PROJECT REPORT

Personal Engineering Project

Interactive Virtual Engineering Laboratory



EE Studio: An Interactive Virtual Engineering Laboratory

AUTHOR	**Asem Ameen Qaid**
DATE	**21 October 2026**
LIVE PLATFORM	**ee-studio-red.vercel.app**
SOURCE CODE	**github.com/AsemQaid/ee-studio**
SCOPE	**7 modules · 37 labs · 485 tests**

Undergraduate Program of Electrical Engineering

Faculty of Industrial Technology

Islamic University of Indonesia

2026

ABSTRACT

EE Studio is a web platform of interactive virtual laboratories that teaches electrical engineering by experiment rather than memorisation. A student moves a slider and watches a physically meaningful system respond: a mass settling on a spring, a transformer's secondary voltage sagging, a power grid approaching voltage collapse.

The platform covers seven modules (Control Systems, Digital Signal Processing, Power Systems, Electronic System Design, Renewable Energy, Machine Learning and Biomedical Systems) with 37 labs. Every lab pairs a numerically verified Python engine with a React interface, a physical visualisation, a theory panel and a real-world applications section. At its first complete milestone the project held 37 labs and 485 passing automated backend tests. The backend is deployed on Railway, the frontend on Vercel, and the source is hosted on GitHub.

Keywords: virtual laboratory, engineering education, power systems, control systems, signal processing, FastAPI, React, numerical verification

CONTENTS

1 Background and Motivation

2 Objectives

3 Theoretical Foundations

4 System Architecture

5 Development Methodology

6 Implementation

7 Testing and Verification

8 Challenges and Lessons Learned

9 Deployment

10 Results, Future Work and Conclusion

1 Background and Motivation

Electrical engineering concepts are abstract and are usually taught as equations and static plots. A student can recite that a second-order system with a damping ratio below one oscillates, yet never see the oscillation change as the damping is varied. Hands-on laboratories close that gap but depend on equipment, supervision and scheduled access. Professional simulators such as MATLAB/Simulink or SPICE are powerful but built for practitioners: they need installation, licences and a learning curve before a beginner can run a first experiment.

EE Studio was designed to sit between these two options: a browser-based virtual laboratory with no installation, in which each experiment is reduced to the few parameters that matter, the result is shown immediately, and the theory and real-world use are one scroll away (Figure 1).

