Managing AVS Device SDK components with Manufactory

Sam Schonstal Dec 01, 2020

Share:

Device Makers AVS Device SDK Alexa Voice Service

Blog_Header_Post_Img

The AVS Device SDK introduced a new framework to simplify integration tasks named Manufactory. Manufactory makes it easier to add, remove, and customize SDK components without digging into the core SDK code. Version 1.21 of the SDK now includes a preview application that demonstrates building an Alexa application using Manufactory. This initial release of Manufactory gives you an idea of what's to come in the next few SDK releases. It includes a preview sample application that will evolve over time, and should stabilize by the summer of 2021. This blog outlines how Manufactory works, the reasoning behind the changes, and provides a glimpse of future AVS Device SDK changes.

Evolution of the AVS Device SDK

Over the last three years, the SDK has grown significantly. Version 1.0, released in August 2017, supported five capability agents and was less the 150k lines of code. Version 1.20, released in June 2020, supports 14 Capability Agents and is over 480k lines of code. With this growth, we realized that we needed to make customization and integration of the SDK easier.

The SDK is adaptable. You are free to add, remove, or customize components to suit your needs. Currently, the SDK uses dependency injection to pass objects created by the application into the SDK core. The default client consumes these objects and orchestrates the distribution throughout the SDK. In version 1.20, the DefaultClient::create() function takes 58 input parameters. The application must create and initialize all of these objects, which leads to increased complexity and integration challenges.

Increase Modularization through Componentization

The SDK consists of the following categories of components**.**

- **Core components**, such as the Directive Sequencer and the Focus Manager, are essential for Alexa interaction and should not be removed or modified.
- **Optional components**, such as Smart Home endpoints or Display Cards, to implement features

- **External components**, such as Alexa Calling and Messaging or Multi-Room Music, aren't included by default and require special qualifications to be added.
- **Other features**, such as media players, wake-word engines or authentication methods, have several different implementations to choose from. The SDK provides sample implementations, but you will likely need to customize or replace them for your finished product

Manufactory allows you to select which components you want to add to your device and simplifies the task of adding and removing features or replacing the sample implementations. Componentization is a cleaner way to customize device configuration and improves maintainability by making it easier to upgrade core SDK components. For example, you can update the device SDK core components without having to modify your application.

Customize Your Device with Manufactory

We started adding Manufactory to the SDK in version 1.20. With Manufactory, you don't need to know the details of the component you want to use – you simply request one from Manufactory and use it. Manufactory creates and initializes objects, handles dependencies and manages the lifecycle of these objects. It automates and abstracts this logic from you, eliminating any complex logic required to initialize components in the application. The diagram below illustrates the basic structure of Manufactory.

Figure 1 Structure of manufactory

When you create a Manufactory, you pass it a component. A component is a collection of factories used to create the objects supporting the application. Manufactory creates and manages these objects.

Manufactory exposes a get<>()method. The application can request an object by invoking get<>() with the desired object type. To do this, Manufactory first checks its object cache to see if it has the proper type to return. If it doesn't already have one in its cache, Manufactory invokes the appropriate factory to create the object.

Implementing Manufactory to Manage Your Application

Subscribe

* **Business Email Address:**

* **Country:**

* **Last Name:**

Submit

1. To use Manufactory in your application, follow these steps
2. Provide a factory function for your object
3. Define a component
4. Add your factory to the component
5. Create a Manufactory using that component
6. Acquire objects from the Manufactory

Providing a Factory Function to Support Manufactory

To manage an object with Manufactory, the defining class must:

- Implement the factory pattern with a static create function
- Return a Standard lib std::shared_ptr or std::unique_pt
- Input parameters must be of types that are accessible to the Manufactory

For example, this class provides a factory function that creates and returns a pointer to an *Audioplayer.*

Copy code

```
class AudioPlayer {
public:
    static std::shared_ptr<AudioPlayer> createAudioPlayer();
};
```

Note: I simplified the code in this blog for brevity. Some namespaces and declarations are removed. The intention is to illustrate the principles of Manufactory. For accurate compliable code, see the AVS Device SDK on GitHub.

Define a Component for Use by Manufactory

Manufactory relies on C++ templates for type checking and resolving dependencies. This means Components must specify the C++ Types they export. A Component is a templated Type unique to the Types in its declaration. When declaring a Component, specify all the Types the Component will export to Manufactory—for example, the following SampleAlexaComponent is a unique Type of a Component that exports an AudioPlayer, SpeakerInterface, UIManager, ConfigurationNode and an AuthDelegateInterface.

