-lambda-runtime*
            - *Example: Python 3: https://github.com/triggermesh/knative-lambda-runtime/blob/master/python-3.7/buildtemplate.yaml*
    - *But instead "run OpenWhisk Action functions on Kube"*
- *Identify Tooling Needs/Options:*
    - *Adopt CLI support for Knative targets (once we understand differences in invocation model)*
        - *E.g., adopt --knative flag on "wsk" CLI*
        - *Whisk deploy: use in runtime?*

Orchestrating a source-to-URL deployment on Kubernetes

***Pre-Req. :** Create Kube Service Account and Kube Secret for Docker Hub used in "Push" of image built by Knative*



Kube Secret *(docker-secret.yaml)*

```
apiVersion: v1
kind: Secret
metadata:
  name: dockerhub-user-pass
  annotations:
    ...
type: kubernetes.io/basic-auth
data:
  username: ${DOCKERHUB_USERNAME_BASE64_ENCODED}
  password: ${DOCKERHUB_PASSWORD_BASE64_ENCODED}
```

Kube ServiceAccount *(service-account.yaml)*

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-bot
secrets:
  - name: dockerhub-user-pass
```

```
$ kubectl apply --filename docker-secret.yaml
secret "basic-user-pass" created
$ kubectl apply --filename service-account.yaml
serviceaccount "build-bot" created
```

Service accounts can be referenced by one or more Kubernetes Services (and their respective pods), <or> specifically in our case by Knative Services

A [Service Account](#) provides an identity for processes that run in a Pod.

https://github.com/knative/docs/tree/master/serving/samples/source-to-url-go

Orchestrating a source-to-URL deployment on Kubernetes

***Pre-Req. :** Create Knative Service Account and Kube Secret for Docker Hub used in "Push" of image built by Knative*

Knative Build Template *(kaniko.yaml)*

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: kaniko
spec:
  parameters:
  - name: IMAGE
    description: image to push to Docker
  - name: DOCKERFILE
    description: Dockerfile to build
    default: /workspace/Dockerfile

  steps:
  - name: build-and-push
    image: gcr.io/kaniko-project/executor
    args:
    - --dockerfile=${DOCKERFILE}
    - --destination=${IMAGE}
```

Notes:

- **Build Template:**
  - *The builder image for **Kaniko** is:* gcr.io/kaniko-project/executor
    - *In the Google Cloud Registry (gcr)*
  - *Parameters:*
    - *IMAGE: target image name the builder will push to DockerHub*
    - *DOCKERFILE: the path within the "builder image" to use to find the runtime (applicaton) 'Dockerfile' to build*
    - *NOTE: in the samples used in Knative, the DOCKERFILE parameter is always defaulted to:*
      - `/workspace/Dockerfile`

```
$ kubectl apply --filename https://raw.githubusercontent.com/knative/build-templates/master/kaniko/kaniko.yaml
```

https://github.com/knative/docs/tree/master/serving/samples/source-to-url-go

Orchestrating a source-to-URL deployment on Kubernetes

- *Build (Knative Build) and deploy (Knative Serving) the Service using the Kaniko*

Kube ServiceAccount

```
build-bot
secrets:
 - dockerhub-user-pass
```

Knative Build Template *(kaniko)*

```
kaniko
spec:
  parameters:
  - IMAGE
  - DOCKERFILE
```

Note: In this example, the Build Configuration (template) integrated into Service resource

Knative Service Template using the Kaniko build template *(kaniko.yaml)*

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: app-from-source
  namespace: default
spec:
  runLatest:
    configuration:          Build Configuration
      build:
        apiVersion: build.knative.dev/v1alpha1
        kind: Build
        spec:
          serviceAccountName: build-bot
          source:
            git:
              url: https://github.com/mchmarny/simple-app.git
              revision: master
          template:
            name: kaniko
            arguments:
              - name: IMAGE
                value: docker.io/{DOCKER_USERNAME}/app-from-source:latest
      revisionTemplate:
        spec:
          container:
            image: docker.io/{DOCKER_USERNAME}/app-from-source:latest
            imagePullPolicy: Always
            env:
              - name: SIMPLE_MSG
                value: "Hello from the sample app!"
```

```
$ kubectl apply -f service.yaml
service "app-from-source" created
```

Notes:

- **Service:**
  - **runtlatest:** defines how to build and run a Knative service using the "latest" tagged revisions to/from DockerHub
- **Build Configuration**:
  - ***serviceAccountName*** *value is defaulted to* ***build-bot*** *which provides access to the Secret (basic-user-pass) for DockerHub "push" of* ***IMAGE***
  - ***source****:*
    - ***git*** *url is the "app" location (i.e., the Dockerfile) used to build the "app" (or function) into the target runtime* ***IMAGE*** *(Knative service runtime)*
  - ***template***
    - *Knative Build Template set to* ***Kaniko***
    - ***Kaniko IMAGE*** *argument's value includes* DOCKER_USERNAME which should be the username (unencoded) that matches that within the basic-user-pass Secret
  - ***SIMPLE_MSG*** *is an Environment variable placed in the application's container runtime environment*

https://github.com/knative/docs/tree/master/serving/samples/source-to-url-go

TriggerMesh: Modifying Kaniko to support their own Runtimes *(proxy and functions)*

Creating a Build Template that reuses the Kaniko build image….

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: knative-python37-runtime
spec:
  parameters:
  - name: IMAGE # Used by Kaniko
  - name: TAG # used by Kaniko
  - name: DIRECTORY
    description: The subdirectory of the workspace/repo
    default: ""
  - name: HANDLER
    default: "function.handler" # See AWS Lamda docs
  steps:
  - name: dockerfile
    image: gcr.io/kaniko-project/executor@<<commit hash>>>
    command:
    - /busybox/sh
    args:
    - -c
    - |
      cd /workspace/${DIRECTORY}
      cat <<EOF > Dockerfile
        FROM gcr.io/triggermesh/knative-lambda-python37
        ENV _HANDLER "${HANDLER}"
        COPY . .
        ENTRYPOINT ["/opt/aws-custom-runtime"]
      EOF
  - name: export
    image: gcr.io/kaniko-project/executor@<<commit hash>>>
    args:
    - --context=/workspace/${DIRECTORY}
    - --dockerfile=/workspace/${DIRECTORY}/Dockerfile
    - --destination=${IMAGE}:${TAG}
```

TriggerMesh's Python runtime example

Dynamically create the Dockerfile for the Serverless "app"

Notes:

- **BuildTemplate** (triggermesh/knative-lambda-runtime)
  - Spec:
    - Define add. Parameters to find the function (source file) within GitHub.
      - DIRECTORY:: *subdir to function's source file*
      - HANDLER: *the function's source file*
        - TM examples use Serverless.com's samples for functions: https://github.com/serverless/examples/blob/master/aws-python-simple-http-endpoint/handler.py
    - IMAGE: same as Kaniko IMAGE (passthrough)
  - steps:
    - "dockerfile" (this is an "atypical builder"
      - Reuse the **Kaniko** "executor" image
      - Execute a CMD which creates a new Dockerfile which will copy the function source file (i.e., HANDLER) into the build image's ENV (environment)
      - ENTRYPOINT: is an AWS Custom Runtime convention
    - "export"
      - The **Kaniko** "executor" image is provided a new "Build Context" (i.e., --context) which includes the TriggerMesh Python runtime along with a copy of the **HANDLER** function it will run

```
kind: BuildTemplate
metadata:
  name: kaniko
spec:
  parameters:
  - name: IMAGE
  - name: DOCKERFILE
  steps:
  - name: build-and-push
    image: gcr.io/kaniko-project/executor
    args:
    - --dockerfile=${DOCKERFILE}
    - --destination=${IMAGE}
```

https://github.com/triggermesh/knative-lambda-runtime/tree/master/python-3.7

Phase 1:
Single-stage Build of OW Runtimes using Kaniko

Understanding the Kube/Knative vs. OpenWhisk Developer approach

Knative devs. are Container dev(ops) people moving "up-the-stack" vs. Serverless app. developers (No-Ops, No Stack)

