Containers Workshop

Global Summit 2023

June 4, 2023

InterSystems®
Creative data technology

Introduction

Agenda

Introduction
- Workshop Structure
- Why Use Containers?

What Are Containers?
- Basic Container Operations
- Persisting Data in a Container

InterSystems IRIS in Containers
- Understanding InterSystems IRIS in Containers
- Using Durable %SYS

A Closer Look at Containers
- Inspecting Container Logs
- Using Build Processes
- Using Docker Compose for Multiple Containers

Putting It All Together
- Build a Containerized Application on InterSystems IRIS

Workshop Presenters

- Derek Robinson
  - Senior Technical Online Course Developer
- Erik Hemdal
  - Principal Quality Development Engineer
- Kerry Kirkham
  - Sales Engineer
- Bob Kuszewski
  - Product Manager — Developer Experience

And several more InterSystems experts are available to assist you throughout the workshop!

Workshop Structure

- 4 hour duration
  - 2pm to 6pm
- Helpers can assist with machine setup
  - Loaner machines available on a limited basis if needed
- Presentations at breakpoints in the exercise
  - Feel free to proceed if you are ahead, or grab some coffee and wait!
- Designed to be taken home
  - By using your own machine, you'll walk out with a running version of your containerized application!

Schedule

2:00 – 2:30: Setup & Introduction (Derek)

2:30 – 3:00: Work on Exercises 1-3

3:00 – 3:15: Presentation (Erik)

3:15 – 3:30: Work on Exercise 4

3:30 – 3:45: Coffee break 

3:45 – 4:00: Presentation (Kerry)

4:00 – 4:30: Work on Exercises 5-6

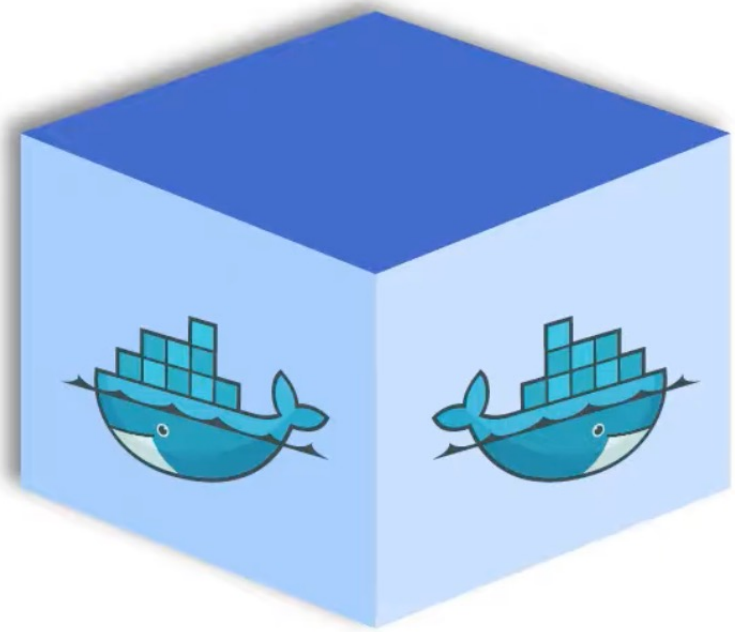
4:30 – 4:45: Presentation (Bob)

4:45 – 6:00: Work on Exercise 7 & Bonus

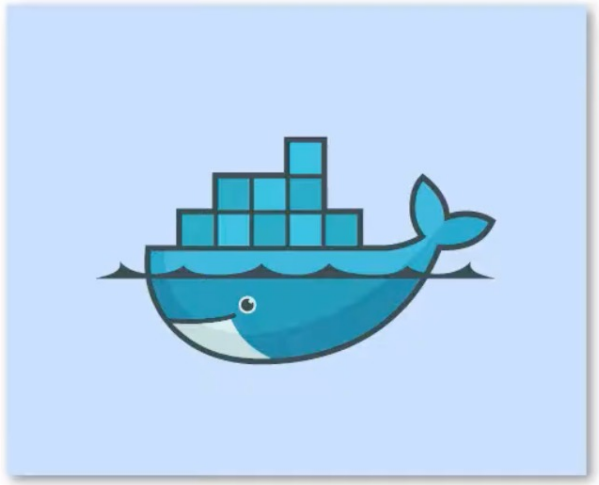
What are Containers?

What is a Container?

