Contents

전자책를 사용 하는 엔터프라이즈 응용 프로그램 패턴 Xamarin.Forms

2020-06-05 • 14 minutes to read • Edit Online

조정 가능 하 고, 유지 관리 가능 하며, 테스트 가능한 Xamarin.Forms 엔터프라이즈 응용 프로그램을 개발 하기 위한 아키텍처 지침

이 전자책는 느슨한 결합을 유지 하면서 MVVM (모델 뷰-ViewModel) 패턴, 종속성 주입, 탐색, 유효성 검사 및 구성 관리를 구현 하는 방법에 대 한 지침을 제공 합니다. 또한 IdentityServer를 사용 하 여 인증 및 권한 부여를 수행 하 고, 컨테이너 화 된 마이크로 서비스에서 데이터에 액세스 하 고, 단위 테스트를 수행 하는 방법도 제공 됩니다.

서문

이 장에서는 가이드의 목적과 범위 및 대상 사용자에 대해 설명 합니다.

소개

엔터프라이즈 앱 개발자는 개발 중에 앱의 아키텍처를 변경할 수 있는 몇 가지 문제를 직면 하 고 있습니다. 따라서 시간이 지남에 따라 앱을 수정 하거나 확장할 수 있도록 앱을 빌드하는 것이 중요 합니다. 이러한 적응성 설계는 어려울 수 있지만 일반적으로 앱에 쉽게 통합 될 수 있는 느슨하게 결합 된 개별 구성 요소로 앱을 분할 해야 합니다.

MVVM

MVVM (모델-뷰-ViewModel) 패턴은 응용 프로그램의 비즈니스 및 프레젠테이션 논리를 UI (사용자 인터페이스)와 완전히 분리 하는 데 도움이 됩니다. 응용 프로그램 논리와 UI를 명확 하 게 분리 하면 수많은 개발 문제를 해결 하 고 응용 프로그램을 더 쉽게 테스트 하 고 유지 관리 하 고 진화 시킬 수 있습니다. 또한 코드 다시 사용 기회를 크게 향상 시킬 수 있으며, 개발자와 UI 디자이너는 앱의 각 부분을 개발할 때 더 쉽게 공동 작업을 수행할 수 있습니다.

종속성 주입

종속성 주입을 통해 이러한 형식에 종속 된 코드에서 구체적인 형식을 분리할 수 있습니다. 일반적으로 인터페이스와 추상 형식 간의 등록 및 매핑 목록과 이러한 형식을 구현 하거나 확장 하는 구체적인 형식을 포함 하는 컨테이너를 사용 합니다.

종속성 주입 컨테이너를 사용 하면 클래스 인스턴스를 인스턴스화하고 컨테이너의 구성에 따라 수명을 관리 하는 기능을 제공 하 여 개체 간의 결합을 줄일 수 있습니다. 개체를 만드는 동안 컨테이너는 개체에 필요한 모든 종속성을 삽입 합니다. 이러한 종속성을 아직 만들지 않은 경우 컨테이너는 먼저 종속성을 만들고 확인 합니다.

느슨하게 결합된 구성 요소 간 통신

Xamarin.Forms `MessagingCenter` 클래스는 게시-구독 패턴을 구현 하 여 개체 및 형식 참조로 연결 하기 불편 한 구성 요소 간에 메시지 기반 통신을 허용 합니다. 이 메커니즘을 통해 게시자와 구독자는 서로에 대 한 참조 없이 통신

할 수 있으며, 구성 요소 간의 종속성을 줄이고 구성 요소를 독립적으로 개발 하 고 테스트할 수 있습니다.

탐색

Xamarin.Forms에는 내부 논리 기반 상태 변경으로 인해 일반적으로 사용자가 UI와 상호 작용 하거나 앱 자체에서 발생 하는 페이지 탐색에 대 한 지원이 포함 되어 있습니다. 그러나 MVVM 패턴을 사용 하는 앱에서 탐색을 구현 하는 것은 복잡할 수 있습니다.

이 장에서는 `NavigationService` 모델 보기에서 모델을 처음 탐색할 때 사용 되는 클래스를 제공 합니다. 뷰 모델 클래스에 탐색 논리를 배치 하면 자동화 된 테스트를 통해 논리를 수행할 수 있습니다. 또한 뷰 모델은 특정 비즈니스 규칙이 적용 되도록 탐색을 제어 하는 논리를 구현할 수 있습니다.

유효성 검사

사용자의 입력을 허용 하는 모든 앱은 입력이 올바른지 확인 해야 합니다. 유효성 검사를 수행 하지 않으면 사용자가 응용 프로그램 실패를 유발 하는 데이터를 제공할 수 있습니다. 유효성 검사는 비즈니스 규칙을 적용 하 고 공격자가 악성 데이터를 삽입 하는 것을 방지 합니다.

MVVM (모델-뷰-ViewModel) 패턴의 컨텍스트에서는 사용자가 수정할 수 있도록 데이터 유효성 검사를 수행 하 고 뷰에 유효성 검사 오류를 알리기 위해 뷰 모델 또는 모델이 종종 필요 합니다.

구성 관리

설정을 사용 하면 응용 프로그램의 동작을 구성 하는 데이터를 분리 하 여 앱을 다시 빌드하지 않고도 동작을 변경할 수 있습니다. 앱 설정은 앱에서 만들고 관리 하는 데이터 이며, 사용자 설정은 앱의 동작에 영향을 주는 앱의 사용자 지정 가능한 설정 이며 자주 다시 조정할 필요가 없습니다.

컨테이너화된 마이크로 서비스

마이크로 서비스는 최신 클라우드 응용 프로그램의 민첩성, 확장성 및 안정성 요구 사항에 적합 한 응용 프로그램 개발 및 배포에 대 한 접근 방식을 제공 합니다. 마이크로 서비스의 주요 이점 중 하나는 독립적으로 확장 될 수 있다는 것입니다. 즉, 수요가 증가 하지 않는 응용 프로그램의 영역을 불필요 하 게 확장 하지 않고도 요구를 지원 하기 위해 더 많은 처리 능력 또는 네트워크 대역폭이 요구 되는 특정 기능 영역을 확장할 수 있습니다.

인증 및 권한 부여

Xamarin.FormsASP.NET MVC 웹 응용 프로그램과 통신 하는 앱에 인증 및 권한 부여를 통합 하는 방법에는 여러 가지가 있습니다. 여기서는 IdentityServer 4를 사용 하는 컨테이너 화 된 identity 마이크로 서비스를 사용 하 여 인증 및 권한 부여를 수행 합니다. IdentityServer는 ASP.NET Core Id와 통합 되어 전달자 토큰 인증을 수행 하는 ASP.NET Core에 대 한 오픈 소스 Openid connect Connect 및 OAuth 2.0 프레임 워크입니다.

원격 데이터에 액세스

많은 최신 웹 기반 솔루션은 웹 서버에서 호스트 되는 웹 서비스를 사용 하 여 원격 클라이언트 응용 프로그램에 대 한 기능을 제공 합니다. 웹 서비스에서 노출 하는 작업은 web API를 구성 하 고, 클라이언트 앱은 API에서 노출 하는 데이터 또는 작업이 구현 되는 방식을 몰라도 웹 API를 활용할 수 있어야 합니다.

유닛 테스트

MVVM 응용 프로그램에서 모델을 테스트 하 고 모델을 확인 하는 것은 다른 클래스를 테스트 하는 것과 동일 하며, 동일한 도구와 기법을 사용할 수 있습니다. 그러나 모델 클래스를 모델링 하 고 보는 데 일반적으로 사용할 수 있는 몇 가지 패턴이 있습니다. 이러한 패턴은 특정 유닛 테스트 기법을 활용 합니다.

사용자 의견

이 프로젝트에는 질문을 게시 하 고 피드백을 제공할 수 있는 커뮤니티 사이트가 있습니다. 커뮤니티 사이트는 GitHub에 있습니다. 또는 전자책에 대 한 피드백을에 전자 메일로 보낼 수 있습니다 dotnet-architecture-ebooks-feedback@service.microsoft.com .

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

엔터프라이즈 앱 개발에 대 한 앞으로

2020-06-05 • 8 minutes to read • Edit Online

이 전자책를 사용 하 여 플랫폼 간 엔터프라이즈 앱을 빌드하는 방법에 대 한 지침을 제공 Xamarin.Forms 합니다. Xamarin.Forms는 개발자가 iOS, Android 및 유니버설 Windows 플랫폼 (UWP)를 비롯 한 여러 플랫폼에서 공유할 수 있는 네이티브 사용자 인터페이스 레이아웃을 쉽게 만들 수 있도록 하는 플랫폼 간 UI 도구 키트입니다. 이 솔루션은 B2E (business to Employee), B2B (business to Business) 및 B2C (Business to Consumer) 앱에 대 한 종합적인 솔루션을 제공 하 여 모든 대상 플랫폼에서 코드를 공유 하 고 TCO (총 소유 비용)를 낮추는 기능을 제공 합니다.

이 가이드는 조정 가능 하 고, 유지 관리 가능 하며, 테스트 가능한 엔터프라이즈 앱을 개발 하기 위한 아키텍처 지침을 제공 Xamarin.Forms MVVM, 종속성 주입, 탐색, 유효성 검사 및 구성 관리를 구현 하는 방법에 대 한 지침을 제공 하 고 느슨한 결합을 유지 합니다. 또한 IdentityServer를 사용 하 여 인증 및 권한 부여를 수행 하 고, 컨테이너 화 된 마이크로 서비스에서 데이터에 액세스 하 고, 단위 테스트를 수행 하는 방법도 제공 됩니다.

이 가이드는 eShopOnContainers 모바일 앱에 대 한 소스 코드와 eShopOnContainers reference 앱에 대 한 소스 코드와 함께 제공 됩니다. EShopOnContainers 모바일 앱은 Xamarin.Forms eShopOnContainers reference 앱 이라고 하는 일련의 컨테이너 화 된 마이크로 서비스에 연결 하는를 사용 하 여 개발 된 플랫폼 간 엔터프라이즈 앱입니다. 그러나 컨테이너 화 된 마이크로 서비스 배포를 방지 하려는 사용자를 위해 모의 서비스의 데이터를 사용 하도록 eShopOnContainers 모바일 앱을 구성할 수 있습니다.

이 가이드의 범위에서 벗어난 내용

이 가이드는에 이미 익숙한 독자를 대상으로 Xamarin.Forms 합니다. 에 대 한 자세한 소개는 Xamarin.Forms Xamarin.Forms 설명서를 참조 하 고를 사용 하 Xamarin.Forms 여 Mobile Apps를 만듭니다 .

이 가이드는 컨테이너 화 된 마이크로 서비스 개발 및 배포에 중점을 둔 .Net 마이크로 서비스: 컨테이너 화 된 .Net 응용 프로그램용 아키텍처를 보완 합니다. 다른 가이드에는 ASP.NET Core 및 Microsoft Azure를 사용 하 여 최신 웹 응용 프로그램을 설계 하 고 개발 하 고, Microsoft 플랫폼 및 도구를 사용 하 여 Docker 응용 프로그램 수명 주기를 컨테이너 화 된, 모바일 앱 개발을 위한 microsoft 플랫폼 및 도구

이 가이드를 사용 해야 하는 사람

이 가이드의 대상 그룹은를 사용 하 여 플랫폼 간 엔터프라이즈 앱을 설계 하 고 구현 하는 방법을 배우는 개발자 및 설계자입니다 Xamarin.Forms .

보조 대상은를 사용 하 여 플랫폼 간 엔터프라이즈 앱 개발을 위해 선택할 방법을 결정 하기 전에 아키텍처 및 기술 개요를 받으려는 기술 의사 결정자입니다 Xamarin.Forms .

이 가이드를 사용하는 방법

이 가이드에서는를 사용 하 여 플랫폼 간 엔터프라이즈 앱을 빌드하는 방법을 집중적으로 설명 Xamarin.Forms 합니다. 따라서 이러한 앱 및 기술 고려 사항을 이해 하기 위한 토대를 제공 하기 위해 전체를 읽어야 합니다. 이 가이드는 샘플 앱과 함께 새로운 엔터프라이즈 앱을 만들기 위한 시작 지점 또는 참조 역할을 할 수도 있습니다. 연결 된 샘플 앱을 새 앱의 템플릿으로 사용 하거나 앱 구성 요소 파트를 구성 하는 방법을 확인 합니다. 그런 다음 아키텍처에 대 한 지침을 보려면이 가이드를 다시 참조 하세요.

을 사용 하 여 플랫폼 간 엔터프라이즈 앱 개발에 대 한 일반적인 이해를 돕기 위해이 가이드를 팀 멤버에 게 자유롭게 전달 하세요 Xamarin.Forms . 용어의 공통 집합에서 모든 작업을 수행 하는 경우에는 아키텍처 패턴 및 사례를 일관 되 게 적용 하는 데 도움이 됩니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

엔터프라이즈 앱 개발 소개

2020-06-05 • 26 minutes to read • Edit Online

플랫폼에 관계 없이 엔터프라이즈 앱 개발자는 다음과 같은 몇 가지 문제를 직면 하 고 있습니다.

- 시간이 지남에 따라 변경 될 수 있는 앱 요구 사항입니다.
- 새로운 비즈니스 기회 및 과제
- 앱의 범위와 요구 사항에 상당한 영향을 줄 수 있는 개발 중의 지속적인 피드백

이러한 점을 염두에 두면 시간이 지남에 따라 쉽게 수정 하거나 확장할 수 있는 앱을 빌드하는 것이 중요 합니다. 이러한 적응성을 설계 하는 것은 앱의 나머지 부분에 영향을 주지 않고 독립적으로 앱의 개별 부분을 독립적으로 개발 하 고 테스트할 수 있는 아키텍처가 필요 하기 때문에 어려울 수 있습니다.

많은 엔터프라이즈 앱은 둘 이상의 개발자를 요구 하기에 충분히 복잡 합니다. 앱을 디자인 하는 방법을 결정 하는 것은 여러 개발자가 앱의 여러 부분에서 독립적으로 작업할 수 있도록 하는 동시에 앱에 통합 된 경우에도 원활 하게 연동 되도록 하는 것이 중요 합니다.

앱을 디자인 하 고 빌드하는 일반적인 접근 방식은 *모놀리식* 앱 이라고 하는 것으로, 구성 요소가 긴밀 하 게 분리 되지 않고 긴밀 하 게 결합 됩니다. 일반적으로이 모놀리식 접근 방식은 응용 프로그램의 다른 구성 요소를 중단 하지 않고 버그를 해결 하기 어려울 수 있으며, 새 기능을 추가 하거나 기존 기능을 대체 하기 어려울 수 있기 때문에 유지 관리가 어렵고 비효율적인 앱을 야기 합니다.

이러한 문제를 효과적으로 해결 하려면 앱을 쉽게 앱에 통합할 수 있는 느슨하게 결합 된 불연속 구성 요소로 분할 합니다. 이러한 접근 방식은 다음과 같은 여러 가지 이점을 제공 합니다.

- 개별 기능을 서로 다른 개인 이나 팀에서 개발, 테스트, 확장 및 유지 관리할 수 있습니다.
- 응용 프로그램의 가로 기능 (예: 인증 및 데이터 액세스), 앱 특정 비즈니스 기능 등의 세로 기능 간의 문제를 완전히 분리 하 고 재사용을 촉진 합니다. 이렇게 하면 응용 프로그램 구성 요소 간의 종속성 및 상호 작용을 보다 쉽게 관리할 수 있습니다.
- 다른 개인 이나 팀이 자신의 전문 지식에 따라 특정 작업 또는 기능에 초점을 맞출 수 있도록 하 여 역할의 분리를 유지 하는 데 도움이 됩니다. 특히 사용자 인터페이스와 앱의 비즈니스 논리를 명확 하 게 구분 합니다.

그러나 느슨하게 결합 된 불연속 구성 요소로 앱을 분할할 때 해결 해야 하는 많은 문제가 있습니다. 내용은 다음과 같습니다.

- 사용자 인터페이스 컨트롤과 해당 논리 간의 문제를 명확 하 게 분리 하는 방법을 결정 합니다. 엔터프라이즈 앱을 만들 때 가장 중요 한 결정 사항 중 하나 Xamarin.Forms 는 코드 숨김으로 비즈니스 논리를 배치할지, 사용자 인터페이스 컨트롤과 논리 간의 문제를 명확 하 게 분리 하 여 앱을 유지 관리 하 고 테스트 가능 하 게 만드는 지 여부입니다. 자세한 내용은 모델-뷰-ViewModel을 참조 하세요.
- 종속성 주입 컨테이너를 사용할지 여부를 결정 합니다. 종속성 주입 컨테이너는 종속성이 주입 된 클래스의 인스턴스를 생성 하 고 컨테이너의 구성에 따라 수명 주기를 관리 하는 기능을 제공 하 여 개체 간의 종속성 결합을 줄입니다. 자세한 내용은 종속성 주입을 참조 하세요.
- 플랫폼에서 제공 하는 이벤트와 개체 및 형식 참조로 연결 하기 불편 한 구성 요소 간 느슨하게 결합 된 메시지 기반 통신을 선택 합니다. 자세한 내용은 느슨하게 결합 한 구성 요소 간의 통신소개를 참조 하세요.
- 탐색 호출 방법 및 탐색 논리가 상주해 야 하는 위치를 포함 하 여 페이지 간에 탐색 하는 방법을 결정 합니다. 자세한 내용은 탐색를 참조하세요.
- 사용자 입력의 유효성을 검사 하 여 정확성을 확인 하는 방법을 결정 합니다. 의사 결정은 사용자 입력의 유효성을 검사 하는 방법과 사용자에 게 유효성 검사 오류에 대해 알리는 방법을 포함 해야 합니다. 자세한 내용은 유효성 검사를 참조 하세요.
- 인증을 수행 하는 방법 및 권한 부여를 사용 하 여 리소스를 보호 하는 방법을 결정 합니다. 자세한 내용은 인증 및 권한 부여를 참조 하세요.

- 데이터를 안정적으로 검색 하는 방법 및 데이터를 캐시 하는 방법을 비롯 하 여 웹 서비스에서 원격 데이터에 액세스 하는 방법을 결정 합니다. 자세한 내용은 원격 데이터 액세스를 참조 하세요.
- 앱을 테스트 하는 방법을 결정 합니다. 자세한 내용은 단위 테스트를 참조 하십시오.

이 가이드에서는 이러한 문제에 대 한 지침을 제공 하 고를 사용 하 여 플랫폼 간 엔터프라이즈 앱을 빌드하기 위한 핵심 패턴 및 아키텍처에 초점을 맞춘 것 Xamarin.Forms 입니다. 이 지침에서는 일반적인 Xamarin.Forms 엔터프라이즈 앱 개발 시나리오를 해결 하 고, MVVM (모델-뷰-ViewModel) 패턴 지원을 통해 프레젠테이션, 프레젠테이션 논리 및 엔터티의 문제를 구분 하 여 조정 가능 하 고, 유지 관리 가능 하 고, 테스트 가능한 코드를 생성 하는 데 도움이 됩니다.

예제 응용 프로그램

이 가이드에는 다음 기능을 포함 하는 온라인 스토어 인 eShopOnContainers 샘플 응용 프로그램이 포함 되어 있습니다.

- 백 엔드 서비스에 대 한 인증 및 권한 부여
- Shirts, 커피 머그잔 및 기타 마케팅 항목의 카탈로그를 검색 합니다.
- 카탈로그를 필터링 합니다.
- 카탈로그에서 항목 순서 지정
- 사용자의 주문 기록 보기
- 설정 구성

샘플 응용 프로그램 아키텍처

그림 1-1은 샘플 응용 프로그램의 아키텍처에 대 한 개략적인 개요를 제공 합니다.

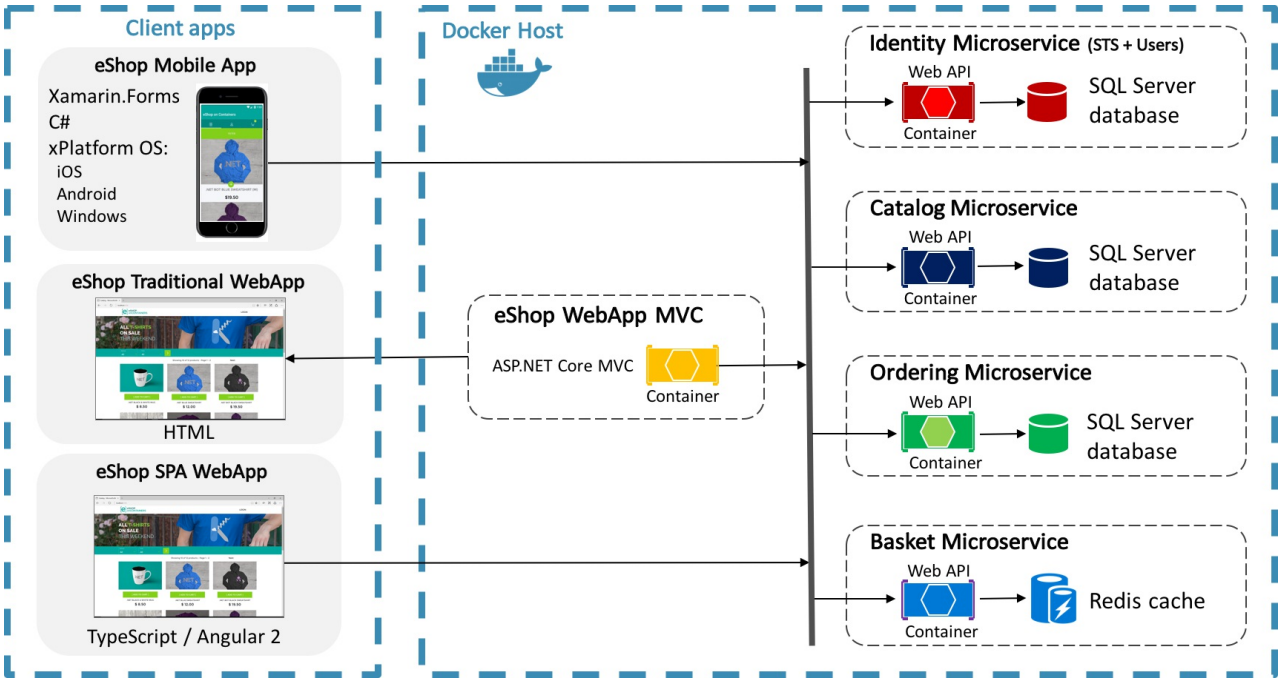

그림 1-1: eShopOnContainers 상위 수준 아키텍처

샘플 응용 프로그램은 다음과 같은 세 개의 클라이언트 앱과 함께 제공 됩니다.

- ASP.NET Core를 사용 하 여 개발 된 MVC 응용 프로그램입니다.
- 2 차원 및 Typescript를 사용 하 여 개발 된 SPA (단일 페이지 응용 프로그램)입니다. 웹 응용 프로그램에 대 한이 방법은 각 작업으로 서버에 대 한 왕복을 수행 하지 않습니다.
- Xamarin.FormsIOS, Android 및 유니버설 Windows 플랫폼 (UWP)를 지 원하는를 사용 하 여 개발 된 모바일 앱입니다.

웹 응용 프로그램에 대 한 자세한 내용은 ASP.NET Core 및 Microsoft Azure를 사용 하 여 최신 웹 응용 프로그램 설

계 및 개발을 참조 하세요.

샘플 응용 프로그램에는 다음과 같은 백 엔드 서비스가 포함 됩니다.

- ASP.NET Core Identity 및 IdentityServer를 사용 하는 id 마이크로 서비스입니다.
- 마이크로 서비스 카탈로그는 EntityFramework Core를 사용 하 여 SQL Server 데이터베이스를 사용 하는 데이터 기반 CRUD (만들기, 읽기, 업데이트, 삭제) 서비스입니다.
- 도메인 기반 디자인 패턴을 사용 하는 도메인 기반 서비스인 주문 마이크로 서비스.
- Redis Cache를 사용 하는 데이터 기반 CRUD 서비스인 바구니 마이크로 서비스.

이러한 백 엔드 서비스는 ASP.NET Core MVC를 사용 하 여 마이크로 서비스로 구현 되 고 단일 Docker 호스트 내에 고유한 컨테이너로 배포 됩니다. 이러한 백 엔드 서비스를 통칭 하 여 eShopOnContainers reference 응용 프로그램이라고 합니다. 클라이언트 앱은 REST (Representational State Transfer) 웹 인터페이스를 통해 백 엔드 서비스와 통신 합니다. 마이크로 서비스 및 Docker에 대 한 자세한 내용은 컨테이너 화 된 마이크로 서비스를 참조 하세요.

백 엔드 서비스의 구현에 대 한 자세한 내용은 .Net 마이크로 서비스: 컨테이너 화 된 .Net 응용 프로그램 아키텍처를 참조 하세요.

모바일 앱

이 가이드에서는를 사용 하 여 플랫폼 간 엔터프라이즈 앱을 빌드하는 데 중점을 Xamarin.Forms eShopOnContainers 모바일 앱을 예로 사용 합니다. 그림 1-2에서는 앞에서 설명한 기능을 제공 하는 eShopOnContainers 모바일 앱의 페이지를 보여 줍니다.

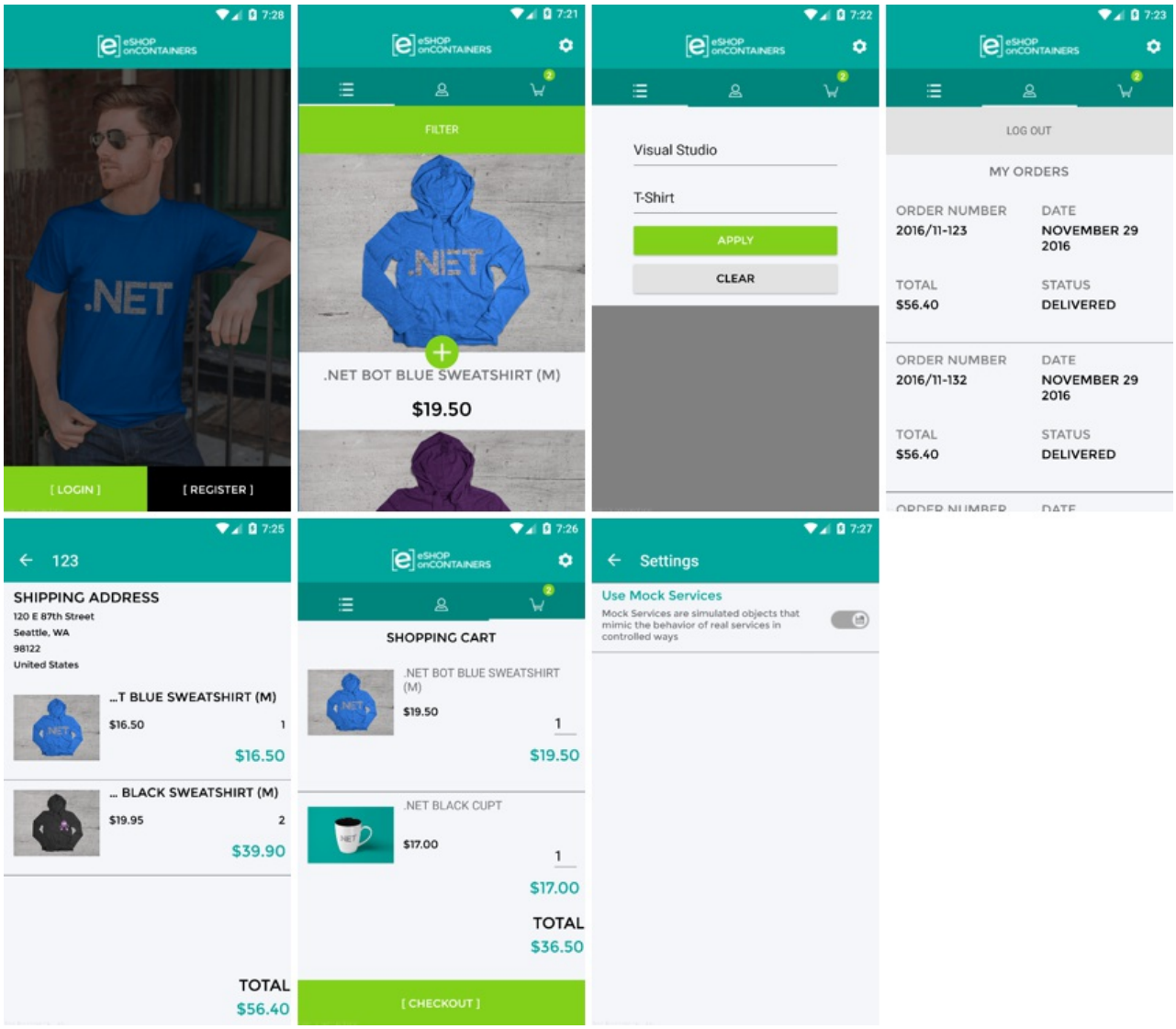

그림 1-2: eShopOnContainers 모바일 앱

모바일 앱은 eShopOnContainers reference 응용 프로그램에서 제공 하는 백 엔드 서비스를 사용 합니다. 그러나 백

엔드 서비스 배포를 방지 하려는 사용자에 대해 모의 서비스의 데이터를 사용 하도록 구성할 수 있습니다.

EShopOnContainers 모바일 앱은 다음 기능을 Xamarin.Forms 수행 합니다.

- XAML
- 컨트롤
- 바인딩
- 컨버터
- 스타일
- 애니메이션
- 명령
- 동작
- 트리거
- 효과
- 사용자 지정 렌더러
- MessagingCenter
- 사용자 지정 컨트롤

이 기능에 대 한 자세한 내용은 Xamarin.Forms 설명서및를 사용 하 Xamarin.Forms 여 Mobile Apps 만들기 를 참조 하세요.

또한 eShopOnContainers 모바일 앱에서 일부 클래스에 대해 단위 테스트가 제공 됩니다.

모바일 앱 솔루션

EShopOnContainers 모바일 앱 솔루션은 소스 코드 및 기타 리소스를 프로젝트로 구성 합니다. 모든 프로젝트는 폴더를 사용 하 여 소스 코드 및 기타 리소스를 범주로 구성 합니다. 다음 표에서는 eShopOnContainers 모바일 앱을 구성 하는 프로젝트를 간략히 설명 합니다.

PROJECT	설명
eShopOnContainers	이 프로젝트는 공유 코드와 공유 UI를 포함 하는 PCL (이식 가능한 클래스 라이브러리) 프로젝트입니다.
eShopOnContainers	이 프로젝트는 android 관련 코드를 저장 하며 Android 앱의 진입점입니다.
eShopOnContainers	이 프로젝트는 ios 관련 코드를 보관 하며 iOS 앱에 대 한 진입점입니다.
eShopOnContainers	이 프로젝트는 UWP (유니버설 Windows 플랫폼) 관련 코드를 포함 하며 Windows 앱의 진입점입니다.
eShopOnContainers. Testrunner.completed	이 프로젝트는 eShopOnContainers 테스트 프로젝트에 대 한 Android test runner입니다.
eShopOnContainers. Testrunner.completed	이 프로젝트는 eShopOnContainers 테스트 프로젝트에 대 한 iOS test runner입니다.
eShopOnContainers. Testrunner.completed	이 프로젝트는 eShopOnContainers 테스트 프로젝트에 대 한 유니버설 Windows 플랫폼 test runner입니다.
eShopOnContainers 테스트	이 프로젝트에는 eShopOnContainers 프로젝트에 대 한 단위 테스트가 포함 되어 있습니다.

EShopOnContainers 모바일 앱의 클래스는 전혀 수정 하지 않고 모든 앱에서 다시 사용할 수 있습니다

Xamarin.Forms .

eShopOnContainers 프로젝트

EShopOnContainers PCL 프로젝트에는 다음 폴더가 포함 되어 있습니다.

폴더	설명
애니메이션	XAML에서 애니메이션을 사용 하도록 설정 하는 클래스를 포함 합니다.
동작	뷰 클래스에 노출 되는 동작을 포함 합니다.
컨트롤	앱에서 사용 하는 사용자 지정 컨트롤을 포함 합니다.
컨버터	바인딩에 사용자 지정 논리를 적용 하는 값 변환기를 포함 합니다.
효과	`EntryLineColorEffect` 특정 컨트롤의 테두리 색을 변경 하는 데 사용 되는 클래스를 포함 `Entry` 합니다.
예외	사용자 지정를 포함 `ServiceAuthenticationException` 합니다.
확장	및 클래스에 대 한 확장 메서드를 포함 `VisualElement` `IEnumerable` 합니다.
도우미	앱에 대 한 도우미 클래스를 포함 합니다.
모델	앱에 대 한 모델 클래스를 포함 합니다.
속성	`AssemblyInfo.cs` .Net 어셈블리 메타 데이터 파일을 포함 합니다.
서비스	응용 프로그램에 제공 되는 서비스를 구현 하는 인터페이스와 클래스를 포함 합니다.
트리거	`BeginAnimation` XAML에서 애니메이션을 호출 하는 데 사용 되는 트리거를 포함 합니다.
유효성 검사	데이터 입력의 유효성 검사와 관련 된 클래스를 포함 합니다.
ViewModels	페이지에 노출 되는 응용 프로그램 논리를 포함 합니다.
보기	앱에 대 한 페이지를 포함 합니다.

플랫폼 프로젝트

플랫폼 프로젝트는 효과 구현, 사용자 지정 렌더러 구현 및 기타 플랫폼별 리소스를 포함 합니다.

요약

Xamarin의 플랫폼 간 모바일 앱 개발 도구 및 플랫폼은 모든 대상 플랫폼 (iOS, Android 및 Windows)에서 코드를 공유 하는 기능을 제공 하 고 총 소유 비용을 낮출 수 있도록 하는 B2E, B2B 및 B2C mobile 클라이언트 앱에 대 한 종합적인 솔루션을 제공 합니다. 앱은 네이티브 플랫폼 모양과 느낌을 유지 하면서 사용자 인터페이스와 앱 논리 코드를 공유할 수 있습니다.

엔터프라이즈 앱 개발자는 개발 중에 앱의 아키텍처를 변경할 수 있는 몇 가지 문제를 직면 하 고 있습니다. 따라서

시간이 지남에 따라 앱을 수정 하거나 확장할 수 있도록 앱을 빌드하는 것이 중요 합니다. 이러한 적응성 설계는 어려울 수 있지만 일반적으로 앱에 쉽게 통합 될 수 있는 느슨하게 결합 된 개별 구성 요소로 앱을 분할 해야 합니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

모델 뷰-ViewModel 패턴

2020-06-05 • 61 minutes to read • Edit Online

Xamarin.Forms일반적으로 개발자 환경에는 XAML에서 사용자 인터페이스를 만든 다음 사용자 인터페이스에서 작동 하는 코드 숨김이 추가 됩니다. 앱이 수정 되 고 크기와 범위가 증가 함에 따라 복잡 한 유지 관리 문제가 발생 할 수 있습니다. 이러한 문제에는 ui 컨트롤과 비즈니스 논리를 긴밀 하 게 결합 하 여 UI를 수정 하는 비용과 이러 한 코드를 단위 테스트 하는 어려움을 포함 합니다.

MVVM (모델-뷰-ViewModel) 패턴은 응용 프로그램의 비즈니스 및 프레젠테이션 논리를 UI (사용자 인터페이스)와 완전히 분리 하는 데 도움이 됩니다. 응용 프로그램 논리와 UI를 명확 하 게 분리 하면 수많은 개발 문제를 해결 하 고 응용 프로그램을 더 쉽게 테스트 하 고 유지 관리 하 고 진화 시킬 수 있습니다. 또한 코드 다시 사용 기회를 크게 향상 시킬 수 있으며, 개발자와 UI 디자이너는 앱의 각 부분을 개발할 때 더 쉽게 공동 작업을 수행할 수 있습니다.

MVVM 패턴

MVVM 패턴에는 모델, 뷰 및 뷰 모델의 세 가지 핵심 구성 요소가 있습니다. 각각은 고유한 용도로 사용 됩니다. 그림 2-1에서는 세 가지 구성 요소 간의 관계를 보여 줍니다.

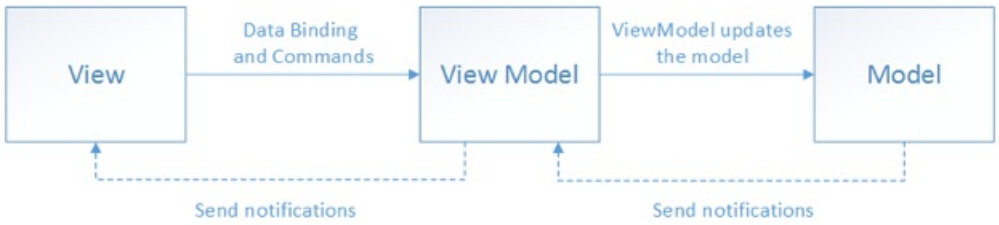

그림 2-1: MVVM 패턴

각 구성 요소의 책임을 이해 하는 것 외에도 서로 상호 작용 하는 방식을 이해 하는 것도 중요 합니다. 높은 수준에서 뷰 모델은 뷰 모델을 "인식" 하 고 뷰 모델은 모델을 "인식" 하지만 모델은 뷰 모델을 인식 하지 않으며 뷰 모델은 뷰를 인식 하지 못합니다. 따라서 뷰 모델은 모델에서 뷰를 격리 하며 모델을 뷰와 독립적으로 발전 시킬 수 있습니다.

MVVM 패턴을 사용 하는 이점은 다음과 같습니다.

- 기존 비즈니스 논리를 캡슐화 하는 기존 모델 구현이 있는 경우 변경 하는 것이 어렵거나 위험할 수 있습니다. 이 시나리오에서 뷰 모델은 모델 클래스의 어댑터 역할을 하므로 모델 코드를 크게 변경 하지 않아도 됩니다.
- 개발자는 뷰를 사용 하지 않고 뷰 모델 및 모델에 대 한 단위 테스트를 만들 수 있습니다. 뷰 모델에 대 한 단위 테스트는 뷰에서 사용 하는 것과 동일한 기능을 실행할 수 있습니다.
- 뷰가 완전히 XAML로 구현 되는 경우 코드를 건드리지 않고도 앱 UI를 다시 만들 수 있습니다. 따라서 새 버전의 뷰가 기존 뷰 모델에서 작동 해야 합니다.
- 디자이너와 개발자는 개발 프로세스 중에 해당 구성 요소에 대해 독립적으로 또는 동시에 작업할 수 있습니다. 개발자는 뷰 모델 및 모델 구성 요소에서 작업할 수 있는 반면 디자이너는 뷰에 집중할 수 있습니다.