- General philosophical differences
  - **Workload granularity**: Container vs. Function
  - **Invocation**: Single Runtime entry-point (application root as "/run", function "baked in") vs. /init and /run (reuse & functionally aware)
  - **Parameters**: as Container Environment Vars. (i.e., environment aware) vs. made avail. As JSON (agnostic of environment)
- OpenWhisk
  - **Build**: Runtimes are language-specific, **reusable "stem cell" images** managed by Control plan via dist. Invoker pools
    - *No "build" concept for Serverless Developers (apart from SDK), focus is on function with a programming*
  - **Serve**: from known pools of Runtime containers (Docker images) and compliant Docker images (e.g., Docker SDK), via CLI
  - **Code**: Function code "injected" into compatible runtimes (*Endpoints exposed via API Gateway service/integration)*
  - **Execution**: Activations caused by CLI or Event-Triggers;
  - **Parameters** Set on a per-invocation basis CLI or (Event) Trigger
- Knative
  - **Build**: Serverless functions built into **dedicated "Application" images** (Services); treated like any other Kube managed Service (image)
    - "Kube compliant" Build steps that "pull" from sources *(e.g., GitHub, S3, filesystem), "push" to a "registry" (e.g., DockerHub or GCR).*
    - *Knative utilizes "**BuildTemplates**" that compatible "**Builder**" (Docker images) use to perform all steps from source (pull) to target (push)*
  - **Serve:** **"**Pull" and deploy a Knative compliant (TBD?) Docker image to a Kube pod with Knative configuration (scale to zero, etc.)
  - **Execution:** Activations by Kube generated Endpoint (domain); accessible via Http(s) (e.g., curl)
    - ***Single endpoint (domain assigned):*** *Functional code is "baked in" to runtime image (part of Knative build)*
    - ***Kube Controlled Scaling:*** *Developer must be aware; Configuration options set by Developer on deployment*
      - *default pod has 3 instances started, scale to zero after inactivity; "wake" via a Knative proxy that detects new invocation*
  - **Parameters**: Set on Service deployment of via Knative Service YAML (i.e., "baked" into Container Env. Vars.), or by Knative Eventing

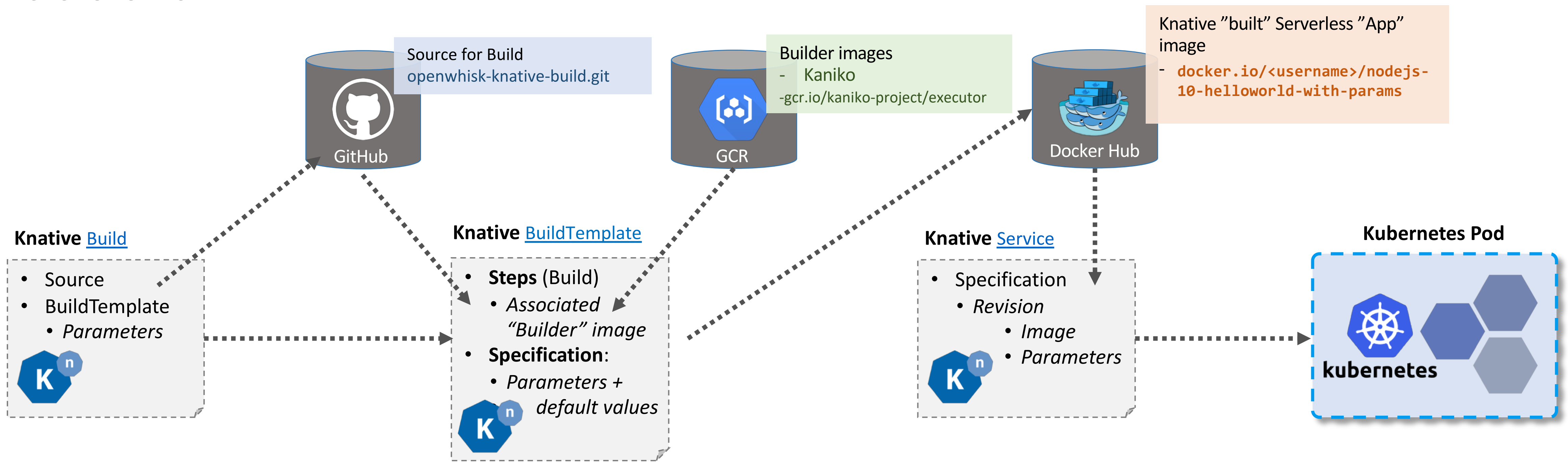
Phase 1: Use Knative to "Build" & "Serve" the NodeJS10 runtime with dedicated Actions

Kube/Knative communities all assume "Container" workload granularity; Functions get "baked" into dedicated containers

- Goals
  - Identify, the least invasive set of changes needed to allow our OpenWhisk runtimes to run with Knative (serving)
  - Identify a definitive set of use cases/scenarios (I.e., Action functions) that can seamlessly work in under Knative or OpenWhisk platforms
  - Utilize Knative Build (YAML) to build an OpenWhisk runtime <or> a Knative runtime and make the runtime (proxy) aware of the environment (host) to execute different logical code paths where needed
    - Exclude unneeded code for whichever is NOT the target platform)
    - Seek maximum code reuse, minimize unique code.
- NodeJS considerations
  - By far, the majority of Serverless functions are impl. In NodeJS compat. Javascript.
  - Ability to showcase (demo/blog) Apache OW runtimes working on Knative/Kube (or even AWS Lambda) seamlessly
  - Get the Knative (Kube) communities to pay attention to OpenWhisk's capabilities/knowledgebase/models/tooling
- Next Steps
  - Submit PR for review/comment ~1 week's time
  - *Complete 2-stage build to allow separate creation of a "stem cell" container* (i.e., separate Dockerfile and Knative build templates)… challenge the notion of Container granularity
  - Carry over knowledge and work against ActionLoop-based runtimes (Go proxy) and see if NodeJS an option there as well
  - Showcase interesting scenarios that we could advantage with a Knative inclusive build pipeline (e.g., retail/debug builds)
    - Note: if all our runtimes can resolve to a single proxy, we might leverage Knative build in many fascinating ways

Build an OpenWhisk Runtime compatible with Knative:

High-level Overview



"Build Configuration"

- **<u>Source</u>**: points to the "source" code (with Dockerfile) as starting point for the Build workspace
- e.g., **`openwhisk-nodejs-runtime`**

- **<u>BuildTemplate</u>**: named build instruction set to follow
  - *Parameters defined in the BuildTemplate's spec.*
  - e.g., **`openwhisk-nodejs-runtime`**

"Build Instructions"

- **<u>Specification</u>**: Parameters (and optional default values) provided to Builder images during build Steps.
- **<u>(Build) Steps</u>**: ordered steps executed by associated Docker "Builder" images (with parameters) against build workspace *<or> "atypically", direct commands to execute*
  - e.g., Kaniko (builder) image
  - ***TARGET****: Final build step is typically a Target (repo.) to place named image in.*
  - *E.g.,* **`docker.io/<username>/nodejs-10-helloworld-with-params`**

Service "Configuration" (Se and Run)

- **<u>Revision</u>**: describes a specific Container and configuration to run, including
  - **<u>Container</u>**
    - Image (name : tag)
    - Parameters (provided as Container Env. Vars.)
- e.g., **`docker.io/<username>/nodejs-10-helloworld-with-params`**

Build an OpenWhisk Runtime compatible with Knative: **BuildTemplate**

Create a Knative Build Template that can build OpenWhisk's NodeJS10 runtime with a function