- Containers are platform-independent units of software
- Code and all dependencies are packaged for application to run quickly and reliably
- Runnable on **any host system** that has a container runtime engine
- Portable and efficient runtime solution

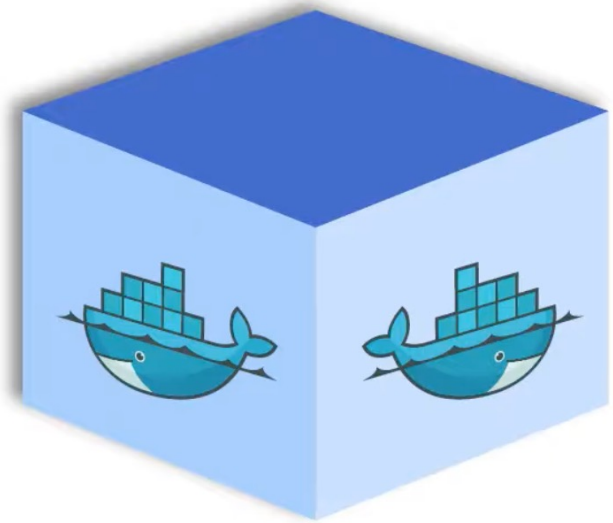


Containers and Images



Container Images are standardized, portable, runnable software bundles.

Think of an installable software package.

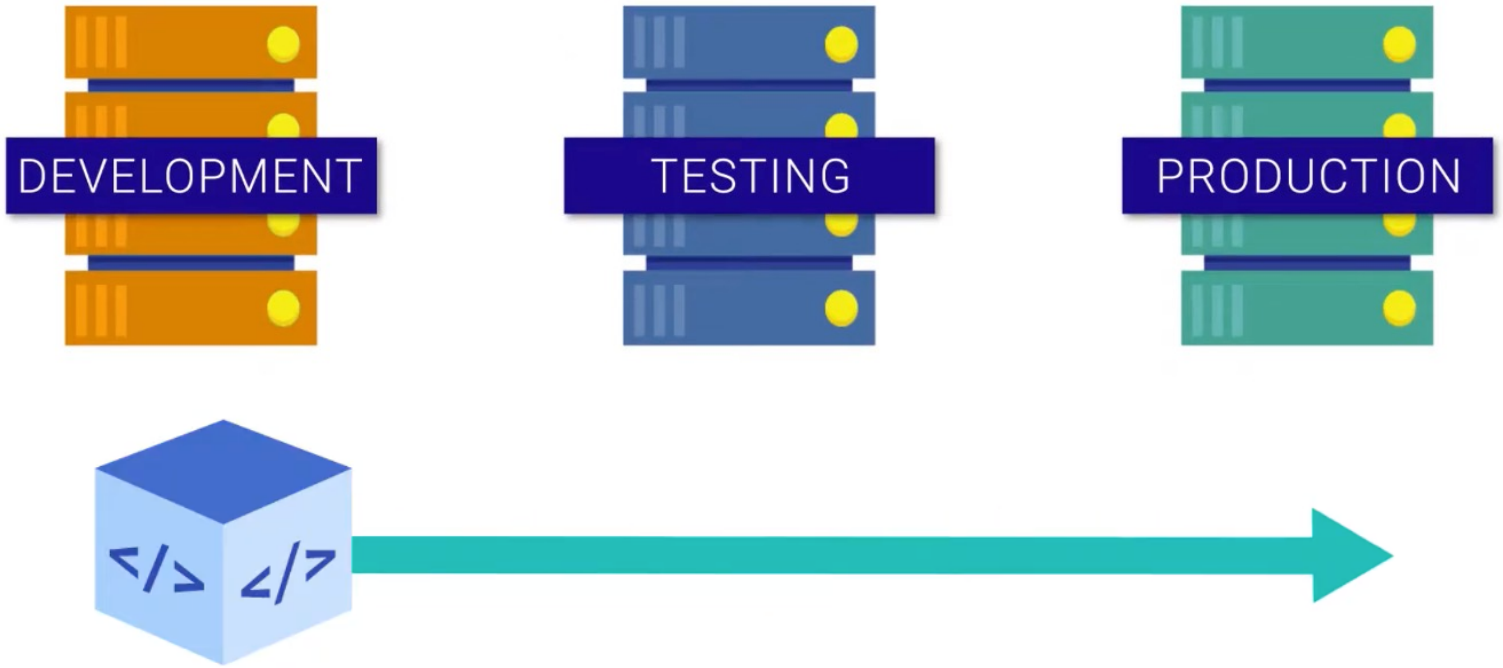


Containers are running instances of images, executed in isolation and resource-controlled.

