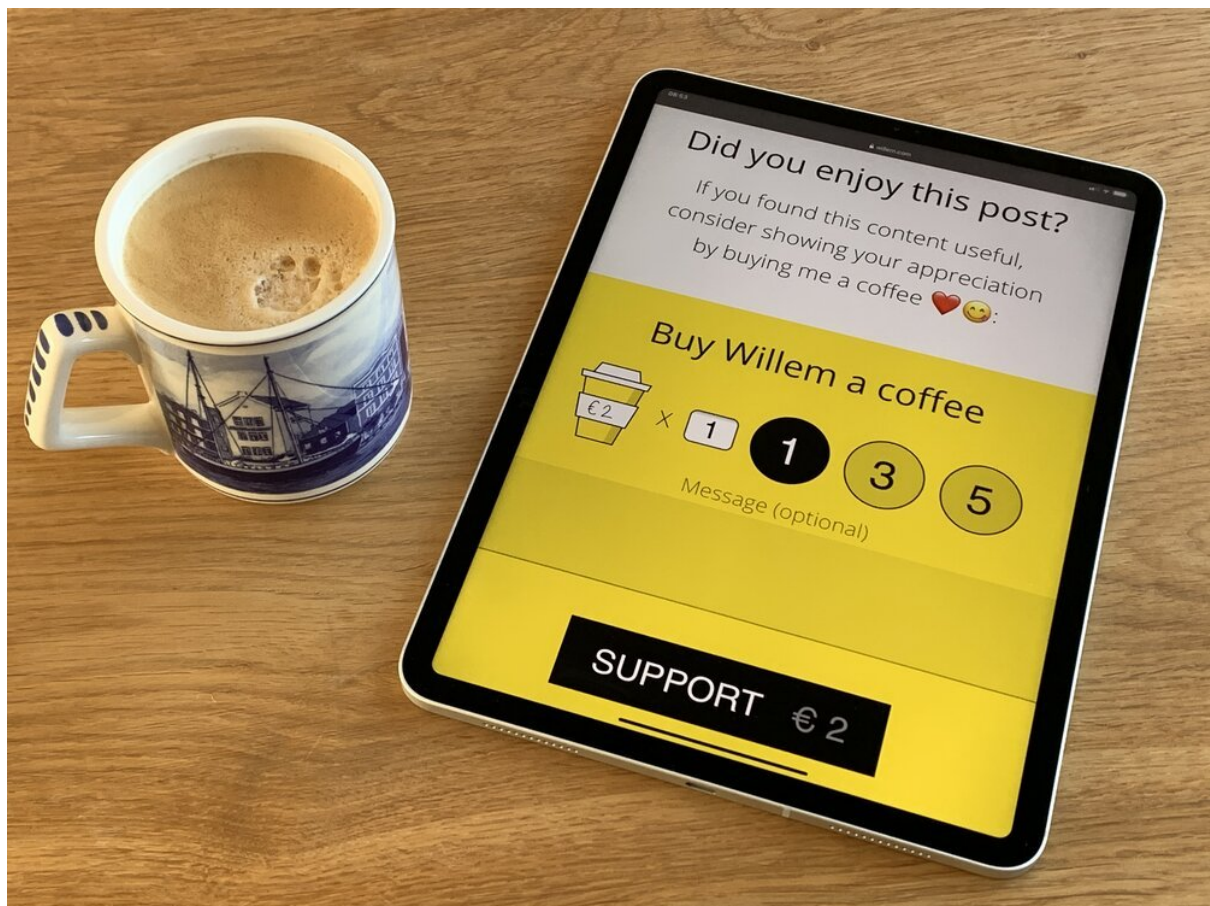


Designing and implementing a (micro) payment system

Monetising my blog with coffee, Apple Pay and Mollie

Willem L. Middelkoop

Mar. 25, 2020



Online payments are now more important than ever as businesses are disrupted by the COVID-19 virus. It drives my customers to seek new ways to make money online. I designed and implemented a (micro)payment system. This post is about achieving simplicity by solving complex challenges.