Copy code

```
SampleAlexaComponent = acsdkManufactory::Component<
   std::shared_ptr<AudioPlayer>,
   std::shared_ptr<SpeakerInterface>,
   std::shared_ptr<UIManage>,
   std::shared_ptr<ConfigurationNode>,
   std::shared_ptr<AuthDelegateInterface>>
```

A factory must be available to the Component to create each of the declared types, or compilation will fail. The Component can, however, hide some types. It can have factories for types used internally that are not exported.

Resolving Dependencies Using Imports

You can build Components from other components, so a parent component can create objects and inject them into the child components that need them. For this process, Components can specify Imports. These imports can be used as input parameters to other factory functions, but they must be available from a parent Component.

The following example shows an AuthorizationDelegateComponent that exports an AuthDelegateInterface, but requires a CBLAuthRequesterInterface, ConfigurationNode, and HttpPostInterface. The AuthorizationDelegateComponent can't be created independently, but it can be included as a part of a parent component that can fulfill these import dependencies.

Copy code

```
AuthorizationDelegateComponent = acsdkManufactory::Component<
    std::shared_ptr<AuthDelegateInterface>,
    Import<std::shared_ptr<CBLAuthRequesterInterface>>,
    Import<std::shared_ptr<ConfigurationNode>>,
    Import<std::unique_ptr< HttpPostInterface>>,
```

Add Factories to Create Objects Exported by Your Component

The ComponentAccumulator class builds Components. It has functions to add various types of objects. Each of these functions returns a ComponentAccumulator so they can be chained together. In the end, the ComponentAccumulator is converted into a Component that is passed to Manufactory. The example below shows a component that satisfies the exports declared by the SampleAlexaComponent declared earlier.

Copy code

```
SampleAlexaComponent getComponent(initParams) {
  return ComponentAccumulator<>()
    .addPrimaryFactory(getAlexaClientSDKInit(initParams))
    .addComponent(acsdkAudioPlayer::getComponent())
    .addComponent(speakerManager::getComponent())
    .addComponent(acsdkShared::getComponent())
    .addComponent(acsdkAuthorizationDelegate::getComponent())
    .addUnloadableFactory(createCBLAuthRequesterInterface)
    .addRetainedFactory(UIManager::create );}
```

In this example, we create the SampleAlexaComponent Component. To do this, several different types of factories and components are added together using the ComponentAccumulator. Descriptions of these items are listed below.

- **addPrimaryFactory()** – Primary Factories are instantiated by Manufactory before anything else. This occurs when the Manufactory is initially created. Here we add, AlexaClientSDKInit.
- **addComponent()** – takes a component as an input parameter. The AudioPlayer and speakerManager are required to satisfy exports declared by SampleAlexaComponent. The acsdkShared component exports the ConfigurationNode and HttpPostInterface to meet the import requirements for the AuthorizationDelegateComponent declared earlier.
- **addUnloadableFactory()** – Unloadable factories are for objects that Manufactory can unload from memory when all references to the pointer are released. The createCBLAuthRequestedInterface exports the CBLAuthRequesterInterface, which is also needed by the AuthorizationDelegateComponent.
- **addRetainedFactory()** – Retained factories tell the Manufactory to keep that object alive as long as the Manufactory is running. The UIManager should always be running.

Note that other 'add' functions are available to add non-shareable objects, object instances, and more. For more details, see the AVS Device SDK on GitHub and the API references.

Creating a Manufactory for Your Application

To create a Manufactory, call the Manufactory create<>() method and pass in the component you wish to use. The only catch is the Manufactory must declare all the Types it exports, and all those Types must be available for export by the Component used to create that Manufactory. The declaration below is almost identical to the Component declaration, except we chose not to export the authentication delegate.

Copy code

```
using SampleAlexaManufactory = acsdkManufactory::Manufactory<
   std::shared_ptr<AudioPlayer>,
   std::shared_ptr<SpeakerInterface>,
   std::shared_ptr<UIManage>,
   std::shared_ptr<ConfigurationNode>>

manufactory = SampleAlexaManufactory::create(SampleAlexaComponent);
```

Use Manufactory to Acquire Objects

Manufactory creates and manages the objects defined in the components it houses. When the application needs an object, it calls the appropriate templated get<>() function on the manufactory instance to acquire a pointer to the object. Manufactory either returns a pointer to an existing object or creates a new one—whatever is appropriate for that object type. In the example below, we acquired a pointer to an instance of the UImanager and use that instance to configure notifications settings.