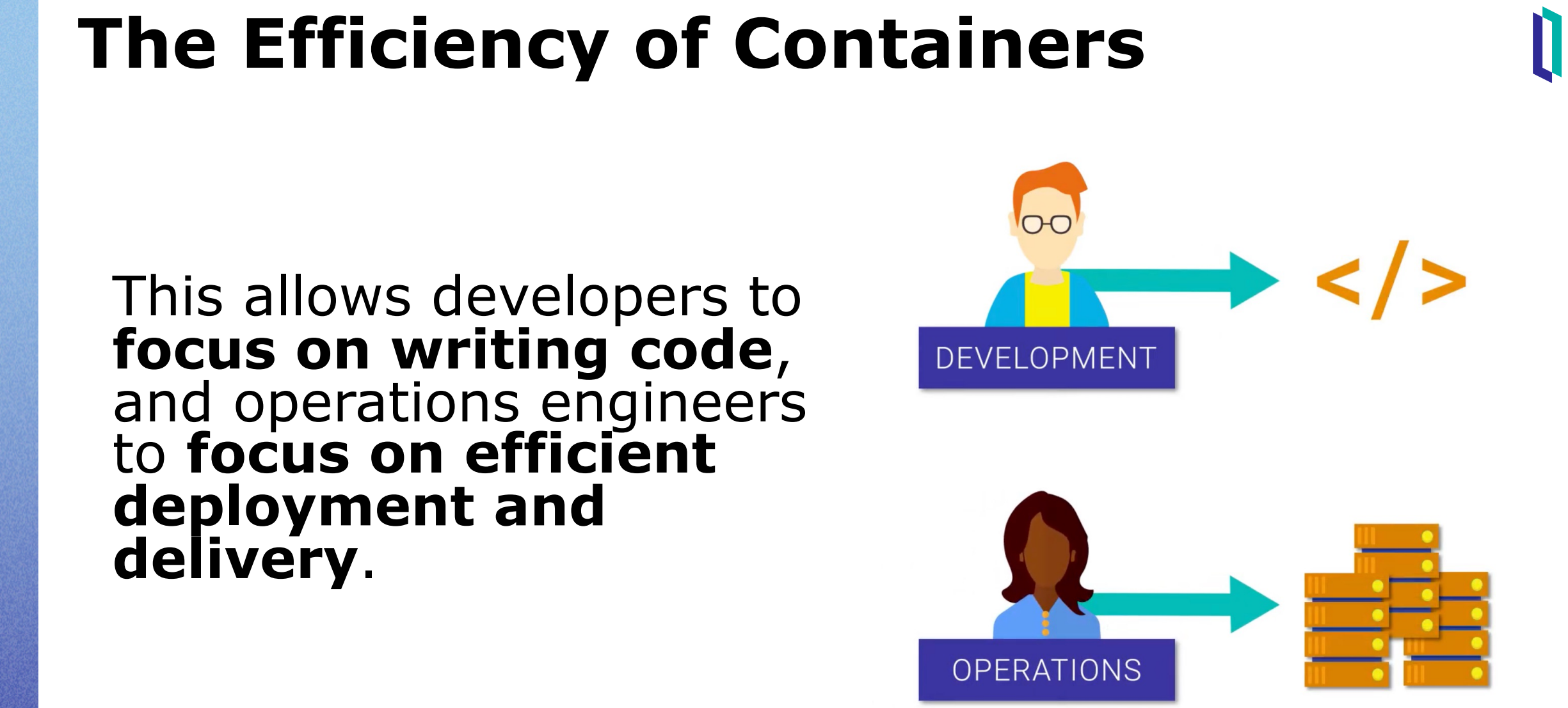
Think of a running instance of that software package.

The Efficiency of Containers

Containers allow you to move your application code through various environments—such as from DEV to TEST to PROD—without the friction of build discrepancies and software dependencies.

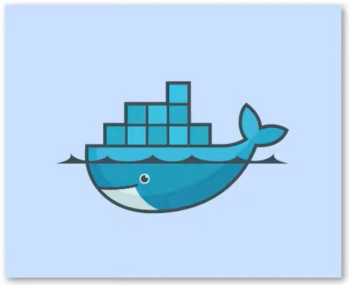


The Efficiency of Containers

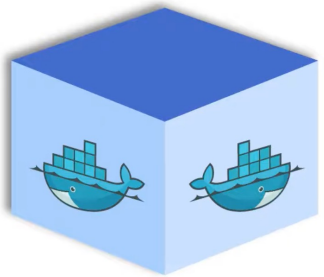


This allows developers to **focus on writing code**, and operations engineers to **focus on efficient deployment and delivery**.

Building and Running Containers



You can **build a container image** using a *Dockerfile*, which defines exactly what the image includes, and a `docker build` command.



You can **run a container** using a `docker run` command, including various criteria for how that container should be created.

Exercise

- Try Exercises 1-3
- Ask for help if you need it
- Next presentation:
  - **3:00 PM**

InterSystems IRIS in Containers

Why are we here?

- To learn about differences between containerized deployments and traditional deployments on virtual machines
- To learn key InterSystems IRIS features for control and persistence (iris-main, Durable %SYS)
- To see some practical details about running InterSystems IRIS in a container

