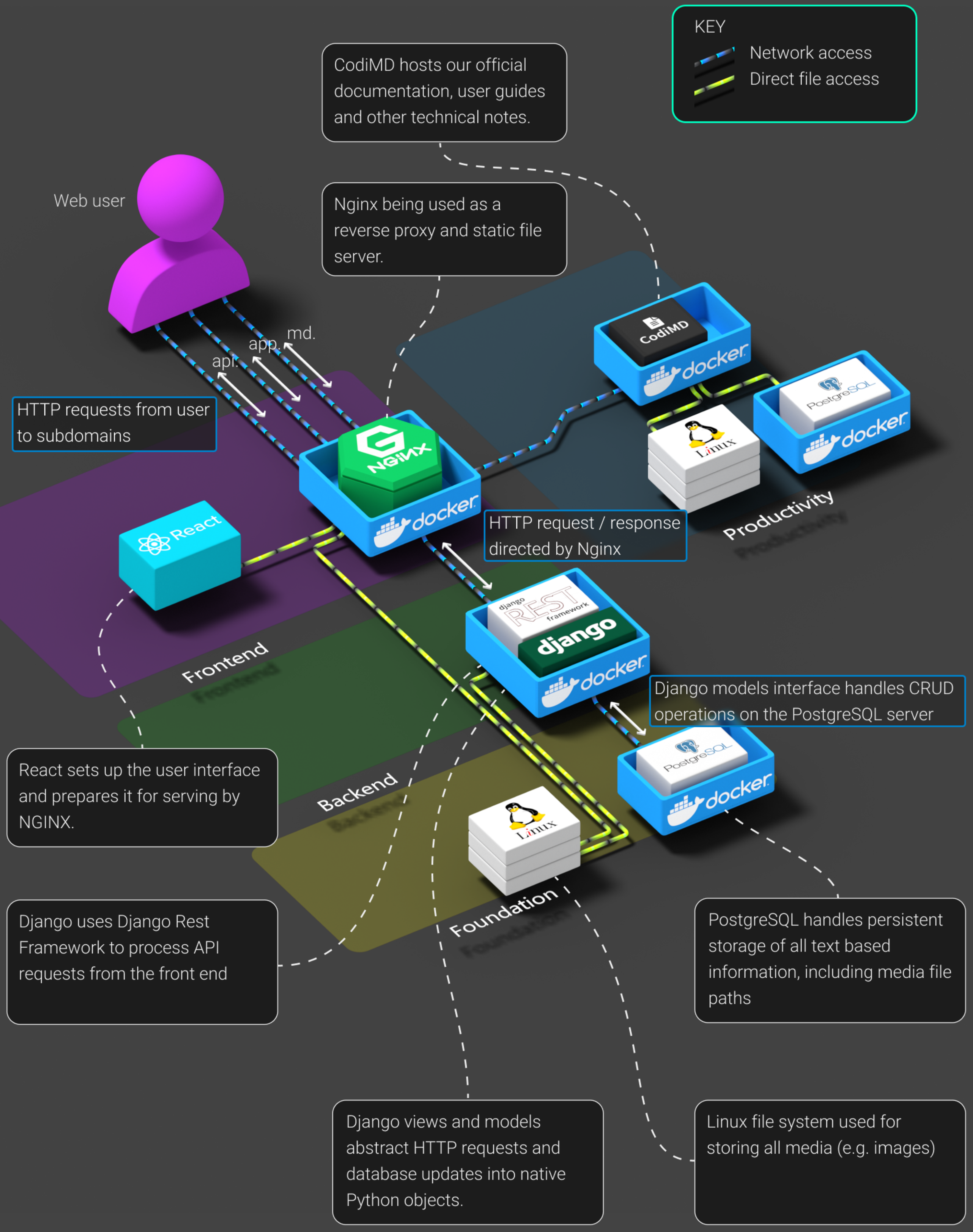
Jinx Software architecture



CodiMD hosts our official documentation, user guides and other technical notes.

Nginx being used as a reverse proxy and static file server.

HTTP requests from user to subdomains

HTTP request / response directed by Nginx

Django models interface handles CRUD operations on the PostgreSQL server

React sets up the user interface and prepares it for serving by NGINX.

