Multi-platform apps
Command-line & server apps
Overview
Get started
Write command-line apps
Fetch data from the internet
Write HTTP servers
Libraries and packages
Google Cloud
Web apps
Environment declarations
Tools & techniques
Resources
Related sites
API reference
Blog

Fetch data from the internet

What you'll learn

- The basics of what HTTP requests and URIs are and what they are used for.
- Making HTTP requests using package:http.
- Decoding JSON strings into Dart objects with dart:convert.
- Converting JSON objects into class-based structures.

Most applications require some form of communication or data retrieval from the internet. Many apps do so through HTTP requests, which are sent from a client to a server to perform a specific action for a resource identified through a URI (Uniform Resource Identifier).

Data communicated over HTTP can technically be in any form, but using JSON (JavaScript Object Notation) is a popular choice due to its human-readability and language independent nature. The Dart SDK and ecosystem also have extensive support for JSON with multiple options to best meet your app's requirements.

In this tutorial, you will learn more about HTTP requests, URIs, and JSON. Then you will learn how to use package:http as well as Dart's JSON support in the dart:convert library to fetch, decode, then use JSON-formatted data retrieved from an HTTP server.

Background concepts

The following sections provide some extra background and information around the technologies and concepts used in the tutorial to facilitate fetching data from the server. To skip directly to the tutorial content, see Retrieve the necessary dependencies.

JSON

JSON (JavaScript Object Notation) is a data-interchange format that has become ubiquitous across application development and client-server communication. It is lightweight but also easy for humans to read and write due to being text based. With JSON, various data types and simple data structures such as lists and maps can be serialized and represented by strings.

Most languages have many implementations and parsers have become extremely fast, so you don't need to worry about interoperability or performance. For more information about the JSON format, see Introducing JSON. To learn more about working with JSON in Dart, see the Using JSON guide.

HTTP requests

HTTP (Hypertext Transfer Protocol) is a stateless protocol designed for transmitting documents, originally between web clients and web servers. You interacted with the protocol to load this page, as your browser uses an HTTP GET request to retrieve the contents of a page from a web server. Since its introduction, use of the HTTP protocol and its various versions have expanded to applications outside the web as well, essentially wherever communication from a client to a server is needed.

HTTP requests sent from the client to communicate with the server are composed of multiple components. HTTP libraries, such as package:http, allow you to specify the following kinds of communication:

- An HTTP method defining the desired action, such as GET to retrieve data or POST to submit new data.
- The location of the resource through a URI.
- The version of HTTP being used.
- Headers that provide extra information to the server.
- An optional body, so the request can send data to the server, not just retrieve it.

To learn more about the HTTP protocol, check out An overview of HTTP on the mdn web docs.

URIs and URLs

To make an HTTP request, you need to provide a URI (Uniform Resource Identifier) to the resource. A URI is a character string that uniquely identifies a resource. A URL (Uniform Resource Locator) is a specific kind of URI that also provides the location of the resource. URLs for resources on the web contain three pieces of information. For this current page, the URL is composed of:

- The scheme used for determining the protocol used: https
- The authority or hostname of the server: dart.dev
- The path to the resource: /tutorials/server/fetch-data.html

There are other optional parameters as well that aren't used by the current page:

- Parameters to customize extra behavior: ?key1=value1&key2=value2
- An anchor, that isn't sent to the server, which points to a specific location in the resource: #uris

To learn more about URLs, see What is a URL? on the mdn web docs.

Retrieve the necessary dependencies

You can directly use dart:io or dart:html to make HTTP requests, however those libraries are platform dependent. package:http provides a cross-platform library for making composable HTTP requests, with optional fine-grained control.

To add a dependency on package:http, run the following dart pub add command from the top of your repo:

```
$ dart pub add http
```

To use package:http in your code, import it and optionally specify a library prefix:

```dart
import 'package:http/http.dart' as http;
```

To learn more specifics about package:http, see its page on the pub.dev site and its API documentation.

Build a URL

As previously mentioned, to make an HTTP request, you first need a URL that identifies the resource being requested or endpoint being accessed.

In Dart, URLs are represented through Uri objects. There are many ways to build an Uri, but due to its flexibility, parsing a string with Uri.parse to create one is a common solution.

