Fetch data dynamically

Contents

What's the point?

- Data on the web is often formatted in JSON.
- JSON is text based and human readable.
- The dart:convert library provides support for JSON.
- Use HttpRequest to dynamically load data.

Web apps often use JSON (JavaScript Object Notation) to pass data between clients and servers. Data can be *serialized* into a JSON string, which is then passed between a client and server, and revived as an object at its destination. This tutorial shows you how to use functions in the dart:convert library to produce and consume JSON data. Because JSON data is typically loaded dynamically, this tutorial also shows how a web app can use an HTTP request to get data from an HTTP server. For web apps, HTTP requests are served by the browser in which the app is running, and thus are subject to the browser's security restrictions.

Note: This page uses embedded DartPads to display runnable examples. If you see empty boxes instead of DartPads, go to the DartPad troubleshooting page.

About JSON

The JSON data format is easy for humans to write and read because it is lightweight and text based. With JSON, various data types and simple data structures such as lists and maps can be serialized and represented by strings.

Try it! The following app displays the JSON string for data of various types. Click **Run** to start the app. Then change the values of the input elements, and check out the JSON format for each data type.

Note: If you see an empty box instead of code, go to the DartPad troubleshooting page.

Dart | HTML | CSS — Install SDK | Format | Reset | Run

```
<h1>It's All About You</h1>

<table>
  <thead>
  <tr>        <th> </th>
    <th>Enter value</th>
    <th>Data type</th>
    <th>JSON string</th>
  </tr>
  </thead>

  <tbody>
  <tr>
    <td align="right">Favorite number:</td>
    <td><input type="text" id="favoriteNumber"></td>
    <td>integer</td>
    <td><textarea class="result" id="intAsJson" readonly></textarea></t
  </tr>

  <tr>
    <td align="right">Do you know pi?</td>
```

It's All About You

	Enter value
Favorite number:	73
Do you know pi?	3.141592
What's your sign?	Cancer
A few of your favorite things?	monkeys parrots lattes

Console — no issues

The `dart:convert` library contains two convenient functions for working with JSON strings:

dart:convert function	Description
json.decode()	Builds Dart objects from a string containing JSON data.
json.encode()	Serializes a Dart object into a JSON string.

To use these functions, you need to import dart:convert into your Dart code:

```
import 'dart:convert';
```

The `json.encode()` and `json.decode()` functions can handle these Dart types automatically:

- `num`
- `String`
- `bool`
- `Null`
- `List`
- `Map`

Serializing data into JSON

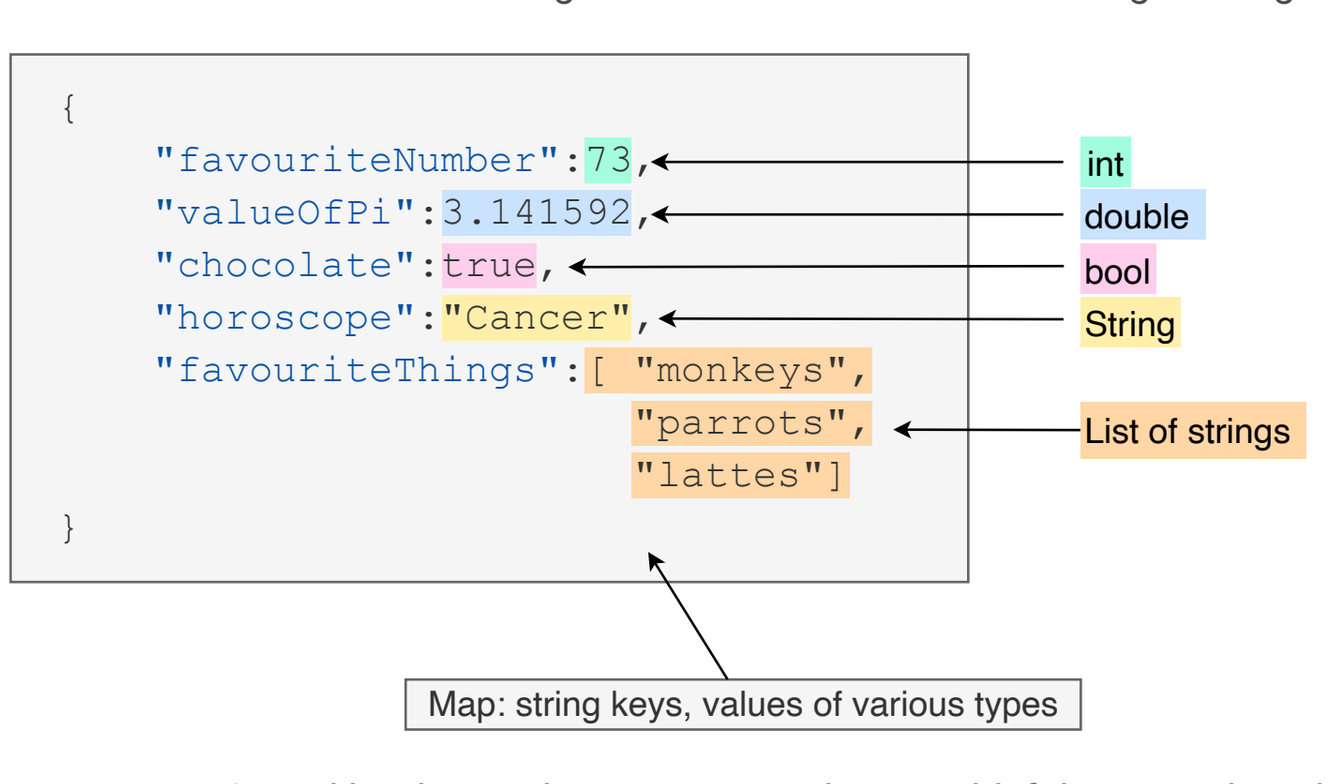
Use the json.encode() function to serialize an object that supports JSON. The `_showJson()` function, from the example, converts all of the data to JSON strings.

```
void _showJson([Event? _]) {
  // Grab the data that will be converted to JSON.
  final favNum = int.tryParse(favoriteNumber.value ?? '');
  final pi = double.tryParse(valueOfPi.value ?? '');
  final chocolate = loveChocolate.checked;
  final sign = horoscope.value;
  final favoriteThings = <String>[
    favOne.value ?? '',
    favTwo.value ?? '',
    favThree.value ?? '',
  ];

  final formData = {
    'favoriteNumber': favNum,
    'valueOfPi': pi,
    'chocolate': chocolate,
    'horoscope': sign,
    'favoriteThings': favoriteThings
  };

  // Convert to JSON and display results.
  intAsJson.text = json.encode(favNum);
  doubleAsJson.text = json.encode(pi);
  boolAsJson.text = json.encode(chocolate);
  stringAsJson.text = json.encode(sign);
  listAsJson.text = json.encode(favoriteThings);
  mapAsJson.text = json.encode(formData);
}
```

Shown below is the JSON string that results from the code using the original values from the app:



- **Numeric** and boolean values appear as they would if they were literal values in code, without quotes or other delineating marks.
- A **boolean** value is either `true` or `false`.
- The **null** value is represented as `null`.
- **Strings** are contained within *double* quotes.
- A **list** is delineated with square brackets; its items are comma-separated. The list in this example contains strings.
- A **map** is delineated with curly brackets; it contains comma-separated key/value pairs, where the key appears first, followed by a colon, followed by the value. In this example, the keys in the map are strings. The values in the map vary in type but they are all JSON-parsable.

Parsing JSON data

Use the json.decode() function from the dart:convert library to create Dart objects from a JSON string. The example initially populates the values in the form from this JSON string:

```
const jsonDataAsString = '''{
  "favoriteNumber": 73,
  "valueOfPi": 3.141592,
  "chocolate": true,
  "horoscope": "Cancer",
  "favoriteThings": ["monkeys", "parrots", "lattes"]
}''';

Map<String, dynamic> jsonData =
    json.decode(jsonDataAsString) as Map<String, dynamic>;
```

This code calls json.decode() with a properly formatted JSON string.

Warning: Dart strings can use either single or double quotes to denote strings. **JSON requires double quotes**.

In this example, the full JSON string is hard coded into the Dart code, but it could be created by the form itself or read from a static file or fetched from a server. An example later in this page shows how to dynamically fetch JSON data from a file that is co-located with the code for the app.

