Google Summer of Code 2021

Dart

Standalone Pub Server

Table Of Contents

Introduction 3

Project Description 3

Project Deliverables 3

Implementation 4

Server 4

Shelf 5

Client 7

Mustache 7

SaSS/SCSS 8

Webdev 9

Pub Tool 10

Database Considerations 11

Testing 11

Sample Code 12

Open Source Contributions 13

Community Contributions 14

My Journey with Flutter and why I chose Dart 15

Timeline 16

Post GSoC and Future Work 17

Commitment 17

Personal Information 17

Introduction

Project Description

Flutter and dart have a wonderful ecosystem for managing packages which allows easy use of dependencies to build projects. The *pub.dev* website is wonderful and feature-rich, hosting more than 15000 open source packages. It also provides detailed statistics through metrics like score, popularity, and likes which helps the community to use and support the best packages developed. This open nature is an amazing notion, but from a business view, might not be best suited for organizations wanting to use proprietary code as packages, as pub does not provide a way to add dependencies that are not hosted on pub.dev, although pubsepc.yaml supports specifying git URLs as a dependency, it has its limitations as versions cannot be locked or repositories cannot be private. This brings a need for a self-hosted private pub repository on which organizations can host their private packages and use them directly in their Flutter/Dart projects without any hassle, natively supported by the pub tool.

Project Deliverables

The general idea of the project will be to develop a server using Dart which can host and serve pub packages and integrates with the pub tool.

★ Design and develop a standalone server using the dart server stack.
★ Use **shelf**, **shelf_router**, and HTML **templating** to aid in development.
★ Add functionality to display the list of uploaded packages.
★ Add functionality to facilitate REST API calls to the server.
★ Modify the **pub tool** to allow authentication to third-party servers.
★ Write unit and integration tests to ensure it is fail-safe.
★ Document the process to set up or modify the server by the user.
★ Write blogs to create awareness among the community for best use.

Implementation

Server

Dart is a wonderful language with great type safety and the recently released null-safety features which makes it less prone to make errors during development. Dart has made using Flutter a delight to develop with, because of benefits such as hot reload. But, thanks to the dart team, dart can also be used as a server-side language and can also compile to javascript to manipulate *DOM* elements using the **dart2js** runtime.

This project is aimed at developing a server that can host pub packages and that can be self-hosted by organizations. The **dart: io** package allows us to create a basic HTTP server for listening to requests.

```dart
Future main() async {
  var server = await HttpServer.bind(
    InternetAddress.loopbackIPv4,
    4040,
  );
  print('Listening on localhost:${server.port}');

  await for (HttpRequest request in server) {
    request.response.write('Hello, world!');
    await request.response.close();
  }
}
```

Although HttpServer class can be used because, as the application size grows, it becomes more difficult to scale and respond to requests. A handler can be declared which will compare the method, URI, and query parameters using conditional statements which might introduce redundant and less efficient code.

An alternative is to use package:**http_server** which provides helper methods to use along with **dart:io. This** provides slightly less boilerplate syntax, and the VirtualDirectory class helps easily serve files without actually needing to transform the file and send it on requests.

```
File targetFile = File('web/index.html');

Future main() async {
  VirtualDirectory staticFiles = VirtualDirectory('.');

  var serverRequests =
      await HttpServer.bind(InternetAddress.loopbackIPv4, 4046);
  await for (var request in serverRequests) {
    staticFiles.serveFile(targetFile, request);
  }
}
```

Shelf

The shelf is a dart package by the dart team, shelf is an alternative to the above-mentioned methods and makes it much easier to implement a web server by providing higher levels of abstraction.. We can accomplish this project by using some basic concepts of shelf.

We can use shelf_io to spin up a server.

```

var server = await io.serve(handler, _hostname, port);
print('Serving at http://${server.address.host}:${server.port}');
```

Shelf uses the concept of Handlers, **Handler** is just a function that accepts a **Request** object and returns a **Response** object.

We can use **Pipeline class** to construct a layer of Middlewares. **Middlewares** sit in between requests and help transform each request to accomplish a certain task, like *Logging* requests or adding *CORS* headers.

