The view rendering component moved from the backend to the client

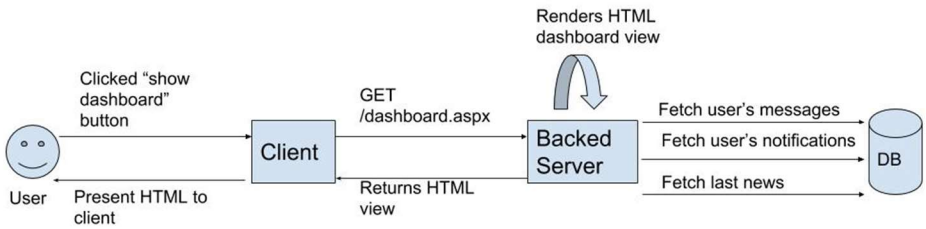
What are the views?
Views are the objects that are shown to the user. They are the building blocks for user interfaces, and they can be HTML pages, iOS UIView, Android View, etc..

The view rendering component

One of the key differences between traditional apps and modern apps, is the place where the views are being rendered.

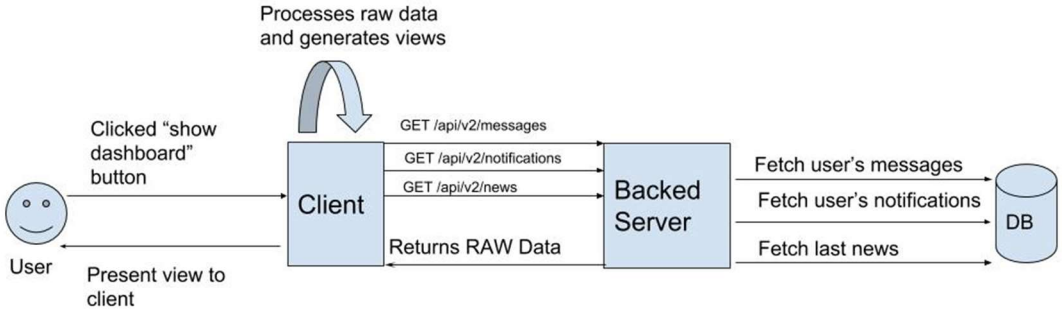
The Rendering Component in Traditional Apps

In the case of traditional apps, the backend server was responsible for generating most of the views.



The Rendering Component in Modern Apps

In modern apps, views are usually rendered by the client itself



What are the elements of the rendering component?

1. Data source: the rendering component consumes data from its data source
2. Data queries: the data is consumed from the data source using queries
3. Consumed data: consumes raw application data in
4. Data access: by design, the rendering component has access to a lot of data and also the option to customize it.
5. Data filtering: should filter the unnecessary data and present only the relevant to the user. There are two types of data filtering:
    a. Data query level:
    The data source itself filters the unnecessary data
    b. Internal level:
    The query returns unnecessary data, that should be filtered by the rendering component before it's presented to the user

Element	**Traditional Apps**	**Modern Apps**
Data Source	Data Base (SQL)	REST APIs
Data Queries	SQL queries	REST API calls
Consumed Data	SQL output	JSON
Data filtering - data query level	Filtered: select * from messages where user_id =... order by 1 limit 5 Unfiltered: select * from messages where user_id =...	Filtered: GET /v2/user/messages?limit=5 Unfiltered: GET /v2/users/messages

Data filtering - internal level	UserMessages user_messages = <SQL Call>
	for (int i=0; i<5; i++)
	{
	UserMessagesTextBox.text+=user_messages[i].toString();
	}
		Client side code to present only 5 messages

Application servers became data sources in modern apps

One of the most important changes of modern apps, is the fact the application servers became data sources.
The app server no longer renders visual views with fixed data, but it is used more as a data source for the rendering component, which is client app (web app , mobile, etc)

The security implications of the new model - Backend as a data source

Since APIs are often used as data sources that are exposed to the internet, every time a developer writes an API endpoint controller, he should ask himself :

- Before fetching an object from the DB: does the client has the right permissions to access the requested object?
- Before returning data to the user: does the client should be exposed to all information of the object?