MVVM를 효과적으로 사용 하는 핵심은 응용 프로그램 코드를 올바른 클래스로 요소를 구분 하는 방법과 클래스가 상호 작용 하는 방식을 이해 하는 것입니다. 다음 섹션에서는 MVVM 패턴의 각 클래스에 대 한 책임을 설명 합니다.

보기

보기는 사용자가 화면에서 볼 수 있는 구조, 레이아웃 및 모양을 정의 하는 작업을 담당 합니다. 각 보기는 비즈니스 논리를 포함 하지 않는 제한 된 코드 숨김으로 XAML로 정의 하는 것이 가장 좋습니다. 그러나 경우에 따라 코드 숨김이 XAML에서 표현 하기 어려운 시각적 동작을 구현 하는 UI 논리 (예: 애니메이션)를 포함할 수 있습니다.

Xamarin.Forms응용 프로그램에서 뷰는 일반적으로 파생 된 `Page` `ContentView` 클래스 또는 파생 클래스입니다. 그러나 뷰는 표시 될 때 개체를 시각적으로 나타내는 데 사용 되는 UI 요소를 지정 하는 데이터 템플릿으로 나타낼 수도 있습니다. 뷰로 표시 되는 데이터 템플릿에는 코드 숨김이 없으며 특정 뷰 모델 유형에 바인딩되도록 설계 되었습니다.

> **TIP**
> 코드 숨김으로 UI 요소를 사용 하거나 사용 하지 않도록 설정 합니다. 뷰 모델에서 명령의 사용 가능 여부 또는 작업이 보류 중임 표시와 같은 보기 표시의 일부 측면에 영향을 주는 논리적 상태 변경을 정의 해야 합니다. 따라서 코드 숨김으로 활성화 하거나 비활성화 하는 것이 아니라 모델 속성 보기에 바인딩하여 UI 요소를 사용 하거나 사용 하지 않도록 설정 합니다.

보기의 상호 작용에 대 한 응답으로 보기 모델에서 코드를 실행 하기 위한 몇 가지 옵션이 있습니다 (예: 단추 클릭 또는 항목 선택). 컨트롤이 명령을 지 원하는 경우 컨트롤의 `Command` 속성을 `ICommand` 뷰 모델의 속성에 데이터 바인딩할 수 있습니다. 컨트롤의 명령이 호출 되 면 뷰 모델의 코드가 실행 됩니다. 명령 외에도 동작은 뷰의 개체에 연결 될 수 있으며 호출 되거나 발생 될 이벤트를 수신 대기할 수 있습니다. 이에 대 한 응답으로 동작은 뷰 모델 `ICommand` 또는 뷰 모델의 메서드에 대해를 호출할 수 있습니다.

ViewModel

뷰 모델은 뷰가 데이터 바인딩할 수 있는 속성 및 명령을 구현 하 고 변경 알림 이벤트를 통해 상태 변경을 뷰에 알립니다. 뷰 모델에서 제공 하는 속성 및 명령은 UI에서 제공 되는 기능을 정의 하지만 뷰는 해당 기능을 표시 하는 방법을 결정 합니다.

> **TIP**
> 비동기 작업을 통해 UI의 응답성을 유지 합니다. 모바일 앱은 사용자의 성능 인식 기능을 개선 하기 위해 UI 스레드를 차단 해제 된 상태로 유지 해야 합니다. 따라서 뷰 모델에서 i/o 작업에 비동기 메서드를 사용 하 고 이벤트를 발생 시켜 속성 변경 내용에 대 한 뷰를 비동기적으로 알립니다.

뷰 모델은 필요한 모델 클래스와 뷰의 상호 작용을 조정 해야 합니다. 일반적으로 뷰 모델과 모델 클래스 사이에는 일 대 다 관계가 있습니다. 뷰 모델은 뷰의 컨트롤이 직접 데이터 바인딩할 수 있도록 뷰에 모델 클래스를 직접 노출 하도록 선택할 수 있습니다. 이 경우 데이터 바인딩을 지원 하 고 알림 이벤트를 변경 하려면 모델 클래스를 디자인 해야 합니다.

각 뷰 모델은 뷰에서 쉽게 사용할 수 있는 형태로 모델의 데이터를 제공 합니다. 이를 위해 뷰 모델에서 데이터 변환을 수행 하는 경우가 있습니다. 뷰가 바인딩될 수 있는 속성을 제공 하기 때문에이 데이터 변환을 뷰 모델에 배치 하는 것이 좋습니다. 예를 들어 뷰 모델은 두 속성의 값을 결합 하 여 뷰가 쉽게 표시 되도록 할 수 있습니다.

> **TIP**
> 변환 계층에서 데이터 변환을 중앙 집중화 합니다. 또한 변환기를 뷰 모델과 뷰 사이에 있는 별도의 데이터 변환 계층으로 사용할 수 있습니다. 예를 들어 뷰 모델에서 제공 하지 않는 특수 형식이 데이터에 필요한 경우이 작업이 필요할 수 있습니다.

뷰 모델에서 뷰를 사용 하 여 양방향 데이터 바인딩에 참여 시키려면 해당 속성에서 이벤트를 발생 시켜야 합니다 `PropertyChanged` . 뷰 모델은 인터페이스를 구현 하 `INotifyPropertyChanged` 고 `PropertyChanged` 속성이 변경 될 때 이벤트를 발생 시켜이 요구 사항을 충족 합니다.

컬렉션의 경우 보기 친화적인 `ObservableCollection<T>` 가 제공 됩니다. 이 컬렉션은 개발자가 컬렉션에서 인터페이스를 구현할 필요가 하므로 컬렉션 변경 알림을 구현 합니다 `INotifyCollectionChanged` .

모델

모델 클래스는 응용 프로그램의 데이터를 캡슐화 하는 비시각적 클래스입니다. 따라서 모델은 일반적으로 비즈니

스 및 유효성 검사 논리와 함께 데이터 모델을 포함 하는 앱의 도메인 모델을 나타내는 것으로 간주할 수 있습니다. 모델 개체의 예로는 Dto (데이터 전송 개체), POCOs (일반 이전 CLR 개체) 및 생성 된 엔터티 및 프록시 개체가 있습니다.

일반적으로 모델 클래스는 데이터 액세스 및 캐싱을 캡슐화 하는 서비스 또는 리포지토리와 함께 사용 됩니다.

뷰에 뷰 모델 연결

뷰 모델은의 데이터 바인딩 기능을 사용 하 여 보기에 연결할 수 있습니다 Xamarin.Forms . 뷰를 생성 하 고 모델을 보고 런타임에 연결 하는 데 사용할 수 있는 여러 가지 방법이 있습니다. 이러한 방법은 첫 번째 컴퍼지션 보기 및 모델 첫 번째 컴퍼지션 이라는 두 가지 범주로 나뉩니다. 보기의 첫 번째 컴퍼지션과 뷰 모델 첫 번째 구성 중에서 선택 하는 것은 기본 설정 및 복잡성의 문제입니다. 그러나 모든 접근 방식은 동일한 목표를 공유 합니다. 즉, 뷰의 BindingContext 속성에 뷰 모델이 할당 됩니다.

보기의 첫 번째 구성을 사용 하 여 앱은 개념적으로 종속 된 뷰 모델에 연결 하는 뷰로 구성 됩니다. 이 방법의 주요 장점은 뷰 모델은 뷰 자체에 종속 되지 않기 때문에 느슨하게 연결 된 유닛 테스트 가능 앱을 쉽게 만들 수 있다는 것입니다. 또한 클래스를 만들고 연결 하는 방법을 이해 하기 위해 코드 실행을 추적 하는 대신 시각적 구조를 따라 응용 프로그램 구조를 이해 하는 것이 쉽습니다. 또한 탐색이 발생할 때 페이지를 구성 하는 탐색 시스템을 사용 하 여 뷰를 처음 생성 하는 것이 좋습니다 Xamarin.Forms . 그러면 뷰 모델의 첫 번째 구성이 복잡 하 고 플랫폼과 맞지 않습니다.

모델을 처음 구성 하는 경우 응용 프로그램은 개념적으로 뷰 모델로 구성 되며, 서비스는 뷰 모델에 대 한 뷰를 찾을 책임이 있습니다. 뷰 만들기를 추상화 하 여 앱의 논리적 비 UI 구조에 집중할 수 있으므로 뷰 모델 첫 번째 컴퍼지션은 일부 개발자에 게 더 자연스럽 게 표시 됩니다. 또한 다른 뷰 모델에서 뷰 모델을 만들 수 있습니다. 그러나 이 방법은 복잡 하며 앱의 다양 한 부분을 만들고 연결 하는 방법을 이해 하기 어려울 수 있습니다.

> **TIP**
> 보기 모델과 뷰를 독립적으로 유지 합니다. 뷰를 데이터 원본의 속성에 바인딩하는 것은 해당 뷰 모델에 대 한 뷰의 주 종속성 이어야 합니다. 특히 및와 같은 뷰 모델에서 뷰 유형을 참조 하지 마세요 `Button` `ListView` . 여기에 설명 된 원칙에 따라 뷰 모델을 격리 하 여 테스트할 수 있으므로 범위를 제한 하 여 소프트웨어 결함의 가능성을 줄일 수 있습니다.

다음 섹션에서는 뷰 모델을 뷰에 연결 하는 주요 방법에 대해 설명 합니다.

선언적으로 뷰 모델 만들기

가장 간단한 방법은 뷰가 XAML에서 해당 뷰 모델을 선언적으로 인스턴스화하는 것입니다. 뷰가 생성 될 때 해당 뷰 모델 개체도 생성 됩니다. 이 방법은 다음 코드 예제에서 보여 줍니다.

```
<ContentPage ... xmlns:local="clr-namespace:eShop">
    <ContentPage.BindingContext>
        <local:LoginViewModel />
    </ContentPage.BindingContext>
    ...
</ContentPage>
```

`ContentPage`가 만들어지면의 인스턴스가 `LoginViewModel` 자동으로 생성 되 고 뷰의로 설정 됩니다 `BindingContext` .

뷰에서 뷰 모델을 선언적으로 생성 하 고 할당 하는 것은 간단 하지만 뷰 모델에서 기본 (매개 변수 없음) 생성자가 필요 하다는 단점이 있습니다.

프로그래밍 방식으로 뷰 모델 만들기

뷰 모델은 해당 속성에 할당 되는 코드를 코드 파일에 포함할 수 있습니다 `BindingContext` . 이는 다음 코드 예제와 같이 뷰의 생성자에서 수행 되는 경우가 많습니다.

```
public LoginView()
{
    InitializeComponent();
    BindingContext = new LoginViewModel(navigationService);
}
```

뷰의 코드 숨김으로 보기 모델을 프로그래밍 방식으로 생성 하 고 할당 하는 것은 간단 합니다. 그러나이 방법의 주요 단점은 뷰가 뷰 모델에 필요한 종속성을 제공 해야 한다는 것입니다. 종속성 주입 컨테이너를 사용 하면 뷰와 뷰 모델 간의 느슨한 결합을 유지 하는 데 도움이 됩니다. 자세한 내용은 종속성 주입을 참조 하세요.

데이터 템플릿으로 정의 된 뷰 만들기

뷰를 데이터 템플릿으로 정의 하 고 뷰 모델 유형과 연결할 수 있습니다. 데이터 템플릿을 리소스로 정의 하거나 뷰 모델을 표시 하는 컨트롤 내에서 인라인으로 정의할 수 있습니다. 컨트롤의 내용은 뷰 모델 인스턴스이고 데이터 템플릿이 시각적으로 표시 하는 데 사용 됩니다. 이 기법은 뷰 모델을 먼저 인스턴스화한 후 뷰를 만든 상황의 예입니다.

뷰 모델 로케이터를 사용 하 여 자동으로 뷰 모델 만들기

뷰 모델 로케이터는 뷰 모델의 인스턴스화 및 뷰에 대 한 연결을 관리 하는 사용자 지정 클래스입니다. EShopOnContainers 모바일 앱에서 클래스에는 `ViewModelLocator` `AutoWireViewModel` 뷰 모델을 뷰와 연결 하는 데 사용 되는 연결 된 속성인가 있습니다. 뷰의 XAML에서이 연결 된 속성은 다음 코드 예제와 같이 뷰 모델이 뷰에 자동으로 연결 되어야 함을 나타내기 위해 true로 설정 됩니다.

```
viewModelBase:ViewModelLocator.AutoWireViewModel="true"
```

`AutoWireViewModel` 속성은 false로 초기화 되는 바인딩 가능한 속성 이며, 해당 값이 변경 되 면 `OnAutoWireViewModelChanged` 이벤트 처리기가 호출 됩니다. 이 메서드는 뷰에 대 한 뷰 모델을 확인 합니다. 다음 코드 예제에서는이 작업을 수행 하는 방법을 보여 줍니다.

```
private static void OnAutoWireViewModelChanged(BindableObject bindable, object oldValue, object newValue)
{
    var view = bindable as Element;
    if (view == null)
    {
        return;
    }

    var viewType = view.GetType();
    var viewName = viewType.FullName.Replace(".Views.", ".ViewModels.");
    var viewAssemblyName = viewType.GetTypeInfo().Assembly.FullName;
    var viewModelName = string.Format(
        CultureInfo.InvariantCulture, "{0}Model, {1}", viewName, viewAssemblyName);

    var viewModelType = Type.GetType(viewModelName);
    if (viewModelType == null)
    {
        return;
    }
    var viewModel = _container.Resolve(viewModelType);
    view.BindingContext = viewModel;
}
```

`OnAutoWireViewModelChanged` 메서드는 규칙 기반 접근 방법을 사용 하 여 뷰 모델을 확인 하려고 합니다. 이 규칙은 다음을 가정 합니다.

- 뷰 모델은 뷰 유형과 동일한 어셈블리에 있습니다.
- 보기는에 있습니다. 뷰 하위 네임 스페이스입니다.
- 뷰 모델은에 있습니다. ViewModels 자식 네임 스페이스입니다.

- 뷰 모델 이름은 뷰 이름에 해당 하며 "ViewModel"으로 끝납니다.

마지막으로 `OnAutoWireViewModelChanged` 메서드는 `BindingContext` 뷰 형식의를 확인 된 뷰 모델 형식으로 설정 합니다. 뷰 모델 유형을 확인 하는 방법에 대 한 자세한 내용은 해결을 참조 하세요.

이 방법은 응용 프로그램에 뷰 모델 인스턴스화 및 뷰에 대 한 연결을 담당 하는 단일 클래스가 있다는 장점이 있습니다.

> **TIP**
> 대체 하기 쉽도록 보기 모델 로케이터를 사용 합니다. 모델 보기 로케이터는 단위 테스트 또는 디자인 타임 데이터와 같은 종속성의 대체 구현에 대 한 대체 지점으로도 사용할 수 있습니다.

기본 뷰 모델 또는 모델의 변경 내용에 대 한 응답으로 뷰를 업데이트 합니다.

뷰에 액세스할 수 있는 모든 뷰 모델 및 모델 클래스는 인터페이스를 구현 해야 합니다 `INotifyPropertyChanged` . 뷰 모델 또는 모델 클래스에서이 인터페이스를 구현 하면 내부 속성 값이 변경 될 때 클래스에서 뷰의 데이터 바인딩된 컨트롤에 변경 알림을 제공할 수 있습니다.

다음 요구 사항을 충족 하 여 적절 한 속성 변경 알림을 사용 하도록 앱을 설계 해야 합니다.

- `PropertyChanged`공용 속성의 값이 변경 되는 경우 항상 이벤트를 발생 시킵니다. `PropertyChanged` XAML 바인딩이 발생 하는 방법에 대 한 정보로 인해 이벤트 발생을 무시할 수 있다고 가정 하지 마세요.
- `PropertyChanged`모델 또는 모델의 다른 속성에서 값을 사용 하는 계산 된 속성에 대해서는 항상 이벤트를 발생 시킵니다.
- `PropertyChanged`속성을 변경 하는 메서드의 끝에서 항상 이벤트를 발생 시킵니다. 또는 개체가 안전한 상태에 있는 것으로 알려져 있습니다. 이벤트를 발생 시키면 이벤트의 처리기를 동기적으로 호출 하 여 작업이 중단 됩니다. 작업 중간에 발생 하는 경우 개체를 노출 하 여 안전 하지 않은 부분적으로 업데이트 된 상태에 있을 때 함수를 콜백 할 수 있습니다. 또한 이벤트에 의해 연계 변경을 트리거할 수도 `PropertyChanged` 있습니다. 일반적으로 연계 변경을 수행 하려면 먼저 업데이트를 완료 해야 합니다.
- `PropertyChanged`속성이 변경 되지 않으면 이벤트를 발생 시 키 지 않습니다. 즉, 이벤트를 발생 시키기 전에 이전 값과 새 값을 비교 해야 합니다 `PropertyChanged` .
- `PropertyChanged`속성을 초기화 하는 경우 뷰 모델의 생성자 중에 이벤트를 발생 시 키 지 않습니다. 뷰의 데이터 바인딩된 컨트롤은이 시점에서 변경 알림을 받도록 구독 하지 않습니다.
- `PropertyChanged`클래스의 공용 메서드에 대 한 단일 동기 호출 내에서 동일한 속성 이름 인수를 사용 하 여 이벤트를 두 개 이상 발생 시 키 지 않습니다. 예를 들어 `NumberOfItems` 백업 저장소가 필드인 속성이 지정 된 `_numberOfItems` 경우 루프를 실행 하는 동안 메서드가 50 번 증가 하는 경우 `_numberOfItems` `NumberOfItems` 모든 작업이 완료 된 후 속성에 대해 속성 변경 알림만 발생 해야 합니다. 비동기 메서드의 경우 `PropertyChanged` 비동기 연속 체인의 각 동기 세그먼트에서 지정 된 속성 이름에 대 한 이벤트를 발생 시킵니다.

EShopOnContainers 모바일 앱은 `ExtendedBindableObject` 다음 코드 예제와 같이 클래스를 사용 하 여 변경 알림을 제공 합니다.

```
public abstract class ExtendedBindableObject : BindableObject
{
    public void RaisePropertyChanged<T>(Expression<Func<T>> property)
    {
        var name = GetMemberInfo(property).Name;
        OnPropertyChanged(name);
    }

    private MemberInfo GetMemberInfo(Expression expression)
    {
        ...
    }
}
```

Xamarin.ios의 클래스는 `BindableObject` `INotifyPropertyChanged` 인터페이스를 구현 하 고 메서드를 제공 `OnPropertyChanged` 합니다. `ExtendedBindableObject`클래스는 `RaisePropertyChanged` 속성 변경 알림을 호출 하는 메서드를 제공 하며, 이렇게 하면 클래스에서 제공 하는 기능을 사용 합니다 `BindableObject` .

EShopOnContainers 모바일 앱의 각 뷰 모델 클래스는 클래스에서 파생 되며이 클래스는 `ViewModelBase` 클래스에서 파생 `ExtendedBindableObject` 됩니다. 따라서 각 뷰 모델 클래스는 클래스의 메서드를 사용 하 여 `RaisePropertyChanged` `ExtendedBindableObject` 속성 변경 알림을 제공 합니다. 다음 코드 예제에서는 eShopOnContainers mobile 앱에서 람다 식을 사용 하 여 속성 변경 알림을 호출 하는 방법을 보여 줍니다.

```
public bool IsLogin
{
    get
    {
        return _isLogin;
    }
    set
    {
        _isLogin = value;
        RaisePropertyChanged(() => IsLogin);
    }
}
```

이러한 방식으로 람다 식을 사용 하면 각 호출에 대해 람다 식을 평가 해야 하기 때문에 성능이 약간 저하 됩니다. 성능 비용이 작으며 일반적으로 앱에 영향을 주지 않지만 많은 변경 알림이 있는 경우 비용이 발생할 수 있습니다. 그러나이 방법의 혜택은 속성 이름을 바꿀 때 컴파일 시간 형식 안전성 및 리팩터링 지원을 제공 한다는 것입니다.

명령 및 동작을 사용 하 여 UI 상호 작용

모바일 앱에서 작업은 일반적으로 단추 클릭과 같은 사용자 작업에 대 한 응답으로 호출 됩니다 .이 작업은 코드 숨김으로 이벤트 처리기를 만들어 구현할 수 있습니다. 그러나 MVVM 패턴에서 작업을 구현 하는 책임은 뷰 모델을 사용 하는 것 이며 코드 숨김으로 코드를 배치 하는 것은 피해 야 합니다.

명령은 UI의 컨트롤에 바인딩할 수 있는 작업을 표시 하는 편리한 방법을 제공 합니다. 작업을 구현 하는 코드를 캡슐화 하 고 뷰의 시각적 표현에서 분리 된 상태로 유지 하는 데 도움을 줍니다. Xamarin.Forms에는 명령에 선언적으로 연결 될 수 있는 컨트롤이 포함 되어 있으며, 이러한 컨트롤은 사용자가 컨트롤과 상호 작용할 때 명령을 호출 합니다.

동작을 통해 컨트롤을 명령에 선언적으로 연결할 수도 있습니다. 그러나 동작은 컨트롤에서 발생 한 이벤트 범위와 연결 된 작업을 호출 하는 데 사용할 수 있습니다. 따라서 동작은 더 많은 유연성과 제어를 제공 하는 동시에 명령 사용 컨트롤과 동일한 많은 시나리오를 처리 합니다. 또한 동작을 사용 하 여 명령 개체 또는 메서드를 명령과 상호 작용 하도록 특별히 설계 되지 않은 컨트롤과 연결할 수도 있습니다.

명령 구현

뷰 모델은 일반적으로 뷰의 바인딩에 대해 인터페이스를 구현 하는 개체 인스턴스인 명령 속성을 노출 `ICommand`

합니다. 많은 컨트롤은 Xamarin.Forms `Command` 뷰 모델에서 제공 하는 개체에 데이터 바인딩될 수 있는 속성을 제공 `ICommand` 합니다. `ICommand` 인터페이스는 `Execute` 작업 자체를 캡슐화 하는 메서드를 정의 하 `CanExecute` 고, 메서드를 호출할 수 있는지 여부를 나타내는 메서드를 정의 하 고, `CanExecuteChanged` 명령이 실행 되어야 하는지 여부에 영향을 주는 변경이 발생할 때 발생 하는 이벤트를 정의 합니다. `Command` 에서 제공 하는 및 `Command<T>` 클래스는 인터페이스를 Xamarin.Forms 구현 합니다 `ICommand` `T` . 여기서은 및에 대 한 인수의 형식입니다 `Execute` `CanExecute` .

뷰 모델 내에는 유형의 `Command` `Command<T>` 뷰 모델에 있는 각 공용 속성에 대 한 또는 유형의 개체가 있어야 합니다 `ICommand` . `Command` 또는 생성자에는 `Command<T>` `Action` 메서드가 호출 될 때 호출 되는 콜백 개체가 필요 `ICommand.Execute` 합니다. `CanExecute` 메서드는 선택적 생성자 매개 변수 이며,는을 `Func` 반환 하는입니다 `bool` .

다음 코드에서는 `Command` 레지스터 명령을 나타내는 인스턴스가 `Register` 뷰 모델 메서드에 대리자를 지정 하여 생성 되는 방법을 보여 줍니다.

```
public ICommand RegisterCommand => new Command(Register);
```

명령은에 대 한 참조를 반환 하는 속성을 통해 뷰에 노출 됩니다 `ICommand` . 메서드는 `Execute` 개체에 대해 호출 될 때 `Command` 생성자에 지정 된 대리자를 통해 뷰 모델의 메서드에 대 한 호출을 전달 하기만 `Command` 합니다.

명령의 대리자를 지정 하는 `async` 경우 및 키워드를 사용 하 여 명령에 의해 비동기 메서드를 호출할 수 있습니다 `await` `Execute` . 이는 콜백이이 `Task` 고 대기 해야 함을 나타냅니다. 예를 들어 다음 코드는 `Command` 뷰 모델 메서드에 대리자를 지정 하 여 로그인 명령을 나타내는 인스턴스가 생성 되는 방법을 보여 줍니다 `SignInAsync` .

```
public ICommand SignInCommand => new Command(async () => await SignInAsync());
```

`Execute` `CanExecute` 클래스를 사용 하 여 명령을 인스턴스화하여 매개 변수를 및 동작에 전달할 수 있습니다 `Command<T>` . 예를 들어 다음 코드에서는 `Command<T>` 메서드가 형식의 인수를 요구 함을 나타내는 데 사용 되는 방법을 보여 줍니다 `NavigateAsync` `string` .

```
public ICommand NavigateCommand => new Command<string>(NavigateAsync);
```

`Command` 및 `Command<T>` 클래스에서 `CanExecute` 각 생성자의 메서드에 대 한 대리자는 선택 사항입니다. 대리자가 지정 되지 않은 경우 `Command` 에 대해이 반환 됩니다 `true` `CanExecute` . 그러나 뷰 모델은 `CanExecute` `ChangeCanExecute` 개체에 대해 메서드를 호출 하 여 명령의 상태 변경을 나타낼 수 있습니다 `Command` . 이로 인해 `CanExecuteChanged` 이벤트가 발생 합니다. 명령에 바인딩된 UI의 모든 컨트롤은 해당 사용 상태를 업데이트 하 여 데이터 바인딩된 명령의 가용성을 반영 합니다.

뷰에서 명령 호출

다음 코드 예제에서는 `Grid` `LoginView` `RegisterCommand` `LoginViewModel` 인스턴스를 사용 하 여의가 클래스의에 바인딩하는 방법을 보여 줍니다 `TapGestureRecognizer` .

```
<Grid Grid.Column="1" HorizontalOptions="Center">
    <Label Text="REGISTER" TextColor="Gray"/>
    <Grid.GestureRecognizers>
        <TapGestureRecognizer Command="{Binding RegisterCommand}" NumberOfTapsRequired="1" />
    </Grid.GestureRecognizers>
</Grid>
```

명령 매개 변수는 속성을 사용 하 여 선택적으로 정의할 수도 있습니다 `CommandParameter` . 필요한 인수의 형식은 `Execute` 및 `CanExecute` 대상 메서드에 지정 되어 있습니다. `TapGestureRecognizer` 사용자가 연결 된 컨트롤과 상호 작용할 때가 대상 명령을 자동으로 호출 합니다. 명령 매개 변수 (제공 된 경우)는 명령 대리자에 인수로 전달 됩

니다 `Execute` .

동작 구현

동작을 사용 하면 기능을 서브 클래스 하지 않고도 UI 컨트롤에 추가할 수 있습니다. 대신 기능은 동작 클래스에서 구현되고 컨트롤 자체의 일부였던 것처럼 컨트롤에 연결됩니다. 동작을 사용 하면 컨트롤의 API와 직접 상호 작용 하 고, 컨트롤에 간단 하 게 연결 하 고, 둘 이상의 뷰나 앱에서 다시 사용 하기 위해 패키지할 수 있으므로 일반적으로 코드 숨김으로 작성 해야 하는 코드를 구현할 수 있습니다. MVVM 컨텍스트에서 동작은 컨트롤을 명령에 연결 하는 데 유용한 방법입니다.

연결 된 속성을 통해 컨트롤에 연결 된 동작을 *연결 된 동작*이라고 합니다. 그러면 동작에서 해당 컨트롤이 연결 된 요소의 노출 된 API를 사용 하 여 뷰의 시각적 트리에서 해당 컨트롤이 나 기타 컨트롤에 기능을 추가할 수 있습니다. EShopOnContainers 모바일 앱은 연결 된 `LineColorBehavior` 동작인 클래스를 포함 합니다. 이 동작에 대 한 자세한 내용은 유효성 검사 오류 표시를 참조 하세요.

Xamarin.Forms동작은 또는 클래스에서 파생 되는 클래스입니다 `Behavior` `Behavior<T>` `T` . 여기서은 동작을 적용할 컨트롤의 형식입니다. 이러한 클래스는 `OnAttachedTo` 및 `OnDetachingFrom` 메서드를 제공 합니다 .이 메서드는 동작을 컨트롤에 연결 하 고 분리할 때 실행 되는 논리를 제공 하도록 재정의 해야 합니다.

EShopOnContainers 모바일 앱에서 `BindableBehavior<T>` 클래스는 클래스에서 파생 `Behavior<T>` 됩니다. 클래스의 목적은 `BindableBehavior<T>` Xamarin.Forms `BindingContext` 동작의를 연결 된 컨트롤에 설정 해야 하는 동작에 대 한 기본 클래스를 제공 하는 것입니다.

`BindableBehavior<T>`클래스는 `OnAttachedTo` 동작의를 설정 하는 재정의 가능한 메서드와 `BindingContext` `OnDetachingFrom` 를 정리 하는 재정의 가능한 메서드를 제공 합니다 `BindingContext` . 또한 클래스는 연결된 컨트롤에 대한 참조를 `AssociatedObject` 속성에 저장합니다.

EShopOnContainers 모바일 앱에는 `EventToCommandBehavior` 발생 하는 이벤트에 대 한 응답으로 명령을 실행 하는 클래스가 포함 되어 있습니다. 이 클래스는 동작이 `BindableBehavior<T>` 사용 될 때 동작에서 속성에 지정 된를 바인딩하고 실행할 수 있도록 클래스에서 파생 `ICommand` `Command` 됩니다. 다음 코드 예제는 `EventToCommandBehavior` 클래스를 보여줍니다.

```
public class EventToCommandBehavior : BindableBehavior<View>
{
    ...
    protected override void OnAttachedTo(View visualElement)
    {
        base.OnAttachedTo(visualElement);

        var events = AssociatedObject.GetType().GetRuntimeEvents().ToArray();
        if (events.Any())
        {
            _eventInfo = events.FirstOrDefault(e => e.Name == EventName);
            if (_eventInfo == null)
                throw new ArgumentException(string.Format(
                        "EventToCommand: Can't find any event named '{0}' on attached type",
                        EventName));

            AddEventHandler(_eventInfo, AssociatedObject, OnFired);
        }
    }

    protected override void OnDetachingFrom(View view)
    {
        if (_handler != null)
            _eventInfo.RemoveEventHandler(AssociatedObject, _handler);

        base.OnDetachingFrom(view);
    }

    private void AddEventHandler(
            EventInfo eventInfo, object item, Action<object, EventArgs> action)
    {
        ...
    }

    private void OnFired(object sender, EventArgs eventArgs)
    {
        ...
    }
}
```

`OnAttachedTo` 및 `OnDetachingFrom` 메서드는 속성에 정의 된 이벤트에 대 한 이벤트 처리기를 등록 및 등록 취소 하는 데 사용 됩니다 `EventName` . 그런 다음 이벤트가 발생 하면 메서드를 호출 하 여 `OnFired` 명령을 실행 합니다.

`EventToCommandBehavior` 이벤트가 발생할 때 명령을 실행 하는 데를 사용 하는 이점은 명령과 상호 작용 하도록 설계 되지 않은 컨트롤과 연결 될 수 있다는 것입니다. 또한 모델을 볼 수 있도록 이벤트 처리 코드를 이동 하 여 단위 테스트를 수행할 수 있습니다.

뷰에서 동작 호출

는 명령을 `EventToCommandBehavior` 지원 하지 않는 컨트롤에 명령을 연결 하는 데 특히 유용 합니다. 예를 들어,는 `ProfileView` 다음 코드와 같이를 사용 하 여 `EventToCommandBehavior` `OrderDetailCommand` `ItemTapped` `ListView` 사용자의 주문이 나열 된에서 이벤트가 발생 하는 경우를 실행 합니다.

```
<ListView>
    <ListView.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="ItemTapped"
            Command="{Binding OrderDetailCommand}"
            EventArgsConverter="{StaticResource ItemTappedEventArgsConverter}" />
    </ListView.Behaviors>
    ...
</ListView>
```

런타임에는와의 `EventToCommandBehavior` 상호 작용에 응답 합니다 `ListView` . 에서 항목을 선택 하면 `ListView` 이벤트가 발생 하 고 `ItemTapped` 에서이 실행 됩니다 `OrderDetailCommand` `ProfileViewModel` . 기본적으로 이벤트에 대 한 이벤트 인수가 명령에 전달 됩니다. 이 데이터는 속성에 지정 된 변환기에서 소스와 대상 사이에 전달 될 때 변환 되며에서의를 `EventArgsConverter` 반환 합니다 `Item` `ListView` `ItemTappedEventArgs` . 따라서 `OrderDetailCommand` 가 실행 되 면 선택한 `Order` 가 등록 된 작업에 매개 변수로 전달 됩니다.

동작에 대 한 자세한 내용은 동작을 참조 하세요.

요약

MVVM (모델-뷰-ViewModel) 패턴은 응용 프로그램의 비즈니스 및 프레젠테이션 논리를 UI (사용자 인터페이스)와 완전히 분리 하는 데 도움이 됩니다. 응용 프로그램 논리와 UI를 명확 하 게 분리 하면 수많은 개발 문제를 해결 하 고 응용 프로그램을 더 쉽게 테스트 하 고 유지 관리 하 고 진화 시킬 수 있습니다. 또한 코드 다시 사용 기회를 크게 향상 시킬 수 있으며, 개발자와 UI 디자이너는 앱의 각 부분을 개발할 때 더 쉽게 공동 작업을 수행할 수 있습니다.

MVVM 패턴을 사용 하 여 앱의 UI와 기본 프레젠테이션 및 비즈니스 논리는 세 개의 개별 클래스로 구분 됩니다. 뷰는 UI와 UI 논리를 캡슐화 합니다. 프레젠테이션 논리 및 상태를 캡슐화 하는 뷰 모델 그리고 모델은 앱의 비즈니스 논리 및 데이터를 캡슐화 합니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

종속성 주입

2020-06-05 • 28 minutes to read • Edit Online

일반적으로 클래스 생성자는 개체를 인스턴스화할 때 호출 되며, 개체가 필요로 하는 모든 값은 생성자에 인수로 전달 됩니다. 이는 종속성 주입의 한 예제 이며 특히 *생성자 주입*이라고 합니다. 개체에 필요한 종속성이 생성자에 삽입 됩니다.

종속성을 인터페이스 형식으로 지정 하 여 종속성 주입을 통해 이러한 형식에 종속 된 코드에서 구체적인 형식을 분리할 수 있습니다. 일반적으로 인터페이스와 추상 형식 간의 등록 및 매핑 목록과 이러한 형식을 구현 하거나 확장 하는 구체적인 형식을 포함 하는 컨테이너를 사용 합니다.

*속성 setter 삽입*및 *메서드 호출 삽입*과 같은 다른 유형의 종속성 주입도 있지만 일반적으로는 표시 되지 않습니다. 따라서이 챕터는 종속성 주입 컨테이너를 사용 하 여 생성자 삽입을 수행 하는 경우에만 중점을 둡니다.

종속성 주입 소개

종속성 주입은 IoC (반전 제어) 패턴의 특수화 된 버전입니다. 반전 하는 우려는 필요한 종속성을 가져오는 프로세스입니다. 종속성 주입을 사용 하는 경우 다른 클래스는 런타임에 개체에 종속성을 삽입 하는 일을 담당 합니다. 다음 코드 예제에서는 `ProfileViewModel` 종속성 주입을 사용할 때 클래스를 구성 하는 방법을 보여 줍니다.

```
public class ProfileViewModel : ViewModelBase
{
    private IOrderService _orderService;

    public ProfileViewModel(IOrderService orderService)
    {
        _orderService = orderService;
    }
    ...
}
```

`ProfileViewModel`생성자는 `IOrderService` 다른 클래스에 의해 삽입 된 인수로 인스턴스를 수신 합니다. `ProfileViewModel`인터페이스 형식에는 클래스의 유일한 종속성이 있습니다. 따라서 클래스는 `ProfileViewModel` 개체의 인스턴스화를 담당 하는 클래스를 알지 못합니다 `IOrderService` . 개체를 인스턴스화하고 클래스에 삽입 하는 것을 담당 하는 클래스를 `IOrderService` `ProfileViewModel` *종속성 주입 컨테이너*라고 합니다.

종속성 주입 컨테이너를 사용 하면 클래스 인스턴스를 인스턴스화하고 컨테이너의 구성에 따라 수명을 관리 하는 기능을 제공 하 여 개체 간의 결합을 줄일 수 있습니다. 개체를 만드는 동안 컨테이너는 개체에 필요한 모든 종속성을 삽입 합니다. 이러한 종속성을 아직 만들지 않은 경우 컨테이너는 먼저 종속성을 만들고 확인 합니다.

> **NOTE**
> 팩터리를 사용 하 여 종속성 주입을 수동으로 구현할 수도 있습니다. 그러나 컨테이너를 사용 하면 수명 관리, 어셈블리 검색을 통한 등록 등의 추가 기능이 제공 됩니다.

종속성 주입 컨테이너를 사용 하는 경우 다음과 같은 몇 가지 이점이 있습니다.

- 컨테이너는 클래스가 해당 종속성을 찾고 수명을 관리할 필요가 없습니다.
- 컨테이너를 사용 하면 클래스에 영향을 주지 않고 구현 된 종속성을 매핑할 수 있습니다.
- 컨테이너는 종속성을 모의 수 있도록 하 여 테스트 용이성을 용이 하 게 합니다.
- 컨테이너는 새 클래스를 앱에 쉽게 추가할 수 있도록 하 여 유지 관리 효율성을 높입니다.

Xamarin.FormsMVVM를 사용 하는 앱의 컨텍스트에서 종속성 주입 컨테이너는 일반적으로 뷰 모델을 등록 하 고 해결 하는 데 사용 되며, 서비스를 등록 하 고이를 뷰 모델에 삽입 하는 데 사용 됩니다.

TinyIoC를 사용 하 여 앱에서 뷰 모델 및 서비스 클래스의 인스턴스화를 관리 하는 eShopOnContainers 모바일 앱과 함께 사용할 수 있는 많은 종속성 주입 컨테이너가 있습니다. TinyIoC는 다양 한 컨테이너를 평가한 후에 선택 되었으며, 잘 알려진 대부분의 컨테이너와 비교할 때 모바일 플랫폼에서 뛰어난 성능을 제공 합니다. 느슨하게 연결 된 앱 빌드를 용이 하 게 하 고, 형식 매핑 등록, 개체 확인, 개체 수명 관리, 종속 개체를 확인 하는 개체의 생성자에 삽입 하는 메서드를 포함 하 여 종속성 주입 컨테이너에서 일반적으로 발견 되는 모든 기능을 제공 합니다. TinyIoC에 대 한 자세한 내용은 TinyIoC on github.com를 참조 하세요.