- *Reusing the **Kaniko** "builder" image and its **Parameters** (i.e., "gcr.io/kaniko-project/executor:latest")*
- *Add Build Parms. for target platform ["openwhisk", "knative"], or Debug enabled/disabled (i.e., "retail" builds)*
- *Provide OpenWhisk **runtime "/init" data** as build parameters, placing them into the **Container as Env. Vars***

Build Template for building Modified OpenWhisk NodeJS10 Runtime

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: openwhisk-nodejs-runtime
spec:
  parameters:
  - name: TARGET_IMAGE_NAME
    description: name of the image to be tagged and pushed
  - name: TARGET_IMAGE_TAG
    description: tag the image before pushing
    default: "latest"
  - name: DOCKERFILE
    description: name of the dockerfile
  - name: OW_RUNTIME_DEBUG
    default: "false"
  - name: OW_RUNTIME_PLATFORM
    description: flag to indicate the platform, one of ["openwhisk", "knative", ... ]
    default: "knative"
  - name: OW_HTTP_METHODS
    default: "[POST]"
1 - name: OW_ACTION_CODE
    description: JavaScript source code to be evaluated
  - name: OW_ACTION_MAIN
    description: name of the function (handler) in the "__OW_ACTION_CODE" block
    default: "main"
  - name: OW_ACTION_BINARY
    description: flag to indicate zip function
    default: "false"
```

```
steps:
  - name: add-ow-env-to-dockerfile
2   image: "gcr.io/kaniko-project/executor:debug"
    command:
    - /busybox/sh
    args:
    - -c
    - |
      cat <<EOF >> ${DOCKERFILE}
        ENV __OW_RUNTIME_DEBUG "${OW_RUNTIME_DEBUG}"
        ENV __OW_RUNTIME_PLATFORM "${OW_RUNTIME_PLATFORM}"
        ENV __OW_HTTP_METHODS "${OW_HTTP_METHODS}"
        ENV __OW_ACTION_CODE "${OW_ACTION_CODE}"
        ENV __OW_ACTION_MAIN "${OW_ACTION_MAIN}"
        ENV __OW_ACTION_BINARY "${OW_ACTION_BINARY}"
      EOF
3   name: build-openwhisk-nodejs-runtime
    image: "gcr.io/kaniko-project/executor:latest"
    args: ["--destination=${TARGET_IMAGE_NAME}:${TARGET_IMAGE_TAG}", \
           "--dockerfile=${DOCKERFILE}"]
```

1. *The Action "name" from /init's data is not passed as it is not used by the function code*
2. *Kaniko "debug" image provides access to Bash allowing us a means to alter the Dockerfile for OW runtime __OW_xxx env. Vars.*
   - *Note: this could be ANY image with a shell and Docker…*
3. *We reuse the Kaniko executor for the final build (and push)*

https://github.com/apache/incubator-openwhisk-devtools/tree/master/knative-build

Build an OpenWhisk Runtime compatible with Knative: **Build**

Use Knative to Build your Serverless container with "Action" (function) and parms. "baked in" from the Build Template

- *This is how we tell the Builder image (reusing Kaniko executor for now) to build our Serverless "app"*

Build (configuration) file for the OpenWhisk NodeJS10 Build Template *with Action code "passed in" (i.e., "**Hello World with Parameters**")*

```
apiVersion: build.knative.dev/v1alpha1
kind: Build
metadata:
  name: nodejs-10-helloworld-with-params
spec:
  serviceAccountName: openwhisk-runtime-builder
  source:
    git:
      url: "https://github.com/mrutkows/openwhisk-knative-build.git"
      revision: "master"
  template:
    name: openwhisk-nodejs-runtime
    arguments:
      - name: TARGET_IMAGE_NAME
        value: "docker.io/${DOCKER_USERNAME}/nodejs-10-helloworld-with-params"
      - name: DOCKERFILE
        value: "./runtimes/javascript/Dockerfile"
      - name: OW_RUNTIME_DEBUG
        value: "true”
      - name: OW_HTTP_METHODS
        value: "[GET]"
      - name: OW_ACTION_NAME
        value: "nodejs-helloworld-with-params"
      - name: OW_ACTION_CODE
        value: "function main() {return {payload: 'Hello ' + process.env.NAME + \
               ' from ' + process.env.PLACE + '!'};}"
```

https://github.com/apache/incubator-openwhisk-devtools/tree/master/knative-build

- *Source: points (currently) to our private repo. which has a minimally modified version of the Apache OpenWhisk NodeJS10 runtime*
- *TARGET_IMAGE_NAME - configure to target DockerHub account where final "Serverless" image will be pushed*
- *DOCKERFILE – configure to tell builder where to find Dockerfile within the workspace to start the build.*
- *OW_RUNTIME_DEBUG – build in DEBUG trace (non-retail build)*
- *OW_HTTP_METHODS –Http Methods supported by the runtime (function); i.e., POST (default), GET, PUT, DELETE*
- *OW_ACTION_NAME –*
  - *If present as Env. Var., the runtime will find name here and use (in Activation data) as the default function name.*
    - *if not overridden by a name supplied in Activation data*
- *OW_ACTION_CODE – … (the code of course)*
  - ***Note**: this is analog to the "Handler" in Lambda; where we could alter the "build" in the future to put things where AWS Lambda expects them…*
  - *2-stage builds (future) could support "pulling" code from GitHub or S3 (i.e., other sources).*

Build an OpenWhisk Runtime compatible with Knative: **Serve**

Use Knative to to serve your "built" Knative "Serverless" image from

- *This is how we tell Knative to deploy our image (i.e., where to pull from and what Env. Vars. to set)*

Service (configuration) file used to deploy our "Serverless app" image
(i.e., OW NodeJS10 + Action code)

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: nodejs-helloworld-with-params
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/${DOCKER_USERNAME}/nodejs-10-helloworld-with-params
            env:
            - name: NAME
              value: Bob
            - name: PLACE
              value: Italy
```

- *container:*
  - *points (currently) to our private repo. which has a the "Serverless app" (i.e., "Hello World with parms.) created during the Build step.*
- *env:*
  - *Lists the names/values passed to the Container process' environment vars. (where the "Serverless app" image is executed).*
    - *process.env.NAME and process.env.PLACE*
  - *Note: at this point we need a discussion of "reuse" value as we will end up with a "pod " of the same function that you must invoke again by a known endpoint (pod).*

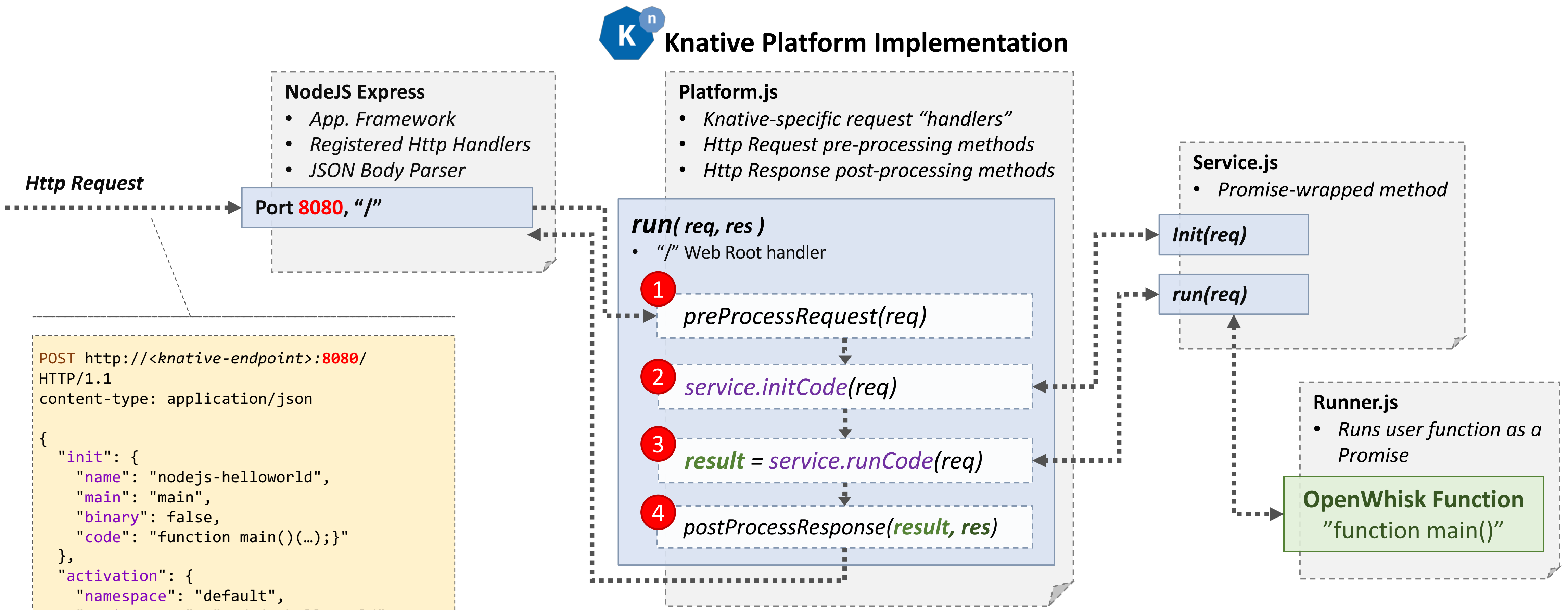
https://github.com/apache/incubator-openwhisk-devtools/tree/master/knative-build