Cascade class helps accomplish a similar function but with **Handlers** instead. We can chain multiple Handlers and perform a required action depending on the incoming route, which we can access by using *request.url.path* property.

```
final chandler = shelf.Cascade().add((request) {
    if (request.url.path == 'hello') {
      return shelf.Response.ok('Helo World');
    } else {
      return shelf.Response.notFound('Error Not Fund');
    }
  }).handler;

  var handler = const shelf.Pipeline()
      .addMiddleware(shelf.logRequests())
      .addHandler(chandler);
```

The package:**shelf_static** can be used to create a static file handler to serve static files like CSS, images, and JSON from the server.

```
final staticHandler = createStaticHandler('public');
```

Although shelf makes it easier to implement a server, it's still hard making API endpoints with HTTP methods, **shelf_router** makes this easier to implement by providing express.js like endpoint functions. These functions support URL paths and query parameters, which makes it easy to query an endpoint with dynamic data.

```
router.get('/', (Request request) {
      return Response.ok("Server is up"),
          headers: {'Content-Type': 'application/json'});
    });
    router.post('/<id>', (Request request, String id) async {
      return Response.ok("Request Successful $id",
          headers: {'Content-Type': 'application/json'});
    });
```

Client

A client displays the content that the server sends, this can be either served directly from the server or can be hosted as a standalone client.

Mustache

Mustache is a popular templating engine for serving HTML content from the server. This can be easily used to serve dynamic content. Mustache provides an easy-to-use syntax. It takes in a key-value pair, and the key to be displayed is placed within double curly braces. For example,

```
// key-value pair
{
user: John
},
// html
<div>
    {{user}}
</div>
```

We can display lists using a slightly different syntax.

```
// key-value pair
{
"users": ["Hans", "Fritz", "Geraldine"]
}
// html
{{#users}}
<div>
    {{•}}
</div>
{{/users}}
```

Mustache templates can be customized to strict or lenient syntax.

SaSS/SCSS

Although HTML can be used to display content on the client-side, without any styling, it may look very dull.. CSS is a great approach for this, but if the project gets complex, it can get very tricky quickly to manage code.
Sass/SCSS helps in making CSS maintainable by providing a handful of features that do not exist in CSS.

Features like **variables, inheritance, nesting and mixins** are some great features that can increase productivity. Sass just transpiles to CSS, so there are no additional settings to a project. Sass files can be compiled using the **sass** command and specifying the input and output files or folders. It can also watch files using the *--watch* flag. With **Dart sass command line**, the sass executable invoked using the *sass* command.
Below is a sample SaSS style file.

```
$font-stack:    Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

Webdev

In case there are considerations to enable interactions on the client, for example, similar to pub.dev, a like button for metric purposes (although not a requirement.), **package:webdev** can be used to manipulate the DOM elements.
dart:html can be used to access all of the DOM API methods like querySelector(), LIElement(), and its children.

```
DivElement pacakges = querySelector('#packages');

void listPackages(String json) {
  for (final p in jsonDecode(json)) {
    pacakges.children.add(DivElement()
      ..text = p['name']
      ..className = 'package'
      ..children.add(ParagraphElement()..text = p['version']));
  }
```

Pub Tool

The pub tool provides an amazing interface to download packages from pub.dev to the pub cache to use in a flutter or dart project. But the problem is that the pub tool only authenticates ***pub.dev.*** An environment variable **PUB_HOSTED_URL** can be used to fetch from a URL other than pub, there is still a lack of third-party server authentication.
This can be a stretch goal, but some initial ideas for approaching this are

Using a key in pubspec.yaml

A custom key-value can be set to point to the third-party server for authentication in the *pubepc.yaml* This can allow pub to know where to upload the package to, since there is already similar functionality in pub publish.

