Volley

Architectural Improvement Essay

Gabrielle Palado

Proposal

The purpose of Volley was to perfect a library that utilizes the framework of caching response data. This achievement has resulted in the improvement of HTTP request and response efficiency. I believe that the most beneficial architectural change that can be applied to this library would be one that makes it easier for developers to make additions to the existing framework. Doing so would subsequently make it easier for the framework to be improved.

My architectural change looked to separate request-handling logic and response-handling logic. In the existing implementation, both were being handled within a single class: Request. While this might be acceptable in the early stages of a small library, it can very quickly become a problem when the implementation gets more complicated. An example of this would be the ImageRequest class - the majority of the methods were related to handling the response rather than the request. Should the logic regarding the request itself get more complicated, the class would be filled with logic that relates to not one purpose, but two.

Having methods with different purposes but living in the same class can be misleading for a developer, and very difficult to disentangle when more complicated logic is implemented. Splitting the existing request class logic into request and response-handling components makes it more semantically understandable for the developer as to where logic belongs.

Description

The main principle I applied to the architecture of the request class was "*composition over inheritance*". The idea behind this principle is if one class handles multiple functionalities, split those functionalities into components and make the class "own" the components it requires. This way, one class can perform several functionalities but the logic for those functionalities doesn't necessarily belong within that class. Instead, the class delegates that work to the corresponding component.

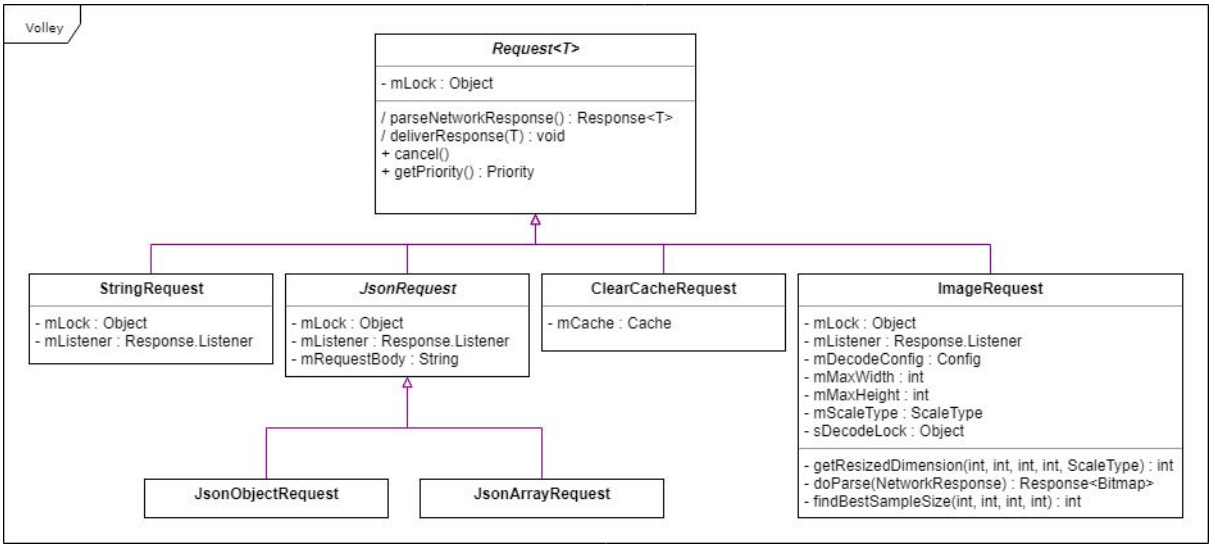

Figure 1: Original Request Structure

Note that the above class diagram omits some fields less-relevant to my changes. The main characteristics to note here is that `Request` is an abstract class with abstract methods `parseNetworkResponse` and `deliverResponse`. These methods are overridden by the children - the implementations of which were the essentially same for three of the concrete classes, but with different types (i.e. String, JsonObject, etc.) involved. In addition to this, three of the concrete classes also overrode `Request#cancel` in order to nullify the listener waiting for the response.