Containers vs. Virtual Machines

A Container is an Application

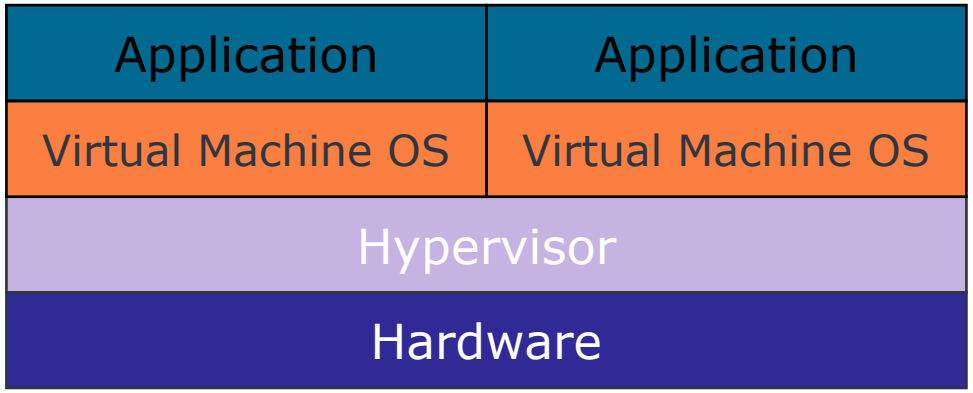
- "Just" processes running code
- Files are ephemeral
- Isolated from the rest of the system (storage, networking, privileges)

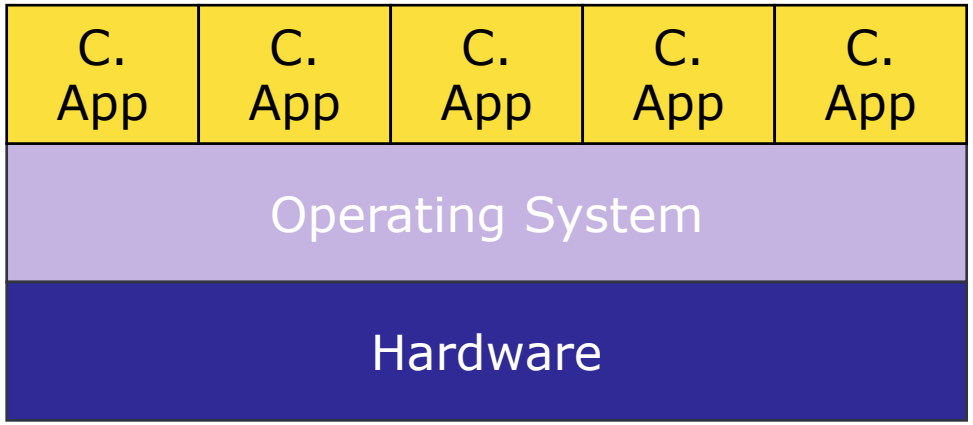
A VM is well… a Machine

- A fully virtualized environment
- OS, Memory, File System, etc.

Different Use Cases

- VMs allow for density of Machines on hardware
- Containers allow for density of Applications on hardware





Containers are ephemeral. Databases aren't.

- Containers are "jails" for software that isolated from the rest of the computer
  - The filesystem they use is isolated
  - Network access is blocked outside the container
  - When the container is deleted, its filesystem goes away too
  - By default, this means they are stateless

- The test.txt file vanished in the BusyBox exercise
  - The file was written into isolated storage that was inaccessible when you started the new container

- That won't work for InterSystems IRIS (or any database):
  - We need to arrange an orderly startup and shutdown (iris start… and iris stop…)
  - We need to manage network access
  - We need **persistent data**

- **Here's how we do it**

Orderly Start/Stop: the iris-main program

- Starting the container starts iris-main, which remains running even if you shutdown IRIS via ‘iris stop’.
- If the container is signaled to exit, iris-main will perform a clean shutdown for you.
- There are many options that we won’t explore now.

```
irisowner@103df966af39: ~
ehemdal@USE7420ehemdal:/GS2023$ docker exec -it GS2023 bash
irisowner@103df966af39:~$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
irisown+     1     0  0 15:10 ?        00:00:00 /tini -- /iris-main
irisown+     7     1  0 15:10 ?        00:00:00 /iris-main
irisown+    13     1  0 15:10 ?        00:00:00 /bin/sh /home/irisowner/irissys/ISCAgentUser
irisown+    18    13  0 15:10 ?        00:00:00 /home/irisowner/irissys/ISCAgent -u
irisown+    19    18  0 15:10 ?        00:00:00 /home/irisowner/irissys/ISCAgent -u
irisown+   766     1  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb -s/ISC/iris.data.d/mgr/ -w/ISC/
irisown+   807   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb WD
irisown+   808   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb GC
irisown+   809   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb JD
irisown+   810   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb AUX1
irisown+   811   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb AUX2
irisown+   812   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb AUX3
irisown+   813   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb AUX4
irisown+   814   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb AUX6
irisown+   815   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb AUX5
irisown+   816   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb AUX7
irisown+   817   766  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb DBXD
irisown+   819     1  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb -s/ISC/iris.data.d/mgr -cj -p13
irisown+   825     1  0 15:10 ?        00:00:00 /usr/irissys/bin/irisdb -s/ISC/iris.data.d/mgr -cj -p13
```

