Data-driven Fixes

This documentation is preliminary. Suggestions for improvements are welcome.

Data-driven fixes is a feature that allows package authors to provide data about changes that have been made to the package's public API and uses that data to help users of the package update their code when updating to a newer version of the package.

The data is used by two tools:

- The analysis server uses it to provide quick fixes (also known as code actions) in the IDE.
- The `dart fix` command uses it to apply bulk edits to the code in a file or directory.

The purpose of this document is to describe how to write and test the data used by these tools. It contains four sections:

- Overview provides an overview of how this feature is intended to be used and what kind of data can be expressed.
- Examples provides examples of how to express some common kinds of changes.
- Reference provides a complete definition of the data and how that data is used to update client code.
- Testing provides a guide for how this data can be tested.

Overview

The data used by this feature describes the changes made to the public API of a package. It's contained in a data file in the package that defines the API.

The data is organized around the individual elements of the API, where an element is anything that can be referenced outside the library in which it's declared. This includes top-level declarations, such as classes and extensions, and their members, such as methods and fields.

The changes to an element that can be described are designed to be both fine-grained and high-level.

By "fine-grained" we mean that the overall change to an API is described in terms of small, composable changes. For example, instead of describing all of the ways in which a method's signature has changed, there are individual changes for adding and removing individual parameters, renaming parameters, etc. This reduces the complexity of describing the overall change.

By "high-level" we mean that the changes are described in terms of how the API changed, not in terms of what needs to be done to client code in response to those changes. For example, if a parameter is added to a method the data would describe the parameter and the value to be used in invocations, but doesn't describe how to find invocations or how to find overrides. It's the responsibility of the tools to figure out how to modify the client code in response to those changes. This reduces the amount of work required of package authors initially and allows the tools to adjust as the Dart language evolves, removing the need for package authors to refine the data.

The description of an element and the changes to that element are encapsulated in a transform and the data file is essentially a list of transforms.

The changes are applied when an analysis of the client code produces diagnostics. For example, if a method is renamed then invocations of the old method will produce a diagnostic indicating that the method isn't defined. This also works if the old method is still declared and marked as being deprecated because a diagnostic is also produced in that case. Because of that, we encourage you to add data for an API when the API is first deprecated. If the API doesn't go through a deprecation period, then the changes should be added to the data when the changes are made.

Examples

This section provides examples of how to express some common kinds of changes.

Each example includes the API being changed, both the old API marked as deprecated and the new API that replaced it. As noted in the Overview, it isn't necessary for the old API to still be declared in order for the tools to be able to apply the changes, but it's included in the examples for clarity.

As a result of always including both the old and new APIs in the examples, the elements are always renamed. It isn't necessary to rename an element in order for the tools to be able to apply the changes.

Rename a method

One of the most common API changes is to rename an element. In this section we'll show how to specify that a method has been renamed. Let's assume that your package defines a class `C` in the file `lib/c.dart` with a renamed method similar to the following:

```
class C {
  @deprecated
  int oldName(String s) => newName(s);

  int newName(String s) { ... }
}
```

In the data file we need to specify which method was renamed, what it's old name was, and what it's new name is. We do that by writing a data file like the following:

```
version: 1
transforms:
  - title: 'Rename to newName'
    date: 2020-11-20
    element:
      uris: ['c.dart']
      method: 'oldName'
      inClass: 'C'
    changes:
      - kind: 'rename'
        newName: 'newName'
```

This tells the tools that any references to the method `C.oldName` should be updated to refer to the method `C.newName`. That includes both invocations of the method as well as references where the method is being torn off. It also tells the tool that any methods that used to override the old method need to be renamed so that they now override the new method.

Reference

This section provides a complete definition of what data can be included in a data file and how that data is used to update client code.

In the following subsections, the text `...` is a placeholder for a value that is described in a different subsection.

data file

A *data file* contains the data used to update clients of a package when the public API of the package has been changed. The data is represented by a single transform set.

The data is stored in a file named `fix_data.yaml` in the `lib` directory of the package. There is a single data file per package.

You might find it useful to include a comment at the beginning of the file that has a link to this documentation for easy reference.

transform set

A *transform set* describes a set of changes made to the API of a single package. It's represented as a map with two keys: `version` and `transforms`.

The `version` key is required. The value of the `version` key is an integer used to identify the version of the file's content. The version described by this document is version `1`. The version key is typically the first line of the file.