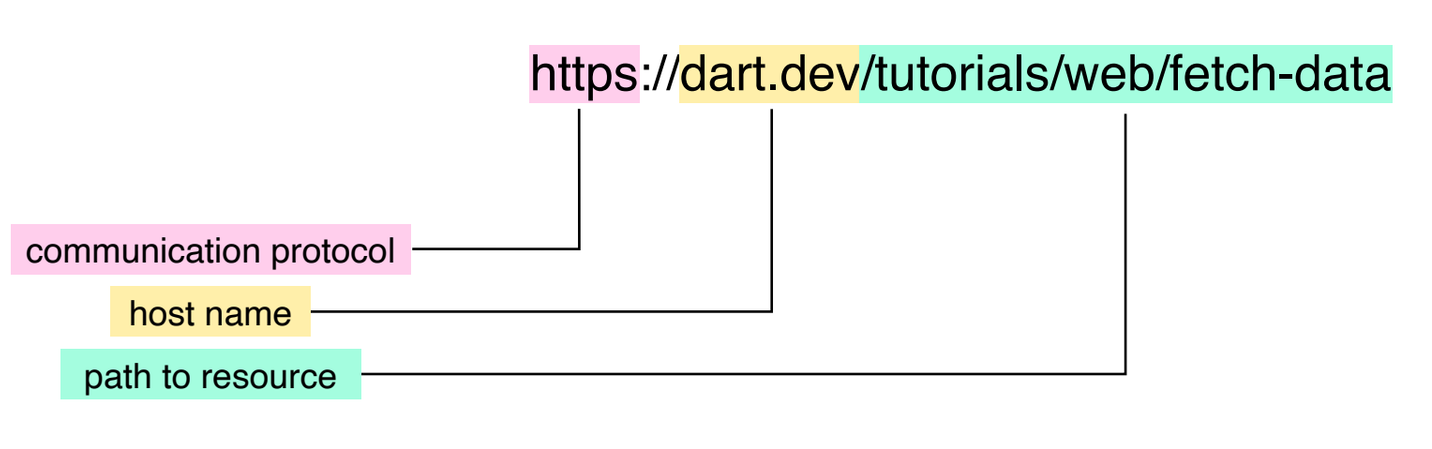
The `json.decode()` function reads the string and builds Dart objects from it. In this example, the `json.decode()` function creates a `Map<String, dynamic>` object based on the information in the JSON string. The `Map` contains objects of various types including an integer, a double, a boolean value, a regular string, and a list. All of the keys in the map are strings.

About URIs and HTTP requests

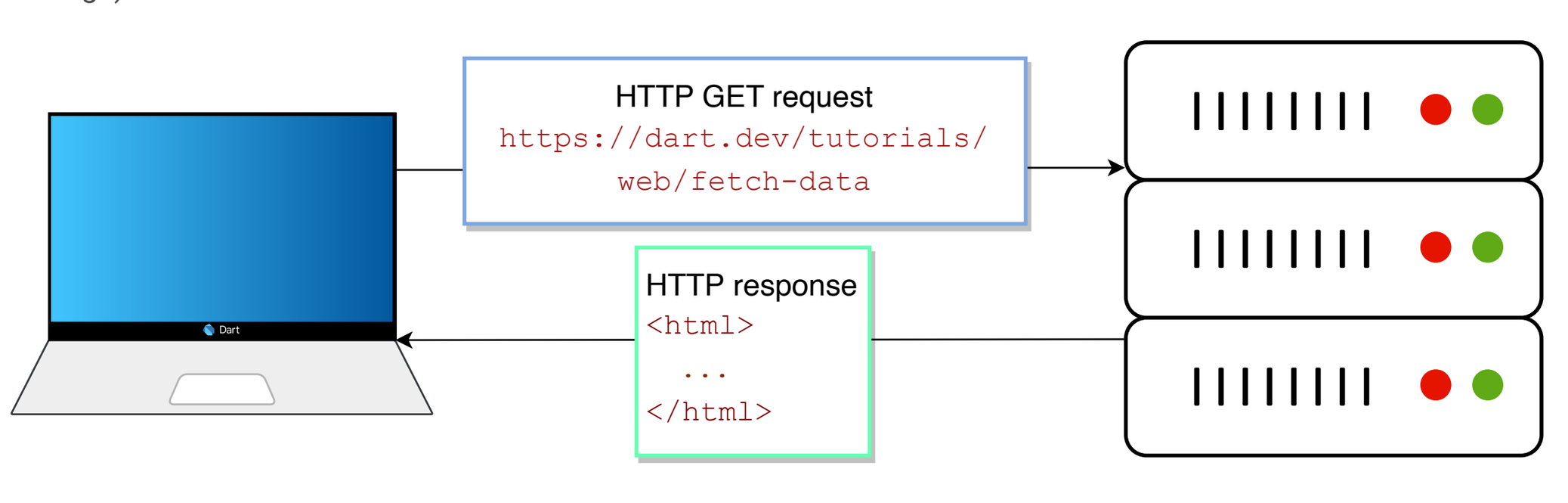
To make an HTTP GET request from within a web app, you need to provide a URI (Uniform Resource Identifier) for the resource. A URI is a character string that uniquely names a resource. A URL (Uniform Resource Locator) is a specific kind of URI that also provides the location of a resource. URLs for resources on the World Wide Web contain three pieces of information:

- The protocol used for communication
- The hostname of the server
- The path to the resource

For example, the URL for this page breaks down as follows:



This URL specifies the HTTP protocol. When you enter an HTTP address into a web browser, the browser sends an HTTP GET request to a web server, and the web server sends an HTTP response that contains the contents of the page (or an error message).



Most HTTP requests in a web browser are simple GET requests asking for the contents of a page. However, the HTTP protocol allows for other types of requests, such as POST for sending data from the client.

A Dart web app running inside of a browser can make HTTP requests. These HTTP requests are handled by the browser in which the app is running. Even though the browser itself can make HTTP requests anywhere on the web, a Dart web app running inside the browser can make only *limited* HTTP requests because of security restrictions. Practically speaking, because of these limitations, HTTP requests from web apps are primarily useful for retrieving information in files specific to and co-located with the app.

Security note: Browsers place tight security restrictions on HTTP requests made by embedded apps. Specifically, any resources requested by a web app must be served from the same origin. That is, the resources must be from the same protocol, host, and port as the app itself. This means that your web app cannot request just any resource from the web with HTTP requests through the browser, even if that request is seemingly harmless (like a GET.)

Some servers do allow cross-origin requests through a mechanism called CORS (Cross-origin resource sharing), which uses headers in an HTTP request to ask for and receive permission. CORS is server specific.

The SDK provides these useful classes for formulating URIs and making HTTP requests:

Dart code	Library	Description
Uri	dart:core	Uniform resource identifier
HttpRequest	dart:html	Client-side HTTP request object. For use in web apps.
HttpRequest	dart:io	Server-side HTTP request object. Does not work in web apps.

Using getString() to load a file

One useful HTTP request your web app *can* make is a GET request for a data file served from the same origin as the app. The example below reads a data file called `portmanteaux.json` that contains a JSON-formatted list of words. When you click the button, the app makes a GET request of the server and loads the file.

Try it! Click **Run** and then click the **Get portmanteaux** button.

Dart | HTML | CSS — Install SDK | Format | Reset | Run

```
import 'dart:async';
import 'dart:convert';
import 'dart:html';

final UListElement wordList = querySelector('#wordList') as UListElemen

void main() {
  querySelector('#getWords')!.onClick.listen(makeRequest);
}

Future<void> makeRequest(Event _) async {
  const path = 'https://dart.dev/f/portmanteaux.json';
  try {
    // Make the GET request
    final jsonString = await HttpRequest.getString(path);
    // The request succeeded. Process the JSON.
    processResponse(jsonString);
  } catch (e) {
    // The GET request failed. Handle the error.
```

UI Output

Console — no issues

This program uses a convenience method, getString(), provided by the HttpRequest class to request the file from the server.

```
Future<void> makeRequest(Event _) async {
  const path = 'https://dart.dev/f/portmanteaux.json';
  try {
    // Make the GET request
    final jsonString = await HttpRequest.getString(path);
    // The request succeeded. Process the JSON.
    processResponse(jsonString);
  } catch (e) {
    // The GET request failed. Handle the error.
    // ···
  }
}

void processResponse(String jsonString) {
  for (final portmanteau in json.decode(jsonString) as List<dynamic>) {
    wordList.children.add(LIElement()..text = portmanteau as String);
  }
}
```

The `getString()` method uses a Future object to handle the request. A Future is a way to perform potentially time-consuming operations, such as HTTP requests, asynchronously. 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. Until then, you can use the code above as a guide and provide your own code for the body of the `processResponse()` function and your own code to handle the error.

Using an HttpRequest object to load a file

The `getString()` method is good for an HTTP GET request that returns a string loaded from a resource. For other cases, you need to create an HttpRequest object, configure its header and other information, and use the send() method to make the request.

This section rewrites the portmanteaux code to explicitly construct an HttpRequest object.