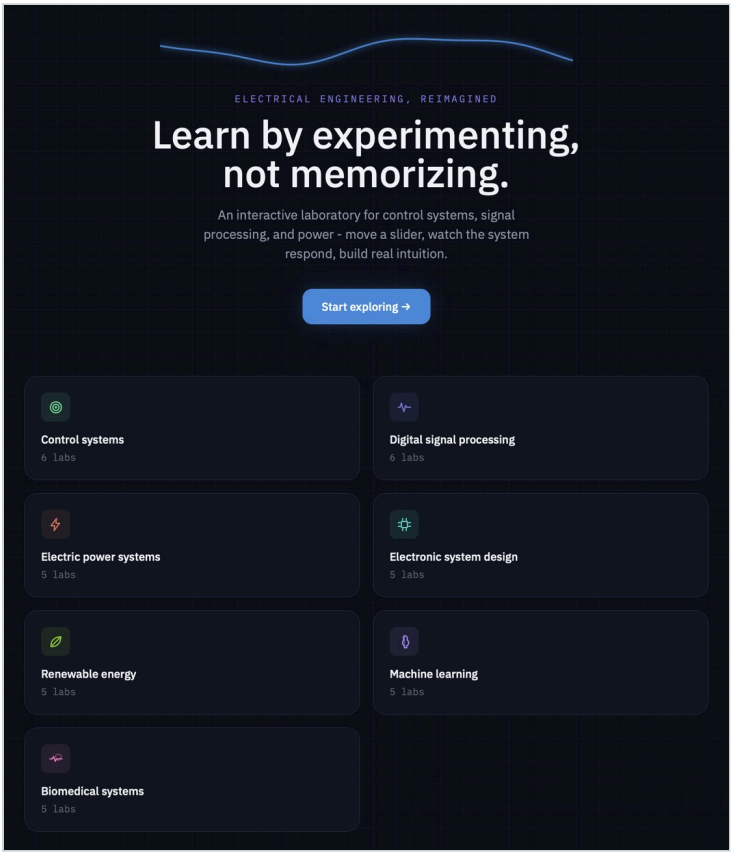


Figure 1. The live EE Studio landing page with the seven modules.

2 Objectives

2.1 Educational

- Turn each core concept into an experiment that a student can run in seconds, with immediate visual feedback.

- Make counterintuitive results visible: a leading power factor that raises voltage, a line that gains voltage at light load, a grid that stops having a solution under heavy load.
- Connect every lab to the real systems that use it.

2.2 Engineering

- Base every lab on a correct mathematical model, verified numerically before any code is written.
- Keep physical units, sign conventions and limiting cases (no load, full load, collapse) explicit and tested.

2.3 Technical

- Build a modular full-stack platform in which a new lab is added by following one repeatable pattern.
- Maintain one design system and a shared component library across all seven modules.
- Ship a tested, deployable product with automated tests on every engine and route.

3 Theoretical Foundations

Each module rests on a small set of established models. This section states the model behind each module and the result its labs are designed to make visible.

3.1 Control Systems

The module is built on the standard second-order system, whose damping ratio ζ and natural frequency ω_n determine its transient behaviour:

$$G(s) = \omega_n^2 / (s^2 + 2\zeta\omega_n s + \omega_n^2) \tag{1}$$

Below $\zeta = 1$ the response oscillates and overshoots; at $\zeta = 1$ it reaches the target fastest without overshoot; above 1 it is slower but never overshoots. The PID lab adds proportional, integral and derivative action and shows why derivative action amplifies noise. The Bode, Root Locus and Nyquist labs analyse the same system in the frequency and complex planes, and the State Space lab restates the mass–spring–damper equation $m\cdot x'' + b\cdot x' + k\cdot x = u$ as $x' = Ax + Bu$, with stability judged from the eigenvalues of A.

3.2 Digital Signal Processing

The module starts from the Nyquist–Shannon sampling theorem: a signal must be sampled at more than twice its highest frequency, otherwise it aliases onto a lower apparent frequency. Convolution then describes how any linear system shapes a signal:

$$y[n] = \Sigma_k \, x[k] \cdot h[n - k] \tag{2}$$

The Filters lab applies Butterworth low-pass and high-pass filters, and the FFT Explorer shows that a square wave contains only odd harmonics, a sawtooth contains all harmonics, and a triangle wave has odd harmonics that decay much faster.

3.3 Power Systems

The module follows power from device to grid. A real transformer is modelled by its equivalent series impedance referred to the secondary, $V_1' = V_2 + I_2(R_{eq} + jX_{eq})$, from which voltage regulation and efficiency follow. A medium-length transmission line uses the nominal-π model with ABCD parameters:

$$V_S = A \cdot V_R + B \cdot I_R, \quad A = 1 + ZY/2, \quad B = Z \tag{3}$$

Three counterintuitive results are built into the labs. A leading power factor can make voltage regulation negative, so the secondary voltage rises under load. The Ferranti effect lets a lightly loaded long line deliver more voltage than it receives because of its shunt capacitance. And power flow is nonlinear: beyond a critical load it has no stable solution, which is voltage collapse. The Fault Analysis lab computes fault current from the Thevenin impedance at the fault bus, and the Protection System lab shows how a distance relay classifies a fault as Zone 1, Zone 2 or out of zone from its apparent impedance.