The version key lets the tooling continue to work with older versions of data files when the format of the file needs to be changed, and allows older versions of the SDK determine when a package is using a newer and not understood format.

The `transforms` key is required. The value of the `transforms` key is a list of transform objects. While the list is allowed to be empty, there's no value in having a data file unless there's at least one transform.

For example, the basic content of a data file generally looks like this:

```
version: 1
transforms:
  - ...
  - ...
```

The transforms can be in any order, but to make it easier to maintain the file we recommend putting newer changes at the beginning of the file, just like you would in a change log. You might find it useful to use comments to group sets of transforms, possibly based on the published version of the package in which the changes were made.

transform

A *transform* describes a set of changes made to a single element in the API. It's represented as a map with seven keys: `title`, `date`, `bulkApply`, `element`, `oneOf`, `changes`, and `variables`.

The `title` key is required. The value of the `title` key is a string that is displayed in the IDE to describe the changes that will be made to the user's code. It's usually shown as a label in a menu, so it should be kept short. And because most APIs can be referenced in multiple ways, the description should be general enough to cover all of those cases. For example, if a method is renamed then the name will need to be updated both at invocation sites and anywhere the old method was being overridden. Using a title like "Invoke newMethod" wouldn't be appropriate for updating an override, so a better title would be "Rename to newMethod", which would work for both cases.

The `date` key is required. The value of the `date` key is a string encoding of a date. The format of the string is required to be the same as the format accepted by the method `DateTime.parse`.

The `bulkApply` key is optional. The value of the `bulkApply` key is a Boolean value indicating whether the transform should be applied when bulk fixes are being made. The default value is `true`. You need to disallow a

transform from being used in a bulk fix tool in cases where there are multiple transforms for a single API. If there's a clear default, then you can allow that one transform to be applied.

The `element` key is required. The value of the `element` key is an element object. It specifies the element in the API that was changed, such as a class or a method.

Either the `oneOf` key or the `changes` key is required, but it isn't valid to have both. These keys are two different ways to specify the list of changes that will be applied when a reference to the element is found.

The value of the `oneOf` key is a list of conditional change objects. For each reference to the element that's found, the first conditional change whose condition is true will determine the list of changes to be applied to that reference. If more than one conditional change has a condition that's true, only the first one will be applied. If none of the conditions is true, then no changes will be applied.

The value of the `changes` key is a list of change objects. The list of changes is unconditionally selected as the changes that will be applied.

The `variables` key is optional.The value of the `variables` key is a variable map.

For example, you can specify a transform like this:

```
title: 'Descriptive title'
date: 2020-09-14
element: ...
changes:
  - ...
  - ...
```

You might find it useful to include a comment to provide more information about each transform, such as design documents, issues, or even PRs that motivated or implemented the change.

element

An *element* describes a single element in the API that was changed, such as a class or a method. It's represented as a map with either two or three keys depending on the kind of element being described. All elements have a `uris` key and a second key that specifies the type and name of the element, such as `class` or `method`. Elements that are members of a top-level element have a third key specifying the containing element.

The `uris` key is required. The value of the `uris` key is a list of uris, each of which is the URI of a public library that includes the element (or the container of the element in the case of members such as methods).

The second key, which is required, indicates the kind and name of the element. The kind is given by the key itself and the name is given as the value of the second key, which is a string. The possible keys, and hence kinds, are:

- class
- constant (from an enum)
- constructor
- enum
- extension
- field
- function (top-level)
- getter
- method

- mixin
- setter
- typedef
- variable (top-level)

If the element is a member of a top-level element, then the third key is required and it indicates the kind and name of the top-level element containing the member. The possible container keys are:

- inClass
- inEnum
- inExtension
- inMixin

For example, you can specify a class named `C`:

```
uris: ['lib.dart']
class: 'C'
```

or specify a method named `m` in a class named `C`:

```
uris: ['lib.dart']
method: 'm'
inClass: 'C'
```

To specify the unnamed constructor in a class, use an empty string for the name of the constructor:

```
uris: ['lib.dart']
constructor: ''
inClass: 'C'
```

Despite the fact that the names of named parameters are used by clients to associate an argument with the parameter, they aren't considered separate API elements and therefore can't be described by an element object. If you have modified the parameters of a method or function, then the element that was changed is the method or function associated with the parameter.

conditional change

A *conditional change* is a list of changes that is applied only when a condition is true. It's represented as a map with two keys: `if` and `changes`.