Phase 1: OW Runtime as a Knative Service

What can we do without a Controller/Invoker Against Existing Use Cases?

Knative Platform Impl. – Overview of Request/Response Processing

Pre/Post-processing of Http requests/responses when built with `OW_RUNTIME_PLATFORM`=“`knative`”



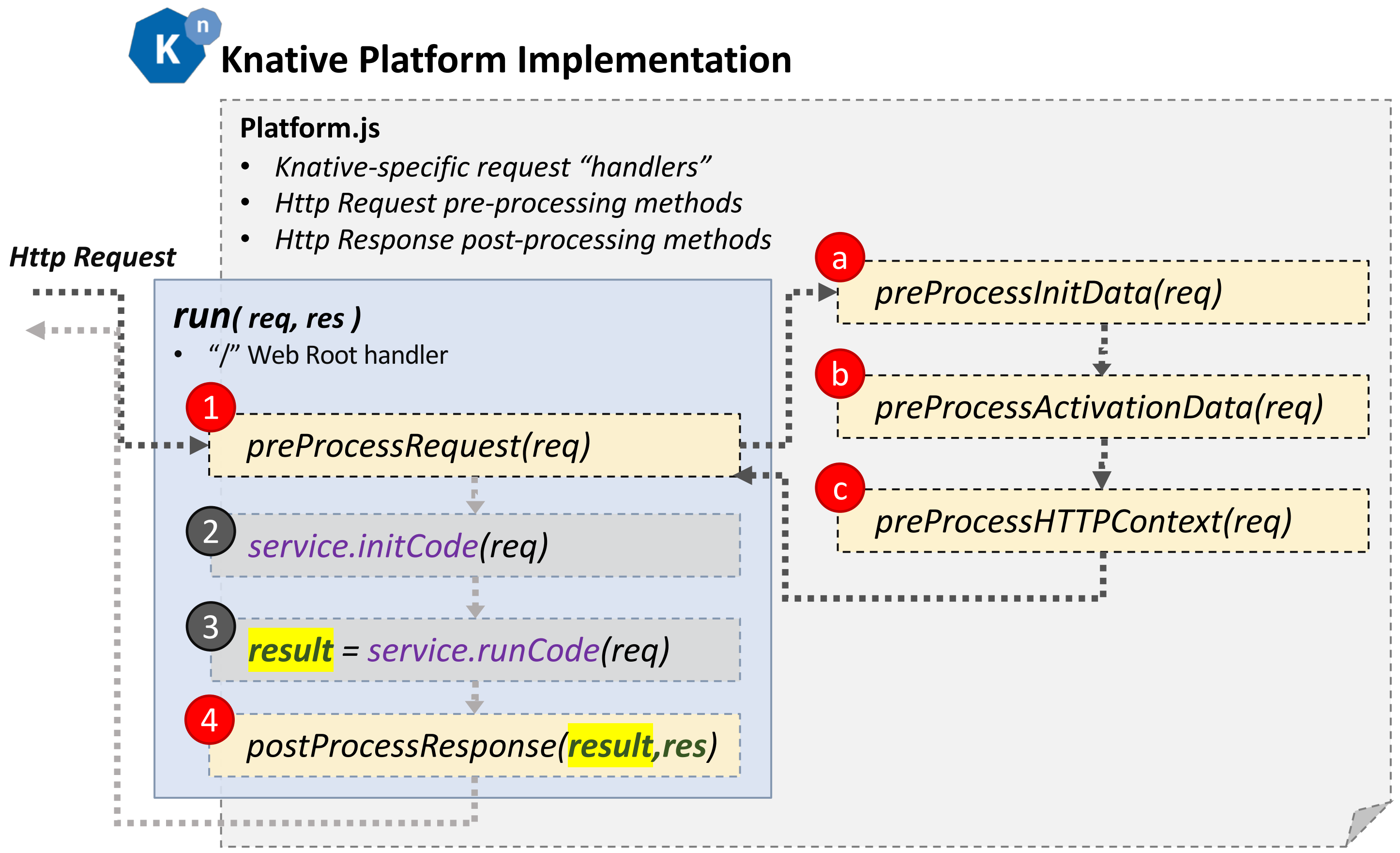
```
POST http://<knative-endpoint>:8080/
HTTP/1.1
content-type: application/json

{
  "init": {
    "name": "nodejs-helloworld",
    "main": "main",
    "binary": false,
    "code": "function main()(…);}"
  },
  "activation": {
    "namespace": "default",
    "action_name": "nodejs-helloworld",
    "activation_id": "",
    "deadline": "4102498800000”
     …
  },
  "value": {
    "name": "Joe",
    "place": "TX"
  }
}
```

- *Reuse existing OpenWhisk handlers service.initCode() and runCode() “as is”*

https://github.com/apache/incubator-openwhisk-devtools/tree/master/knative-build

Knative Platform Impl. – Pre/Post-processing Details

- Runtime's single "/" Entrypoint performs several functions that the Controller normally would provide



Knative Platform Implementation
Platform.js
Knative-specific request "handlers"
Http Request pre-processing methods
Http Response post-processing methods
Http Request
run(req, res)
"/" Web Root handler
1 preProcessRequest(req)
2 service.initCode(req)
3 result = service.runCode(req)
4 postProcessResponse(result,res)
a preProcessInitData(req)
b preProcessActivationData(req)
c preProcessHTTPContext(req)

a *preProcessInitData(req)*

- **IF**: OW Init. data is "baked in" to runtime use it on the original OW *init*() function
  - i.e., __OW_ACTION_xxx is in process environment
- **ELSE**: look in the request body for JSON init. Data and use that instead on the original OW *init*() function.
- **Note**: **__OW_ACTION_NAME** moved to Activation Data ONLY if it does not already contain a valid value for Action Name.
- **Note**: Normal init() processing will error if "code" is baked in and also is supplied in init. Data.

b *preProcessActivationData(req)*

- Move all keys/values in Activation Data to process environment variables for the function to access
  - i.e., Uppercase key name and prepend with "__OW_"

b *preProcessHTTPContext(req)*

- Move request context information to process environment variables prepended with "__OW_"
  - i.e., METHOD, HEADERS, PATH, NAMESPACE, USER, BODY (Base64 encoded), and QUERY

4 *postProcessResponse(result, res)*

- Move/format function's Http-related JSON data (i.e., **result**) to actual Http response protocol format
  - Move **result .statusCode** (e.g., 200, etc.) Http Response header
  - Move **result .headers** to Http Response header
  - Move **result .body** to Http Response body
  - Delete any OpenWhisk values from Http Response body