Understanding new technology

When developing new technology and apps I always try to imagine how it would work best. The easiest way to find out, is to design the product for yourself. Just like Steve Jobs had Apple develop their presentation app, Keynote, for himself. It is one of his famous jokes about [him being an underpaid beta tester](#)



Steve Jobs announcing the Keynote app at MacWorld 2003, joking that it was build for himself, him being an underpaid beta tester

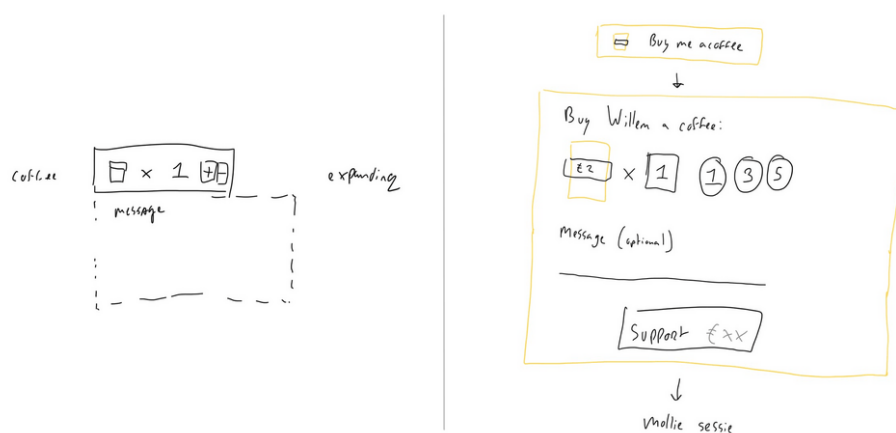
If you design the product for yourself, think about what you would really, *really*, want. For me, this means the most utterly simple user interaction model possible: I want my software to be simple, easy.

Use case: buy me coffee

For the purpose of developing the payment technology, I came up with the use case of adding a "buy me coffee" button to my website. If it works well, people can use it to show their appreciation for this blog by buying me a coffee. I think this use case is a perfect challenge to design (and build) a (micro) payment system.

Sketching ideas

Before programming a single line of code, I often design software by sketching ideas by hand. I find the creative process of drawing helpful in exploring ideas, layouts, processes. It is one of the reasons why I like working on a tablet so much, digitally sketching ideas allows me to quickly explore multiple ideas by copying/pasting them into multiple iterations.



Interface concept sketches, exploring different options to leave a message and buying multiple cups of coffee

Prototyping

Once you have explored ideas for your user interface, it is time to develop a prototype. When you build a prototype you can test the user interface on different devices, see how it looks and feels.



Translating the design sketch (left) into a prototype (right) on iPad

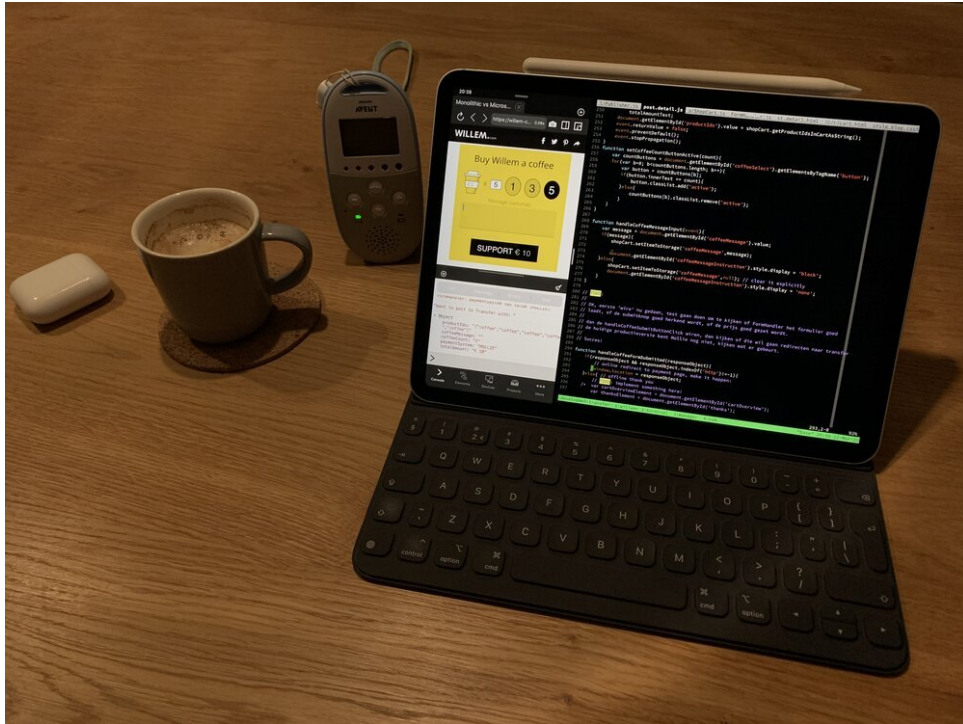
During prototyping I often look at the design sketch while programming the first version. On iPad you can do this using split screen, where I place the design sketch right next to my code, using [VIM](#) in [tmux](#) over [Mosh](#) with the brilliant [Blink app for iPadOS](#). I created the coffee cup using [Picta Graphic for iPad](#), a beautiful drawing app that exports to vector images (SVG).



Designing the coffee cup using Picta Graphic for iPad

Frontend development

The next step is to develop a fully working user interface, making the buttons work. During development you often need to figure out what is going on, track bugs and finetune mechanisms. I use the [Inspect Browser](#) for iPadOS to have a working miniature touch screen interface right next to my code. It allows me to rapidly test and perfect the working user interface.

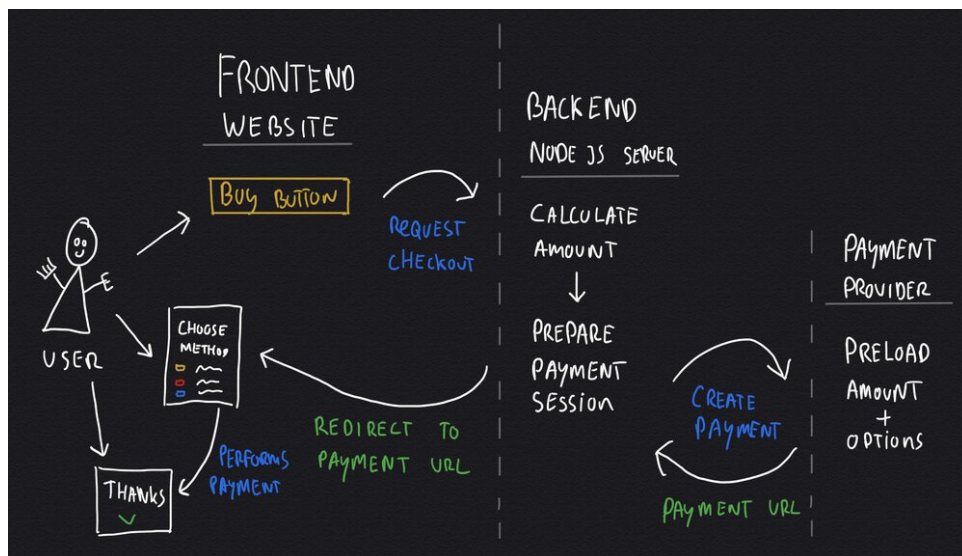


Developing the user interface using Blink and Inspect Browser on iPad

Backend development

Another crucial step is to develop the backend technology to prepare and track online payments. This is server side technology that will handle the communication between the frontend user interface (website) and payment provider (banks, credit cards, Apple Pay, etc).

Designing the backend server is usually a little more abstract compared to designing a (visible) frontend user interface. I find sketching UML sequence diagrams very useful when figuring out what API calls should happen in what order. [Sequence diagrams](#) visualise program interactions in time sequence.



Sketched sequence diagram of payment process

You see three columns in the payment process sequence diagram: frontend, backend and payment provider. Each column represents a different computer that communicates with the others. The frontend runs on the user's device, the backend server runs on a VPS managed by me and the payment provider is an external server. The user interacts with the website by clicking the "buy (me coffee) button", this triggers a chain of API calls that flow from right to left.

The "backend server" is responsible for calculating the amount that the user needs to pay. It is calculated on the server, not in the client, to prevent malicious users to manipulate the price calculation (as frontend technology can be manipulated through the web browser).

A payment session is prepared by informing the payment provider that the user will make a payment for a given amount. This process uses a secret API key that allows the payment provider to know to which bank account payments should go. The payment provider returns a payment URL, that is used to redirect the user to the actual payment page.




Mollie is a payment provider that supports many different payment methods, including credit cards, PayPal, iDEAL, Sofort, Giropay, SEPA, various bank apps and Apple Pay

The payment provider I use is [Mollie](#), an [Amsterdam based](#) company with a fantastic payment platform. Their API's are [well documented](#) and they provide [packages](#) to easily implement their payment system into your backend server.

```

1  const payment = await mollieClient.payments.create({
2    amount: {
3      currency: 'EUR',
4      value: '10.00',
5    },
6    method: 'creditcard',
7    description: 'My first payment',
8    redirectUrl: 'https://shop.org/order/12345/',
9    webhookUrl: 'https://shop.org/payments/webhook/',
10   });
11
12   // Redirect the consumer to `payment.getPaymentUrl()`.
13   payment.getPaymentUrl()

```

	Amount	Status	Details
	€ 10,00	PAID	10022

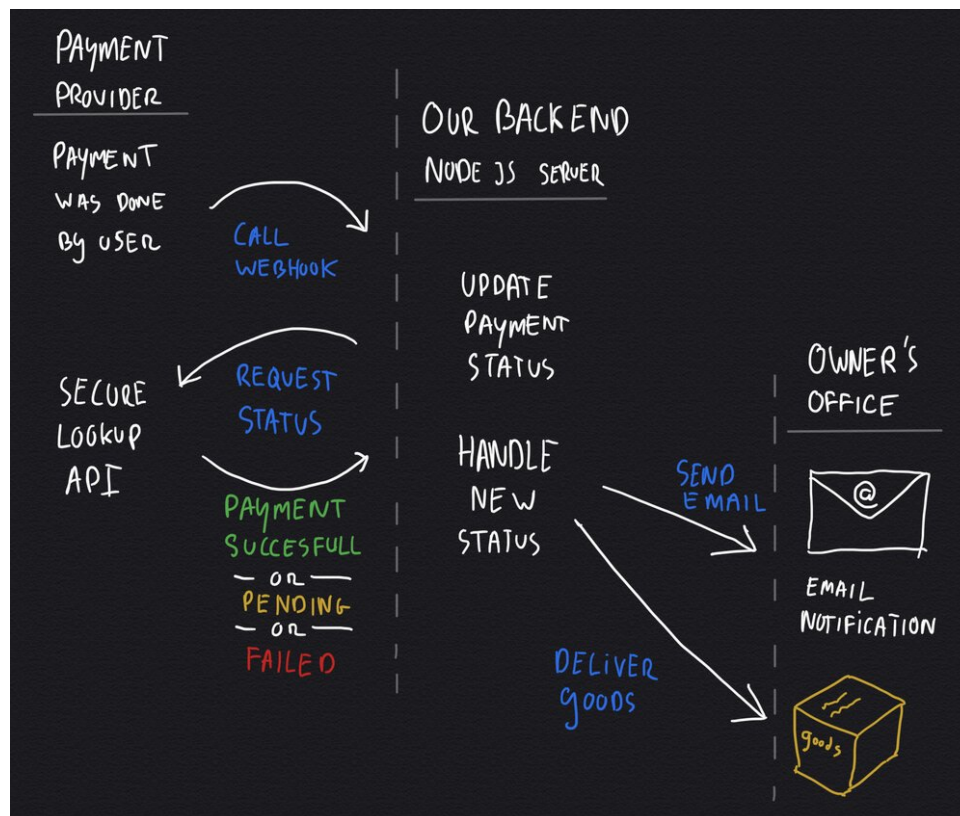
Calling Mollie from NodeJS using promise based JavaScript

In harmony with my other backend servers, I choose to implement the backend server in [NodeJS](#). It runs on practically anything (from a tiny Raspberry PI, to large computing clouds). Alternatively you can use [PHP](#), [Ruby](#), [Python](#) or any of the other available [packages](#). The [Mollie API reference](#) offers detailed information on the values you can expect.

Asynchronous payment status updates

Few people realise that retrieving the payment status is an asynchronous process that can have different statuses. In a perfect world you instantly know whether a payment succeeds or fails. But due to the distributed nature of online payments where backends communicate with different servers (banks, credit cards, etc.) you never know precisely when a payment will complete.

Think of situations where a user closes the payment page, leaves his or her device unattended, or when a bank has an interruption with their online banking system. It happens, and it is your task to design the backend server to handle payment status updates separately from the primary user interaction process.



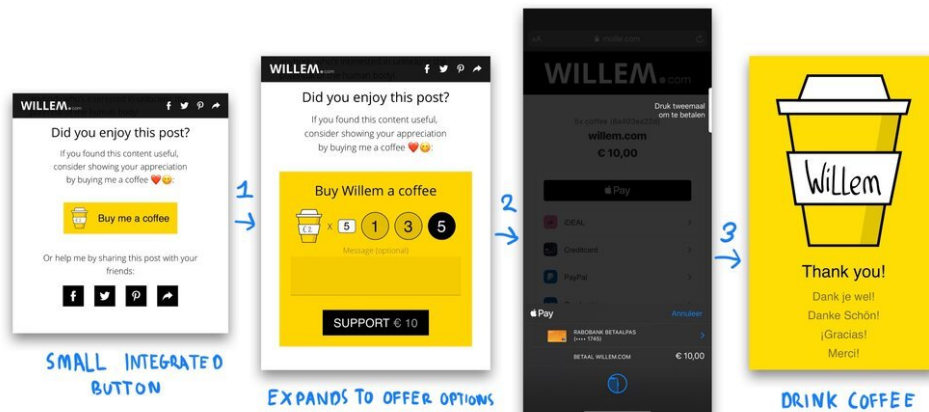
Handling payment status updates

The payment provider calls a webhook on our backend server. It delivers a message like "Look, something changed on payment XYZ". This message only says that *something* was changed, it does not tell us *what* the new status is. This is for security reasons: otherwise malicious users might be able to mimic the call to our webhook. Instead, our backend server requests the actual payment status through a secure channel using our secret API key. The payment provider returns the payment status which is then handled by our backend server. Depending on the payment status, appropriate actions are taken, like sending out email notifications, starting the delivery of goods.

Fullstack: Combining frontend and backend

The last step is to connect the frontend with the backend technology, making the entire process work. This is what people call "fullstack development" as you're working on the entire system. Fullstack development is a different kind of art that few developers truly master, it requires you to work (debug and develop) on multiple ends simultaneously; often in different programming languages at once!

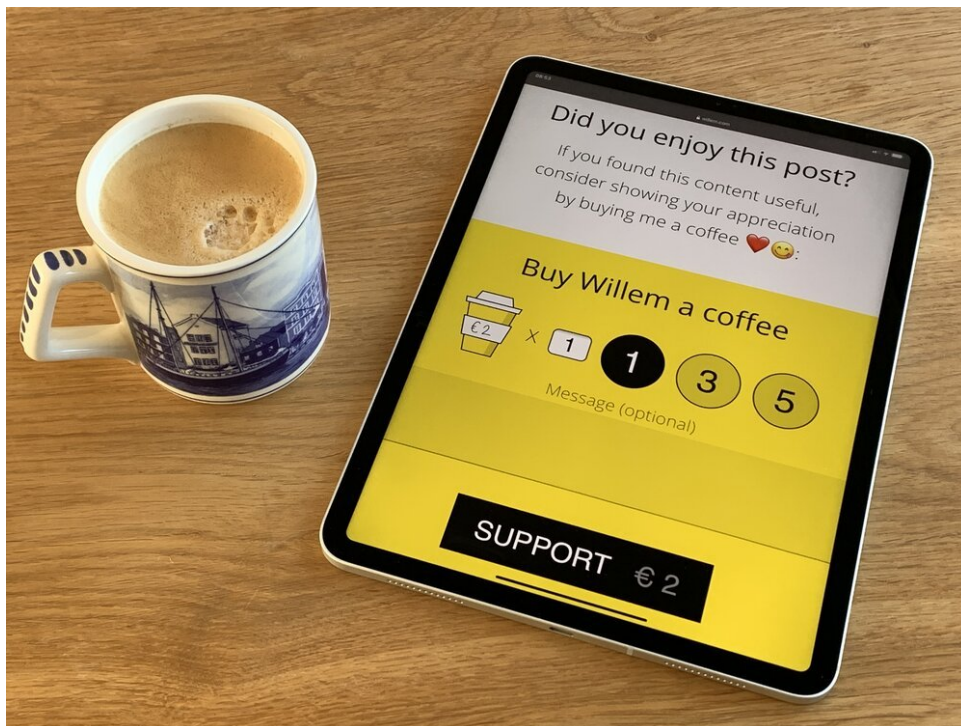
In practice this means that you'll be monitoring different parts of the programming, see if they work well together. Watching server logs while you try to perform a payment using the newly created frontend technology. It's hard, but it is also very rewarding as it results in a working product!



Connecting frontend with backend technology, simple as 1-2-3

The resulting "buy me a coffee" button is a deceptively simple 3 step process for the user:

- **step 1:** click a simple "buy me a coffee"-button, no complex order form, no irritating captcha code: just a single button
- **step 2:** the interface expands in place, preserving context and offering the option to leave a message or buy me multiple cups of coffee. In addition, other interface elements such as the social media buttons are automatically hidden to enhance focus on the coffee interface
- **step 3:** the payment provider screen allows the user to perform the payment using any available payment method, such as Apple Pay. When the payment succeeds, it's time to drink coffee!



Time to drink coffee

Conclusion

Designing and implementing a payment system is not that hard once you know what you need to do. Yet, it involves many steps to achieve that understanding. While my "buy-me-a-coffee"-button appears simple, it required solving many different design and programming challenges.

It's hard to overstate the importance of simplicity, yet it is often very hard to achieve because it requires an understanding of all aspects of the challenge. To quote Einstein: *"The definition of genius is taking the complex and making it simple."* Now, please feel free to try the button yourself: it's right here