The `if` key is required. The value of the `if` key is a condition. Any variables declared for the whole transform can be referenced in the condition.

The `changes` key is required. The value of the `changes` key is a list of change objects.

condition

A *condition* is a Boolean-valued expression. It's represented by a string that uses a subset of Dart's expression syntax to express the condition. The currently supported syntax is:

```
<condition> ::= <logicalAndExpression>
```

```
<logicalAndExpression> ::= <equalityExpression> ('&&' <equalityExpression>)*

<equalityExpression> ::= <primary> (<equalityOperator> <primary>)?

<equalityOperator> ::= '==' | '!='

<primary> ::= <variableName> | <stringLiteral>
```

where `<variableName>` is the name of a variable from a variable map and `<stringLiteral>` is a single-line string literal with single quote delimiters.

change

A *change* describes a single change that was made to the element. It's represented as a map. The number and names of the keys depends on the kind of change being described, but all changes have a `kind` key.

The `kind` key is required. The value of the `kind` key is a string indicating the kind of change. The valid kinds are:

- addParameter
- addTypeParameter
- removeParameter
- rename
- renameParameter

Individual kinds of changes are described in the section below, titled by the kind.

addParameter

An *add parameter* change indicates that a parameter was added to a function or method. It has five keys: `kind`, `index`, `name`, `style` and `argumentValue`.

The `index` key is required. The value of the `index` key is the zero-based index of the parameter after all the changes related to parameters have been applied. It's used to know where to add new parameters in overrides and, if the parameter is a positional parameter, where to add new arguments at invocation sites.

The `name` key is required. The value of the `name` key is the name of the parameter. It's used as the name of new parameters added in overrides and, if the parameter is a named parameter, as the name of the argument at invocation sites.

The `style` key is required. The value of the `style` key is one of the following strings: `'optional_positional'`, `'required_positional'`, `'optional_named'`, `'required_named'`.

The `defaultValue` key is required if the added parameter is an optional parameter whose type is non-nullable. The value of the `defaultValue` key is a code template object whose value will be used when adding the parameter to any methods that override the element.

The `argumentValue` key is required if the added parameter is a required parameter or is an optional positional parameter that was added before at least one pre-existing optional positional parameter. The value of the `argumentValue` key is a code template object whose value will be used as the value of the corresponding argument at invocation sites.

```
kind: 'addParameter'
index: 0
name: 'a'
```

```
style: optional_positional
argumentValue: ...
```

addTypeParameter

An *addTypeParameter* change indicates that an element was given a new type parameter. It has five keys: `kind`, `index`, `name`, `extends` and `argumentValue`.

The `index` key is required. The value of the `index` key is an integer indicating the index of the new type parameter after all the changes related to type parameters have been applied. It's used to know where to add new type parameters in overrides and where to add new type arguments in references to the element.

The `name` key is required. The value of the `name` key is a string containing the name of the type parameter. It's used as the name of new type parameters added in overrides.

The `extends` key is required if the type parameter has an extends clause. The value of the `extends` key is a code template object whose value will be used in the `extends` clause of new type parameters added in overrides.

The `argumentValue` key is required. The value of the `argumentValue` key is a code template object whose value will be used as the type argument when updating references to the element.

```
kind: 'addTypeParameter'
index: 0
name: 'T'
extends: ...
argumentValue: ...
```

removeParameter

A *remove parameter* change indicates that one of the parameters of a function or method was removed. It has three keys: `kind`, `index`, and `name`.

The `index` key is required unless the `name` key is provided, and isn't allowed if a `name` key is provided. The `index` key specifies that a positional parameter was removed and its value is the zero-based index of the parameter before the function was modified. It's used to know which parameters to remove in overrides and which arguments to remove at invocation sites.

For example, if the second positional parameter is removed you would write:

```
kind: 'removeParameter'
index: 1
```

The `name` key is required unless the `index` key is provided, and isn't allowed if an `index` key is provided. The `name` key specifies that a named parameter was removed and its value is the name of the parameter before the function was modified. It's used to know which parameters to remove in overrides and which arguments to remove at invocation sites.

For example, if the named parameter `p` is removed you would write:

```
kind: 'removeParameter'
name: 'p'
```

rename

A *rename* change indicates that an element was renamed. It has two keys: `kind` and `newName`.