Knative Platform Impl. – Functional view of capabilities

What we can/cannot do within the runtime to provide OW functional equivalency under Knative/Kubernetes

OW Function Class	**OpenWhisk Capability** *(Supported via Native OW Platformw with Controller+Runtime)*	**Supported** *(Knative-Built Runtime)*	**Methodology**	**Notes / Caveats**
Basic	JSON in/JSON out interface	Yes	Pre/Post processing preserves JSON In/Out contract. Even preserving existing init(), run() methods used by the OW impl.	*None*
Basic	pass in environment variables as parameters	Yes	JSON "values" data preserved, allow existing OpenWhisk init() method to move them from Environment Variables.	*None*
Http (Web*)	pass HTTP traffic into function container by transforming incoming http workload to OW-complian web action	Yes	*preProcessHTTPContext(req)*	*None*
Http (Web)	allow anonymous invocation via HTTP GET, PUT, DELETE, POST	Yes	• *Performed at runtime initialization as part of Knative build* • *BuildTemplate (Build) allows parameter to declare list of HTTP Methods supported by the associated function*	Note: PATCH, HEAD, OPTIONS are *not currently supported*; *however, these are not featured in any known test cases / examples.* Tracked/Discussed under *Issue # 212*
Http (Web)	Env. Var. mapping (Standard) – (all __OW_* variables)	Yes	*preProcessInitData(req), preProcessActivationData(req)*	*None*
Http (Web)	Env. Var, mapping (Non-Standard) to OW paramaters	Yes	*preProcessInitData(req), preProcessActivationData(req)*	*None*
Http (Web)	Query Parameters - mapping to function args.	Yes	*preProcessHTTPContext(req)*	Mapped to __OW_QUERY

Continued on next page …

- Web Actions are effectively HTTP Actions with the ability to declare a public endpoint, which would be done in conjunction with an API Gateway or similar (Kube) Service
- Http "`raw = true| false`" : actions effectively only distinguish functions that declare themselves able to handle "raw" http input (body) data with associated .ext Content-Type

Knative Platform Impl. – Functional view of capabilities (continued)

What we can/cannot do within the runtime to provide OW functional equivalency under Knative/Kubernetes

OW Function Class	**OpenWhisk Capability** *(Supported via Native OW Platformw with Controller+Runtime)*	**Supported** *(Knative-Built Runtime)*	**Methodology**	**Notes / Caveats**
Http (Web)	Body Prameters – mapping to function args.	No (WIP)	*TBD*	Mapped to __*OW_BODY* Tracked/discussed under Issue # 213
Http (Web)	Content Extensions - Support invocation of non-standard extensions e.g. {QUALIFIED ACTION NAME}.{EXT}	No (WIP)	• Could allow function authors to declare in Build Template (build time) or in Service (runtime)	• Tracked/discussed under Issue # 214
Http (Web)	FORM data - Support Web Action FORM data	No (WIP)	• *Work under discussion, planned or In-progress.*	• Tracked/discussed under Issue # 215
Http (Web)	Inferred Content-Type from Non-JSON body	No (WIP)	Reponse Content-Type inferred from body, Work-in-progress	• Tracked/discussed under Issue # 216
Http (web)	Bad Request - when __*ow_** are part of invocation	No (WIP)	Mark the incoming invocation as a bad request if body/query has any of __*ow_** reserved variables.	• Tracked/discussed under Issue # 217
Http (Web)	Protected Parameters – protecting action parameters with final annotation	No (WIP)	Given action parameters should be protected with final annotation.	• Tracked/discussed under Issue # 218
Basic	invocable via HTTP POST via api key	N/A	API Key (if provided) on Activation is preserved	Would require an API Gateway service as part of a larger IAM cloud platform.
Http (Web)	deviate from current openwhisk URL ok	N/A	Under Knative, the endpoint is assigned/determined both by the Kube Namespace, as well as the Knative (Kube) Service name	If specific Web endpoints that follow the OW naming convention are needed, this would need to be mapped at platform ingress

Phase 2
2-Stage Build using Knative Build Templates

Modifying Kaniko to support OpenWhisk Runtimes

For now, since we need to modify the actual OpenWhisk Runtime, we will have 2 Build Templates:

Build Template for building Modified OpenWhisk NodeJS10 Runtime

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: openwhisk-nodejs-knative-runtime
spec:
  parameters:
  - name: TARGET_IMAGE_NAME         # Passed to Kanico as an arg.
  - name: TARGET_IMAGE_TAG          # Passed to Kanico as an arg.
  - name: WORKSPACE_SUBDIRECTORY    # The subdir. of the workspace/repo e.g.,
  - name: __OW_RUNTIME_DEBUG
    default: false
  - name: __OW_RUNTIME_PLATFORM     # one of enum[ "openwhisk", "knative", ...] or ERROR
    default: openwhisk
  steps:
  - name: dockerfile
    image: gcr.io/kaniko-project/executor@<<commit hash>>> # Note: will want to use latest
    command:
    - /busybox/sh
    args:
    - -c
    - |
      cd /workspace/${WORKSPACE_SUBDIRECTORY}
      cat <<EOF > Dockerfile
        # Append these to OpenWhisk NodeJS10 runtime's Dockerfile
        ENV __OW_RUNTIME_DEBUG "${__OW_RUNTIME_DEBUG}"
        ENV __OW_RUNTIME_PLATFORM "${__OW_RUNTIME_PLATFORM}"
        COPY . .
        ~~ENTRYPOINT ["/opt/aws-custom-runtime"]~~
      EOF
  - name: export
    image: gcr.io/kaniko-project/executor@<<commit hash>>> # Note: will want to use latest
    args:
    - --context=/workspace/${WORKSPACE_SUBDIRECTORY}
    - --dockerfile=/workspace/${WORKSPACE_SUBDIRECTORY}/Dockerfile
    - --destination=${TARGET_IMAGE_NAME}:${TARGET_IMAGE_TAG}
```

Build Template for building the TARGET image with /init data (i.e., Action code)

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: openwhisk-nodejs-runtime-application
spec:
  parameters:
  - name: TARGET_IMAGE_NAME         # Passed to Kanico as an arg.
  - name: TARGET_IMAGE_TAG          # Passed to Kanico as an arg.
  - name: WORKSPACE_SUBDIRECTORY    # The subdir. of the workspace/repo
  ~~- name: HANDLER # TBD~~
  - name: __OW_ACTION_CODE
  - name: __OW_ACTION_NAME:         # e.g., helloNodeJS
  - name: __OW_ACTION_MAIN
    default: main
  - name: __OW_ACTION_BINARY
    default: false
  steps:
  - name: dockerfile
    image: gcr.io/kaniko-project/executor@<<commit hash>>> # Note: will want to use latest
    command:
    - /busybox/sh
    args:
    - -c
    - |
      cd /workspace/${DIRECTORY}
      cat <<EOF > Dockerfile
        FROM docker.io/${DOCKER_USERNAME}/nodejs-10-action:latest
        ENV __OW_ACTION_CODE "${__OW_ACTION_CODE}"
        ENV # etc.
        COPY . .
        ~~ENTRYPOINT ["/opt/aws-custom-runtime"]~~
      EOF
  - name: export
    image: gcr.io/kaniko-project/executor@<<commit hash>>> # Note: will want to use latest
    args:
    - --context=/workspace/${DIRECTORY}
    - --dockerfile=/workspace/${DIRECTORY}/Dockerfile
    - --destination=${TARGET_IMAGE_NAME}:${TARGET_IMAGE_TAG}
```

Modifying Kaniko to support OpenWhisk Runtimes

For now, since we need to modify the actual OpenWhisk Runtime, we will have 2 Build Templates:

Knative Service Template for building NodeJS10 image with our modifications:
- *No /init data (i.e., no Action code)*