```
1
2  name: cherry
3  description: A new Dart pacakge
4  version: 0.0.1
5  homepage: null
6  publish_to: https://mypubdev.com
```

Use CLI login

If not for the above method, a command can be used to log in to obtain an OAuth token that can be stored in the **.PUB_CACHE,** similar to how pub saves credentials in *credentials.json*.. A sample command would be

dart pub login <server-url> --token <secret-token>

The token can be generated as a url and validated in the browser.

Database Considerations

To store the versioning, and package information, a database should be considered. Currently, pub.dev stores all the contents on the Google cloud platform with Redi and cloud storage. Implementing a more open storage solution would help organizations choose their own cloud provider to host the server on..

Redis - Redis is a powerfully fast, in-memory data store. Redis stores key-value pairs and objects and are schemaless with no definition to store values.
Because of its nature, Redis can be used to fetch values quickly.
Data can be loaded from a database and stored in Redis for quick access.

Appwrite- Appwrite is an open-source firebase alternative with features like database, functions, storage, and other useful services that can be self-hosted on any cloud provider. Appwrite storage can be used to host the package ***tar*** files.
Appwrite recently announced **dart sdk** for Appwrite which provides support for using Appwrite on dart servers,

Alternatively, we could make an ***abstraction*** for users to use *GCS, S3,* or other storage solutions where **tarballs** can be stored, with a single **JSON** file to store all information if the upload requests are minimum, or can be stored as a folder per package structure.

Testing

Testing is an important aspect of any software project and is helpful in making the application fail-safe under multiple circumstances of changing code.
We can use Dart's test suite for writing tests. We can implement unit tests and integration tests for handlers and routes making sure each handler or route returns the appropriate data and handles errors.

```dart
HttpServer server;
String baseUrl;
Router app;

setUp(() async {
  app = Router();
  app.mount('/api/', Packages().router);
  server = await serve(app, 'localhost', 8083);
  baseUrl = 'http://${server.address.host}:${server.port}';
});