The `newName` key is required. The value of the `newName` key is a string containing the new name of the element. It's used to replace the old name of the element (provided in the element object) in references to the element.

For example, to rename an element to `B` you would write:

```
kind: 'rename'
newName: 'B'
```

This change is only intended to support simple renames, such as renaming a class or a method. It therefore assumes that the new element is like the old element in several ways:

- it's in the same library (that is, the URI's by which it's imported don't change),
- it's the same kind of element (for example, you can't rename a class and change it to be a mixin using this change), and
- if the element being renamed is a member (such as a method), then the renamed element is a member of the same container.

If the change that was made doesn't fit within those constraints, then using a rename to capture it might not work in all situations. You should consider using a replacedBy change instead.

renameParameter

A *renameParameter* change indicates that a named parameter in a function or method was renamed. As such, the element in the transform must be a method or function. It has three keys: `kind`, `oldName`, and `newName`.

The `oldName` key is required. The value of the `oldName` key is a string containing the old name of the parameter. It's used to locate the parameter that was renamed.

The `newName` key is required. The value of the `newName` key is a string containing the new name of the parameter. It's used to replace the old name of the parameter, both in overrides of a method and in references to the method or function.

For example, to rename the parameter `a` to `b` you would write:

```
kind: 'renameParameter'
oldName: 'a'
newName: 'b'
```

replacedBy

A *replacedBy* change indicates that the specified element was replaced by another element. It has two keys: `kind` and `newElement`.

The `newElement` key is required. The value of the `newElement` key is an element representing the element that replaces the specified element.

For example, to replace a top level variable `v` with a static field `C.f` you would write:

```
kind: 'replacedBy'
newElement:
  uris: ['lib.dart']
  field: 'f'
  inClass: 'C'
```

This change currently has two limitations that you should be aware of.

First, it doesn't allow the list of URIs for the new element to be different from the list for the old element. That means that it currently might not correctly update the imports to make the new element visible in scope.

Second, it doesn't support replacing every kind of element, nor can the replacement element be every possible kind. What it does support is the following cases:

- Replacing one constructor with a different constructor, even when the constructors are in different classes. It doesn't correctly handle the case of replacing a `const` constructor with a non-`const` constructor.
- Replacing a function (either a top-level function or a static method) with a different function.
- Replacing a getter (either a top-level getter, a top-level variable, a static getter, or a static field) with a different getter.
- Replacing a setter (either a top-level setter, a top-level non-`final` variable, a static setter, or a non-`final` static field) with a different setter.

As with constructors, if a static member is being replaced by another static member, the members can be in different containers (classes, mixins, or extensions).

There isn't currently any way for the tool to detect whether a method, getter, or setter is static, so it's possible to specify an instance member. As a result, specifying an instance member (as either the replaced or replacing element) might produce invalid changes for your users.

Similarly, there isn't currently any way for the tool to detect whether a top-level variable or a static field is `final` (which includes `const`). As a result, specifying a final element (as either the replaced or replacing element) might produce invalid changes for your users.

code template

A *code template* describes code that can be generated when fixing user code. It's represented as a map with three keys: `expression`, `requiredIf`, and `variables`.

The `expression` key is required. The value of the key is a template: a string containing Dart code, possibly with embedded references to variables. Variables are referenced by enclosing the name of the variable between `{%` and `%}` delimiters.

The `requiredIf` key is optional, but is only allowed if the code template is inside an `addParameter` change and the style of the parameter is `'optional_named'`. The value of the `requiredIf` key is a condition that will be evaluated for each invocation of the method or function that was changed. Normally arguments are not added for optional named parameters, but the argument *will* be added if the `requiredIf` key is provided and the condition evaluates to `true`.

The `variables` key is optional. The value of the `variables` key is a variable map. If neither the `expression` template nor the `requiredIf` condition use any variables then the `variables` key can be omitted.

Variables serve two purposes. The first, and most obvious, is to allow the replacement text to contain values based on the context in which it will appear. For example, if a parameter was removed then it's possible that the value of the corresponding argument will need to appear somewhere else in the updated code.

The second purpose is to allow the tool to automate some of the changes for you. For example, if the replacement text needs to refer to an imported name, you should use a variable to do so, and the definition of the variable can specify where the name is imported from. The tool can then determine whether any changes need to be made to the imports in the library being edited and make them correctly.

