

# Gen AI projects knowledge share

Four use-cases using ChatGPT

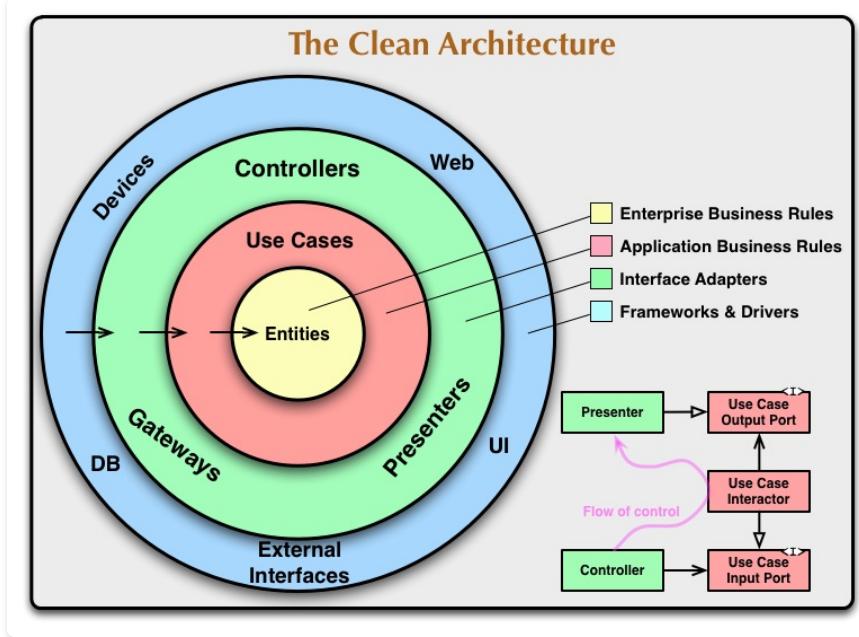
Press Space for next page →

# Table of contents

- Hexagonal architecture on the front-end
- Separating the state for rest
- Testing the store
- Presentational component vs container component
- Testing libraries for react: Vitest, React Testing Library, Mock Server Worker
- Links



# Hexagonal architecture



- Hexagonal architecture/Clean architecture/onion architecture: same idea under many names
- Domain logic should be in its own dedicated layer
- Domain layer does not depend on any other layer (may require dependency inversion)
- Ports/adapters pattern that let replace implementation easily (SQL to elasticSearch, REST to graphQL, etc...)

# Clean architecture on the backend

```
async function sendFeedbackController(request: Request, response: Response): Promise<void> {
  const feedback = z
    .object({
      payload: z.any(),
      type: feedbackTypeSchema,
    })
    .parse(request.body);
  const user = parseAuthenticatedUser(response);
  await sendFeedback({ feedback, user });
  response.send({});
}
```

- On the backend, we remove request handling and db logic from domain services (handlers and repositories)
- See above the handler above that parse the request, calls a domain service, then send the response

# Repository pattern: makes database change much easier

```
async function getReviewsByRating({ reviewFilters } : { reviewFilters: ReviewFilters }): Promise<Aggs[]> {
  if (elasticRepository.hasReviewIndex()) {
    return elasticRepository.aggregateReviewsByRating({ reviewFilters });
  }
  return SQLrepository.aggregateReviews({
    reviewFilters,
    attributes: ['rating', [Sequelize.literal('COUNT(DISTINCT(reviews.id))'), 'count']],
    group: ['rating'],
  });
}
```

- Repository pattern: create a dedicated module to handle db logic. Quite heavy, but lets us switch from one database to another easily.
- Remark: this does NOT respect Clean Architecture by the letter. Dependency inversion wasn't done, and the SQL syntax leaks out of the SQL repository.

# Hexagonal on the front-end: the UI as an afterthought

David K Piano · Follow

Model and develop your app's logic as if it will be used in many different UIs (e.g., web, CLI, native, etc.), even if it won't be.

