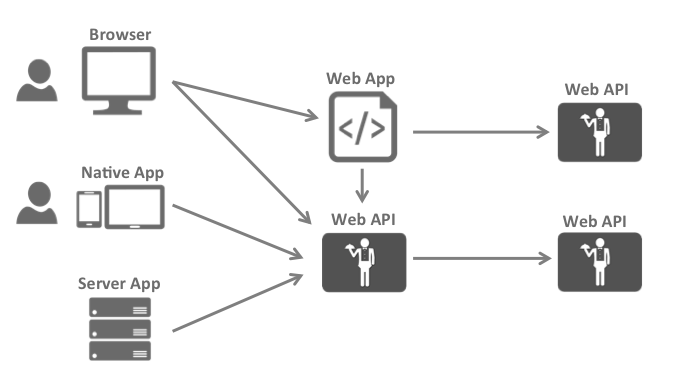
**The Big Picture**

Most modern applications look more or less like this:



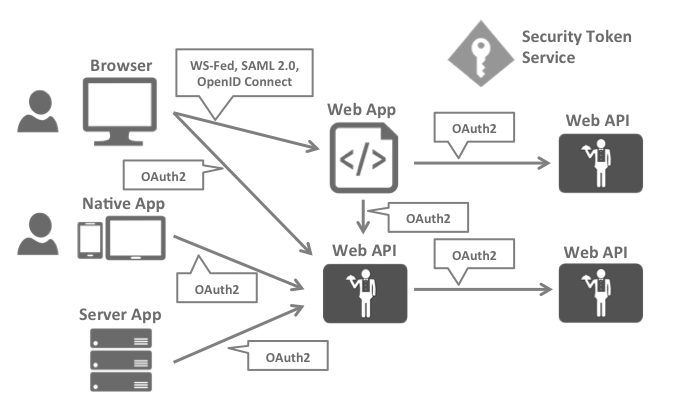
The most common interactions are:

* Browsers communicate with web applications
* Web applications communicate with web APIs (sometimes on their own, sometimes on behalf of a user)
* Browser-based applications communicate with web APIs
* Native applications communicate with web APIs
* Server-based applications communicate with web APIs
* Web APIs communicate with web APIs (sometimes on their own, sometimes on behalf of a user)

Typically each and every layer (front-end, middle-tier and back-end) has to protect resources and implement authentication and/or authorization – often against the same user store.

Outsourcing these fundamental security functions to a security token service prevents duplicating that functionality across those applications and endpoints.

Restructuring the application to support a security token service leads to the following architecture and protocols:



Such a design divides security concerns into two parts:

**Authentication**

Authentication is needed when an application needs to know the identity of the current user. Typically these applications manage data on behalf of that user and need to make sure that this user can only access the data for which he is allowed. The most common example for that is (classic) web applications – but native and JS-based applications also have a need for authentication.

The most common authentication protocols are SAML2p, WS-Federation and OpenID Connect – SAML2p being the most popular and the most widely deployed.

OpenID Connect is the newest of the three, but is considered to be the future because it has the most potential for modern applications. It was built for mobile application scenarios right from the start and is designed to be API friendly.

**API Access**

Applications have two fundamental ways with which they communicate with APIs – using the application identity, or delegating the user’s identity. Sometimes both methods need to be combined.

OAuth2 is a protocol that allows applications to request access tokens from a security token service and use them to communicate with APIs. This delegation reduces complexity in both the client applications as well as the APIs since authentication and authorization can be centralized.

**OpenID Connect and OAuth 2.0 – better together**

OpenID Connect and OAuth 2.0 are very similar – in fact OpenID Connect is an extension on top of OAuth 2.0. The two fundamental security concerns, authentication and API access, are combined into a single protocol - often with a single round trip to the security token service.

We believe that the combination of OpenID Connect and OAuth 2.0 is the best approach to secure modern applications for the foreseeable future. IdentityServer4 is an implementation of these two protocols and is highly optimized to solve the typical security problems of today’s mobile, native and web applications.

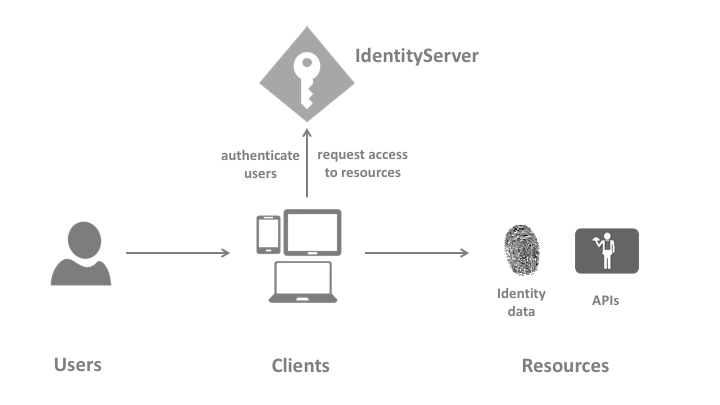
**How IdentityServer4 can help**

IdentityServer is middleware that adds the spec compliant OpenID Connect and OAuth 2.0 endpoints to an arbitrary ASP.NET Core application.

