Step 1:

File: cmd/dockerd/docker.go

```
func main() {
    if reexec.Init() {
        return
    }

    // initial log formatting; this setting is updated after the daemon
configuration is loaded.
    logrus.SetFormatter(&logrus.TextFormatter{
        TimestampFormat: jsonmessage.RFC3339NanoFixed,
        FullTimestamp:   true,
    })

    // Set terminal emulation based on platform as required.
    _, stdout, stderr := term.StdStreams()

    initLogging(stdout, stderr)
    configureGRPCLog()

    onError := func(err error) {
        fmt.Fprintf(stderr, "%s\n", err)
        os.Exit(1)
    }

    cmd, err := newDaemonCommand()
    if err != nil {
        onError(err)
    }
    cmd.SetOut(stdout)
    if err := cmd.Execute(); err != nil {
        onError(err)
    }
}
```

What Does it do?

```
1. Check if the process is a re-executed process.
2. Set the log format.
3. Set the terminal emulation.
4. Initialize the logging.
5. Configure the GRPC log.
6. Create a new daemon command.
7. Execute the command.
```

Step 2:

File: cmd/dockerd/docker.go

```
func newDaemonCommand() (*cobra.Command, error) {
    cfg, err := config.New()
    if err != nil {
        return nil, err
    }
    opts := newDaemonOptions(cfg)

    cmd := &cobra.Command{
        Use:           "dockerd [OPTIONS]",
        Short:         "A self-sufficient runtime for containers.",
        SilenceUsage:  true,
        SilenceErrors: true,
        Args:          cli.NoArgs,
        RunE: func(cmd *cobra.Command, args []string) error {
            opts.flags = cmd.Flags()
            return runDaemon(opts)
        },
        DisableFlagsInUseLine: true,
        Version:               fmt.Sprintf("%s, build %s",
dockerversion.Version, dockerversion.GitCommit),
    }
    cli.SetupRootCommand(cmd)

    flags := cmd.Flags()
    flags.BoolP("version", "v", false, "Print version information and
quit")
    defaultDaemonConfigFile, err := getDefaultDaemonConfigFile()
    if err != nil {
        return nil, err
    }
    flags.StringVar(&opts.configFile, "config-file",
defaultDaemonConfigFile, "Daemon configuration file")
    configureCertsDir()
    opts.installFlags(flags)
    if err := installConfigFlags(opts.daemonConfig, flags); err != nil {
        return nil, err
    }
    installServiceFlags(flags)

    return cmd, nil
}
```

What Does it do?

```
1. Create a new daemon command.
2. Setup the root command.
3. Set the flags.
```

```
4. Install the flags.
5. Install the service flags.
```

Step 3:

File: cmd/dockerd/docker_unix.go

```
func runDaemon(opts *daemonOptions) error {
    daemonCli := NewDaemonCli()
    return daemonCli.start(opts)
}
```

What Does it do?

```
1. Create a new daemon cli.
2. Start the daemon cli.
```

Step 4:

File: cmd/dockerd/daemon.go

```
func (cli *DaemonCli) start(opts *daemonOptions) (err error) {
    if cli.Config, err = loadDaemonCliConfig(opts); err != nil {
        return err
    }
    if err := checkDeprecatedOptions(cli.Config); err != nil {
        return err
    }

    serverConfig, err := newAPIServerConfig(cli.Config)
    if err != nil {
        return err
    }

    if opts.Validate {
        // If config wasn't OK we wouldn't have made it this far.
        _, _ = fmt.Fprintln(os.Stderr, "configuration OK")
        return nil
    }

    configureProxyEnv(cli.Config)
    configureDaemonLogs(cli.Config)

    logrus.Info("Starting up")

    cli.configFile = &opts.configFile
```

```
    cli.flags = opts.flags

    if cli.Config.Debug {
        debug.Enable()
    }

    if cli.Config.Experimental {
        logrus.Warn("Running experimental build")
    }

    if cli.Config.IsRootless() {
        logrus.Warn("Running in rootless mode. This mode has feature
limitations.")
    }
    if rootless.RunningWithRootlessKit() {
        logrus.Info("Running with RootlessKit integration")
        if !cli.Config.IsRootless() {
            return fmt.Errorf("rootless mode needs to be enabled for
running with RootlessKit")
        }
    }

    // return human-friendly error before creating files
    if runtime.GOOS == "linux" && os.Geteuid() != 0 {
        return fmt.Errorf("dockerd needs to be started with root
privileges. To run dockerd in rootless mode as an unprivileged user, see
https://docs.docker.com/go/rootless/")
    }

    if err := setDefaultUmask(); err != nil {
        return err
    }

    // Create the daemon root before we create ANY other files (PID, or
migrate keys)
    // to ensure the appropriate ACL is set (particularly relevant on
Windows)
    if err := daemon.CreateDaemonRoot(cli.Config); err != nil {
        return err
    }

    if err := system.MkdirAll(cli.Config.ExecRoot, 0700); err != nil {
        return err
    }

    potentiallyUnderRuntimeDir := []string{cli.Config.ExecRoot}

    if cli.Pidfile != "" {
        pf, err := pidfile.New(cli.Pidfile)
        if err != nil {
            return errors.Wrap(err, "failed to start daemon")
        }
        potentiallyUnderRuntimeDir = append(potentiallyUnderRuntimeDir,
cli.Pidfile)
```

```
        defer func() {
            if err := pf.Remove(); err != nil {
                logrus.Error(err)
            }
        }()
    }

    if cli.Config.IsRootless() {
        // Set sticky bit if XDG_RUNTIME_DIR is set && the file is actually
under XDG_RUNTIME_DIR
        if _, err :=
homedir.StickRuntimeDirContents(potentiallyUnderRuntimeDir); err != nil {
            // StickRuntimeDirContents returns nil error if XDG_RUNTIME_DIR
is just unset
            logrus.WithError(err).Warn("cannot set sticky bit on files
under XDG_RUNTIME_DIR")
        }
    }

    cli.api = apiserver.New(serverConfig)

    hosts, err := loadListeners(cli, serverConfig)
    if err != nil {
        return errors.Wrap(err, "failed to load listeners")
    }

    ctx, cancel := context.WithCancel(context.Background())
    waitForContainerDShutdown, err := cli.initContainerD(ctx)
    if waitForContainerDShutdown != nil {
        defer waitForContainerDShutdown(10 * time.Second)
    }
    if err != nil {
        cancel()
        return err
    }
    defer cancel()

    stopc := make(chan bool)
    defer close(stopc)

    trap.Trap(func() {
        cli.stop()
        <-stopc // wait for daemonCli.start() to return
    }, logrus.StandardLogger())

    // Notify that the API is active, but before daemon is set up.
    preNotifyReady()

    pluginStore := plugin.NewStore()

    if err := cli.initMiddlewares(cli.api, serverConfig, pluginStore); err
!= nil {
        logrus.Fatalf("Error creating middlewares: %v", err)
    }
```

```
    d, err := daemon.NewDaemon(ctx, cli.Config, pluginStore)
    if err != nil {
        return errors.Wrap(err, "failed to start daemon")
    }

    d.StoreHosts(hosts)

    // validate after NewDaemon has restored enabled plugins. Don't change
order.
    if err := validateAuthzPlugins(cli.Config.AuthorizationPlugins,
pluginStore); err != nil {
        return errors.Wrap(err, "failed to validate authorization plugin")
    }

    cli.d = d

    if err := startMetricsServer(cli.Config.MetricsAddress); err != nil {
        return errors.Wrap(err, "failed to start metrics server")
    }

    c, err := createAndStartCluster(cli, d)
    if err != nil {
        logrus.Fatalf("Error starting cluster component: %v", err)
    }

    // Restart all autostart containers which has a swarm endpoint
    // and is not yet running now that we have successfully
    // initialized the cluster.
    d.RestartSwarmContainers()

    logrus.Info("Daemon has completed initialization")

    routerOptions, err := newRouterOptions(cli.Config, d)
    if err != nil {
        return err
    }
    routerOptions.api = cli.api
    routerOptions.cluster = c

    initRouter(routerOptions)

    go d.ProcessClusterNotifications(ctx, c.GetWatchStream())

    cli.setupConfigReloadTrap()

    // The serve API routine never exits unless an error occurs
    // We need to start it as a goroutine and wait on it so
    // daemon doesn't exit
    serveAPIWait := make(chan error)
    go cli.api.Wait(serveAPIWait)

    // after the daemon is done setting up we can notify systemd api
    notifyReady()
```

```
    // Daemon is fully initialized and handling API traffic
    // Wait for serve API to complete
    errAPI := <-serveAPIWait
    c.Cleanup()

    // notify systemd that we're shutting down
    notifyStopping()
    shutdownDaemon(d)

    // Stop notification processing and any background processes
    cancel()

    if errAPI != nil {
        return errors.Wrap(errAPI, "shutting down due to ServeAPI error")
    }

    logrus.Info("Daemon shutdown complete")
    return nil
}
```