This will force you to avoid coupling logic with presentation, which mainstream frameworks (React, Angular, Vue) unintentionally encourage.

3:08 PM · Feb 4, 2019

165

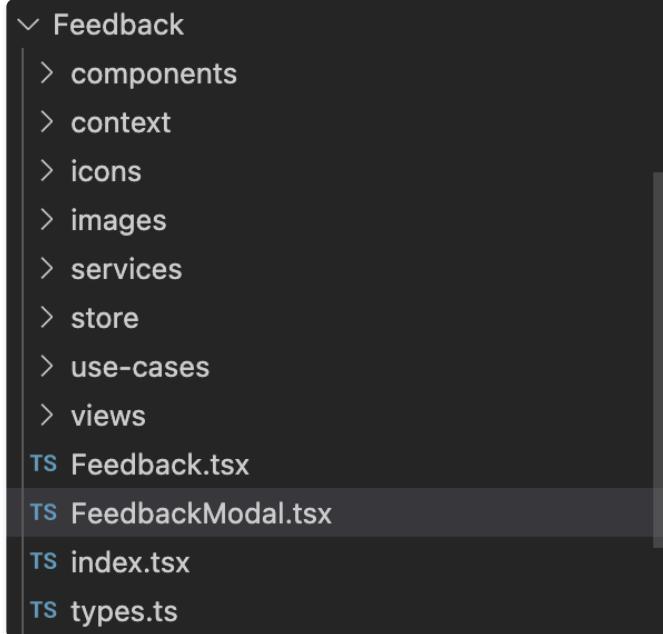
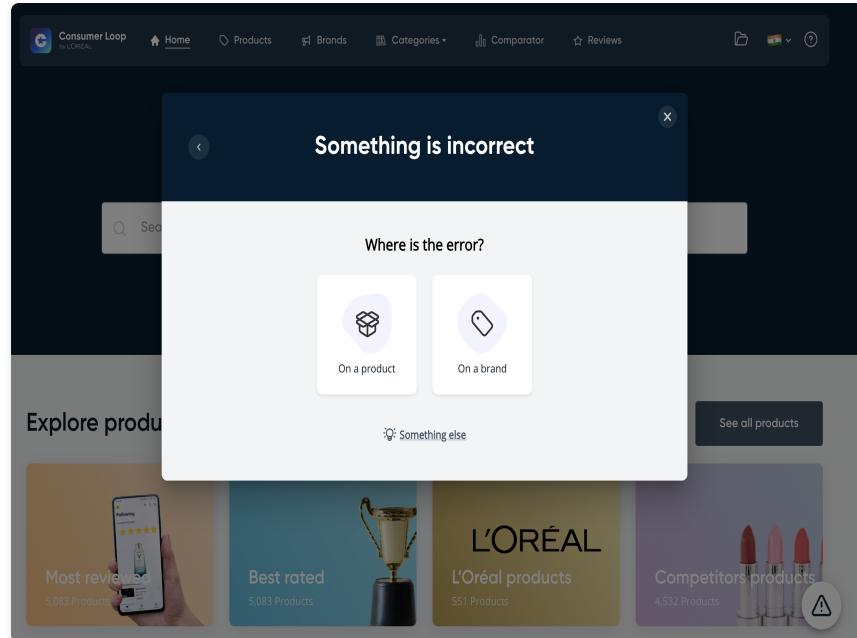
Reply

Copy link

Read 10 replies

- The core idea: remove all UI and query from the domain logic. Imagine that there are two ways to interact with app, UI or CLI. The core logic should not be duplicated.
- All HTTP calls (Web sockets, ...) should be extracted to dedicated services
- UI interfaces extracted to presentational components
- So in practice the domain logic ends up identified to with the application global state (WARNING: this may an abuse of language)