Dart | HTML | CSS — Install SDK | Format | Reset | Run

```
import 'dart:async';
import 'dart:html';
import 'dart:convert';

final UListElement wordList = querySelector('#wordList') as UListElemen

void main() {
  querySelector('#getWords')!.onClick.listen(makeRequest);
}

Future<void> makeRequest(Event _) async {
  const path = 'https://dart.dev/f/portmanteaux.json';
  final httpRequest = HttpRequest();
  httpRequest
    ..open('GET', path)
    ..onLoadEnd.listen((e) => requestComplete(httpRequest))
    ..send('');
}
```

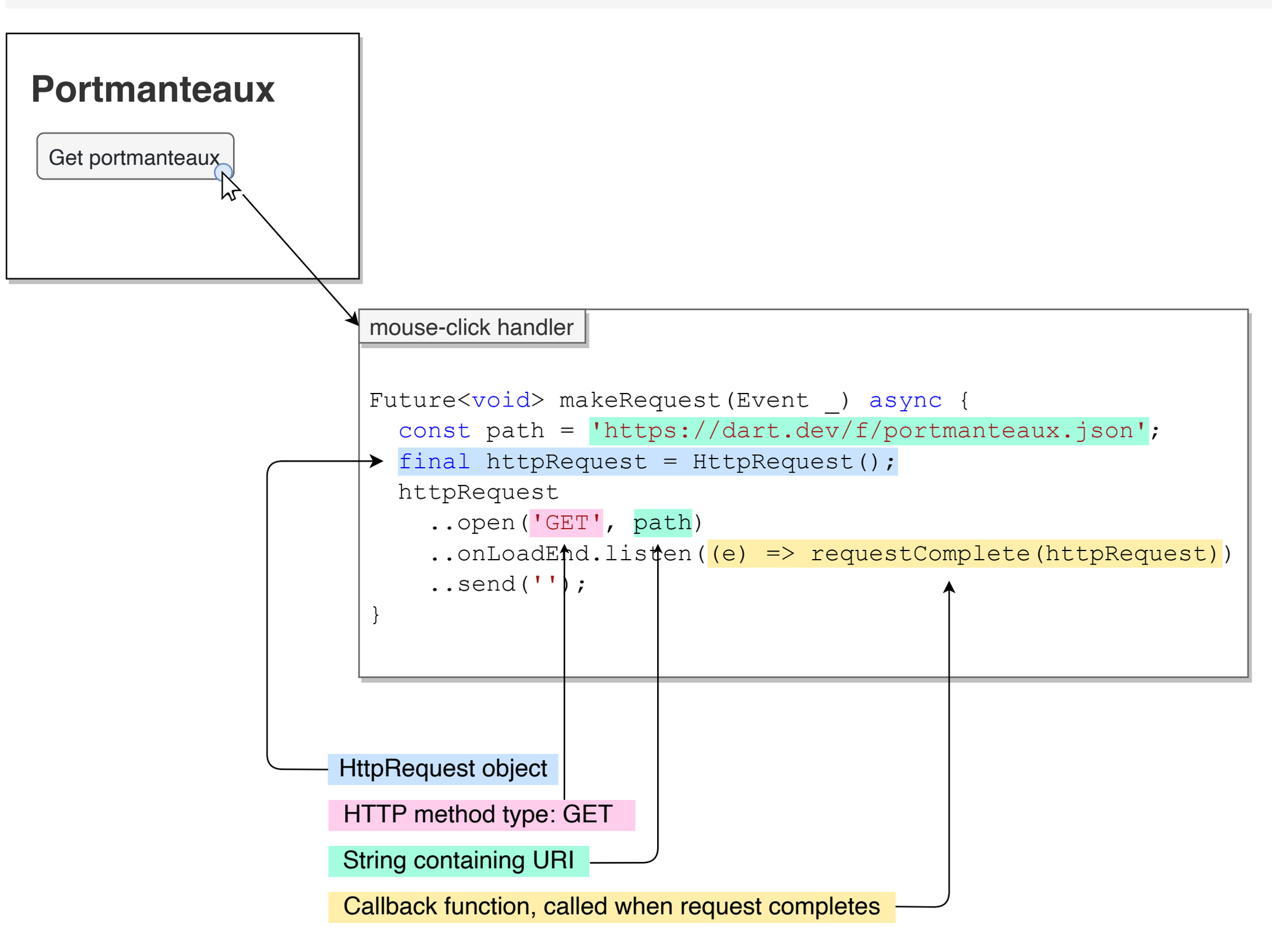
UI Output

Console — no issues

Setting up the HttpRequest object

The mouse-click handler for the button creates an HttpRequest object, configures it with a URI and callback function, and then sends the request. Let's take a look at the Dart code:

```
Future<void> makeRequest(Event _) async {
  const path = 'https://dart.dev/f/portmanteaux.json';
  final httpRequest = HttpRequest();
  httpRequest
    ..open('GET', path)
    ..onLoadEnd.listen((e) => requestComplete(httpRequest))
    ..send('');
}
```