Copy code

```
uiManager = manufactory->get<std::shared_ptr<UIManager>>();

uiManager->configureSettingsNotifications(params);
```

Replace Components Using Manufactory

Manufactory allows you to pick and choose which components you want to use in your application. You can start with the AlexaClientComponent in the Preview Alexa Client of the AVS Device SDK and add or remove optional components. Some optional components are wrapped in an #ifdef tag in the Preview Alexa Client, allowing addition of the component code with compiler flags demonstrating this flexibility.

It's also straightforward to replace an implementation. For example, you might replace the authorization delegate to use an application-based authentication instead of the Code Based Authentication. To do this, simply replace the factory method passed to addFactory(). As long as it supports the same AuthDelegate interface, no other code in the application needs to change.

The only change required to support MyNewUIManager, other than writing MyNewUIManager, is changing the factory passed to the component through the addRetainedFactory() method. Nothing else in the application needs to change to adapt to the new implementation.

Maintaining Backwards Compatibility with the Sample Application

Manufactory is a significant change to the AVS Device SDK. We added some of these changes in version 1.20 of the SDK. To minimize churn over the next several releases, we added a preview client. The Preview Alexa Client, found under avs-device-sdk/applications, gives you a sneak peek into the Manufactory implementation. Note that the Preview Alexa Client will change dramatically as the implementation evolves. Consider the preview client as an 'alpha' release for you to follow along. As of this writing, it's not recommended to use the previewAlexaClient for production. If you want to build the preview application in version 1.21, Run CMake as you usually would then run 'make PreviewAlexaClient'.

Migrating to Manufactory

The current Sample Application relies on the default client. The 58 input parameters to the DefaultClient::create()function are initialized in the sample application making the sampleApplication.cpp file 1769 lines long. From version 1.20 going forward, this Sample Application will not change much. We will only update the Sample Application to support new Alexa features. We will not make any more changes to the Sample Application to support Manufactory. The changes to support Manufactory are going to be made in the default client.

Figure 2 Legacy application

In the transition phase, there will be both a Sample application, and its replacement, Preview Alexa Client. The sample application will behave the same way as the legacy application. However, the new Preview Application will use Manufactory to initialize objects. It will pass the Manufactory to a new overloaded create() function in the Default client, which will get the objects from the manufactory and pass them to the rest of the application.

Figure 3 Transition phase

Ultimately, the sample app will be eliminated and default client will be replaced by a simpler ‘Alexa Client’ with a single run() function taking one parameter, Manufactory. Manufactory will also be available to the Application to allow access to objects for customization purposes and to interface with any external device components.

Figure 4 Future Vision

Explore the Preview Alexa Client to Learn about Manufactory

Use the Preview Application to explore and follow along as the SDK evolves. While the actual implementation of Manufactory is quite complex, using it is relatively straightforward. You simply pass your objects to Manufactory through components and then use them. As we streamline the interfaces and evolve the implementation, customizing the SDK will become even more manageable. Over the next several releases, you will see additional changes to add modularization and customization to the SDK until we stabilize the implementation in mid-2021. So, don't worry, you don't have to consume any of it yet. The Preview Application is there for you to explore, and we will maintain the current Sample Application as-is for all of 2021. The Preview Application will become the primary Sample Application in 2022.

The AVS Device SDK team is dedicated to helping you easily build Alexa devices. We do this by continuing to iterate on a robust solution that is reliable, customizable and easy to maintain. We look forward to the creative and innovative ways you integrate Alexa into your products. Feel free to provide feedback through your Amazon Point of contact, Github Issues or me @sschonstal.

Back to Top

Alexa Skills Kit

- Alexa Skills Kit
- Learn
- Design
- Build
- Launch

Resources

- Getting Started
- Tutorials
- Documentation
- Developer Forum
- Agencies and Tools

Alexa Voice Service

- Alexa Voice Service
- Learn
- Design
- Build
- Launch

AVS Resources

- Getting Started
- AVS Device SDK
- AVS API
- Dev Kits for AVS

Connected Devices

- Alexa Smart Home
- Alexa Gadgets

Agreements

- Agreements and Terms
- Program Materials License Agreement
- Amazon Developers Services Portal Terms of Use

Blogs

- Alexa Skills Kit Blog
- Device Makers Blog
- AWS Blog
- Alexa Science

Support

- Amazon Developer Support
- Contact Us
- Forums
- Manage Email Preferences

Follow Us: