Spain

Blog Post
Author: Werner Donné

Asynchronous User Interfaces

SWORD
UPGRADE YOUR BUSINESS

We have reactive technology for clients and servers, where basically data flows in only one direction. The “happy path” is efficient, while the unhappy one doesn’t have to be, because it occurs less frequently. We pride ourselves that we can do all of that asynchronously and in a non-blocking way. This maximises resource usage, because nothing has to wait so there is a lot less idle time. We stay clear of request/response scenarios, since that implies waiting.

NASA Cassini mission

There is, however, one kind of resource we seem to forget too often and that is the human resource, one of the most expensive kinds. Most applications are still completely synchronous. You perform an action for which a remote service is addressed and then you have to wait for the result. As developers, we need to provide super short latencies in order to satisfy the impatient user. Even services that work completely asynchronously need to expose some fake synchronous API, because user interfaces expect that.

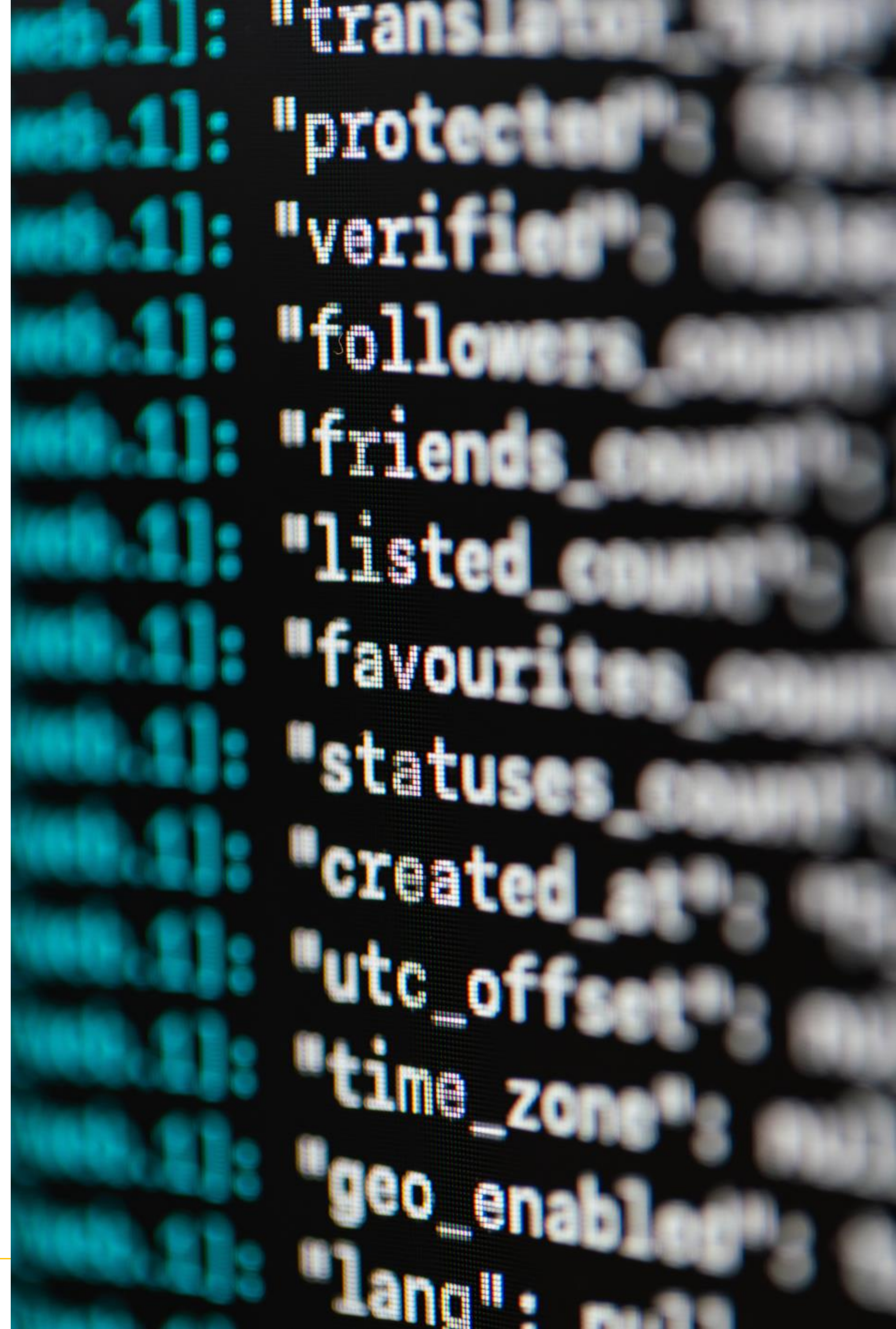
However, we can apply the same reactive principles to the user experience. Requests and responses will have to be decoupled. When sending a request to a service the response will be merely the confirmation it has been received well. Actual processing occurs at some later point, but will be immediate in most cases. The response is pushed to the client when it is available. In the meantime the user just continues to work. This scenario assumes the “happy path” is the most likely outcome. It is certainly the case for the underlying technology. Is there a reason to assume otherwise for the user? Or would they make mistakes all the time?

In the vast majority of cases there is no reason to wait for the outcome of a request, because it is very likely to be fine. In order for the user to feel confident about this there needs to be a very discrete feedback mechanism. It could be something like a traffic light. When the dot is green everything is in order. When there are pending requests it is orange. When the orange dot flashes something is taking longer than expected and when there is a problem it is red. Clicking on the dot would reveal the list of pending requests or errors. Clicking through on an error, in turn, would bring the user back to the user interface context where the data was entered. As a consequence, requests and the context in which they occur should always be correlated.

Imagine how this would work for intensive data entry. Users could enter data in forms and submit them without looking, as in the good old mainframe days. The latency of a roundtrip becomes less critical, because the user expects everything to succeed or to be notified a bit later when it doesn't. This resembles how we work in real life. When you have a list of task to do for which some kind of feedback is need, then you don't wait for the feedback of one task before starting with the next.

There are a few technical constraints to make this work. An HTTP POST will always return status code 202 (Accepted), which means the request has been received well, but will be processed later. It is important that a request is always stored durably in some messaging system before being processed. You don't want to lose the user's work. So the status code 202 acknowledges that fact.

We also need a mechanism to push responses back to the client. Here Server-Sent Events are an interesting option. In a reactive client this doesn't require a lot of change. The messages sent by services can be dispatched internally in the same way the response of an Ajax-call would be.



Websockets are an alternative, because they provide two-way communication where sending and receiving are decoupled. However, in my opinion Server-Sent Events are a better match, because requests and responses are also physically decoupled. For services that wish to push messages this is easier. The push channel can be something completely different than the connection management needed for accepting requests. With websockets you need a reliable long lasting connection from the Internet down to the actual process that deals with the communication. This may involve API gateways, load-balancers, etc. It is often a challenge. An additional advantage of Server-Sent Events is that requests can be issued over plain old HTTP.

Another technical challenge is state management in the client. When the user is entering data for a part of a larger entity, that entity may be updated through the response channel, either because of a previous command or changes coming from another user. When this happens the data store in the client shouldn’t be simply replaced, otherwise the user might lose work. Instead the part the user is changing should be merged with the incoming data. When there are conflicts this should be shown clearly.

This touches on the broader problem of concurrency control. Often the last save wins method is employed, but this is not very user-friendly. There are two ways to improve this. If the data can be easily merged, all changes can happen concurrently without loss of someone’s data. When there are conflicts they should be shown right in the part of the UI corresponds to it. The other option is to provide some check-out mechanism where a piece of data is reserved for a user. Merging and conflict resolution provide the most fluid user experience. It also goes well with an event driven design, because then changes could come from anywhere, not only from other users.

So we don’t just need non-blocking calls at the technical level. We also need a non-blocking user experience. This implies a different approach for developers, but also a re-education of the users, so they no longer think in terms of requests and immediate responses.