🎬 Video Title:

Introducing TaskFlow: From struggling with HTML to deploying a Full-Stack Vanilla SPA with PHP, JWT and Redis

---

🎙️ TaskFlow: Video Transcript

🎯 The Journey

Hi, I am Angel Valentino, a self-taught software engineer. Just about a year ago, I was struggling to understand what the href attribute does in anchor tags and how to manage layouts using Flexbox. Now, I am introducing TaskFlow, a full stack productivity application that I built completely from scratch without relying on any frameworks or libraries.

The frontend is a single page application written in vanilla JavaScript, interacting with a RESTful API built with pure PHP, using JSON Web Tokens for authentication and Redis for rate limiting, with data stored in a separate MySQL database. Also supporting local storage for anonymous users.

However, I couldn't have done this by myself, I do not think anyone truly can. I relied heavily on documentation, YouTube, and any source of information I could find. Special thanks to Net Ninja, Dave Hollingworth, and dcode. They helped me get started and kept me moving forward. As well as the freecodecamp team, giving me the push I needed at the start.

💡 What is TaskFlow?

TaskFlow isn't just an app. It is the result of more than a year of relentless learning, debugging, late nights, and breakthroughs. It started as a simple portfolio project about a year ago but has evolved into much more. It is my idea of the perfect minimalist productivity application with no filler, just the necessary information, functionality, and widgets. Designed to help users stay focused, organized, and free from distractions.

It combines a clean task manager, a Pomodoro timer, daily motivational quotes, anonymous user task management with local storage, and even an enhanced task manager view to make managing your tasks even easier. All of this comes in a responsive and accessible user interface. It is mobile optimized, and even supports basic Progressive Web App functionality.

The main idea is to keep productivity simple: what you need to do and when, ordered by date; you complete it, you forget it.

Live App Demo

Everything catches up with the backend when you're online, and stores locally when you're not. All of this runs inside a custom single-page application, no frameworks. Let me quickly show you how it works:

Task Manager Overview

Here is the task manager, clean and simple. You can add a task, with the description being optional. I find this feature essential, because it is often frustrating when task management apps require descriptions, even though many tasks may not require one.

After adding a task, you can edit it at any time. Both the add and edit task forms are designed to help you preserve your data. For example, if you make any changes in the form and try to navigate away, a modal will appear asking if you want to keep your information. This helps prevent accidental data loss.

Once a task is added, you can mark it as complete or delete it as needed. The app also offers the option to delete all tasks, or to delete only active or completed tasks. Additionally, the manager includes smooth search functionality that allows you find any task by title. Matching search terms are highlighted to make it easier to find what you need. If there are no matches, an empty task placeholder is shown.

All of these features work seamlessly for both authenticated users, who benefit from persistent storage synced across the MySQL database, and anonymous users, whose tasks are saved locally in the browser storage.

Enhanced Task Manager Overview

All of the former functionality, except the task adding feature, is included in the enhanced task manager view, allowing users even easier task handling. I decided not to include the add task functionality here because I want users to see a quote before adding any task.

By enhancing their belief in what they can do, users can plan ahead even better rather than being just alone. Light must be found in the darkness, as the brightest day comes after the darkest night. I believe that to plan your future, you need a little help finding motivation. Even though it is still just a quote, it may be the beginning of a kingdom of ideas.

Motivational Quote Generator Overview

I strongly believe that the most important part of the application is the quote machine. As I mentioned before, everyone needs a little push when they are at their lowest; very few have the ability to truly get up on their own after falling.

But this is more than just a quote machine, it's tied into the theme change logic. With every new quote, you also get a new mood based on the theme that the laws of causality have chosen for you. Additionally, it allows users to share the quote.

While the UI is based on a FreeCodeCamp example project, all of the underlying logic, from theme management to sharing functionality, is entirely my own.

Pomodoro Timer Overview

And the last widget, a timer designed around the pomodoro technique. It gives the user two sessions to choose from: a short one with 25 minutes of work and 5 minutes of rest, and a long one with 45 minutes of work and 10 minutes of rest.

Both are fully accessible and come with the coolest sound I could ever find for changing sessions, stopping or resetting the current timer. They are logs falling though, who could have said that.

Architecture Overview

Let's break down the tech stack:

What is MVC and how it is used

The MVC architecture is divided into three core parts, each with a specific responsibility:

- **Model:** Handles data and business logic. This includes things such as fetching data, running complex validations, or interacting with a database. In the backend API, instead of directly using a model for simple CRUD operations, a gateway is used. This allows for a clean separation of responsibilities. If the app scales, a model layer can be introduced later to support more advanced data handling and logic.
- **View:** Focuses on rendering data to the users. In the frontend, views are responsible for displaying content. Note that the backend API has no concept of views since it only deals with data and responses.
- **Controller:** Acts as the glue between models and views. It performs basic validation, prepares data, and in the client, it also attaches event listeners to DOM elements defined in the view. Controllers ensure that interactions from the user are handled appropriately and the data flows correctly throughout the application.

Client overview

The frontend is a custom single-page application built entirely with vanilla JavaScript, following a modular Model View Controller architecture. The project is compiled using Webpack and Babel for performance, optimization and compatibility with older browsers. It includes custom implementations for routing and UI component handling. For secure communication with the backend, the client sends a Bearer token with each API request, allowing authenticated users to access protected resources and receive JSON responses.

Using views with custom vanilla SPA MVC architecture

Views are generated when a route is dispatched. Each view can contain various components, ranging from global ones to more specific ones. For example, the DashboardPage may include TaskManager, which is specific to the dashboard and stored

within the same folder. Additionally, views can incorporate logic for modals that use global components, such as a ConfirmModal.

Using components with custom vanilla SPA MVC architecture

In this setup, components are designed purely for rendering HTML and building the user interface. They are intentionally kept simple and focused, with all logic, data handling, and user interaction managed outside the component itself. The only dynamic behavior allowed inside a component is conditional rendering, such as displaying user-specific content based on authentication.

Components can be local or global depending on their purpose. For instance, the TaskManager component on the dashboard acts as the container for several subcomponents such as TaskManagerTabs, AddTaskPrompt, and SearchTaskPrompt. This modular design supports reuse, easy maintenance, and clear separation of concerns.

API overview

The backend is a RESTful API written in pure PHP following an MVC architecture mirroring the client. It handles JWT authentication, rate limiting with Redis, and secure refresh token flow. Redis is used to implement IP and device ID rate limits, as well as rotation detection, effectively blocking most brute force attempts and abuse. The database is a secure, normalized MySQL instance that is completely separated from the API and communicates via encrypted SSL.

Client and API Routing

The frontend router service handles navigation in the single-page application, ensuring only one view is active at a time. It maps routes such as /login, /register, and /tasks to their respective views and initializes the appropriate controllers and models. It also ensures smooth transitions by cleaning up lingering processes, such as aborting fetch requests, clearing intervals, and resetting modals.

On the backend, the API uses a simple vanilla router to decide which API request to handle and which controller should take care of it. Each route is linked to a specific controller that processes the request and decides what needs to be done, such as validating the data, managing the database, or sending back escaped JSON.

Key concepts and services

Services centralize shared logic such as API communication, authentication, and utility functions to keep controllers and models clean and focused. They enhance reusability, streamline resource handling, and help maintain a consistent user experience. On the backend, the API also relies on these services to handle authentication, error management, email notifications, rate limiting, and more.

⚔️🛡️💡 Biggest Challenges and Solutions

🐱‍👤 Modal Handler Class Manager

Among all the reusable logic and services I have built, one stands out: the Modal Handler.

This service manages the entire modal lifecycle, ensuring accessibility, focus management, and proper event handling. It includes features such as trapping focus, closing modals when the Escape key is pressed or when the overlay is clicked, and stacking modals to manage multiple layers by determining which modal is active based on its position in the stack. This class is used extensively throughout the client application to provide essential accessibility functionality in a reusable and consistent way.

What makes it truly special is that it is not limited to work with just modals. The logic is flexible enough to support prompts and other interactive components, even without an overlay. As well as integrating seamlessly with my custom client router.

But the part I am most proud of is that it was not built in a day, a week, or even a month. It evolved over the course of a year, starting as a basic utility function that managed minimal accessibility and ended up growing into a robust, battle-tested class that handles accessibility and the necessary ARIA attributes, modal stacking, modal patterns, precise event delegation, and proper cleanup of lingering event listeners attached to the body, all built around the singleton pattern.

🔒 Refresh Token Management

Handling refresh tokens was something I struggled with quite a bit. It is a tricky part to get right, and I am sure some bugs may still be lurking around. That said, I am pretty happy with how it turned out. Everything is properly connected, from logging out the client when the refresh token expires to handling API error responses.

For a regular user, using the API should never cause issues with refresh token management. It is fully integrated with a custom fetch handler that manages all fetch requests and authentication headers.

I also built a token handler that manages race conditions on the client side. When multiple requests need a token at the same time, only one request is sent to the refresh endpoint. Once the token is received, it is passed to all pending requests as well as any errors if needed. This way, both the client and backend tokens are refreshed together.

It is quite a complex functionality to get right, and I still don’t think it’s perfect, especially because the app uses asynchronous API calls and allows aborting requests to avoid mismatched UI. Although the client cancels the request, the backend still processes it because of the asynchronous approach. However, it’s a small price to pay for having such a modular architecture. WebSockets could have been used to let the server detect client cancellations in real time but that would have made the system much more complicated.

🌐 Deploying the API

By far the hardest part of this journey was deploying the API. It was easily the most challenging task in my engineering career. I had to go through a crash course on deployment, which ideally would take months to learn step by step, but I compressed it in about a week, working long hours. Nevertheless, did it. The API is now deployed on a hardened Linux server protected with SSH keys, Fail2Ban, and HTTPS.

One important lesson I learned is that I could have hosted both my API and database on a lower tier DigitalOcean droplet for a cheaper amount instead of using separate dedicated services for my API and database. However, my lack of experience led me to choose the improved approach, which was more expensive but also more modular and future-proof. So, keep that in mind.

Overall, deploying an API on a Linux droplet server is quite a straightforward process, once you do it once. Technology has evolved quite a lot. Tools like Certbot make managing SSL certificates easy, and DigitalOcean provides thorough documentation. I initially planned to deploy on Heroku, but their dynamic IPs made it impossible to whitelist my dedicated database. So, I chose a droplet server with a static IP instead. The only issue left was with email subscriptions.

At first, I thought of using Gmail, but most providers block common ports to prevent spam and abuse. After talking with support, they told me I could use port 2525 with SendGrid. However, I ended up using Mailgun using the limited free tier including one hundred emails per day. Although I have rate limits on endpoints, that email limit is quite low. Since I only send transactional emails such as password resets and welcome messages, it does not matter as much tough. And, If the app grows, it can easily be upgraded to a bigger plan.

🔒⚡✅ Security, Performance & Best Practices

The backend uses a lightweight, custom implementation of JSON Web Token, built in PHP. The token handling is managed through the JWTCodec and Auth service classes. The codec is based on the implementation by Dave Hollingworth, adapted to fit the specific needs of this project.

Security was a major focus throughout the project, with several measures in place including JSON Web Tokens, CORS protection, HTTPS, and SSL for the database connection. Rate limiting is implemented using Redis, with validation based on IP addresses and device IDs, as well as detecting any rotation of those keys. Additionally, prepared statements are used to prevent SQL injection, and all necessary JSON sent to the client is properly escaped. However, no application is completely secure. Ongoing vigilance and improvements are always necessary due to the evolving nature of cybersecurity.

The client and API are completely separated and have been thoroughly tested for CORS vulnerabilities, CSRF attacks, and include audit logging. I also achieved an A+ rating for SSL encryption from SSL Labs for my domain, ensuring strong security and encryption from client to database.

I’m also proud to say that TaskFlow scores a full score on Lighthouse audits for performance, accessibility, best practices and SEO, with only a slight drop in performance when logged in or using mobile devices.

Authentication Overview

The API secures its authenticated routes using Bearer Authentication with JSON Web Tokens. Including a separate token for password reset functionality.

Each token comes with a few important characteristics to maintain security. First, tokens are assigned a specific type, which helps prevent their misuse in unintended contexts. They also have a defined expiration time to limit how long they remain valid, reducing the risk of unauthorized use. Finally, all tokens are securely signed using a secret key on the server. This signature guarantees the integrity of the token’s data and prevents any tampering.

Authentication Flow

The authentication process begins with the user logging in by submitting their credentials. Once verified, the backend generates two tokens: an access token valid for 5 minutes, and a refresh token valid for 5 days. The client includes the access token in the Authorization header of all protected requests.

When the access token expires, the client sends the refresh token to the /refresh endpoint to obtain a new pair of access and refresh tokens, allowing the session to continue without requiring the user to log in again.

Logging out is handled by sending a request to the /logout endpoint with the current refresh token. The backend invalidates the token, and the client clears its stored ones, effectively ending the session.

For password recovery, if a user forgets their password, they can request a reset link. The system sends an email containing a secure JWT-based link that’s valid for 10 minutes, allowing the user to safely reset their password.

Lastly, it’s worth noting that the /quotes endpoint is the only route that doesn't require any authentication.

Why vanilla JS and pure PHP? Why not a Framework or Libraires?

I chose to avoid any frameworks on purpose to prove to myself that I could build a modern single-page application from the ground up. No React. No Laravel. Just clean JavaScript and PHP, a custom system that I built from scratch, and full control of every part of the UI logic and backend interactions. This forced me to truly understand how everything works, not just using someone’s else idea.

One major benefit of the architecture I followed for both the client and the API is modular design. Each component functions independently, making it easy to reuse or update its parts without breaking the rest of the application.

Another key advantage is separation of concerns. Instead of blending everything together, dynamic behavior is handled by services, controllers, models, gateways or views. While client components focus strictly on rendering the user interface.

Altogether, this makes the codebase easier to maintain and understand. The result is a clean and organized app where the UI is sharp and the logic is handled exactly where it should be.

🛤️ What I have learned

All started just as a simple portfolio project but ended up evolving into my current biggest project, the one I take the most pride in and that had the biggest impact on me. Evolving from a developer into a true engineer. I learned to think architecturally and understand how systems work from the ground up, including modern security standards.

One of the most surprising things was the sheer number of attempts to breach any public server, which is absolutely insane. The funniest part is that I did not realize this until about a week after deploying. Once I installed Fail2Ban, it blocked around 900 IPs within just a couple of minutes.

But I believe the most important lesson is that you become an engineer the moment you are comfortable with not knowing everything yet understand that it is your job to find a way to fix that gap. At the same time, you accept that this is a cycle that will never truly end.

🌬️ Possible Future Improvements and closing

Nonetheless, I am quite content with how the app turned out. Honestly, I wanted to keep going, but I was already past my scheduled deadline by over a week, and I believe I achieved most of what I set out to do.

Possible future improvements include refactoring the client-side code to TypeScript for strict typing, adding user management features such as account deletion and profile updates, improving Progressive Web App support, adding a theme picker for customization, and introducing a colorblind-friendly palette to enhance accessibility. These plans may evolve based on user feedback and priorities though.

If you are starting out or stuck on your first programming language, persistence alone is not enough. It comes from a deeper purpose that you must truly feel. Pain and struggle are essential parts of growth.

When I struggled with JavaScript, I hated the confusion but kept going. Without realizing it I was following the hero’s archetypical journey by facing fear and pain to find strength. Ancient stories and rites show that facing your darkest fears is how you become stronger and reborn.

Many give up, because sometimes you may never return from that darkness, but if you push through you will come out changed and tougher. So next time a snippet code drives you mad do not just keep going. Integrate the pain into your growth. This process shapes you not only as a developer but as a person and as a soul.

It is not easy, but once you face your fears head on, it gets easier each day. What you are fighting is more than just code. It is your mind confronting its deepest fears, the embodiment of death. Now you must face it.

Every line of code you write matters.
If I could build this in a year and a half, so can you.
Check out the GitHub repository for the full documentation and source code. Try it out, and feel free to reach out with any thoughts or feedback.

Thanks for watching, and welcome to TaskFlow.