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: nodejs-10-action
  namespace: default
spec:
  runLatest:
    configuration:
      build:
        apiVersion: build.knative.dev/v1alpha1
        kind: Build
        spec:
          serviceAccountName: openwhisk-runtime-builder
          source:
            git:
              url: https://github.com/mrutkows/openwhisk-knative-build.git
              revision: master
          template:
            name: kaniko
            arguments:
              - name: IMAGE
                value: docker.io/{DOCKER_USERNAME}/nodejs-10-action:latest
              - name: DOCKERFILE
                value: ./runtimes/javascript/Dockerfile
      revisionTemplate:
        spec:
          container:
            image: docker.io/{DOCKER_USERNAME}/nodejs-10-action:latest
            imagePullPolicy: Always
```

Build Template for building the TARGET image with /init data (i.e., Action code)

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: openwhisk-nodejs-runtime
spec:
  parameters:
  - name: TARGET_IMAGE_NAME # Used by Kanico
  - name: TARGET_IMAGE_TAG # used by Kanicko default: latest
  - name: DIRECTORY # The subdir. of the workspace/repo
  ~~- name: HANDLER # TBD~~
  - name: __OW_ACTION_CODE
  - name: __OW_ACTION_NAME: helloNodeJS
  - name: __OW_ACTION_MAIN
    default: main
  - name: __OW_ACTION_BINARY
    default: false
  - name: __OW_ACTION_CODE: # "function main() {return {payload: 'Hello'};}"
  - name: __OW_DEBUG
    default: false
  steps:
  - name: dockerfile
    image: gcr.io/kaniko-project/executor@<<commit hash>>> # Note: will want to use latest
    command:
    - /busybox/sh
    args:
    - -c
    - |
      cd /workspace/${DIRECTORY}
      cat <<EOF > Dockerfile
        FROM docker.io/${DOCKER_USERNAME}/nodejs-10-action:latest
        ENV __OW_ACTION_CODE "${__OW_ACTION_CODE}”
        ENV # etc.
        COPY . .
        ~~ENTRYPOINT ["/opt/aws-custom-runtime"]~~
      EOF
  - name: export
    image: gcr.io/kaniko-project/executor@<<commit hash>>> # Note: will want to use latest
    args:
    - --context=/workspace/${DIRECTORY}
    - --dockerfile=/workspace/${DIRECTORY}/Dockerfile
    - --destination=${TARGET_IMAGE_NAME}:${TARGET_IMAGE_TAG}
```

