Treat type definitions with respect

by Jacek Tomaszewski
Full-stack Software Engineer & Team Lead at AppUnite
jtom.me / jtompl / jackthenomad.com

Code gradually becomes a mess.

Code gradually becomes a mess.

Apply clean code practices to prevent it.

boy/girl-scout rule

SOLID

tests

DRY

more tests

apply style guide

avoid giant files

use standard data structures

dependency injection

KISS

separate responsibilities

small functions

code review

use meaningful names

throw specific exceptions

YAGNI

abstract away third parties

use comments

use adapters

TDD, DDD

Clean code practices

Code
=
?

Code

=

Interface + ...

Code

=

Interface + Implementation

Interface

```
interface MoneyInputProps
  extends Omit<TextInputProps, "value" | "onChange"> {
  value?: Money | null;
  min?: Money;
  max?: Money;
  onChange?(value: Money | null): void;
}

export function MoneyInput(props: MoneyInputProps): ReactElement;
```

Implementation

```
export function MoneyInput(props: MoneyInputProps): ReactElement {
  const [state, setState] = useState({
    value: props.value,
    text:
      props.value?.cents !== undefined
        ? formatMoney(value)
        : undefined
  });

  // ... and so on
}
```

When applying clean code practices,
do we apply them
to the interface code as well?

Clean code practices

Practice	Implementation	Interface
Use accurate naming	Name internal variables, functions well	Name public interfaces well

Clean code practices

Practice	Implementation	Interface
Use accurate naming	Name internal variables, functions well	Name public interfaces well
Single Responsibility Principle	Don't do multiple things at once	Do one thing and do it well

Clean code practices

Practice	Implementation	Interface
Use accurate naming	Name internal variables, functions well	Name public interfaces well
Single Responsibility Principle	Don't do multiple things at once	Do one thing and do it well
Interface Segregation Principle	-	Keep interfaces minimal

Clean code practices

Practice	**Implementation**	**Interface**
Use accurate naming	Name internal variables, functions well	Name public interfaces well
Single Responsibility Principle	Don't do multiple things at once	Do one thing and do it well
Interface Segregation Principle	-	Keep interfaces minimal
Dependency Inversion Principle	-	Don't rely on implementation; Rely on an abstract interface

Clean code practices

Practice	**Implementation**	**Interface**
Use accurate naming	Name internal variables, functions well	Name public interfaces well
Single Responsibility Principle	Don't do multiple things at once	Do one thing and do it well
Interface Segregation Principle	-	Keep interfaces minimal
Dependency Inversion Principle	-	Don't rely on implementation; Rely on an abstract interface
YAGNI (You Aren't Gonna Need It)	Don't implement unneeded things	Don't support unneeded things

Clean code practices

Practice	**Implementation**	**Interface**
Use accurate naming	Name internal variables, functions well	Name public interfaces well
Single Responsibility Principle	Don't do multiple things at once	Do one thing and do it well
Interface Segregation Principle	-	Keep interfaces minimal
Dependency Inversion Principle	-	Don't rely on implementation; Rely on an abstract interface
YAGNI (You Aren't Gonna Need It)	Don't implement unneeded things	Don't support unneeded things
KISS (Keep It Simple Stupid)	Keep the implementation simple	Keep the interface simple

Clean code practices

Practice	**Implementation**	**Interface**
Use accurate naming	Name internal variables, functions well	Name public interfaces well
Single Responsibility Principle	Don't do multiple things at once	Do one thing and do it well
Interface Segregation Principle	-	Keep interfaces minimal
Dependency Inversion Principle	-	Don't rely on implementation; Rely on an abstract interface
YAGNI (You Aren't Gonna Need It)	Don't implement unneeded things	Don't support unneeded things
KISS (Keep It Simple Stupid)	Keep the implementation simple	Keep the interface simple
Avoid giant files	Extract implementation bits to methods, classes, files	Group related interfaces, and put each group in a separate file