The variables that can be referenced in the template are the variables defined in the code template and any variables defined in the enclosing transform whose name isn't hidden by a variable defined in the code template.

```
expression: '{% type %}({% arg %})'
variables:
  arg: ...
  type: ...
```

variable map

A *variable map* defines one or more variables. It's represented as a map. The keys of the map are variable names and the values are value objects.

The name of a variable is any identifier containing either uppercase or lowercase letters.

value

A *value* describes a way of computing a string that can be injected into a template. It's represented as a map. The number and names of the keys depends on the kind of value being described, but all value objects have a `kind` key.

The `kind` key is required. The value of the `kind` key is a string indicating the kind of value. The valid kinds are:

- fragment
- import

Individual kinds of value objects are described in the sections below, titled by the kind.

fragment

A *fragment* value indicates that the value is a fragment copied from the code being updated. It has two keys: `kind` and `value`.

The `value` key is required. The `value` key specifies the fragment of code to be copied and is expressed as a chain of references in a simplified AST model of the code. The details follow the example.

```
kind: fragment
value: 'arguments[0]'
```

The code fragment is described by a dot separated non-empty list of accessors:

```
<fragment> ::= <accessor> ('.' <accessor>)*
```

Each accessor consists of an identifier followed by an index operator:

```
<accessor> ::= <identifier> '[' (<integer> | <identifier>) ']'
```

When an accessor is evaluated it takes a target node and either produces a result node or fails. The target node of the first accessor is always the node representing the invocation being updated (whether it's the invocation of a function, method, getter, setter or constructor). The target node of all other accessors is the result node of the preceding accessor.

There are currently two supported identifiers: `arguments` and `typeArguments`.

The `arguments` accessor takes any node that includes an argument list and returns one of the arguments to the invocation. The index operator selects the argument to be returned. A zero-based integer is used to select a positional argument and an identifier is used to select a named argument.

The `typeArguments` accessor takes any node that includes a type argument list and returns one of the type arguments. The index operator selects the argument to be returned using a zero-based integer.

Here are some examples:

- `arguments[0]` copies the code for the first argument in the argument list.
- `arguments[child]` copies the code for the argument named `child`.
- `arguments[0].typeArguments[0]` copies the code for the first type argument in the expression that is the first argument in the argument list.

import

An *import* value indicates that the value is a top-level declaration from a library. It has three keys: `kind`, `uris` and `name`.

The `uris` key is required. The value of the `uris` key is a list of uris, each of which is the URI of a public library from which the name can be imported. They will be used to add an `import` directive, if one is needed, to any library in which the value of the code template will be used. If there are multiple URIs in the list and none of them are imported into the library being modified, then the first URI will be used.

The `name` key is required. The `name` key specifies the name of the top-level declaration and its value is a string containing that name. The name is also the value of the variable. Any use of the variable in the template will ensure that the library is imported into the library being changed and that the name is visible.

```
kind: import
uris: [ 'package:flutter/material.dart' ]
name: 'Widget'
```

uri

A *uri* is a string containing a URI. The URI can be one of the following:

- a `dart:` URI,
- a `package:` URI, or
- an abbreviated URI.

An abbreviated URI is the portion of the URI following the name of the package containing the data file. For example, if a package named `sample` contains a library named `sample.dart` in the root of the `lib` directory, then within the data file for the package `sample` the URI for that library can be written as either `package:sample/sample.dart` or `sample.dart`.

test folder

A *test folder* contains dart files and their corresponding golden master `.expect` files that are used to test the data in the data file. These files are located in a folder (conventionally named `test_fixes`) in the package directory. See the Testing section for more documentation.

You might find it useful to include a `README.md` file that has a link to this documentation for easy reference.

Testing

This section provides a guide for testing data driven fixes.

All the test files are contained in the test folder. Every dart file has a golden counterpart file, which is named as `<dart-file-name>.dart.expect`.

Every test has two parts:

1. All possible usages of the public API that is changed by the data driven fix. This is contained in the dart file.
2. All the usages in the dart file with the expected change applied. This is contained in the golden file.

Extending on the example shown in Rename a method, the change can be tested as below:

```
// test_fixes/C.dart
import 'package:<package-name>/C.dart';

C.oldName('Fix me'); // usage before the change.
```

```
// test_fixes/C.dart.expect
import 'package:<package-name>/C.dart';

C.newName('Fix me'); // expected usage after the change.
```

To run these tests locally, execute the below command in the test folder.

```
dart fix --compare-to-golden
```