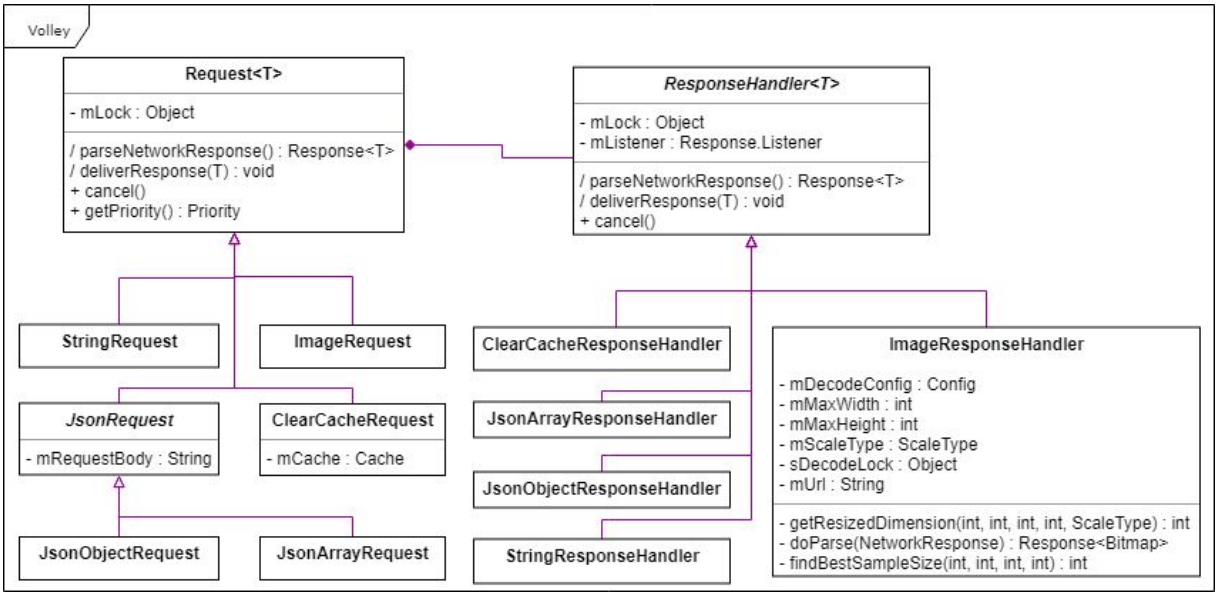

Figure 2: New Request and ResponseHandler Structure

To make this change, I made Request a concrete class - the two abstract methods, parseNetworkResponse and deliverResponse, delegate the work to a ResponseHandler component passed into the constructor. This ResponseHandler not only houses the logic for these operations, it also handles the cancellation of the listener waiting for the response to be parsed. This operation is invoked by the cancel method within the Request class if the ResponseHandler is present. When moving methods and fields to belong in the ResponseHandler, I evaluated whether the logic was involved in the processing of the Request or if it was involved in the processing of the Response.

The delegation of response operations to the ResponseHandler reduced the Request children classes to constructors injecting the relevant ResponseHandler concrete type into the super constructor. It also reduced the repetition of response delivery and listener cancellation logic to being written once in the base ResponseHandler class with generics.

Evaluation

The main concern I wanted to address with this architectural change was the understandability of the library. Should Request and/or Response logic get more complicated, it is easy to see where the logic belongs when having a Request that has a ResponseHandler component. Separating the Request object into two makes more semantic sense than having one object that contains logic for both “operations”.

This architectural change would make it easier for future users of the library to understand the logic flow involved in the creation and processing of Requests, and subsequently make it easier to extend. It has also improved the reusability of the request-handling logic in the sense that, should different request types handle parsing or delivering in such a way that only differs in type, the different Requests can utilize the same ResponseHandler but with different generics specified.