Clean code practices

pls don't skip that part

Practice	Implementation	**Interface**
Use accurate naming	Name internal variables, functions well	Name public interfaces well
Single Responsibility Principle	Don't do multiple things at once	Do one thing and do it well
Interface Segregation Principle	-	Keep interfaces minimal
Dependency Inversion Principle	-	Don't rely on implementation; Rely on an abstract interface
YAGNI (You Aren't Gonna Need It)	Don't implement unneeded things	Don't support unneeded things
KISS (Keep It Simple Stupid)	Keep the implementation simple	Keep the interface simple
Avoid giant files	Extract implementation bits to methods, classes, files	Group related interfaces, and put each group in a separate file

7 ways to keep your interface code clean

1. Use accurate naming

1. Use accurate naming

```
interface Authorization {
  accessToken: string;
  expiresIn: number;
  refreshToken: string;
}

class AuthorizationService {
  getAccessToken(): Promise<Authorization>;
}
```

1. Use accurate naming

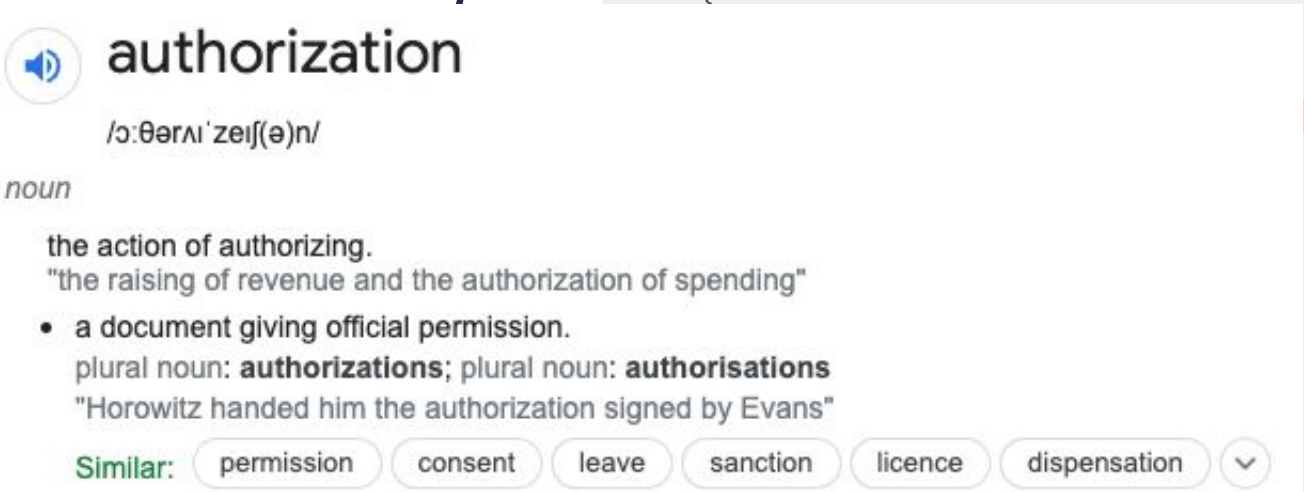
is that really an "authorization"?

```
interface Authorization {
  accessToken: string;
  expiresIn: number;
  refreshToken: string;
}

class AuthorizationService {
  getAccessToken(): Promise<Authorization>;
}
```

1. Use accurate naming

```
interface Authorization {
  accessToken: string;
  expiresIn: number;
  refreshToken: string;
}
```

```
ation>;
```



authorization

/ɔːθərʌɪˈzeɪʃ(ə)n/

noun

the action of authorizing.
"the raising of revenue and the authorization of spending"

- a document giving official permission.
  plural noun: **authorizations**; plural noun: **authorisations**
  "Horowitz handed him the authorization signed by Evans"

Similar: permission consent leave sanction licence dispensation

1. Use accurate naming

```
interface Authorization {
  accessToken: string;
  expiresIn: number;
  refreshToken: string;
}

class AuthorizationService {
  getAccessToken(): Promise<Authorization>;
}
```

1. Use accurate naming

```
interface AccessToken {
  accessToken: string;
  expiresIn: number;
  refreshToken: string;
}

class AuthorizationService {
  getAccessToken(): Promise<AccessToken>;
}
```

1. Use accurate naming

```
interface UseInfiniteScrollOptions<TItem> {
  fetchItems(cursor?: string):
Promise<PaginatedList<TItem>>;
}

interface UseInfiniteScrollResult<TItem> {
  items?: TItem[];
  loading: boolean;
  error?: Error;
  hasMore: boolean;
  fetchMore(): Promise<void>;
}

function useInfiniteScroll<TItem>(
  options: UseInfiniteScrollOptions<TItem>
): UseInfiniteScrollResult<TItem>;
```

1. Use accurate naming

```
interface UseInfiniteScrollOptions<TItem> {
  fetchItems(cursor?: string):
Promise<PaginatedList<TItem>>;
}

interface UseInfiniteScrollResult<TItem> {
  items?: TItem[];
  loading: boolean;
  error?: Error;
  hasMore: boolean;
  fetchMore(): Promise<void>;
}

function useInfiniteScroll<TItem>(
  options: UseInfiniteScrollOptions<TItem>
): UseInfiniteScrollResult<TItem>;
```

what does it have to do with scrolling?

1. Use accurate naming

```
interface UseInfiniteListOptions<TItem> {
  fetchItems(cursor?: string):
Promise<PaginatedList<TItem>>;
}

interface UseInfiniteListResult<TItem> {
  items?: TItem[];
  loading: boolean;
  error?: Error;
  hasMore: boolean;
  fetchMore(): Promise<void>;
}

function useInfiniteList<TItem>(
  options: UseInfiniteListOptions<TItem>
): UseInfiniteListResult<TItem>;
```

2. Drop unused attributes

2. Drop unused attributes

```
interface User {
  id: string;
  firstName: string;
  lastName: string;
  email: string;
  birthDate: Date;
  settings: UserSettings;
  // ... etc.
}

export function UserWelcomeMessage(props: {
user: User }): ReactElement {
  return <p>Hello
{props.user.firstName}!</p>;
}
```

2. Drop unused attributes

not really needed, is it?

```
interface User {
  id: string;
  firstName: string;
  lastName: string;
  email: string;
  birthDate: Date;
  settings: UserSettings;
  // ... etc.
}

export function UserWelcomeMessage(props: {
user: User }): ReactElement {
  return <p>Hello
{props.user.firstName}!</p>;
}
```

2. Drop unused attributes

```
interface User {
  firstName: string;
}

export function UserWelcomeMessage(props: {
user: User }): ReactElement {
  return <p>Hello
{props.user.firstName}!</p>;
}
```

2. Drop unused attributes

```
export interface ListItemProps {
  title?: string;
}

export interface CompanyListItemProps extends
ListItemProps {
  company: {
    name: string;
  };
}

export function CompanyListItem({
  company,
  ...props
}: CompanyListItemProps): ReactElement {
  return <ListItem {...props}
title={company.name} />;
}
```

2. Drop unused attributes

interface says that I can pass "title".

yet, it won't have any effect ☹️

```
export interface ListItemProps {
  title?: string;
}

export interface CompanyListItemProps extends
ListItemProps {
  company: {
    name: string;
  };
}

export function CompanyListItem({
  company,
  ...props
}: CompanyListItemProps): ReactElement {
  return <ListItem {...props}
title={company.name} />;
}
```

2. Drop unused attributes

```
export interface ListItemProps {
  title?: string;
}

export interface CompanyListItemProps extends
Omit<ListItemProps, "title"> {
  company: {
    name: string;
  };
}

export function CompanyListItem({
  company,
  ...props
}: CompanyListItemProps): ReactElement {
  return <ListItem {...props}
title={company.name} />;
}
```