Persisting data: Durable %SYS

- To persist data, IRIS containers recognize an environment variable $ISC_DATA_DIRECTORY that you set at runtime – in your Docker run command
- That directory must point to a directory outside of the container, on the host machine.
- On first startup, IRIS will move the contents of the manager's directory into $ISC_DATA_DIRECTORY, so IRISSYS, logs, journals, CPF, and necessary data will be persistent.
  - $ISC_DATA_DIRECTORY must be empty at first startup
  - If a later version of IRIS is started in a new container, it will be upgraded in-place using $ISC_DATA_DIRECTORY

Details of run.sh: putting it all together

- Let’s put everything together and look at a specific, simple run command

ehemdal@USE7420ehemdal: /GS2023

```
ehemdal@USE7420ehemdal:/GS2023$ cat run.sh
# Simple run command to launch an InterSystems IRIS container on Linux
# create the directory 'durable' with mode 777

docker run \
    --publish 80:52773 \
    --publish 1972:1972 \
    --volume=$PWD/durable:/ISC \
    --rm=true \
    --detach \
    --env ISC_DATA_DIRECTORY=/ISC/iris.data.d \
    --name=GS2023 \
    containers.intersystems.com/intersystems/iris-community:2023.1.0.229.0
ehemdal@USE7420ehemdal:/GS2023$
```

The run command: publishing network ports

```
docker run  \
  --publish 80:52773 \
  --publish 1972:1972 \
  --volume=$PWD/durable:/ISC \
  --rm=true \
  --detach \
  --env ISC_DATA_DIRECTORY=/ISC/iris.data.d \
  --name=GS2023 \
  containers.intersystems.com/intersystems/iris-
community:2023.1.0.229.0
```

- These options open (“publish”) network ports so connections can be made from outside the container.
- --publish host-port:container-port
- This exposes the container’s private Web server outside on port 80, even though it is port 52773 inside the container.
- The SuperServer is available on port 1972 both inside the container and outside on the host.
- **Change these ports if they conflict with some other application on your computer.**

The run command: accessing persistent storage

```
docker run  \
  --publish 80:52773 \
  --publish 1972:1972 \
  --volume=$PWD/durable:/ISC \
  --rm=true \
  --detach \
  --env ISC_DATA_DIRECTORY=/ISC/iris.data.d \
  --name=GS2023 \
  containers.intersystems.com/intersystems/iris-
community:2023.1.0.229.0
```

- The --volume option sets up a **bind mount** which lets the container use a (persistent) directory on the host.
- The host directory is named "durable" under the current working directory. Inside the container, it will be accessed at /ISC.
- The host directory needs to be writable by the container's user – irisowner – so that IRIS can create the Durable %SYS directory here.
- For our exercise, you can make it world-writable.
- You can define more bind mounts for other directories if you need them.

The run command: set the Durable %SYS directory

```
docker run  \
  --publish 80:52773 \
  --publish 1972:1972 \
  --volume=$PWD/durable:/ISC \
  --rm=true \
  --detach \
  --env ISC_DATA_DIRECTORY=/ISC/iris.data.d \
  --name=GS2023 \
  containers.intersystems.com/intersystems/iris-
community:2023.1.0.229.0
```

- Here is how we set the ISC_DATA_DIRECTORY environment variable.
- The Durable %SYS directory will be `/ISC/iris.data.d` inside the container.
- It will be `$PWD/durable/iris.data.d` on the host, based on where you start.

Notice that the Durable %SYS directory

`$PWD/durable/iris.data.d`

is not the same as the bind mount directory

`$PWD/durable`

so that the Durable %SYS directory will be empty at startup.

The run command: more options

```
docker run  \
  --publish 80:52773 \
  --publish 1972:1972 \
  --volume=$PWD/durable:/ISC \
  --rm=true \
  --detach \
  --env ISC_DATA_DIRECTORY=/ISC/iris.data.d \
  --name=GS2023 \
  containers.intersystems.com/intersystems/iris-
community:2023.1.0.229.0
```

- `--rm` tells Docker to remove the container when it exits, so stopped containers don't accumulate.
- This is OK because your data is persisted in the Durable %SYS directory and in any other bind mounts you use.
- `--detach` tells Docker to run the container in the background. If you leave this out, you'll see the messages.log file scrolling on your terminal.
- `--name` assigns a name to the container, so you don't have to remember the long container ID.

The run command: the image and where to get it

```
docker run  \
  --publish 80:52773 \
  --publish 1972:1972 \
  --volume=/home/ehemdal/GS2023/durable:/ISC \
  --rm=true \
  --detach \
  --env ISC_DATA_DIRECTORY=/ISC/iris.data.d \
  --name=GS2023 \
  containers.intersystems.com/intersystems/iris-community:2023.1.0.229.0
```

- Finally, this specifies the container image to run
- `containers.intersystems.com` is the container registry to pull from.
- `/intersystems/iris-community` is the image to use.
- `2023.1.0.229.0` is the tag, or version we want.

