POS Extensibility

Agenda

- Background Information: What should I know before getting started?
  - TypeScript/JavaScript Modules
  - Extensibility Terminology
- Supported Extension Types: What type of extensions are supported?
- Design Overview: How does POS extensibility work?
  - Discovering Extensions
  - Extension Modules
  - POS API
  - POS UI SDK
- Retail SDK POS Solution Overview: What is included in the Retail SDK?

TypeScript/JavaScript Modules

- In TypeScript/JavaScript modules are small units of independent, reusable code.
- Each module exports specific components which make up it's public interface.
- Modules can import (load) other modules which facilitates code separation and reuse
- The module pattern is widely used throughout the industry with a few different types of modules
  - ES 2015, System, AMD, CommonJS
- TypeScript abstracts the module type which means code can compile to different JavaScript module types depending on the compiler setting
  - POS uses the 'System' module format

POS Extensibility Terminology

- **Extension Point**: A specific location at which POS can extended, or a technique through which new functionality can be added to POS
  - Ex. The PreOperationTrigger invocation point, adding a new view
- **Extension**: An individual component that helps to customize POS by utilizing a specific extension point
  - Ex. A new operation, new view, a PreOperationTrigger implementation
- **Extension Package**: A set of extensions that when combined enable a custom end to end POS scenario
  - Ex. A package that contains an operation that navigates to a new

What type of extensions are supported?

Categorizing POS Extensions

- Two categories of extension types:
  - ***Extend*** existing POS functionality – These extensions modify POS by adding additional functionality to existing pages or workflows
  - ***Create*** new functionality – These extensions supplement POS by introducing new pages or workflows that do not exist “out of the box”

*Extend*ing POS Functionality

- Extending POS Pages
  - Custom App Bar Buttons - Adding custom buttons to the App Bar on select pages
    - Product Details, Customer Details, Product Search, Customer Search, Show Journal
  - Custom Column Sets – The ability to replace the grid columns with a custom column set on select pages
    - Product Search, Customer Search, Show Journal, Inventory Lookup
- Extending POS Workflows
  - Triggers – POS Triggers were ported to the new extensibility model

Create-ing New Functionality

- Custom Controls
- Templated Dialogs
- New Operations
- New Request Handlers for the POS Runtime
- Custom Views

How does POS extensibility work?

Extension Package Discovery

- Each extension package is listed in a JSON file, extensions.json, which specifies the root directory for each extension package
  - Must be in the extensions root directory (%POSRoot%\Extensions)
  - Extension package paths are relative to extensions root directory
  - Used to discover all the extension packages that should be loaded

Extension Discovery

- Each extension package contains a manifest file, manifest.json, which defines the extensions contained in the package and important information about the package
  - Each entry in the manifest corresponds to an extension
  - POS uses the manifest to load the extensions
  - Manifest layout must match the predefined schema

Extension Modules

- Each extension is comprised of one or more modules
- Modules are loaded using the paths provided in the manifest file
- Each extension module must have only a default export
- Extension modules can import/load modules from the POS extensibility libraries to the required components
- Extension classes must inherit from base classes exposed from the POS libraries