3. Keep it simple stupid

3. Keep it simple stupid

```
interface InfiniteListProps {
  onEndReached?(): void;
  intersectionObserverOptions?: {
    rootMargin?: string;
    threshold?: number | number[];
  };
  children: ReactNode;
}

function InfiniteList<TItem>(props:
InfiniteListProps): ReactElement;
```

what's that intersection observer thing? 🤔

3. Keep it simple stupid

```
interface InfiniteListProps {
  onEndReached?(): void;
  intersectionObserverOptions?: {
    rootMargin?: string;
    threshold?: number | number[];
  };
  children: ReactNode;
}

function InfiniteList<TItem>(props:
InfiniteListProps): ReactElement;
```

3. Keep it simple stupid

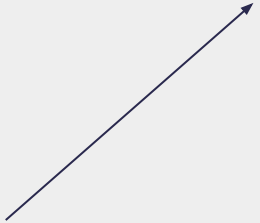
```
interface InfiniteListProps {
  onEndReached?(): void;
  onEndReachedMargin?: number;
  children: ReactNode;
}

function InfiniteList<TItem>(props:
InfiniteListProps): ReactElement;
```

3. Keep it simple stupid

```
interface Task {
  id: string;
  name: string;
  priority: "low" | "high" | "none";
}
```

3. Keep it simple stupid

```ts
interface Task {
  id: string;
  name: string;
  priority: "low" | "high" | "none";
}
```



why `"none"` and not simply `null`? (or `undefined`)

3. Keep it simple stupid

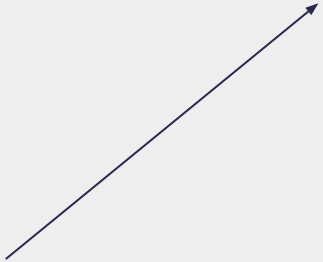
```
interface Task {
  id: string;
  name: string;
  priority: "low" | "high" | null;
}
```

3. Keep it simple stupid

```
function getProducts(): Product[] | null;
```

3. Keep it simple stupid

```
function getProducts(): Product[] | null;
```



why does it return null, while an empty array would suffice? 🤨
is there any difference in meaning between `[]` and `null` in here?

3. Keep it simple stupid

```
function getProducts(): Product[];
```

4. Define interfaces close to their implementation

4. Define interfaces close to their implementation

```
// src/types.d.ts
interface Product {
  id?: string;
  name: string;
  createdAt: Date;
}

// src/domain/product.ts
export function getProduct(id: string):
Promise<Product>;

export function createProduct(input:
Product): Promise<Product>;

// src/api/product.ts
async function resolveProduct(id: string):
GraphQLProduct {
  const product = await getProduct(id);
  return {
    id: product.id!,
    name: product.name
  };
}
```

a global "types" file containing all the types?
looks like a giant "hole" to me,
into which people will just put many unrelated stuff

4. Define interfaces close to their implementation

```ts
// src/types.d.ts
interface Product {
  id?: string;
  name: string;
  createdAt: Date;
}

// src/domain/product.ts
export function getProduct(id: string):
Promise<Product>;

export function createProduct(input:
Product): Promise<Product>;

// src/api/product.ts
async function resolveProduct(id: string):
GraphQLProduct {
  const product = await getProduct(id);
  return {
    id: product.id!,
    name: product.name
  };
}
```

a global "types" file containing all the types?
looks like a giant "hole" to me,
into which people will just put many unrelated stuff

4. Define interfaces close to their implementation

why is that id optional?
is it possible for `getProduct()`
to return a product with no id?

```ts
// src/types.d.ts
interface Product {
  id?: string;
  name: string;
  createdAt: Date;
}

// src/domain/product.ts
export function getProduct(id: string):
Promise<Product>;

export function createProduct(input:
Product): Promise<Product>;

// src/api/product.ts
async function resolveProduct(id: string):
GraphQLProduct {
  const product = await getProduct(id);
  return {
    id: product.id!,
    name: product.name
  };
}
```

a global "types" file containing all the types?
looks like a giant "hole" to me,
into which people will just put many unrelated stuff

4. Define interfaces close to their implementation

why is that id optional?
is it possible for `getProduct()`
to return a product with no id?

