Detailed Breakdown of Each Step:

1. Initialize Base Guice Injector (BASE_INJECTOR = createBaseInjector();)

Purpose: This is the very first step in setting up the Dependency Injection (DI) container. The "base" injector provides fundamental, framework-level components that are always needed, regardless of the specific service.

What it does: The createBaseInjector() method (not shown, but implied) would bind core modules like ConfigServiceModule (responsible for loading configuration files), MarshallerModule (for JSON/data serialization), and other foundational utilities.

Result: A minimal, but functional, Guice injector is available, capable of providing basic AppOps utilities.

2. Load Core Service Configuration (ServiceConfigurationLoader configLoader = ...; ServiceConfiguration serviceConfig = configLoader.getCoreConfig();)

Purpose: To locate and parse the most fundamental configuration file for this specific service. This file typically defines the service's name, deployment mode, and primary profile.

ServiceConfigurationLoader: This class is instantiated with the BASE_INJECTOR (so it can use core utilities) and serviceArgs (to know the desired profile, config file paths, etc.).

getCoreConfig(): This method reads the main configuration file (e.g., app.yml or core.yml for the overall service) and extracts critical information like the serviceName.

serviceName = serviceConfig.getServiceName();: The service's unique name is extracted and stored, primarily for logging and error reporting throughout the startup process.

3. Set Current Service Context (Service service = new Service(serviceName); this.service = service;)

Purpose: To create a simple Service object that holds the name of the currently running service. This object can then be injected elsewhere using the @CurrentService annotation (as seen in

ApiRestInvoker), allowing components to know which service they belong to.

4. Load All Application-Specific Configurations (Map<String, ServiceConfiguration> serviceConfigMap = configLoader.loadServiceConfiguration();)

Purpose: This is where all the application-specific configurations (like UserServiceConfig, GithubAuthConfig, PageRestrictionConfig, AutoLoginConfig that you've shown) are discovered and loaded.

loadServiceConfiguration(): This method likely scans predefined configuration locations (e.g., config/ directory) for .yml or .properties files relevant to the service and its enabled sub-components. It parses these files into a Map where keys might be service names or configuration domains, and values are ServiceConfiguration objects containing the parsed data.

Result: All necessary configuration data is now available in raw, parsed form, ready to be bound to your @Config annotated POJOs.

5. (TODO) Legacy System Property Setup (System.setProperty(...))

Purpose: These lines are a remnant of older configuration patterns. They set global system properties for currentProfile and baseUrl.

"TODO: these two should be phased out": This comment is significant. It indicates that setting system properties for configuration is considered less ideal in modern AppOps (or Guice-based) applications, as it relies on global state and can be less testable. The framework aims to move towards pure dependency injection for all configuration access.

6. Initialize Main Service Guice Injector (SERVICE_INJECTOR = createServiceInjector(BASE_INJECTOR, serviceConfigMap, serviceConfig);)

Purpose: This is the most important step for setting up the full Dependency Injection graph for your application. This injector will be responsible for creating and wiring all your service components (handlers, converters, DAOs, etc.) and injecting the appropriate configuration objects.

createServiceInjector():

- It uses the BASE_INJECTOR as a parent, inheriting all core framework bindings.

- It takes the serviceConfigMap (all loaded raw configurations) and serviceConfig (the core service configuration).

- Crucially: This method will:

  - Read the serviceConfigMap to identify which specific Guice modules need to be loaded for this service (e.g., JettyWebServiceModule if it's a web service, or custom modules defined in your config files like TeacherServiceModule).

  - Bind your @Config annotated POJOs: It's at this stage that the raw configuration data from serviceConfigMap is used to instantiate and populate instances of your UserServiceConfig, GithubAuthConfig, etc., and bind them into the SERVICE_INJECTOR so they can be injected wherever UserServiceConfig (with @Config) is requested.

Result: A fully configured Guice SERVICE_INJECTOR is ready. Any class requesting dependencies (including @Config objects) can now be instantiated by this injector.

7. Set Current Deployment Context for Injection (CurrentDeploymentProvider currentDeployment = ...; currentDeployment.setCurrentServiceConfig(serviceConfig);)

Purpose: To make the serviceConfig (the core configuration for the current running service) available for injection into other components using a Guice Provider. This provides a consistent way to access details about the current deployment.

8. Prepare for Web Service Deployment (ServiceServletContextListener.setInjector(SERVICE_INJECTOR);)

Purpose: This line integrates the AppOps Guice injector with a standard Servlet container (like Jetty, which will be used later).

ServiceServletContextListener: This is likely a ServletContextListener implementation that Jetty (or any servlet container) would pick up. By setting the SERVICE_INJECTOR on it, any servlets, filters, or other web components defined in your web application can then use this injector to get their dependencies. This is how AppOps bridges its Guice DI container with the servlet environment.

9. Determine and Execute Run Mode (Jetty vs. No-Jetty)

This conditional block decides whether the service will run as a web application (with an embedded Jetty server) or as a non-web service (e.g., a background worker, a console application).

if (serviceArgs.getJettyRunMode()) (Jetty Mode):

- ServiceJettyLauncher appLauncher = SERVICE_INJECTOR.getInstance(ServiceJettyLauncher.class);: Retrieves the Jetty launcher component from the now-ready SERVICE_INJECTOR.
- appLauncher.getJettyContainer().readyServer(...): Configures the embedded Jetty server, setting its listening port (serviceConfig.getWebConfig().getPort()) and other Jetty-specific settings (serviceConfig.getJettyConfig()).
- appLauncher.deployServiceDirect(serviceConfig);: This step deploys the web application context onto Jetty. This involves registering servlets, filters, and other web-related components defined by your service.
- initializeServices(...): This is a crucial call. Even for a web service, other non-web-related components and modules still need to be initialized. This method (which is a private helper method within ServiceEntryPoint) iterates through the serviceConfigMap and invokes ServiceInitializer classes (like TeacherServiceInitializer) for each identified service/module. This is where:
  - Service metadata is generated.
  - Database schemas are registered (as seen in TeacherServiceInitializer).
  - SessionFactory objects are created and stored (also in TeacherServiceInitializer).
  - Any custom ServiceInitializer logic is executed.
  - The actual binding of raw config data to @Config annotated POJOs happens here.
- appLauncher.startService();: The configured Jetty server is started, making the service accessible via its defined HTTP endpoints.

- appLauncher.joinService();: This call blocks the main thread, preventing the JVM from exiting. It keeps the Jetty server running indefinitely until it receives a shutdown signal.

else (No-Jetty Mode):

- initializeServices(...): If Jetty is not run, the service is likely a background process. However, it still needs to go through the same initialization of its internal components, configurations, and modules. The initializeServices method ensures this setup happens even without a web server. The application will then likely perform its background task and exit, or remain running if it's a listener for other events (e.g., message queues).

10. Log Success

logger.log(Level.ALL, "Service started successfully for " + serviceConfig.getServiceName());

11. Handle and Log Initialization Failure (catch (Exception e))

Purpose: To gracefully handle any exceptions that occur during the complex startup process.

Logging: The exception is logged, providing crucial debugging information.

Re-throw: The exception is re-thrown as an AppEntryPointException, providing a clear indication that the service failed to start from its entry point.

12. Final Logging on Exit (finally { ... })

Purpose: This finally block ensures that a "Exiting app" message is logged, regardless of whether the service started successfully or failed.

Note: For long-running web services, this finally block's "Exiting" message might be slightly misleading. It typically implies the end of the main method's execution. In a true long-running service, the joinService() call keeps the thread alive, and the "Exiting" message would only appear if the service is explicitly shut down. For short-lived processes (like command-line tools or one-off workers), this log makes more sense.