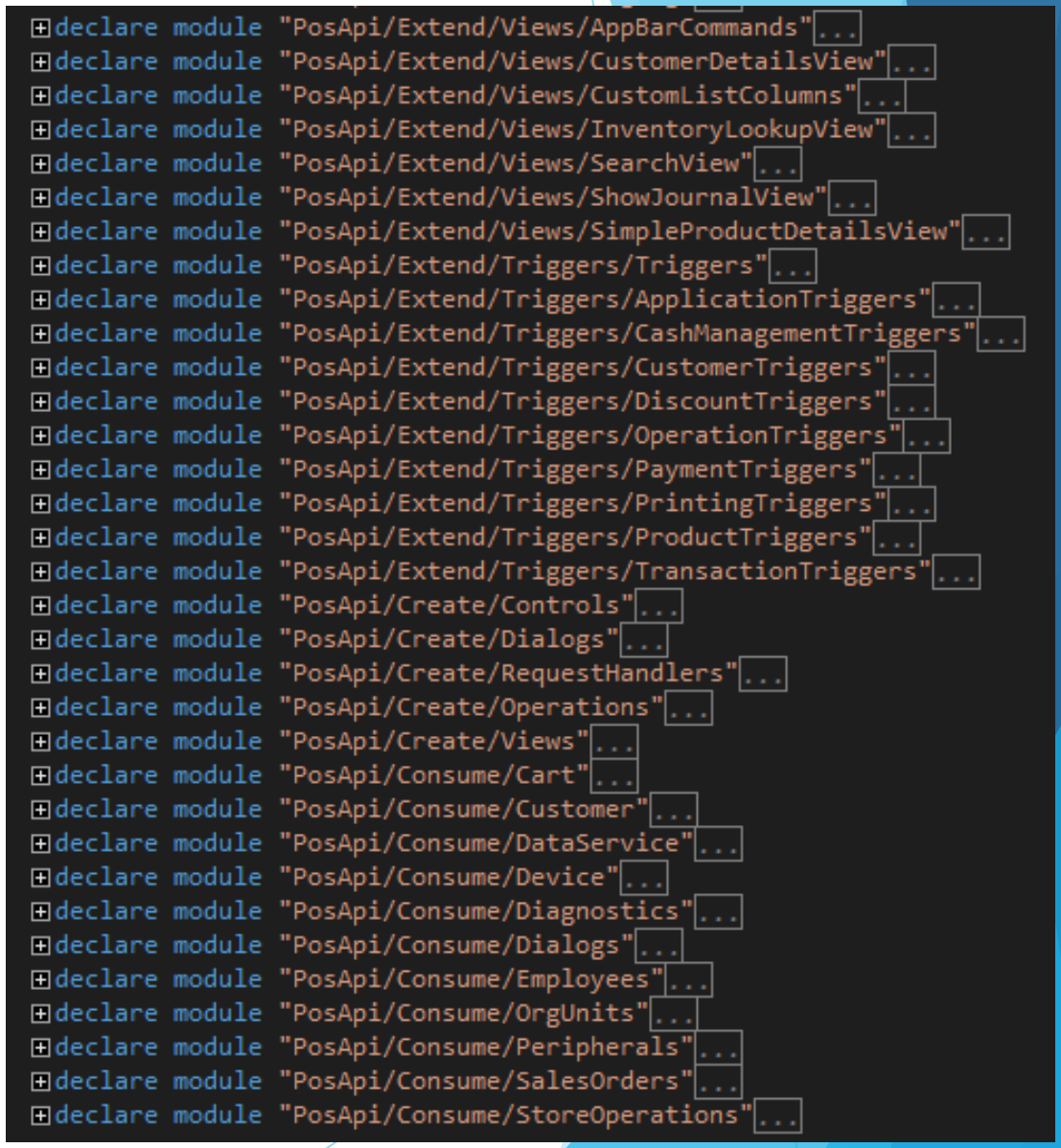
```
import * as Triggers from "PosApi/Extend/Triggers/ApplicationTriggers";

/**
 * Example implementation of an ApplicationStart trigger that logs to the console.
 */
export default class ApplicationStartTrigger extends Triggers.ApplicationStartTrigger {
    /**
     * Executes the trigger functionality.
     * @param {Triggers.IApplicationStartTriggerOptions} options The options provided to
     */
    public execute(options: Triggers.IApplicationStartTriggerOptions): Promise<void> {
        console.log("Executing ApplicationStartTrigger...");
        return Promise.resolve();
    }
}
```

POS Extensibility Libraries

- POS exposes two module libraries to extensions
- POS API – Contains modules with the components required to create extensions.
  - Ex. Base Classes, Application Infrastructure, Data Model, Helper Classes
  - Will be used by all extensions
- POS UI SDK – Contains modules that expose POS controls to extensions to help custom views maintain a consistent look and feel
  - Recommended when creating extensions that have a user interface

POS API

- Primary module library used by all extension types
  - Type declarations in PosApi.d.ts
- Module structure is organized based on how the module will be used by the extension
  - ***Extend*** – Modules that help extensions ***extend*** POS functionality
    - Subdirectories are organized based on the area of the product to be extended
      - Ex. PosApi/Extend/Views/CustomerDetailsView, PosApi/Extend/Triggers/ProductTriggers
  - ***Create*** – Modules that help extensions ***create*** new functionality
    - Ex. PosApi/Create/Operations provides the types needed to create a new operation
  - ***Consume*** – Modules that allow extensions to ***consume*** POS functionality



Consuming POS Functionality

- Components in the consume section are grouped based on their feature area
  - Ex. Cart, Customer, Device, Employees, OrgUnits, SalesOrders, StoreOperations
- Components that can be used across different feature areas are grouped based on the functionality of the component
  - Ex. Dialogs, DataService, Diagnostics, Peripherals
- These modules are imported from “PosApi/Consume/*
- The “Consume” modules consist primarily of Request and Response types for the POS Runtime

