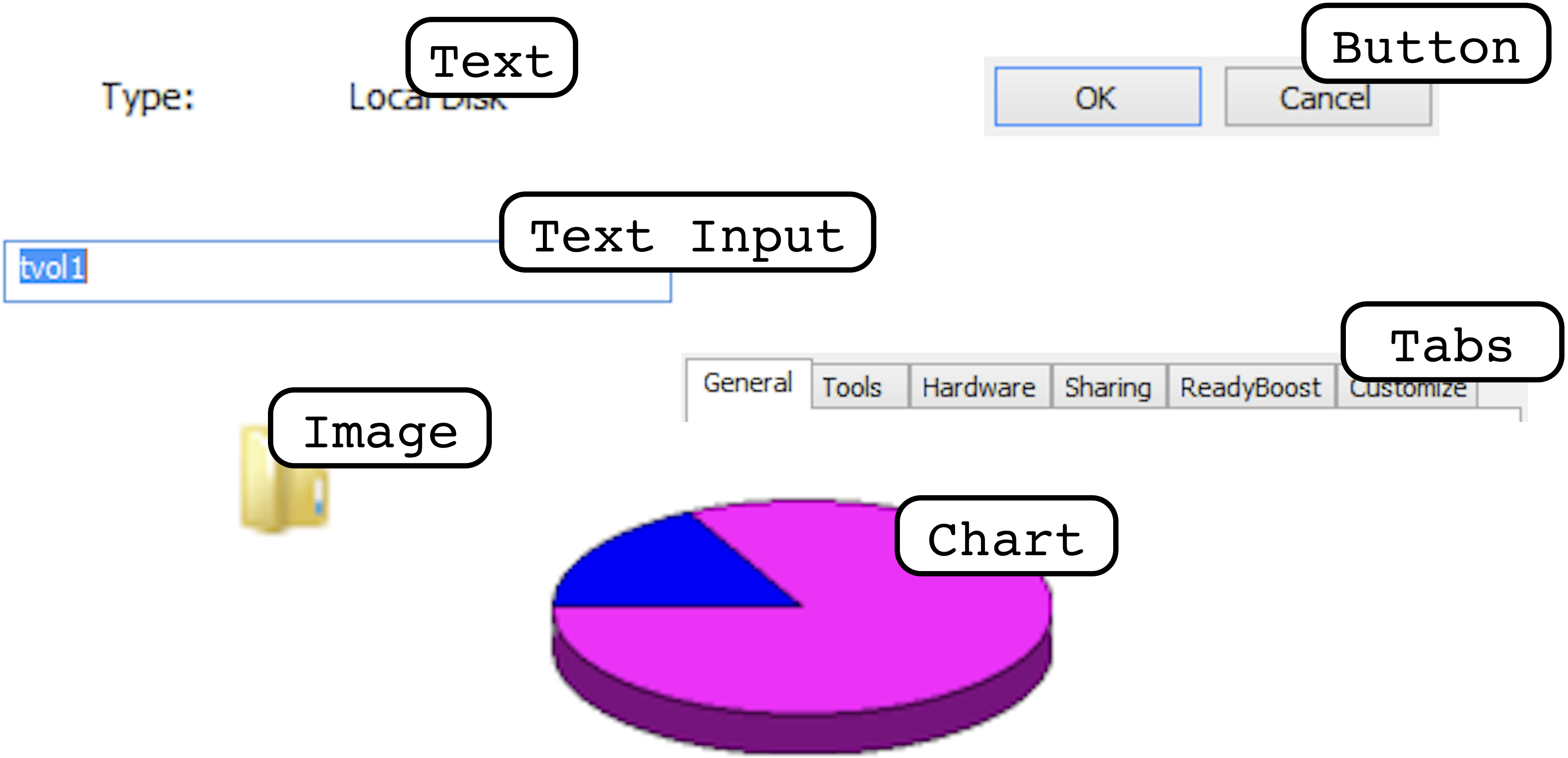
PV252 Lecture 1

Part 1: Element hierarchy

Basic building blocks of user interfaces:

`Elements`, `Views`, `Widgets`, ...

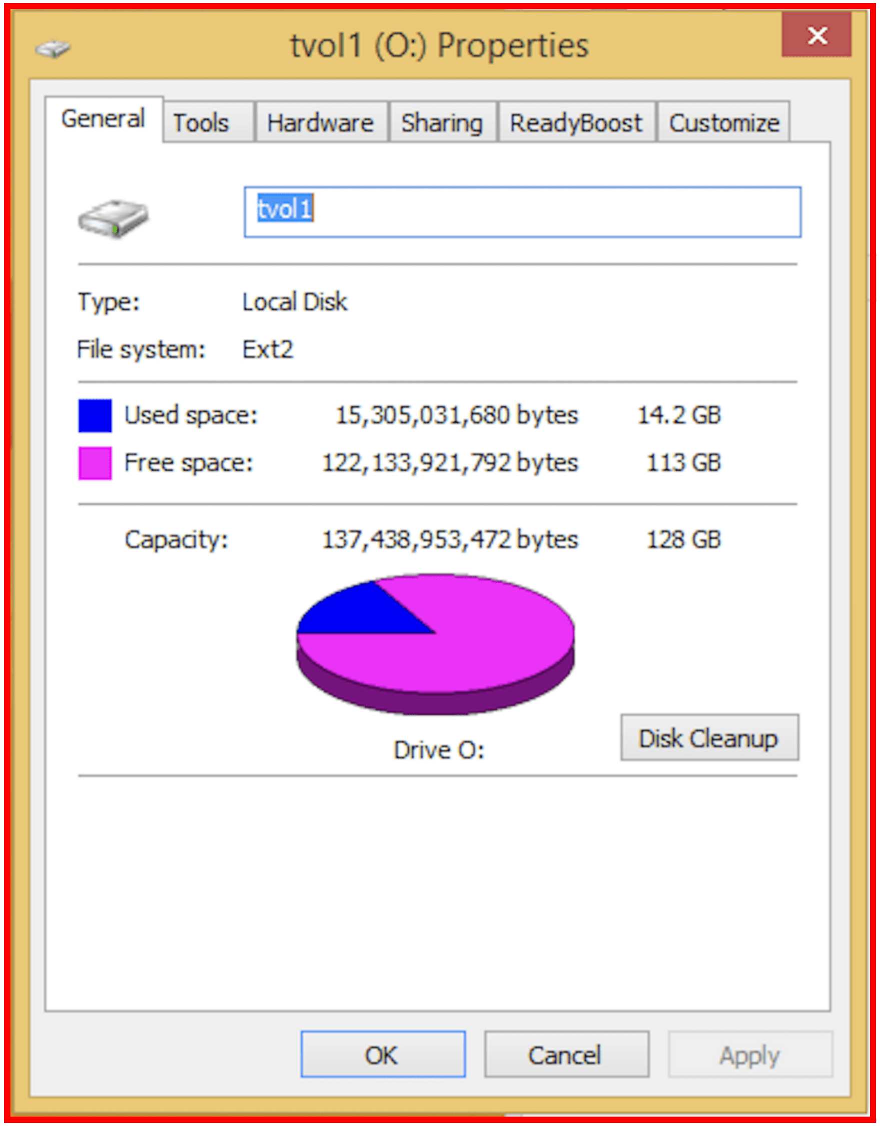


Each `Element` is responsible for how a specific piece of information is rendered.

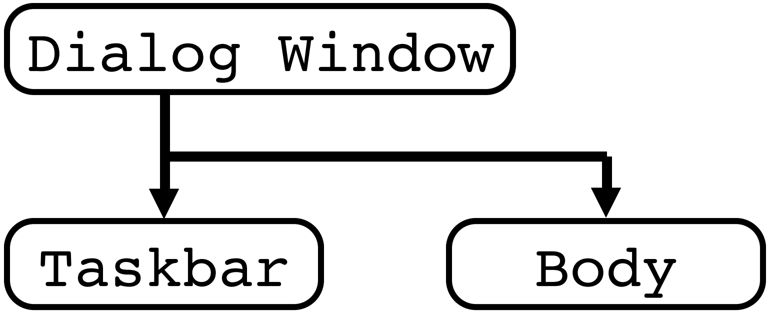
```
// A gross oversimplification...
class Button {

    draw(canvas) {
        canvas.drawRectangle(self.bounds, self.backgroundColor)
        canvas.drawText(self.text, self.centerPoint(), self.fontStyle)
        canvas.drawBitmap(self.icon, self.iconPoint())
    }

}
```

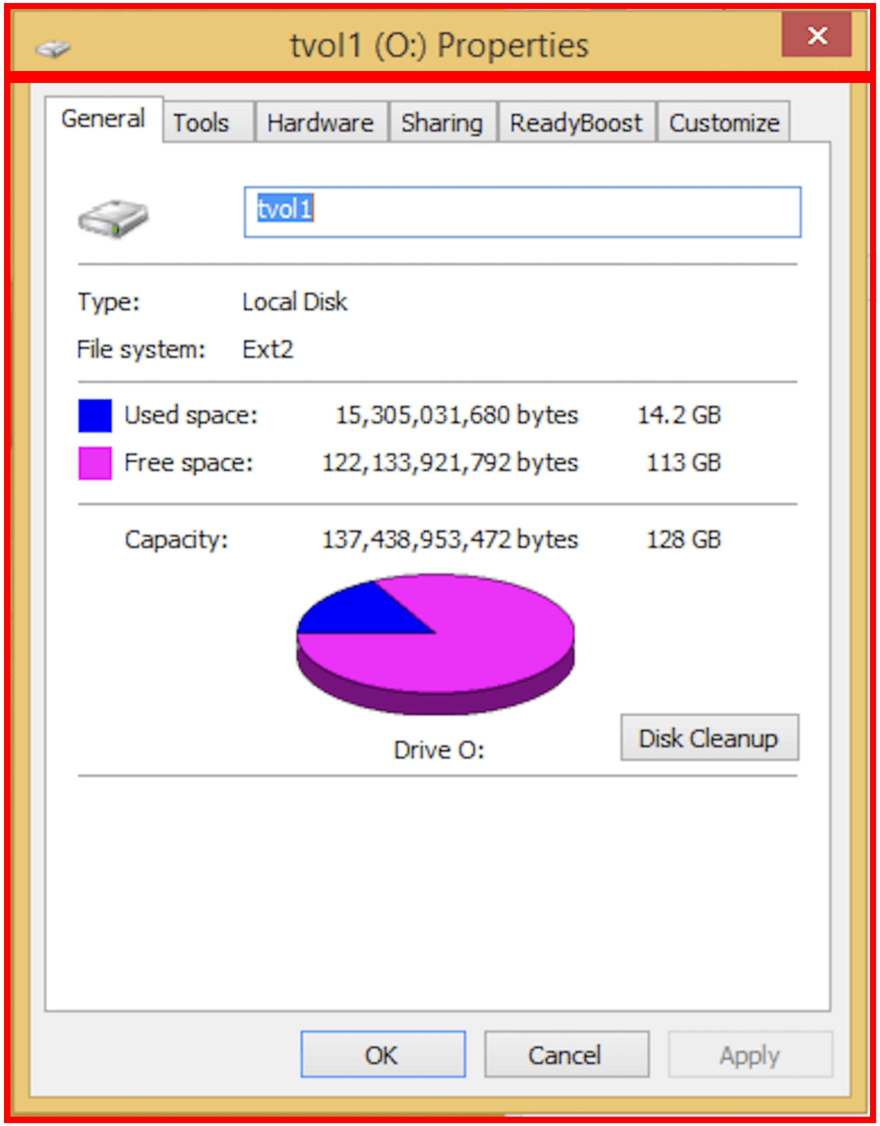
UI Elements are structured as a tree

Dialog Window

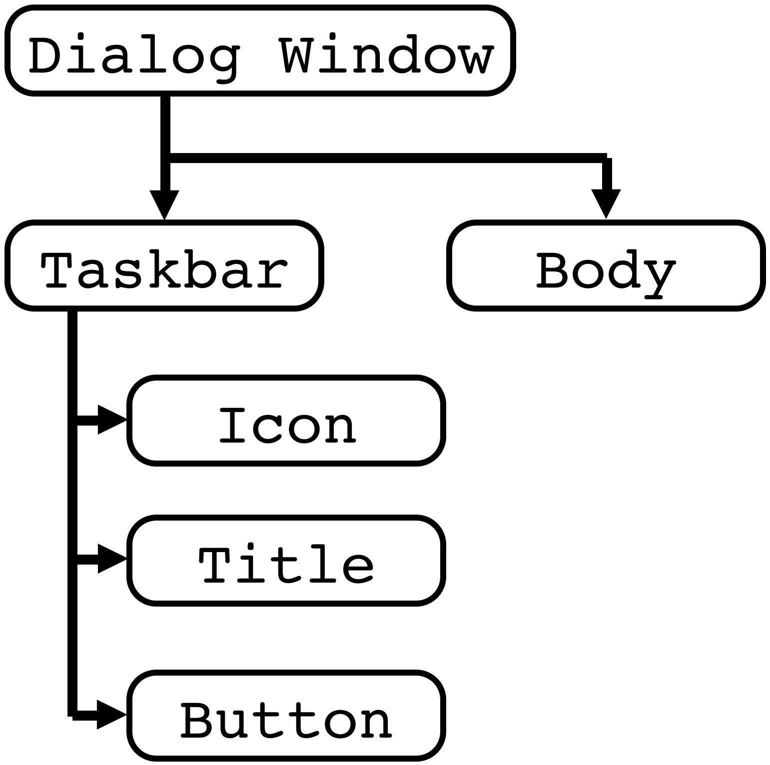


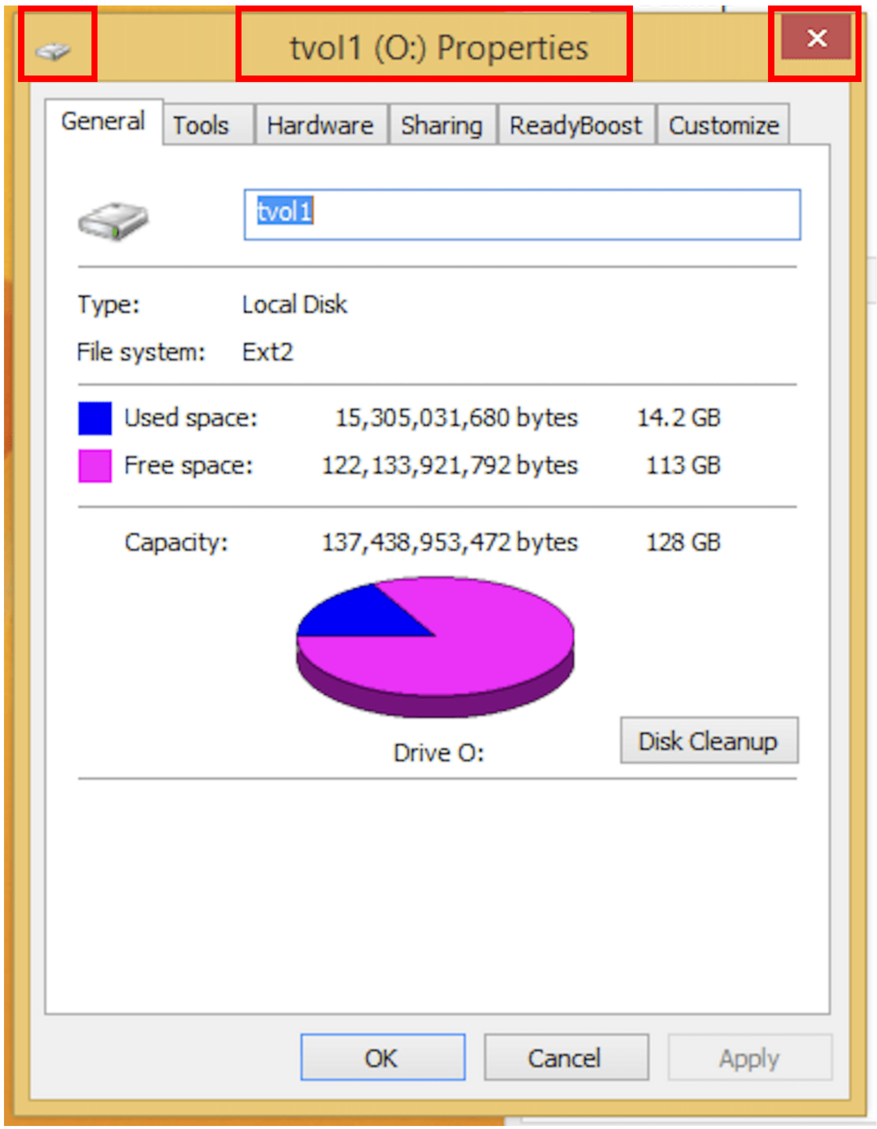
UI Elements are structured as a tree



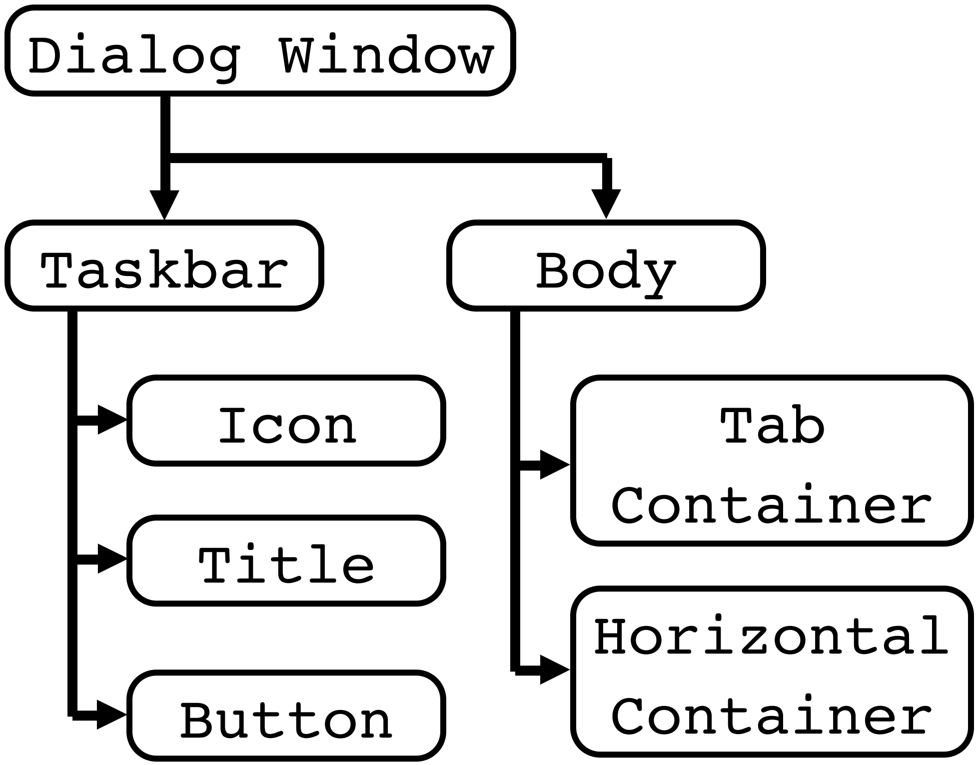


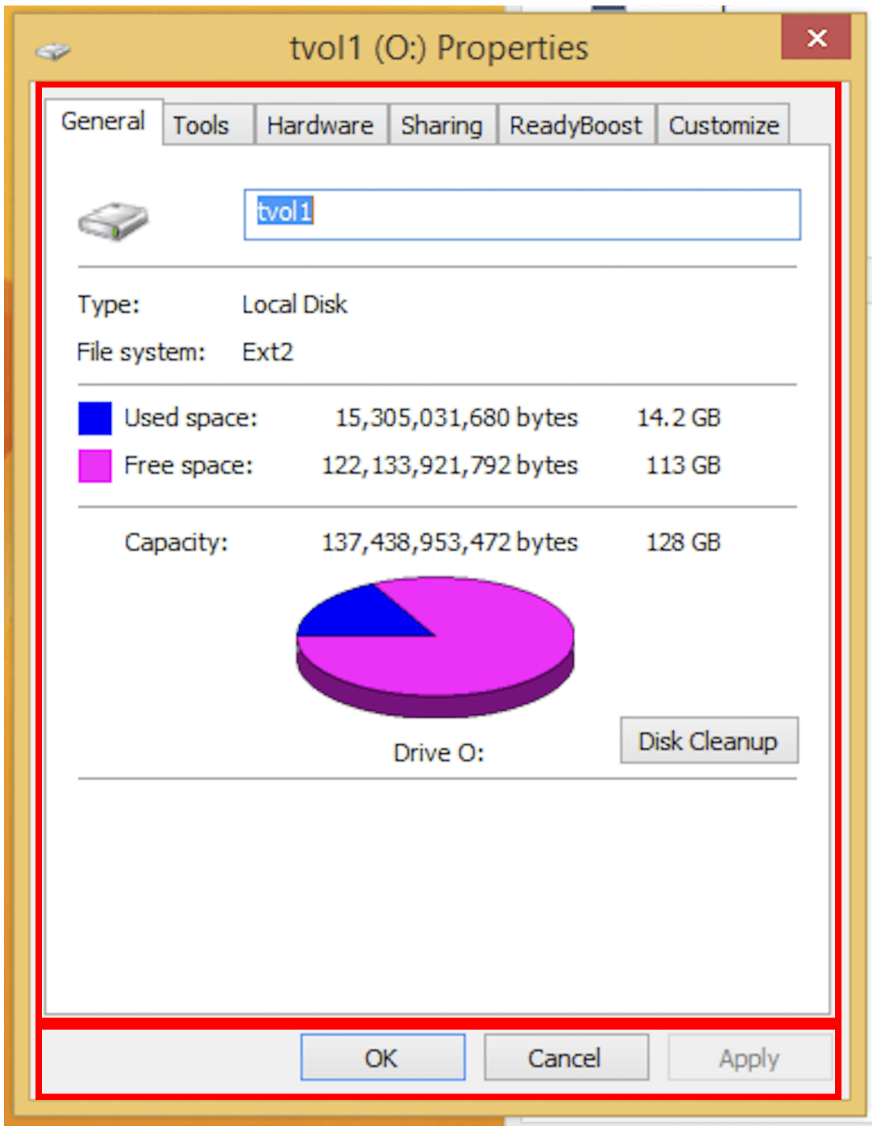
UI Elements are structured as a tree.



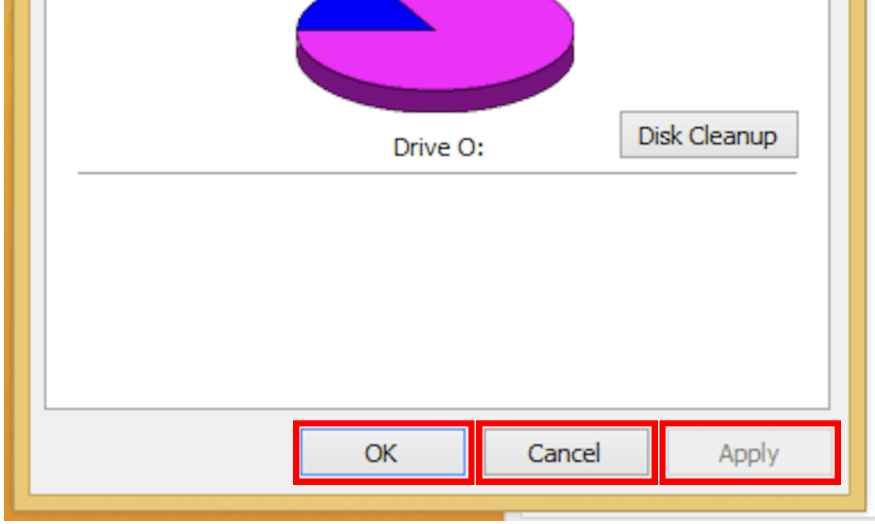


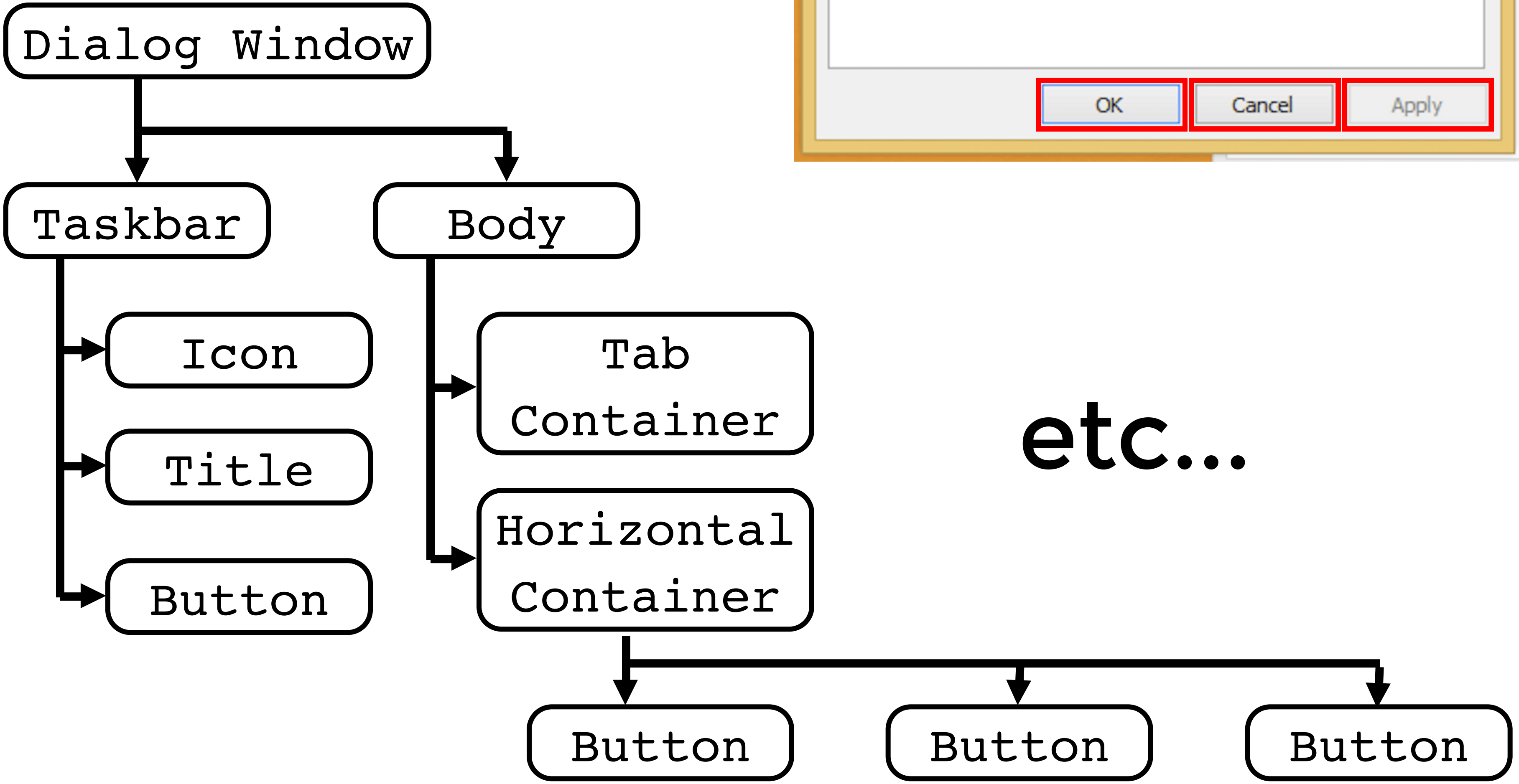
UI Elements are structured as a tree.





UI Elements are structured as a tree.





etc...

Part 2: Basis of interactivity

So far, the Element hierarchy is completely *static*… how does user interaction come into play?

So far, the Element hierarchy is completely *static*... how does user interaction come into play?

`Event`

Events

Events

- `Events` are generated *(mostly)* in relation to input devices or some other external "stimuli".

Events

- `Events` are generated *(mostly)* in relation to input devices or some other external "stimuli".
- `Event` is delivered by the "platform" (browser, OS, etc.) to an `Element` based on its position, focus, etc.

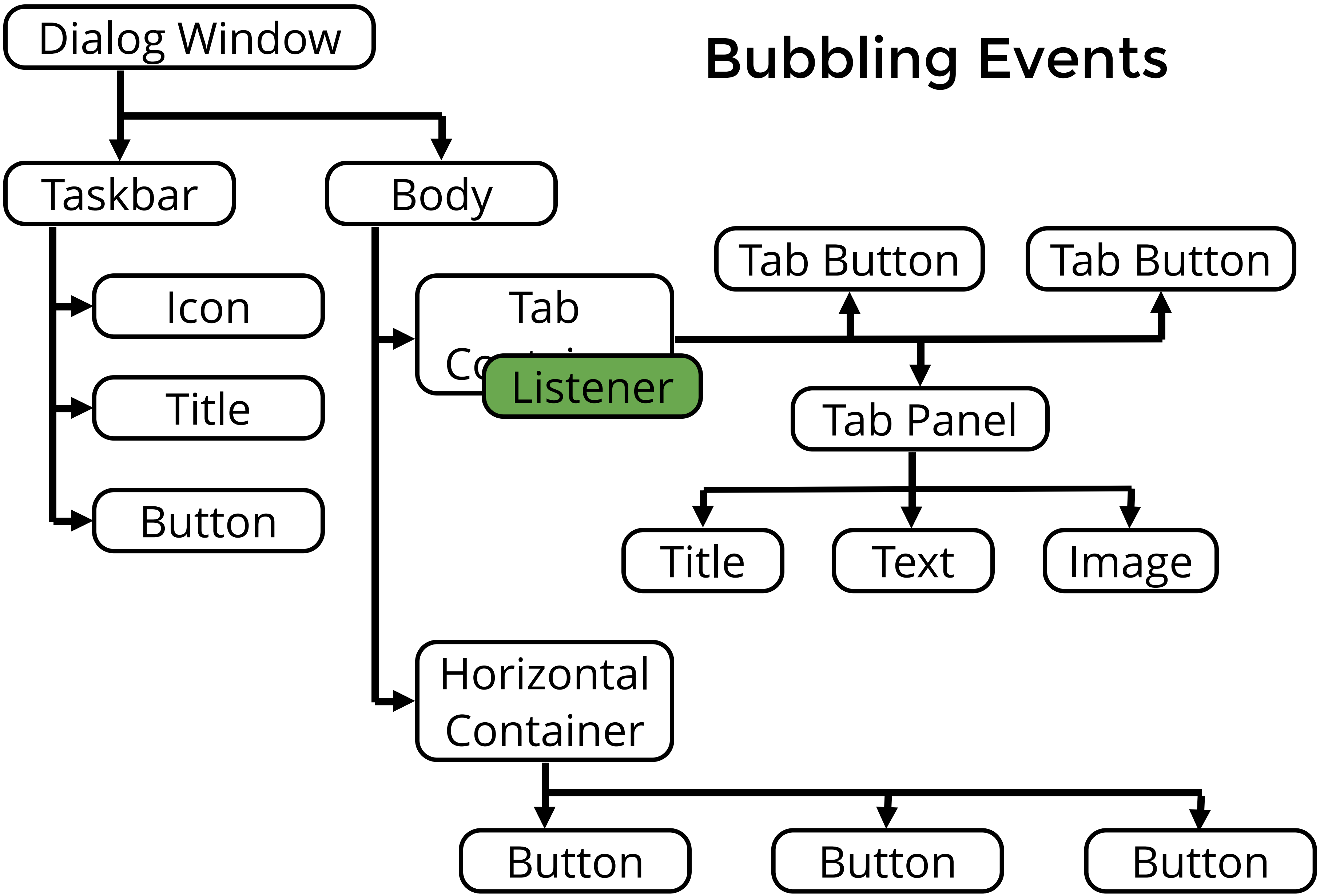
Events

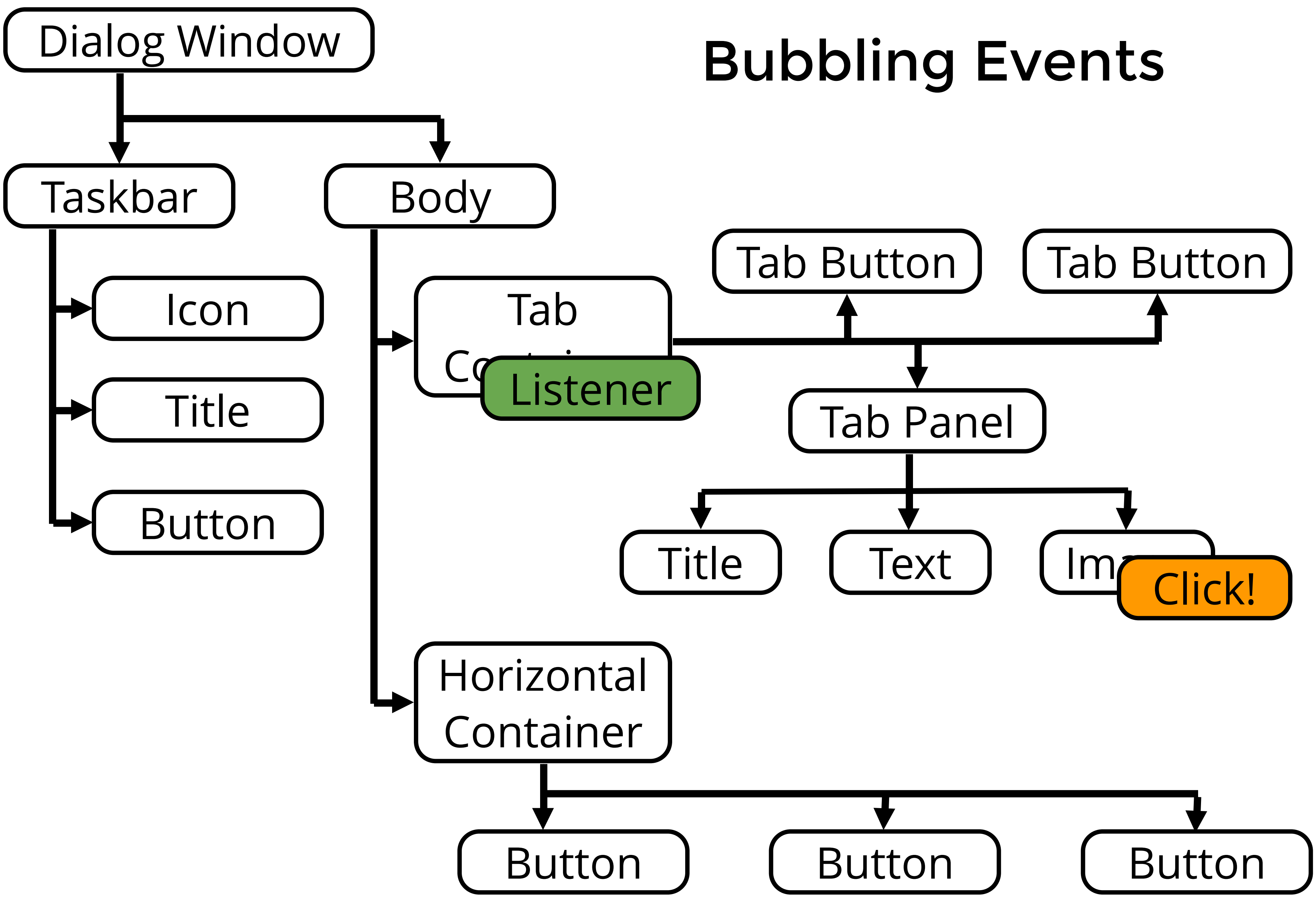
- `Events` are generated *(mostly)* in relation to input devices or some other external "stimuli".
- `Event` is delivered by the "platform" (browser, OS, etc.) to an `Element` based on its position, focus, etc.
- A *capturing* `Event` is delivered to an `Element` and that `Element` either consumes the `Event` it or not.

Events

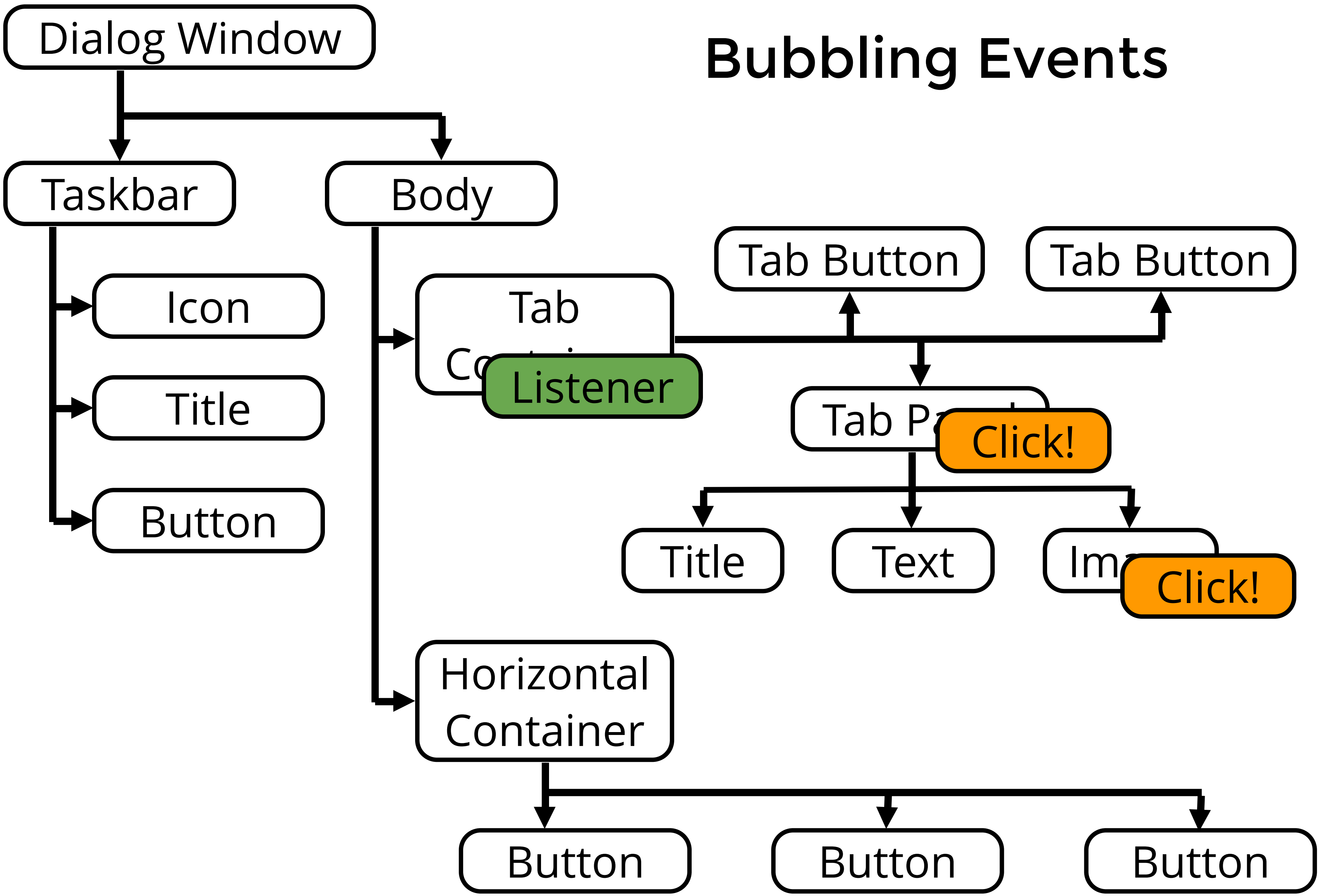
- `Events` are generated *(mostly)* in relation to input devices or some other external "stimuli".
- `Event` is delivered by the "platform" (browser, OS, etc.) to an `Element` based on its position, focus, etc.
- A *capturing* `Event` is delivered to an `Element` and that `Element` either consumes the `Event` it or not.
- A *bubbling* `Event` is delivered to an `Element` and if it is not consumed by that `Element`, it is propagated upwards in the Element tree.

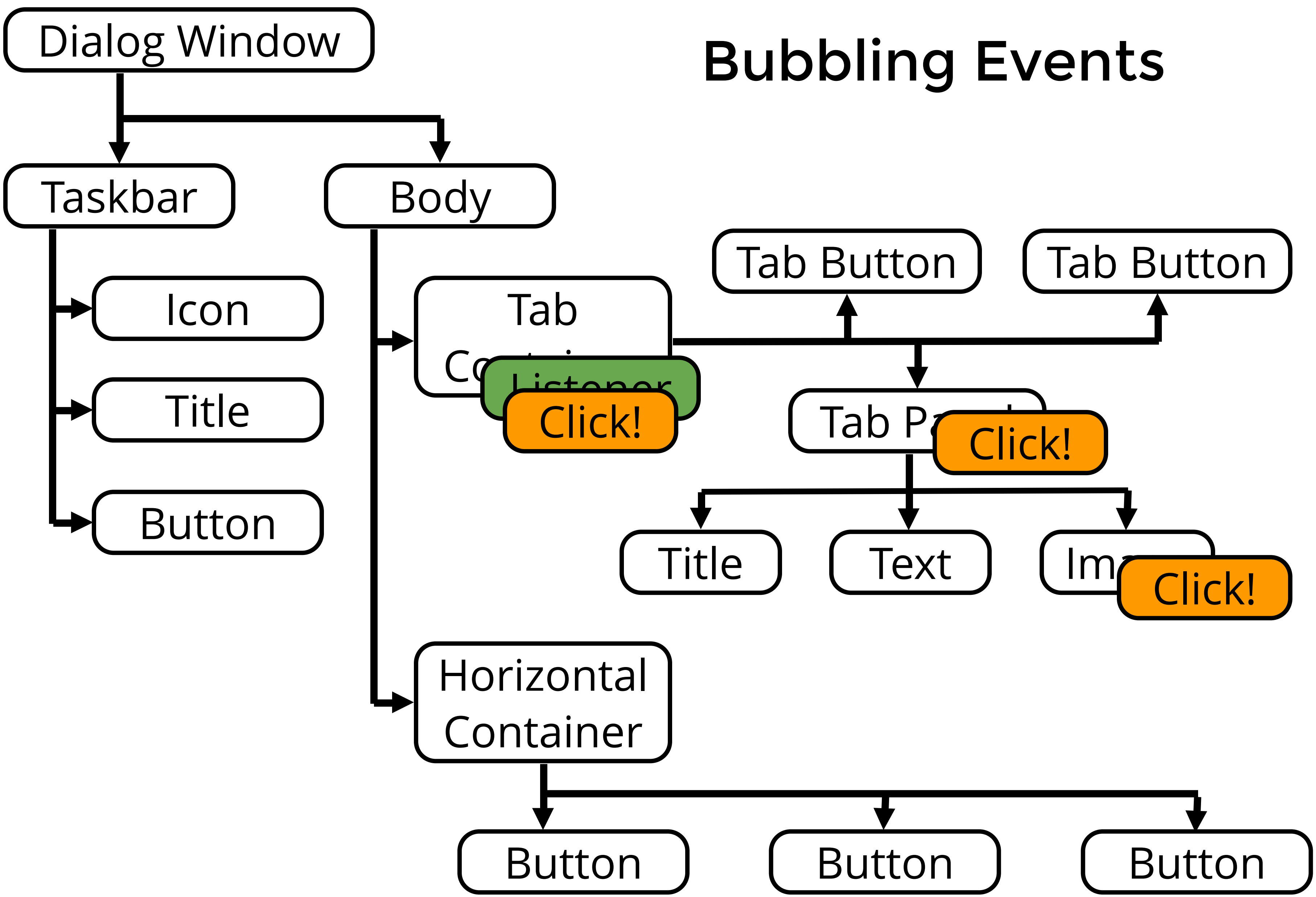
Bubbling Events

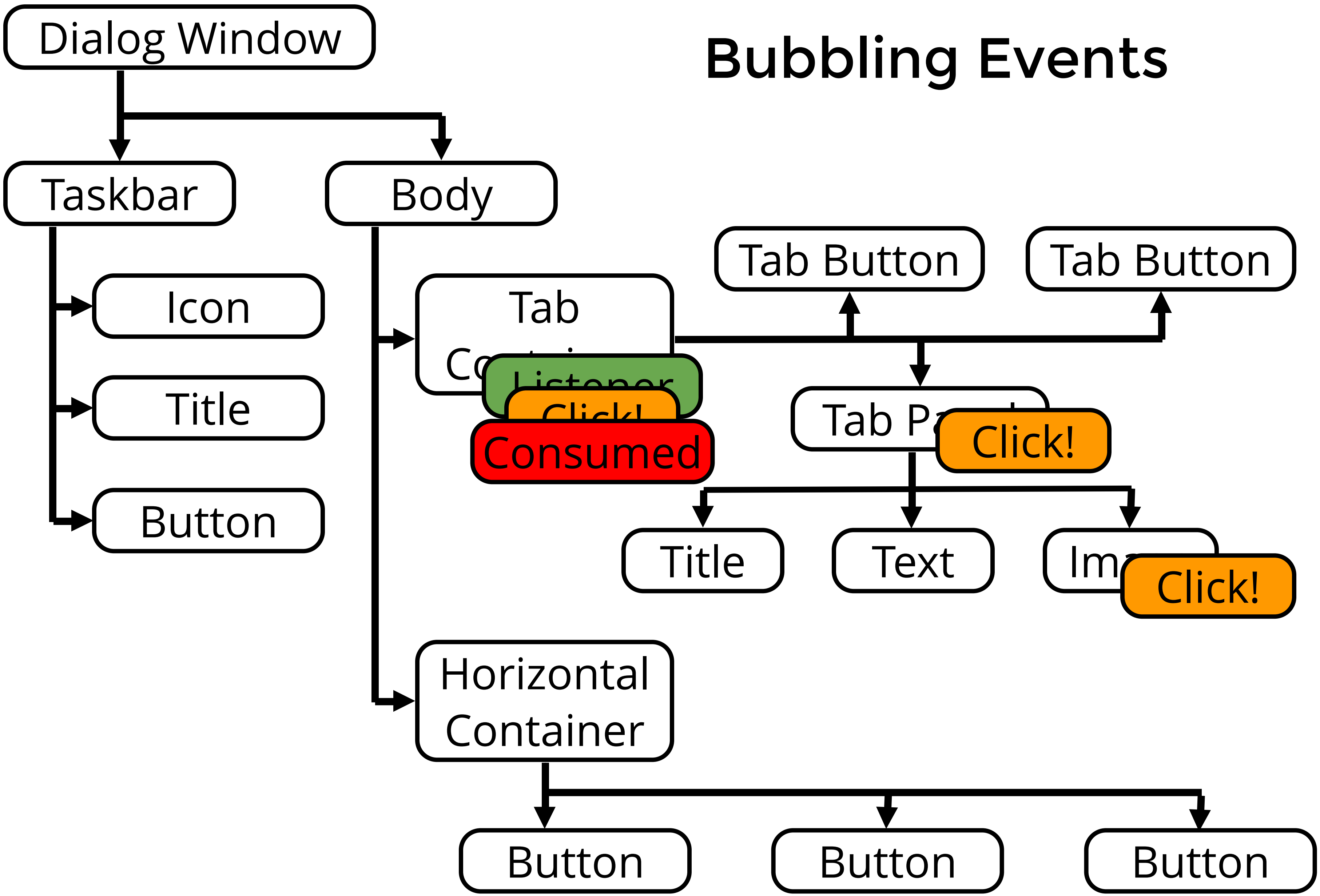



Bubbling Events
Dialog Window
Taskbar
Body
Icon
Title
Button
Tab Container
Listener
Tab Button
Tab Button
Tab Panel
Title
Text
Image
Click!
Horizontal Container
Button
Button
Button

Bubbling Events




Dialog Window
Bubbling Events
Taskbar
Body
Icon
Title
Button
Tab Button
Tab Button
Tab
Click!
Tab Pa
Click!
Title
Text
Im
Click!
Horizontal Container
Button
Button
Button


Bubbling Events
Dialog Window
Taskbar
Body
Icon
Title
Button
Tab Button
Tab Button
Tab
Consumed
Click!
Title
Text
Click!
Horizontal Container
Button
Button
Button

Events [edit]

HTML events [edit]

Common events [edit]

There is a huge collection of events that can be generated by most element nodes:

- Mouse events.[3][4]
- Keyboard events.
- HTML frame/object events.
- HTML form events.
- User interface events.
- Mutation events (notification of any changes to the structure of a document).
- Progress events[5] (used by XMLHttpRequest and File API[6]).

Note that the event classification above is not exactly the same as W3C's classification.

Category	Type	Attribute	Description	Bubbles	Cancelable
	click	onclick	Fires when the pointing device button is clicked over an element. A click is defined as a mousedown and mouseup over the same screen location. The sequence of these events is: • mousedown • mouseup • click	Yes	Yes
	dblclick	ondblclick	Fires when the pointing device button is double-clicked over an element	Yes	Yes
	mousedown	onmousedown	Fires when the pointing device button is pressed over an element	Yes	Yes
	mouseup	onmouseup	Fires when the pointing device button is released over an element	Yes	Yes
	mouseover	onmouseover	Fires when the pointing device is moved onto an element	Yes	Yes
	mousemove[7]	onmousemove	Fires when the pointing device is moved while it is over an element	Yes	Yes

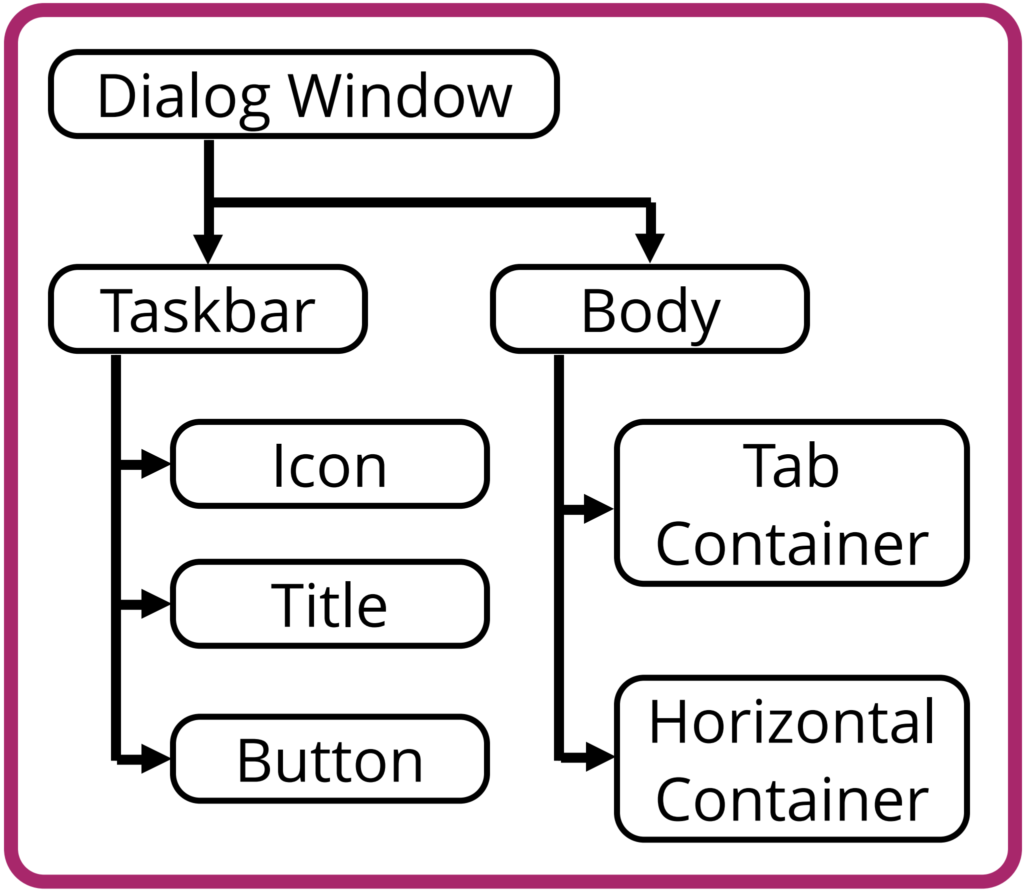
Part 3: State hierarchy

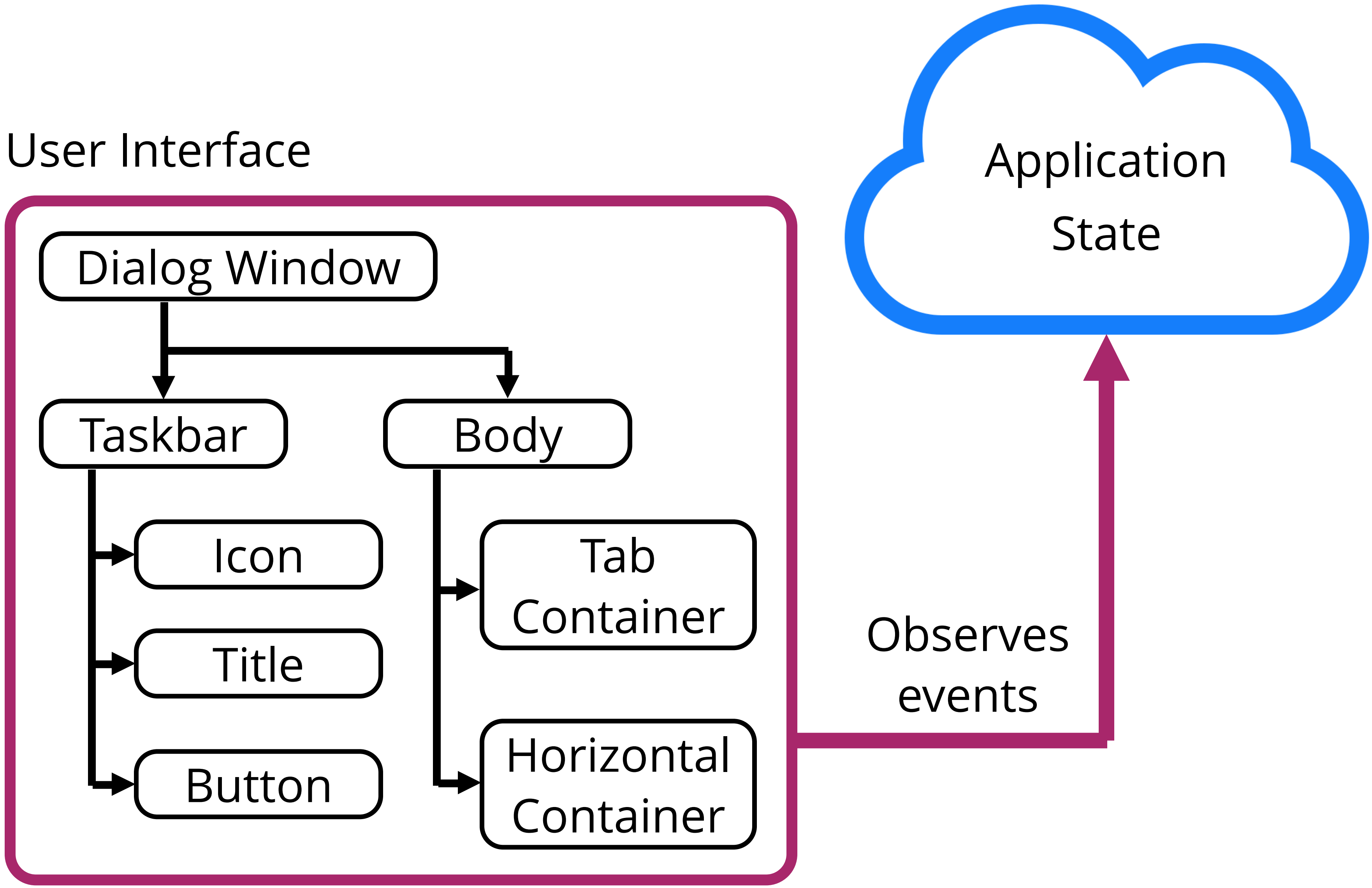
So what happens if an `Event` is consumed?

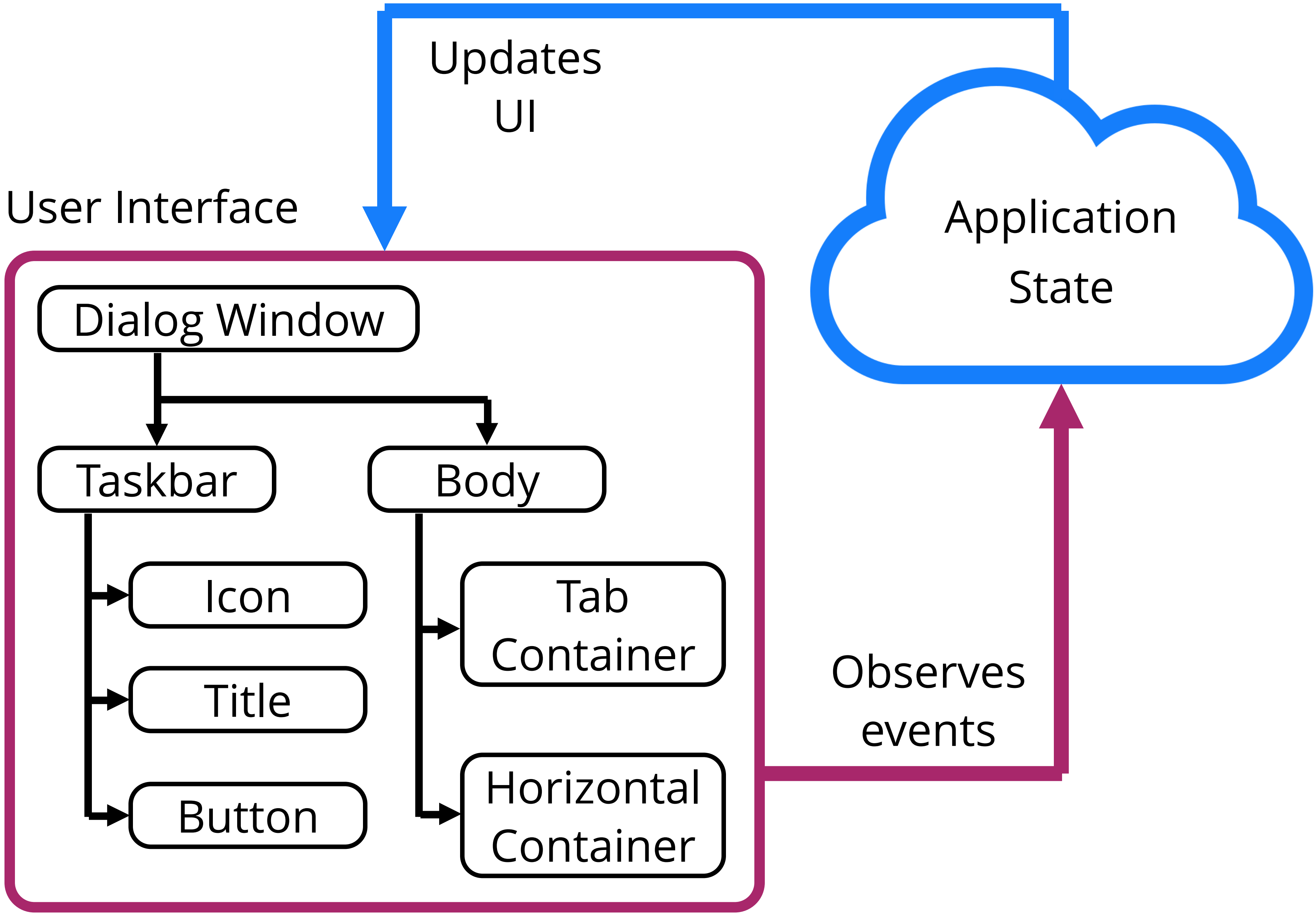
So what happens if an `Event` is consumed?

Until this point, the vast majority of UI systems agree on this architecture, even if they sometimes have abstractions to hide it from the programmer...

User Interface




User Interface
Dialog Window
Taskbar
Body
Icon
Title
Button
Tab Container
Horizontal Container
Application State
Observes events


Updates
UI
User Interface
Application
State
Dialog Window
Taskbar
Body
Icon
Title
Button
Tab
Container
Horizontal
Container
Observes
events

These are the most fundamental questions that
UI frameworks/architectures are trying to answer:

- Where is application state stored?
- How is application state updated?
- How is state change propagated to the user interface?

A useful framework for thinking about state...

(what goes into each category depends on the application)

Element state

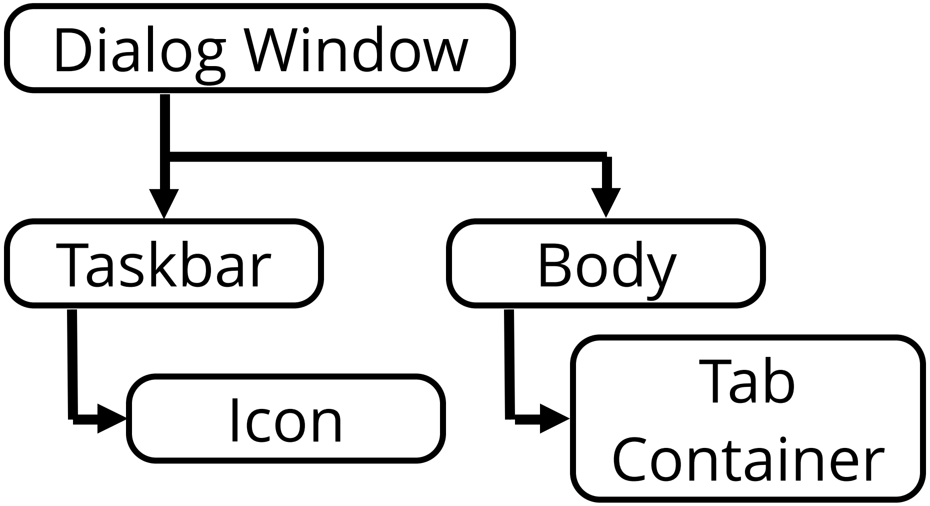
- "Stuff that you lose if you remove an element."
- Position, scroll location, input field values, ...
- Stored by the individual UI `Elements`.

A useful framework for thinking about state...

(what goes into each category depends on the application)

Element state

- "Stuff that you lose if you remove an element."
- Position, scroll location, input field values, ...
- Stored by the individual UI `Elements`.

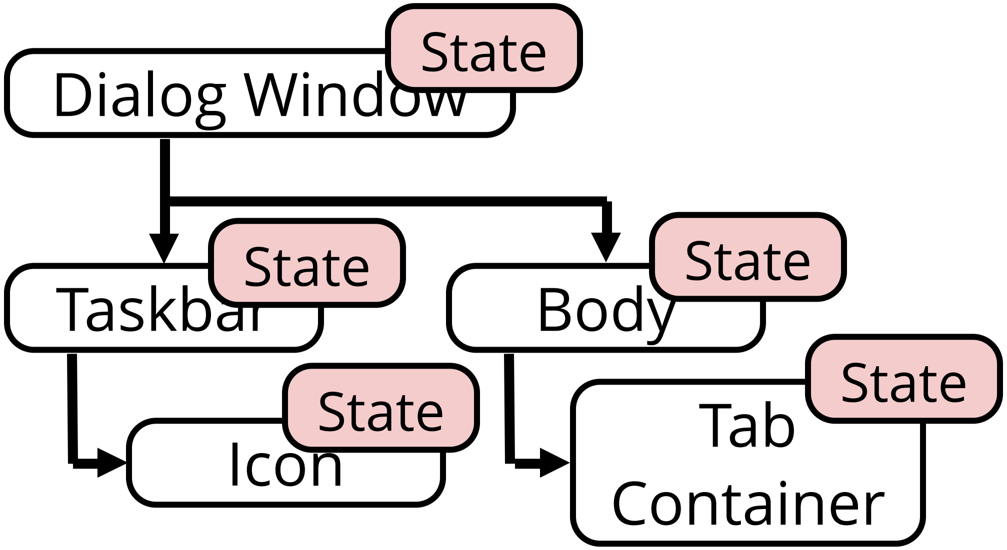


A useful framework for thinking about state...

(what goes into each category depends on the application)

Element state

- "Stuff that you lose if you remove an element."
- Position, scroll location, input field values, ...
- Stored by the individual UI `Elements`.



A useful framework for thinking about state...

(what goes into each category depends on the application)

Session state

- "Stuff that you lose if you close the window."
- E-shop cart, login credentials, navigation...
- Stored by the browser or by the server (and linked to browser using session cookies).
- Session ends when the browser says it ends (for "normal" apps, it's usually when you close the app).

A useful framework for thinking about state...

(what goes into each category depends on the application)

Session state

- "Stuff that you lose if you close the window."
- E-shop cart, login credentials, navigation...
- Stored by the browser or by the server (and linked to browser using session cookies).
- Session ends when the browser says it ends (for "normal" apps, it's usually when you close the app).

Browser

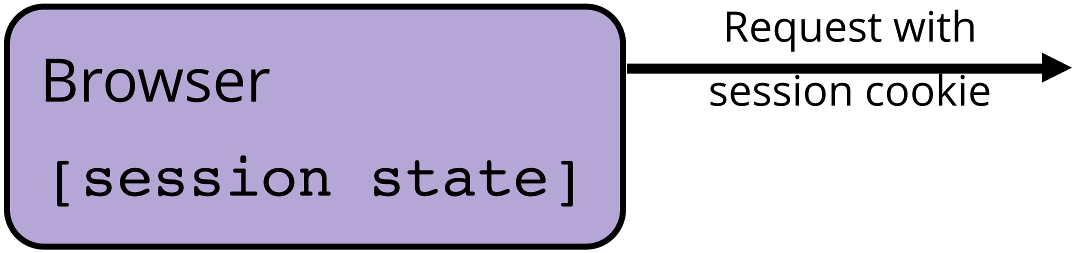
`[session state]`

A useful framework for thinking about state...

(what goes into each category depends on the application)

Session state

- "Stuff that you lose if you close the window."
- E-shop cart, login credentials, navigation...
- Stored by the browser or by the server (and linked to browser using session cookies).
- Session ends when the browser says it ends (for "normal" apps, it's usually when you close the app).

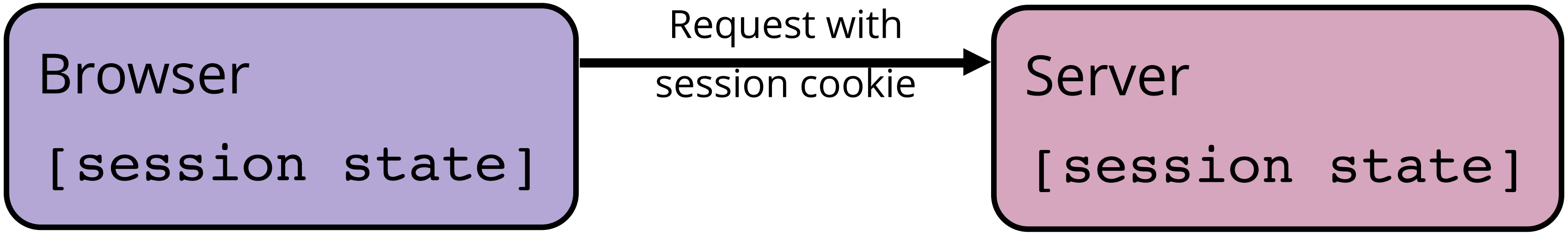


A useful framework for thinking about state...

(what goes into each category depends on the application)

Session state

- "Stuff that you lose if you close the window."
- E-shop cart, login credentials, navigation...
- Stored by the browser or by the server (and linked to browser using session cookies).
- Session ends when the browser says it ends (for "normal" apps, it's usually when you close the app).

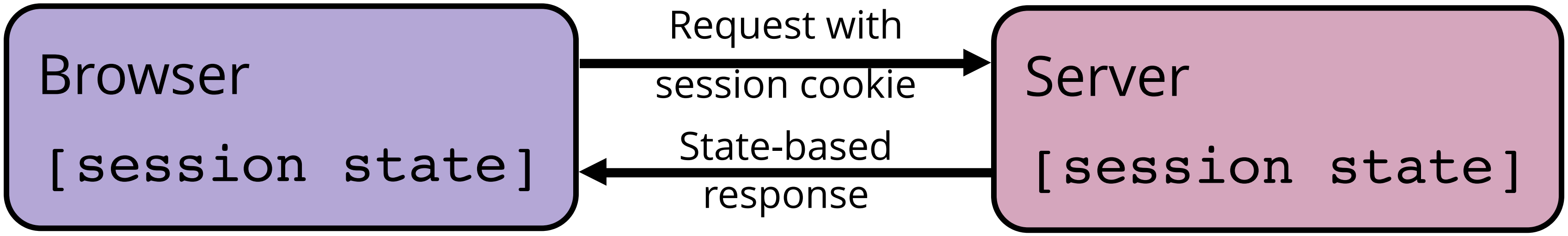


A useful framework for thinking about state...

(what goes into each category depends on the application)

Session state

- "Stuff that you lose if you close the window."
- E-shop cart, login credentials, navigation...
- Stored by the browser or by the server (and linked to browser using session cookies).
- Session ends when the browser says it ends (for "normal" apps, it's usually when you close the app).



A useful framework for thinking about state...

(what goes into each category depends on the application)

Persistent state

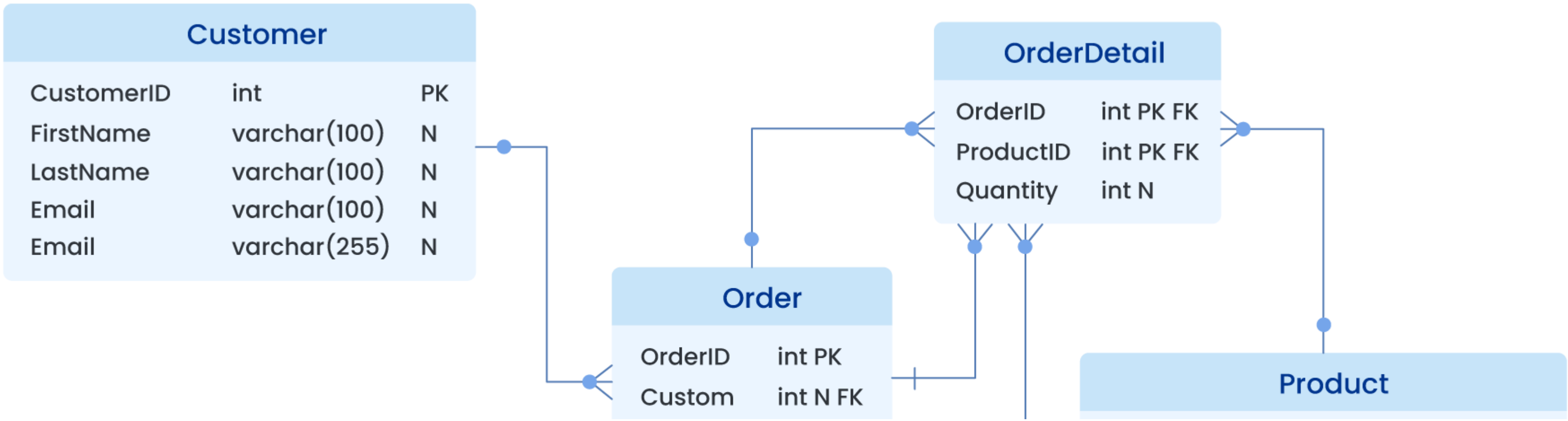
- "Stuff that you lose if the hard drive breaks down."
- Account information, settings, "content", ...
- Stored in files or databases, both within the browser (locally) or on a server (remotely).

A useful framework for thinking about state...

(what goes into each category depends on the application)

Persistent state

- "Stuff that you lose if the hard drive breaks down."
- Account information, settings, "content", ...
- Stored in files or databases, both within the browser (locally) or on a server (remotely).



Element state

"what the UI looks like right now"

Session state

"what the user is doing"

Persistent state

"what the user has saved"

Part 4: Client vs. Server

Client-side rendering

Server-side rendering

Not rendering in the GPU sense, but
"where is state turned into HTML".

Client-side rendering

Server-s

We started here

rendering

Not rendering in the GPU sense, but
"where is state turned into HTML".

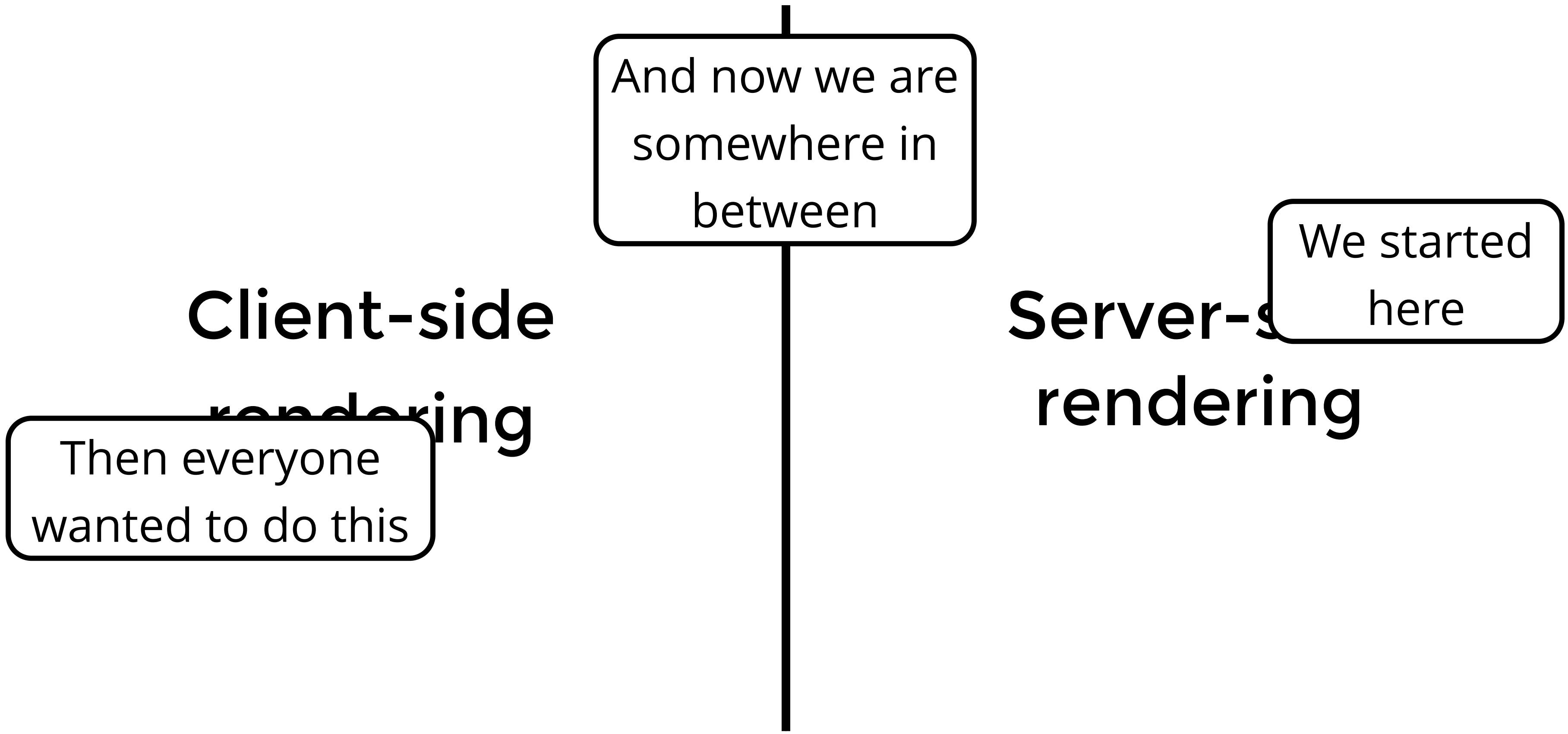
Client-side rendering

Then everyone wanted to do this

Server-s rendering

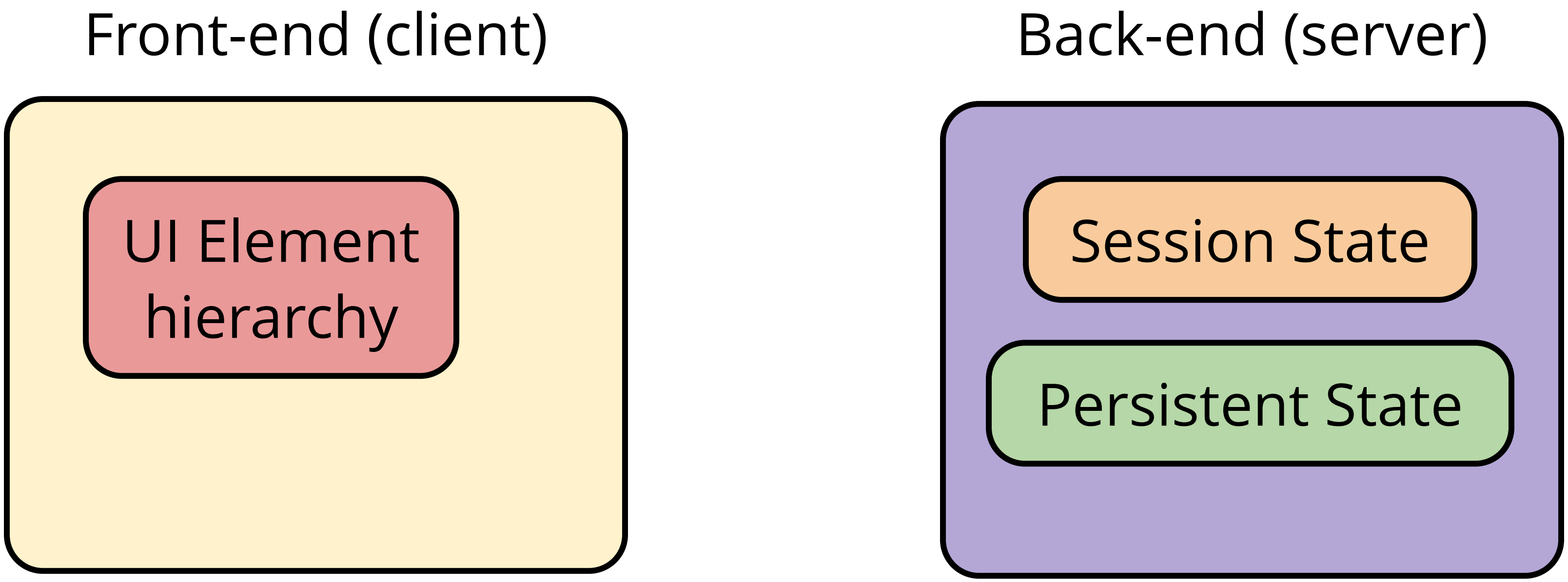
We started here

Not rendering in the GPU sense, but *"where is state turned into HTML"*.

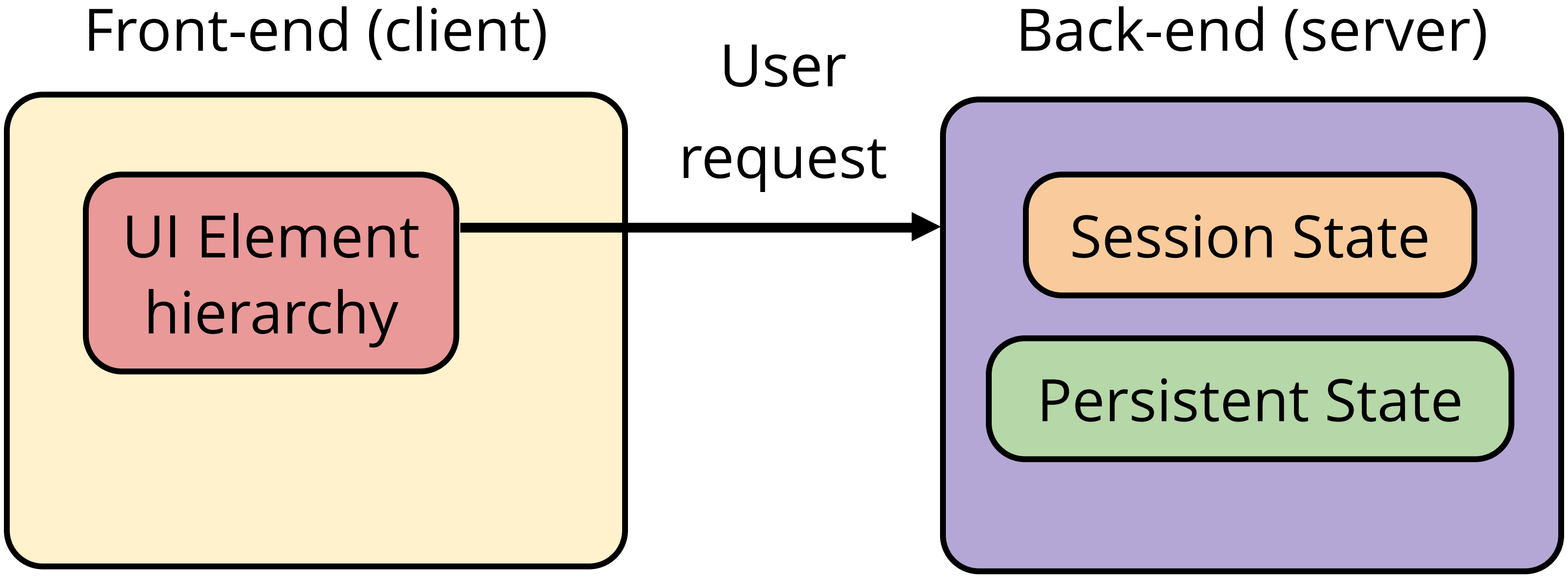


Not rendering in the GPU sense, but *"where is state turned into HTML"*.