3.4 Electronic System Design

The module covers the building blocks of analog circuits: op-amp gain set by feedback resistors and limited by the supply rails; diode rectification, with an ideal half-wave average of V_p/π and a full-wave average of $2V_p/\pi$; BJT biasing on the load line; passive RC and RL filters with cutoff

$$f_c = 1 / (2\pi RC) \tag{4}$$

and the comparison of linear and switching voltage regulators by efficiency.

3.5 Renewable Energy

A photovoltaic cell is characterised by its I–V curve and a single maximum power point; a hotter cell produces less power because its voltage falls faster than its current rises. MPPT algorithms such as Perturb and Observe climb the power curve toward the peak. Wind turbine output follows the cube of wind speed but is capped by the Betz limit:

$$P = \tfrac{1}{2} \cdot \rho \cdot A \cdot v^3 \cdot C_p, \quad C_p \le 0.593 \tag{5}$$

Battery storage introduces state of charge, depth of discharge and round-trip efficiency, and the grid-tied inverter builds an AC waveform by pulse-width modulation and is judged by total harmonic distortion against IEEE 1547.

3.6 Machine Learning and Biomedical Systems

The Machine Learning module teaches learning as optimisation, $\theta \leftarrow \theta - \alpha \nabla J(\theta)$, and exposes the learning-rate trade-off: too small and training is slow, too large and it oscillates or diverges. The Biomedical module applies signal processing and system modelling to physiology (ECG, EEG, blood pressure and pulse oximetry) and uses a compartmental pharmacokinetic model with half-life $t_{1/2} = 0.693\ V_d / CL$.

4 System Architecture

EE Studio is a two-tier web application in one monorepo (Figure 2). A React single-page application renders the labs, and a FastAPI service performs every calculation. All engineering mathematics lives in an engine layer of pure Python functions with no web dependencies, so that each model can be tested in isolation. The route layer only validates the request with Pydantic and calls an engine. When a student moves a slider, the lab page sends a JSON request to `/api/v1/{module}/{lab}` and renders the returned values in charts and in the physical visualisation.

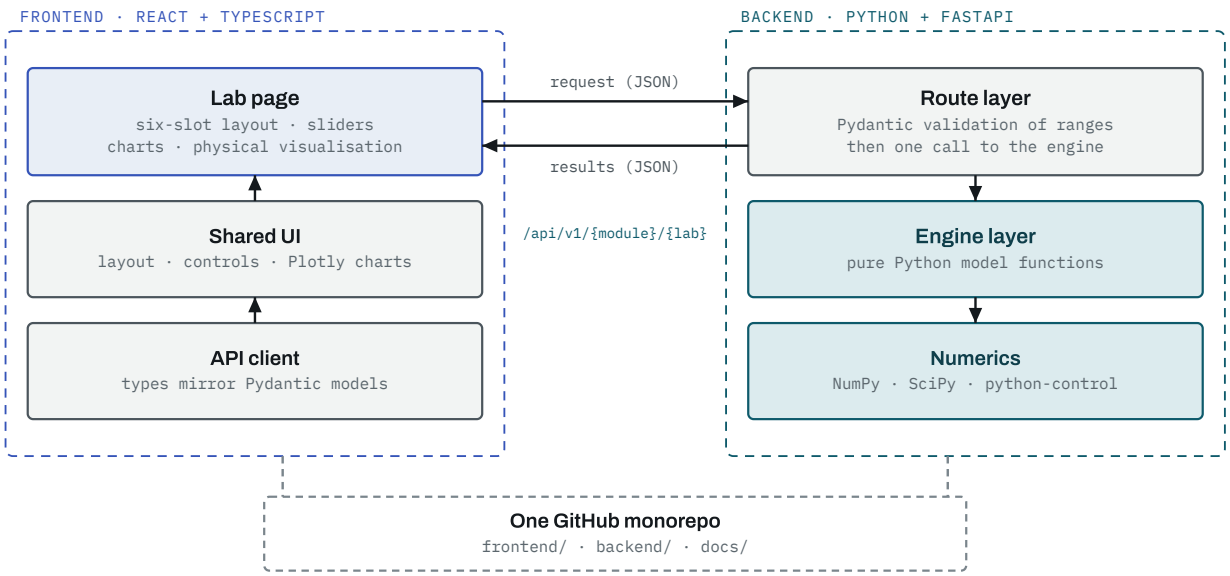


Figure 2. System architecture: frontend, backend and repository.