If I pass `id` into `createProduct`, will that work?
p.s. why do I have to pass "createdAt";
can't you infer that for me?

```ts
// src/types.d.ts
interface Product {
  id?: string;
  name: string;
  createdAt: Date;
}

// src/domain/product.ts
export function getProduct(id: string):
Promise<Product>;

export function createProduct(input:
Product): Promise<Product>;

// src/api/product.ts
async function resolveProduct(id: string):
GraphQLProduct {
  const product = await getProduct(id);
  return {
    id: product.id!,
    name: product.name
  };
}
```

4. Define interfaces close to their implementation

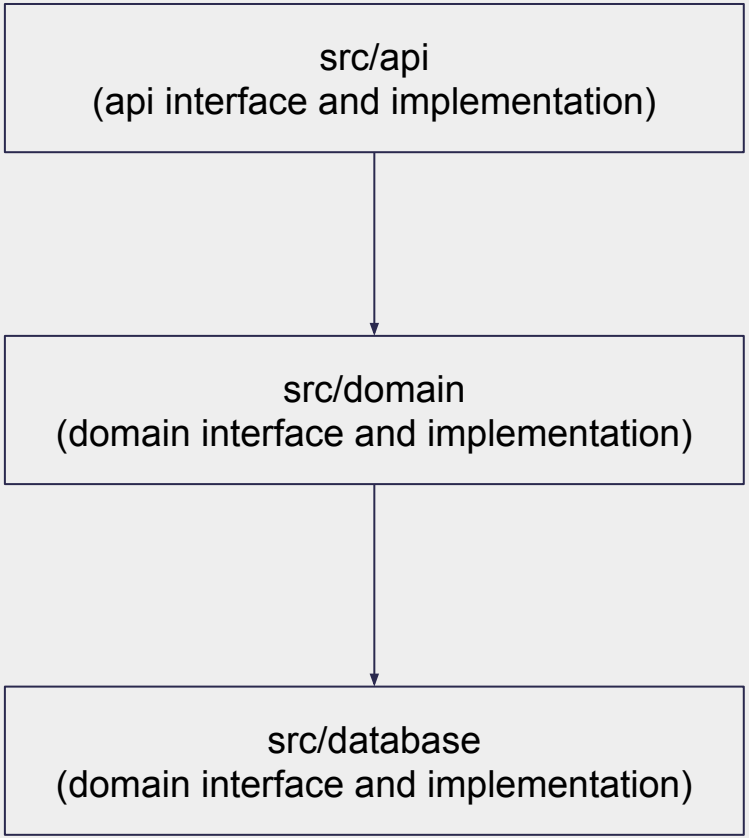
```
// src/domain/product.ts
interface Product {
  id: string;
  name: string;
  createdAt: Date;
}

interface CreateProductInput {
  name: string;
}

export function getProduct(id: string):
Promise<Product>;

export function createProduct(input:
CreateProductInput): Promise<Product>;
```

4. Define interfaces close to their implementation

```
// src/api/product.ts
async function resolveProduct(id: string):
GraphQLProduct {
  return await getProduct(id);
}
```

5. Separate each application layer

5. Separate each application layer



5. Separate each application layer

```
// src/database/product.ts
interface Product { ... }

interface CreateProductInput {
  name: string;
  creatorId: string;
}

export function createProduct(input:
CreateProductInput): Promise<Product>;

// src/domain/product.ts
import { CreateProductInput } from
"src/database/product";

export function createProduct(
  user: User,
  input: CreateProductInput
): Promise<Product>;
```

5. Separate each application layer

why do I have to pass `input.creatorId`?
Can it be different from `user.id` ?

```
// src/database/product.ts
interface Product { ... }

interface CreateProductInput {
  name: string;
  creatorId: string;
}

export function createProduct(input:
CreateProductInput): Promise<Product>;

// src/domain/product.ts
import { CreateProductInput } from
"src/database/product";

export function createProduct(
  user: User,
  input: CreateProductInput
): Promise<Product>;
```