TinyIoC에서 `TinyIoCContainer` 형식은 종속성 주입 컨테이너를 제공 합니다. 그림 3-1에서는 개체를 인스턴스화 하고 `IOrderService` 클래스에 삽입 하는이 컨테이너를 사용할 때의 종속성을 보여 줍니다 `ProfileViewModel` .

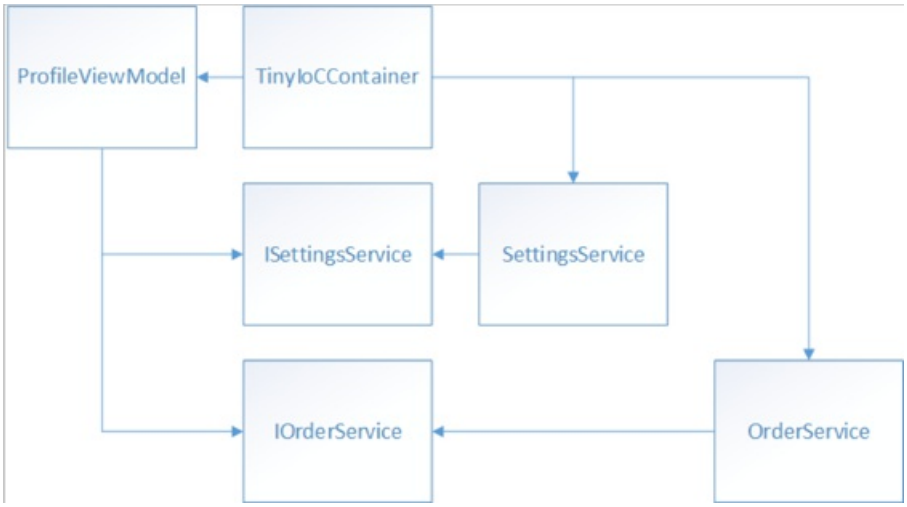

그림 3-1: 종속성 주입을 사용 하는 경우의 종속성

런타임에 개체를 인스턴스화하려면 컨테이너에서 인스턴스화해야 하는 인터페이스의 구현을 알아야 합니다 `IOrderService` `ProfileViewModel` . 여기에는 다음이 포함 됩니다.

- 컨테이너는 인터페이스를 구현 하는 개체를 인스턴스화하는 방법을 결정 `IOrderService` 합니다. 이를 *등록*이라고 합니다.
- 인터페이스를 구현 하는 개체 및 개체를 인스턴스화하는 컨테이너 `IOrderService` `ProfileViewModel` 입니다. 이를 *해결*이라고 합니다.

결국 응용 프로그램은 개체를 사용 하 여 완료 되 `ProfileViewModel` 고 가비지 수집에 사용할 수 있게 됩니다. 이 시점에서 `IOrderService` 다른 클래스가 동일한 인스턴스를 공유 하지 않는 경우 가비지 수집기는 인스턴스를 삭제 해야 합니다.

> **TIP**
> 컨테이너 독립적인 코드를 작성 합니다. 항상 컨테이너 독립적인 코드를 작성 하 여 사용 중인 특정 종속성 컨테이너에서 앱을 분리 합니다.

등록

종속성을 개체에 삽입 하려면 먼저 종속성의 형식을 컨테이너에 등록 해야 합니다. 일반적으로 형식을 등록 하려면 컨테이너에 인터페이스를 전달 하 고 인터페이스를 구현 하는 구체적인 형식을 전달 해야 합니다.

코드를 통해 컨테이너에 형식 및 개체를 등록 하는 방법에는 두 가지가 있습니다.

- 컨테이너에 형식 또는 매핑을 등록 합니다. 필요한 경우 컨테이너는 지정 된 형식의 인스턴스를 작성 합니다.
- 컨테이너의 기존 개체를 singleton으로 등록 합니다. 필요한 경우 컨테이너는 기존 개체에 대 한 참조를 반환

합니다.

> **TIP**
> 종속성 주입 컨테이너가 항상 적절 한 것은 아닙니다. 종속성 주입에는 작은 앱에 적합 하지 않거나 유용 하지 않을 수 있는 추가 복잡성 및 요구 사항이 도입 되었습니다. 클래스에 종속성이 없거나 다른 형식에 대 한 종속성이 아닌 경우 컨테이너에 배치 하는 것이 적합 하지 않을 수 있습니다. 또한 클래스에는 형식에 대 한 정수 계열 종속성 집합이 하나 있지만 변경 되지 않는 경우 컨테이너에 배치 하는 것이 적합 하지 않을 수 있습니다.

종속성 주입이 필요한 형식의 등록은 앱의 단일 메서드에서 수행 해야 하며, 앱이 해당 클래스 간의 종속성을 인식하도록 앱의 수명 주기 초기에이 메서드를 호출 해야 합니다. EShopOnContainers 모바일 앱에서이는 `ViewModelLocator` 개체를 빌드하고 `TinyIoCContainer` 해당 개체에 대 한 참조를 포함 하는 응용 프로그램의 유일한 클래스인 클래스에 의해 수행 됩니다. 다음 코드 예제에서는 eShopOnContainers 모바일 앱이 클래스에서 개체를 선언 하는 방법을 보여 줍니다 `TinyIoCContainer` `ViewModelLocator` .

```
private static TinyIoCContainer _container;
```

형식은 생성자에 등록 됩니다 `ViewModelLocator` . 이를 위해 먼저 `TinyIoCContainer` 다음 코드 예제에서 보여 주는 인스턴스를 만듭니다.

```
_container = new TinyIoCContainer();
```

그런 다음 형식이 개체에 등록 `TinyIoCContainer` 되며, 다음 코드 예제에서는 형식 등록의 가장 일반적인 형태를 보여 줍니다.

```
_container.Register<IRequestProvider, RequestProvider>();
```

`Register` 여기에 표시 된 메서드는 인터페이스 형식을 구체적 형식에 매핑합니다. 기본적으로 각 인터페이스 등록은 모든 종속 개체가 동일한 공유 인스턴스를 받도록 singleton으로 구성 됩니다. 따라서 생성자를 통해를 `RequestProvider` 삽입 해야 하는 개체에서 공유 하는 단일 인스턴스만 컨테이너에 존재 합니다 `IRequestProvider` .

구체적 형식은 다음 코드 예제와 같이 인터페이스 형식에서 매핑을 사용 하지 않고 직접 등록할 수도 있습니다.

```
_container.Register<ProfileViewModel>();
```

기본적으로 각 구체적 클래스 등록은 모든 종속 개체가 새 인스턴스를 받도록 다중 인스턴스로 구성 됩니다. 따라서 `ProfileViewModel` 이 확인 되 면 새 인스턴스가 만들어지고 컨테이너에서 필요한 종속성이 삽입 됩니다.

해결 방법

형식을 등록 한 후에는 종속성으로 확인 하거나 삽입할 수 있습니다. 형식을 확인 하 고 컨테이너가 새 인스턴스를 만들어야 하는 경우 모든 종속성을 인스턴스에 삽입 합니다.

일반적으로 형식이 확인 되 면 다음 세 가지 중 하나가 발생 합니다.

1. 형식이 등록 되지 않은 경우 컨테이너는 예외를 throw 합니다.
2. 형식이 singleton으로 등록 된 경우 컨테이너는 singleton 인스턴스를 반환 합니다. 이 형식이에 대해 처음으로 호출 되 면 컨테이너는 필요한 경우이를 만들고 해당에 대 한 참조를 유지 관리 합니다.
3. 형식이 singleton으로 등록 되지 않은 경우 컨테이너는 새 인스턴스를 반환 하 고이에 대 한 참조를 유지 하지 않습니다.

다음 코드 예제에서는 `RequestProvider` 이전에 TinyIoC에 등록 된 형식을 확인할 수 있는 방법을 보여 줍니다.

```
var requestProvider = _container.Resolve<IRequestProvider>();
```

이 예제에서는 TinyIoC에 `IRequestProvider` 종속성과 함께 형식에 대 한 구체적인 형식을 확인 하 라는 메시지가 표시 됩니다. 일반적으로 `Resolve` 메서드는 특정 형식의 인스턴스가 필요할 때 호출 됩니다. 확인 된 개체의 수명을 제어 하는 방법에 대 한 자세한 내용은 확인 된 개체의 수명 관리를 참조 하세요.

다음 코드 예제에서는 eShopOnContainers 모바일 앱이 뷰 모델 유형 및 해당 종속성을 인스턴스화하는 방법을 보여 줍니다.

```
var viewModel = _container.Resolve(viewModelType);
```

이 예에서는 요청 된 뷰 모델에 대 한 뷰 모델 유형을 확인 하도록 TinyIoC를 요청 하 고, 컨테이너는 종속성도 확인 합니다. 형식을 확인할 때 `ProfileViewModel` 확인할 종속성은 `ISettingsService` 개체와 `IOrderService` 개체입니다. 및 클래스를 등록할 때 인터페이스 등록이 사용 `SettingsService` 되었으므로 `OrderService` TinyIoC는 및 클래스에 대 한 singleton 인스턴스를 반환한 `SettingsService` `OrderService` 다음 클래스의 생성자에 전달 `ProfileViewModel` 합니다. EShopOnContainers 모바일 앱에서 모델을 생성 하 고 보기에 연결 하는 방법에 대 한 자세한 내용은 뷰 모델 로케이터를 사용 하 여 자동으로 뷰 모델 만들기를 참조 하세요.

> **NOTE**
> 컨테이너를 사용하여 형식을 등록하고 확인하는 경우, 특히 앱의 각 페이지 탐색에 대한 종속성을 다시 생성하는 경우 컨테이너의 리플렉션 사용으로 인해 성능비용이 발생합니다. 종속성이 많거나 깊은 경우에는 생성 비용이 많이 증가할 수 있습니다.

확인 된 개체의 수명 관리

구체적 클래스 등록을 사용 하 여 형식을 등록 한 후 TinyIoC의 기본 동작은 형식이 확인 될 때마다 또는 종속성 메커니즘이 다른 클래스에 인스턴스를 삽입 하는 경우에 등록 된 형식의 새 인스턴스를 만드는 것입니다. 이 시나리오에서 컨테이너는 확인 된 개체에 대 한 참조를 보유 하지 않습니다. 그러나 인터페이스 등록을 사용 하 여 형식을 등록 하는 경우 TinyIoC의 기본 동작은 개체의 수명을 단일 항목으로 관리 하는 것입니다. 따라서 인스턴스가 범위 내에 있는 동안 인스턴스는 범위 내에 유지 되 고, 컨테이너가 범위를 벗어나면 가비지가 수집 되 고, 코드에서 명시적으로 컨테이너를 삭제 하면 삭제 됩니다.

기본 TinyIoC 등록 동작은 흐름 `AsSingleton` 및 API 메서드를 사용 하 여 재정의할 수 있습니다 `AsMultiInstance` . 예를 들어 메서드를 `AsSingleton` 메서드와 함께 사용 하 여 `Register` 컨테이너에서 메서드를 호출할 때 형식의 singleton 인스턴스를 만들거나 반환할 수 있습니다 `Resolve` . 다음 코드 예제에서는 TinyIoC가 클래스의 단일 인스턴스를 만들도록 지시 하는 방법을 보여 줍니다 `LoginViewModel` .

```
_container.Register<LoginViewModel>().AsSingleton();
```

`LoginViewModel` 형식이 처음 확인 될 때 컨테이너는 새 개체를 만들고 `LoginViewModel` 해당 개체에 대 한 참조를 유지 관리 합니다. 의 이후 해상도에서 `LoginViewModel` 컨테이너는 이전에 만든 개체에 대 한 참조를 반환 합니다 `LoginViewModel` .

> **NOTE**
> 단일 항목로 등록 된 형식은 컨테이너가 삭제 될 때 삭제 됩니다.

요약

종속성 주입을 통해 이러한 형식에 종속 된 코드에서 구체적인 형식을 분리할 수 있습니다. 일반적으로 인터페이스와 추상 형식 간의 등록 및 매핑 목록과 이러한 형식을 구현 하거나 확장 하는 구체적인 형식을 포함 하는 컨테이너를 사용 합니다.

TinyIoC는 잘 알려진 대부분의 컨테이너와 비교할 때 모바일 플랫폼에서 뛰어난 성능을 제공 하는 경량 컨테이너입니다. 느슨하게 연결 된 앱 빌드를 용이 하 게 하 고, 형식 매핑 등록, 개체 확인, 개체 수명 관리 및 종속 개체를 확인 하는 개체의 생성자에 포함 하는 메서드를 포함 하 여 종속성 주입 컨테이너에서 일반적으로 발견 되는 모든 기능을 제공 합니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

느슨하게 결합된 구성 요소 간 통신

2020-06-05 • 17 minutes to read • Edit Online

게시-구독 패턴은 게시자가 구독자로 알려진 수신자를 몰라도 메시지를 보내는 메시징 패턴입니다. 마찬가지로 구독자는 게시자를 전혀 알지 못해도 특정 메시지를 수신 대기합니다.

.NET의 이벤트는 게시-구독 패턴을 구현하는데, 이는 컨트롤과 컨트롤을 포함하는 페이지처럼 느슨한 결합이 필요하지 않은 경우 구성 요소 간의 통신 레이어를 위한 가장 간단한 접근 방식입니다. 그러나 게시자 수명과 구독자 수명은 상호 개체 참조에 의해 결합되어 있으며, 구독자 유형에는 게시자 유형에 대한 참조가 있어야 합니다. 이로 인해 특히 정적 이벤트를 구독하는 수명이 짧은 개체 또는 수명이 긴 개체가 있는 경우 메모리 관리 문제가 발생할 수 있습니다. 이벤트 처리기가 제거되지 않은 경우, 구독자는 게시자에 있는 해당 구독자에 대한 참조에 의해 활성 상태로 유지되고, 이로 인해 구독자의 가비지 수집이 방지 또는 지연됩니다.

MessagingCenter 소개

Xamarin.Forms `MessagingCenter` 클래스는 게시-구독 패턴을 구현 하 여 개체 및 형식 참조로 연결 하기 불편 한 구성 요소 간에 메시지 기반 통신을 허용 합니다. 이 메커니즘을 통해 게시자와 구독자는 서로에 대 한 참조 없이 통신할 수 있으며, 구성 요소 간의 종속성을 줄이고 구성 요소를 독립적으로 개발 하 고 테스트할 수 있습니다.

`MessagingCenter` 클래스는 멀티 캐스트 게시-구독 기능을 제공 합니다. 즉, 단일 메시지를 게시 하는 여러 게시자가 있을 수 있으며 동일한 메시지를 수신 대기 하는 여러 구독자가 있을 수 있습니다. 그림 4-1에서는이 관계를 보여 줍니다.

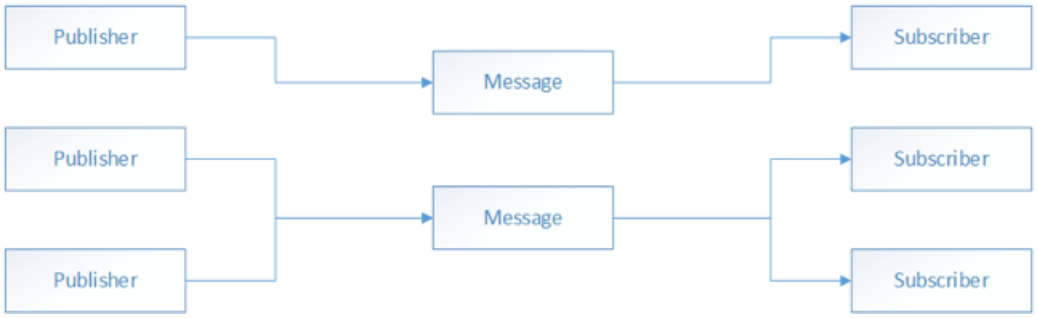

그림 4-1: 멀티 캐스트 게시-구독 기능

게시자는 메서드를 사용 하 여 메시지 `MessagingCenter.Send` 를 보내며, 구독자는 메서드를 사용 하 여 메시지를 수신 대기 `MessagingCenter.Subscribe` 합니다. 또한 구독자는 필요한 경우 메서드를 사용 하 여 메시지 구독에서 구독을 취소할 수 있습니다 `MessagingCenter.Unsubscribe` .

내부적으로 `MessagingCenter` 클래스는 약한 참조를 사용 합니다. 즉, 개체가 활성 상태로 유지되지 않으며 가비지 수집될 수 있습니다. 따라서 클래스가 더 이상 메시지를 받지 않으려는 경우에만 메시지의 구독을 취소해야 합니다.

EShopOnContainers 모바일 앱은 클래스를 사용 하 여 `MessagingCenter` 느슨하게 결합 된 구성 요소 간에 통신 합니다. 앱은 세 개의 메시지를 정의 합니다.

- `AddProduct`이 메시지는 바구니 `CatalogViewModel` 에 항목이 추가 될 때 클래스에 의해 게시 됩니다. 반환 시 클래스는 `BasketViewModel` 메시지를 구독 하 고 장바구니의 항목 수를 응답으로 증가 시킵니다. 또한 `BasketViewModel` 클래스는이 메시지에서 구독을 취소 합니다.
- `Filter`이 메시지는 `CatalogViewModel` 사용자가 카탈로그에서 표시 된 항목에 브랜드 또는 형식 필터를 적용 할 때 클래스에 의해 게시 됩니다. 반환 시 클래스는 `CatalogView` 메시지를 구독 하 고 필터 조건과 일치 하는 항목만 표시 되도록 UI를 업데이트 합니다.

- `ChangeTab`이 메시지는가 성공적으로 생성 되 고 `MainViewModel` `CheckoutViewModel` 새 주문의 전송으로 이동하면 클래스에 의해 게시 됩니다 `MainViewModel` . 반환 시 클래스는 `MainView` 메시지를 구독 하 고 사용자의 주문을 표시 하기 위해 **내 프로필** 탭이 활성화 되도록 UI를 업데이트 합니다.

> **NOTE**
> 클래스는 `MessagingCenter` 느슨하게 결합 된 클래스 간의 통신을 허용 하지만이 문제에 대 한 유일한 아키텍처 솔루션은 제공 하지 않습니다. 예를 들어 뷰 모델과 뷰 간의 통신은 바인딩 엔진과 속성 변경 알림을 통해 구현할 수도 있습니다. 또한 탐색 하는 동안 데이터를 전달 하 여 두 뷰 모델 간의 통신을 수행할 수도 있습니다.

EShopOnContainers 모바일 앱에서 `MessagingCenter` 는 다른 클래스에서 발생 하는 동작에 대 한 응답으로 UI에서를 업데이트 하는 데 사용 됩니다. 따라서 메시지는 동일한 스레드에서 메시지를 받는 구독자와 함께 UI 스레드에 게시 됩니다.

> **TIP**
> UI 업데이트를 수행할 때 UI 스레드로 마샬링합니다. UI를 업데이트 하기 위해 백그라운드 스레드에서 보낸 메시지가 필요한 경우 메서드를 호출 하 여 구독자의 UI 스레드에서 메시지를 처리 합니다 `Device.BeginInvokeOnMainThread` .

에 대 한 자세한 내용은 `MessagingCenter` Messagingcenter를 참조 하세요.

메시지 정의

`MessagingCenter`메시지는 메시지를 식별 하는 데 사용 되는 문자열입니다. 다음 코드 예제에서는 eShopOnContainers 모바일 앱 내에서 정의 된 메시지를 보여 줍니다.

```
public class MessengerKeys
{
    // Add product to basket
    public const string AddProduct = "AddProduct";

    // Filter
    public const string Filter = "Filter";

    // Change selected Tab programmatically
    public const string ChangeTab = "ChangeTab";
}
```

이 예제에서 메시지는 상수를 사용 하 여 정의 됩니다. 이 방법의 장점은 컴파일 시간 형식 안전성 및 리팩터링 지원을 제공 한다는 것입니다.

메시지 게시

게시자는 오버 로드 중 하나를 사용 하 여 구독자에 게 메시지를 알립니다 `MessagingCenter.Send` . 다음 코드 예제에서는 메시지를 게시 하는 방법을 보여 줍니다 `AddProduct` .

```
MessagingCenter.Send(this, MessengerKeys.AddProduct, catalogItem);
```

이 예제에서 메서드는 `Send` 세 개의 인수를 지정 합니다.

- 첫 번째 인수는 sender 클래스를 지정 합니다. 보낸 사람 클래스는 메시지를 받으려는 모든 구독자가 지정 해야 합니다.
- 두 번째 인수는 메시지를 지정합니다.

- 세 번째 인수는 구독자로 보낼 페이로드 데이터를 지정 합니다. 이 경우 페이로드 데이터는 `CatalogItem` 인스턴스입니다.

`Send`메서드는 화재 및 잊은 방법을 사용 하 여 메시지와 해당 페이로드 데이터를 게시 합니다. 따라서 메시지를 수신하도록 등록된 구독자가 없는 경우에도 메시지가 전송됩니다. 이 경우 보낸 메시지는 무시됩니다.

> **NOTE**
> `MessagingCenter.Send`메서드는 제네릭 매개 변수를 사용 하 여 메시지를 배달 하는 방법을 제어할 수 있습니다. 따라서 여러 다른 구독자가 메시지 id를 공유 하지만 다른 페이로드 데이터 형식을 보내는 여러 메시지를 수신할 수 있습니다.

메시지 구독

구독자는 오버 로드 중 하나를 사용 하 여 메시지를 받도록 등록할 수 있습니다 `MessagingCenter.Subscribe` . 다음 코드 예제에서는 eShopOnContainers mobile 앱이 메시지를 구독 하 고 처리 하는 방법을 보여 줍니다 `AddProduct` .

```
MessagingCenter.Subscribe<CatalogViewModel, CatalogItem>(
    this, MessageKeys.AddProduct, async (sender, arg) =>
{
    BadgeCount++;

    await AddCatalogItemAsync(arg);
});
```

이 예제에서 메서드는 `Subscribe` 메시지를 구독 하 `AddProduct` 고 메시지 수신에 대 한 응답으로 콜백 대리자를 실행 합니다. 람다 식으로 지정 된이 콜백 대리자는 UI를 업데이트 하는 코드를 실행 합니다.

> **TIP**
> 변경할 수 없는 페이로드 데이터를 사용 하는 것이 좋습니다. 여러 스레드가 받은 데이터에 동시에 액세스할 수 있으므로 콜백 대리자 내에서 페이로드 데이터를 수정 하지 마세요. 이 시나리오에서는 동시성 오류가 발생 하지 않도록 페이로드 데이터를 변경할 수 없습니다.

구독자는 게시 된 메시지의 모든 인스턴스를 처리할 필요가 없으며이는 메서드에 지정 된 제네릭 형식 인수에 의해 제어 될 수 있습니다 `Subscribe` . 이 예에서 구독자는 `AddProduct` `CatalogViewModel` 페이로드 데이터가 인스턴스인 클래스에서 전송 된 메시지만 받습니다 `CatalogItem` .

메시지에서 구독 취소

구독자는 더 이상 수신하지 않을 메시지를 구독 취소할 수 있습니다. 이는 `MessagingCenter.Unsubscribe` 다음 코드 예제에서 보여 주는 것 처럼 오버 로드 중 하나를 사용 하 여 수행 됩니다.

```
MessagingCenter.Unsubscribe<CatalogViewModel, CatalogItem>(this, MessengerKeys.AddProduct);
```

이 예제에서 `Unsubscribe` 메서드 구문은 메시지 수신을 구독할 때 지정 된 형식 인수를 반영 합니다 `AddProduct` .

요약

Xamarin.Forms `MessagingCenter` 클래스는 게시-구독 패턴을 구현 하 여 개체 및 형식 참조로 연결 하기 불편 한 구성 요소 간에 메시지 기반 통신을 허용 합니다. 이 메커니즘을 통해 게시자와 구독자는 서로에 대 한 참조 없이 통신할 수 있으며, 구성 요소 간의 종속성을 줄이고 구성 요소를 독립적으로 개발 하 고 테스트할 수 있습니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

엔터프라이즈 앱 탐색

2020-06-05 • 32 minutes to read • Edit Online

Xamarin.Forms에는 내부 논리 기반 상태 변경으로 인해 일반적으로 사용자가 UI와 상호 작용 하거나 앱 자체에서 발생 하는 페이지 탐색에 대 한 지원이 포함 됩니다. 그러나 다음 문제를 충족 해야 하므로 MVVM (모델-뷰-ViewModel) 패턴을 사용 하는 앱에서 탐색을 구현 하는 것이 복잡할 수 있습니다.

- 보기 간에 밀접 한 결합 및 종속성을 사용 하지 않는 방법을 사용 하 여 탐색할 뷰를 식별 하는 방법입니다.
- 탐색할 뷰가 인스턴스화되어 초기화 되는 프로세스를 조정 하는 방법입니다. MVVM를 사용 하는 경우 뷰 및 뷰 모델을 인스턴스화하고 뷰의 바인딩 컨텍스트를 통해 서로 연결 해야 합니다. 앱에서 종속성 주입 컨테이너를 사용 하는 경우 뷰 및 뷰 모델 인스턴스화에 특정 생성 메커니즘이 필요할 수 있습니다.
- 뷰 우선 탐색을 수행할지 아니면 모델 우선 탐색을 볼 지 여부를 지정 합니다. 뷰 우선 탐색을 사용 하 여 탐색 할 페이지는 보기 유형의 이름을 참조 합니다. 탐색 하는 동안 해당 뷰 모델 및 기타 종속 서비스와 함께 지정 된 뷰가 인스턴스화됩니다. 다른 방법으로는 탐색할 페이지가 뷰 모델 형식의 이름을 참조 하는 모델 우선 탐색 보기를 사용 하는 것이 좋습니다.
- 보기 및 뷰 모델에서 앱의 탐색 동작을 명확 하 게 구분 하는 방법입니다. MVVM 패턴은 앱의 UI와 해당 프레젠테이션 및 비즈니스 논리를 구분 합니다. 그러나 앱의 탐색 동작은 종종 응용 프로그램의 UI 및 프레젠테이션 파트를 포괄 합니다. 사용자는 종종 뷰에서 탐색을 시작 하 고 탐색의 결과로 보기가 대체 됩니다. 그러나 탐색은 종종 뷰 모델 내에서 시작 하거나 조정 해야 할 수도 있습니다.
- 초기화를 위해 탐색 하는 동안 매개 변수를 전달 하는 방법입니다. 예를 들어 사용자가 뷰로 이동 하 여 주문 세부 정보를 업데이트 하는 경우 올바른 데이터를 표시할 수 있도록 주문 데이터를 뷰에 전달 해야 합니다.
- 특정 비즈니스 규칙을 맞는 확인 하기 위해 교차 좌표 탐색 하는 방법입니다. 예를 들어 잘못 된 데이터를 수정 하거나 보기 내에서 수행 된 데이터 변경 내용을 제출 하거나 취소 하 라는 메시지가 표시 되도록 보기에서 이동 하기 전에 사용자에 게 메시지가 표시 될 수 있습니다.

이 장에서는 `NavigationService` 모델 우선 페이지 탐색 보기를 수행 하는 데 사용 되는 클래스를 제공 하 여 이러한 문제를 해결 합니다.

> **NOTE**
>
> `NavigationService` 앱에서 사용 하는은 ContentPage 인스턴스 간에 계층적 탐색을 수행 하는 용도로만 디자인 되었습니다. 서비스를 사용 하 여 다른 페이지 형식 사이를 탐색 하면 예기치 않은 동작이 발생할 수 있습니다.

페이지 간 이동

탐색 논리는 뷰의 코드 숨김이 나 데이터 바인딩된 뷰 모델에 있을 수 있습니다. 보기에 탐색 논리를 배치 하는 것이 가장 간단한 방법일 수 있지만 단위 테스트를 통해 쉽게 테스트할 수는 없습니다. 뷰 모델 클래스에 탐색 논리를 배치 하면 단위 테스트를 통해 논리를 실행할 수 있음을 의미 합니다. 또한 뷰 모델은 특정 비즈니스 규칙이 적용 되도록 탐색을 제어 하는 논리를 구현할 수 있습니다. 예를 들어 앱에서 사용자가 처음에 입력 한 데이터가 유효한 지 확인 하지 않고 페이지에서 다른 위치로 이동 하는 것을 허용 하지 않을 수 있습니다.

`NavigationService` 일반적으로 클래스는 테스트를 촉진 하기 위해 뷰 모델에서 호출 됩니다. 그러나 뷰 모델에서 뷰로 이동 하려면 뷰 모델에서 뷰를 참조 해야 하며, 특히 활성 뷰 모델이 연결 되지 않은 뷰를 사용 하는 것이 좋습니다. 따라서 여기에 `NavigationService` 표시 된는 뷰 모델 유형을 탐색할 대상으로 지정 합니다.

EShopOnContainers 모바일 앱은 클래스를 사용 하 여 `NavigationService` 뷰 모델-첫 번째 탐색을 제공 합니다. 이 클래스는 `INavigationService` 다음 코드 예제에 표시 된 인터페이스를 구현 합니다.

```
public interface INavigationService
{
    ViewModelBase PreviousPageViewModel { get; }
    Task InitializeAsync();
    Task NavigateToAsync<TViewModel>() where TViewModel : ViewModelBase;
    Task NavigateToAsync<TViewModel>(object parameter) where TViewModel : ViewModelBase;
    Task RemoveLastFromBackStackAsync();
    Task RemoveBackStackAsync();
}
```

이 인터페이스는 구현 하는 클래스에서 다음 메서드를 제공 하도록 지정 합니다.

메서드	목적
`InitializeAsync`	앱이 시작 될 때 두 페이지 중 하나를 탐색 합니다.
`NavigateToAsync`	지정 된 페이지에 대 한 계층적 탐색을 수행 합니다.
`NavigateToAsync(parameter)`	지정 된 페이지에 대 한 계층적 탐색을 수행 하 여 매개 변수를 전달 합니다.
`RemoveLastFromBackStackAsync`	탐색 스택에서 이전 페이지를 제거 합니다.
`RemoveBackStackAsync`	탐색 스택에서 이전 페이지를 모두 제거 합니다.

또한 `INavigationService` 인터페이스는 구현 하는 클래스에서 속성을 제공 하도록 지정 합니다 `PreviousPageViewModel` . 이 속성은 탐색 스택의 이전 페이지와 연결 된 뷰 모델 유형을 반환 합니다.

> **NOTE**
> `INavigationService` 인터페이스는 일반적으로 메서드를 지정 `GoBackAsync` 합니다 .이 메서드를 사용 하 여 프로그래밍 방식으로 탐색 스택의 이전 페이지로 돌아갈 수 있습니다. 그러나이 메서드는 필요 하지 않으므로 eShopOnContainers 모바일 앱에서 누락 되었습니다.

NavigationService 인스턴스 만들기

인터페이스를 구현 하는 `NavigationService` 클래스는 `INavigationService` 다음 코드 예제에서 보여 주는 것 처럼 Autofac 종속성 주입 컨테이너를 사용 하 여 singleton으로 등록 됩니다.

```
builder.RegisterType<NavigationService>().As<INavigationService>().SingleInstance();
```

`INavigationService` 인터페이스는 `ViewModelBase` 다음 코드 예제에서 보여 주는 것 처럼 클래스 생성자에서 확인 됩니다.

```
NavigationService = ViewModelLocator.Resolve<INavigationService>();
```

그러면 `NavigationService` 클래스의 메서드에서 만든 Autofac 종속성 주입 컨테이너에 저장 된 개체에 대 한 참조가 반환 됩니다 `InitNavigation` `App` . 자세한 내용은 앱이 시작 될 때 탐색을 참조 하세요.

`ViewModelBase` 클래스는 `NavigationService` 형식의 속성에 인스턴스를 저장 합니다 `NavigationService` `INavigationService` . 따라서 클래스에서 파생 되는 모든 뷰 모델 클래스는 `ViewModelBase` 속성을 사용 하 여 `NavigationService` 인터페이스에 지정 된 메서드에 액세스할 수 있습니다 `INavigationService` . 그러면 `NavigationService` Autofac 종속성 주입 컨테이너에서 각 뷰 모델 클래스로 개체를 삽입 하는 오버 헤드가 발생

하지 않습니다.

탐색 요청 처리

Xamarin.Forms사용자가 `NavigationPage` 필요에 따라 페이지를 앞뒤로 탐색할 수 있는 계층적 탐색 환경을 구현하는 클래스를 제공 합니다. 계층적 탐색에 대한 자세한 내용은 계층적 탐색을 참조하세요.

클래스를 직접 사용 하는 대신 `NavigationPage` eShopOnContainers 앱은 `NavigationPage` `CustomNavigationView` 다음 코드 예제와 같이 클래스를 클래스에 래핑합니다.

```
public partial class CustomNavigationView : NavigationPage
{
    public CustomNavigationView() : base()
    {
        InitializeComponent();
    }

    public CustomNavigationView(Page root) : base(root)
    {
        InitializeComponent();
    }
}
```

이 래핑의 목적은 `NavigationPage` 클래스의 XAML 파일 내에서 인스턴스의 스타일을 쉽게 지정 하기 위한 것입니다.

탐색은 `NavigateToAsync` 다음 코드 예제와 같이 탐색 중인 페이지의 뷰 모델 형식을 지정 하는 메서드 중 하나를 호출 하 여 뷰 모델 클래스 내에서 수행 됩니다.

```
await NavigationService.NavigateToAsync<MainViewModel>();
```

다음 코드 예제에서는 `NavigateToAsync` 클래스에서 제공 하는 메서드를 보여 줍니다 `NavigationService` .

```
public Task NavigateToAsync<TViewModel>() where TViewModel : ViewModelBase
{
    return InternalNavigateToAsync(typeof(TViewModel), null);
}

public Task NavigateToAsync<TViewModel>(object parameter) where TViewModel : ViewModelBase
{
    return InternalNavigateToAsync(typeof(TViewModel), parameter);
}
```

각 메서드는 클래스에서 파생 되는 모든 뷰 모델 클래스에서 `ViewModelBase` 메서드를 호출 하 여 계층적 탐색을 수행할 수 있도록 `InternalNavigateToAsync` 합니다. 또한 두 번째 방법에서는 `NavigateToAsync` 탐색 데이터를 탐색 중인 뷰 모델에 전달 되는 인수로 지정할 수 있습니다. 여기서는 일반적으로 초기화를 수행 하는 데 사용 됩니다. 자세한 내용은 탐색 하는 동안 매개 변수 전달을 참조 하세요.

`InternalNavigateToAsync` 메서드는 탐색 요청을 실행 하 고 다음 코드 예제에 표시 됩니다.

```
private async Task InternalNavigateToAsync(Type viewModelType, object parameter)
{
    Page page = CreatePage(viewModelType, parameter);

    if (page is LoginView)
    {
        Application.Current.MainPage = new CustomNavigationView(page);
    }
    else
    {
        var navigationPage = Application.Current.MainPage as CustomNavigationView;
        if (navigationPage != null)
        {
            await navigationPage.PushAsync(page);
        }
        else
        {
            Application.Current.MainPage = new CustomNavigationView(page);
        }
    }

    await (page.BindingContext as ViewModelBase).InitializeAsync(parameter);
}

private Type GetPageTypeForViewModel(Type viewModelType)
{
    var viewName = viewModelType.FullName.Replace("Model", string.Empty);
    var viewModelAssemblyName = viewModelType.GetTypeInfo().Assembly.FullName;
    var viewAssemblyName = string.Format(
                CultureInfo.InvariantCulture, "{0}, {1}", viewName, viewModelAssemblyName);
    var viewType = Type.GetType(viewAssemblyName);
    return viewType;
}

private Page CreatePage(Type viewModelType, object parameter)
{
    Type pageType = GetPageTypeForViewModel(viewModelType);
    if (pageType == null)
    {
        throw new Exception($"Cannot locate page type for {viewModelType}");
    }

    Page page = Activator.CreateInstance(pageType) as Page;
    return page;
}
```

`InternalNavigateToAsync` 메서드는 먼저 메서드를 호출 하 여 뷰 모델에 대 한 탐색을 수행 합니다 `CreatePage` . 이 메서드는 지정 된 뷰 모델 유형에 해당 하는 뷰를 찾고이 뷰 유형의 인스턴스를 만들고 반환 합니다. 뷰 모델 유형에 해당 하는 뷰를 찾는 것은 다음을 전제로 하는 규칙 기반 방법을 사용 합니다.

- 뷰는 뷰 모델 유형과 동일한 어셈블리에 있습니다.
- 보기는에 있습니다. 뷰 하위 네임 스페이스입니다.
- 뷰 모델은에 있습니다. ViewModels 자식 네임 스페이스입니다.
- 뷰 이름은 "모델"이 제거 된 모델 이름 보기에 해당 합니다.

뷰가 인스턴스화된 경우 해당 뷰 모델에 연결 됩니다. 이를 수행 하는 방법에 대 한 자세한 내용은 뷰 모델 로케이터를 사용 하 여 자동으로 뷰 모델 만들기를 참조 하세요.

만드는 뷰가 인 경우 `LoginView` 클래스의 새 인스턴스 내에 래핑되어 `CustomNavigationView` 속성에 할당 됩니다 `Application.Current.MainPage` . 그렇지 않으면 `CustomNavigationView` 인스턴스가 검색 되 고 null이 아닌 경우 `PushAsync` 메서드를 호출 하 여 생성 되는 뷰를 탐색 스택으로 푸시합니다. 그러나 검색 된 인스턴스가 인 경우에는 `CustomNavigationView` `null` 만들어지는 뷰가 클래스의 새 인스턴스 내에 래핑되어 `CustomNavigationView` 속

성에 할당 됩니다 `Application.Current.MainPage` . 이 메커니즘을 사용 하면 탐색 하는 동안 페이지가 비어 있을 때와 데이터를 포함 하는 경우 탐색 스택에 올바르게 추가 됩니다.

> **TIP**
> 페이지 캐싱을 고려 합니다. 페이지 캐싱은 현재 표시 되지 않은 뷰에 대해 메모리 소비가 발생 합니다. 그러나 페이지 캐싱이 없으면 페이지 및 해당 뷰 모델의 XAML 구문 분석과 생성이 새 페이지를 탐색할 때마다 발생 하며,이는 복잡 한 페이지의 성능에 영향을 줄 수 있습니다. 컨트롤을 많이 사용 하지 않는 잘 디자인 된 페이지의 경우 성능이 충분 해야 합니다. 그러나 페이지 캐싱은 저속 페이지 로드 시간이 발생 하는 경우에 유용할 수 있습니다.