The following snippet shows two ways to create a Uri object pointing to mock JSON-formatted information about package:http hosted on this site:

```dart
// Parse the entire URI, including the scheme
Uri.parse('https://dart.dev/f/packages/http.json');

// Specifically create a URI with the https scheme
Uri.https('dart.dev', '/f/packages/http.json');
```

To learn about other ways of building and interacting with URIs, see the library tour's discussion about URIs.

Make a network request

If you just need to quickly fetch a string representation of a requested resource, you can use the top-level read function found in package:http that returns a Future<String> or throws a ClientException if the request wasn't successful. The following example uses read to retrieve the mock JSON-formatted information about package:http as a string, then prints it out:

Note: Many functions in package:http, including read, access the network and perform potentially time-consuming operations, therefore they do so asynchronously and return a Future. If you haven't encountered futures yet, you can learn about them—as well as the async and await keywords—in the asynchronous programming codelab.

```dart
void main() async {
  final httpPackageUrl = Uri.https('dart.dev', '/f/packages/http.json');
  final httpPackageInfo = await http.read(httpPackageUrl);
  print(httpPackageInfo);
}
```

This results in the following JSON-formatted output, which can also be seen in your browser at https://dart.dev/f/packages/http.json.

```json
{
  "name": "http",
  "latestVersion": "0.13.5",
  "description": "A composable, multi-platform, Future-based API for HTTP requests.",
  "publisher": "dart.dev",
  "repository": "https://github.com/dart-lang/http"
}
```

Note the structure of the data (in this case a map), as you will need it when decoding the JSON later on.

If you need other information from the response, such as the status code or the headers, you can instead use the top-level get function that returns a Future with a Response.

The following snippet uses get to get the whole response in order to exit early if the request was not successful, which is indicated with a status code of **200**:

```dart
void main() async {
  final httpPackageUrl = Uri.https('dart.dev', '/f/packages/http.json');
  final httpPackageResponse = await http.get(httpPackageUrl);
  if (httpPackageResponse.statusCode != 200) {
    print('Failed to retrieve the http package!');
    return;
  }
  print(httpPackageResponse.body);
}
```

There are many other status codes besides **200** and your app might want to handle them differently. To learn more about what different status codes mean, see HTTP response status codes on the mdn web docs.

Some server requests require more information, such as authentication or user-agent information; in this case you might need to include HTTP headers. You can specify headers by passing in a Map<String, String> of the key-value pairs as the headers optional named parameter:

```dart
await http.get(Uri.https('dart.dev', '/f/packages/http.json'),
    headers: {'User-Agent': '<product name>/<product-version>'});
```

Make multiple requests

If you're making multiple requests to the same server, you can instead keep a persistent connection through a Client, which has similar methods to the top-level ones. Just make sure to clean up with the close method when done:

```dart
void main() async {
  final httpPackageUrl = Uri.https('dart.dev', '/f/packages/http.json');
  final client = http.Client();
  try {
    final httpPackageInfo = await client.read(httpPackageUrl);
    print(httpPackageInfo);
  } finally {
    client.close();
  }
}
```

To enable the client to retry failed requests, import package:http/retry.dart and wrap your created Client in a RetryClient:

```dart
import 'package:http/http.dart' as http;
import 'package:http/retry.dart';

void main() async {
  final httpPackageUrl = Uri.https('dart.dev', '/f/packages/http.json');
  final client = RetryClient(http.Client());
  try {
    final httpPackageInfo = await client.read(httpPackageUrl);
    print(httpPackageInfo);
  } finally {
    client.close();
  }
}
```

The RetryClient has a default behavior for how many times to retry and how long to wait between each request, but its behavior can be modified through parameters to the RetryClient() or RetryClient.withDelays() constructors.

package:http has much more functionality and customization, so make sure to check out its page on the pub.dev site and its API documentation.

Decode the retrieved data

While you now have made a network request and retrieved the returned data as string, accessing specific portions of information from a string can be a challenge.

Since the data is already in a JSON format, you can use Dart's built-in json.decode function in the dart:convert library to convert the raw string into a JSON representation using Dart objects. In this case, the JSON data is represented in a map structure and, in JSON, map keys are always strings, so you can cast the result of json.decode to a Map<String, dynamic>:

```dart
import 'dart:convert';

import 'package:http/http.dart' as http;

void main() async {
  final httpPackageUrl = Uri.https('dart.dev', '/f/packages/http.json');
  final httpPackageInfo = await http.read(httpPackageUrl);
  final httpPackageJson = json.decode(httpPackageInfo) as Map<String, dynamic>;
  print(httpPackageJson);
}
```

Create a structured class to store the data

To provide the decoded JSON with more structure, making it easier to work with, create a class that can store the retrieved data using specific types depending on the schema of your data.