5. Separate each application layer

```ts
// src/database/product.ts
interface Product { ... }

interface CreateProductInput {
  name: string;
  creatorId: string;
}

export function createProduct(input:
CreateProductInput): Promise<Product>;

// src/domain/product.ts
interface CreateProductInput {
  name: string;
}

export function createProduct(
  user: User,
  input: CreateProductInput
): Promise<Product>;
```

6. Abstract away third parties

6. Abstract away third parties

```
function saveFile(file: File):
Promise<AWS.S3.PutObjectOutput> {
  const { bucketName, objectKey } =
getS3FilePath(file);

  return new Promise((resolve, reject) => {
    s3.putObject({
      Bucket: bucketName,
      Key: objectKey
    }, (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    })
  })
}
```

6. Abstract away third parties

such a nice "saveFile" function,
but why does it throw those vast aws.s3
internals at me 😢

```
function saveFile(file: File):
Promise<AWS.S3.PutObjectOutput> {
  const { bucketName, objectKey } =
getS3FilePath(file);

  return new Promise((resolve, reject) => {
    s3.putObject({
      Bucket: bucketName,
      Key: objectKey
    }, (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    })
  })
}
```

6. Abstract away third parties

```
function saveFile(file: File):
Promise<void> {
  const { bucketName, objectKey } =
getS3FilePath(file);

  return new Promise((resolve, reject) => {
    s3.putObject({
      Bucket: bucketName,
      Key: objectKey
    }, (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve();
      }
    })
  })
}
```

6. Abstract away third parties

```
function saveFile(file: File): Promise<{ url:
string }> {
  const { bucketName, objectKey } =
getS3FilePath(file);

  return new Promise((resolve, reject) => {
    s3.putObject(
      {
        Bucket: bucketName,
        Key: objectKey,
      },
      (err, data) => {
        if (err) {
          reject(err);
        } else {
          const url = getS3FileUrl(file);
          resolve({ url });
        }
      }
    );
  });
}
```

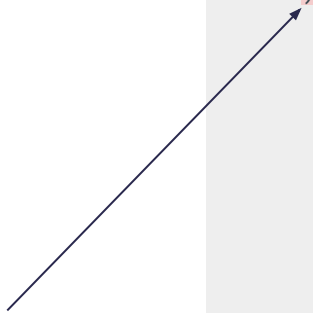
7. Define public interfaces explicitly

7. Define public interfaces explicitly

```ts
interface UsePaginatedListOptions<TItem> {
  // ...
}

function usePaginatedList<TItem>(
  options: UsePaginatedListOptions<TItem>
) {
  const [state, setState] = useState(
    // ...
  );

  // ... long and complex implementation code

  return {
    ...state,
    items,
  }
}
```

7. Define public interfaces explicitly

what does this function return? 🧐
idk, would need to investigate...



```
interface UsePaginatedListOptions<TItem> {
  // ...
}

function usePaginatedList<TItem>(
  options: UsePaginatedListOptions<TItem>
) {
  const [state, setState] = useState(
    // ...
  );

  // ... long and complex implementation code

  return {
    ...state,
    items,
  }
}
```

7. Define public interfaces explicitly

```ts
interface UsePaginatedListOptions<TItem> {
  // ...
}

interface UsePaginatedListResult<TItem> {
  items?: TItem[];
  loading: boolean;
  error?: Error;
  hasMore: boolean;
  fetchMore(): Promise<void>;
}

function usePaginatedList<TItem>(
  options: UsePaginatedListOptions<TItem>
): UsePaginatedListResult<TItem> {
  // ...
}
```

Summing up

Keep your interface code clean

It's just as important as the implementation code.

(... or even more. Implementation is just a detail 😉)

7 ways to keep your interface code clean

1. Use accurate naming
2. Drop unused attributes
3. Keep it simple stupid
4. Define interfaces close to their implementation
5. Separate each application layer
6. Abstract away third parties
7. Define public interfaces explicitly

Thanks!

P.S. We're hiring! 💼
jobs.appunite.com

P.S.2. I'm writing a book about full-stack development!
Subscribe to **jackthenomad.com** to get info about it. 📚