Typically, you build (or re-use) an application that contains a login and logout page (and maybe consent - depending on your needs), and the IdentityServer middleware adds the necessary protocol heads to it, so that client applications can talk to it using those standard protocols.

**Terminology**

The specs, documentation and object model use a certain terminology that you should be aware of.



**IdentityServer**

IdentityServer is an OpenID Connect provider - it implements the OpenID Connect and OAuth 2.0 protocols.

Different literature uses different terms for the same role - you probably also find security token service, identity provider, authorization server, IP-STS and more.

But they are in a nutshell all the same: a piece of software that issues security tokens to clients.

IdentityServer has a number of jobs and features - including:

* protect your resources
* authenticate users using a local account store or via an external identity provider
* provide session management and single sign-on
* manage and authenticate clients
* issue identity and access tokens to clients
* validate tokens

**User**

A user is a human that is using a registered client to access resources.

**Client**

A client is a piece of software that requests tokens from IdentityServer - either for authenticating a user (requesting an identity token) or for accessing a resource (requesting an access token). A client must be first registered with IdentityServer before it can request tokens.

Examples for clients are web applications, native mobile or desktop applications, SPAs, server processes etc.

**Resources**

Resources are something you want to protect with IdentityServer - either identity data of your users, or APIs.

Every resource has a unique name - and clients use this name to specify to which resources they want to get access to.

**Identity data** Identity information (aka claims) about a user, e.g. name or email address.

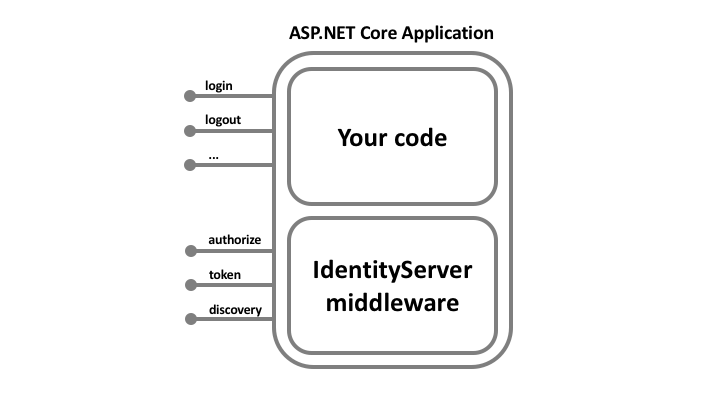
**APIs** APIs resources represent functionality a client wants to invoke - typically modelled as Web APIs, but not necessarily.

**Identity Token**

An identity token represents the outcome of an authentication process. It contains at a bare minimum an identifier for the user (called the *sub* aka subject claim) and information about how and when the user authenticated. It can contain additional identity data.

**Access Token**

An access token allows access to an API resource. Clients request access tokens and forward them to the API. Access tokens contain information about the client and the user (if present). APIs use that information to authorize access to their data.



The hosting application can be as complex as you want, but we typically recommend to keep the attack surface as small as possible by including authentication related UI only.

**Overview**

The quickstarts provide step by step instructions for various common IdentityServer scenarios. They start with the absolute basics and become more complex - it is recommended you do them in order.

* adding IdentityServer to an ASP.NET Core application
* configuring IdentityServer
* issuing tokens for various clients
* securing web applications and APIs
* adding support for EntityFramework based configuration
* adding support for ASP.NET Identity

Every quickstart has a reference solution - you can find the code in the [samples](https://github.com/IdentityServer/IdentityServer4/tree/main/samples/Quickstarts) folder.

**Preparation**

The first thing you should do is install our templates:

dotnet **new** -i IdentityServer4.Templates

They will be used as a starting point for the various tutorials.

**Note**

If you are using private NuGet sources do not forget to add the –nuget-source parameter: –nuget-source <https://api.nuget.org/v3/index.json>

OK - let’s get started!

**Note**

The quickstarts target the IdentityServer 4.x and ASP.NET Core 3.1.x - there are also quickstarts for [ASP.NET Core 2](http://docs.identityserver.io/en/aspnetcore2/quickstarts/0_overview.html) and [ASP.NET Core 1](http://docs.identityserver.io/en/aspnetcore1/quickstarts/0_overview.html).