What Does it do?

```
1. Create a new daemon cli.
2. Load the daemon cli config.
3. Check the deprecated options.
4. Configure the proxy env.
5. Configure the daemon logs.
6. Enable the debug mode.
7. Check if the daemon is running in rootless mode.
8. Create the daemon root.
9. Create the daemon exec root.
10. Create the pid file.
11. Create the daemon api.
12. Load the listeners.
13. Initialize the containerd.
14. Initialize the middlewares.
15. Create the daemon.
16. Validate the authorization plugins.
17. Start the metrics server.
18. Create and start the cluster.
19. Restart the swarm containers.
20. Process the cluster notifications.
21. Setup the config reload trap.
22. Wait for the api to complete.
23. Notify the systemd api.
24. Shutdown the daemon.
25. Notify the systemd that the daemon is shutting down.
26. Stop the notification processing and background processes.
27. Check if there is any error in the api.
28. Shutdown the daemon.
```

Step 5:

File: cmd/dockerd/daemon.go

```
func initRouter(opts routerOptions) {
    decoder := runconfig.ContainerDecoder{
        GetSysInfo: func() *sysinfo.SysInfo {
            return opts.daemon.RawSysInfo()
        },
    }

    routers := []router.Router{
        // we need to add the checkpoint router before the container router
or the DELETE gets masked
        checkpointrouter.NewRouter(opts.daemon, decoder),
        container.NewRouter(opts.daemon, decoder,
opts.daemon.RawSysInfo().CgroupUnified),
        image.NewRouter(
            opts.daemon.ImageService(),
            opts.daemon.ReferenceStore,
            opts.daemon.ImageService().DistributionServices().ImageStore,
            opts.daemon.ImageService().DistributionServices().LayerStore,
        ),
        systemrouter.NewRouter(opts.daemon, opts.cluster, opts.buildkit,
opts.features),
        volume.NewRouter(opts.daemon.VolumesService(), opts.cluster),
        build.NewRouter(opts.buildBackend, opts.daemon, opts.features),
        sessionrouter.NewRouter(opts.sessionManager),
        swarmrouter.NewRouter(opts.cluster),
        pluginrouter.NewRouter(opts.daemon.PluginManager()),
        distributionrouter.NewRouter(opts.daemon.ImageService()),
    }

    grpcBackends := []grpcrouter.Backend{}
    for _, b := range []interface{}{opts.daemon, opts.buildBackend} {
        if b, ok := b.(grpcrouter.Backend); ok {
            grpcBackends = append(grpcBackends, b)
        }
    }
    if len(grpcBackends) > 0 {
        routers = append(routers, grpcrouter.NewRouter(grpcBackends...))
    }

    if opts.daemon.NetworkControllerEnabled() {
        routers = append(routers, network.NewRouter(opts.daemon,
opts.cluster))
    }

    if opts.daemon.HasExperimental() {
        for _, r := range routers {
```

```
            for _, route := range r.Routes() {
                if experimental, ok := route.(router.ExperimentalRoute); ok
{
                    experimental.Enable()
                }
            }
        }
    }

    opts.api.InitRouter(routers...)
}
```