Exercise

- Try Exercise 4 - Pull and Run InterSystems IRIS Containers
- Ask for help if you need it
- When you finish, take a **coffee break.**
- Next presentation:
  - **3:45 PM**

A Closer Look at Containers

Run a command in a container with docker exec

Overview

- The `docker exec` command is used to run a command in a container

  `-it` for Interactive Terminal programs

  `-u <username>` to specify the user

  `-e <name>=<value>` sets environment var

  `-d` to run in the background

Examples

- Run a non-interactive command:

```
$ docker exec <container ID> ls
```

- Interactive Terminals

```
$ docker exec -it <container ID> sh
```

- Interactive IRIS Session

```
$ docker exec -it <container ID> iris
session iris
```

Container Logs

- The *docker logs* command shows information logged by a running container.
  - STDIN is the command's input stream, which may include input from the keyboard or input from another command.
  - STDOUT is usually a command's normal output
  - STDERR is typically used to output error messages.
- By default, docker logs shows the command's STDOUT and STDERR
- If your image runs a non-interactive process such as a web server or a database, that application may send its output to log files instead of STDOUT and STDERR

Logs Command Options

- Use the container's ID to inspect the logs it generated:

  $ docker container logs [OPTIONS] <container ID>

Command [Options]	Description
--details	Display additional details provided to logs.
--follow, **-f**	Follow log output.
--since	Display logs since a specified timestamp (e.g. 2022-05-06T14:48:33Z) or relative (e.g. 20m).
--tail, **-n**	Specify the number of lines to show from the end of the logs.
--timestamps, **-t**	Show timestamp.
--until	Display logs before a specified timestamp.

Container Logs Location

- Docker, by default, writes the standard output and standard error of all your containers in files using the JSON format.
- The default logging driver for Docker is the JSON-file driver that stores logs on the host in the following directory:

```
/var/lib/docker/containers/<container_id>/<container_id>-json.log
└──────────────────┘
Docker Root Dir:
```

Container Logs Location

- Docker, by default, writes the standard output and standard error of all your containers in files using the JSON format.
- The default logging driver for Docker is the JSON-file driver that stores logs on the host in the following directory:

/var/lib/docker/containers/<container_id>/<container_id>-json.log

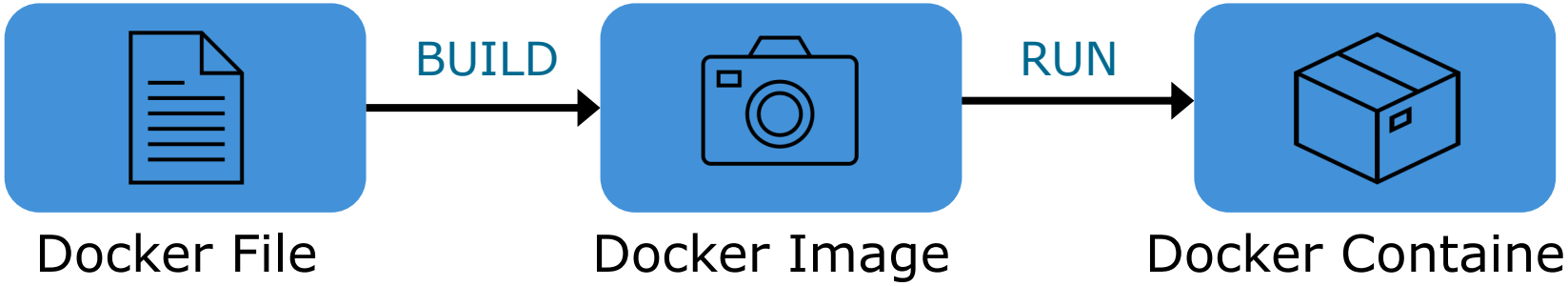
```
kirkham@ukmbp16kkirkham data % docker container ps
CONTAINER ID   IMAGE       COMMAND                  CREATED      STATUS                  PORTS
                NAMES
03ecd220255d   coffee:v1   "/tini -- /iris-main"   6 days ago   Up 6 days (healthy)   2188/tcp, 53773/tcp, 54773/tcp, 0.0.0.0:9998->1972/tcp, 0.0.0.0:99
99->52773/tcp   distracted_lederberg
kirkham@ukmbp16kkirkham data % docker run -it -v /var/lib/docker:/var/lib/docker alpine sh
/ # ls var/lib/docker/containers
03ecd220255d3e1dcac305758e045ada9095331649857d87b36d4ee69c957aa5  6c71e3009ad9dd67f5239c9919699fa923b436eb2c70c7c9667ebf6ca0322134
404d8d987cadfe005aeacb5def4b34a863b449badbb606fd38848cc21c7fd25c  a16485353fec306243a5abb3795c297d371e7938ac770c71e47827ec89267a5a
49eba977e7de5ad0eed55d482b0370f9d96d82c1f593a847c4e7184fe06da8be
/ #
/ # cat var/lib/docker/containers/03ecd220255d3e1dcac305758e045ada9095331649857d87b36d4ee69c957aa5/03ecd220255d3e1dcac305758e045ada9095331649857d87b36d4
ee69c957aa5-json.log
{"log":"[INFO] Executing command /home/irisowner/irissys/startISCAgent.sh 2188...\n","stream":"stdout","time":"2023-05-26T09:55:24.557981367Z"}
{"log":"[INFO] Writing status to file: /home/irisowner/irissys/iscagent.status\n","stream":"stdout","time":"2023-05-26T09:55:24.681599309Z"}
```