뷰가 생성 되 고 탐색 되 면 `InitializeAsync` 뷰의 연결 된 뷰 모델의 메서드가 실행 됩니다. 자세한 내용은 탐색 하는 동안 매개 변수 전달을 참조 하세요.

앱이 시작 될 때 탐색

앱이 시작 되 면 `InitNavigation` 클래스의 메서드가 `App` 호출 됩니다. 다음 코드 예제에서는 이 메서드를 보여줍니다.

```
private Task InitNavigation()
{
    var navigationService = ViewModelLocator.Resolve<INavigationService>();
    return navigationService.InitializeAsync();
}
```

메서드는 `NavigationService` 메서드를 호출 하기 전에 Autofac 종속성 주입 컨테이너에 새 개체를 만들고 해당 개체에 대 한 참조를 반환 합니다 `InitializeAsync` .

> **NOTE**
> 인터페이스를 `INavigationService` 클래스에서 확인 하면 `ViewModelBase` 해당 컨테이너는 `NavigationService` initnavigation 메서드가 호출 될 때 생성 된 개체에 대 한 참조를 반환 합니다.

다음 코드 예제는 `NavigationService` `InitializeAsync` 메서드를 보여줍니다.

```
public Task InitializeAsync()
{
    if (string.IsNullOrEmpty(Settings.AuthAccessToken))
        return NavigateToAsync<LoginViewModel>();
    else
        return NavigateToAsync<MainViewModel>();
}
```

`MainView` 앱에 인증에 사용 되는 캐시 된 액세스 토큰이 있는 경우를 탐색 합니다. 그렇지 않으면를 `LoginView` 탐색 합니다.

Autofac 종속성 주입 컨테이너에 대 한 자세한 내용은 종속성 주입 소개를 참조 하세요.

탐색 하는 동안 매개 변수 전달

인터페이스에서 지정 하는 메서드 중 하나를 사용 하 여 `NavigateToAsync` `INavigationService` 탐색 데이터를 탐색 중인 뷰 모델에 전달 되는 인수로 지정할 수 있습니다. 여기서는 일반적으로 초기화를 수행 하는 데 사용 됩니다.

예를 들어, `ProfileViewModel` 클래스에는 `OrderDetailCommand` 사용자가 페이지에서 순서를 선택할 때 실행 되는가 포함 됩니다 `ProfileView` . 그러면 `OrderDetailAsync` 다음 코드 예제에 표시 된 메서드를 실행 합니다.

```
private async Task OrderDetailAsync(Order order)
{
    await NavigationService.NavigateToAsync<OrderDetailViewModel>(order);
}
```

이 메서드는에 대 한 탐색을 호출 하 여 `OrderDetailViewModel` `Order` 페이지에서 사용자가 선택한 순서를 나타내는 인스턴스를 전달 합니다 `ProfileView` . `NavigationService` 클래스가를 만들면 `OrderDetailView` `OrderDetailViewModel` 클래스가 인스턴스화되고 뷰의에 할당 됩니다 `BindingContext` . 로 이동한 후 `OrderDetailView` `InternalNavigateToAsync` 메서드는 `InitializeAsync` 뷰의 연결 된 뷰 모델의 메서드를 실행 합니다.

`InitializeAsync` 메서드는 재정의 가능한 메서드로 클래스에서 정의 됩니다 `ViewModelBase` . 이 메서드는 `object` 탐색 작업을 수행 하는 동안 뷰 모델에 전달할 데이터를 나타내는 인수를 지정 합니다. 따라서 탐색 작업에서 데이터를 수신 하려는 뷰 모델 클래스는 `InitializeAsync` 필요한 초기화를 수행 하기 위해 메서드의 고유한 구현을 제공 합니다. 다음 코드 예제에서는 클래스의 메서드를 보여 줍니다 `InitializeAsync` `OrderDetailViewModel` .

```
public override async Task InitializeAsync(object navigationData)
{
    if (navigationData is Order)
    {
        ...
        Order = await _ordersService.GetOrderAsync(
                        Convert.ToInt32(order.OrderNumber), authToken);
        ...
    }
}
```

이 메서드는 `Order` 탐색 작업 중에 뷰 모델로 전달 된 인스턴스를 검색 하 고이 인스턴스를 사용 하 여 인스턴스에서 전체 주문 세부 정보를 검색 합니다 `OrderService` .

동작을 사용 하 여 탐색 호출

탐색은 일반적으로 사용자 조작을 통해 뷰에서 트리거됩니다. 예를 들어는 `LoginView` 성공적인 인증 후에 탐색을 수행 합니다. 다음 코드 예제에서는 동작에 따라 탐색을 호출 하는 방법을 보여 줍니다.

```
<WebView ...>
    <WebView.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="Navigating"
            EventArgsConverter="{StaticResource WebNavigatingEventArgsConverter}"
            Command="{Binding NavigateCommand}" />
    </WebView.Behaviors>
</WebView>
```

런타임에는와의 `EventToCommandBehavior` 상호 작용에 응답 합니다 `WebView` . 에서 `WebView` 웹 페이지로 이동 하면 이벤트가 발생 하 고 `Navigating` 에서이 실행 됩니다 `NavigateCommand` `LoginViewModel` . 기본적으로 이벤트에 대 한 이벤트 인수가 명령에 전달 됩니다. 이 데이터는 속성에 지정 된 변환기에서 소스와 대상 사이에 전달 될 때 변환 되며 `EventArgsConverter` 에서을 반환 합니다 `Url` `WebNavigatingEventArgs` . 따라서을 실행 하면 `NavigationCommand` 웹 페이지의 Url이 등록 된에 매개 변수로 전달 됩니다 `Action` .

그러면는 `NavigationCommand` `NavigateAsync` 다음 코드 예제에 표시 된 메서드를 실행 합니다.

```
private async Task NavigateAsync(string url)
{
    ...
    await NavigationService.NavigateToAsync<MainViewModel>();
    await NavigationService.RemoveLastFromBackStackAsync();
    ...
}
```

이 메서드는에 대 한 탐색을 호출 하 고, 탐색 다음에 탐색 `MainViewModel` 스택에서 페이지를 제거 합니다 `LoginView` .

탐색 확인 또는 취소

사용자가 탐색을 확인 하거나 취소할 수 있도록 앱이 탐색 작업 중에 사용자와 상호 작용 해야 할 수 있습니다. 예를 들어 사용자가 데이터 입력 페이지를 완전히 완료 하기 전에 탐색 하려고 할 때이 작업이 필요할 수 있습니다. 이 경우 앱은 사용자가 페이지에서 다른 곳으로 이동 하거나 이동 하기 전에 탐색 작업을 취소할 수 있는 알림을 제공 해야 합니다. 이는 탐색이 호출 되는지 여부를 제어 하기 위해 알림의 응답을 사용 하 여 뷰 모델 클래스에서 수행할 수 있습니다.

요약

Xamarin.Forms에는 내부 논리 기반 상태 변경으로 인해 일반적으로 사용자가 UI와 상호 작용 하거나 앱 자체에서 발생 하는 페이지 탐색에 대 한 지원이 포함 되어 있습니다. 그러나 MVVM 패턴을 사용 하는 앱에서 탐색을 구현 하는 것은 복잡할 수 있습니다.

이 장에서는 `NavigationService` 뷰 모델에서 뷰 모델을 처음 탐색 하는 데 사용 되는 클래스를 제공 했습니다. 뷰 모델 클래스에 탐색 논리를 배치 하면 자동화 된 테스트를 통해 논리를 수행할 수 있습니다. 또한 뷰 모델은 특정 비즈니스 규칙이 적용 되도록 탐색을 제어 하는 논리를 구현할 수 있습니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

엔터프라이즈 앱의 유효성 검사

2020-06-05 • 28 minutes to read • Edit Online

사용자의 입력을 허용 하는 모든 앱은 입력이 올바른지 확인 해야 합니다. 예를 들어 앱은 특정 범위의 문자만 포함 하는 입력을 확인 하거나 특정 길이 이거나 특정 형식과 일치 시킬 수 있습니다. 유효성 검사를 수행 하지 않으면 사용자가 응용 프로그램 실패를 유발 하는 데이터를 제공할 수 있습니다. 유효성 검사는 비즈니스 규칙을 적용 하 고 공격자가 악성 데이터를 삽입 하는 것을 방지 합니다.

MVVM (모델-뷰-ViewModel) 패턴의 컨텍스트에서는 사용자가 수정할 수 있도록 데이터 유효성 검사를 수행 하 고 뷰에 유효성 검사 오류를 알리기 위해 뷰 모델 또는 모델이 종종 필요 합니다. EShopOnContainers 모바일 앱은 뷰 모델 속성에 대 한 동기 클라이언트 쪽 유효성 검사를 수행 하 고, 잘못 된 데이터가 포함 된 컨트롤을 강조 표시 하 고, 데이터가 잘못 된 이유를 사용자에 게 알리는 오류 메시지를 표시 하 여 사용자에 게 유효성 검사 오류를 알립니다. 그림 6-1에서는 eShopOnContainers 모바일 앱에서 유효성 검사를 수행 하는 데 관련 된 클래스를 보여 줍니다.

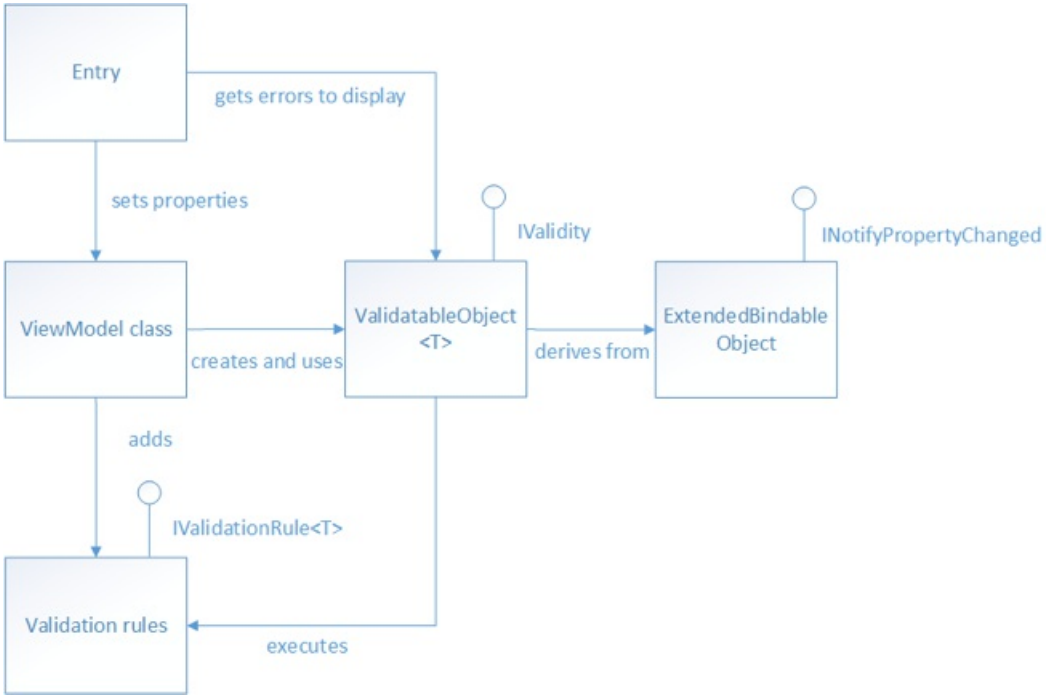

그림 6-1: eShopOnContainers 모바일 앱의 유효성 검사 클래스

유효성 검사가 필요한 뷰 모델 속성은 유형이 `ValidatableObject<T>` 며 각 `ValidatableObject<T>` 인스턴스의 속성에 유효성 검사 규칙이 추가 됩니다 `Validations` . 유효성 검사는 `Validate` `ValidatableObject<T>` 유효성 검사 규칙을 검색 하 고 속성에 대해 실행 하는 인스턴스의 메서드를 호출 하 여 뷰 모델에서 호출 됩니다 `ValidatableObject<T>` `Value` . 유효성 검사 오류는 인스턴스의 속성에 배치 되 `Errors` `ValidatableObject<T>` 고 `IsValid` 인스턴스의 속성은 `ValidatableObject<T>` 유효성 검사의 성공 또는 실패 여부를 표시 하도록 업데이트 됩니다.

속성 변경 알림은 클래스에서 제공 `ExtendedBindableObject` 하므로, `Entry` 컨트롤은 `IsValid` `ValidatableObject<T>` 입력 된 데이터가 유효한 지 여부에 대 한 알림을 받기 위해 뷰 모델 클래스의 인스턴스 속성에 바인딩할 수 있습니다.

유효성 검사 규칙 지정

유효성 검사 규칙은 `IValidationRule<T>` 다음 코드 예제와 같이 인터페이스에서 파생 되는 클래스를 만들어 지정 합니다.

```
public interface IValidationRule<T>
{
    string ValidationMessage { get; set; }
    bool Check(T value);
}
```

이 인터페이스는 유효성 검사 규칙 클래스가 필요한 유효성 검사를 수행 하는 데 사용 되는 메서드를 제공 해야 하도록 지정 하 `boolean` `Check` 고, `ValidationMessage` 유효성 검사에 실패 하면 표시 되는 유효성 검사 오류 메시지를 값으로 갖는 속성을 지정 합니다.

다음 코드 예제에서는 `IsNotNullOrEmptyRule<T>` `LoginView` eShopOnContainers 모바일 앱에서 모의 서비스를 사용할 때 사용자가 입력 한 사용자 이름 및 암호의 유효성 검사를 수행 하는 데 사용 되는 유효성 검사 규칙을 보여 줍니다.

```
public class IsNotNullOrEmptyRule<T> : IValidationRule<T>
{
    public string ValidationMessage { get; set; }

    public bool Check(T value)
    {
        if (value == null)
        {
            return false;
        }

        var str = value as string;
        return !string.IsNullOrWhiteSpace(str);
    }
}
```

`Check`메서드는 `boolean` 값 인수가 `null` 이거나 비어 있거나 공백 문자로만 구성 되어 있는지 여부를 나타내는을 반환 합니다.

EShopOnContainers 모바일 앱에서 사용 되지 않지만 다음 코드 예제에서는 전자 메일 주소의 유효성을 검사 하는 유효성 검사 규칙을 보여 줍니다.

```
public class EmailRule<T> : IValidationRule<T>
{
    public string ValidationMessage { get; set; }

    public bool Check(T value)
    {
        if (value == null)
        {
            return false;
        }

        var str = value as string;
        Regex regex = new Regex(@"^([\w\.\-]+)@([\w\-]+)((\.(\w){2,3})+)$");
        Match match = regex.Match(str);

        return match.Success;
    }
}
```

`Check`메서드는 `boolean` 값 인수가 유효한 전자 메일 주소 인지 여부를 나타내는을 반환 합니다. 이는 생성자에 지정 된 정규식 패턴의 첫 번째 항목에 대 한 값 인수를 검색 하 여 수행 됩니다 `Regex` . 입력 문자열에서 정규식 패턴을 찾을 수 있는지 여부는 개체의 속성 값을 확인 하 여 확인할 수 있습니다 `Match` `Success` .

> **NOTE**
> 속성 유효성 검사에는 종속 속성이 포함 될 수도 있습니다. 종속 속성의 예는 속성 A의 유효한 값 집합이 속성 B에 설정 된 특정 값에 따라 달라 지는 경우입니다. 속성 A의 값이 허용 되는 값 중 하나 인지 확인 하려면 속성 B의 값을 검색 해야 합니다. 또한 속성 B의 값이 변경 되 면 A 속성의 유효성을 다시 검사 해야 합니다.

속성에 유효성 검사 규칙 추가

EShopOnContainers 모바일 앱에서 유효성 검사가 필요한 뷰 모델 속성은 형식으로 선언 됩니다 `ValidatableObject<T>` `T` . 여기서은 유효성을 검사할 데이터의 형식입니다. 다음 코드 예제에서는 이러한 두 속성의 예를 보여 줍니다.

```
public ValidatableObject<string> UserName
{
    get
    {
        return _userName;
    }
    set
    {
        _userName = value;
        RaisePropertyChanged(() => UserName);
    }
}

public ValidatableObject<string> Password
{
    get
    {
        return _password;
    }
    set
    {
        _password = value;
        RaisePropertyChanged(() => Password);
    }
}
```

유효성 검사를 수행 하려면 `Validations` `ValidatableObject<T>` 다음 코드 예제에서 보여 주는 것 처럼 유효성 검사 규칙을 각 인스턴스의 컬렉션에 추가 해야 합니다.

```
private void AddValidations()
{
    _userName.Validations.Add(new IsNotNullOrEmptyRule<string>
    {
        ValidationMessage = "A username is required."
    });
    _password.Validations.Add(new IsNotNullOrEmptyRule<string>
    {
        ValidationMessage = "A password is required."
    });
}
```

이 메서드는 유효성 검사에 실패 한 경우 표시 되는 유효성 검사 오류 메시지를 지정 하는 유효성 검사 `IsNotNullOrEmptyRule<T>` `Validations` 규칙의 `ValidatableObject<T>` 속성에 대 한 값을 지정 하 여 각 인스턴스의 컬렉션에 유효성 검사 규칙을 추가 합니다 `ValidationMessage` .

유효성 검사 트리거

EShopOnContainers 모바일 앱에서 사용 되는 유효성 검사 방식은 속성의 유효성 검사를 수동으로 트리거하고 속성이 변경 될 때 자동으로 유효성 검사를 트리거할 수 있습니다.

수동으로 유효성 검사 트리거

뷰 모델 속성에 대해 유효성 검사를 수동으로 트리거할 수 있습니다. 예를 들어, 사용자가 모의 서비스를 사용할 때에서 **로그인** 단추를 누르면 eShopOnContainers 모바일 앱에서이 오류가 발생 합니다 `LoginView` . 명령 대리자는의 메서드를 호출 합니다 .이 메서드는 `MockSignInAsync` `LoginViewModel` `Validate` 다음 코드 예제와 같이 메서드를 실행 하 여 유효성 검사를 호출 합니다.

```
private bool Validate()
{
    bool isValidUser = ValidateUserName();
    bool isValidPassword = ValidatePassword();
    return isValidUser && isValidPassword;
}

private bool ValidateUserName()
{
    return _userName.Validate();
}

private bool ValidatePassword()
{
    return _password.Validate();
}
```

`Validate` 메서드는 `LoginView` 각 인스턴스에서 Validate 메서드를 호출 하 여 사용자가 입력 한 사용자 이름과 암호의 유효성을 검사 합니다 `ValidatableObject<T>` . 다음 코드 예제에서는 클래스의 Validate 메서드를 보여 줍니다 `ValidatableObject<T>` .

```
public bool Validate()
{
    Errors.Clear();

    IEnumerable<string> errors = _validations
        .Where(v => !v.Check(Value))
        .Select(v => v.ValidationMessage);

    Errors = errors.ToList();
    IsValid = !Errors.Any();

    return this.IsValid;
}
```

이 메서드는 컬렉션을 지운 `Errors` 다음 개체의 컬렉션에 추가 된 유효성 검사 규칙을 검색 `Validations` 합니다. `Check` 검색 된 각 유효성 검사 규칙에 대 한 메서드가 실행 되 고, `ValidationMessage` 데이터 유효성 검사에 실패 한 모든 유효성 검사 규칙의 속성 값이 `Errors` 인스턴스의 컬렉션에 추가 됩니다 `ValidatableObject<T>` . 마지막으로 `IsValid` 속성이 설정 되 고, 해당 값이 호출 메서드에 반환 되어 유효성 검사의 성공 여부를 나타냅니다.

속성이 변경 될 때 유효성 검사 트리거

바인딩된 속성이 변경 될 때마다 유효성 검사를 트리거할 수도 있습니다. 예를 들어의 양방향 바인딩이 `LoginView` 또는 속성을 설정 하는 경우 `UserName` `Password` 유효성 검사가 트리거됩니다. 다음 코드 예제에서는 이 작업을 수행 하는 방법을 보여 줍니다.

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    <Entry.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="TextChanged"
            Command="{Binding ValidateUserNameCommand}" />
    </Entry.Behaviors>
    ...
</Entry>
```

`Entry`컨트롤이 인스턴스의 속성에 바인딩되고 `UserName.Value` `ValidatableObject<T>` 컨트롤의 `Behaviors` 컬렉션에 `EventToCommandBehavior` 인스턴스가 추가 됩니다. 이 동작은에서 `ValidateUserNameCommand` 발생 하는 [] 이벤트에 대 한 응답으로를 실행 합니다 .이 이벤트는의 `TextChanged` `Entry` 텍스트가 변경 될 때 발생 합니다 `Entry` . 그러면 `ValidateUserNameCommand` 대리자는 `ValidateUserName` 인스턴스에서 메서드를 실행 하는 메서드를 실행 합니다 `Validate` `ValidatableObject<T>` . 따라서 사용자가 컨트롤의 사용자 이름에 대해 문자를 입력할 때마다 `Entry` 입력 한 데이터의 유효성 검사가 수행 됩니다.

동작에 대 한 자세한 내용은 동작 구현을 참조 하세요.

유효성 검사 오류 표시

EShopOnContainers 모바일 앱은 잘못 된 데이터를 포함 하는 컨트롤을 빨간색 줄로 강조 표시 하 고 잘못 된 데이터를 포함 하는 컨트롤 아래에서 데이터가 잘못 된 이유를 사용자에 게 알리는 오류 메시지를 표시 하 여 사용자에게 유효성 검사 오류를 알립니다. 잘못 된 데이터를 수정 하면 줄이 검정색으로 바뀌고 오류 메시지가 제거 됩니다. 그림 6-2에서는 유효성 검사 오류가 있을 때 eShopOnContainers 모바일 앱의 LoginView를 보여 줍니다.

그림 6-2: 로그인 중 유효성 검사 오류 표시

잘못 된 데이터를 포함 하는 컨트롤 강조 표시

`LineColorBehavior`연결 된 동작은 `Entry` 유효성 검사 오류가 발생 한 컨트롤을 강조 표시 하는 데 사용 됩니다. 다음 코드 예제에서는 `LineColorBehavior` 연결 된 동작을 컨트롤에 연결 하는 방법을 보여 줍니다 `Entry` .

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    <Entry.Style>
        <OnPlatform x:TypeArguments="Style">
            <On Platform="iOS, Android" Value="{StaticResource EntryStyle}" />
            <On Platform="UWP" Value="{StaticResource UwpEntryStyle}" />
        </OnPlatform>
    </Entry.Style>
    ...
</Entry>
```

`Entry`컨트롤은 다음 코드 예제와 같이 명시적 스타일을 사용 합니다.

```
<Style x:Key="EntryStyle"
       TargetType="{x:Type Entry}">
    ...
    <Setter Property="behaviors:LineColorBehavior.ApplyLineColor"
            Value="True" />
    <Setter Property="behaviors:LineColorBehavior.LineColor"
            Value="{StaticResource BlackColor}" />
    ...
</Style>
```

이 스타일은 `ApplyLineColor` `LineColor` 컨트롤에 연결 된 동작의 및 연결 된 속성을 설정 합니다 `LineColorBehavior` `Entry` . 스타일에 대한 자세한 내용은 스타일을 참조하세요.

연결 된 속성의 값 `ApplyLineColor` 이 설정 되거나 변경 되 면 `LineColorBehavior` 연결 된 동작은 메서드를 실행 합니다 `OnApplyLineColorChanged` .이 메서드는 다음 코드 예제에 나와 있습니다.

```
public static class LineColorBehavior
{
    ...
    private static void OnApplyLineColorChanged(
                BindableObject bindable, object oldValue, object newValue)
    {
        var view = bindable as View;
        if (view == null)
        {
            return;
        }

        bool hasLine = (bool)newValue;
        if (hasLine)
        {
            view.Effects.Add(new EntryLineColorEffect());
        }
        else
        {
            var entryLineColorEffectToRemove =
                    view.Effects.FirstOrDefault(e => e is EntryLineColorEffect);
            if (entryLineColorEffectToRemove != null)
            {
                view.Effects.Remove(entryLineColorEffectToRemove);
            }
        }
    }
}
```

이 메서드의 매개 변수는 동작이 연결 된 컨트롤의 인스턴스와 연결 된 속성의 이전 값과 새 값을 제공 합니다 `ApplyLineColor` . `EntryLineColorEffect` 클래스는 연결 된 속성이 인 경우 컨트롤의 컬렉션에 추가 되 고 `Effects` `ApplyLineColor` `true` , 그렇지 않으면 컨트롤의 컬렉션에서 제거 됩니다 `Effects` . 동작에 대 한 자세한 내용은 동작 구현을 참조 하세요.

`EntryLineColorEffect` `RoutingEffect` 클래스는 클래스를, 다음 코드 예제에서 볼 수 있습니다.

```
public class EntryLineColorEffect : RoutingEffect
{
    public EntryLineColorEffect() : base("eShopOnContainers.EntryLineColorEffect")
    {
    }
}
```

클래스는 플랫폼별 효과를 `RoutingEffect` 래핑하여 플랫폼별 효과를 나타냅니다. 이는 플랫폼별 효과에 대한 형

식 정보에 컴파일 시간 액세스가 없으므로 효과 제거 프로세스를 간소화합니다. 는 `EntryLineColorEffect` 기본 클래스 생성자를 호출 하 여 해상도 그룹 이름 연결로 구성 된 매개 변수를 전달 하 고 각 플랫폼별 효과 클래스에서 지정 된 고유 ID를 전달 합니다.

다음 코드 예제에서는 `eShopOnContainers.EntryLineColorEffect` iOS에 대 한 구현을 보여 줍니다.

```
[assembly: ResolutionGroupName("eShopOnContainers")]
[assembly: ExportEffect(typeof(EntryLineColorEffect), "EntryLineColorEffect")]
namespace eShopOnContainers.iOS.Effects
{
    public class EntryLineColorEffect : PlatformEffect
    {
        UITextField control;

        protected override void OnAttached()
        {
            try
            {
                control = Control as UITextField;
                UpdateLineColor();
            }
            catch (Exception ex)
            {
                Console.WriteLine("Can't set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached()
        {
            control = null;
        }

        protected override void OnElementPropertyChanged(PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (args.PropertyName == LineColorBehavior.LineColorProperty.PropertyName ||
                args.PropertyName == "Height")
            {
                Initialize();
                UpdateLineColor();
            }
        }

        private void Initialize()
        {
            var entry = Element as Entry;
            if (entry != null)
            {
                Control.Bounds = new CGRect(0, 0, entry.Width, entry.Height);
            }
        }

        private void UpdateLineColor()
        {
            BorderLineLayer lineLayer = control.Layer.Sublayers.OfType<BorderLineLayer>()
                                                              .FirstOrDefault();

            if (lineLayer == null)
            {
                lineLayer = new BorderLineLayer();
                lineLayer.MasksToBounds = true;
                lineLayer.BorderWidth = 1.0f;
                control.Layer.AddSublayer(lineLayer);
                control.BorderStyle = UITextBorderStyle.None;
            }
```

```
            lineLayer.Frame = new CGRect(0f, Control.Frame.Height-1f, Control.Bounds.Width, 1f);
            lineLayer.BorderColor = LineColorBehavior.GetLineColor(Element).ToCGColor();
            control.TintColor = control.TextColor;
        }

        private class BorderLineLayer : CALayer
        {
        }
    }
}
```

`OnAttached` 메서드는 컨트롤의 네이티브 컨트롤을 검색 Xamarin.Forms `Entry` 하 고 메서드를 호출 하 여 선 색을 업데이트 합니다 `UpdateLineColor` . `OnElementPropertyChanged` 재정의는 `Entry` 연결 된 속성이 변경 될 경우 선 색을 업데이트 하 여 컨트롤의 바인딩 가능한 속성 변경 내용에 응답 하 고 `LineColor` , `Height` 변경의 속성을 변경 `Entry` 합니다. 효과에 대한 자세한 내용은 효과를 참조하세요.

컨트롤에 유효한 데이터가 입력 되 면 `Entry` 컨트롤 아래쪽에 검정 줄이 적용 되어 유효성 검사 오류가 없음을 표시 합니다. 그림 6-3에서는이에 대 한 예를 보여 줍니다.

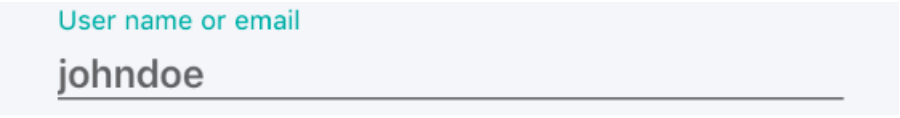

그림 6-3: 유효성 검사 오류를 나타내는 검정 선

`Entry` 컨트롤에는 `DataTrigger` 해당 컬렉션에 추가 된도 있습니다 `Triggers` . 다음 코드 예제에서는을 보여 줍니다 `DataTrigger` .

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    ...
    <Entry.Triggers>
        <DataTrigger
            TargetType="Entry"
            Binding="{Binding UserName.IsValid}"
            Value="False">
            <Setter Property="behaviors:LineColorBehavior.LineColor"
                    Value="{StaticResource ErrorColor}" />
        </DataTrigger>
    </Entry.Triggers>
</Entry>
```

그러면 `DataTrigger` 속성이 모니터링 되 `UserName.IsValid` 고, 값이 이면가 실행 되어 연결 된 `false` `Setter` `LineColor` 동작의 연결 된 속성이 빨강으로 변경 됩니다 `LineColorBehavior` . 그림 6-4에서는이에 대 한 예를 보여 줍니다.

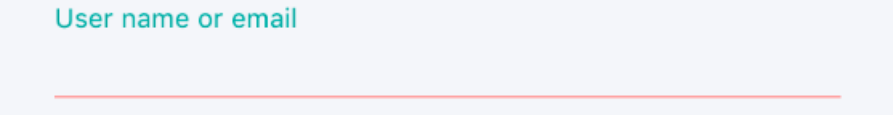

그림 6-4: 유효성 검사 오류를 나타내는 빨간색 선

입력 한 데이터가 유효 `Entry` 하지 않으면 컨트롤의 줄은 빨간색으로 유지 되 고, 그렇지 않은 경우에는 입력 된 데이터가 유효한 것으로 표시 되도록 검은색으로 변경 됩니다.

트리거에 대 한 자세한 내용은 트리거를 참조 하세요.

오류 메시지 표시

UI는 데이터가 유효성 검사에 실패 한 각 컨트롤 아래의 레이블 컨트롤에서 유효성 검사 오류 메시지를 표시 합니다. 다음 코드 예제에서는 사용자가 `Label` 올바른 사용자 이름을 입력 하지 않은 경우 유효성 검사 오류 메시지를 표시 하는를 보여 줍니다.

```
<Label Text="{Binding UserName.Errors, Converter={StaticResource FirstValidationErrorConverter}}"
       Style="{StaticResource ValidationErrorLabelStyle}" />
```

각는 `Label` `Errors` 유효성 검사 중인 뷰 모델 개체의 속성에 바인딩됩니다. `Errors`속성은 클래스에서 제공 되며 `ValidatableObject<T>` 형식입니다 `List<string>` . 속성은 `Errors` 여러 유효성 검사 오류를 포함할 수 있으므로 `FirstValidationErrorConverter` 인스턴스는 표시를 위해 컬렉션에서 첫 번째 오류를 검색 하는 데 사용 됩니다.

요약

EShopOnContainers 모바일 앱은 뷰 모델 속성에 대 한 동기식 클라이언트 쪽 유효성 검사를 수행 하 고, 잘못 된 데이터가 포함 된 컨트롤을 강조 표시 하 고, 데이터가 잘못 된 이유를 사용자에 게 알리는 오류 메시지를 표시 하 여 사용자에 게 유효성 검사 오류를 알립니다.

유효성 검사가 필요한 뷰 모델 속성은 유형이 `ValidatableObject<T>` 며 각 `ValidatableObject<T>` 인스턴스의 속성에 유효성 검사 규칙이 추가 됩니다 `Validations` . 유효성 검사는 `Validate` `ValidatableObject<T>` 유효성 검사 규칙을 검색 하 고 속성에 대해 실행 하는 인스턴스의 메서드를 호출 하 여 뷰 모델에서 호출 됩니다 `ValidatableObject<T>` `Value` . 유효성 검사 오류는 인스턴스의 속성에 배치 되 `Errors` `ValidatableObject<T>` 고 `IsValid` 인스턴스의 속성은 `ValidatableObject<T>` 유효성 검사의 성공 또는 실패 여부를 표시 하도록 업데이트 됩니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

구성 관리

2020-06-05 • 17 minutes to read • Edit Online

설정을 사용 하면 응용 프로그램의 동작을 구성 하는 데이터를 분리 하 여 앱을 다시 빌드하지 않고도 동작을 변경할 수 있습니다. 설정에는 앱 설정 및 사용자 설정의 두 가지 유형이 있습니다.

앱 설정은 앱에서 만들고 관리 하는 데이터입니다. 고정 웹 서비스 끝점, API 키 및 런타임 상태와 같은 데이터가 포함 될 수 있습니다. 앱 설정은 앱의 존재 여부와 연결 되며 해당 앱에만 의미가 있습니다.

사용자 설정은 앱의 동작에 영향을 주는 앱의 사용자 지정 가능한 설정 이며 자주 다시 조정할 필요가 없습니다. 예를 들어 앱을 사용 하면 사용자가 데이터를 검색할 위치와 화면에 표시 하는 방법을 지정할 수 있습니다.

Xamarin.Forms에는 설정 데이터를 저장 하는 데 사용할 수 있는 영구 사전이 포함 되어 있습니다. 이 사전은 속성을 사용 하 여 액세스할 수 `Application.Current.Properties` 있으며, 응용 프로그램에 배치 된 모든 데이터는 앱이 절전 모드로 전환 될 때 저장 되 고 앱이 다시 시작 되거나 다시 시작 될 때 복원 됩니다. 또한 클래스에는 `Application` `SavePropertiesAsync` 필요할 때 앱에서 해당 설정을 저장할 수 있도록 하는 메서드도 있습니다. 이 사전에 대 한 자세한 내용은 속성 사전을 참조 하세요.

영구적 사전을 사용 하 여 데이터를 저장 하는 경우 Xamarin.Forms 에는 데이터를 쉽게 바인딩할 수 없다는 단점이 있습니다. 따라서 eShopOnContainers 모바일 앱은 NuGet에서 사용할 수 있는 xam. s a s. 설정 라이브러리를 사용 합니다. 이 라이브러리는 각 플랫폼에서 제공 하는 네이티브 설정 관리를 사용 하는 동시에 앱 및 사용자 설정을 유지 하 고 검색 하기 위한 일관 되 고 형식이 안전한 플랫폼 간 방법을 제공 합니다. 또한 데이터 바인딩을 사용 하 여 라이브러리에서 제공 하는 설정 데이터에 액세스 하는 것도 간단 합니다.

> **NOTE**
> Xam. Plugin. 설정 라이브러리는 앱과 사용자 설정을 둘 다 저장할 수 있지만 둘 사이를 구분 하지 않습니다.

설정 클래스 만들기

Xam. s s d. 설정 라이브러리를 사용 하는 경우 앱에 필요한 앱 및 사용자 설정을 포함 하는 단일 정적 클래스를 만들어야 합니다. 다음 코드 예제에서는 eShopOnContainers 모바일 앱의 Settings 클래스를 보여 줍니다.

```
public static class Settings
{
    private static ISettings AppSettings
    {
        get
        {
            return CrossSettings.Current;
        }
    }
    ...
}
```

설정은 API를 통해 읽고 쓸 수 있으며 `ISettings` ,이는 Xam. 플러그 인 설정 라이브러리에서 제공 됩니다. 이 라이브러리는 API에 액세스 하는 데 사용할 수 있는 단일 항목을 제공 `CrossSettings.Current` 하며, 응용 프로그램의 설정 클래스는 속성을 통해이 singleton을 노출 해야 합니다 `ISettings` .

> **NOTE**
> 플러그 인에 대 한 지시문을 사용 하 고 있습니다. 설정 및 플러그 인에 대 한 액세스를 필요로 하는 클래스에는 설정 및 플러그 인을 추가 해야 합니다.

설정 추가

각 설정은 키, 기본 값 및 속성으로 구성 됩니다. 다음 코드 예제에서는 eShopOnContainers 모바일 앱이 연결 하는 온라인 서비스에 대 한 기준 URL을 나타내는 사용자 설정에 대 한 세 가지 항목을 모두 보여 줍니다.

```
public static class Settings
{
    ...
    private const string IdUrlBase = "url_base";
    private static readonly string UrlBaseDefault = GlobalSetting.Instance.BaseEndpoint;
    ...

    public static string UrlBase
    {
        get
        {
            return AppSettings.GetValueOrDefault<string>(IdUrlBase, UrlBaseDefault);
        }
        set
        {
            AppSettings.AddOrUpdateValue<string>(IdUrlBase, value);
        }
    }
}
```

키는 항상 키 이름을 정의 하는 const 문자열이 며,이 값은 설정의 기본값은 필수 형식의 정적 읽기 전용 값입니다. 기본값을 제공 하면 설정 되지 않은 설정이 검색 되는 경우 유효한 값을 사용할 수 있습니다.

`UrlBase` 정적 속성은 API의 두 가지 메서드를 사용 하 여 `ISettings` 설정 값을 읽거나 씁니다. `ISettings.GetValueOrDefault` 메서드는 플랫폼별 저장소에서 설정 값을 검색 하는 데 사용 됩니다. 설정에 대해 정의 된 값이 없는 경우에는 기본값이 대신 검색 됩니다. 마찬가지로, 메서드를 사용 하 여 `ISettings.AddOrUpdateValue` 설정의 값을 플랫폼별 저장소에 유지 합니다.

클래스 내에서 기본값을 정의 하는 대신 `Settings` 문자열은 `UrlBaseDefault` 클래스에서 해당 값을 가져옵니다 `GlobalSetting` . 다음 코드 예제에서는 `BaseEndpoint` `UpdateEndpoint` 이 클래스의 속성과 메서드를 보여 줍니다.