https://github.com/mrutkows/openwhisk-knative-build/

Modifying Kaniko to support OpenWhisk Runtimes *(proxy and functions)*

Reference the OpenWhisk runtime image instead of TriggerMesh

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: openwhisk-nodejs-runtime
spec:
  parameters:
  - name: TARGET_IMAGE_NAME # Used by Kanico
  - name: TARGET_IMAGE_TAG # used by Kanicko default: latest
  - name: DIRECTORY # The subdir. of the workspace/repo
  ~~- name: HANDLER # TBD~~
  - name: __OW_ACTION_CODE
  - name: __OW_ACTION_NAME: helloNodeJS
  - name: __OW_ACTION_MAIN
    default: main
  - name: __OW_ACTION_BINARY
    default: false
  - name: __OW_ACTION_CODE: # "function main() {return {payload: 'Hello'};}"
  - name: __OW_DEBUG
    default: false
  steps:
  - name: dockerfile
    image: gcr.io/kaniko-project/executor@<<commit hash>>> # Note: will want to use latest
    command:
    - /busybox/sh
    args:
    - -c
    - |
      cd /workspace/${DIRECTORY}
      cat <<EOF > Dockerfile
        FROM docker.io/${DOCKER_USERNAME}/nodejs-10-action:latest
        ENV __OW_ACTION_CODE "${__OW_ACTION_CODE}"
        ENV # etc.
        COPY . .
        ~~ENTRYPOINT ["/opt/aws-custom-runtime"]~~
      EOF
  - name: export
    image: gcr.io/kaniko-project/executor@<<commit hash>>> # Note: will want to use latest
    args:
    - --context=/workspace/${DIRECTORY}
    - --dockerfile=/workspace/${DIRECTORY}/Dockerfile
    - --destination=${TARGET_IMAGE_NAME}:${TARGET_IMAGE_TAG}
```

Notes:

- **BuildTemplate**
  - Spec:
    - TBD
  - steps:
    - “dockerfile”
      - TBD
    - “export”
      - TBD

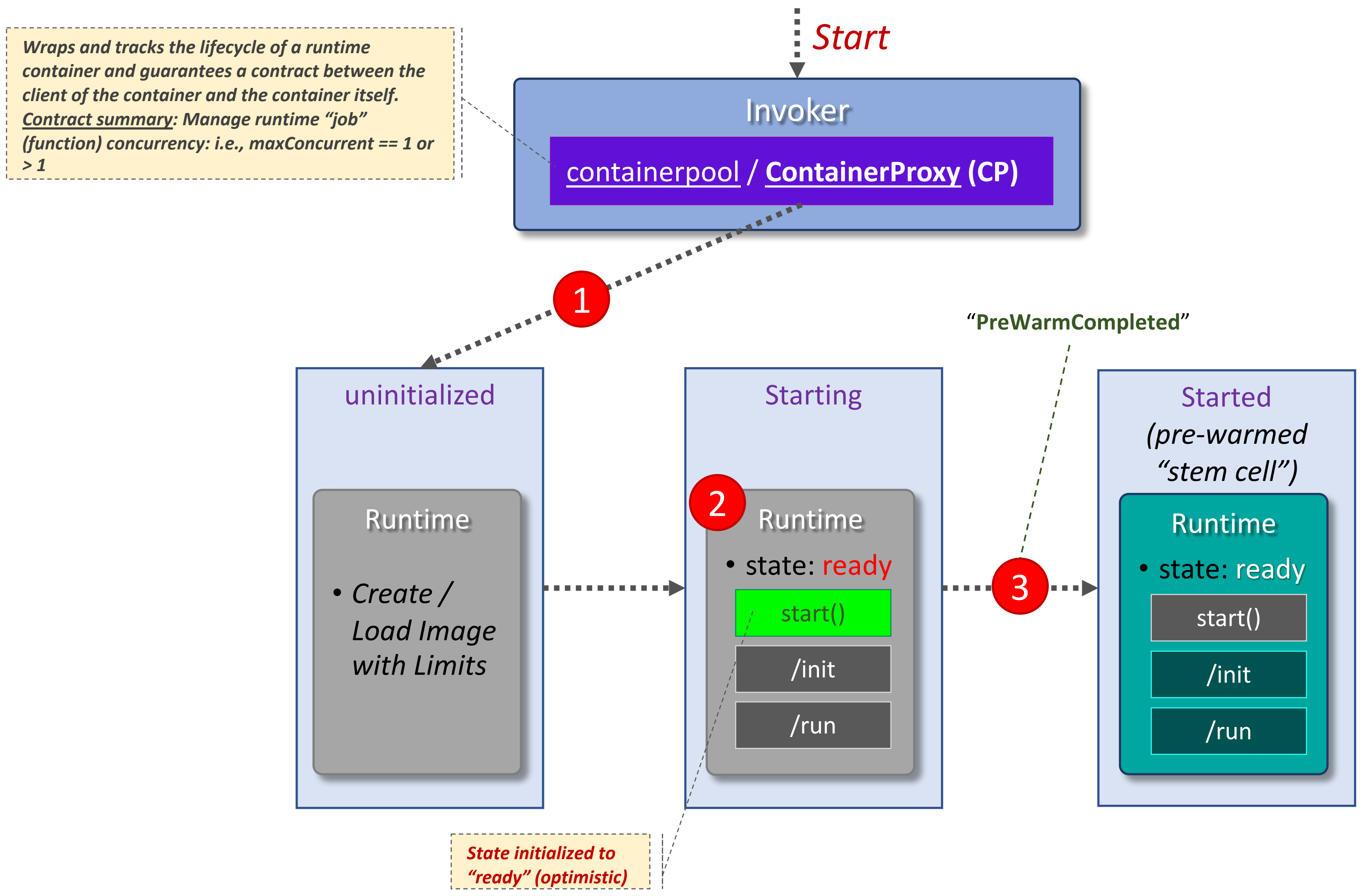
https://github.com/mrutkows/openwhisk-knative-build/

Understanding an OpenWhisk Runtime Invocation sequence

(using NodeJ)

OpenWhisk: Invoker interaction with Runtimes: “Stem-cell”

Lifecycle (state) mgmt. of an OpenWhisk Runtime within the Invoker “ContainerProxy”



Wraps and tracks the lifecycle of a runtime container and guarantees a contract between the client of the container and the container itself. Contract summary: Manage runtime “job” (function) concurrency: i.e., maxConcurrent == 1 or > 1
Start
Invoker
containerpool / ContainerProxy (CP)
1
uninitialized
Runtime
• Create / Load Image with Limits
Starting
2
Runtime
• state: ready
start()
/init
/run
“PreWarmCompleted”
3
Started (pre-warmed “stem cell”)
Runtime
• state: ready
start()
/init
/run
State initialized to “ready” (optimistic)

1 Start *(event from Controller)*

- CP :“loads” runtime image w/ Limits
  - Sets state to “**Starting**”
- Creates “fake” Pre-warmed” data

2
- **RT: initializes**
  - Creates app server (Http Proxy)
  - registers /init and /run handlers
    - *All other routes set to error**
  - Invokes start():
    - starts http listener (IP, port)
    - sets timeout to 0

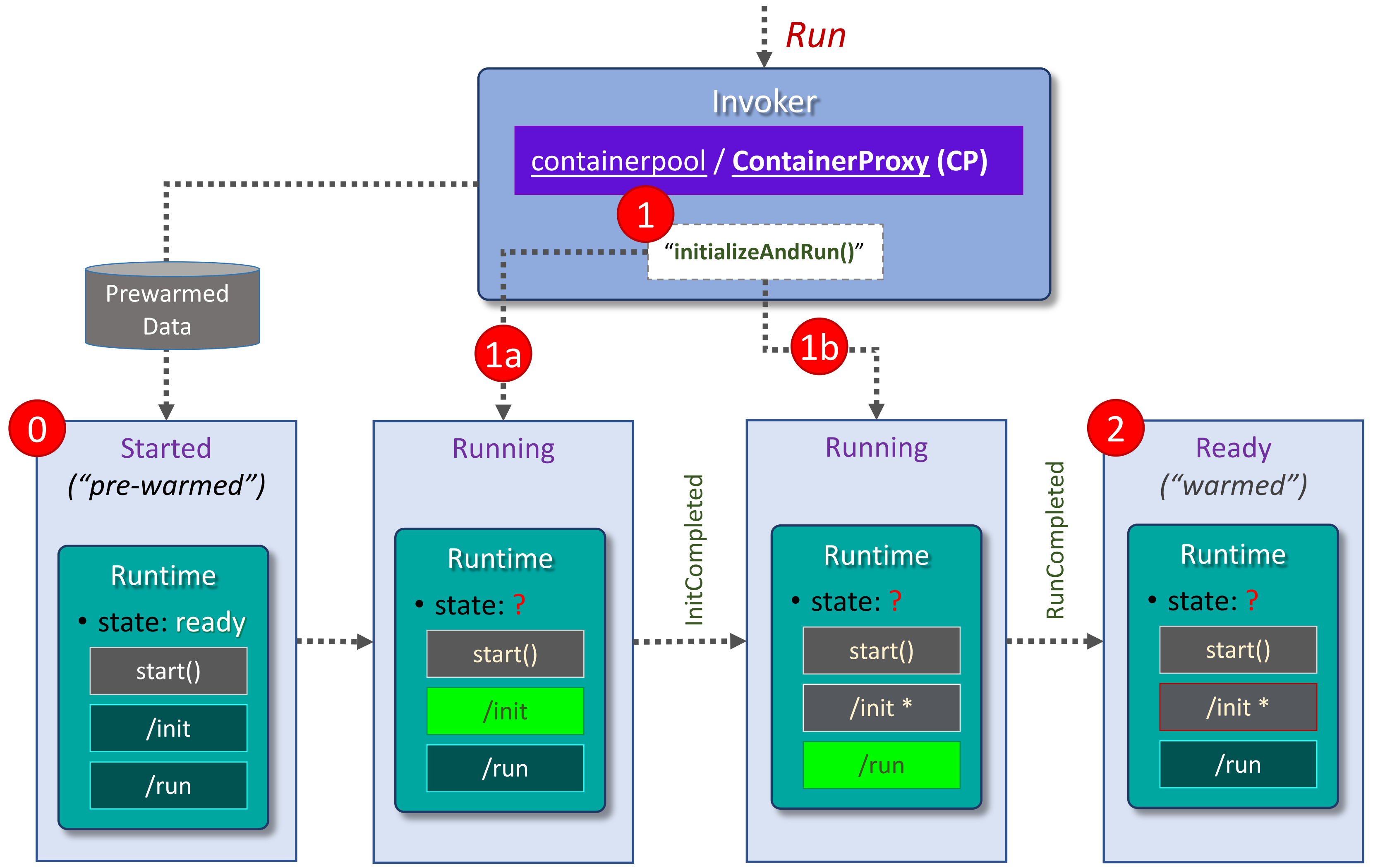
3
- **PreWarmCompleted**
  - CP: Waits in “**Starting**” state for “**PreWarmCompleted**”
  - CP: Waits in the “**Started**” state for a “**Run**” event *(from Controller)*
  - The container is considered a pre-warmed “stem cell”
    - i.e., ready for any function…

** the app.use() middleware assures all other endpoints besides /init and /run result in a 500 HTTP error return codes*

hhttps://github.com/apache/incubator-openwhisk/blob/master/core/invoker/src/main/scala/org/apache/openwhisk/core/containerpool/ContainerProxy.scala

OpenWhisk: Invoker interaction with Runtimes: Run (Cold & Pre-warmed)

Lifecycle (state) mgmt. of an OpenWhisk Runtime within the Invoker "ContainerProxy"



0 Run (Cold, No Pre-Warmed data)
- CP : Preforms all steps shown for "Start" event
  - BUT, with actual pre-warmed data
- CP: Invokes **initializeAndRun**() method against container.
- CP: Waits in "Running" state
  - Skips "Starting" state

1 Run (Pre-warmed)
- CP : invokes **initializeAndRun**() once "**PreWarmCompleted**" (event) is detected.

1a **initializeAndRun()**
- RT: /init
- *Current code does not allow "re-init" with new functional code **

1b **initializeAndRun()**
- RT: /run

2

hhttps://github.com/apache/incubator-openwhisk/blob/master/core/invoker/src/main/scala/org/apache/openwhisk/core/containerpool/ContainerProxy.scala

OpenWhisk Runtime: NodeJS: Invocation sequence with entry points/call stacks

NodeJS10 Runtime: Docker container build & layout

Dockerfile:

```
FROM node:10.15.0-stretch
RUN apt-get update && apt-get install -y \
imagemagick \
unzip \
&& rm -rf /var/lib/apt/lists/*

WORKDIR /nodejsAction

COPY . .
# COPY the package.json to root container, so we can
install npm packages a level up from user's packages,
so user's packages take precedence
COPY ./package.json /

RUN cd / && npm install --no-package-lock \
&& npm cache clean –force

EXPOSE 8080

CMD node --expose-gc app.js
```

Container layout (using Interactive Bash shell):