Using Build Process

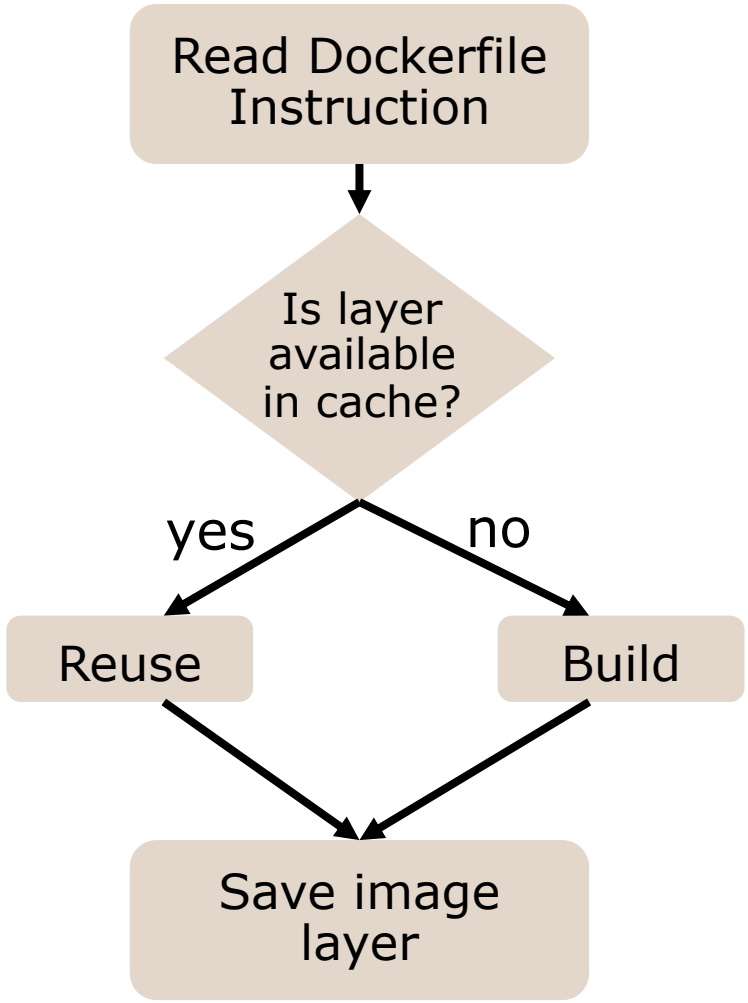
Container Image Build Process

To build the container image, we'll use a Dockerfile. A Dockerfile is simply a text-based file with no file extension that contains a script of instructions. Docker uses this script to build a container image.

- Images are built out of read-only layers.
- Dockerfiles specify image layer contents.
- Key Dockerfile commands: FROM, RUN, COPY and ENTRYPOINT



Build Cache



- Docker will cache the results of the first build of a Dockerfile, allowing subsequent builds to be much faster
- Cached build steps will be used until a change in the Dockerfile is found by the builder.
- After completion, the resulting image layer is labelled with a hash of the content of all current image layers
- The docker image history command allows us to inspect the build cache history of an image

  $ docker image history myimage:latest
- Using the build cache effectively is crucial for images that involve lengthy compile or download steps; in general, moving commands that change frequently as late as possible in the Dockerfile will minimize build times.

Dockerfile Commands

- FROM: base image to start from
- RUN: run a command in the environment defined so far
- CMD and ENTRYPOINT: define default behavior
- COPY and ADD: copy files into container

For the exercise we will use the following:

```
# Deriving our container from a prebuilt one
FROM node:10-slim
COPY server.js .
EXPOSE 8080
# Run the following default command when the container is run
CMD node server.js
```

Dockerfile Commands

- FROM: base image to start from
- RUN: run a command in the environment defined so far
- CMD and ENTRYPOINT: define default behavior
- COPY and ADD: copy files into container

For the exercise we will use the following:

```
# Deriving our container from a prebuilt one
FROM node:10-slim
COPY server.js .
EXPOSE 8080
# Run the following default command when the container is run
CMD node server.js
```

Image that contains the minimal packages needed to run node

Dockerfile Commands

- FROM: base image to start from
- RUN: run a command in the environment defined so far
- CMD and ENTRYPOINT: define default behavior
- COPY and ADD: copy files into container

For the exercise we will use the following:

```
# Deriving our container from a prebuilt one
FROM node:10-slim
COPY server.js .
EXPOSE 8080
# Run the following default command when the container is run
CMD node server.js
```

Simple Node.js web server that handles HTTP requests

Dockerfile Commands

- FROM: base image to start from
- RUN: run a command in the environment defined so far
- CMD and ENTRYPOINT: define default behavior
- COPY and ADD: copy files into container