```
public class GlobalSetting
{
    ...
    public string BaseEndpoint
    {
        get { return _baseEndpoint; }
        set
        {
            _baseEndpoint = value;
            UpdateEndpoint(_baseEndpoint);
        }
    }
    ...

    private void UpdateEndpoint(string baseEndpoint)
    {
        RegisterWebsite = string.Format("{0}:5105/Account/Register", baseEndpoint);
        CatalogEndpoint = string.Format("{0}:5101", baseEndpoint);
        OrdersEndpoint = string.Format("{0}:5102", baseEndpoint);
        BasketEndpoint = string.Format("{0}:5103", baseEndpoint);
        IdentityEndpoint = string.Format("{0}:5105/connect/authorize", baseEndpoint);
        UserInfoEndpoint = string.Format("{0}:5105/connect/userinfo", baseEndpoint);
        TokenEndpoint = string.Format("{0}:5105/connect/token", baseEndpoint);
        LogoutEndpoint = string.Format("{0}:5105/connect/endsession", baseEndpoint);
        IdentityCallback = string.Format("{0}:5105/xamarincallback", baseEndpoint);
        LogoutCallback = string.Format("{0}:5105/Account/Redirecting", baseEndpoint);
    }
}
```

`BaseEndpoint`속성이 설정 될 때마다 `UpdateEndpoint` 메서드가 호출 됩니다. 이 메서드는 일련의 속성을 업데이트 합니다 .이 속성은 모두 `UrlBase` `Settings` eShopOnContainers 모바일 앱이 연결 하는 다른 끝점을 나타내는 클래스에서 제공 하는 사용자 설정을 기반으로 합니다.

사용자 설정에 대 한 데이터 바인딩

EShopOnContainers 모바일 앱에서은 `SettingsView` 두 가지 사용자 설정을 노출 합니다. 이러한 설정을 사용 하면 앱이 Docker 컨테이너로 배포 된 마이크로 서비스에서 데이터를 검색 해야 하는지 여부 또는 앱이 인터넷에 연결 되지 않아도 되는 모의 서비스에서 데이터를 검색 해야 하는지 여부를 구성할 수 있습니다. 컨테이너 화 된 마이크로 서비스에서 데이터를 검색 하도록 선택 하는 경우 마이크로 서비스에 대 한 기본 끝점 URL을 지정 해야 합니다. 그림 7-1에서는 `SettingsView` 사용자가 컨테이너 화 된 마이크로 서비스에서 데이터를 검색 하도록 선택한 경우를 보여 줍니다.

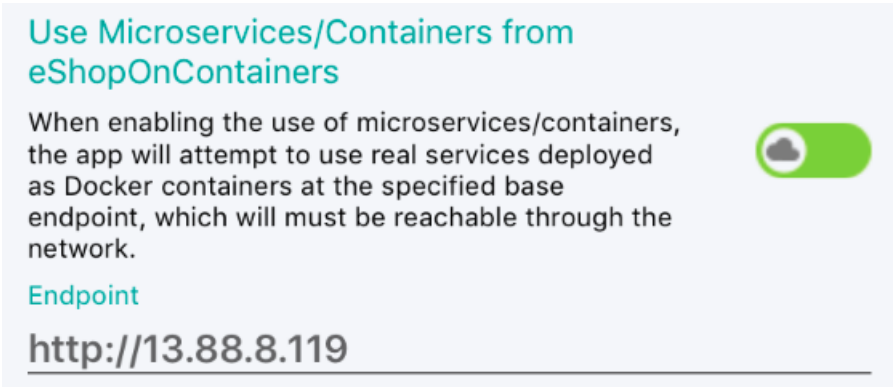

그림 7-1: eShopOnContainers 모바일 앱에서 노출 하는 사용자 설정

데이터 바인딩을 사용 하 여 클래스에 의해 노출 되는 설정을 검색 하 고 설정할 수 있습니다 `Settings` . 이는 클래스의 속성에 액세스 하 `Settings` 고 설정 값이 변경 된 경우 속성 변경 알림을 발생 시키는 모델 속성을 보기 위해 뷰 바인딩의 컨트롤에 의해 이루어집니다. EShopOnContainers 모바일 앱에서 모델을 생성 하 고 보기에 연결 하는 방법에 대 한 자세한 내용은 뷰 모델 로케이터를 사용 하 여 자동으로 뷰 모델 만들기를 참조 하세요.

다음 코드 예제에서는 `Entry` `SettingsView` 사용자가 컨테이너 화 된 마이크로 서비스에 대 한 기본 끝점 URL을 입력할 수 있도록 하는의 컨트롤을 보여 줍니다.

```
<Entry Text="{Binding Endpoint, Mode=TwoWay}" />
```

이 `Entry` 컨트롤은 `Endpoint` `SettingsViewModel` 양방향 바인딩을 사용 하 여 클래스의 속성에 바인딩됩니다. 다음 코드 예제에서는 끝점 속성을 보여 줍니다.

```
public string Endpoint
{
    get { return _endpoint; }
    set
    {
        _endpoint = value;

        if(!string.IsNullOrEmpty(_endpoint))
        {
            UpdateEndpoint(_endpoint);
        }

        RaisePropertyChanged(() => Endpoint);
    }
}
```

`Endpoint` 속성이 설정 되어 있으면 제공 된 `UpdateEndpoint` 값이 유효 하 고 속성 변경 알림이 발생 한 경우 메서드가 호출 됩니다. 다음 코드 예제는 `UpdateEndpoint` 메서드를 보여줍니다.

```
private void UpdateEndpoint(string endpoint)
{
    Settings.UrlBase = endpoint;
}
```

이 메서드는 `UrlBase` `Settings` 사용자가 입력 한 기본 끝점 URL 값을 사용 하 여 클래스의 속성을 업데이트 합니다. 그러면 해당 값이 플랫폼별 저장소에 유지 됩니다.

`SettingsView`을 탐색 하는 경우 `InitializeAsync` 클래스의 메서드가 `SettingsViewModel` 실행 됩니다. 다음 코드 예제에서는 이 메서드를 보여줍니다.

```
public override Task InitializeAsync(object navigationData)
{
    ...
    Endpoint = Settings.UrlBase;
    ...
}
```

메서드는 속성을 `Endpoint` 클래스의 속성 값으로 설정 합니다 `UrlBase` `Settings` . 속성에 액세스 하는 경우 Xam. s a s. `UrlBase` 설정 라이브러리에서 플랫폼별 저장소의 설정 값을 검색 합니다. 메서드를 호출 하는 방법에 대 한 자세한 내용은 `InitializeAsync` 탐색 하는 동안 매개 변수 전달을 참조 하세요.

이 메커니즘을 사용 하면 사용자가 SettingsView로 이동할 때마다 사용자 설정이 플랫폼별 저장소에서 검색 되고 데이터 바인딩을 통해 표시 됩니다. 그런 다음 사용자가 설정 값을 변경 하는 경우 데이터 바인딩은 플랫폼별 저장소에 즉시 다시 저장 되도록 합니다.

요약

설정을 사용 하면 응용 프로그램의 동작을 구성 하는 데이터를 분리 하 여 앱을 다시 빌드하지 않고도 동작을 변

경할 수 있습니다. 앱 설정은 앱에서 만들고 관리 하는 데이터 이며, 사용자 설정은 앱의 동작에 영향을 주는 앱의 사용자 지정 가능한 설정 이며 자주 다시 조정할 필요가 없습니다.

Xam. s a s. 설정 라이브러리는 앱 및 사용자 설정을 유지 하 고 검색 하기 위한 일관 되 고 형식이 안전한 크로스 플랫폼 방식을 제공 하며, 데이터 바인딩을 사용 하 여 라이브러리를 사용 하 여 만든 설정에 액세스할 수 있습니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

컨테이너화된 마이크로 서비스

2020-06-05 • 37 minutes to read • Edit Online

클라이언트-서버 응용 프로그램을 개발 하면 각 계층에서 특정 기술을 사용 하는 계층화 된 응용 프로그램을 구축 하는 데 집중 했습니다. 이러한 응용 프로그램을 종종 *모놀리식* 응용 프로그램 이라고 하며, 최대 부하를 위해 미리 확장 된 하드웨어로 패키지 됩니다. 이 개발 방법의 주요 단점은 각 계층 내의 구성 요소 간 긴밀 한 결합, 개별 구성 요소를 쉽게 확장할 수 없고 테스트 비용입니다. 간단한 업데이트는 계층의 나머지 부분에 예기치 않은 영향을 미칠 수 있으므로 응용 프로그램 구성 요소를 변경 하려면 전체 계층을 retested 하 고 다시 배포 해야 합니다.

특히 클라우드의 연령와 관련 하 여 개별 구성 요소를 쉽게 확장할 수 없습니다. 모놀리식 응용 프로그램은 도메인 별 기능을 포함 하며 일반적으로 프런트 엔드, 비즈니스 논리 및 데이터 저장소와 같은 기능 계층으로 구분 됩니다. 모놀리식 응용 프로그램은 그림 8-1에 나와 있는 것 처럼 전체 응용 프로그램을 여러 컴퓨터에 복제 하 여 크기를 조정 합니다.

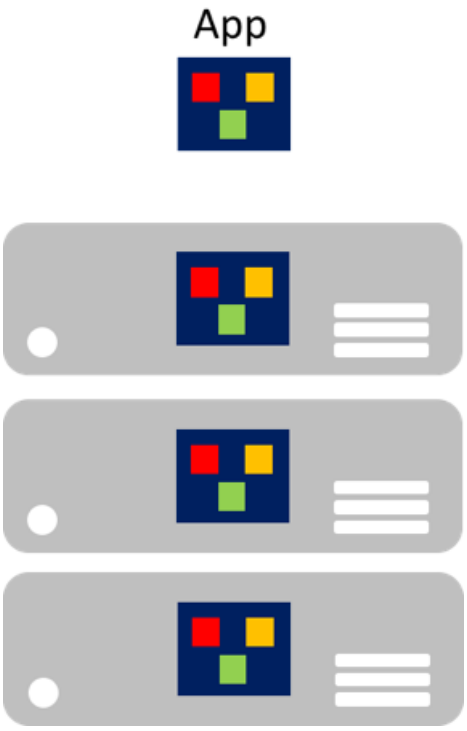

그림 8-1: 모놀리식 응용 프로그램 크기 조정 방법

마이크로 서비스

마이크로 서비스는 최신 클라우드 응용 프로그램의 민첩성, 확장성 및 안정성 요구 사항에 적합 한 방법인 응용 프로그램 개발 및 배포에 대 한 다양 한 접근 방식을 제공 합니다. 마이크로 서비스 응용 프로그램은 응용 프로그램의 전체 기능을 제공 하기 위해 함께 작동 하는 독립적인 구성 요소로 분해 됩니다. 마이크로 서비스 라는 용어는 각 마이크로 서비스 단일 함수를 구현 하도록 응용 프로그램을 독립적인 문제를 반영 하기에 충분 한 작은 서비스로 구성 해야 함을 강조 합니다. 또한 각 마이크로 서비스는 잘 정의 된 계약을 사용 하 여 다른 마이크로 서비스에서 데이터와 통신 하 고 데이터를 공유할 수 있도록 합니다. 마이크로 서비스의 일반적인 예로는 쇼핑 카트, 재고 처리, 구매 하위 시스템, 지불 처리 등이 있습니다.

마이크로 서비스는 함께 확장 되는 자이언트 모놀리식 응용 프로그램에 비해 독립적으로 확장 될 수 있습니다. 즉, 요청을 지 원하는 데 더 많은 처리 능력 또는 네트워크 대역폭이 필요한 특정 기능 영역은 응용 프로그램의 다른 영역을 불필요 하 게 확장 하는 대신 크기를 조정할 수 있습니다. 그림 8-2에서는 마이크로 서비스를 독립적으로 배포 및 확장 하 여 여러 컴퓨터에서 서비스 인스턴스를 만들어이 접근 방식을 보여 줍니다.

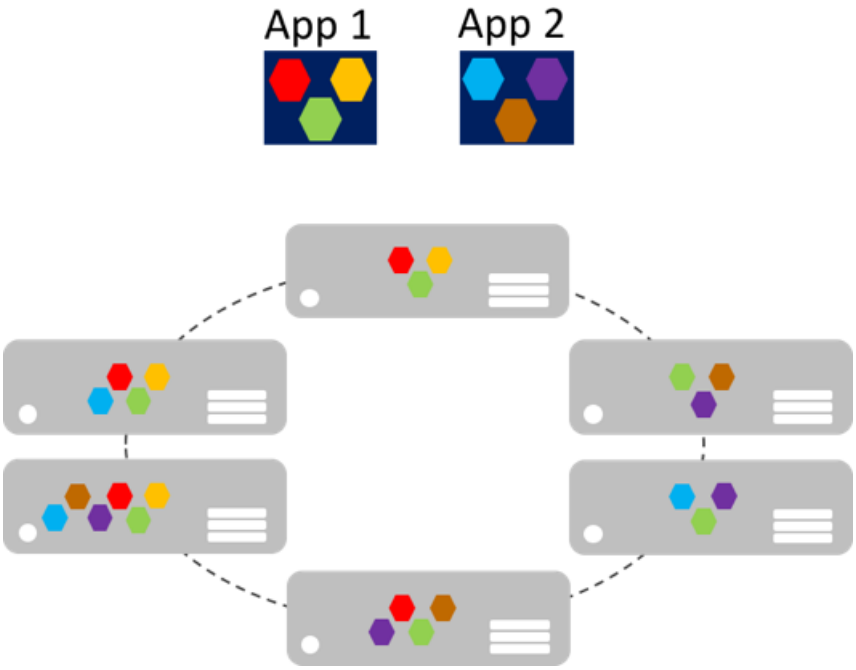

그림 8-2: 마이크로 서비스 응용 프로그램 크기 조정 방법

마이크로 서비스 확장은 거의 즉시 수행할 수 있으므로 응용 프로그램을 변경 하는 로드에 대응할 수 있습니다. 예를 들어 응용 프로그램의 웹 연결 기능에서 단일 마이크로 서비스는 추가 들어오는 트래픽을 처리 하기 위해 규모를 확장 해야 하는 응용 프로그램의 유일한 마이크로 서비스 수 있습니다.

응용 프로그램 확장성을 위한 클래식 모델은 영구 데이터를 저장 하기 위해 공유 외부 데이터 저장소를 사용 하 여 부하 분산 된 상태 비저장 계층을 보유 하는 것입니다. 상태 저장 마이크로 서비스는 자체의 영구 데이터를 관리 하며, 일반적으로 이러한 데이터를 저장 된 서버에 로컬로 저장 하 여 네트워크 액세스와 서비스 간 작업의 복잡성을 방지 합니다. 이렇게 하면 데이터를 가장 빨리 처리할 수 있으며 시스템을 캐시할 필요가 없습니다. 또한 확장 가능한 상태 저장 마이크로 서비스는 일반적으로 단일 서버에서 지원할 수 있는 데이터 크기 및 전송 처리량을 관리 하기 위해 인스턴스 간에 데이터를 분할 합니다.

마이크로 서비스는 독립 업데이트도 지원 합니다. 마이크로 서비스 간의 이러한 느슨한 결합은 빠르고 안정적인 응용 프로그램 진화를 제공 합니다. 독립적인 분산 특성은 특정 시간에 단일 마이크로 서비스 인스턴스의 하위 집합만 업데이트 하는 롤링 업데이트를 지원 합니다. 따라서 문제가 감지 되 면 모든 인스턴스를 잘못 된 코드나 구성으로 업데이트 하기 전에 버그가 있는 업데이트를 롤백할 수 있습니다. 마찬가지로, 마이크로 서비스는 일반적으로 스키마 버전 관리를 사용 하므로, 어떤 마이크로 서비스 인스턴스가 통신 하 고 있는지에 관계 없이 업데이트가 적용 되는 경우 클라이언트가 일관 된 버전을 볼 수 있습니다.

따라서 마이크로 서비스 응용 프로그램은 모놀리식 응용 프로그램에 비해 많은 이점을 제공 합니다.

- 각 마이크로 서비스는 상대적으로 작고 관리 하기 쉬우며 진화 하 고 있습니다.
- 각 마이크로 서비스는 다른 서비스와 독립적으로 개발 및 배포할 수 있습니다.
- 각 마이크로 서비스은 독립적으로 확장 될 수 있습니다. 예를 들어 카탈로그 서비스 또는 시장 바구니 서비스는 주문 서비스 보다 확장 해야 할 수 있습니다. 따라서 결과 인프라는 규모 확장 시 더 효율적으로 리소스를 사용 합니다.
- 각 마이크로 서비스는 문제를 격리 합니다. 예를 들어 서비스에 문제가 있는 경우 해당 서비스에만 영향을 줍니다. 다른 서비스는 계속 해 서 요청을 처리할 수 있습니다.
- 각 마이크로 서비스는 최신 기술을 사용할 수 있습니다. 마이크로 서비스는 자치이 고 side-by-side로 실행 되기 때문에 모놀리식 응용 프로그램에서 사용할 수 있는 이전 프레임 워크를 사용 하는 대신 최신 기술 및 프레임 워크를 사용할 수 있습니다.

그러나 마이크로 서비스 기반 솔루션은 잠재적인 단점이 있습니다.

- 응용 프로그램을 마이크로 서비스로 분할 하는 방법을 선택 하는 것은 각 마이크로 서비스 데이터 원본에 대 한 책임을 포함 하 여 완벽 하 게 완전 하 게 자율적으로 수행 해야 하기 때문에 어려울 수 있습니다.

- 개발자는 응용 프로그램에 복잡성 및 대기 시간을 추가 하는 서비스 간 통신을 구현 해야 합니다.
- 여러 마이크로 서비스 간의 원자성 트랜잭션은 일반적으로 불가능 합니다. 따라서 비즈니스 요구 사항은 마이크로 서비스 간의 최종 일관성을 수용 해야 합니다.
- 프로덕션 환경에서는 많은 독립 서비스의 시스템 손상을 배포 하 고 관리 하는 작업 복잡성이 있습니다.
- 클라이언트-마이크로 서비스 간 직접 통신을 통해 마이크로 서비스의 계약을 리팩터링 하기가 어려울 수 있습니다. 예를 들어 시스템을 서비스로 분할 하는 방법을 시간이 지남에 따라 변경 해야 할 수도 있습니다. 단일 서비스를 두 개 이상의 서비스로 분할 하 고 두 서비스를 병합할 수 있습니다. 클라이언트가 마이크로 서비스와 직접 통신 하는 경우이 리팩터링 작업은 클라이언트 앱과의 호환성을 손상 시킬 수 있습니다.

컨테이너화

컨테이너 화는 응용 프로그램 및 해당 버전의 종속성 집합과 배포 매니페스트 파일로 추상화 되는 환경 구성을 포함 하는 소프트웨어 개발에 대 한 방법으로, 컨테이너 이미지로 함께 패키지 되 고, 하나의 단위로 테스트 되 고, 호스트 운영 체제에 배포 됩니다.

컨테이너는 다른 컨테이너 또는 호스트의 리소스를 건드리지 않고도 응용 프로그램을 실행할 수 있는 격리 된 리소스 제어 및 휴대용 운영 환경입니다. 따라서 컨테이너는 새로 설치 된 물리적 컴퓨터 또는 가상 컴퓨터 처럼 보이고 작동 합니다.

그림 8-3에 나와 있는 것 처럼 컨테이너와 가상 컴퓨터 간에는 많은 유사점이 있습니다.

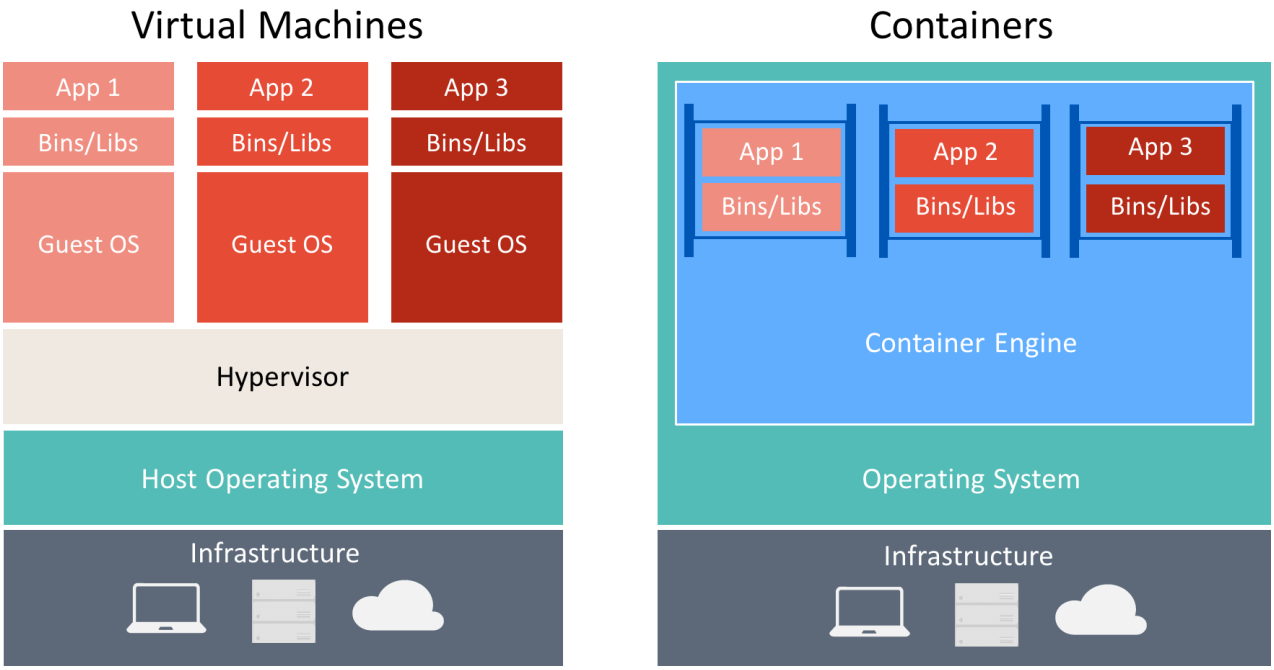

그림 8-3: 가상 컴퓨터 및 컨테이너 비교

컨테이너는 운영 체제를 실행 하 고, 파일 시스템을 포함 하며, 네트워크를 통해 실제 또는 가상 머신과 같이 액세스할 수 있습니다. 그러나 컨테이너에서 사용 하는 기술 및 개념은 가상 머신과 매우 다릅니다. 가상 컴퓨터에는 응용 프로그램, 필요한 종속성 및 전체 게스트 운영 체제가 포함 됩니다. 컨테이너에는 응용 프로그램 및 해당 종속성이 포함 되지만, 운영 체제를 호스트 운영 체제에서 격리 된 프로세스로 실행 되는 다른 컨테이너와 공유 합니다 .이 컨테이너는 컨테이너 당 특수 한 가상 컴퓨터 내에서 실행 되는 Hyper-v 컨테이너와는 별도로 실행 됩니다. 따라서 컨테이너는 리소스를 공유 하 고 일반적으로 가상 컴퓨터 보다 리소스가 덜 필요 합니다.

컨테이너 지향 개발 및 배포 방법의 장점은 일관 되지 않은 환경에서 발생 하는 문제를 대부분 제거 하 고 발생 하는 문제를 제거 하는 것입니다. 또한 컨테이너는 필요에 따라 새 컨테이너를 등록 하 여 빠른 응용 프로그램 확장 기능을 허용 합니다.

컨테이너를 만들고 작업 하는 경우 주요 개념은 다음과 같습니다.

- 컨테이너 호스트: 컨테이너를 호스트 하도록 구성 된 물리적 또는 가상 컴퓨터입니다. 컨테이너 호스트는 하나

이상의 컨테이너를 실행 합니다.

- 컨테이너 이미지: 이미지는 서로 위에 누적 된 계층화 된 파일 시스템의 합집합으로 구성 되며 컨테이너의 기반이 됩니다. 이미지는 상태가 없으며 다른 환경에 배포 될 때 변경 되지 않습니다.
- 컨테이너: 컨테이너는 이미지의 런타임 인스턴스입니다.
- 컨테이너 OS 이미지: 컨테이너가 이미지에서 배포됩니다. 컨테이너 운영 체제 이미지는 컨테이너를 구성 하는 잠재적으로 많은 이미지 계층에서 첫 번째 계층입니다. 컨테이너 운영 체제는 변경할 수 없으며 수정할 수 없습니다.
- 컨테이너 리포지토리: 컨테이너 이미지가 만들어질 때마다 이미지와 해당 종속성이 로컬 리포지토리에 저장됩니다. 이러한 이미지는 컨테이너 호스트에서 여러 번 다시 사용할 수 있습니다. 컨테이너 이미지는 다른 컨테이너 호스트에서 사용할 수 있도록 Docker 허브와 같은 공용 또는 개인 레지스트리에 저장할 수도 있습니다.

마이크로 서비스 기반 응용 프로그램을 구현할 때 기업이 점점 더 많은 컨테이너를 채택 하 고 있으며 Docker가 대부분의 소프트웨어 플랫폼과 클라우드 공급 업체에서 채택 하는 표준 컨테이너 구현이 되었습니다.

EShopOnContainers reference 응용 프로그램은 그림 8-4에 나와 있는 것 처럼 Docker를 사용 하 여 4 개의 컨테이너 화 된 백 엔드 마이크로 서비스를 호스팅합니다.

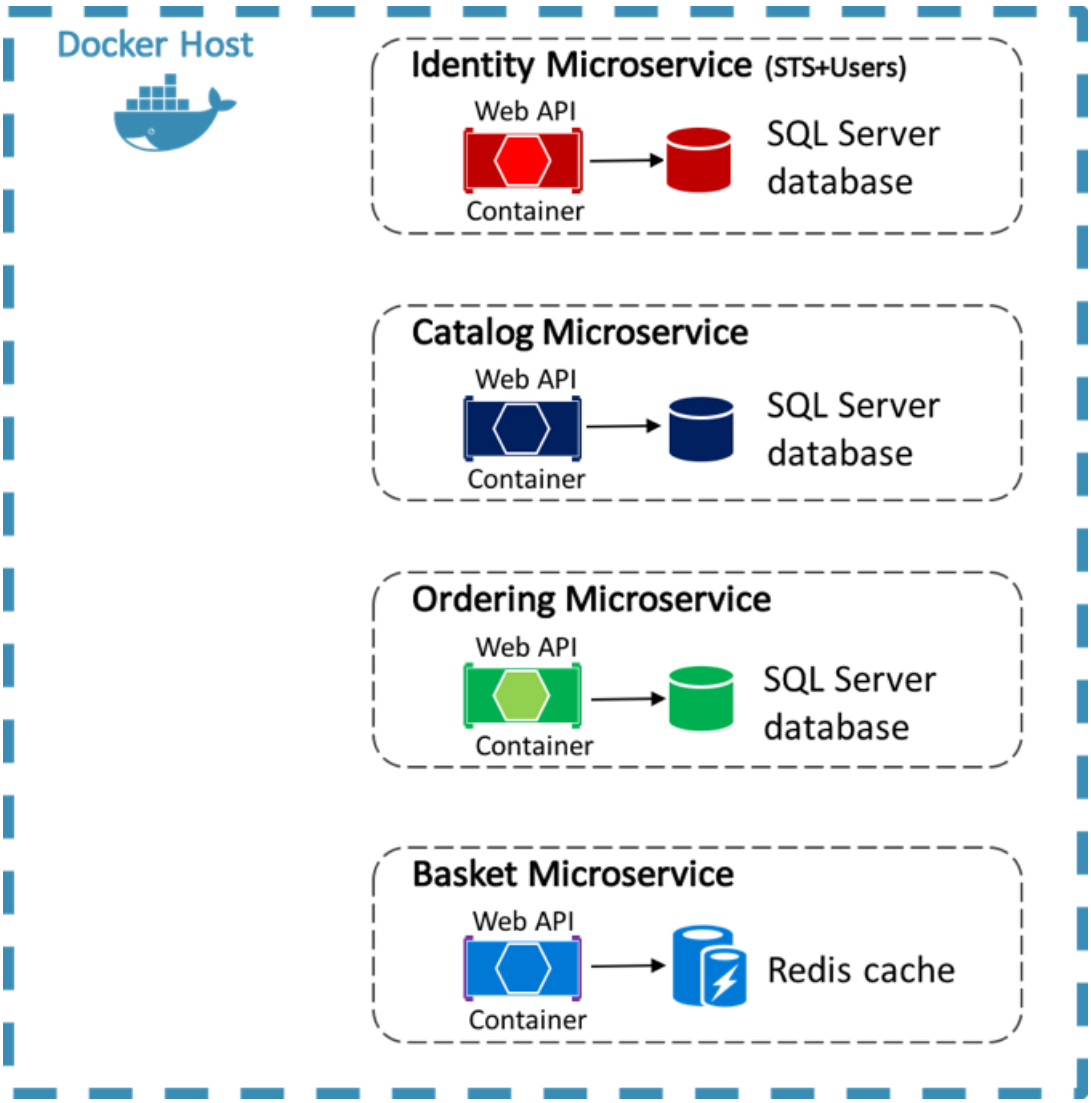

그림 8-4: eShopOnContainers reference 응용 프로그램 백 엔드 마이크로 서비스

참조 응용 프로그램의 백 엔드 서비스 아키텍처는 공동 마이크로 서비스 및 컨테이너 형식의 여러 자치 하위 시스템으로 분해 됩니다. 각 마이크로 서비스는 id 서비스, 카탈로그 서비스, 주문 서비스 및 바구니 서비스의 단일 기능 영역을 제공 합니다.

각 마이크로 서비스에는 자체 데이터베이스가 있으므로 다른 마이크로 서비스에서 완전히 분리 될 수 있습니다. 필요한 경우에는 응용 프로그램 수준 이벤트를 사용 하 여 서로 다른 마이크로 서비스의 데이터베이스 간 일관성을 달성 합니다. 자세한 내용은 마이크로 서비스 간 통신을 참조 하세요.

참조 응용 프로그램에 대 한 자세한 내용은 .Net 마이크로 서비스: 컨테이너 화 된 .Net 응용 프로그램 아키텍처를 참조 하세요.

클라이언트와 마이크로 서비스 간의 통신

EShopOnContainers 모바일 앱은 그림 8-5에 표시 된 *직접 클라이언트-마이크로 서비스* 통신을 사용 하 여 컨테이너 화 된 백 엔드 마이크로 서비스와 통신 합니다.

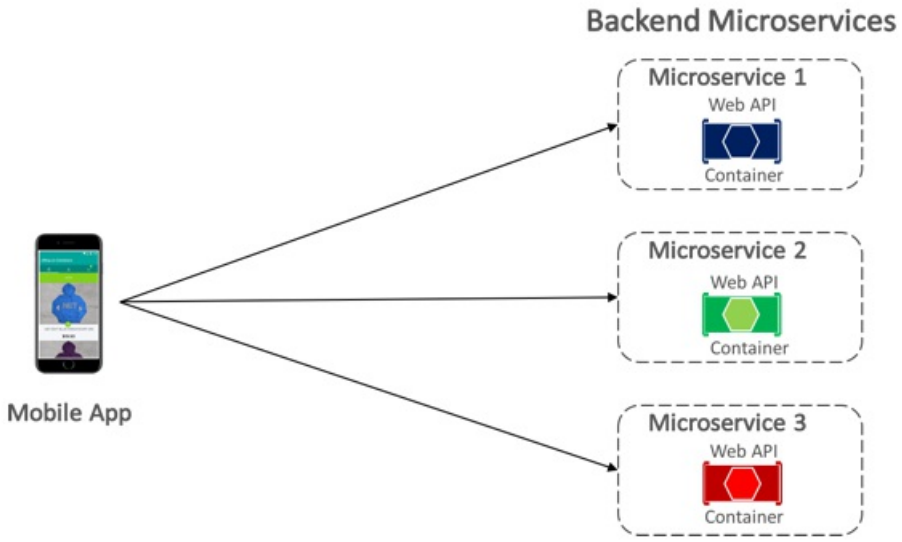

그림 8-5: 클라이언트-마이크로 서비스 간 직접 통신

클라이언트-마이크로 서비스 간 직접 통신을 사용 하면 모바일 앱은 각 마이크로 서비스에 대 한 요청을 각 마이크로 서비스에 대해 다른 TCP 포트를 사용 하 여 공용 끝점을 통해 직접 요청 합니다. 프로덕션에서 끝점은 일반적으로 사용 가능한 인스턴스 간에 요청을 분산 하는 마이크로 서비스의 부하 분산 장치에 매핑됩니다.

> **TIP**
> API 게이트웨이 통신을 사용 하는 것이 좋습니다. 클라이언트-마이크로 서비스 간 직접 통신은 크고 복잡 한 마이크로 서비스 기반 응용 프로그램을 빌드할 때 단점이 있을 수 있지만 작은 응용 프로그램에는 적합 하지 않습니다. 수십 개의 마이크로 서비스를 사용 하 여 초대형 마이크로 서비스 기반 응용 프로그램을 디자인 하는 경우 API 게이트웨이 통신을 사용 하는 것이 좋습니다. 자세한 내용은 .Net 마이크로 서비스: 컨테이너 화 된 .Net 응용 프로그램 아키텍처를 참조 하세요.

마이크로 서비스 간 통신

마이크로 서비스 기반 응용 프로그램은 여러 컴퓨터에서 실행 될 수 있는 분산 시스템입니다. 각 서비스 인스턴스는 일반적으로 프로세스입니다. 따라서 서비스는 각 서비스의 특성에 따라 HTTP, TCP, 고급 메시지 큐 프로토콜 (AMQP) 또는 이진 프로토콜과 같은 프로세스 간 통신 프로토콜을 사용 하 여 상호 작용 해야 합니다.

마이크로 서비스 간 통신에 대 한 두 가지 일반적인 방법은 데이터를 쿼리할 때 HTTP 기반 REST 통신을 사용 하 고 여러 마이크로 서비스에서 업데이트를 통신할 때 간단한 비동기 메시징을 사용 하는 것입니다.

비동기 메시징 기반 통신은 여러 마이크로 서비스에서 변경 내용을 전파 하는 경우에 중요 합니다. 이 접근 방식을 사용 하는 경우 마이크로 서비스는 비즈니스 엔터티를 업데이트할 때 처럼 주목할 만한 문제가 발생 하면 이벤트를 게시 합니다. 다른 마이크로 서비스는 이러한 이벤트를 구독 합니다. 그런 다음 마이크로 서비스가 이벤트를 받으면 자체 비즈니스 엔터티를 업데이트 하 여 더 많은 이벤트를 게시 하 게 될 수 있습니다. 이 게시-구독 기능은 일반적으로 이벤트 버스를 사용 하 여 구현 됩니다.

그림 8-6에 표시 된 것 처럼 이벤트 버스는 구성 요소가 서로를 명시적으로 인식 하지 않고도 마이크로 서비스 간의 게시-구독 통신을 허용 합니다.

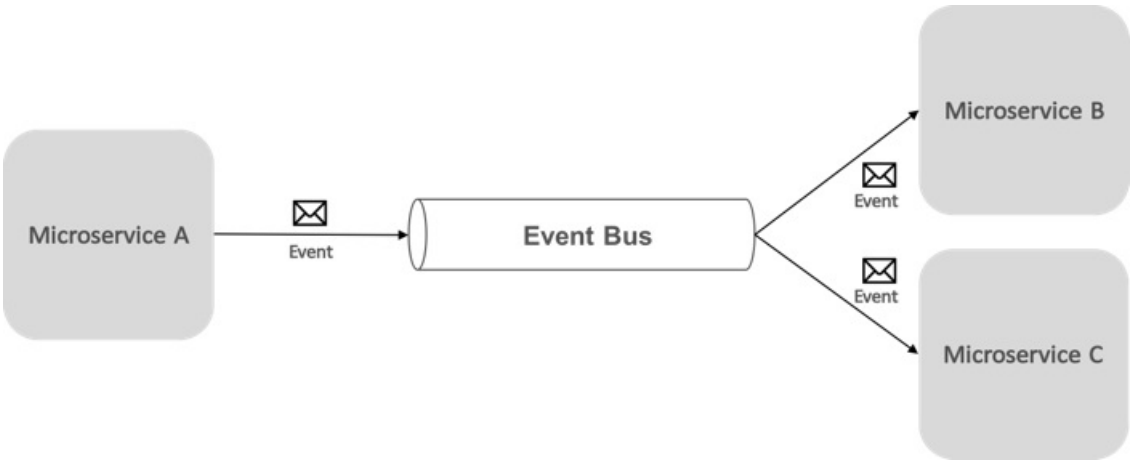

그림 8-6: 게시-이벤트 버스를 사용 하 여 구독

응용 프로그램 관점에서 이벤트 버스는 단순히 인터페이스를 통해 노출 되는 게시-구독 채널입니다. 그러나 이벤트 버스가 구현 되는 방식은 달라질 수 있습니다. 예를 들어 이벤트 버스 구현에서는 RabbitMQ, Azure Service Bus 또는 NServiceBus 및 MassTransit와 같은 다른 서비스 버스를 사용할 수 있습니다. 그림 8-7에서는 eShopOnContainers reference 응용 프로그램에서 이벤트 버스를 사용 하는 방법을 보여 줍니다.

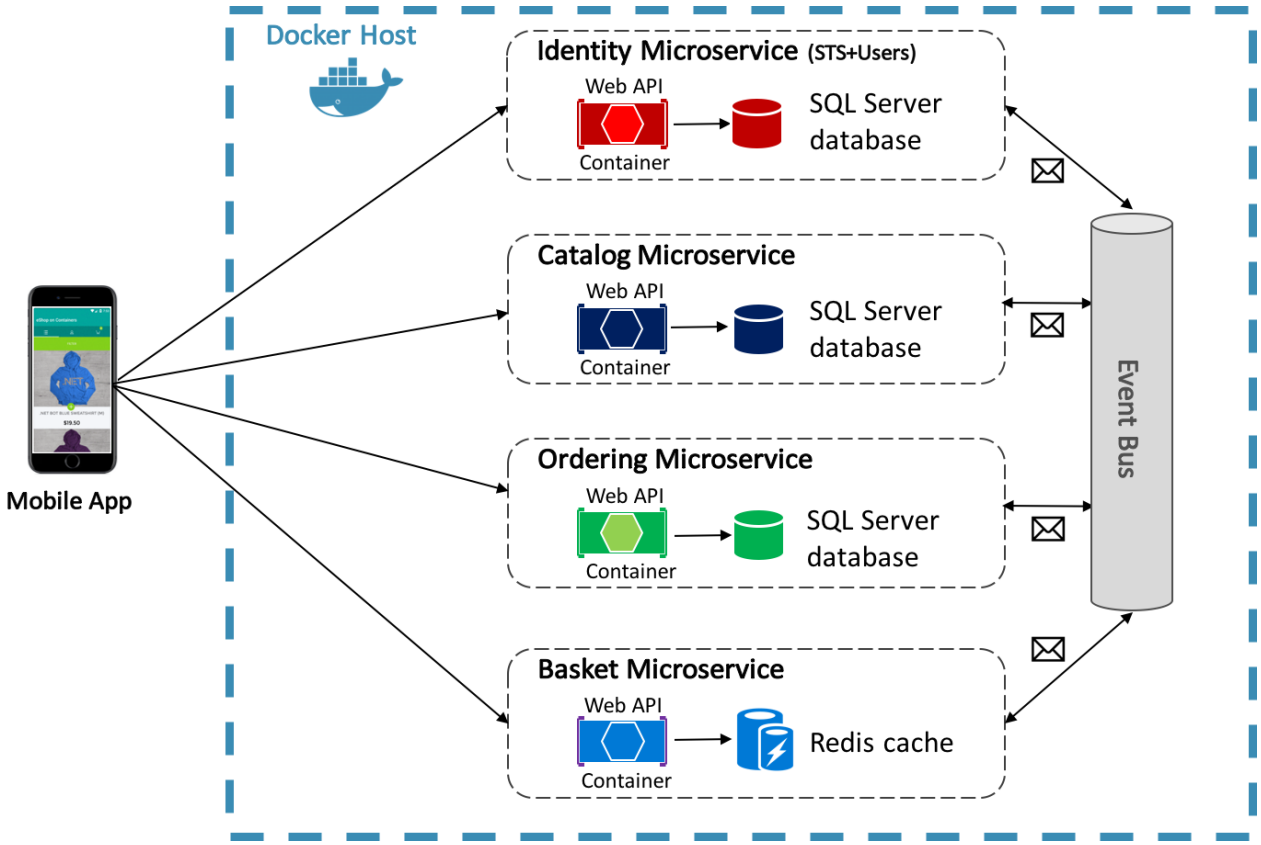

그림 8-7: 참조 응용 프로그램의 비동기 이벤트 기반 통신

RabbitMQ를 사용 하 여 구현 된 eShopOnContainers 이벤트 버스는 일대다 비동기 게시-구독 기능을 제공 합니다. 즉, 이벤트를 게시 한 후 동일한 이벤트를 수신 하는 여러 구독자가 있을 수 있습니다. 그림 8-9에서는이 관계를 보여 줍니다.

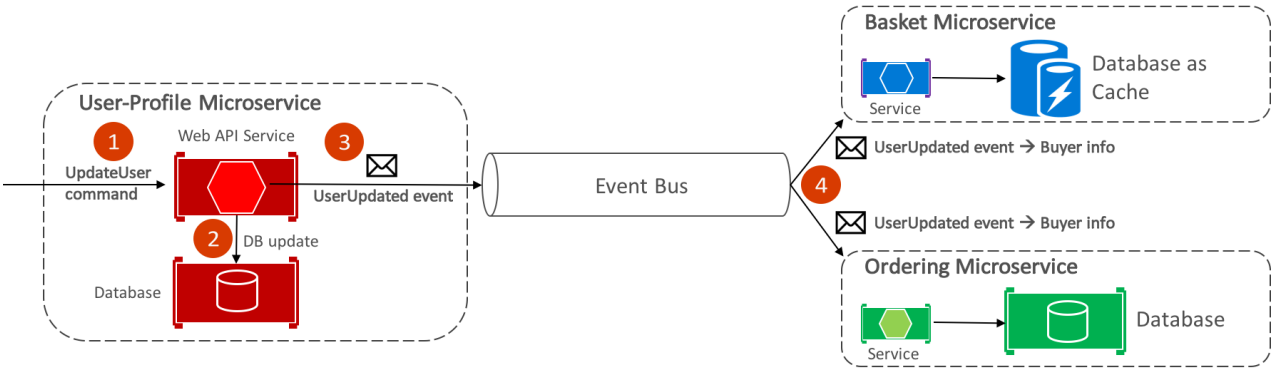

그림 8-9: 일대다 통신

이 일대다 통신 방법은 이벤트를 사용 하 여 여러 서비스에 걸쳐 있는 비즈니스 트랜잭션을 구현 하 여 서비스 간의 최종 일관성을 보장 합니다. 궁극적으로 일관 된 트랜잭션은 일련의 분산 단계로 구성 됩니다. 따라서 사용자 프로필 마이크로 서비스가 UpdateUser 명령을 받으면 해당 데이터베이스에서 사용자의 세부 정보를 업데이트 하고 UserUpdated 이벤트를 이벤트 버스에 게시 합니다. 바구니 마이크로 서비스 및 주문 마이크로 서비스 모두이 이벤트를 수신 하 고 해당 데이터베이스에서 해당 구매자 정보를 업데이트 하는 것을 구독 합니다.

> **NOTE**
> RabbitMQ를 사용 하 여 구현 된 eShopOnContainers 이벤트 버스는 개념 증명 으로만 사용 하기 위한 것입니다. 프로덕션 시스템의 경우 대체 이벤트 버스 구현을 고려해 야 합니다.

이벤트 버스 구현에 대 한 자세한 내용은 .Net 마이크로 서비스: 컨테이너 화 된 .Net 응용 프로그램 아키텍처를 참조 하세요.

요약

마이크로 서비스는 최신 클라우드 응용 프로그램의 민첩성, 확장성 및 안정성 요구 사항에 적합 한 응용 프로그램 개발 및 배포에 대 한 접근 방식을 제공 합니다. 마이크로 서비스의 주요 이점 중 하나는 독립적으로 확장 될 수 있다는 것입니다. 즉, 수요가 증가 하지 않는 응용 프로그램의 영역을 불필요 하 게 확장 하지 않고도 요구를 지원 하기 위해 더 많은 처리 능력 또는 네트워크 대역폭이 요구 되는 특정 기능 영역을 확장할 수 있습니다.

컨테이너는 다른 컨테이너 또는 호스트의 리소스를 건드리지 않고도 응용 프로그램을 실행할 수 있는 격리 된 리소스 제어 및 휴대용 운영 환경입니다. 마이크로 서비스 기반 응용 프로그램을 구현할 때 기업이 점점 더 많은 컨테이너를 채택 하 고 있으며 Docker가 대부분의 소프트웨어 플랫폼과 클라우드 공급 업체에서 채택 하는 표준 컨테이너 구현이 되었습니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

인증 및 권한 부여

2020-06-05 • 57 minutes to read • Edit Online

인증은 사용자의 이름 및 암호와 같은 id 자격 증명을 가져오고 기관에 대해 해당 자격 증명의 유효성을 검사 하는 프로세스입니다. 자격 증명이 유효 하면 자격 증명을 제출한 엔터티가 인증 된 id로 간주 됩니다. Id가 인증 되 면 권한 부여 프로세스에서 해당 id에 지정 된 리소스에 대 한 액세스 권한이 있는지 여부를 확인 합니다.

ASP.NET MVC 웹 응용 프로그램과 통신 하는 앱에 인증 및 권한 부여를 통합 하는 방법에는 여러 가지가 있습니다. 예를 들어 Xamarin.Forms ASP.NET Core id, 외부 인증 공급자 (예: Microsoft, Google, Facebook 또는 Twitter) 및 인증 미들웨어를 사용 합니다. EShopOnContainers 모바일 앱은 IdentityServer 4를 사용 하는 컨테이너 화 된 identity 마이크로 서비스를 사용 하 여 인증 및 권한 부여를 수행 합니다. 모바일 앱은 사용자를 인증 하거나 리소스에 액세스 하기 위해 IdentityServer에서 보안 토큰을 요청 합니다. IdentityServer 사용자를 대신 하 여 토큰을 발급 하려면 사용자가 IdentityServer에 로그인 해야 합니다. 그러나 IdentityServer는 인증을 위한 사용자 인터페이스 또는 데이터베이스를 제공 하지 않습니다. 따라서 eShopOnContainers 참조 응용 프로그램에서 ASP.NET Core Identity가이 목적으로 사용 됩니다.

인증

응용 프로그램에서 현재 사용자의 id를 알고 있어야 하는 경우 인증이 필요 합니다. 사용자를 식별 하는 ASP.NET Core의 기본 메커니즘은 개발자가 구성한 데이터 저장소에 사용자 정보를 저장 하는 ASP.NET Core Id 멤버 자격 시스템입니다. 일반적으로 사용자 지정 저장소나 타사 패키지를 사용 하 여 Azure storage, Azure Cosmos DB 또는 기타 위치에 id 정보를 저장할 수는 있지만이 데이터 저장소는 일반적으로 EntityFramework 저장소입니다.

로컬 사용자 데이터 저장소를 사용 하 고 쿠키를 통해 요청 간에 id 정보를 유지 하는 인증 시나리오의 경우 (ASP.NET MVC 웹 응용 프로그램에서 일반적으로) ASP.NET Core Identity가 적합 한 솔루션입니다. 그러나 쿠키는 항상 데이터를 유지 하 고 전송 하는 자연 스러운 수단이 아닙니다. 예를 들어 모바일 앱에서 액세스 하는 RESTful 끝점을 노출 하는 ASP.NET Core 웹 응용 프로그램은이 시나리오에서 쿠키를 사용할 수 없으므로 일반적으로 전달자 토큰 인증을 사용 해야 합니다. 그러나 전달자 토큰은 모바일 앱에서 수행 된 웹 요청의 권한 부여 헤더에 쉽게 검색 하 고 포함할 수 있습니다.

IdentityServer 4를 사용 하 여 전달자 토큰 발급

IdentityServer 4 는 로컬 ASP.NET Core id 사용자에 대 한 보안 토큰 발급을 비롯 한 여러 인증 및 권한 부여 시나리오에 사용할 수 있는 ASP.NET Core에 대 한 오픈 소스 openid connect Connect 및 OAuth 2.0 프레임 워크입니다.

> **NOTE**
> Openid connect Connect와 OAuth 2.0은 매우 비슷하며 책임이 서로 다릅니다.

Openid connect Connect는 OAuth 2.0 프로토콜을 기반으로 하는 인증 계층입니다. OAuth 2는 응용 프로그램이 보안 토큰 서비스에서 액세스 토큰을 요청 하 고 Api와 통신 하는 데 사용할 수 있는 프로토콜입니다. 이 위임은 인증 및 권한 부여를 중앙 집중화할 수 있으므로 클라이언트 응용 프로그램 및 Api의 복잡성을 줄입니다.

Openid connect Connect와 OAuth 2.0의 조합은 인증 및 API 액세스의 두 가지 기본적인 보안 문제를 결합 하 고, IdentityServer 4는 이러한 프로토콜의 구현입니다.

EShopOnContainers reference 응용 프로그램과 같은 직접 클라이언트-마이크로 서비스 통신을 사용 하는 응용 프로그램에서 STS (보안 토큰 서비스)로 작동 하는 전용 인증 마이크로 서비스는 그림 9-1에 표시 된 것 처럼 사용자를 인증 하는 데 사용할 수 있습니다. 클라이언트-마이크로 서비스 간 직접 통신에 대 한 자세한 내용은 클라이언트와 마이크로 서비스 간 통신을 참조 하세요.

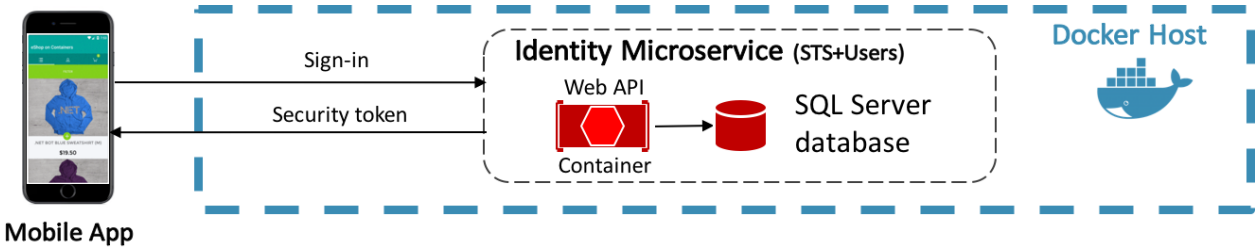

그림 9-1: 전용 인증 마이크로 서비스 인증

EShopOnContainers 모바일 앱은 IdentityServer 4를 사용 하 여 인증을 수행 하 고 Api에 대 한 액세스를 제어 하는 id 마이크로 서비스와 통신 합니다. 따라서 모바일 앱은 사용자를 인증 하거나 리소스에 액세스 하기 위해 IdentityServer의 토큰을 요청 합니다.

- IdentityServer를 사용 하 여 사용자를 인증 하는 것은 인증 프로세스의 결과를 나타내는 *id* 토큰을 요청 하는 모바일 앱에 의해 수행 됩니다. 최소한의 경우 사용자에 대 한 식별자와 사용자가 인증 하는 방법 및 시기에 대 한 정보를 포함 합니다. 추가 id 데이터를 포함할 수도 있습니다.
- IdentityServer를 사용 하 여 리소스에 액세스 하는 것은 API 리소스에 대 한 액세스를 허용 하는 *액세스* 토큰을 요청 하는 모바일 앱에 의해 구현 됩니다. 클라이언트는 액세스 토큰을 요청 하 고 API에 전달 합니다. 액세스 토큰에는 클라이언트 및 사용자 (있는 경우)에 대 한 정보가 포함 됩니다. 그런 다음 Api는 해당 정보를 사용 하 여 해당 데이터에 대 한 액세스 권한을 부여 합니다.

> **NOTE**
> 클라이언트는 IdentityServer에 등록 해야 토큰을 요청할 수 있습니다.

웹 응용 프로그램에 **IdentityServer** 추가

ASP.NET Core 웹 응용 프로그램에서 IdentityServer 4를 사용 하려면 웹 응용 프로그램의 Visual Studio 솔루션에 추가 해야 합니다. 자세한 내용은 IdentityServer 설명서에서 개요 를 참조 하세요.

IdentityServer가 웹 응용 프로그램의 Visual Studio 솔루션에 포함 되 면 웹 응용 프로그램의 HTTP 요청 처리 파이프라인에 추가 해야 Openid connect Connect 및 OAuth 2.0 끝점에 대 한 요청을 처리할 수 있습니다. 이는 `Configure` `Startup` 다음 코드 예제에서 보여 주는 것 처럼 웹 응용 프로그램의 클래스에서 메서드를 통해 얻을 수 있습니다.

```
public void Configure(
    IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    ...
    app.UseIdentity();
    ...
}
```

웹 응용 프로그램의 HTTP 요청 처리 파이프라인에서 순서가 중요 합니다. 따라서 로그인 화면을 구현 하는 UI 프레임 워크 앞에 IdentityServer를 파이프라인에 추가 해야 합니다.

IdentityServer 구성

`ConfigureServices` `Startup` `services.AddIdentityServer` EShopOnContainers reference 응용 프로그램의 다음 코드 예제에서 설명한 것 처럼 메서드를 호출 하 여 웹 응용 프로그램의 클래스에 있는 메서드에서 IdentityServer을 구성 해야 합니다.

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddIdentityServer(x => x.IssuerUri = "null")
        .AddSigningCredential(Certificate.Get())
        .AddAspNetIdentity<ApplicationUser>()
        .AddConfigurationStore(builder =>
            builder.UseSqlServer(connectionString, options =>
                options.MigrationsAssembly(migrationsAssembly)))
        .AddOperationalStore(builder =>
            builder.UseSqlServer(connectionString, options =>
                options.MigrationsAssembly(migrationsAssembly)))
        .Services.AddTransient<IProfileService, ProfileService>();
}
```

메서드를 호출한 후에 `services.AddIdentityServer` 는 다음을 구성 하기 위해 추가 흐름 api가 호출 됩니다.

- 서명에 사용 되는 자격 증명입니다.
- 사용자가 액세스를 요청할 수 있는 API 및 id 리소스입니다.
- 요청 토큰에 연결 되는 클라이언트입니다.
- ASP.NET Core Id입니다.

> **TIP**
> IdentityServer 4 구성을 동적으로 로드 합니다. IdentityServer 4의 Api를 사용 하면 구성 개체의 메모리 내 목록에서 IdentityServer를 구성할 수 있습니다. EShopOnContainers reference 응용 프로그램에서 이러한 메모리 내 컬렉션은 응용 프로그램에 하드 코딩 됩니다. 그러나 프로덕션 시나리오에서는 구성 파일이 나 데이터베이스에서 동적으로 로드할 수 있습니다.

ASP.NET Core Identity를 사용 하도록 IdentityServer를 구성 하는 방법에 대 한 자세한 내용은 IdentityServer 설명서에서 ASP.NET Core Identity 사용 을 참조 하세요.

API 리소스 구성

API 리소스를 구성할 때 `AddInMemoryApiResources` 메서드에는 컬렉션이 필요 합니다 `IEnumerable<ApiResource>` . 다음 코드 예제에서는 `GetApis` eShopOnContainers reference 응용 프로그램에서이 컬렉션을 제공 하는 메서드를 보여 줍니다.

```
public static IEnumerable<ApiResource> GetApis()
{
    return new List<ApiResource>
    {
        new ApiResource("orders", "Orders Service"),
        new ApiResource("basket", "Basket Service")
    };
}
```

이 메서드는 IdentityServer가 주문 및 바구니 Api를 보호 하도록 지정 합니다. 따라서 IdentityServer 관리 액세스 토큰은 이러한 Api를 호출할 때 필요 합니다. 형식에 대 한 자세한 내용은 `ApiResource` IdentityServer 4 설명서의 API 리소스 를 참조 하세요.

Id 리소스 구성

Id 리소스를 구성할 때 `AddInMemoryIdentityResources` 메서드에는 컬렉션이 필요 합니다 `IEnumerable<IdentityResource>` . Id 리소스는 사용자 ID, 이름, 전자 메일 주소 등의 데이터입니다. 각 id 리소스는 고유한 이름을 가지 며 임의의 클레임 유형을 할당할 수 있으며,이는 사용자의 id 토큰에 포함 됩니다. 다음 코드 예제에서는 `GetResources` eShopOnContainers reference 응용 프로그램에서이 컬렉션을 제공 하는 메서드를 보여 줍니다.

```
public static IEnumerable<IdentityResource> GetResources()
{
    return new List<IdentityResource>
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile()
    };
}
```

Openid connect Connect 사양은 일부 표준 id 리소스를 지정 합니다. 최소 요구 사항은 사용자의 고유 ID를 내보내기 위한 지원이 제공 된다는 것입니다. 이는 id 리소스를 노출 하 여 이루어집니다 `IdentityResources.OpenId` .

> **NOTE**
>
> `IdentityResources` 클래스는 Openid connect Connect 사양 (openid connect, 메일, 프로필, 전화, 주소)에 정의 된 모든 범위를 지원 합니다.

또한 IdentityServer는 사용자 지정 id 리소스의 정의를 지원 합니다. 자세한 내용은 IdentityServer 설명서의 사용자 지정 id 리소스 정의 를 참조 하세요. 형식에 대 한 자세한 내용은 `IdentityResource` IdentityServer 4 설명서의 id 리소스 를 참조 하세요.

클라이언트 구성

클라이언트는 IdentityServer에서 토큰을 요청할 수 있는 응용 프로그램입니다. 일반적으로 각 클라이언트에 대해 다음 설정을 최소한으로 정의 해야 합니다.

- 고유한 클라이언트 ID입니다.
- 토큰 서비스와의 허용 되는 상호 작용 (권한 유형 이라고 함)입니다.
- Id 및 액세스 토큰이 전송 되는 위치 (리디렉션 URI)입니다.
- 클라이언트에서 액세스할 수 있는 리소스의 목록 (범위 라고 함)입니다.

클라이언트를 구성할 때 `AddInMemoryClients` 메서드에는 컬렉션이 필요 합니다 `IEnumerable<Client>` . 다음 코드 예제에서는 `GetClients` eShopOnContainers reference 응용 프로그램에서이 컬렉션을 제공 하는 메서드의 eShopOnContainers mobile 앱에 대 한 구성을 보여 줍니다.

```
public static IEnumerable<Client> GetClients(Dictionary<string,string> clientsUrl)
{
    return new List<Client>
    {
        ...
        new Client
        {
            ClientId = "xamarin",
            ClientName = "eShop Xamarin OpenId Client",
            AllowedGrantTypes = GrantTypes.Hybrid,
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },
            RedirectUris = { clientsUrl["Xamarin"] },
            RequireConsent = false,
            RequirePkce = true,
            PostLogoutRedirectUris = { $"{clientsUrl["Xamarin"]}/Account/Redirecting" },
            AllowedCorsOrigins = { "http://eshopxamarin" },
            AllowedScopes = new List<string>
            {
                IdentityServerConstants.StandardScopes.OpenId,
                IdentityServerConstants.StandardScopes.Profile,
                IdentityServerConstants.StandardScopes.OfflineAccess,
                "orders",
                "basket"
            },
            AllowOfflineAccess = true,
            AllowAccessTokensViaBrowser = true
        },
        ...
    };
}
```

이 구성은 다음 속성에 대 한 데이터를 지정 합니다.

- `ClientId`: 클라이언트에 대 한 고유 ID입니다.
- `ClientName`: 로깅 및 동의 화면에 사용 되는 클라이언트 표시 이름입니다.
- `AllowedGrantTypes`: 클라이언트가 IdentityServer와 상호 작용 하는 방법을 지정 합니다. 자세한 내용은 인증 흐름 구성을 참조 하세요.
- `ClientSecrets`: 토큰 끝점에서 토큰을 요청할 때 사용 되는 클라이언트 암호 자격 증명을 지정 합니다.
- `RedirectUris`: 토큰 또는 권한 부여 코드를 반환할 허용 되는 Uri를 지정 합니다.
- `RequireConsent`: 동의 화면이 필요한 지 여부를 지정 합니다.
- `RequirePkce`: 인증 코드를 사용 하는 클라이언트가 증명 키를 보내야 하는지 여부를 지정 합니다.
- `PostLogoutRedirectUris`: 로그 아웃 후에 리디렉션할 허용 되는 Uri를 지정 합니다.
- `AllowedCorsOrigins`: IdentityServer 원본에서 크로스-원본 호출을 허용할 수 있도록 클라이언트의 원본을 지정 합니다.
- `AllowedScopes`: 클라이언트에서 액세스할 수 있는 리소스를 지정 합니다. 기본적으로 클라이언트에는 모든 리소스에 대 한 액세스 권한이 없습니다.
- `AllowOfflineAccess`: 클라이언트에서 새로 고침 토큰을 요청할 수 있는지 여부를 지정 합니다.

인증 흐름 구성

클라이언트와 IdentityServer 간의 인증 흐름은 속성에서 grant 유형을 지정 하 여 구성할 수 있습니다 `Client.AllowedGrantTypes`. Openid connect Connect 및 OAuth 2.0 사양은 다음을 비롯 한 다양 한 인증 흐름을 정의 합니다.

- 암시적 이 흐름은 브라우저 기반 응용 프로그램에 최적화 되어 있으며 사용자 인증 전용 이거나 인증 및 액세스 토큰 요청에 사용 되어야 합니다. 모든 토큰은 브라우저를 통해 전송 되므로 새로 고침 토큰과 같은 고급 기능

을 사용할 수 없습니다.

- 인증 코드입니다. 이 흐름은 클라이언트 인증을 지 원하는 한편 브라우저 전면 채널과 달리 백 채널에서 토큰을 검색 하는 기능을 제공 합니다.
- 혼합. 이 흐름은 암시적 및 권한 부여 코드 권한 유형을 조합한 것입니다. Id 토큰은 브라우저 채널을 통해 전송 되 고 인증 코드와 같은 다른 아티팩트와 함께 서명 된 프로토콜 응답을 포함 합니다. 응답의 유효성 검사를 성공적으로 수행한 후에는 백 채널을 사용 하 여 액세스 및 새로 고침 토큰을 검색 해야 합니다.

> **TIP**
> 하이브리드 인증 흐름을 사용 합니다. 하이브리드 인증 흐름은 브라우저 채널에 적용 되는 여러 공격을 완화 하 고, 액세스 토큰을 검색 하 고 토큰을 새로 고치는 네이티브 응용 프로그램에 권장 되는 흐름입니다.

인증 흐름에 대 한 자세한 내용은 IdentityServer 4 설명서의 Grant Types 를 참조 하십시오.

인증 수행

IdentityServer 사용자를 대신 하 여 토큰을 발급 하려면 사용자가 IdentityServer에 로그인 해야 합니다. 그러나 IdentityServer는 인증을 위한 사용자 인터페이스 또는 데이터베이스를 제공 하지 않습니다. 따라서 eShopOnContainers 참조 응용 프로그램에서 ASP.NET Core Identity가이 목적으로 사용 됩니다.

EShopOnContainers 모바일 앱은 그림 9-2에 나와 있는 하이브리드 인증 흐름을 사용 하 여 IdentityServer를 사용 하 여 인증 합니다.

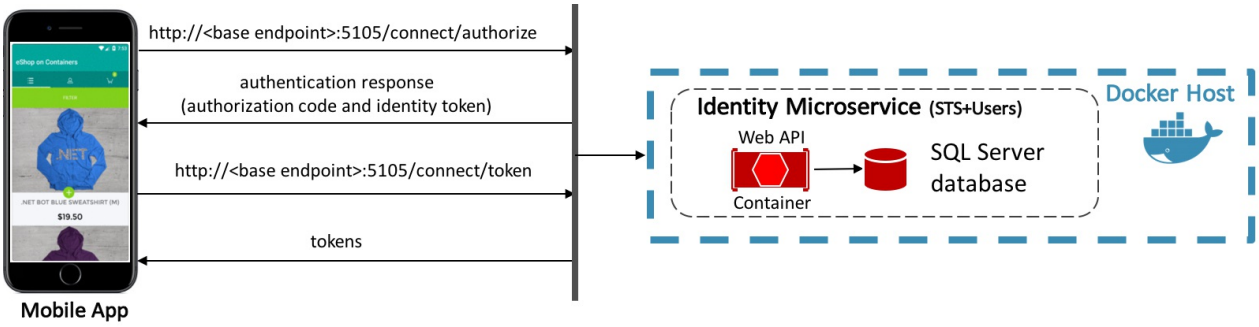

그림 9-2: 로그인 프로세스에 대 한 개략적인 개요

에 대 한 로그인 요청이 발생 `<base endpoint>:5105/connect/authorize` 합니다. 성공적인 인증 후 IdentityServer는 인증 코드와 id 토큰을 포함 하는 인증 응답을 반환 합니다. 그런 다음 인증 코드는 `<base endpoint>:5105/connect/token` 액세스, id 및 새로 고침 토큰을 사용 하 여 응답 하는에 전송 됩니다.

EShopOnContainers 모바일 앱은에 요청을 전송 하 여 추가 매개 변수를 사용 하 여 IdentityServer를 로그 아웃 합니다 `<base endpoint>:5105/connect/endsession` . 로그 아웃 발생 후 IdentityServer는 post 로그 아웃 리디렉션 URI 를 모바일 앱으로 다시 전송 하 여 응답 합니다. 그림 9-3에서는이 프로세스를 보여 줍니다.

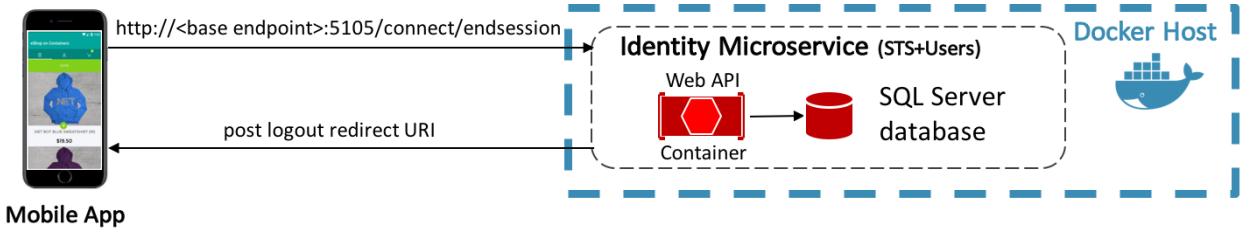

그림 9-3: 로그 아웃 프로세스에 대 한 개략적인 개요

EShopOnContainers 모바일 앱에서 IdentityServer와의 통신은 인터페이스를 구현 하는 클래스에 의해 수행 됩니다 `IdentityService` `IIdentityService` . 이 인터페이스는 구현 하는 클래스에서 `CreateAuthorizationRequest` , 및 메서드를 제공 하도록 지정 합니다 `CreateLogoutRequest` `GetTokenAsync` .

로그인

사용자가에서 **로그인** 단추를 누르면 `LoginView` `SignInCommand` 클래스의가 `LoginViewModel` 실행 되어 메서드가 실행 됩니다 `SignInAsync` . 다음 코드 예제에서는 이 메서드를 보여줍니다.

```
private async Task SignInAsync()
{
    ...
    LoginUrl = _identityService.CreateAuthorizationRequest();
    IsLogin = true;
    ...
}
```

이 메서드는 클래스의 메서드를 호출 합니다 .이 메서드는 `CreateAuthorizationRequest` `IdentityService` 다음 코드 예제에 나와 있습니다.

```
public string CreateAuthorizationRequest()
{
    // Create URI to authorization endpoint
    var authorizeRequest = new AuthorizeRequest(GlobalSetting.Instance.IdentityEndpoint);

    // Dictionary with values for the authorize request
    var dic = new Dictionary<string, string>();
    dic.Add("client_id", GlobalSetting.Instance.ClientId);
    dic.Add("client_secret", GlobalSetting.Instance.ClientSecret);
    dic.Add("response_type", "code id_token");
    dic.Add("scope", "openid profile basket orders locations marketing offline_access");
    dic.Add("redirect_uri", GlobalSetting.Instance.Callback);
    dic.Add("nonce", Guid.NewGuid().ToString("N"));
    dic.Add("code_challenge", CreateCodeChallenge());
    dic.Add("code_challenge_method", "S256");

    // Add CSRF token to protect against cross-site request forgery attacks.
    var currentCSRFToken = Guid.NewGuid().ToString("N");
    dic.Add("state", currentCSRFToken);

    var authorizeUri = authorizeRequest.Create(dic);
    return authorizeUri;
}
```

이 메서드는 필수 매개 변수를 사용 하 여 IdentityServer의 authorization 끝점에 대 한 URI를 만듭니다. 권한 부여 끝점은 `/connect/authorize` 사용자 설정으로 노출 된 기본 끝점의 포트 5105에 있습니다. 사용자 설정에 대 한 자세한 내용은 구성 관리를 참조 하세요.

> **NOTE**
> EShopOnContainers 모바일 앱의 공격 노출 영역을 OAuth에 대 한 코드 교환 (PKCE) 확장을 구현 하 여 줄일 수 있습니다. PKCE는 권한 부여 코드가 가로채 면 사용 되지 않도록 보호 합니다. 이는 비밀 검증 도구를 생성 하는 클라이언트, 권한 부여 요청에 전달 되는 해시 및 인증 코드를 교환 때 해시 되지 않은 형태로 제공 됩니다. PKCE에 대 한 자세한 내용은 Internet 엔지니어링 작업 Force 웹 사이트에서 OAuth 공용 클라이언트의 코드 교환에 대 한 증명 키 를 참조 하세요.

반환 된 URI는 `LoginUrl` 클래스의 속성에 저장 됩니다 `LoginViewModel` . `IsLogin` 속성이 이면의이 `true` 표시 됩니다 `WebView` `LoginView` . 데이터는 속성 `WebView` `Source` 을 `LoginUrl` 클래스의 속성에 `LoginViewModel` IdentityServer `LoginUrl` 속성이 IdentityServer의 인증 끝점으로 설정 된 경우 로그인 요청을 호출 합니다. IdentityServer이 요청을 수신 하 고 사용자가 인증 되지 않은 경우는 `WebView` 그림 9-4에 표시 된 구성 된 로그인 페이지로 리디렉션됩니다.

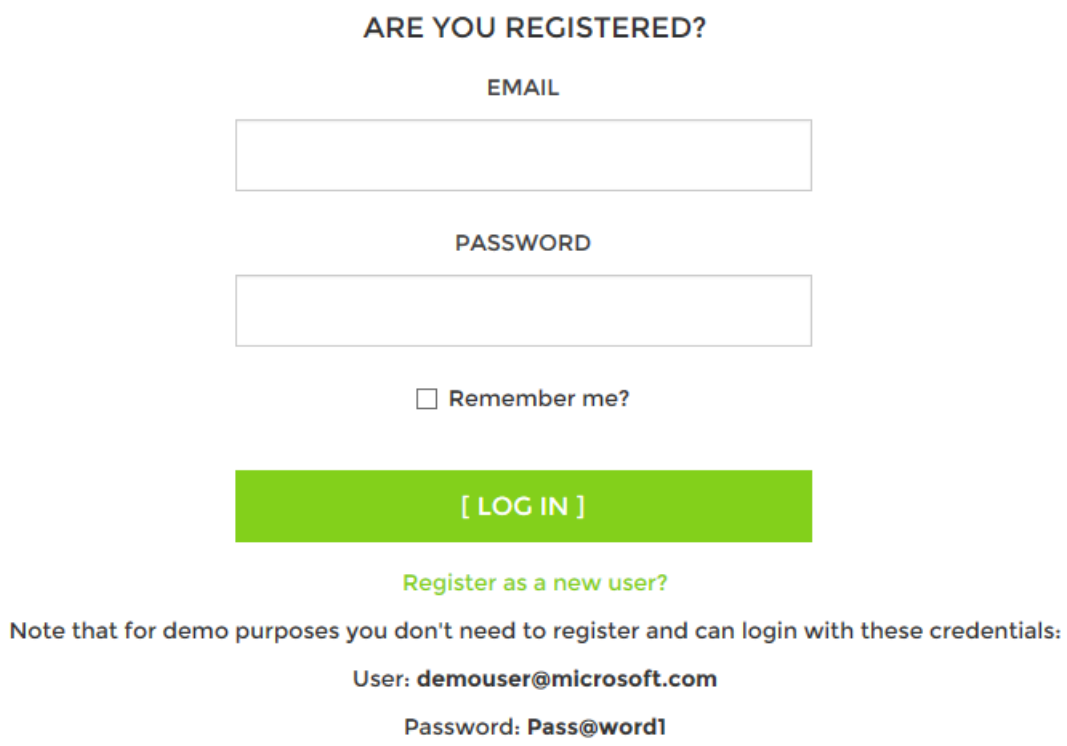

그림 9-4: 웹 보기에서 표시 하는 로그인 페이지

로그인이 완료 되 면이 `WebView` 반환 URI로 리디렉션됩니다. 이 `WebView` 탐색은 `NavigateAsync` 클래스의 메서드를 실행 합니다 .이는 `LoginViewModel` 다음 코드 예제에 나와 있습니다.

```
private async Task NavigateAsync(string url)
{
    ...
    var authResponse = new AuthorizeResponse(url);
    if (!string.IsNullOrWhiteSpace(authResponse.Code))
    {
        var userToken = await _identityService.GetTokenAsync(authResponse.Code);
        string accessToken = userToken.AccessToken;

        if (!string.IsNullOrWhiteSpace(accessToken))
        {
            Settings.AuthAccessToken = accessToken;
            Settings.AuthIdToken = authResponse.IdentityToken;

            await NavigationService.NavigateToAsync<MainViewModel>();
            await NavigationService.RemoveLastFromBackStackAsync();
        }
    }
    ...
}
```

이 메서드는 반환 URI에 포함 된 인증 응답을 구문 분석 하 고, 유효한 인증 코드가 있으면 IdentityServer의 토큰 끝점에 요청 하 여 인증 코드, pkce 비밀 검증 도구 및 기타 필수 매개 변수를 전달 합니다. 토큰 끝점은 `/connect/token` 사용자 설정으로 노출 된 기본 끝점의 포트 5105에 있습니다. 사용자 설정에 대 한 자세한 내용은 구성 관리를 참조 하세요.

> **TIP**
> 반환 Uri의 유효성을 검사 합니다. EShopOnContainers 모바일 앱은 반환 URI의 유효성을 검사 하지 않지만, 가장 좋은 방법은 반환 URI가 알려진 위치를 참조 하는지 확인 하 여 열린 리디렉션 공격을 방지 하는 것입니다.

토큰 끝점이 유효한 인증 코드 및 PKCE 비밀 검증 도구를 수신 하면 액세스 토큰, id 토큰 및 새로 고침 토큰을 사용하 여 응답 합니다. 그러면 API 리소스에 대 한 액세스를 허용 하는 액세스 토큰 및 id 토큰이 응용 프로그램 설정으로 저장 되 고 페이지 탐색이 수행 됩니다. 따라서 eShopOnContainers 모바일 앱의 전반적인 효과는 다음과 같습니다. 사용자가 IdentityServer를 사용 하 여 성공적으로 인증할 수 있는 경우를 `MainView` `TabbedPage` 선택 된 탭으로 표시 하는 인 페이지를 탐색 `CatalogView` 합니다.

페이지 탐색에 대 한 자세한 내용은 탐색을 참조 하세요. 탐색에서 뷰 모델 메서드를 실행 하는 방법에 대 한 자세한 내용은 `WebView` 동작을 사용 하 여 탐색 호출을 참조 하세요. 응용 프로그램 설정에 대 한 자세한 내용은 구성 관리를 참조 하세요.

> **NOTE**
> EShopOnContainers는 앱이에서 모의 서비스를 사용 하도록 구성 된 경우에도 모의 로그인을 허용 합니다 `SettingsView` .
> 이 모드에서 앱은 IdentityServer와 통신 하지 않고 대신 사용자가 자격 증명을 사용 하 여 로그인 할 수 있도록 허용 합니다.

로그 아웃

사용자가에서 **로그 아웃** 단추를 누르면 `ProfileView` `LogoutCommand` 클래스의가 `ProfileViewModel` 실행 되어 메서드가 실행 됩니다 `LogoutAsync` . 이 메서드는 페이지로의 페이지 탐색 `LoginView` 을 수행 하 여 `LogoutParameter` 로 설정 된 인스턴스를 `true` 매개 변수로 전달 합니다. 페이지를 탐색 하는 동안 매개 변수를 전달 하는 방법에 대 한 자세한 내용은 탐색 하는 동안 매개 변수 전달

뷰가 생성 되 고 탐색 되 면 `InitializeAsync` 뷰의 연결 된 뷰 모델의 메서드가 실행 되며,이 메서드는 `Logout` `LoginViewModel` 다음 코드 예제에 표시 된 클래스의 메서드를 실행 합니다.

```
private void Logout()
{
    var authIdToken = Settings.AuthIdToken;
    var logoutRequest = _identityService.CreateLogoutRequest(authIdToken);

    if (!string.IsNullOrEmpty(logoutRequest))
    {
        // Logout
        LoginUrl = logoutRequest;
    }
    ...
}
```

이 메서드는 `CreateLogoutRequest` 클래스에서 메서드를 호출 `IdentityService` 하 여 응용 프로그램 설정에서 검색된 id 토큰을 매개 변수로 전달 합니다. 응용 프로그램 설정에 대 한 자세한 내용은 구성 관리를 참조 하세요. 다음 코드 예제는 `CreateLogoutRequest` 메서드를 보여줍니다.

```
public string CreateLogoutRequest(string token)
{
    ...
    return string.Format("{0}?id_token_hint={1}&post_logout_redirect_uri={2}",
        GlobalSetting.Instance.LogoutEndpoint,
        token,
        GlobalSetting.Instance.LogoutCallback);
}
```

이 메서드는 필수 매개 변수를 사용 하 여 IdentityServer의 end session 끝점에 대 한 URI를 만듭니다. End session 끝점은 `/connect/endsession` 사용자 설정으로 노출 된 기본 끝점의 포트 5105에 있습니다. 사용자 설정에 대 한 자세한 내용은 구성 관리를 참조 하세요.

반환 된 URI는 `LoginUrl` 클래스의 속성에 저장 됩니다 `LoginViewModel` . `IsLogin` 속성이 인 동안의 `true` 는 `WebView` `LoginView` 표시 됩니다. 데이터는 속성 `WebView` `Source` 을 `LoginUrl` 클래스의 속성에 바인딩한

`LoginViewModel` 다음, `LoginUrl` 속성이 IdentityServer의 end session Endpoint로 설정 된 경우 IdentityServer에 대한 로그 아웃 요청을 만듭니다. 사용자가 로그인 하는 경우 IdentityServer가이 요청을 받으면 로그 아웃이 발생 합니다. 인증은 ASP.NET Core에서 쿠키 인증 미들웨어에 의해 관리 되는 쿠키를 사용 하 여 추적 됩니다. 따라서 IdentityServer에서 로그 아웃 하면 인증 쿠키가 제거 되 고 사후 로그 아웃 리디렉션 URI가 클라이언트에 다시 전송 됩니다.

모바일 앱에서가 `WebView` 사후 로그 아웃 리디렉션 URI로 리디렉션됩니다. 이 `WebView` 탐색은 `NavigateAsync` 클래스의 메서드를 실행 합니다 .이는 `LoginViewModel` 다음 코드 예제에 나와 있습니다.

```
private async Task NavigateAsync(string url)
{
    ...
    Settings.AuthAccessToken = string.Empty;
    Settings.AuthIdToken = string.Empty;
    IsLogin = false;
    LoginUrl = _identityService.CreateAuthorizationRequest();
    ...
}
```

이 메서드는 응용 프로그램 설정에서 id 토큰과 액세스 토큰을 모두 지우고 속성을로 설정 하 여 `IsLogin` `false` 페이지의이 `WebView` `LoginView` 보이지 않게 합니다. 마지막으로 `LoginUrl` 속성은 다음에 사용자가 로그인을 시작할 때 준비 하는 데 필요한 매개 변수를 사용 하 여 IdentityServer의 AUTHORIZATION 끝점URI로 설정 됩니다.

페이지 탐색에 대 한 자세한 내용은 탐색을 참조 하세요. 탐색에서 뷰 모델 메서드를 실행 하는 방법에 대 한 자세한 내용은 `WebView` 동작을 사용 하 여 탐색 호출을 참조 하세요. 응용 프로그램 설정에 대 한 자세한 내용은 구성 관리를 참조 하세요.

> **NOTE**
> 또한 eShopOnContainers를 사용 하면 앱이 SettingsView에서 모의 서비스를 사용 하도록 구성 된 경우 모의 로그 아웃을 사용할 수 있습니다. 이 모드에서 앱은 IdentityServer와 통신 하지 않고 응용 프로그램 설정에서 저장 된 토큰을 모두 지웁니다.

권한 부여

인증 후에는 ASP.NET Core 웹 Api에서 액세스 권한을 부여 해야 하는 경우가 많습니다 .이 경우 서비스에서 일부 인증 된 사용자에 게 Api를 사용할 수 있지만 all은 허용 되지 않습니다.

컨트롤러 또는 작업에 권한 부여 특성을 적용 하 여 ASP.NET Core MVC 경로에 대 한 액세스를 제한할 수 있습니다 .이는 다음 코드 예제에 표시 된 것 처럼 컨트롤러 또는 작업에 대 한 액세스를 인증 된 사용자로 제한 합니다.

```
[Authorize]
public class BasketController : Controller
{
    ...
}
```

권한이 없는 사용자가 특성으로 표시 된 컨트롤러나 작업에 액세스 하려고 하면 `Authorize` MVC 프레임 워크에서 401 (권한 없음) HTTP 상태 코드를 반환 합니다.

> **NOTE**
> 특성에 매개 변수를 지정 `Authorize` 하 여 API를 특정 사용자로 제한할 수 있습니다. 자세한 내용은 권한 부여를 참조하세요.

IdentityServer는 제어 권한 부여를 제공 하는 액세스 토큰을 권한 부여 워크플로에 통합할 수 있습니다. 이 방법은 그림 9-5에 나와 있습니다.

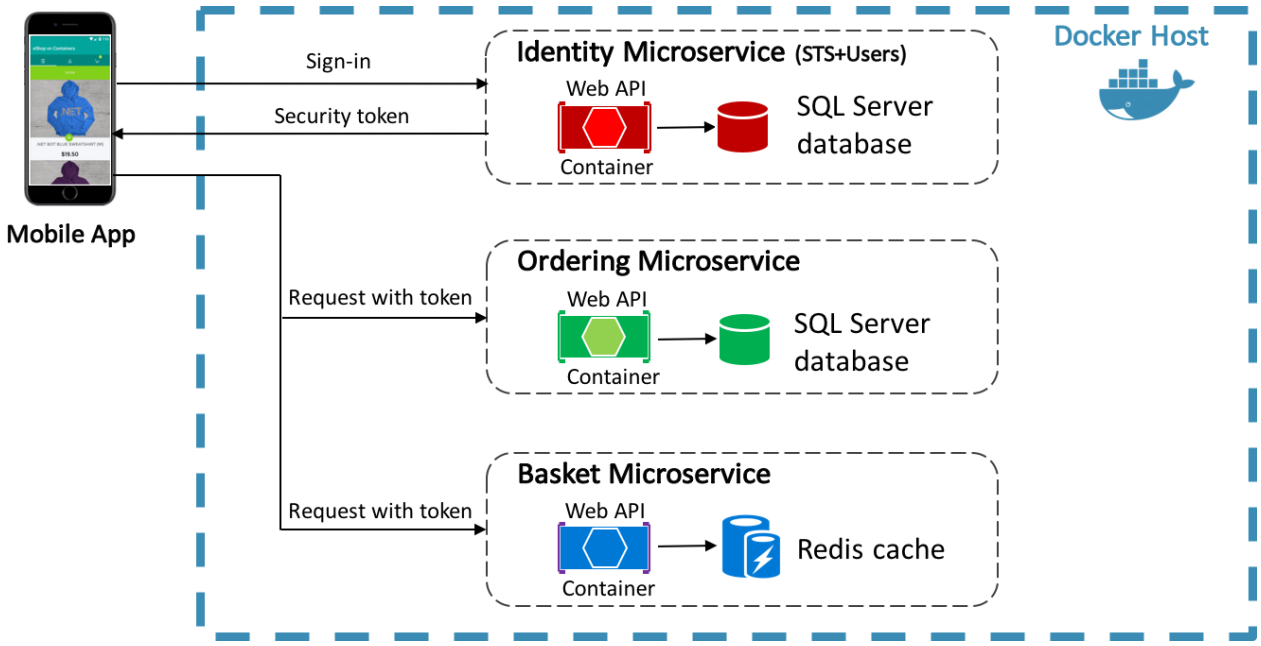

그림 9-5: 액세스 토큰에의 한 권한 부여

EShopOnContainers 모바일 앱은 id 마이크로 서비스와 통신 하 고 인증 프로세스의 일부로 액세스 토큰을 요청 합니다. 그런 다음 액세스 토큰은 주문 및 바구니 마이크로 서비스에 의해 노출 되는 Api에 액세스 요청의 일부로 전달 됩니다. 액세스 토큰에는 클라이언트 및 사용자에 대 한 정보가 포함 됩니다. 그런 다음 Api는 해당 정보를 사용 하 여 해당 데이터에 대 한 액세스 권한을 부여 합니다. Api를 보호 하도록 IdentityServer을 구성 하는 방법에 대 한 자세한 내용은 Api 리소스 구성을 참조 하세요.

권한 부여를 수행 하도록 **IdentityServer** 구성

IdentityServer를 사용 하 여 권한 부여를 수행 하려면 해당 권한 부여 미들웨어를 웹 응용 프로그램의 HTTP 요청 파이프라인에 추가 해야 합니다. 미들웨어는 `ConfigureAuth` 메서드에서 호출 되는 웹 응용 프로그램의 클래스에 있는 메서드에서 추가 되며 `Startup` `Configure` eShopOnContainers reference 응용 프로그램의 다음 코드 예제에서 보여 줍니다.

```
protected virtual void ConfigureAuth(IApplicationBuilder app)
{
    var identityUrl = Configuration.GetValue<string>("IdentityUrl");
    app.UseIdentityServerAuthentication(new IdentityServerAuthenticationOptions
    {
        Authority = identityUrl.ToString(),
        ScopeName = "basket",
        RequireHttpsMetadata = false
    });
}
```

이 메서드는 유효한 액세스 토큰을 사용 하 여 API에만 액세스할 수 있도록 합니다. 미들웨어는 들어오는 토큰의 유효성을 검사 하 여 신뢰할 수 있는 발급자 로부터 보내고 토큰을 수신 하는 API와 토큰을 사용할 수 있는지 확인 합니다. 따라서 주문 또는 바구니 컨트롤러를 검색 하면 액세스 토큰이 필요 함을 나타내는 401 (권한 없음) HTTP 상태 코드가 반환 됩니다.

> **NOTE**
> 또는를 사용 하 여 MVC를 추가 하기 전에 IdentityServer의 인증 미들웨어를 웹 응용 프로그램의 HTTP 요청 파이프라인에 추가 해야 합니다 `app.UseMvc()` `app.UseMvcWithDefaultRoute()` .

Api에 대 한 액세스 요청 만들기

주문 및 바구니 마이크로 서비스를 요청 하는 경우 다음 코드 예제에 표시 된 것 처럼 인증 프로세스 중에 IdentityServer에서 얻은 액세스 토큰을 요청에 포함 해야 합니다.

```
var authToken = Settings.AuthAccessToken;
Order = await _ordersService.GetOrderAsync(Convert.ToInt32(order.OrderNumber), authToken);
```

액세스 토큰은 응용 프로그램 설정으로 저장 되며 플랫폼별 저장소에서 검색 되 고 클래스의 메서드에 대 한 호출에 포함 됩니다 `GetOrderAsync` `OrderService` .

마찬가지로, 다음 코드 예제와 같이 IdentityServer protected API에 데이터를 보낼 때 액세스 토큰을 포함 해야 합니다.

```
var authToken = Settings.AuthAccessToken;
await _basketService.UpdateBasketAsync(new CustomerBasket
{
    BuyerId = userInfo.UserId,
    Items = BasketItems.ToList()
}, authToken);
```

액세스 토큰은 플랫폼별 저장소에서 검색 되 고 클래스의 메서드에 대 한 호출에 포함 됩니다 `UpdateBasketAsync` `BasketService` .

`RequestProvider` EShopOnContainers 모바일 앱의 클래스는 클래스를 사용 하 여 `HttpClient` eShopOnContainers reference 응용 프로그램에서 노출 하는 RESTful api에 대 한 요청을 수행 합니다. 권한 부여가 필요한 주문 및 바구니 Api에 대 한 요청을 만들 때 유효한 액세스 토큰을 요청에 포함 해야 합니다. 이렇게 하려면 `HttpClient` 다음 코드 예제에서 보여 주는 것 처럼 인스턴스의 헤더에 액세스 토큰을 추가 합니다.

```
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);
```

`DefaultRequestHeaders` 클래스의 속성은 `HttpClient` 각 요청과 함께 전송 되는 헤더를 노출 하 고 액세스 토큰은 문자열이 접두사로 추가 된 헤더에 추가 됩니다 `Authorization` `Bearer` . 요청이 RESTful API로 전송 되 면 헤더의 값이 `Authorization` 추출 되 고 유효성을 검사 하 여 신뢰할 수 있는 발급자 로부터 전송 되 고 사용자에 게이를 수신 하는 api를 호출할 수 있는 권한이 있는지 여부를 확인 하는 데 사용 됩니다.

EShopOnContainers 모바일 앱이 웹 요청을 만드는 방법에 대 한 자세한 내용은 원격 데이터 액세스를 참조 하세요.

요약

Xamarin.FormsASP.NET MVC 웹 응용 프로그램과 통신 하는 앱에 인증 및 권한 부여를 통합 하는 방법에는 여러 가지가 있습니다. EShopOnContainers 모바일 앱은 IdentityServer 4를 사용 하는 컨테이너 화 된 identity 마이크로 서비스를 사용 하 여 인증 및 권한 부여를 수행 합니다. IdentityServer는 ASP.NET Core Id와 통합 되어 전달자 토큰 인증을 수행 하는 ASP.NET Core에 대 한 오픈 소스 Openid connect Connect 및 OAuth 2.0 프레임 워크입니다.

모바일 앱은 사용자를 인증 하거나 리소스에 액세스 하기 위해 IdentityServer에서 보안 토큰을 요청 합니다. 리소스에 액세스할 때 권한 부여를 필요로 하는 Api에 대 한 요청에 액세스 토큰을 포함 해야 합니다. IdentityServer의 미들웨어는 들어오는 액세스 토큰의 유효성을 검사 하 여 신뢰할 수 있는 발급자 로부터 보내고이를 수신 하는 API에서 사용할 수 있는지 확인 합니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)

원격 데이터에 액세스

2020-06-05 • 60 minutes to read • Edit Online

많은 최신 웹 기반 솔루션은 웹 서버에서 호스트 되는 웹 서비스를 사용 하 여 원격 클라이언트 응용 프로그램에 대 한 기능을 제공 합니다. 웹 API는 웹 서비스에 표시되는 작업으로 구성됩니다.

클라이언트 앱은 API에서 노출 하는 데이터 또는 작업이 어떻게 구현 되는지 몰라도 웹 API를 활용할 수 있어야 합니다. 이렇게 하려면 API가 클라이언트 앱 및 웹 서비스에서 사용할 데이터 형식 및 클라이언트 앱과 웹 서비스 간에 교환 되는 데이터의 구조에 동의할 수 있도록 하는 일반적인 표준에 따라 유럽 연합 합니다.

Representational State Transfer 소개

REST (Representational State Transfer)는 하이퍼 미디어를 기반으로 분산 시스템을 구축 하기 위한 아키텍처 스타일입니다. REST 모델의 주요 이점은 개방형 표준을 기반으로 하며, 모델 또는이를 액세스 하는 클라이언트 앱의 구현을 특정 구현에 바인딩하지 않는다는 것입니다. 따라서 Microsoft ASP.NET Core MVC를 사용 하 여 REST 웹 서비스를 구현할 수 있으며, 클라이언트 앱은 HTTP 요청을 생성 하 고 HTTP 응답을 구문 분석할 수 있는 모든 언어 및 도구 집합을 사용 하 여 개발할 수 있습니다.

REST 모델은 탐색 체계를 사용 하 여 네트워크를 통해 개체 및 서비스를 나타냅니다 (리소스 라고 함). REST를 구현 하는 시스템은 일반적으로 HTTP 프로토콜을 사용 하 여 이러한 리소스에 액세스 하기 위한 요청을 전송 합니다. 이러한 시스템에서 클라이언트 앱은 리소스를 식별 하는 URI 형식으로 요청을 제출 하 고 해당 리소스에 대해 수행할 작업을 나타내는 HTTP 메서드 (예: GET, POST, PUT 또는 DELETE)를 제출 합니다. HTTP 요청의 본문에는 작업을 수행 하는 데 필요한 데이터가 포함 되어 있습니다.

> **NOTE**
> REST는 상태 비저장 요청 모델을 정의 합니다. 따라서 HTTP 요청은 독립적 이어야 하 고 순서에 관계 없이 발생할 수 있습니다.

REST 요청의 응답은 표준 HTTP 상태 코드를 사용 합니다. 예를 들어 유효한 데이터를 반환하는 요청은 HTTP 응답 코드 200(정상)을 포함해야 하는 반면, 지정된 리소스를 찾거나 삭제하는 데 실패한 요청은 HTTP 상태 코드 404(찾을 수 없음)가 포함된 응답을 반환해야 합니다.

RESTful web API는 연결 된 리소스 집합을 노출 하 고, 앱이 해당 리소스를 조작 하 고 이러한 리소스를 쉽게 탐색할 수 있도록 하는 핵심 작업을 제공 합니다. 이러한 이유로 일반적인 RESTful web API를 구성 하는 Uri는 노출 되는 데이터를 기반으로 하며 HTTP에서 제공 하는 기능을 사용 하 여이 데이터에 대해 작동 합니다.

HTTP 요청에 클라이언트 앱에 포함 된 데이터와 웹 서버의 해당 응답 메시지는 미디어 유형 이라고 하는 다양 한 형식으로 표시 될 수 있습니다. 클라이언트 앱이 메시지 본문에 데이터를 반환 하는 요청을 보내면 요청 헤더에서 처리할 수 있는 미디어 형식을 지정할 수 있습니다 `Accept` . 웹 서버에서이 미디어 유형을 지 원하는 경우 `Content-Type` 메시지 본문에 있는 데이터의 형식을 지정 하는 헤더를 포함 하는 응답으로 회신할 수 있습니다. 클라이언트 앱이 응답 메시지를 구문 분석 하 고 결과를 적절 하 게 해석 해야 합니다.

REST에 대 한 자세한 내용은 api 디자인 및 api 구현을 참조 하세요.

RESTful Api 사용

EShopOnContainers 모바일 앱은 MVVM (모델-뷰-ViewModel) 패턴을 사용 하며, 패턴의 모델 요소는 앱에서 사용 되는 도메인 엔터티를 나타냅니다. EShopOnContainers reference 응용 프로그램의 컨트롤러 및 리포지토리 클래스는 이러한 여러 모델 개체를 허용 하 고 반환 합니다. 따라서 모바일 앱과 컨테이너 화 된 마이크로 서비스 간에 전달 되는 모든 데이터를 저장 하는 Dto (데이터 전송 개체)로 사용 됩니다. Dto를 사용 하 여 웹 서비스에 데이터

를 전달 하 고 데이터를 수신 하는 경우의 주요 혜택은 단일 원격 호출에서 더 많은 데이터를 전송 하 여 앱이 수행 해야 하는 원격 호출의 수를 줄일 수 있다는 것입니다.

웹 요청 수행

EShopOnContainers 모바일 앱은 클래스를 사용 하 여 `HttpClient` HTTP를 통해 요청을 수행 하며, JSON은 미디어 유형으로 사용 됩니다. 이 클래스는 URI로 식별 된 리소스에서 HTTP 요청을 비동기적으로 보내고 HTTP 응답을 받기 위한 기능을 제공 합니다. `HttpResponseMessage` 클래스는 http 요청이 수행 된 후 REST API에서 받은 http 응답 메시지를 나타냅니다. 상태 코드, 헤더 및 모든 본문을 포함 하 여 응답에 대 한 정보를 포함 합니다. 합니다 `HttpContent` 클래스를 나타내는 HTTP 본문 및 콘텐츠 헤더와 같은 `Content-Type` 고 `Content-Encoding` 입니다. `ReadAs` `ReadAsStringAsync` `ReadAsByteArrayAsync` 데이터의 형식에 따라 및와 같은 메서드를 사용 하 여 콘텐츠를 읽을 수 있습니다.

GET 요청 만들기

`CatalogService` 클래스는 카탈로그 마이크로 서비스에서 데이터 검색 프로세스를 관리 하는 데 사용 됩니다. `RegisterDependencies` 클래스의 메서드에서 `ViewModelLocator` `CatalogService` 클래스는 `ICatalogService` autofac 종속성 주입 컨테이너를 사용 하 여 형식에 대 한 형식 매핑으로 등록 됩니다. 그런 다음 클래스의 인스턴스를 `CatalogViewModel` 만들 때 해당 생성자는 `ICatalogService` Autofac가 확인 하 고 클래스의 인스턴스를 반환 하는 형식을 허용 `CatalogService` 합니다. 종속성 주입에 대 한 자세한 내용은 종속성 주입 소개를 참조 하세요.

그림 10-1에서는에서 표시 하기 위해 카탈로그 마이크로 서비스에서 카탈로그 데이터를 읽는 클래스의 상호 작용을 보여 줍니다 `CatalogView` .

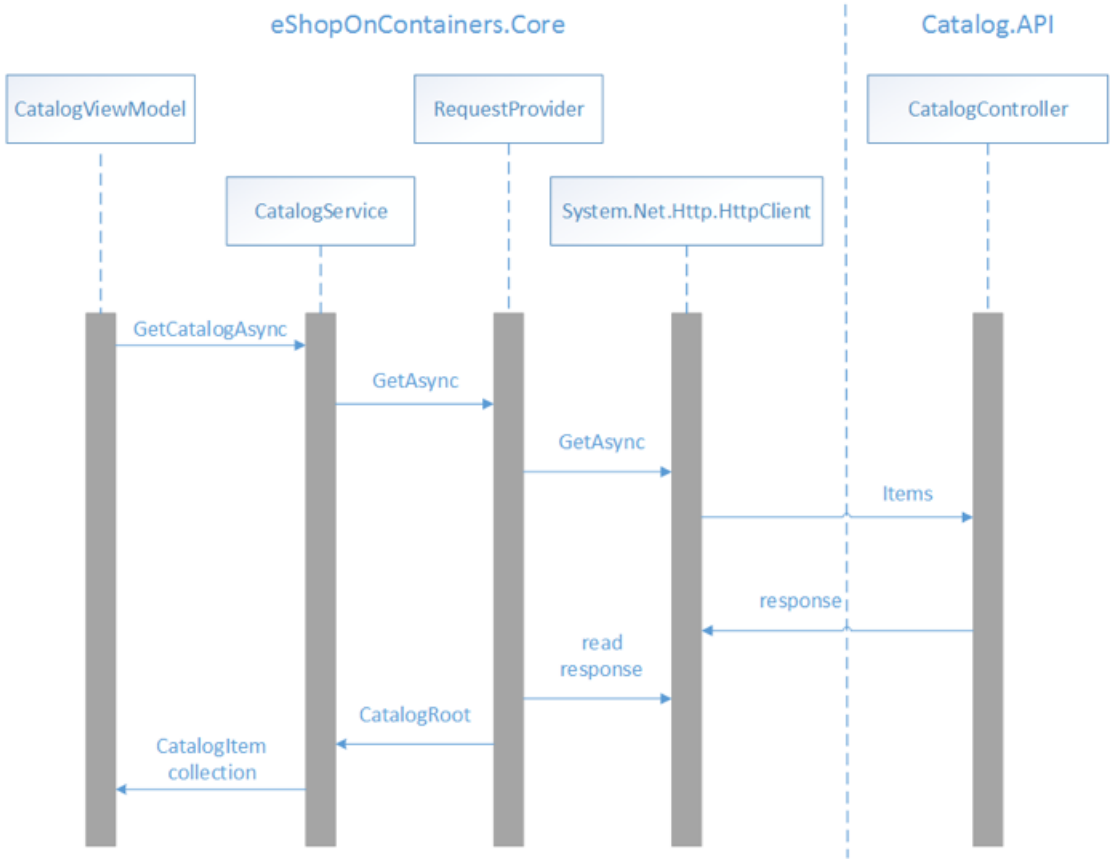

그림 10-1: 카탈로그 마이크로 서비스에서 데이터 검색

`CatalogView` 을 탐색 하면 `OnInitialize` 클래스의 메서드가 `CatalogViewModel` 호출 됩니다. 이 메서드는 다음 코드 예제에서 보여 주는 것 처럼 카탈로그 마이크로 서비스에서 카탈로그 데이터를 검색 합니다.

```
public override async Task InitializeAsync(object navigationData)
{
    ...
    Products = await _productsService.GetCatalogAsync();
    ...
}
```

이 메서드는 `GetCatalogAsync` `CatalogService` Autofac에 의해에 삽입 된 인스턴스의 메서드를 호출 `CatalogViewModel` 합니다. 다음 코드 예제는 `GetCatalogAsync` 메서드를 보여줍니다.

```
public async Task<ObservableCollection<CatalogItem>> GetCatalogAsync()
{
    UriBuilder builder = new UriBuilder(GlobalSetting.Instance.CatalogEndpoint);
    builder.Path = "api/v1/catalog/items";
    string uri = builder.ToString();

    CatalogRoot catalog = await _requestProvider.GetAsync<CatalogRoot>(uri);
    ...
    return catalog?.Data.ToObservableCollection();
}
```

이 메서드는 요청을 보낼 리소스를 식별 하는 URI를 빌드하고, 클래스를 사용 하 여 `RequestProvider` 리소스에서 GET HTTP 메서드를 호출한 다음 결과를로 반환 합니다 `CatalogViewModel` . 클래스에는 `RequestProvider` 리소스를 식별 하는 URI 형식, 해당 리소스에 대해 수행할 작업을 나타내는 HTTP 메서드 및 작업을 수행 하는 데 필요한 데이터를 포함 하는 본문으로 요청을 전송 하는 기능이 포함 되어 있습니다. 클래스를에 삽입 하는 방법에 대 한 자세한 내용은 `RequestProvider` `CatalogService class` 종속성 주입 소개를 참조 하세요.

다음 코드 예제에서는 `GetAsync` 클래스의 메서드를 보여 줍니다 `RequestProvider` .

```
public async Task<TResult> GetAsync<TResult>(string uri, string token = "")
{
    HttpClient httpClient = CreateHttpClient(token);
    HttpResponseMessage response = await httpClient.GetAsync(uri);

    await HandleResponse(response);
    string serialized = await response.Content.ReadAsStringAsync();

    TResult result = await Task.Run(() =>
        JsonConvert.DeserializeObject<TResult>(serialized, _serializerSettings));

    return result;
}
```

이 메서드는 `CreateHttpClient` `HttpClient` 적절 한 헤더 집합을 사용 하 여 클래스의 인스턴스를 반환 하는 메서드를 호출 합니다. 그런 다음 URI로 식별 되는 리소스에 비동기 GET 요청을 전송 하 고 응답을 인스턴스에 저장 합니다 `HttpResponseMessage` . `HandleResponse`그런 다음 응답이 성공 HTTP 상태 코드를 포함 하지 않는 경우 예외를 throw 하는 메서드를 호출 합니다. 그런 다음 응답을 문자열로 읽어 JSON에서 개체로 변환 하 `CatalogRoot` 고로 반환 합니다 `CatalogService` .

`CreateHttpClient`메서드는 다음 코드 예제에서 표시 됩니다.

```
private HttpClient CreateHttpClient(string token = "")
{
    var httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));

    if (!string.IsNullOrEmpty(token))
    {
        httpClient.DefaultRequestHeaders.Authorization =
            new AuthenticationHeaderValue("Bearer", token);
    }
    return httpClient;
}
```

이 메서드는 클래스의 새 인스턴스를 만들고 `HttpClient` `Accept` 인스턴스에서 만든 요청의 헤더를로 설정 합니다 `HttpClient` .이 `application/json` 경우 JSON을 사용 하 여 응답의 콘텐츠 형식을 지정 해야 함을 나타냅니다. 그런 다음 액세스 토큰이 메서드에 인수로 전달 된 경우 `CreateHttpClient` `Authorization` 인스턴스에 의해 생성 된 모든 요청의 헤더에 추가 되 고 문자열을 접두사로 추가 합니다 `HttpClient` `Bearer` . 권한 부여에 대 한 자세한 내용은 권한 부여를 참조 하세요.

`GetAsync` 클래스의 메서드가 `RequestProvider` 를 호출 하면 `HttpClient.GetAsync` `Items` `CatalogController` 다음 코드 예제에 표시 된 대로 Catalog. API 프로젝트의 클래스에 있는 메서드가 호출 됩니다.

```
[HttpGet]
[Route("[action]")]
public async Task<IActionResult> Items(
    [FromQuery]int pageSize = 10, [FromQuery]int pageIndex = 0)
{
    var totalItems = await _catalogContext.CatalogItems
        .LongCountAsync();

    var itemsOnPage = await _catalogContext.CatalogItems
        .OrderBy(c=>c.Name)
        .Skip(pageSize * pageIndex)
        .Take(pageSize)
        .ToListAsync();

    itemsOnPage = ComposePicUri(itemsOnPage);
    var model = new PaginatedItemsViewModel<CatalogItem>(
        pageIndex, pageSize, totalItems, itemsOnPage);

    return Ok(model);
}
```

이 메서드는 EntityFramework를 사용 하 여 SQL 데이터베이스에서 카탈로그 데이터를 검색 하 고, 성공 HTTP 상태 코드 및 JSON 형식 인스턴스의 컬렉션을 포함 하는 응답 메시지로 반환 합니다 `CatalogItem` .

POST 요청 만들기

클래스는 바구니 마이크로 서비스를 사용 하 여 `BasketService` 데이터 검색 및 업데이트 프로세스를 관리 하는 데 사용 됩니다. `RegisterDependencies` 클래스의 메서드에서 `ViewModelLocator` `BasketService` 클래스는 `IBasketService` autofac 종속성 주입 컨테이너를 사용 하 여 형식에 대 한 형식 매핑으로 등록 됩니다. 그런 다음 클래스의 인스턴스를 `BasketViewModel` 만들 때 해당 생성자는 `IBasketService` Autofac가 확인 하 고 클래스의 인스턴스를 반환 하는 형식을 허용 `BasketService` 합니다. 종속성 주입에 대 한 자세한 내용은 종속성 주입 소개를 참조 하세요.

그림 10-2에서는 바구니 마이크로 서비스에 게 표시 되는 바구니 데이터를 보내는 클래스의 상호 작용을 보여 줍니다 `BasketView` .

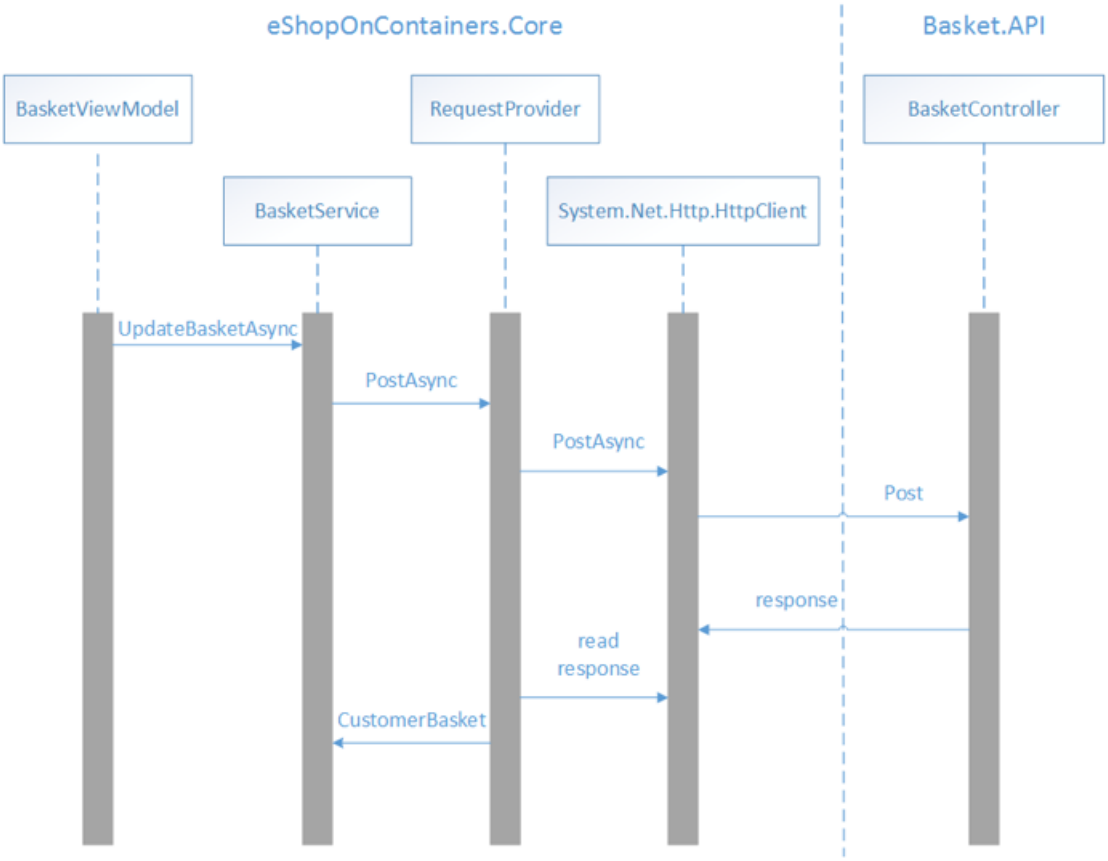

그림 10-2: 바구니에 데이터 보내기 마이크로 서비스

바구니에 항목이 추가 되 면 `ReCalculateTotalAsync` 클래스의 메서드가 `BasketViewModel` 호출 됩니다. 이 메서드는 바구니에 있는 항목의 total 값을 업데이트 하 고 다음 코드 예제와 같이 바구니 데이터를 바구니 마이크로 서비스 보냅니다.

```
private async Task ReCalculateTotalAsync()
{
    ...
    await _basketService.UpdateBasketAsync(new CustomerBasket
    {
        BuyerId = userInfo.UserId,
        Items = BasketItems.ToList()
    }, authToken);
}
```

이 메서드는 `UpdateBasketAsync` `BasketService` Autofac에 의해에 삽입 된 인스턴스의 메서드를 호출 `BasketViewModel` 합니다. 다음 메서드는 메서드를 보여 줍니다 `UpdateBasketAsync` .

```
public async Task<CustomerBasket> UpdateBasketAsync(CustomerBasket customerBasket, string token)
{
    UriBuilder builder = new UriBuilder(GlobalSetting.Instance.BasketEndpoint);
    string uri = builder.ToString();
    var result = await _requestProvider.PostAsync(uri, customerBasket, token);
    return result;
}
```

이 메서드는 요청을 보낼 리소스를 식별 하는 URI를 빌드하고, 클래스를 사용 하 여 `RequestProvider` 리소스에서 POST HTTP 메서드를 호출한 다음 결과를로 반환 합니다 `BasketViewModel` . 인증 프로세스 중에 IdentityServer에서 얻은 액세스 토큰은 바구니 마이크로 서비스 요청에 권한을 부여 하는 데 필요 합니다. 권한 부여에 대 한 자세한 내용은 권한 부여를 참조 하세요.

다음 코드 예제에서는 클래스의 메서드 중 하나를 보여 줍니다 `PostAsync` `RequestProvider` .

```
public async Task<TResult> PostAsync<TResult>(
    string uri, TResult data, string token = "", string header = "")
{
    HttpClient httpClient = CreateHttpClient(token);
    ...
    var content = new StringContent(JsonConvert.SerializeObject(data));
    content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
    HttpResponseMessage response = await httpClient.PostAsync(uri, content);

    await HandleResponse(response);
    string serialized = await response.Content.ReadAsStringAsync();

    TResult result = await Task.Run(() =>
        JsonConvert.DeserializeObject<TResult>(serialized, _serializerSettings));

    return result;
}
```

이 메서드는 `CreateHttpClient` `HttpClient` 적절 한 헤더 집합을 사용 하 여 클래스의 인스턴스를 반환 하는 메서드를 호출 합니다. 그런 다음 URI에 의해 식별 되는 리소스에 비동기 POST 요청을 전송 하 고, serialize 된 바구니 데이터를 JSON 형식으로 보내고, 응답을 인스턴스에 저장 합니다 `HttpResponseMessage` . `HandleResponse` 그런 다음 응답이 성공 HTTP 상태 코드를 포함 하지 않는 경우 예외를 throw 하는 메서드를 호출 합니다. 그런 다음 응답을 문자열로 읽어 JSON에서 개체로 변환 하 `CustomerBasket` 고로 반환 합니다 `BasketService` . 메서드에 대 한 자세한 내용은 `CreateHttpClient` GET 요청 만들기를 참조 하세요.

`PostAsync` 클래스의 메서드가 `RequestProvider` 를 호출 하는 경우, `HttpClient.PostAsync` `Post` `BasketController` 다음 코드 예제와 같이 바구니 프로젝트의 클래스에서 메서드가 호출 됩니다.

```
[HttpPost]
public async Task<IActionResult> Post([FromBody]CustomerBasket value)
{
    var basket = await _repository.UpdateBasketAsync(value);
    return Ok(basket);
}
```

이 메서드는 클래스의 인스턴스를 사용 하 여 `RedisBasketRepository` 바구니 데이터를 Redis cache에 보관 하 고, 성공 HTTP 상태 코드 및 JSON 형식의 인스턴스가 포함 된 응답 메시지로 반환 합니다 `CustomerBasket` .

삭제 요청 만들기

그림 10-3에서는 바구니 마이크로 서비스 바구니 데이터를 삭제 하는 클래스의 상호 작용을 보여 줍니다 `CheckoutView` .

