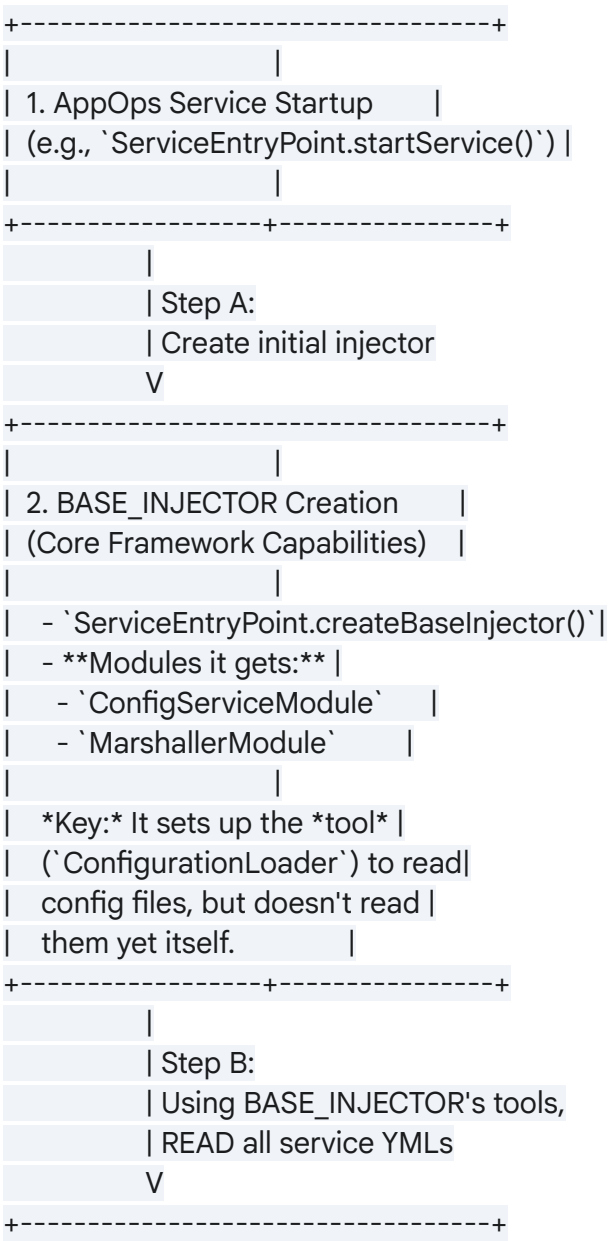
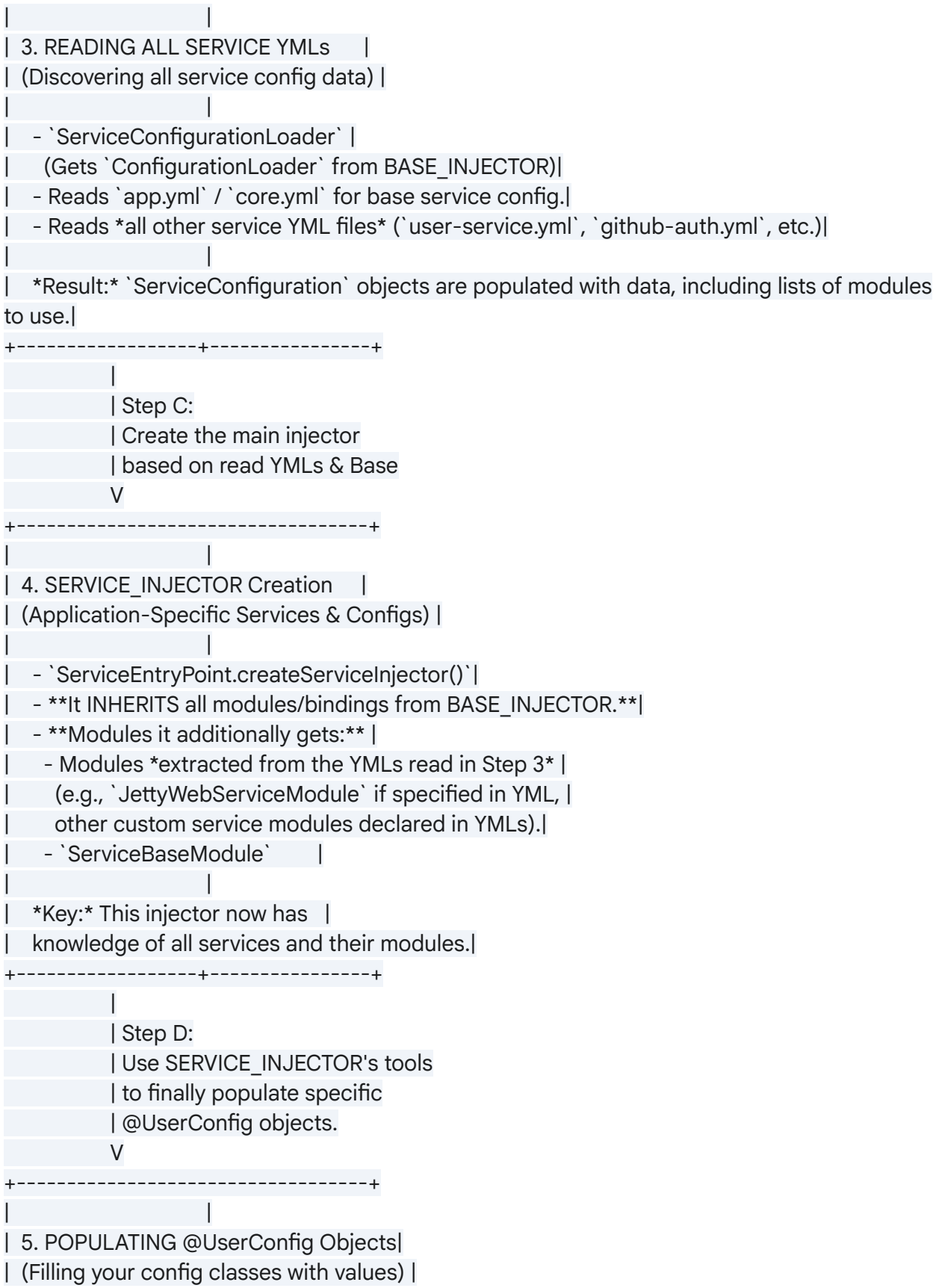
You're asking for a very specific and crucial distinction in the AppOps startup process! Let's clarify how the BASE_INJECTOR and SERVICE_INJECTOR interact with configuration files and modules.

Your understanding is partially correct: the BASE_INJECTOR sets up the *ability* to load configurations, and the SERVICE_INJECTOR uses that ability to bring in the rest.

Here's the refined flow, focusing on your question:




| |
| 3. READING ALL SERVICE YMLs |
| (Discovering all service config data) |
| |
| - `ServiceConfigurationLoader` |
| (Gets `ConfigurationLoader` from BASE_INJECTOR)|
| - Reads `app.yml` / `core.yml` for base service config.|
| - Reads *all other service YML files* (`user-service.yml`, `github-auth.yml`, etc.)|
| |
| *Result:* `ServiceConfiguration` objects are populated with data, including lists of modules to use.|
+----------+----------+
| Step C:
| Create the main injector
| based on read YMLs & Base
V
+----------+
| |
| 4. SERVICE_INJECTOR Creation |
| (Application-Specific Services & Configs) |
| |
| - `ServiceEntryPoint.createServiceInjector()`|
| - **It INHERITS all modules/bindings from BASE_INJECTOR.**|
| - **Modules it additionally gets:** |
| - Modules *extracted from the YMLs read in Step 3* |
| (e.g., `JettyWebServiceModule` if specified in YML, |
| other custom service modules declared in YMLs).|
| - `ServiceBaseModule` |
| |
| *Key:* This injector now has |
| knowledge of all services and their modules.|
+----------+----------+
| Step D:
| Use SERVICE_INJECTOR's tools
| to finally populate specific
| @UserConfig objects.
V
+----------+
| |
| 5. POPULATING @UserConfig Objects|
| (Filling your config classes with values) |

```
|                                  |
|   - `ServiceEntryPoint.initializeServices()`|
|   - Calls `ServiceEntryPoint.loadConfig()` for each service.|
|   - Uses `ConfigurationLoader` (from SERVICE_INJECTOR) to |
|     load specific `UserServiceConfig.java`, `GithubAuthConfig.java`, etc., |
|     instances into `ConfigurationStore` with values from the YMLs. |
|                                  |
|   *Result:* `Authenticator.java` can now `@Inject @Config UserServiceConfig` |
|   and get a fully populated object. |
+----------------+---------------+
                 |
                 | Step E:
                 | Service ready (and Jetty starts if needed)
                 V
+---------------------------------+
|                                 |
|  6. AppOps Service Running      |
|  (All configurations applied)   |
|                                 |
+---------------------------------+
```

**Key Takeaways for your question:**

- **BASE_INJECTOR:** Its main role is to set up the **core framework services**, particularly the **ConfigurationLoader** and Marshaller. It *doesn't* directly read app.yml or configure specific service settings based on it. It provides the *engine* for reading.
- **ServiceConfigurationLoader (using BASE_INJECTOR's ConfigurationLoader):** This is the component that *reads* app.yml/core.yml and all other service-specific .yml files. It parses these files into ServiceConfiguration objects, which contain information about the modules and other settings.
- **SERVICE_INJECTOR:**
  - It **inherits everything from BASE_INJECTOR**. So, it also has access to the ConfigurationLoader.
  - Crucially, it gets **additional modules** (like JettyWebServiceModule or your custom business logic modules) that were **identified and extracted from the .yml files in Step 3**.
  - Once the SERVICE_INJECTOR is created, it's used in **Step 5 (initializeServices()/loadConfig())** to finally take the parsed configuration data and populate the actual @UserConfig objects into the ConfigurationStore, making them available for injection into your services like Authenticator.java.

So, the YML files are *read* after BASE_INJECTOR is set up, and the SERVICE_INJECTOR is then *built* based on the information derived from those YMLs, and *then* used to actually populate

the final configuration classes.