For the exercise we will use the following:

```
# Deriving our container from a prebuilt one
FROM node:10-slim
COPY server.js .
EXPOSE 8080
# Run the following default command when the container is run
CMD node server.js
```

Inform Docker that the container listens on port 8080 at runtime

Dockerfile Commands

- FROM: base image to start from
- RUN: run a command in the environment defined so far
- CMD and ENTRYPOINT: define default behavior
- COPY and ADD: copy files into container

For the exercise we will use the following:

```
# Deriving our container from a prebuilt one
FROM node:10-slim
COPY server.js .
EXPOSE 8080
# Run the following default command when the container is run
CMD node server.js
```

Set the command to be executed when running the image

Using Docker Compose for Multiple Containers

Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

Compose works in all environments: production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

YAML – Yet Another Markup Language

- Whitespace matters!
- **Tabs are not allowed** as indentation for YAML. **Use two spaces** instead.
- Indentation of whitespace is used to denote structure.

- List members are denoted by a leading hyphen (**-**).
- YAML is case sensitive
- The files should have **.yaml** as the extension

```
version: '3.2'

services:
  iris:
    image: intersystems/iris:2023.1.0.299.0
    restart: on-failure
    cpu_percent: 25

    ports:
    - "9091:1972"   # superserver default port
    - "9092:52773"  # webserver port

    volumes:
    - common_shared:/common_shared
    - type: bind
      source: ./advanced_analytics/shared/
      target: /shared

    environment:
    - IRIS_MASTER_HOST=datalake
    - IRIS_MASTER_PORT=51773
    - IRIS_MASTER_NAMESPACE=APP

    networks:
    - datalake-tier
```

2 spaces

list

Docker Compose Example

```yaml
version: '3.7'
services:
  # iris container
  iris:
    image: containers.intersystems.com/intersystems/iris-community:2023.1.0.229.0
    ports:
    - "8881:1972"
    volumes:
    - type: bind
    source: ./durable/compose
    target: /iris-shared
    . . .

  # web gateway container
  webgateway:
    image: containers.intersystems.com/intersystems/webgateway:2023.1.0.229.0
    ports:
    - "8882:80"
    . . .
```

A Service is an abstract definition of a computing resource within an application

Docker Compose Example

```
version: '3.7'
services:
  # iris container
  iris:
    image: containers.intersystems.com/intersystems/iris-community:2023.1.0.229.0
    ports:
    - "8881:1972"
    volumes:
    - type: bind
    source: ./durable/compose
    target: /iris-shared
    . . .

  # web gateway container
  webgateway:
    image: containers.intersystems.com/intersystems/webgateway:2023.1.0.229.0
    ports:
    - "8882:80"
    . . .
```

The service definition contains the configuration that is applied to each container started for that service

Docker Compose Example

```yaml
version: '3.7'
services:
  # iris container
  iris:
    image: containers.intersystems.com/intersystems/iris-community:2023.1.0.229.0
    ports:
    - "8881:1972"
    volumes:
    - type: bind
    source: ./durable/compose
    target: /iris-shared
    . . .

  # web gateway container
  webgateway:
    image: containers.intersystems.com/intersystems/webgateway:2023.1.0.229.0
    ports:
    - "8882:80"
    . . .
```

image specifies the image to start the container from

Docker Compose Example

```yaml
version: '3.7'
services:
  # iris container
  iris:
    image: containers.intersystems.com/intersystems/iris-community:2023.1.0.229.0
    ports:
    - "8881:1972"
    volumes:
    - type: bind
    source: ./durable/compose
    target: /iris-shared
    . . .

  # web gateway container
  webgateway:
    image: containers.intersystems.com/intersystems/webgateway:2023.1.0.229.0
    ports:
    - "8882:80"
    . . .
```

Additional set of runtime arguments

Using Docker Compose

Process for using Compose:

- Define your app's environment with a Dockerfile so it can be reproduced anywhere.
- Define the services that make up your app in docker-compose.ymal so they can be run together in an isolated environment.
- Run docker compose up, command starts and runs your entire app.

Can create and destroy these environments in just a few commands:

```
$ docker compose up -d
$./run_tests
$ docker compose down
```

Exercises

- Try Exercises 4.4, 5 & 6
- Ask for help if you need it
- Next presentation:
  - **4:30 PM**

Putting It All Together

Demo Coffee Shop Application

Coffee - It’s The Next Big Thing!

What’s this about?

- Use what we learned so far to build a container from a basic IRIS application
- Familiarize yourself with working with your container
- Push and pull your container from a container registry
- Build & Run a multi-container application with docker-compose

- Bonus: Learn how to use VS Code & DBeaver with your container

Exercises

- Try Exercise 7 – Build Your Own Containerized Application
- Then, try the BONUS Exercise
- Ask for help if you need it

Next Steps: Containerize your own applications!

- Do you feel comfortable doing that now?
- If you have questions, please ask!