Django uses Django Rest Framework to process API requests from the front end

PostgreSQL handles persistent storage of all text based information, including media file paths

Django views and models abstract HTTP requests and database updates into native Python objects.

Linux file system used for storing all media (e.g. images)

Design decisions:

Django
We chose this because it is a robust framework which implements plenty of boiler plate code. If we used a more basic framework, we would need to implement a lot of what django does for us. Django comes with an object-relational mapper which provide an intuitive interface for database operations. The automated setup of secure users was another drawcard.

PostgrSQL
Since Django absracts away all of the database interactions, our choice here wasn't too important. We went for PostgreSQL because it's powerful and is known to integrate especially nicely with Django.

Docker
Across our team, we have members working on Linux, Mac and Windows. This being the case it made sense for us to make use of a system that minimised any potential conflicts that might arise due to the difference in operating systems people were running. Docker seemed like the best option on the market for ensuring consistancy.

React
For the front end, our primary goal was to choose a framework that allowed for the complicated interaction between front end and back end that would allow us to complete all requirements of the project. We also wanted something that had a lot of resources on the internet so that we could pick it up reletively quickly. React ticked these boxes while also making use of functional programming, a paradigm we all liked.

J i n x Use case example

Use Case: Adding a section to portfolio

01. User uses a browser to request https://app.jinx.systems

02. NGINX obtains the static files for the front page from the file system and sends it to the user. The static files were previously generated by React when deploying.

03. User's browser executes the fetched javascript to render the home page

04. User clicks the 'Login' button

05. Similar to steps 2–3, NGINX responds with the login page and user's browser displays it

06. User enters in their username and password and submits the form.

07. The user's browser sends the form data as a JSON file containing the username and password to the back end (https://api.jinx.systems) as a POST request.

08. NGINX receives the POST request and forwards it on to Django which uses 'views' to handle the request.

09. The login view uses serializers to convert the JSON data into a Django model object.

10. Django then uses its 'queryset' interface to convert the Django object to a SQL query and sends that to PostgreSQL.

11. Assuming that the user exists, PostgreSQL returns the stored hashed password for the user.

12. If the provided password is correct, Django generates an authentication token and stores it. It also sends it back though NGINX to the user's browser.

13. The returned authentication token is stored locally on the user's computer so that they don't have to log in again every time they visit the site.

J i n x   Use case example

14. Once the token is received, the frontend code will redirect the page to the user's portfolio page.

15. The data for the portfolio page is obtained from the backend through a GET request to the api. The frontend attaches the authentication token to the request header.

16. Django checks the provided token against the database to authenticate the request.

17. If the request is authenticated (valid token), Django responds with the portfolio data for that user.

18. The user is presented with their existing portfolio presented in an editable format. They click the '+' button and choose 'Image Section'.

19. Javascript code (the front end) ensures that a new image upload section appears below where the '+' button was clicked.

20. User clicks the image upload button and chooses an image from their computer.

21. Upon choosing one and clicking 'Okay', the image data is converted to JSON format (including file name and actual image data) and sent to Django via NGINX.

22. As before, DRF serialises the data, converting it to native Django format.

23. Django stores the actual image data on the Linux file system and sends a SQL query to save the image metadata (path and file name) to the database.

24. PostgreSQL receives the SQL query and the information is stored in the database.

25. Django uses DRF to create a response message which includes the path of the image.

26. The front end puts the image URL into the DOM (what the browser renders) which causes the browser to fetch the image to display.

27. The browser sends a GET request to fetch the image from our server. NGINX responds to this request and sends the image to the browser. The browser then displays this image.

J i n x Use case example

28. User clicks the publish button. Javascript code collects all section information on the page and converts it to a JSON list. This JSON object is sent to Django.

29. As before, Django uses DRF and SQL to convert and store the incoming information and also send back a response JSON object.

30. Upon receiving the JSON confirmation, Javascript redirects the user to their ‘portfolio display’ page.

31. Javascript sends a request to the API to obtain data for all the portfolio information.

32. Django uses DRF and SQL to fetch the data from the database and responds with a JSON object.

33. The user’s browser and Javascript render the data correctly on the user’s screen.