Table 1. Technology stack

Layer	Technology	Reason for choice
Frontend	React, TypeScript, Vite	Component model for reusable labs; static typing catches API mismatches at build time
Styling and motion	Tailwind CSS, Framer Motion	One design system of tokens and panels
Charts	Plotly (react-plotly.js)	Time series, Bode, Nyquist, pole-zero and 3D surfaces from one library

Layer	Technology	Reason for choice
Backend	Python, FastAPI, Pydantic	Fast API with strict request validation and automatic OpenAPI documentation
Numerics	NumPy, SciPy, python-control	Verified FFT, filter design, root finding and transfer-function simulation
Testing	pytest	Unit tests on engines and integration tests on routes
Delivery	GitHub, Railway, Vercel	Source, backend and frontend hosting, all with free tiers

5 Development Methodology

Every lab follows the same seven-step pipeline, and its first gate is mathematical rather than visual: the model is checked numerically before an engine is written (Figure 3).

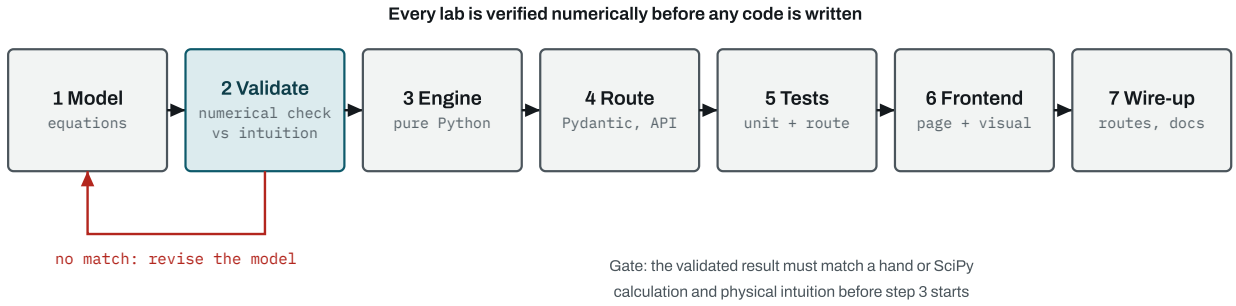


Figure 3. Lab development pipeline: seven steps and one decision gate.

The rules behind the pipeline were applied to all labs: validate the mathematics numerically, then check the result against physical intuition; prefer well-tested libraries to hand-derived solvers; keep engines pure and routes thin; test the hand-verified case exactly plus edge cases such as no load, full load and collapse; record every counterintuitive result and how it was verified in the engine's docstring; and document each lab's objective, theory, model, calculations, visualisation and tests.

The platform was built through human-directed, AI-assisted development. The author set the scope, reviewed each lab in the browser and decided what to change, while Claude and Claude Code generated and tested the code under these conventions.

6 Implementation

6.1 Backend Engines and Routes

Each lab has an engine file, a route file and two test files. Engines are pure functions that take plain numbers and return result objects. Routes declare their request and response models with Pydantic, enforce physical ranges (for example a power factor above zero and at most one) and delegate to the engine. Each module has one router aggregating its labs, and all seven are registered under `/api/v1`.

6.2 Frontend Lab Framework

Every lab page is composed from the same six-slot layout (Figure 4). Reusable controls and chart components are shared across modules, which is why later labs needed few new components. Each lab keeps TypeScript types that mirror its Pydantic models and a small API client built on one shared request helper.