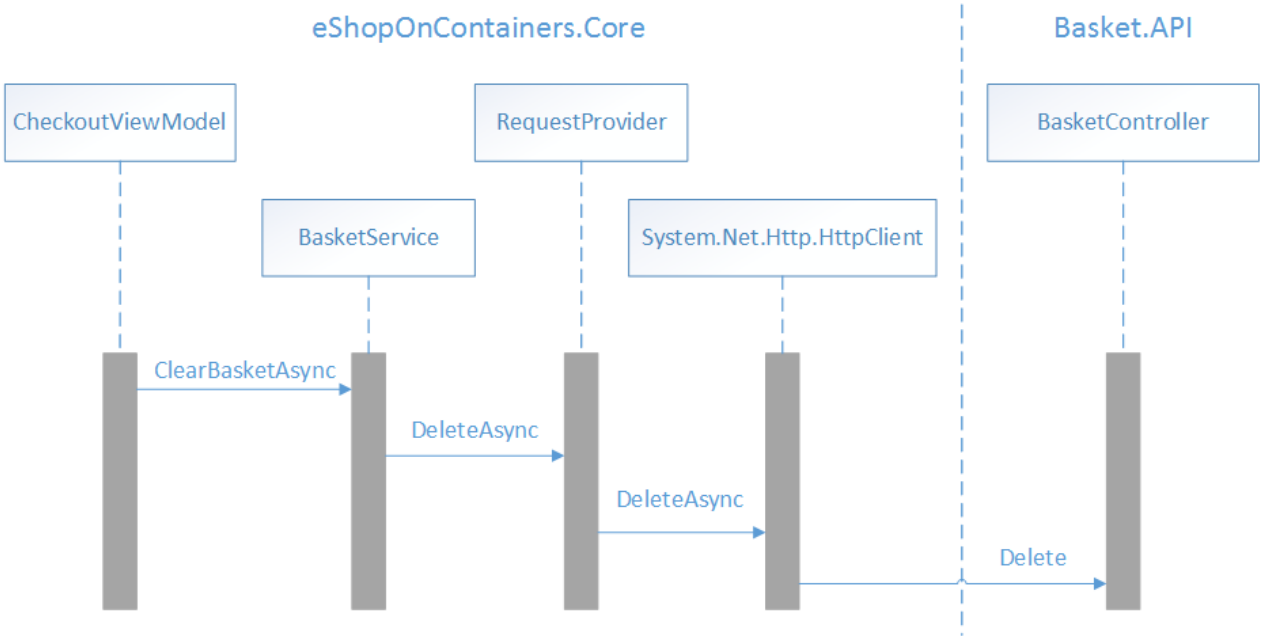

그림 10-3: 바구니 마이크로 서비스에서 데이터 삭제

체크 아웃 프로세스를 호출 하는 경우 `CheckoutAsync` 클래스의 메서드가 `CheckoutViewModel` 호출 됩니다. 이 메서드는 다음 코드 예제에서 보여 주는 것 처럼 장바구니를 지우기 전에 새 주문을 만듭니다.

```
private async Task CheckoutAsync()
{
    ...
    await _basketService.ClearBasketAsync(_shippingAddress.Id.ToString(), authToken);
    ...
}
```

이 메서드는 `ClearBasketAsync` `BasketService` Autofac에 의해에 삽입 된 인스턴스의 메서드를 호출 `CheckoutViewModel` 합니다. 다음 메서드는 메서드를 보여 줍니다 `ClearBasketAsync` .

```
public async Task ClearBasketAsync(string guidUser, string token)
{
    UriBuilder builder = new UriBuilder(GlobalSetting.Instance.BasketEndpoint);
    builder.Path = guidUser;
    string uri = builder.ToString();
    await _requestProvider.DeleteAsync(uri, token);
}
```

이 메서드는 요청을 보낼 리소스를 식별 하는 URI를 빌드하고 클래스를 사용 하 여 `RequestProvider` 리소스에 대 한 DELETE HTTP 메서드를 호출 합니다. 인증 프로세스 중에 IdentityServer에서 얻은 액세스 토큰은 바구니 마이크로 서비스 요청에 권한을 부여 하는 데 필요 합니다. 권한 부여에 대 한 자세한 내용은 권한 부여를 참조 하세요.

다음 코드 예제에서는 `DeleteAsync` 클래스의 메서드를 보여 줍니다 `RequestProvider` .

```
public async Task DeleteAsync(string uri, string token = "")
{
    HttpClient httpClient = CreateHttpClient(token);
    await httpClient.DeleteAsync(uri);
}
```

이 메서드는 `CreateHttpClient` `HttpClient` 적절 한 헤더 집합을 사용 하 여 클래스의 인스턴스를 반환 하는 메서드를 호출 합니다. 그런 다음 URI로 식별 되는 리소스에 비동기 삭제 요청을 제출 합니다. 메서드에 대 한 자세한 내용은 `CreateHttpClient` GET 요청 만들기를 참조 하세요.

`DeleteAsync` 클래스의 메서드가 `RequestProvider` 를 호출 하는 경우, `HttpClient.DeleteAsync` `Delete` `BasketController` 다음 코드 예제와 같이 바구니 프로젝트의 클래스에서 메서드가 호출 됩니다.

```
[HttpDelete("{id}")]
public void Delete(string id)
{
    _repository.DeleteBasketAsync(id);
}
```

이 메서드는 클래스의 인스턴스를 사용 하 여 `RedisBasketRepository` Redis cache에서 바구니 데이터를 삭제 합니다.

데이터 캐싱

자주 액세스 하는 데이터를 앱에 가까이 있는 빠른 저장소에 캐싱하여 앱 성능을 향상 시킬 수 있습니다. 빠른 저장소가 원래 원본 보다 더 가까운 앱에 있는 경우 캐싱은 데이터를 검색할 때 응답 시간을 크게 향상 시킬 수 있습니다.

가장 일반적인 형태의 캐싱은 캐시를 참조 하 여 앱에서 데이터를 검색 하는 읽기-캐싱입니다. 캐시에 데이터가 없으면 데이터 저장소에서 데이터가 검색된 후 캐시에 추가됩니다. 앱은 캐시 배제 패턴을 사용 하 여 읽기-읽기 캐싱을 구현할 수 있습니다. 이 패턴은 항목이 현재 캐시에 있는지 여부를 확인 합니다. 항목이 캐시에 없으면 데이터 저장소에서 읽어서 캐시에 추가 됩니다. 자세한 내용은 캐시 배제 패턴을 참조 하세요.

> **TIP**
> 자주 읽고 자주 변경 되지 않는 데이터를 캐시 합니다. 이 데이터는 앱에서 처음 검색할 때 필요할 때 캐시에 추가할 수 있습니다. 즉, 응용 프로그램은 데이터 저장소에서 한 번만 데이터를 인출 해야 하며, 그 후에는 캐시를 사용 하 여 후속 액세스를 만족할 수 있습니다.

EShopOnContainers reference 응용 프로그램과 같은 분산 응용 프로그램은 다음 캐시 중 하나 또는 둘 모두를 제공 해야 합니다.

- 여러 프로세스 또는 컴퓨터에서 액세스할 수 있는 공유 캐시.
- 응용 프로그램을 실행 하는 장치에 로컬로 데이터가 저장 되는 개인 캐시.

EShopOnContainers 모바일 앱은 앱의 인스턴스를 실행 하는 장치에 로컬로 데이터가 보관 되는 개인 캐시를 사용합니다. EShopOnContainers reference 응용 프로그램에서 사용 하는 캐시에 대 한 자세한 내용은 .Net 마이크로서비스: 컨테이너 화 된 .Net 응용 프로그램 아키텍처를 참조 하세요.

> **TIP**
> 캐시는 언제 든 지 사라질 수 있는 임시 데이터 저장소로 생각할 수 있습니다. 데이터가 원래 데이터 저장소 뿐만 아니라 캐시에 유지 되는지 확인 합니다. 캐시를 사용할 수 없게 되 면 데이터가 손실 될 가능성이 최소화 됩니다.

데이터 만료 관리

캐시 된 데이터가 항상 원래 데이터와 일치 하는 것은 바람직하지 않습니다. 원래 데이터 저장소의 데이터는 캐시 된 후 변경 되어 캐시 된 데이터가 부실 해질 수 있습니다. 따라서 앱은 캐시의 데이터가 최대한 최신 상태 인지 확인 하는 데 도움이 되는 전략을 구현 해야 하지만 캐시의 데이터가 오래 된 경우 발생 하는 상황을 감지 하 고 처리 할 수도 있습니다. 대부분의 캐싱 메커니즘을 사용 하면 데이터를 만료 하도록 캐시를 구성할 수 있으므로 데이터를 최신으로 유지 하는 기간을 줄일 수 있습니다.

TIP

캐시를 구성할 때 기본 만료 시간을 설정 합니다. 많은 캐시가 만료를 구현 하며,이는 데이터를 무효화 하 고 지정 된 기간 동안 액세스 하지 않은 경우 캐시에서 제거 합니다. 그러나 만료 기간을 선택할 때는 주의 해야 합니다. 너무 짧으면 데이터가 너무 빨리 만료 되 고 캐싱의 이점이 줄어듭니다. 너무 오래 된 경우 데이터 위험이 부실 합니다. 따라서 만료 시간은 데이터를 사용 하는 앱에 대 한 액세스 패턴과 일치 해야 합니다.

캐시 된 데이터가 만료 되 면 캐시에서 제거 해야 하며, 앱은 원래 데이터 저장소에서 데이터를 검색 하 여 캐시에 다시 저장 해야 합니다.

데이터가 너무 오래 지속 될 수 있는 경우 캐시가 채워질 수 있습니다. 따라서 제거 라고 하는 프로세스에서 일부 항목을 제거 하려면 캐시에 새 항목을 추가 하는 요청이 필요할 수 *있습니다.* 캐싱 서비스는 일반적으로 가장 최근에 사용 된 방식으로 데이터를 제거 합니다. 그러나 가장 최근에 사용한 것을 비롯 한 다른 제거 정책 및 선입 first (선입 out)가 있습니다. 자세한 내용은 캐싱 지침을 참조 하세요.

이미지 캐싱

EShopOnContainers 모바일 앱은 캐시를 활용 하는 원격 제품 이미지를 사용 합니다. 이러한 이미지는 `Image` 컨트롤 및 `CachedImage` FFImageLoading 라이브러리에서 제공 하는 컨트롤에 의해 표시 됩니다.

Xamarin.Forms `Image` 컨트롤은 다운로드 된 이미지의 캐싱을 지원 합니다. 캐싱은 기본적으로 사용 하도록 설정 되며 24 시간 동안 로컬로 이미지를 저장 합니다. 또한 속성을 사용 하 여 만료 시간을 구성할 수 있습니다 `CacheValidity` . 자세한 내용은 다운로드 한 이미지 캐싱을 참조 하세요.

FFImageLoading의 `CachedImage` 컨트롤은 컨트롤을 대체 하 여 Xamarin.Forms `Image` 보충 기능을 사용할 수 있는 추가 속성을 제공 합니다. 이 기능 중에서 컨트롤은 오류를 지원 하 고 이미지 자리 표시자를 로드 하는 동안 구성 가능한 캐싱을 제공 합니다. 다음 코드 예제에서는 eShopOnContainers 모바일 앱이의 컨트롤에서 사용 하는 데이터 템플릿인의 컨트롤을 사용 하는 방법을 보여 줍니다 `CachedImage` `ProductTemplate` `ListView` `CatalogView` .

```
<ffimageloading:CachedImage
    Grid.Row="0"
    Source="{Binding PictureUri}"
    Aspect="AspectFill">
    <ffimageloading:CachedImage.LoadingPlaceholder>
        <OnPlatform x:TypeArguments="ImageSource">
            <On Platform="iOS, Android" Value="default_campaign" />
            <On Platform="UWP" Value="Assets/default_campaign.png" />
        </OnPlatform>
    </ffimageloading:CachedImage.LoadingPlaceholder>
    <ffimageloading:CachedImage.ErrorPlaceholder>
        <OnPlatform x:TypeArguments="ImageSource">
            <On Platform="iOS, Android" Value="noimage" />
            <On Platform="UWP" Value="Assets/noimage.png" />
        </OnPlatform>
    </ffimageloading:CachedImage.ErrorPlaceholder>
</ffimageloading:CachedImage>
```

`CachedImage` 컨트롤은 `LoadingPlaceholder` 및 속성을 `ErrorPlaceholder` 플랫폼별 이미지로 설정 합니다. 속성은 속성 `LoadingPlaceholder` 으로 지정 된 이미지를 검색 하는 동안 표시할 이미지를 지정 하 고 속성은 `Source` `ErrorPlaceholder` 속성으로 지정 된 이미지를 검색 하려고 할 때 오류가 발생 하는 경우 표시할 이미지를 지정 합니다 `Source` .

이름에서 알 때 컨트롤은 `CachedImage` 속성 값에 지정 된 시간 동안 장치의 원격 이미지를 캐시 합니다 `CacheDuration` . 이 속성 값을 명시적으로 설정 하지 않으면 30 일의 기본값이 적용 됩니다.

복원 력 증대

원격 서비스 및 리소스와 통신 하는 모든 앱은 일시적인 오류에 중요 해야 합니다. 일시적인 오류에는 서비스에 대 한 네트워크 연결이 일시적으로 손실 되거나, 서비스를 일시적으로 사용할 수 없거나, 서비스가 사용 중일 때 발생 하는 시간 초과가 포함 됩니다. 이러한 오류는 자동으로 수정 되는 경우가 많으며, 적절 한 지연 후 작업이 반복 되 면 성공할 가능성이 높습니다.

일시적인 오류는 모든 예측 가능한 상황에서 철저 하 게 테스트 된 경우에도 앱의 인식 품질에 큰 영향을 줄 수 있습니다. 원격 서비스와 통신 하는 앱이 안정적으로 작동 하도록 하려면 다음을 모두 수행할 수 있어야 합니다.

- 오류가 발생 하는 경우 오류를 검색 하 고 오류가 일시적이 될 가능성이 있는지 확인 합니다.
- 오류가 일시적인 것일 가능성이 있는 것으로 확인 되 면 작업을 다시 시도 하 고 작업을 다시 시도한 횟수를 추적 합니다.
- 재시도 횟수, 각 시도 사이의 지연 시간 및 시도 실패 후 수행할 작업을 지정 하는 적절 한 재시도 전략을 사용 합니다.

이 일시적인 오류 처리는 재시도 패턴을 구현 하는 코드에서 원격 서비스에 액세스 하려는 모든 시도를 래핑하여 구현할 수 있습니다.

재시도 패턴

응용 프로그램이 원격 서비스에 요청을 보내려고 할 때 오류를 검색 하는 경우 다음과 같은 방법으로 오류를 처리할 수 있습니다.

- 작업을 다시 시도 하 고 있습니다. 앱에서 실패 한 요청을 즉시 다시 시도할 수 있습니다.
- 지연 후 작업을 다시 시도 하는 중입니다. 앱은 요청을 다시 시도 하기 전에 적절 한 시간 동안 대기 해야 합니다.
- 작업을 취소 하 고 있습니다. 응용 프로그램에서 작업을 취소 하 고 예외를 보고 해야 합니다.

앱의 비즈니스 요구 사항에 맞게 다시 시도 전략을 조정 해야 합니다. 예를 들어 재시도 횟수 및 다시 시도 간격을 시도 중인 작업으로 최적화 하는 것이 중요 합니다. 작업이 사용자 상호 작용의 일부인 경우 재시도 간격은 짧고, 사용자가 응답을 기다릴 수 없도록 하기 위해 몇 번의 재시도만 시도 해야 합니다. 작업이 장기 실행 워크플로의 일부인 경우 워크플로를 취소 하거나 다시 시작 하는 데 비용이 많이 들고 시간이 오래 걸리는 경우에는 시도 사이에 대기 하 고 더 많은 시간을 다시 시도 하는 것이 적절 합니다.

> **NOTE**
> 시도 간 지연 시간을 최소화 하 고 재시도 횟수를 최소화 하는 적극적인 재시도 전략은 거의 또는 대량으로 실행 되는 원격 서비스의 성능을 저하 시킬 수 있습니다. 또한 이러한 다시 시도 전략은 계속 해 서 실패 한 작업을 수행 하려고 하는 경우 앱의 응답성에 영향을 줄 수 있습니다.

한 번의 재시도 후에도 요청이 계속 실패 하는 경우 앱에서 더 많은 요청을 동일한 리소스로 이동 하 여 오류를 보고 하는 것이 더 좋습니다. 그런 다음 설정 된 기간이 지나면 응용 프로그램은 리소스에 대해 하나 이상의 요청을 만들어 성공 했는지 확인할 수 있습니다. 자세한 내용은 회로 차단기 패턴을 참조하세요.

> **TIP**
> 무한 재시도 메커니즘을 구현하지 않습니다. 유한 횟수의 재시도를 사용 하거나 회로 차단기 패턴을 구현 하 여 서비스를 복구할 수 있도록 합니다.

EShopOnContainers 모바일 앱은 RESTful 웹 요청을 만들 때 현재 재시도 패턴을 구현 하지 않습니다. 그러나 FFImageLoading 라이브러리에서 제공하는 컨트롤 `CachedImage` 은 이미지 로드를 다시 시도하여 일시적인 오류 처리를 지원합니다. 이미지 로드에 실패 하는 경우 추가 시도가 수행 됩니다. 시도 횟수는 속성으로 지정 되며 `RetryCount` , 속성에 의해 지정 된 지연 이후에 다시 시도 됩니다 `RetryDelay` . 이러한 속성 값을 명시적으로 설정 하지 않으면 속성에 대해 3 개의 기본값이 적용 되 `RetryCount` 고 속성에는 250ms 적용 됩니다 `RetryDelay` . 컨트롤에 대 한 자세한 내용은 `CachedImage` 이미지 캐싱을 참조 하세요.

EShopOnContainers reference 응용 프로그램은 재시도 패턴을 구현 합니다. 재시도 패턴을 클래스와 결합 하는 방법에 대 한 자세한 내용은 `HttpClient` .Net 마이크로 서비스: 컨테이너 화 된 .Net 응용 프로그램 아키텍처를 참조 하세요.

다시 시도 패턴에 대 한 자세한 내용은 재시도 패턴을 참조 하세요.

회로 차단기 패턴

경우에 따라 수정 하는 데 시간이 오래 걸리는 예상 이벤트로 인해 오류가 발생할 수 있습니다. 이러한 오류는 전체 서비스 오류에 대 한 일부 연결 손실과의 범위를 지정할 수 있습니다. 이러한 상황에서는 응용 프로그램에서 성공할 가능성이 없는 작업을 다시 시도 하는 것은 무의미 하지 않으며, 대신 작업이 실패 한 것을 허용 하 고 그에 따라이 오류를 처리 해야 합니다.

회로 차단기 패턴은 응용 프로그램이 실패할 가능성이 있는 작업을 반복적으로 실행 하는 것을 방지할 수 있으며, 앱에서 오류가 해결 되었는지 여부도 검색할 수 있도록 합니다.

> **NOTE**
> 회로 차단기 패턴의 목적은 재시도 패턴과 다릅니다. 재시도 패턴을 사용 하면 응용 프로그램이 성공 하다 고 가정 하에 작업을 다시 시도할 수 있습니다. 회로 차단기 패턴은 응용 프로그램이 실패할 가능성이 있는 작업을 수행할 수 없도록 합니다.

회로 차단기는 실패할 수 있는 작업에서 프록시 역할을 합니다. 프록시는 발생 한 최근 오류 수를 모니터링 하 고, 이 정보를 사용 하 여 작업을 계속 진행할 수 있는지 여부를 결정 하거나, 예외를 즉시 반환할 것인지 여부를 결정 합니다.

EShopOnContainers 모바일 앱은 현재 회로 차단기 패턴을 구현 하지 않습니다. 그러나 eShopOnContainers는 그렇지 않습니다. 자세한 내용은 .Net 마이크로 서비스: 컨테이너 화 된 .Net 응용 프로그램 아키텍처를 참조 하세요.

> **TIP**
> 재시도 및 회로 차단기 패턴을 결합 합니다. 앱은 재시도 패턴을 사용 하 여 회로 차단기를 통해 작업을 호출 하 여 재시도 및 회로 차단기 패턴을 결합할 수 있습니다. 그러나 재시도 논리는 회로 차단기에서 반환되는 모든 예외에 민감하며, 회로 차단기에서 오류가 일시적임을 나타내는 경우 재시도를 중단합니다.

회로 차단기 패턴에 대 한 자세한 내용은 회로 차단기 패턴을 참조 하세요.

요약

많은 최신 웹 기반 솔루션은 웹 서버에서 호스트 되는 웹 서비스를 사용 하 여 원격 클라이언트 응용 프로그램에 대 한 기능을 제공 합니다. 웹 서비스에서 노출 하는 작업은 web API를 구성 하 고, 클라이언트 앱은 API에서 노출 하는 데이터 또는 작업이 구현 되는 방식을 몰라도 웹 API를 활용할 수 있어야 합니다.

자주 액세스 하는 데이터를 앱에 가까이 있는 빠른 저장소에 캐싱하여 앱 성능을 향상 시킬 수 있습니다. 앱은 캐시 배제 패턴을 사용 하 여 읽기-읽기 캐싱을 구현할 수 있습니다. 이 패턴은 항목이 현재 캐시에 있는지 여부를 확인 합니다. 항목이 캐시에 없으면 데이터 저장소에서 읽어서 캐시에 추가 됩니다.

웹 Api와 통신할 때 앱은 일시적인 오류를 인식 해야 합니다. 일시적인 오류에는 서비스에 대 한 네트워크 연결이 일시적으로 손실 되거나, 서비스를 일시적으로 사용할 수 없거나, 서비스가 사용 중일 때 발생 하는 시간 초과가 포함 됩니다. 이러한 오류는 자동으로 수정 되는 경우가 많으며, 적절 한 지연 후에 작업이 반복 되 면 성공할 가능성이 높습니다. 따라서 응용 프로그램은 일시적인 오류 처리 메커니즘을 구현 하는 코드에서 web API에 액세스 하려는 모든 시도를 래핑합니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)

- eShopOnContainers (GitHub) (샘플)

엔터프라이즈 앱 유닛 테스트

2020-06-05 • 24 minutes to read • Edit Online

모바일 앱에는 데스크톱 및 웹 기반 응용 프로그램에서 걱정 하지 않아도 되는 고유한 문제가 있습니다. 모바일 사용자는 사용 하는 장치, 네트워크 연결, 서비스 가용성, 기타 다양 한 요인에 따라 다릅니다. 따라서 모바일 앱은 품질, 안정성 및 성능을 개선 하기 위해 실제 세계에서 사용 되는 것으로 테스트 해야 합니다. 단위 테스트, 통합 테스트, 사용자 인터페이스 테스트를 비롯 하 여 앱에서 수행 해야 하는 다양 한 테스트 유형과 단위 테스트를 가장 일반적인 테스트 형식으로 사용할 수 있습니다.

단위 테스트는 응용 프로그램의 작은 단위 (일반적으로 메서드)를 사용 하 여 코드의 나머지 부분에서 분리 하 고 예상 대로 작동 하는지 확인 합니다. 이러한 목표는 각 기능 단위가 예상 대로 수행 되는지 확인 하 여 응용 프로그램 전체에서 오류가 전파 되지 않도록 하는 것입니다. 문제가 발생 하는 버그를 검색 하는 것은 보조 실패 지점에서 간접적으로 버그의 효과를 관찰 하는 것이 더 효율적입니다.

유닛 테스트는 소프트웨어 개발 워크플로의 필수적인 부분인 코드 품질에 가장 큰 영향을 미칠 수 있습니다. 메서드를 작성 하는 즉시 표준, 경계, 잘못 된 입력 데이터 사례에 대 한 응답으로 메서드의 동작을 확인 하는 단위 테스트를 작성 하 고 코드에 의해 수행 된 모든 명시적 또는 암시적 가정을 확인 하는 단위 테스트를 작성 해야 합니다. 또는 테스트 기반 개발을 사용 하 여 단위 테스트를 코드 보다 먼저 작성 합니다. 이 시나리오에서 단위 테스트는 디자인 설명서와 기능 사양으로 작동 합니다.

> **NOTE**
> 단위 테스트는 작동 하는 데 사용 되지만 잘못 된 업데이트로 방해 받은 기능에 대해 매우 효과적입니다.

단위 테스트는 일반적으로 정렬-동작 어설션 패턴을 사용 합니다.

- 단위 테스트 메서드의 *정렬* 섹션은 개체를 초기화 하 고 테스트 중인 메서드에 전달 되는 데이터의 값을 설정 합니다.
- *Act* 섹션은 필요한 인수를 사용 하 여 테스트 중인 메서드를 호출 합니다.
- *Assert* 섹션은 테스트 중인 메서드의 작업이 예상 대로 작동 하는지 확인 합니다.

이 패턴을 따라 단위 테스트를 읽고 일관성 있게 유지할 수 있습니다.

종속성 주입 및 유닛 테스트

느슨하게 결합 된 아키텍처를 채택 하는 동기 중 하나는 유닛 테스트를 용이 하 게 하는 것입니다. Autofac에 등록 된 유형 중 하나는 `OrderService` 클래스입니다. 다음 코드 예제에서는이 클래스의 개요를 보여 줍니다.

```
public class OrderDetailViewModel : ViewModelBase
{
    private IOrderService _ordersService;

    public OrderDetailViewModel(IOrderService ordersService)
    {
        _ordersService = ordersService;
    }
    ...
}
```

`OrderDetailViewModel` 클래스는 `IOrderService` 개체를 인스턴스화할 때 컨테이너가 확인 하는 형식에 대 한 종속성을 갖습니다 `OrderDetailViewModel` . 그러나 `OrderService` 클래스를 단위 테스트 하는 개체를 만드는 대신

`OrderDetailViewModel` `OrderService` 테스트 목적을 위한 모의 개체로 개체를 대체 합니다. 그림 10-1에서는이 관계를 보여 줍니다.

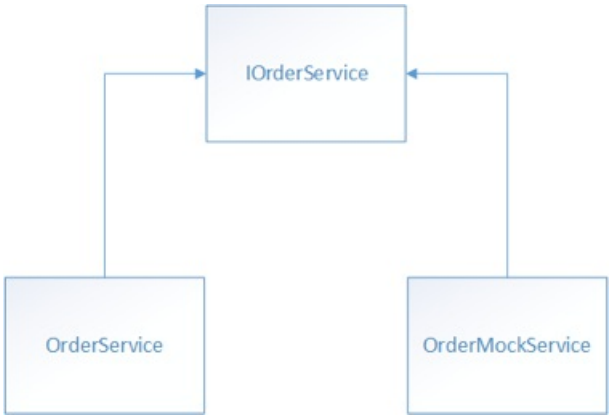

그림 10-1: IOrderService 인터페이스를 구현 하는 클래스

이 방법을 사용 하면 `OrderService` 런타임에 개체를 클래스로 전달할 수 `OrderDetailViewModel` 있으며, 테스트 용이성의 관심사에서 테스트 시간에 클래스를 `OrderMockService` 클래스로 전달할 수 있습니다 `OrderDetailViewModel` . 이 방법의 주요 이점은 웹 서비스, 데이터베이스 등의 더 많은 리소스를 요구 하지 않고 단위 테스트를 실행할 수 있다는 것입니다.

MVVM 응용 프로그램 테스트

MVVM 응용 프로그램에서 모델을 테스트 하 고 모델을 확인 하는 것은 다른 클래스를 테스트 하는 것과 동일 하며, 단위 테스트 및 mock와 같은 도구와 기법을 사용할 수 있습니다. 그러나 모델 클래스를 모델링 하 고 보는 데 일반적으로 사용할 수 있는 몇 가지 패턴이 있습니다. 이러한 패턴은 특정 유닛 테스트 기법을 활용 합니다.

> **TIP**
> 각 단위 테스트를 사용 하 여 하나의 항목을 테스트 합니다. 단위 테스트를 실행 하는 데는 단위 동작의 두 가지 이상의 측면이 필요 하지 않습니다. 이렇게 하면 읽고 업데이트 하기 어려운 테스트가 진행 됩니다. 오류를 해석할 때 혼동을 일으킬 수도 있습니다.

EShopOnContainers 모바일 앱은 Xunit 을 사용 하 여 두 가지 유형의 단위 테스트를 지 원하는 유닛 테스트를 수행 합니다.

- 팩트는 항상 true 인 테스트로, 고정 조건을 테스트 합니다.
- 이론는 특정 데이터 집합에만 적용 되는 테스트입니다.

EShopOnContainers 모바일 앱에 포함 된 단위 테스트는 팩트 테스트 이므로 각 단위 테스트 메서드는 특성으로 데코 레이트 됩니다 `[Fact]` .

> **NOTE**
> xUnit 테스트는 test runner에 의해 실행 됩니다. Test runner를 실행 하려면 필요한 플랫폼에 대 한 eShopOnContainers. Testrunner.completed 프로젝트를 실행 합니다.

비동기 기능 테스트

MVVM 패턴을 구현 하는 경우 뷰 모델은 일반적으로 서비스에 대 한 작업을 비동기적으로 호출 합니다. 이러한 작업을 호출 하는 코드에 대 한 테스트는 일반적으로 mock를 실제 서비스의 대체 항목으로 사용 합니다. 다음 코드 예제에서는 뷰 모델에 모의 서비스를 전달 하 여 비동기 기능을 테스트 하는 방법을 보여 줍니다.

```
[Fact]
public async Task OrderPropertyIsNotNullAfterViewModelInitializationTest()
{
    var orderService = new OrderMockService();
    var orderViewModel = new OrderDetailViewModel(orderService);

    var order = await orderService.GetOrderAsync(1, GlobalSetting.Instance.AuthToken);
    await orderViewModel.InitializeAsync(order);

    Assert.NotNull(orderViewModel.Order);
}
```

이 단위 테스트는 `Order` `OrderDetailViewModel` 메서드를 호출한 후 인스턴스의 속성에 값이 있는지 확인 `InitializeAsync` 합니다. `InitializeAsync`뷰 모델의 해당 뷰를 탐색 하면 메서드가 호출 됩니다. 탐색에 대 한 자세한 내용은 탐색을 참조 하세요.

인스턴스를 `OrderDetailViewModel` 만들 때 `OrderService` 인스턴스는 인수로 지정 될 것으로 예상 됩니다. 그러나는 `OrderService` 웹 서비스에서 데이터를 검색 합니다. 따라서 `OrderMockService` 클래스의 모의 버전인 인스턴스는 `OrderService` 생성자에 대 한 인수로 지정 됩니다 `OrderDetailViewModel` . 그런 다음 작업을 호출 하는 뷰 모델의 `InitializeAsync` 메서드가 호출 되 면 `IOrderService` 웹 서비스와 통신 하는 대신 모의 데이터가 검색 됩니다.

INotifyPropertyChanged 구현 테스트

인터페이스를 구현 `INotifyPropertyChanged` 하면 뷰가 뷰 모델 및 모델에서 발생 하는 변경 내용에 반응할 수 있습니다. 이러한 변경 내용은 컨트롤에 표시 되는 데이터로 제한 되지 않습니다. 또한 애니메이션을 시작 하거나 컨트롤을 사용 하지 않도록 설정 하는 모델 상태 보기와 같이 뷰를 제어 하는 데 사용 됩니다.

단위 테스트에서 직접 업데이트할 수 있는 속성은 이벤트 처리기를 이벤트에 연결 하 `PropertyChanged` 고 속성의 새 값을 설정한 후 이벤트가 발생 하는지 여부를 확인 하 여 테스트할 수 있습니다. 다음 코드 예제에서는 이러한 테스트를 보여 줍니다.

```
[Fact]
public async Task SettingOrderPropertyShouldRaisePropertyChanged()
{
    bool invoked = false;
    var orderService = new OrderMockService();
    var orderViewModel = new OrderDetailViewModel(orderService);

    orderViewModel.PropertyChanged += (sender, e) =>
    {
        if (e.PropertyName.Equals("Order"))
            invoked = true;
    };
    var order = await orderService.GetOrderAsync(1, GlobalSetting.Instance.AuthToken);
    await orderViewModel.InitializeAsync(order);

    Assert.True(invoked);
}
```

이 단위 테스트는 클래스의 메서드를 호출 하 여 `InitializeAsync` `OrderViewModel` 해당 `Order` 속성을 업데이트 합니다. `PropertyChanged`속성에 대해 이벤트가 발생 하는 경우 단위 테스트가 통과 됩니다 `Order` .

메시지 기반 통신 테스트

클래스를 사용 하 여 `MessagingCenter` 느슨하게 결합 된 클래스 간에 통신 하는 뷰 모델은 다음 코드 예제에서 보여 주는 것 처럼 테스트 중인 코드에서 보내는 메시지를 구독 하 여 단위 테스트를 수행할 수 있습니다.

```
[Fact]
public void AddCatalogItemCommandSendsAddProductMessageTest()
{
    bool messageReceived = false;
    var catalogService = new CatalogMockService();
    var catalogViewModel = new CatalogViewModel(catalogService);

    Xamarin.Forms.MessagingCenter.Subscribe<CatalogViewModel, CatalogItem>(
        this, MessageKeys.AddProduct, (sender, arg) =>
    {
        messageReceived = true;
    });
    catalogViewModel.AddCatalogItemCommand.Execute(null);

    Assert.True(messageReceived);
}
```

이 단위 테스트는에서 `CatalogViewModel` `AddProduct` 실행 되는에 대 한 응답으로 메시지를 게시 하는지 확인 `AddCatalogItemCommand` 합니다. `MessagingCenter`클래스가 멀티 캐스트 메시지 구독을 지원 하기 때문에 단위 테스트는 메시지를 구독 하 `AddProduct` 고 수신에 대 한 응답으로 콜백 대리자를 실행할 수 있습니다. 람다 식으로 지정 되는이 콜백 대리자는 `boolean` 문에서 테스트 동작을 확인 하는 데 사용 하는 필드를 설정 `Assert` 합니다.

예외 처리 테스트

다음 코드 예제에서 보여 주는 것 처럼 잘못 된 작업 또는 입력에 대해 특정 예외가 throw 되었는지 확인 하는 단위 테스트를 작성할 수도 있습니다.

```
[Fact]
public void InvalidEventNameShouldThrowArgumentExceptionText()
{
    var behavior = new MockEventToCommandBehavior
    {
        EventName = "OnItemTapped"
    };
    var listView = new ListView();

    Assert.Throws<ArgumentException>(() => listView.Behaviors.Add(behavior));
}
```

컨트롤에 라는 이벤트가 없으므로이 단위 테스트에서 예외를 throw 합니다 `ListView` `OnItemTapped` . `Assert.Throws<T>`메서드는 `T` 가 예상 되는 예외의 형식인 제네릭 메서드입니다. 메서드에 전달 된 인수는 `Assert.Throws<T>` 예외를 throw 하는 람다 식입니다. 따라서 람다 식에서을 throw 하는 경우 단위 테스트가 전달 됩니다 `ArgumentException` .

> **TIP**
> 예외 메시지 문자열을 검사 하는 단위 테스트를 작성 하지 마십시오. 예외 메시지 문자열은 시간이 지남에 따라 변경 될 수 있으므로 현재 상태를 사용 하는 단위 테스트는 불안정로 간주 됩니다.

유효성 검사 테스트

유효성 검사 구현을 테스트 하는 데는 두 가지 측면이 있습니다. 유효성 검사 규칙이 올바르게 구현 되었는지 테스트 하 고 `ValidatableObject<T>` 클래스가 예상 대로 수행 되는지 테스트 합니다.

유효성 검사 논리는 일반적으로 출력이 입력에 따라 달라 지는 자체 포함 된 프로세스 이기 때문에 간단 하 게 테스트할 수 있습니다. `Validate` 다음 코드 예제에서 보여 주는 것 처럼 하나 이상의 연결 된 유효성 검사 규칙을 포함 하는 각 속성에 대해 메서드를 호출 하는 결과에 대 한 테스트를 수행 해야 합니다.

```
[Fact]
public void CheckValidationPassesWhenBothPropertiesHaveDataTest()
{
    var mockViewModel = new MockViewModel();
    mockViewModel.Forename.Value = "John";
    mockViewModel.Surname.Value = "Smith";

    bool isValid = mockViewModel.Validate();

    Assert.True(isValid);
}
```

이 단위 테스트는 `ValidatableObject<T>` 인스턴스의 두 속성 모두에 데이터가 있는 경우 유효성 검사가 성공 하는지 확인 합니다 `MockViewModel` .

유효성 검사의 성공 여부를 확인 하는 것 외에도, 유효성 검사 단위 테스트 `Value` 에서는 `IsValid` 각 인스턴스의, 및 속성 값을 검사 `Errors` 하 여 `ValidatableObject<T>` 클래스가 예상 대로 수행 되는지 확인 해야 합니다. 다음 코드 예제에서는이 작업을 수행 하는 단위 테스트를 보여 줍니다.

```
[Fact]
public void CheckValidationFailsWhenOnlyForenameHasDataTest()
{
    var mockViewModel = new MockViewModel();
    mockViewModel.Forename.Value = "John";

    bool isValid = mockViewModel.Validate();

    Assert.False(isValid);
    Assert.NotNull(mockViewModel.Forename.Value);
    Assert.Null(mockViewModel.Surname.Value);
    Assert.True(mockViewModel.Forename.IsValid);
    Assert.False(mockViewModel.Surname.IsValid);
    Assert.Empty(mockViewModel.Forename.Errors);
    Assert.NotEmpty(mockViewModel.Surname.Errors);
}
```

이 단위 테스트는 `Surname` 의 속성에 `MockViewModel` 데이터가 없고 `Value` `IsValid` `Errors` 각 인스턴스의, 및 속성이 `ValidatableObject<T>` 올바르게 설정 된 경우 유효성 검사에 실패 하는지 확인 합니다.

요약

단위 테스트는 응용 프로그램의 작은 단위 (일반적으로 메서드)를 사용 하 여 코드의 나머지 부분에서 분리 하 고 예상 대로 작동 하는지 확인 합니다. 이러한 목표는 각 기능 단위가 예상 대로 수행 되는지 확인 하 여 응용 프로그램 전체에서 오류가 전파 되지 않도록 하는 것입니다.

종속 개체를 종속 개체의 동작을 시뮬레이트하는 모의 개체로 바꿔서 테스트 중인 개체의 동작을 격리할 수 있습니다. 이렇게 하면 웹 서비스, 데이터베이스 등의 리소스를 사용 하지 않고도 단위 테스트를 실행할 수 있습니다.

MVVM 응용 프로그램에서 모델을 테스트 하 고 모델을 확인 하는 것은 다른 클래스를 테스트 하는 것과 동일 하며, 동일한 도구와 기법을 사용할 수 있습니다.

관련 링크

- 다운로드 전자책 (2Mb PDF)
- eShopOnContainers (GitHub) (샘플)