The following snippet shows a class-based representation that can store the package information returned from the mock JSON file you requested. This structure assumes all fields except the repository are required and provided every time.

```dart
class PackageInfo {
  final String name;
  final String latestVersion;
  final String description;
  final String publisher;
  final Uri? repository;

  PackageInfo({
    required this.name,
    required this.latestVersion,
    required this.description,
    required this.publisher,
    this.repository,
  });
}
```

Encode the data into your class

Now that you have a class to store your data in, you need to add a mechanism to convert the decoded JSON into a PackageInfo object.

Convert the decoded JSON by manually writing a fromJson method matching the earlier JSON format, casting types as necessary and handling the optional repository field:

```dart
class PackageInfo {
  // ···

  factory PackageInfo.fromJson(Map<String, dynamic> json) {
    final repository = json['repository'] as String?;

    return PackageInfo(
      name: json['name'] as String,
      latestVersion: json['latestVersion'] as String,
      description: json['description'] as String,
      publisher: json['publisher'] as String,
      repository: repository != null ? Uri.tryParse(repository) : null,
    );
  }
}
```

A handwritten method, such as in the previous example, is often sufficient for relatively simple JSON structures, but there are more flexible options as well. To learn more about JSON serialization and deserialization, including automatic generation of the conversion logic, see the Using JSON guide.

Convert the response to an object of your structured class

Now you have a class to store your data and a way to convert the decoded JSON object into an object of that type. Next, you can write a function that pulls everything together:

1. Create your URI based off a passed-in package name.
2. Use http.get to retrieve the data for that package.
3. If the request didn't succeed, throw an Exception or preferably your own custom Exception subclass.
4. If the request succeeded, use json.decode to decode the response body into a JSON string.
5. Converted the decoded JSON string into a PackageInfo object using the PackageInfo.fromJson factory constructor you created.

```dart
Future<PackageInfo> getPackage(String packageName) async {
  final packageUrl = Uri.https('dart.dev', '/f/packages/$packageName.json');
  final packageResponse = await http.get(packageUrl);

  // If the request didn't succeed, throw an exception
  if (packageResponse.statusCode != 200) {
    throw PackageRetrievalException(
      packageName: packageName,
      statusCode: packageResponse.statusCode,
    );
  }

  final packageJson = json.decode(packageResponse.body) as Map<String, dynamic>;

  return PackageInfo.fromJson(packageJson);
}

class PackageRetrievalException implements Exception {
  final String packageName;
  final int? statusCode;

  PackageRetrievalException({required this.packageName, this.statusCode});
}
```

Utilize the converted data

Now that you've retrieved data and converted it to a more easily accessible format, you can use it however you'd like. Some possibilities include outputting information to a CLI, or displaying it in a web or Flutter app.

Here is complete, runnable example that requests, decodes, then displays the mock information about the http and path packages:

Dart | Install SDK | Format | Reset | Run

```dart
import 'dart:convert';

import 'package:http/http.dart' as http;

void main() async {
  await printPackageInformation('http');
  print('');
  await printPackageInformation('path');
}

Future<void> printPackageInformation(String packageName) async {
  final PackageInfo packageInfo;

  try {
    packageInfo = await getPackage(packageName);
  } on PackageRetrievalException catch (e) {
    print(e);
    return;
  }

  print('Information about the $packageName package:');
  print('Latest version: ${packageInfo.latestVersion}');
  print('Description: ${packageInfo.description}');
  print('Publisher: ${packageInfo.publisher}');

  final repository = packageInfo.repository;
  if (repository != null) {
    print('Repository: $repository');
```

Console

no issues

Flutter note: For another example that covers fetching then displaying data in Flutter, see the Fetching data from the internet Flutter recipe.

What next?

Now that you have retrieved, parsed, and used data from the internet, consider learning more about Concurrency in Dart. If your data is large and complex, you can move retrieval and decoding to another isolate as a background worker to prevent your interface from becoming unresponsive.