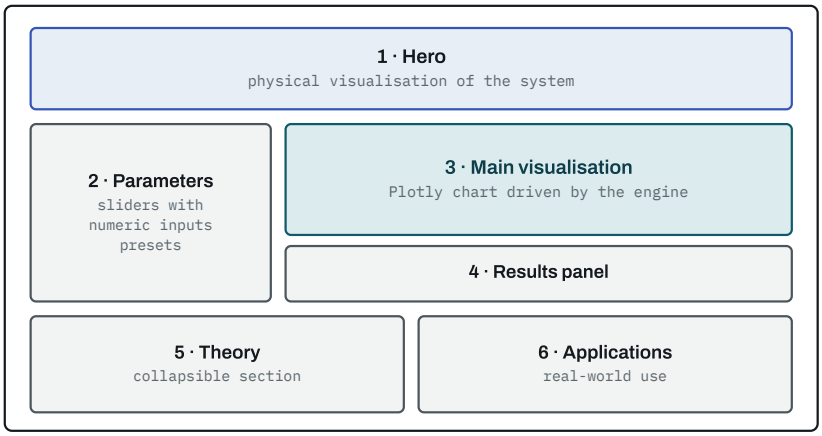


Figure 4. The six-slot layout shared by every lab page.

6.3 Design System and Physical Visualisations

A single design system keeps the modules coherent: one colour is reserved for data and signals, one for interactive elements and one for warnings only, so colour always carries meaning, and each module has its own identity colour. Every lab has a physical picture of the system it studies, driven by the same computed values as the charts: a mass on a spring, a rotating phasor, a wheel under strobe light, a transformer core that glows with its losses, and a two-bus network that flashes when voltage collapses.

6.4 Enhancement Phase

After the base labs were complete, each was reviewed as a student would use it and extended where the learning value was weak. Typical additions were multi-level challenges that unlock in sequence, comparison modes, guided hints, gain and phase margin indicators on the Bode

plot, and renaming abstract coefficients to physical ones (the State Space lab uses stiffness k, damping b and inertia m instead of a_0, a_1 and a_2). The same review added previous/next navigation between labs.

7 Testing and Verification

The platform holds seven modules and 37 labs, with 485 backend tests passing at the first complete milestone (Table 2, Figure 5).

Table 2. Module and lab inventory

Module	Labs	Labs	Tests
Control Systems	Step Response, PID Controller, Bode Plot, Root Locus, Nyquist, State Space	6	76
Digital Signal Processing	Signal Generator, Sampling, Convolution, Filters, FFT Explorer, Frequency Spectrum	6	81
Power Systems	Transformer, Transmission Line, Power Flow, Fault Analysis, Protection System	5	65
Electronic System Design	Op-Amp, Diode Rectifier, BJT Biasing, RC/RL Filter Design, Voltage Regulator	5	60
Renewable Energy	Solar PV, MPPT Algorithm, Wind Turbine, Battery Storage, Grid Inverter	5	58
Machine Learning	Regression, Logistic Regression, Neural Network, K-Means Clustering, PCA	5	73
Biomedical Systems	ECG, Neural Signal (EEG), Blood Pressure, Pulse Oximetry, Pharmacokinetics	5	72
Total		**37**	**485**