```
$ docker run -it openwhisk/action-nodejs-v10 sh
# pwd
/nodejsAction

# ls
CHANGELOG.md  app.js  package.json  runner.js  src

# ls src
service.js
```

Filesystem view (filesystem starts in WORKDIR)

```
nodejsAction\
|-- CHANGELOG.md
|-- app.js
|-- package.json
|-- runner.js
|-- src\
    |-- service.js
```

https://github.com/apache/incubator-openwhisk-runtime-nodejs

OpenWhisk Runtime: NodeJS: Invocation sequence with entry points/call stacks

- ***NodeJS Runtime Initialization: The runtime application uses the Express Application Framework***
- ***Wraps all "handlers" (endpoints) exported***

core/nodejsActionBase/app.js:

```
var config = {
        'port': 8080,
        'apiHost': process.env.__OW_API_HOST,
        'allowConcurrent': process.env.__OW_ALLOW_CONCURRENT
};

var bodyParser = require('body-parser');
var express    = require('express');
var app = express();

/**
 * instantiate an object which handles REST calls from the Invoker
 */
var service = require('./src/service').getService(config);

app.set('port', config.port);
app.use(bodyParser.json({ limit: "48mb" }));

app.post('/init', wrapEndpoint(service.initCode));
app.post('/run',  wrapEndpoint(service.runCode));

app.use(function(err, req, res, next) {
    console.error(err.stack);
    res.status(500).json({ error: "Bad request." });
  });

service.start(app);
```

```
/**
 * Wraps an endpoint written to return a Promise into an express endpoint,
 * producing the appropriate HTTP response and closing it for all controlable
 * failure modes.
 *
 * The expected signature for the promise value (both completed and failed)
 * is { code: int, response: object }.
 *
 * @param ep a request=>promise function
 * @returns an express endpoint handler
 */
function wrapEndpoint(ep) {
    return function (req, res) {
        try {
            ep(req).then(function (result) {
                res.status(result.code).json(result.response);
            }).catch(function (error) {
                if (typeof error.code === "number" &&
                    typeof error.response !== "undefined") {
                    res.status(error.code).json(error.response);
                } else {
                    console.error("[wrapEndpoint]", "invalid errored promise",
                         JSON.stringify(error));
                    res.status(500).json({ error: "Internal error." });
                }
            });
        } catch (e) {
            console.error("[wrapEndpoint]", "exception caught", e.message);
            res.status(500).json({ error: "Internal error (exception)." });
        }
    }
}
```

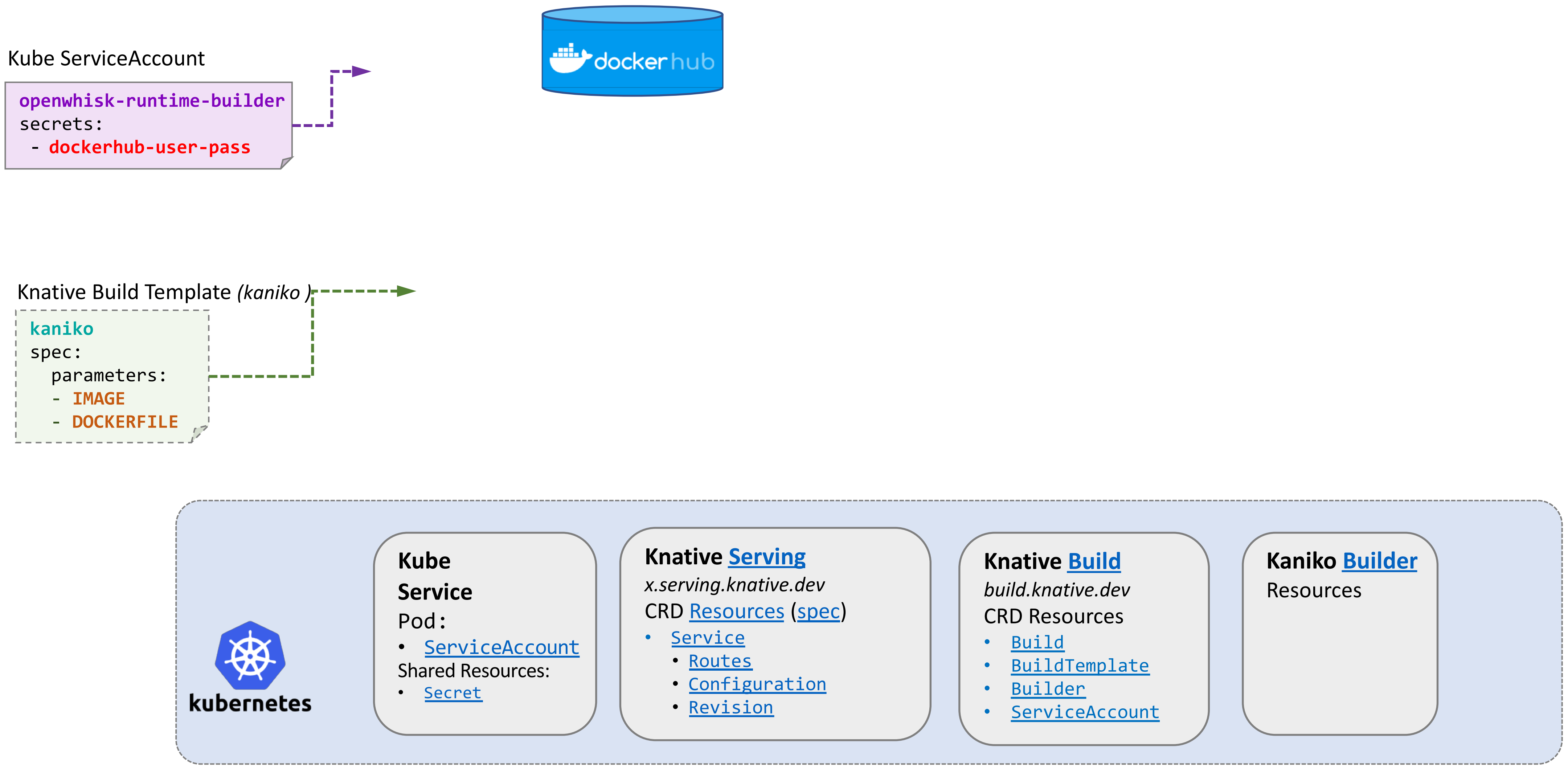
https://github.com/apache/incubator-openwhisk-runtime-nodejs

Backup Materials

Phase 1: Showing Kubernetes/Knative resources at all build and deploy stages

Using NodeJS 10 GitHub source as an example

Knative Service Template for building NodeJS10 image with our modifications:
- No /init data (i.e., no Action code)



Kube ServiceAccount

```
openwhisk-runtime-builder
secrets:
 - dockerhub-user-pass
```

Knative Build Template *(kaniko)*

```
kaniko
spec:
  parameters:
  - IMAGE
  - DOCKERFILE
```

Kube Service
Pod:
- ServiceAccount

Shared Resources:
- Secret

Knative Serving
x.serving.knative.dev
CRD Resources (spec)
- Service
  - Routes
  - Configuration
  - Revision

Knative Build
build.knative.dev
CRD Resources
- Build
- BuildTemplate
- Builder
- ServiceAccount

Kaniko Builder
Resources

kubernetes

Issue: Runtime Single Entrypoint – HTTP Body – Key collision

- ***If you want a "stem cell" (init not "baked into" dedicated Runtime image),***
- ***Then we will need to separate logical data within the Http Request body***

__Name__ Key in Init data collides with "Name" key used for function's parameter data

```
POST http://localhost:8080/ HTTP/1.1
content-type: application/json

{
  "init": {
    "name" : "nodejs-helloworld-with-params",
    "main" : "main",
    "binary": false,
    "code" : "function main(params) {return {payload: 'Hello ' + params.name + ' from ' + params.place +  '!'};}"
  },
  "activation": {
    "namespace": "default",
    "action_name": "nodejs-helloworld-with-params",
    "api_host": "",
    "api_key": "",
    "activation_id": "",
    "deadline": "4102498800000"
  },
  "value": {
    "name" : "Joe",
    "place" : "TX"
  }
}
```