



UWP



Import and export



WINDOWS > GUIDES > ANNOTATIONS > IMPORT AND EXPORT

Import and export annotations from a database in UWP

Using the export and import functionality provided by Nutrient UWP SDK, you can export any changes made to annotations to a database of your choice, and then import them when you open the document again to restore the changes.

This approach allows you to save on bandwidth, as you won't need to export the whole document whenever there are any modifications. Rather, you only send the annotation information whenever necessary, retaining the same base file.

To enable this type of setup, you can use either [Instant JSON](#) or [XFDF](#) as the export format. Both of these can be converted into strings for easy storage on whichever database you choose.

Exporting and importing Instant JSON from a database

The Instant JSON API on Nutrient UWP SDK allows you to get a `JsonObject` containing annotation data, which can then be stored and reused when opening a file. For that, you can call the `Document.ExportInstantJsonAsync` method as shown below:

```
var instantJsonObject = await PDFView.Document.ExportInstantJsonAsync();
```



Then, a call to `Stringify` from the `JsonObject` will give you a plain `string` that can be stored in whichever database in whichever way you prefer.



ASK AI

When getting Instant JSON information from your database, reload it into a document:

```
1  ...
2  var instantJsonObject = JsonObject.Parse(instantJsonString);
3  await PDFView.Document.ImportInstantJsonAsync(instantJsonObject);
```

Using Instant JSON, you can also manually go through the annotations present in a document and only serialize the ones you want using the `ToJson` method from `IAnnotation`. More information on Instant JSON, including its uses and properties, can be found in our [aply named guide](#).

Exporting and importing XFDF from a database

Using XFDF, you're able to export the annotation information to either an `IDataSink` or `DataWriter`. The examples below show an `InMemoryRandomAccessStream` handling the data inside the `DataWriter`, so its contents are only present in memory during the `using` statement:

```
1  using (DataWriter dataWriter = new DataWriter(new InMemoryRandomAccessStream)
2  {
3      await PDFView.Document.ExportXfdfToDataWriterAsync(dataWriter);
4  }
```

The end result of this operation is a series of characters written into the `DataWriter` following the XFDF XML-based standard. For retrieving this data in the form of a `string`, you can do the following conversion to `IBuffer`:

```
1  using (DataWriter dataWriter = new DataWriter(new InMemoryRandomAccessStream)
2  {
3      await PDFView.Document.ExportXfdfToDataWriterAsync(dataWriter);
4
5      IBuffer buffer = dataWriter.DetachBuffer();
6      var annotationData = Encoding.UTF8.GetString(buffer.ToArray());
7
8      ...
9  }
```

With this information saved to the database of your choice, you can import it when opening a document. After getting the XFDF from your database and opening the file, create a `stream` containing the `string` content:

```

1  using (Stream stream = Encoding.UTF8.GetBytes(xfdfAnnotationString).AsBuffer())
2  {
3      ...
4  }

```



After that, create an `IDataProvider` to make the stream usable by the `Document`, which will then be passed in as a parameter when calling `Document.ImportXfdfAsync` for loading in the XFDF data:

```

1  using (Stream stream = Encoding.UTF8.GetBytes(xfdfAnnotationString).AsBuffer())
2  {
3      var xfdfProvider = new RandomAccessStreamDataProvider(stream.AsRandomAccessStream());
4      await PDFView.Document.ImportXfdfAsync(xfdfProvider);
5  }

```



For more information, refer to our [XFDF-specific guide](#). Moreover, our [custom data providers guide](#) contains extra information on `IDataProvider`s and how to use them.

Was this helpful?

☒ YES

☐ NO

Questions? [Contact us](#)