What Does it do?

```
## TODO
```

Step 6:

File: api/server/router/container/container.go

```
// NewRouter initializes a new container router
func NewRouter(b Backend, decoder httputils.ContainerDecoder, cgroup2 bool)
router.Router {
    r := &containerRouter{
        backend: b,
        decoder: decoder,
        cgroup2: cgroup2,
    }
    r.initRoutes()
    return r
}
```

What Does it do?

```
1. Create a new container router.
2. Initialize the routes.
```

Step 7:

File: api/server/router/container/container.go

```
func (r *containerRouter) initRoutes() {
    r.routes = []router.Route{
        // HEAD
```

```
        router.NewHeadRoute("/containers/{name:.*}/archive",
r.headContainersArchive),
        // GET
        router.NewGetRoute("/containers/json", r.getContainersJSON),
        router.NewGetRoute("/containers/{name:.*}/export",
r.getContainersExport),
        router.NewGetRoute("/containers/{name:.*}/changes",
r.getContainersChanges),
        router.NewGetRoute("/containers/{name:.*}/json",
r.getContainersByName),
        router.NewGetRoute("/containers/{name:.*}/top",
r.getContainersTop),
        router.NewGetRoute("/containers/{name:.*}/logs",
r.getContainersLogs),
        router.NewGetRoute("/containers/{name:.*}/stats",
r.getContainersStats),
        router.NewGetRoute("/containers/{name:.*}/attach/ws",
r.wsContainersAttach),
        router.NewGetRoute("/exec/{id:.*}/json", r.getExecByID),
        router.NewGetRoute("/containers/{name:.*}/archive",
r.getContainersArchive),
        // POST
        router.NewPostRoute("/containers/create", r.postContainersCreate),
        router.NewPostRoute("/containers/{name:.*}/kill",
r.postContainersKill),
        router.NewPostRoute("/containers/{name:.*}/pause",
r.postContainersPause),
        router.NewPostRoute("/containers/{name:.*}/unpause",
r.postContainersUnpause),
        router.NewPostRoute("/containers/{name:.*}/restart",
r.postContainersRestart),
        router.NewPostRoute("/containers/{name:.*}/start",
r.postContainersStart),
        router.NewPostRoute("/containers/{name:.*}/stop",
r.postContainersStop),
        router.NewPostRoute("/containers/{name:.*}/wait",
r.postContainersWait),
        router.NewPostRoute("/containers/{name:.*}/resize",
r.postContainersResize),
        router.NewPostRoute("/containers/{name:.*}/attach",
r.postContainersAttach),
        router.NewPostRoute("/containers/{name:.*}/copy",
r.postContainersCopy), // Deprecated since 1.8 (API v1.20), errors out
since 1.12 (API v1.24)
        router.NewPostRoute("/containers/{name:.*}/exec",
r.postContainerExecCreate),
        router.NewPostRoute("/exec/{name:.*}/start",
r.postContainerExecStart),
        router.NewPostRoute("/exec/{name:.*}/resize",
r.postContainerExecResize),
        router.NewPostRoute("/containers/{name:.*}/rename",
r.postContainerRename),
        router.NewPostRoute("/containers/{name:.*}/update",
r.postContainerUpdate),
```

```
        router.NewPostRoute("/containers/prune", r.postContainersPrune),
        router.NewPostRoute("/commit", r.postCommit),
        // PUT
        router.NewPutRoute("/containers/{name:.*}/archive",
r.putContainersArchive),
        // DELETE
        router.NewDeleteRoute("/containers/{name:.*}", r.deleteContainers),
    }
}
```

What Does it do?

```
## TODO
```

Step 8:

File: api/server/router/container/container_routes.go

```
func (s *containerRouter) postContainersCreate(ctx context.Context, w
http.ResponseWriter, r *http.Request, vars map[string]string) error {
    if err := httputils.ParseForm(r); err != nil {
        return err
    }
    if err := httputils.CheckForJSON(r); err != nil {
        return err
    }

    name := r.Form.Get("name")

    config, hostConfig, networkingConfig, err :=
s.decoder.DecodeConfig(r.Body)
    if err != nil {
        if errors.Is(err, io.EOF) {
            return errdefs.InvalidParameter(errors.New("invalid JSON: got
EOF while reading request body"))
        }
        return err
    }
    version := httputils.VersionFromContext(ctx)
    adjustCPUShares := versions.LessThan(version, "1.19")

    // When using API 1.24 and under, the client is responsible for
removing the container
    if hostConfig != nil && versions.LessThan(version, "1.25") {
        hostConfig.AutoRemove = false
    }

    if hostConfig != nil && versions.LessThan(version, "1.40") {
```

```
        // Ignore BindOptions.NonRecursive because it was added in API
1.40.
        for _, m := range hostConfig.Mounts {
            if bo := m.BindOptions; bo != nil {
                bo.NonRecursive = false
            }
        }
        // Ignore KernelMemoryTCP because it was added in API 1.40.
        hostConfig.KernelMemoryTCP = 0

        // Older clients (API < 1.40) expects the default to be shareable,
make them happy
        if hostConfig.IpcMode.IsEmpty() {
            hostConfig.IpcMode = container.IPCModeShareable
        }
    }
    if hostConfig != nil && versions.LessThan(version, "1.41") &&
!s.cgroup2 {
        // Older clients expect the default to be "host" on cgroup v1 hosts
        if hostConfig.CgroupnsMode.IsEmpty() {
            hostConfig.CgroupnsMode = container.CgroupnsModeHost
        }
    }

    if hostConfig != nil && versions.LessThan(version, "1.42") {
        for _, m := range hostConfig.Mounts {
            // Ignore BindOptions.CreateMountpoint because it was added in
API 1.42.
            if bo := m.BindOptions; bo != nil {
                bo.CreateMountpoint = false
            }

            // These combinations are invalid, but weren't validated in API
< 1.42.
            // We reset them here, so that validation doesn't produce an
error.
            if o := m.VolumeOptions; o != nil && m.Type != mount.TypeVolume
{
                m.VolumeOptions = nil
            }
            if o := m.TmpfsOptions; o != nil && m.Type != mount.TypeTmpfs {
                m.TmpfsOptions = nil
            }
            if bo := m.BindOptions; bo != nil {
                // Ignore BindOptions.CreateMountpoint because it was added
in API 1.42.
                bo.CreateMountpoint = false
            }
        }
    }

    if hostConfig != nil && versions.GreaterThanOrEqualTo(version, "1.42")
{
        // Ignore KernelMemory removed in API 1.42.
```

```
        hostConfig.KernelMemory = 0
        for _, m := range hostConfig.Mounts {
            if o := m.VolumeOptions; o != nil && m.Type != mount.TypeVolume
{
                return errdefs.InvalidParameter(fmt.Errorf("VolumeOptions
must not be specified on mount type %q", m.Type))
            }
            if o := m.BindOptions; o != nil && m.Type != mount.TypeBind {
                return errdefs.InvalidParameter(fmt.Errorf("BindOptions
must not be specified on mount type %q", m.Type))
            }
            if o := m.TmpfsOptions; o != nil && m.Type != mount.TypeTmpfs {
                return errdefs.InvalidParameter(fmt.Errorf("TmpfsOptions
must not be specified on mount type %q", m.Type))
            }
        }
    }

    if hostConfig != nil && runtime.GOOS == "linux" &&
versions.LessThan(version, "1.42") {
        // ConsoleSize is not respected by Linux daemon before API 1.42
        hostConfig.ConsoleSize = [2]uint{0, 0}
    }

    var platform *specs.Platform
    if versions.GreaterThanOrEqualTo(version, "1.41") {
        if v := r.Form.Get("platform"); v != "" {
            p, err := platforms.Parse(v)
            if err != nil {
                return errdefs.InvalidParameter(err)
            }
            platform = &p
        }
    }

    if hostConfig != nil && hostConfig.PidsLimit != nil &&
*hostConfig.PidsLimit <= 0 {
        // Don't set a limit if either no limit was specified, or
"unlimited" was
        // explicitly set.
        // Both `0` and `-1` are accepted as "unlimited", and historically
any
        // negative value was accepted, so treat those as "unlimited" as
well.
        hostConfig.PidsLimit = nil
    }

    ccr, err := s.backend.ContainerCreate(types.ContainerCreateConfig{
        Name:             name,
        Config:           config,
        HostConfig:       hostConfig,
        NetworkingConfig: networkingConfig,
        AdjustCPUShares:  adjustCPUShares,
        Platform:         platform,
```

```
    })
    if err != nil {
        return err
    }

    return httputils.WriteJSON(w, http.StatusCreated, ccr)
}
```