test('Return Method Forbidden if Header does not contain Admin', () async {
  final client = HttpClient();
  final url = Uri.parse('$baseUrl/api/packages/provider');
  final request = await client.deleteUrl(url);
  final response = await request.close();
  expect(response.statusCode, HttpStatus.forbidden);
});
```

Sample Code

Exploring the Dart server stack has been a really fun experiment. Coming from a Node.js background, the features that Dart provides for server-side programming are really amazing and just display how flexible Dart is.

The sample repository mainly consists 3 projects,

1. **Shelf_example** - this project showcases the usage of ***shelf*** along with Cascade and Pipeline with handlers for rendering different HTML templates using Mustache engine and SaSS. It is a compilation of all Flutter conferences in one place.
2. **Shelf_router_example** - this project mimics the pub API for fetching packages, it implements appropriate routes using ***shelf_router*** package and responds with JSON data from a file using the hosted pub repository specification.
3. **Webdev_example** - is a simple standalone client built using the ***webdev*** package, it makes api calls to the above project and displays a list of packages by manipulating DOM elements.

Open Source Contributions

Flutter and dart have been a great learning journey, the dart team has been very welcoming and supportive of contributions, and the open nature of it makes it truly incredible. Below listed are some of the contributions I got to make over the past few months.

Flutter/DartContributions

dart-lang/site-www
PR- link

dart-lang/pub-dev
PR - link
issues - link

flutter/plugins
PR - link

flutter/flutter
PR - link
Issues - link

flutter/website
PR - link

Other Contributions

FilledStacks/stacked
PR- link

izhour/auto_size_text_field
PR - link

macbrey/clay_containers
PR - link

Jaguar_dart/observable_ish
PR - link

rknell/FlutteEnumsToString
PR - link

Community Contributions

The ***Flutter community*** is one of the best tech communities I have ever come across on **Twitter**. I try my best to ***evangelize*** and guide newcomers into the community, especially during the ***#30DaysofFlutter*** duration before the release of **Flutter 2.** I was a runner-up during the **Flutter Vikings** content Build Viking by stream.io and featured on their blog. I was also featured on the **official Flutter Twitter handle** for participating in **#AdoptAwidget**

I currently have **three** packages *published* on **pub.dev** and love the dart ecosystem and how easy it is to publish packages. My latest package was a *Dart CLI app* that generates files according to architecture, here's my **Publisher page**.

Stackoverflow has helped me a lot in my flutter development journey. I regularly contribute to the flutter tag on StackOverflow, here's my **StackOverflow profile**,

I often write on **Medium** about flutter and submit articles to the Flutter community publication, currently, I have **three** articles, here's my **Medium blog**.

I have also conducted multiple sessions on Flutter in the **Google DSC** chapter of my college *DSC KSSEM,* which is *a* local community. My most recent community talk was at **Flutter Bangalore** on **State management with Riverpod**.

Additionally, I'm a mentor at **Girlscript Summer of code**, where I am helping students contribute to a ***flutter favorite package***, **AnimatedTextKit**, and **ForestTreeTagging**.

I've built a lot of **sample apps** and full-blown apps on various different concepts such as API services and *state management techniques* on my **Github**.

My Journey with Flutter and why I chose Dart

I adopted Flutter during its first *stable release*, and since then, it has been an amazing development experience. Flutter is the reason for the person I am today as it is the first development framework I started using, and within a few months I was able to bag an *internship at a startup* and I'm grateful for all the opportunities it has provided me.

I'm currently a *final-year undergrad student* and a really passionate software developer. Since I've started Flutter, I have got the opportunity to work part-time for a startup for most of last year where I helped them accomplish multiple project goals.
I also got to *intern* at a second startup where I was able to improve the codebase to follow the most optimal practices. I recently got an opportunity to work as a consultant for a startup showing them all the reasons to use *Flutter* and helping them understand the goal. Some of the apps I helped build during my time at these companies were **photocopy.ai**, **headstrt**.

I'm currently also the **Google Developer Student Clubs lead** for my college chapter **DSC KSSEM**, along with my core team I have helped hundreds of students learn Google technologies and contribute to open source, and the best events have been on ***Flutter study jams*** where I teach my juniors about flutter with regular sessions. I'm currently also an **MLH Fellow** in the **MLH Fellowship program** powered by **Github**, which is a *mentorship* program similar to GSoC, I'm helping other fellows get to know ***Flutter and Dart*** better by giving **talks** and including it in projects we build.

Coming from a *Java* background, **Dart** has been the most convenient and type-safe language to work with providing amazing developer productivity. The interoperability features it provides from different languages are just fantastic.
I hope to lead this and show organizations how easy and good it is to use Dart on the server, and how dart would increase their productivity as a type and null safe language.

Timeline

Until May 17

- Learn more about the Dart Server ecosystem.
- Contribute to dart-lang organization to better understand the contribution process.

May 17 - June 7

- Interact and get to know the Mentor and their availability due to different time zones.
- Understand the goals and expectations of the organization for this project.
- Understand the complete workflow of the project, and code conventions to be followed.
- Discuss and finalize initial goals, end goals, and stretch goals for the project.
- Create a basic design of the entire architecture of the server.
- Finalize tools and dependencies required for the project.

June 7 - July 12

- Start and finish the basic server with all API routes and functionalities.
- Develop the client side UI with styling using css, and ensure responsive in UI
- Write tests and ensure edge cases are handled properly.
- Accomplish all initial and end goals.
- Prepare for first evaluations.

July 16 - August 16

- Start documenting any remaining API elements and developer docs for the project.
- Start focusing on stretch goals and implementing them.
- Document the entire process through blogs on the Dart medium publication.
- Ensure all project goals are met and prepare for final evaluations.

Post GSoC and Future Work

After completing all the features of the server application, I'd continue to contribute to the **dart-lang** organization and try to build the dart server ecosystem better. I primarily use ***Typescript*** and ***GraphQL*** for writing backend applications, and I'd like to bring this support to Dart as well. I'd also like to build ORM(Object Relational Mapping) packages so the community can use them to build more complex applications. I'd also keep contributing to the ***Flutter framework*** as I am primarily a Flutter developer and would like to keep helping improve the framework I love.

Commitment

I'd like to specify that I will be free during the Summer for the duration of the GSoC period and will have **NO** other obligations or commitments with any other form of work. I will be able to fulfill the stipulated **175 hours, 10-week** duration without any conflicts in my schedule.

Personal Information

Email :
Resume :
Socials : LinkedIn, Twitter, Github, StackOverflow, Website
Location :
Timezone :