# Hexagonal on the front-end: the feedback module example



# The feedback module example: an HTTP service

```
function useNewFeedback({  
  options = {},  
}: {  
  options?: MutationOptions<Feedback>;  
}): SafeMutation<Feedback> {  
  const mutationFn = usePostMutationFn({ input: `/api/feedback` });  
  return usePostMutation<Feedback>({ options: { mutationFn, ...options } });  
}
```

- Service handling HTTP requests to backend

# The feedback module example: a view

```
function IncorrectBrandName(): JSX.Element {
  const { brandId, newFeedbackMutation } = useCollaborationContext();
  const [correctName, setCorrectName] = useState<string>('');
  if (brandId === null) return <></>;
  return (
    <>
      <Header title="The name is incorrect" />
      <Body>
        <TextInput
          onChange={(event) => setCorrectName(event.target.value)}
          value={correctName}
        />
        <Actions
          onMainClick={() => sendFeedback({
            type: 'incorrectBrandName',
            payload: { brandId, correctName },
          })}
        />
      </Body>
    </>
  );
}
```

- View contains presentational logic

# The feedback module example: the store

```
function useFeedbackStore({  
  params = {},  
  useNewFeedback,  
}: FeedbackStoreProps): IFeedbackStore {  
  const [isModalOpen, setIsModalOpen] = useState<boolean>(false);  
  
  const historyStore = useFeedbackHistory({});  
  const { view } = historyStore;  
  
  const { reverse, ...navigationMethods } = useReverse(historyStore);  
  const { clearHistory, goBack, goBackToStart } = navigationMethods;  
  
  const { goTo, productId, setProductId, brandId, setBrandId } = useSelection({  
    ...navigationMethods,  
    params,  
  });  
  
  const newFeedbackMutation = useNewFeedback({ options: { onMutate: () => goTo('success') } });  
  
  return { ... };  
}
```

# The feedback module example: a use-case

```
function useFeedbackHistory(): IFeedbackHistory {
  const [viewHistory, setViewHistory] = useState<IView>(['root']);

  const view = viewHistory[viewHistory.length - 1];

  const goTo = useCallback((newView) => setViewHistory((previous) => [...previous, newView]), []);
  const goBack = useMemo(() => {
    if (viewHistory.length <= 1) return null;
    return () => setViewHistory((previous) => previous.slice(0, -1));
  }, [viewHistory.length]);
  const goBackToStart = useMemo(() => {
    if (viewHistory.length <= 1) return null;
    return () => setViewHistory(['root']);
  }, [viewHistory.length]);

  return { view, goTo, goBack, goBackToStart };
}
```

- Navigation use-case: the user may go back to previous view, or back to start
- Use-case is extracted in its dedicated hook

# Testing the navigation use-case

```
describe('useFeedbackStore', () => {
  describe('goBack', () => {
    it('should go back to previous view', () => {
      const feedbackStore = setupStore({});
      act(() => feedbackStore.goTo('missingBrand'));
      act(() => feedbackStore.goTo('success'));

      act(() => {
        if (feedbackStore.goBack === null) return;
        feedbackStore.goBack();
      });

      expect(feedbackStore.view).toEqual('missingBrand');
    });
  });
});
```

- We test the use-case on the store of the module global state
- Remark: we did NOT test the use-case hook directly. That would couple the test to the implementation, require more tests, and cover less code.

# Limitations - UI is complex and valuable

danabra.mov @dan\_abramov · Follow

Dogmatic adherence to this line of thinking lead people to "HTML/CSS/JS must be separate concerns!" once.

Modeling BL separately has merits. But treating UI as second class citizen ignores that many of our problems *\*are\** in that domain. [overreacted.io/the-elements-o...](https://overreacted.io/the-elements-of-react/)

9:38 PM · Feb 5, 2019

7 7 · Reply · Copy link

Read 2 replies

- UI is a critical part of the value created by our application. There is not a single CLI interface with mass-adoption. And it can be very complicated, much more complex than query handlers for instance. So it's a mistake to treat UI as an implementation detail.
- Dogmatic separation of state/presentation can create artificial boundaries. Simple components can (and should) own their own state/presentation.

