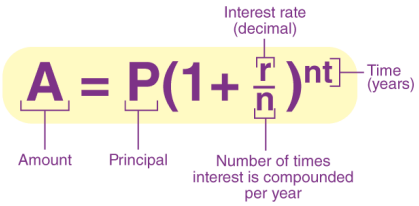
1. The tools I have used are detailed as follows

Django: It provides a secure web framework and is easy to use.
Django REST Framework: to build JSON APIs by using serialisers to handle input validation
Django-cors-headers: lets the frontend communicate to the backend
React: to build the frontend in reusable components
React router: to navigate between the few pages I put in
Node.js is used to run react
SQLite: the default django database, doesn't require set up so I am able to prototype and test quickly
VSCode IDE: the IDE i normally work with, has built in terminals
Git: version control

2. While working at IKEA as a product developer, I got to build and maintain Django/React based digital tools. On one occasion we encountered a problem where the frontend react based dashboard was not displaying updated data correctly after changed in the backend. The Django-based API was returning data but the UI was showing errors.
   To solve the issue I took a number of steps. Firstly using the developer tools to just confirm if the frontend was making the request and whether the API response was correct. I then also verified the backendlogs to ensure the django view was actually engaged and returning the expected JSON response. The naming format had changed between the backend response causing the react component to break.
   I developed some unit tests on the backend side to ensure that the api wasn't consistently returning the incorrect format. I ran some manual testing and wrote a simple frontend test to confirm the react components rendered correctly. I also received feedback from my colleagues throughout the process, supporting my testing and error discovery process. I was able to document the issues found and share these with my colleagues to prevent similar errors in the future.

3. To test a Django backend I use Django's built in unittest framework. It supports writing tests for API endpoints. The tests run in a temporary database so they don't affect the stored data.
   For the react testing I unit test the individual components, testing the user interactions and making sure the ui updates correctly. Also, I use mock api requests to test forms submit correctly, separate to a live backend.
   To ensure they are comprehensive and reliable we cover all the major flows and edge cases. The tests are kept small and specific to a given section. This makes them independent and repeatable if there are changes made. The tests are run and updated during development to ensure any changes are accounted for.

4. To build a financial web application I would create a REST API endpoint to handle any calculations. Using the compound interest formula:



The endpoint would validate the input to ensure the values are numeric, positive if needed, the compounding freq is valid. Error messages should be displayed if these fail, only returning the calculated amount as a JSON once the checks are passed.
For the front end, I would build a simple form to collect the inputs from the user, doing basic validation before sending them to the backend to ensure the calculations can be trusted. The inputs can then be submitted to the backend API. The calculated amount can then be displayed

5. To ensure edge cases and boundary conditions are handled properly I consider both backend and frontend edge cases. Implementing pagination, filtering and sorting, I use Django's ORM and DRF's built- in tools, validating all query parameters. I also include default values for optional parameters to not break the API. I would also write some simple unit tests for common boundary issues.
   For the front end I would handle simple issues like empty lists, no results, out of range page numbers and such, validating user inputs. I'd implement graceful error handling if the API returns an error.
   While working on a project at IKEA, there were some edge cases involving missing and partial metadata for some files. Also when testing on a larger number of files, issues began to arise with pagination and rendering issues. In the bakend I implemented validation for required metadata fields. For the frontend I added some fallbacks for missing fields of metadata. I also ensured the pagination would work for the higher volume of datasets it would inevitably require.

6. I would use a Django REST API using a two pointer technique on the sorted list of integers to find the pair with the smallest floored difference to the result. The results would be returned as a JSON. This would be a more efficient approach on a large dataset. For the frontend in React I would create a form to collect the integers and the target value, ensuring the inputs are numeric and not null/empty. It would then submit the values to the backend, displaying the result once reached, potentially displaying a loading state or error messages.