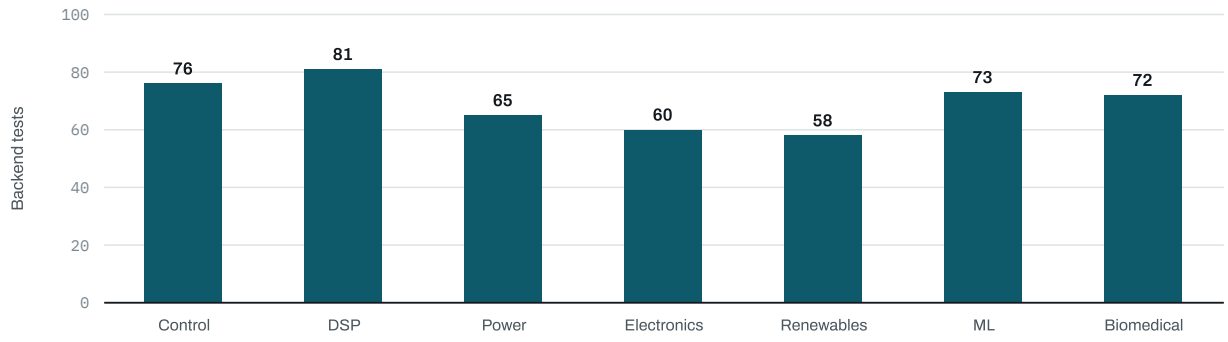


Figure 5. Passing backend tests per module.

The tests work in two layers. Engine tests assert results against values computed independently by hand or with SciPy, for example a transformer regulation of 8.41 % at full load and 0.8 lagging power factor, or an aliased frequency of 2 Hz for a 10 Hz tone sampled at 8 Hz. Route tests confirm the response shape for valid requests and a 422 rejection for out-of-range input, such as a zero power factor or a negative resistance.

Tests also encode physical behaviour, not only numbers: load-bus voltage must fall as load rises, a leading power factor must give lower regulation than a lagging one, a zero-amplitude tone must produce no spectral peak, and a heavy enough load must make power flow report non-convergence instead of returning a false value. Noise in the spectrum lab uses a fixed random seed so that the tests are deterministic. The production build adds a second check: the strict TypeScript compiler caught missing type declarations that the development server had tolerated.

8 Challenges and Lessons Learned

The most valuable lessons came from bugs that looked plausible but were wrong (Table 3).

Table 3. Main issues and their resolution

Issue	Cause	Resolution
Power flow showed voltage rising as load increased	Sign error in hand-expanded sin/cos power-injection equations	Rebuilt on the admittance matrix, $S = V \cdot \mathrm{conj}(Y_{bus} \cdot V)$; regression test for the correct direction
Root Locus crashed at K = 0	A zero numerator broke python-control's automatic time horizon	Special case for K = 0
Nyquist curve distorted near the origin	Magnitudes grow without bound near the origin	Plotted magnitude clipped to a radius based on the real-axis crossing
Page crashed reading undefined fields	New backend fields were read before the backend returned them	Optional chaining with fallbacks; engines always return every field
State Space returned 422 after a rename	A stale backend process still held port 8000	Old process stopped; the route also accepts the old field names
Railway could not start the backend	The build tool could not infer the start command	Procfile with the uvicorn command
Vercel build failed	Missing type declarations for Plotly	Declaration file added
Deep links returned 404 on Vercel	A single-page app has no file at each route	Rewrite rule serving index.html for every path

Three lessons apply beyond this project. Check every numerical result against physical intuition, because a wrong sign can produce a clean-looking curve. When behaviour contradicts the code, suspect a stale process or cache before suspecting the code. And build the

production configuration early, because a strict build and a different host expose problems that a local development server hides.

9 Deployment

The platform is deployed as two services fed from one GitHub repository (Figure 6): the frontend on Vercel and the backend on Railway, each redeploying when the main branch changes. The frontend reaches the backend through the `VITE_API_URL` environment variable. The first deployment returned HTTP 405 from every lab because the frontend was not reaching the Railway API; once the API address was configured for the Vercel build, all labs returned data from the live backend.