Wider attack surface

APIs expose more endpoints

In order to fetch all the relevant data for the view, the client accesses multiple endpoints.
For example: In a “Dashboard” view
Traditional apps:

- The client would access GET /dashboard.aspx

Modern apps:

- The client would access:
    - /v2/users/me/notifications
    - /v2/leader_board
    - /v2/last_news

- Another reason that APIs expose more endpoints than traditional apps, is because of CI/CD.

More parameters are sent from the client

1. **Application servers are more stateless**
   Application servers don’t render views, so they are less aware of the client state by design. Also, there is less use of state mechanisms like viewstates and even cookies which represent the client state.

   It leads to a situation where clients usually send more parameters in order to customize the data that will be returned from the server to fit the user’s state. Examples:
   GET /api/v2/news?**filter**=<KEY_WORDS>&**limit**=<Number>&**sort**=desc
   GET /api/v2/company/**4491**/users

2. **Object-based data access is more common**
   Object-based data access is a flow where a client can access a specific object by sending a value that represents it. These values can be numbers, phone numbers, email addresses and more.
   This flow might lead to object level access control issues / IDOR

- Traditional apps:
  - A bit less common: object based data access is less common, because they expose fewer entrance points by design and are more stateful.
  - Security "tricks":
    Tricks are mechanisms to secure object based data access endpoints without actually implementing a proper access control.
    Since traditional apps tend to be more stateful, there were more tricks. For example:
    - The value sent by the user represented an index in a list (1-9 for example) and not the actual ID of the object. The map between the list ID and the actual object ID is stored in the server.
    - The ID of the object is stored in a signed viewstate, that the client can not manipulate without making the server return an error
- Modern apps:
  - Very common: Object based data access exist in almost every API. It's super common to see patterns like:
    - GET /api/v2/user/{user_id}/details
    - GET /api/v2/documents/getDcoument?id={document_id}
    - PUT /api/v2/documents
      {"id":{document_id},"content":"blablabla"}
    - Etc
  - No security tricks:
    APIs tend to be more stateless, and tricks like using a signed viewstate or an index from a list can't really work.

-

XSS

In most of the cases, the rendering component should be responsible to protect the app from Cross Site Scripting attacks. The encoding/filtering of unsafe data before showing it to the user, should be done during the rendering process.

Internal data filtering

- Traditional apps: in many cases an internal data filtering was a good approach.
  For example: once a client sent a request to get a public profile of another user, the BE server would fetch all the user's details from the DB, including sensitive information. Then the backend application would filter out the sensitive data and return only public information (profile picture, first name,..) to the client.
- Modern apps: internal data filtering is not secured. A regular user won't see the filtered data, but an attacker can easily sniff the traffic and obtain all of it.

APIs standards

APIs are data sources that should be consumed by various clients, like mobile devices, web applications, B2B partners, other programmers, etc.. clients might use older or newer client versions.
Because of this nature, APIs usually follow common standards and expose documentation.

The REST protocol encourages developers to build APIs in a very specific structure.
It's a great approach that allows easier frontend development, but also exposes more information about the underlying implementation.

More guessable endpoints
It's easier to find sensitive endpoints.
Administrative endpoints: An attacker who detected a call to "api/users/last_updates" would try to access "api/**admins**/last_updates"
Old endpoint version: It's easier to find old endpoints. Example:
The endpoint "/api/v2/login" requires two factor authentication, while "api/**v1**/login" doesn't

More guessable functions
Let's say that we have a "comments" feature that exposes two functionalities :

- A user can post a comment
- An admin can delete comments

The second feature doesn't implement function level access control, so regular users can access it and delete arbitrary comments.

Common ways for developers to implement the same feature are

	Traditional Apps	**API**
Post new comment (user)	POST /news/111/ new_comment	POST /api/news/111/comments
Delete a comment (admin)	POST /admin_panel/manage_news/delete_comment {“comment_index”:”3”} (Harder to guess)	DELETE /api/news/111/comments/2 (Very easy to guess)

While it’s not that easy for attackers to guess the function path “ /admin_panel/manage_news/delete_comment”, it would be very simple to guess the function method and path in APIs, since it’s structured and standardized.