# Why does it work? Human/machine have different strengths

case/tester	Human	Machine
State machine	Bad	Very good
UI	Very good	Bad

- Humans visual capabilities are very advanced. We can detect visual bugs very easily without any automated testing. But we do struggle to check that algorithms correctly implement business logic.
- The reverse is true for machines. So the highest ROI testing strategy is to extract the state/BL logic to test automatically, and check visually that the UI is correct.

# Presentational components vs container components



Dan Abramov

Mar 23, 2015 · 5 min read · Listen

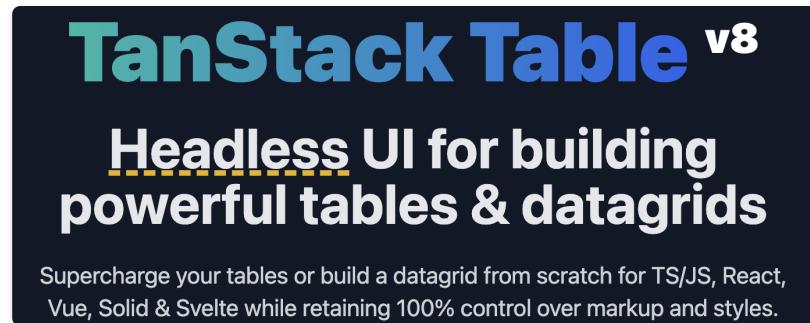


## Presentational and Container Components

- Famous article distinguishing presentation components (mainly UI) from container components (mainly state)
- old nomenclature: smart vs dumb
- Remark: don't be dogmatic, use hooks, etc...

# Headless libraries

Some libraries only handle the stateful portion of the components. There are called "headless" libraries.

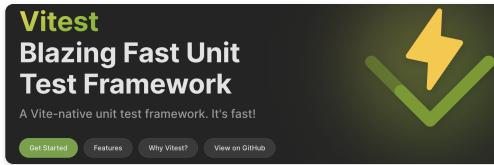


See tanstack Table, which in React is a simple hook:

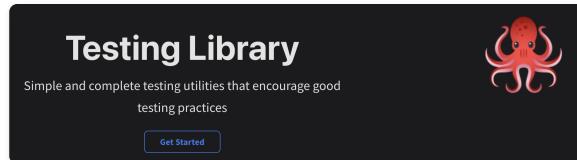
```
const table = useReactTable({  
  data,  
  columns,  
  getCoreRowModel: getCoreRowModel(),  
})
```

# Testing libraries (React)

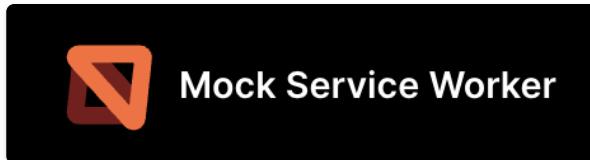
- Vitest - jest but cooler (and much faster)



- React Testing library - jsx testing made easy



- Mock service worker - intercept HTTP calls to avoid mocking



## Architecture:

- Clean architecture: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>
- Presentational and Container Components: [https://medium.com/@dan\\_abramov/smart-and-dumb-components-7ca2f9a7c7d0](https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0)
- UI as an afterthought: <https://michel.codes/blogs/ui-as-an-afterthought>

## Libraries:

- Mock service worker: <https://mswjs.io/>
- React Testing library: <https://testing-library.com/>
- Vitest: <https://vitest.dev/>

## React tests:

- How to test custom React hooks: <https://kentcdodds.com/blog/how-to-test-custom-react-hooks>
- Stop mocking fetch: <https://kentcdodds.com/blog/stop-mocking-fetch>

## QUESTIONS AND REMARKS



**Ekimetrics.**