Sending the request

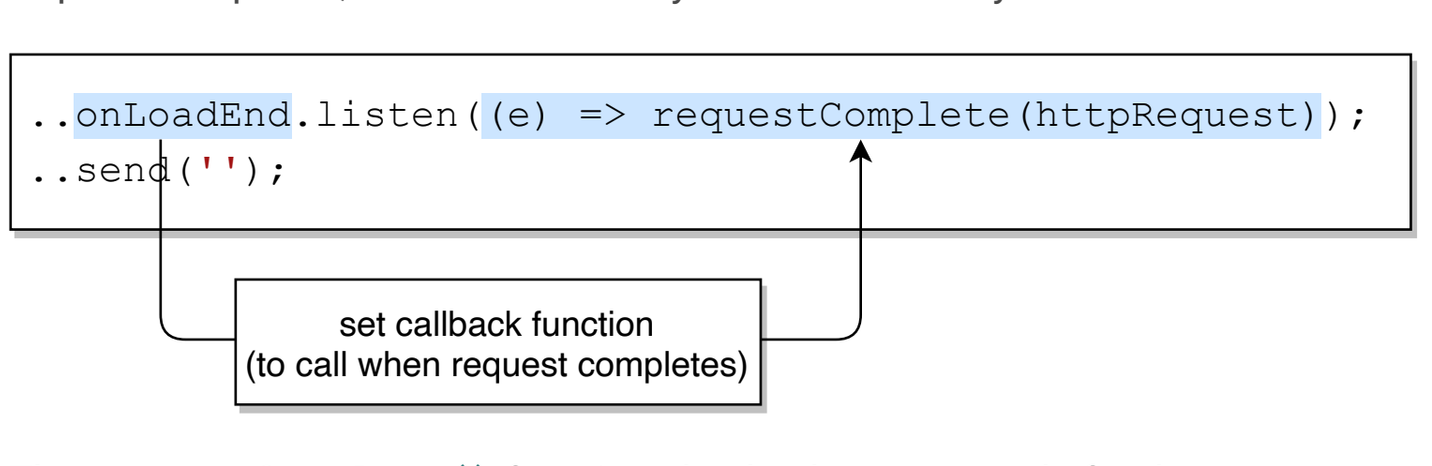
The send() method sends the request to the server.

```
httpRequest.send('');
```

Because the request in this example is a simple GET request, the code can send an empty string. For other types of requests, such as POST requests, this string can contain relevant data. You can also configure the HttpRequest object by setting various header parameters using the setRequestHeader() method.

Handling the response

To handle the HTTP response, you need to set up a callback function before calling `send()`. Our example sets up a one-line callback function for onLoadEnd events that in turn calls `requestComplete()`. This callback function is called when the request completes, either successfully or unsuccessfully.



The `requestComplete()` function checks the status code for the request.

```
void requestComplete(HttpRequest request) {
  if (request.status == 200) {
    final response = request.responseText;
    if (response != null) {
      processResponse(response);
      return;
    }
  }

  // The GET request failed. Handle the error.
  // ···
}
```

If the status code is 200, the file was found and loaded successfully. The content of the requested file (`portmanteaux.json`) is returned in the responseText property of an HttpRequest object.

Populating the UI from JSON

The data file in the portmanteaux example, `portmanteaux.json`, contains the following JSON-formatted list of strings:

```
[
  "portmanteau", "fantabulous", "spork", "smog",
  "spanglish", "gerrymander", "turducken", "stagflation",
  "bromance", "freeware", "oxbridge", "palimony", "netiquette",
  "brunch", "blog", "chortle", "Hassenpfeffer", "Schnitzelbank"
]
```

Upon request, the server reads the file and sends it as a single string to the client program.

Using json.decode(), the app easily converts the JSON-formatted list of words to a Dart list of strings, creates a new LIElement for each one, and adds it to the `<ul>` element on the page.

```
void processResponse(String jsonString) {
  for (final portmanteau in json.decode(jsonString) as List<dynamic>) {
    wordList.children.add(LIElement()..text = portmanteau as String);
  }
}
```

Other resources

- Using JSON
- Asynchronous programming: futures, async, await