POS Runtime and Extension Context

- The POS Runtime is an implementation of the request-response pattern
  - Similar to the CRT request-response implementation
  - Used throughout POS to execute a wide variety of functionality
- The POS Runtime is exposed to extensions through the extension context
- The extension context is provided to all extensions and contains application infrastructure components

```
/**
 * Represents the type interface for the context object passed to all extensions.
 */
export interface IExtensionContext {
    runtime: IRuntime;
    logger: IExtensionLogger;
    extensibleEnumerationManager: IExtensibleEnumerationManager;
    extensionPackageInfo: IExtensionPackageInfo;
    navigator: INavigator;
    resources: IExtensionResourceManager;
}
```

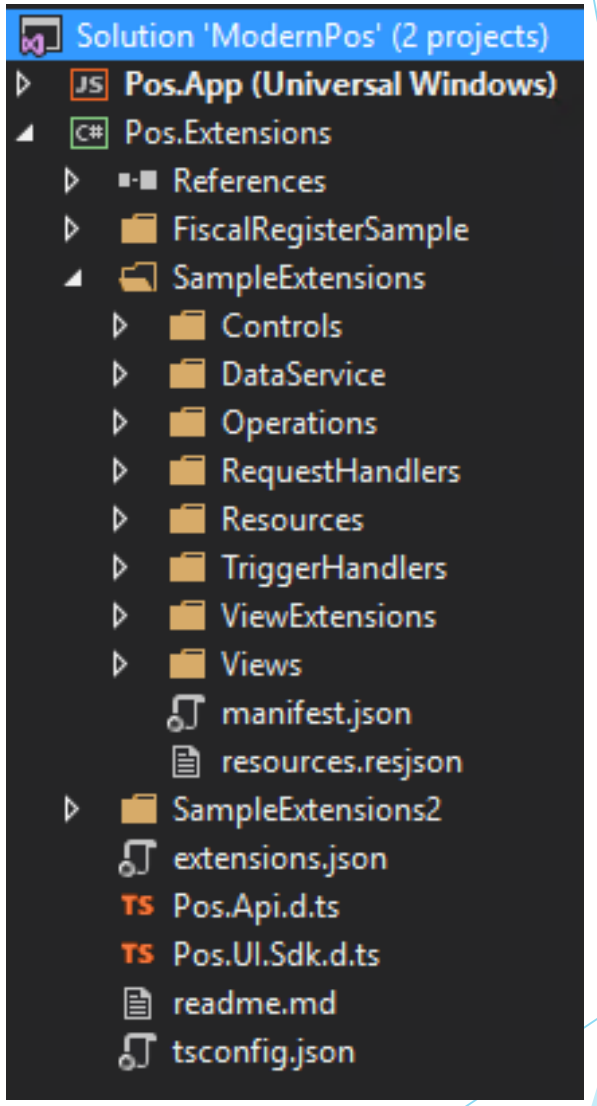
POS UI SDK

- Library containing modules that expose POS controls to extension views
- Each control consists of a class derived from PosControl and a Knockout binding
  - Extension view will import the PosControl class from “PosUISdk/Controls/*”
- The knockout binding uses the PosControl class as the data for the binding
  - Each binding handler is prefixed with “msPos” in order to avoid name collisions
    - Ex. msPosAppBar, msPosAppBarCommand
- Supported Controls:
  - AppBar, AppBarCommand
  - DataList
  - DatePicker
  - HeaderSplitview
  - Loader
  - Menu, MenuCommand, ToggleMenu, ToggleMenuCommand
  - Pivot, PivotItem
  - TimePicker
  - ToggleSwitch

What is included in the Retail SDK?

POS Solution Overview

- Two projects in the POS Solutions:
  - Pos.App/Pos.Web and Pos.Extensions
- Pos.Extensions project contains all the extension packages
  - Each extension package is in a separate directory
- Pos.App/Pos.Web projects only contain JavaScript, HTML and CSS files
  - Removed TypeScript and .scss files
  - These projects are only present to re-package the application with the extensions included and should not be changed



Summary

- Key Takeaways
  - Each extension is a separate module that POS loads based on the information in the manifest file
  - The POS API library is the primary public interface for POS
    - The modules in it will be used when creating all extensions
    - The modules in it are divided based on the functionality they enable for extensions
      - ***Extend*** – Modules that help extensions ***extend*** core POS functionality
      - ***Create*** – Modules that help extensions ***create*** new functionality
      - ***Consume*** – Modules that allow extensions to ***consume*** core POS functionality
    - The extension context provides all extensions with the application infrastructure components
  - The POS UI SDK should be used to help custom views maintain a consistent appearance
  - The POS solutions in the Retail SDK contain two projects: Pos.App/Pos.Web and Pos.Extensions
    - Pos.App/Pos.Web should be considered read-only