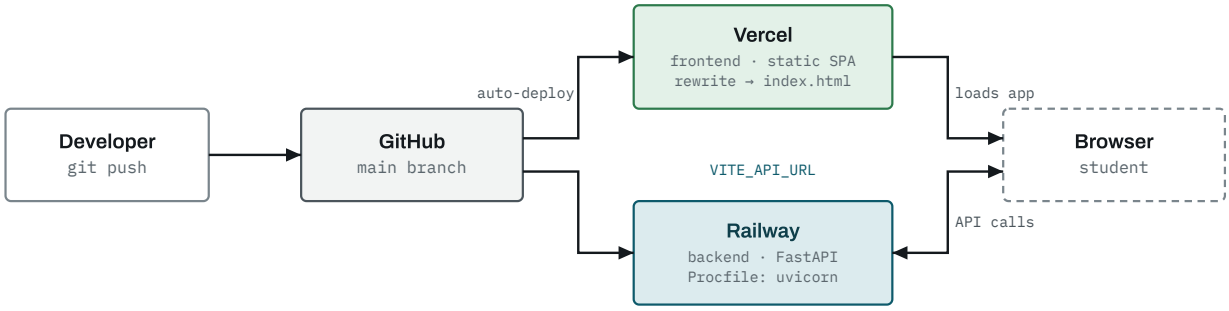


Figure 6. Deployment topology.

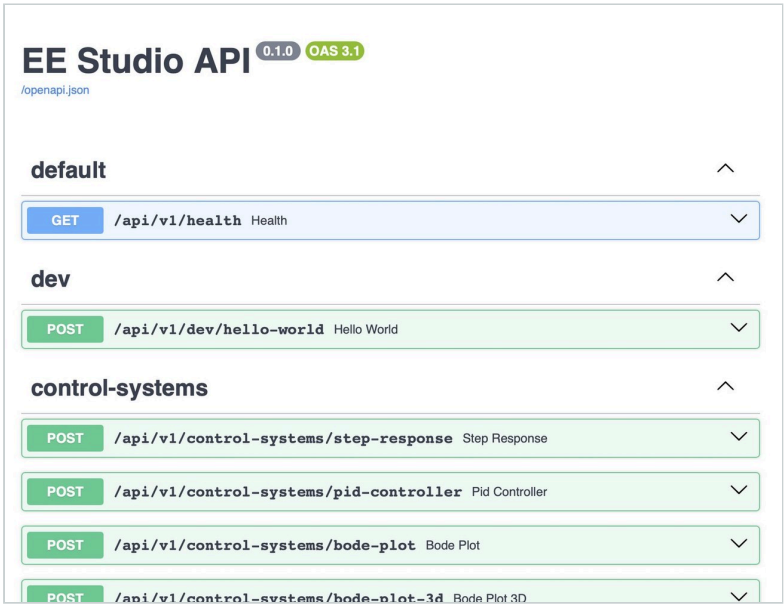


Figure 7. Interactive API documentation served by the deployed backend.

10 Results, Future Work and Conclusion

10.1 Results

- **Coverage:** 7 modules and 37 labs, each with a verified engine, a validated route, tests, a lab page, a physical visualisation and a documentation page.
- **Quality:** 485 passing backend tests at the first complete milestone, including regression tests for the bugs in Table 3.
- **Repeatability:** once the conventions and shared components were in place, later modules were generated in 10 to 20 minutes per batch of labs.
- **Delivery:** source on GitHub, a live Railway backend and a live Vercel frontend at ee-studio-red.vercel.app.

10.2 Future Work

1. Complete the Renewable Energy module with an Energy Management lab that coordinates solar, wind, battery, grid and loads over a day.
2. Extend the enhancement phase to the Machine Learning and Biomedical modules.
3. Add a continuous-integration pipeline that runs the backend tests and the strict frontend build on every push.
4. Add learner features such as saved progress, and a mobile and accessibility review of every lab.
5. Grow the catalogue with further modules built on the same pipeline.

10.3 Conclusion

EE Studio shows that abstract engineering theory becomes easier to grasp when each concept is an experiment: a student changes a parameter and sees the physics respond. The discipline of the project was as important as its scope. Every model was verified numerically before it was coded, the engineering mathematics was kept separate from the interface, and every bug that exposed a wrong assumption became a test. The result is a tested, documented and deployed platform of 37 labs and a repeatable pattern for adding more.