What Does it do?

```
## TODO
```

Step 9:

File: daemon/create.go

```
func (daemon *Daemon) containerCreate(opts createOpts)
(containertypes.CreateResponse, error) {
    start := time.Now()
    if opts.params.Config == nil {
        return containertypes.CreateResponse{},
errdefs.InvalidParameter(errors.New("Config cannot be empty in order to
create a container"))
    }

    warnings, err := daemon.verifyContainerSettings(opts.params.HostConfig,
opts.params.Config, false)
    if err != nil {
        return containertypes.CreateResponse{Warnings: warnings},
errdefs.InvalidParameter(err)
    }

    if opts.params.Platform == nil && opts.params.Config.Image != "" {
        if img, _ := daemon.imageService.GetImage(opts.params.Config.Image,
opts.params.Platform); img != nil {
            p := maximumSpec()
            imgPlat := v1.Platform{
                OS:           img.OS,
                Architecture: img.Architecture,
                Variant:      img.Variant,
            }

            if !images.OnlyPlatformWithFallback(p).Match(imgPlat) {
                warnings = append(warnings, fmt.Sprintf("The requested
image's platform (%s) does not match the detected host platform (%s) and no
specific platform was requested", platforms.Format(imgPlat),
platforms.Format(p)))
            }
        }
```

```
    }

    err = verifyNetworkingConfig(opts.params.NetworkingConfig)
    if err != nil {
        return containertypes.CreateResponse{Warnings: warnings},
errdefs.InvalidParameter(err)
    }

    if opts.params.HostConfig == nil {
        opts.params.HostConfig = &containertypes.HostConfig{}
    }
    err = daemon.adaptContainerSettings(opts.params.HostConfig,
opts.params.AdjustCPUShares)
    if err != nil {
        return containertypes.CreateResponse{Warnings: warnings},
errdefs.InvalidParameter(err)
    }

    ctr, err := daemon.create(opts)
    if err != nil {
        return containertypes.CreateResponse{Warnings: warnings}, err
    }
    containerActions.WithValues("create").UpdateSince(start)

    if warnings == nil {
        warnings = make([]string, 0) // Create an empty slice to avoid
https://github.com/moby/moby/issues/38222
    }

    return containertypes.CreateResponse{ID: ctr.ID, Warnings: warnings},
nil
}
```

What Does it do?

```
## TODO
```

Step 10:

File: daemon/create.go

```
func (daemon *Daemon) verifyContainerSettings(hostConfig
*containertypes.HostConfig, config *containertypes.Config, update bool)
(warnings []string, err error) {
    // First perform verification of settings common across all platforms.
    if err = validateContainerConfig(config); err != nil {
        return warnings, err
    }
```

```
    if err := validateHostConfig(hostConfig); err != nil {
        return warnings, err
    }

    // Now do platform-specific verification
    warnings, err = verifyPlatformContainerSettings(daemon, hostConfig,
update)
    for _, w := range warnings {
        logrus.Warn(w)
    }
    return warnings, err
}
```

What Does it do?

```
## TODO
```