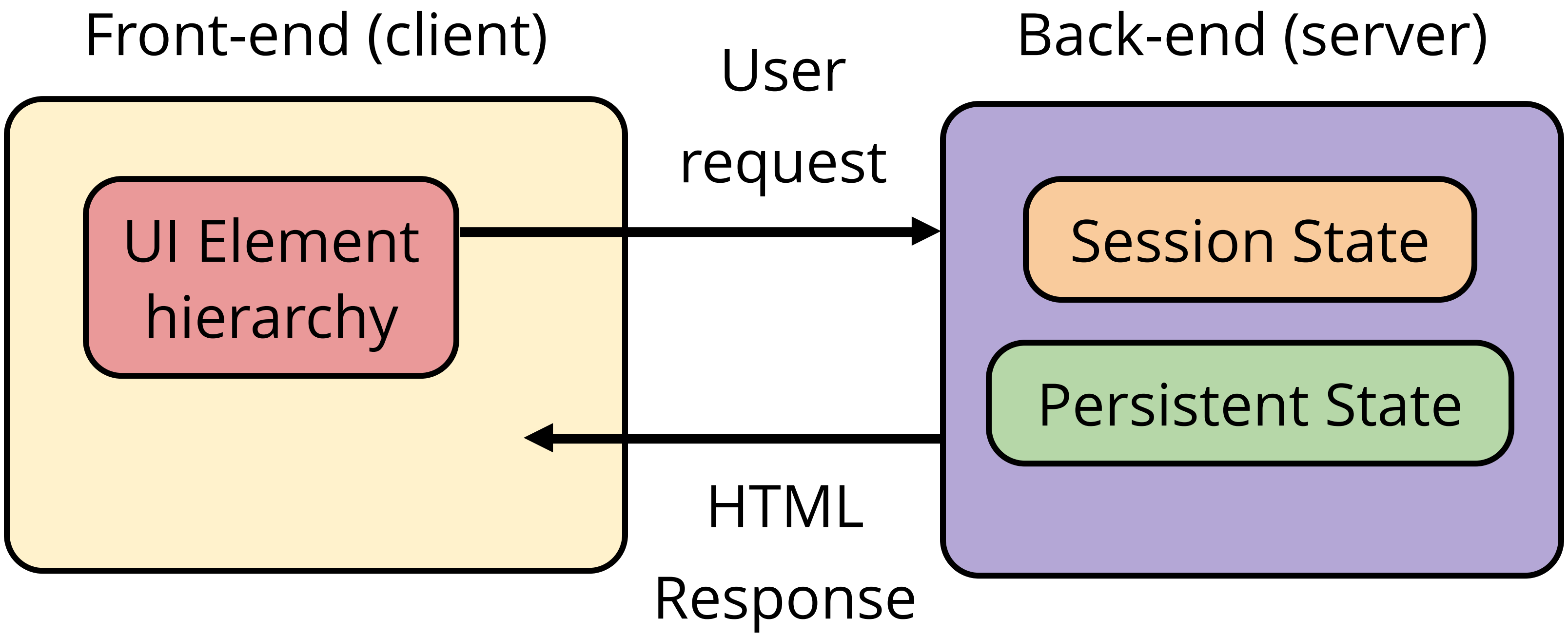
Server-side rendering



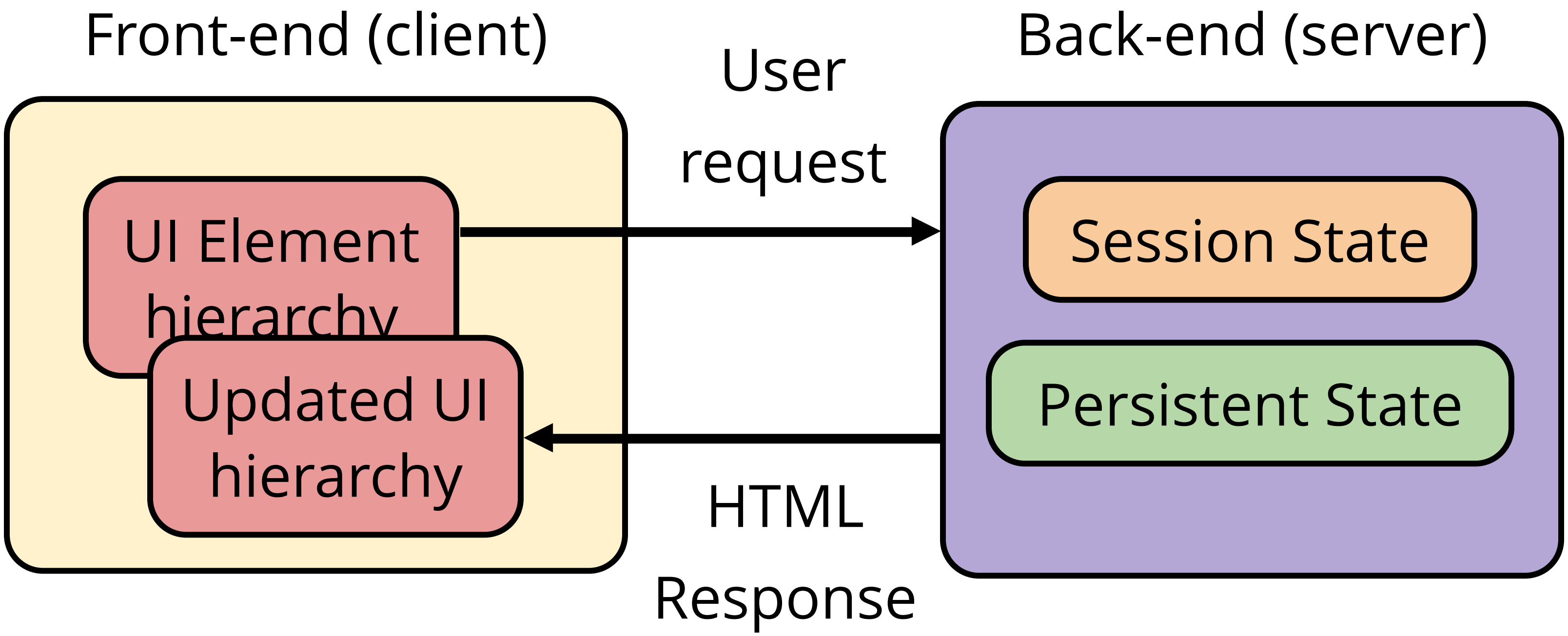
Server-side rendering



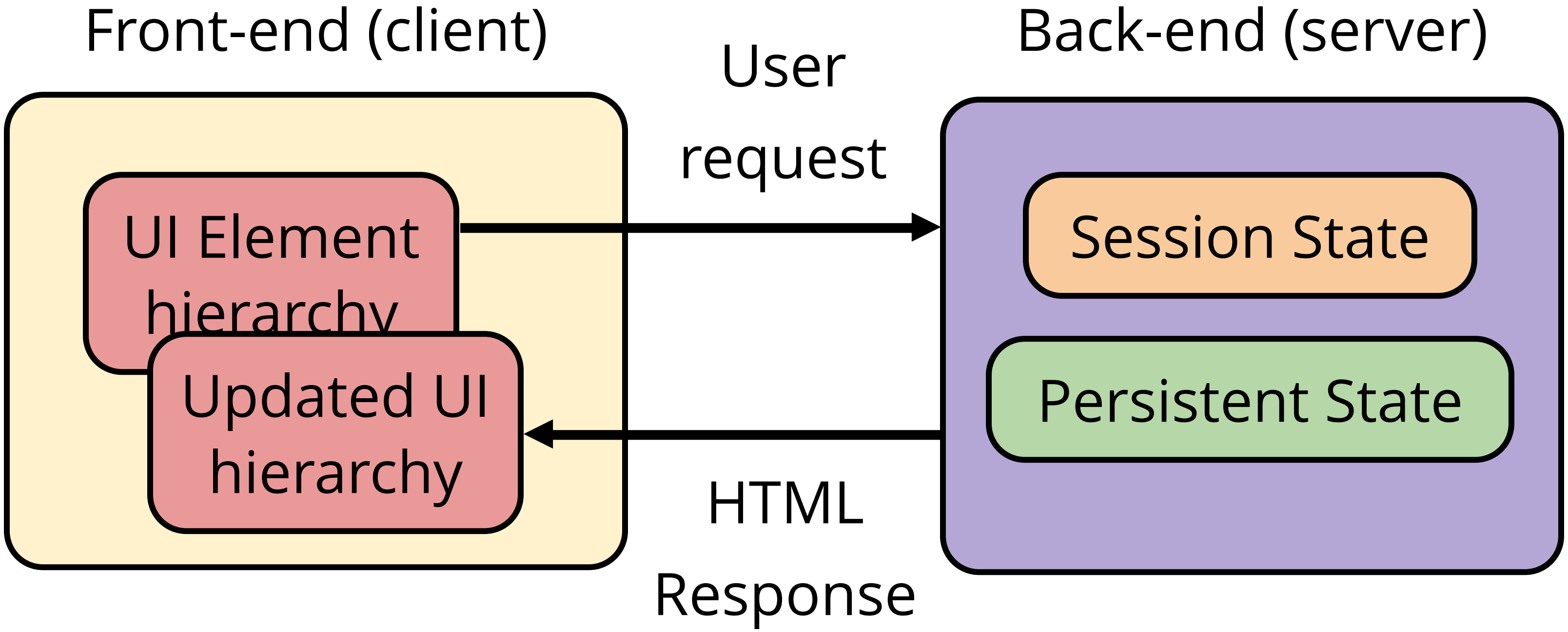
Server-side rendering



Server-side rendering

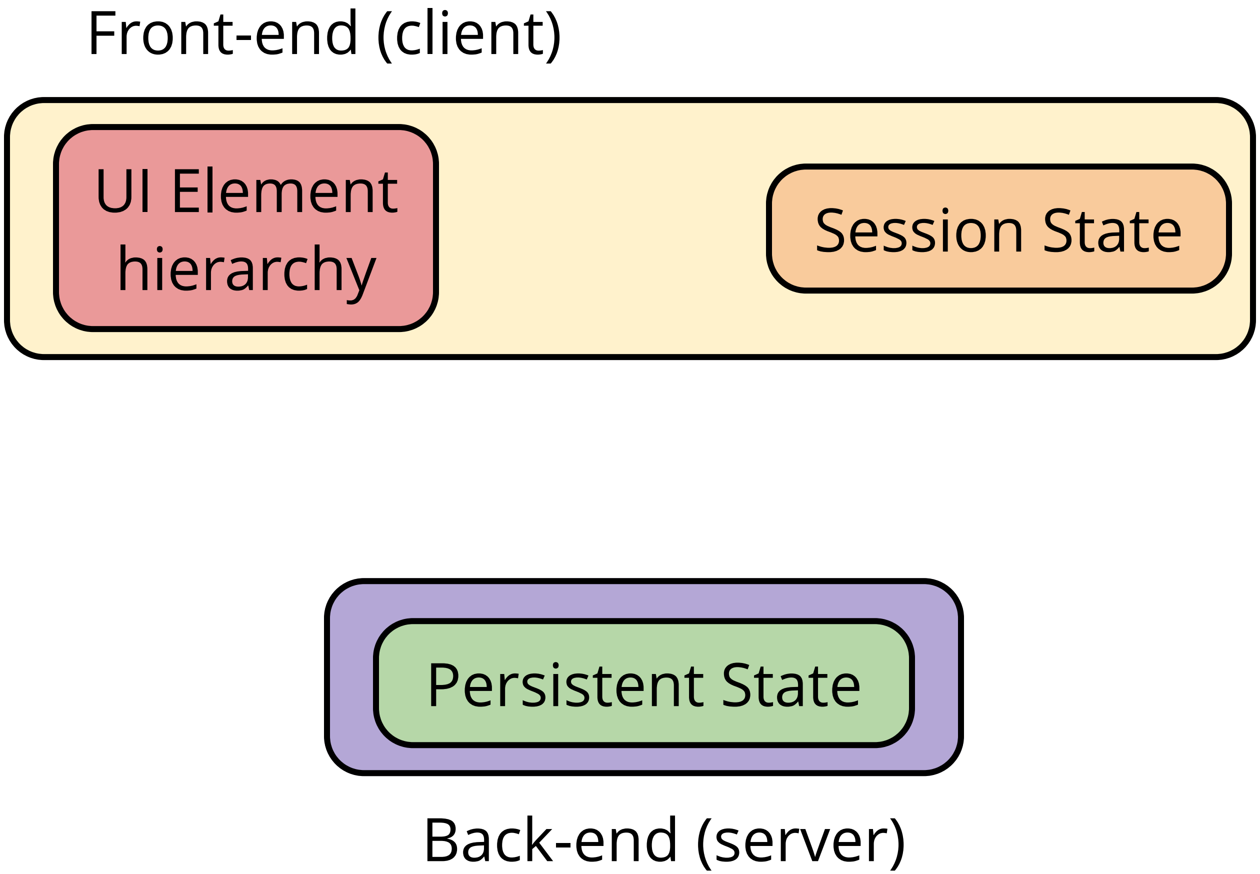


Server-side rendering

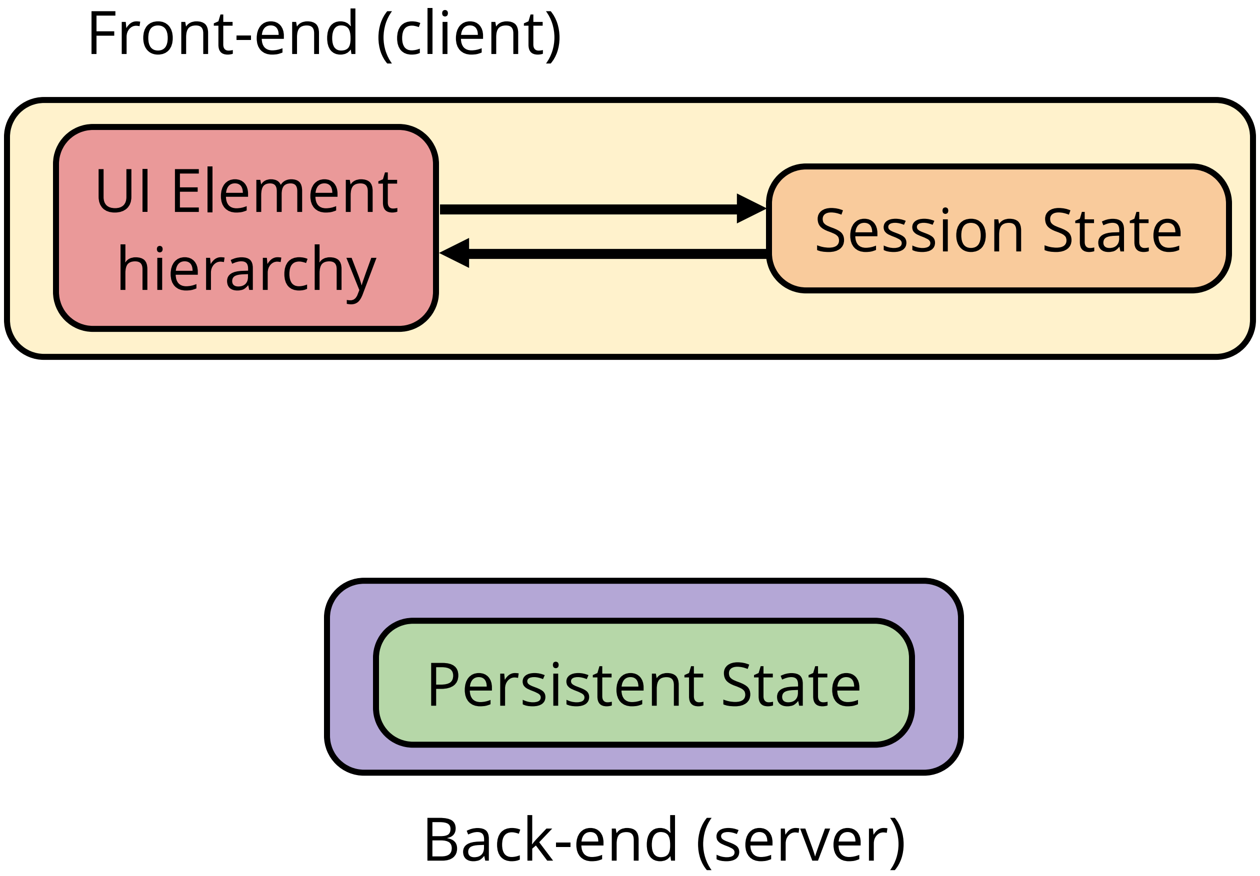


Element state is still managed by the client and can be non-trivial, so the client is not entirely state-less. But for the most part, client is not meaningfully changing the UI hierarchy.

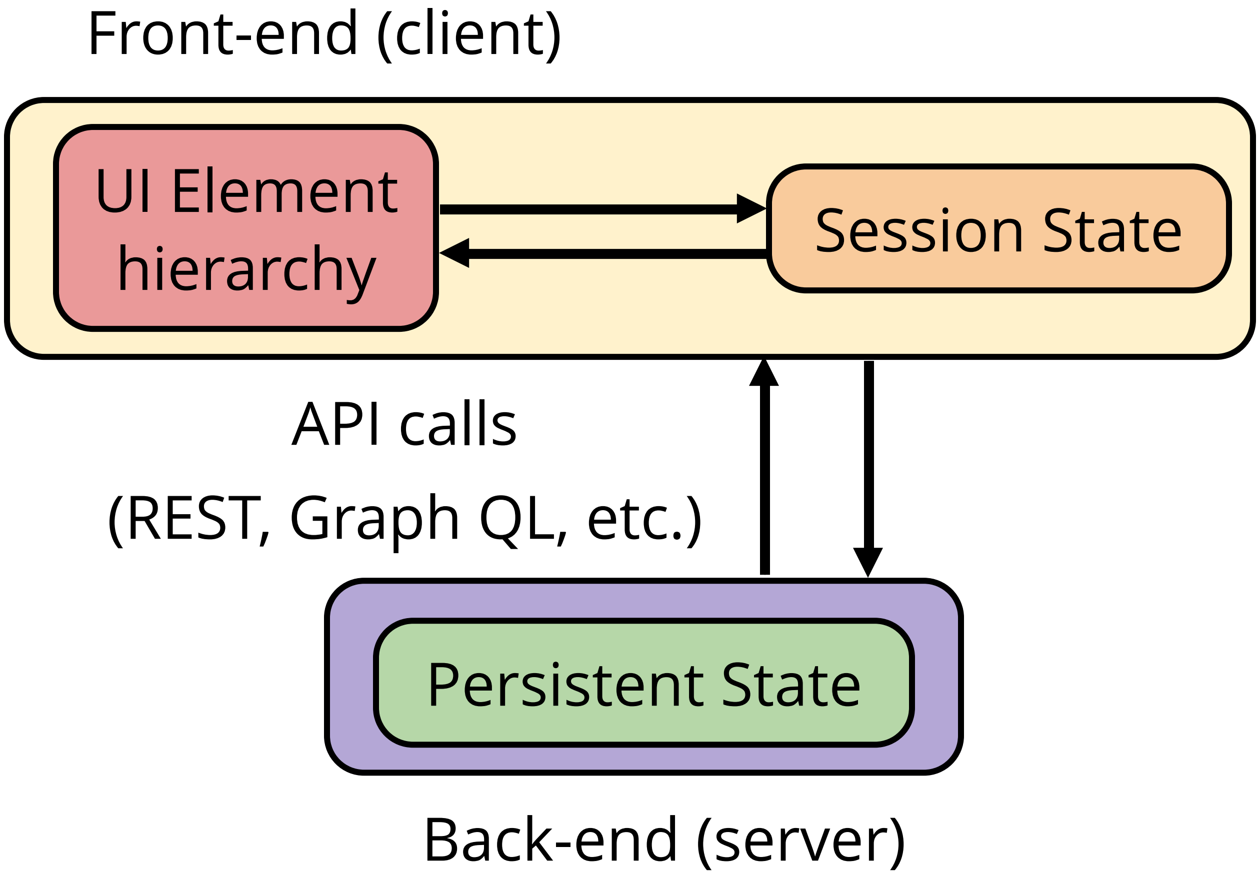
Client-side rendering



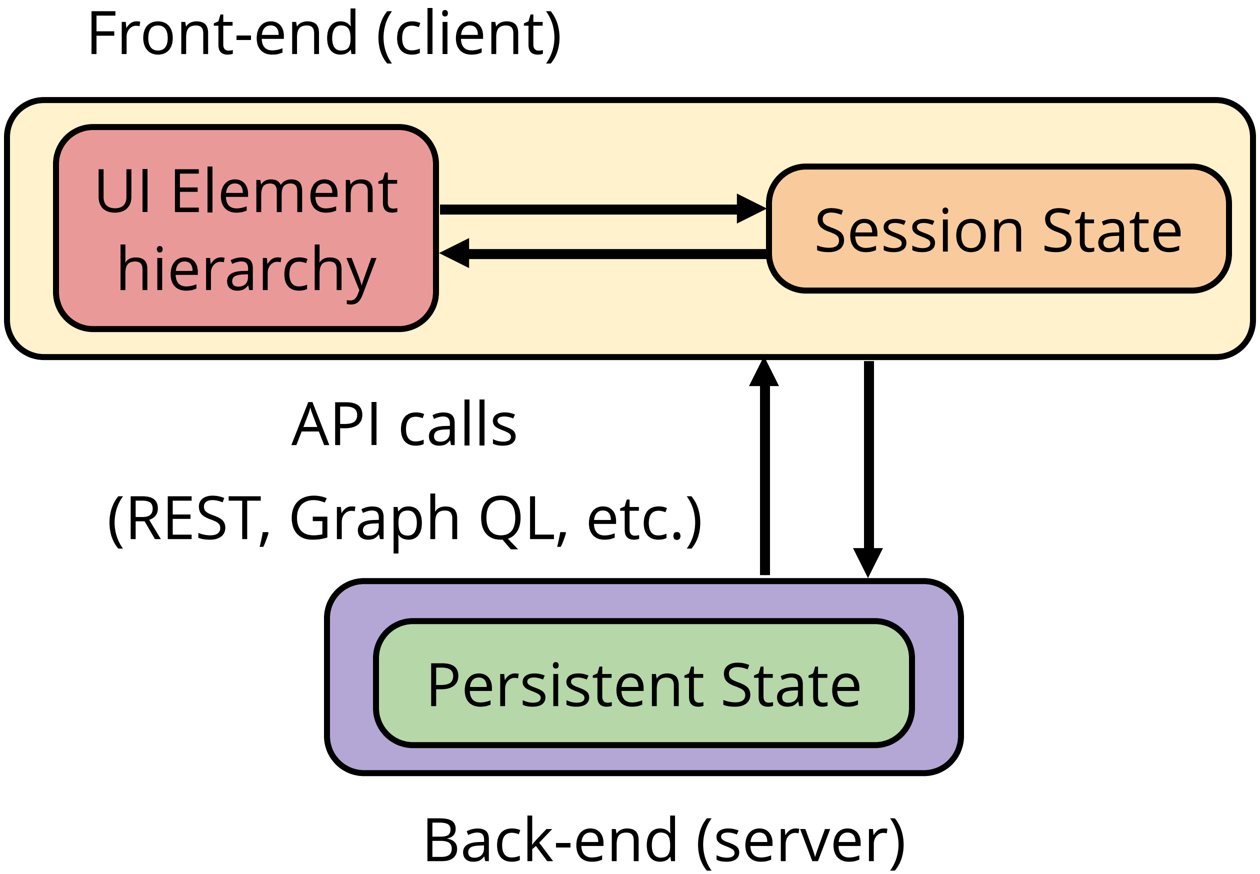
Client-side rendering



Client-side rendering



Client-side rendering



Client is responsible for updating the UI hierarchy based on both the local state and the data received from the server. Server's only responsibility is serving and storing data. (Less common, but client can also store some of the persistent state)

Which is "better"?

Which is "better"?

- Any sufficiently large project will usually mix both, at least to some extent.

Which is "better"?

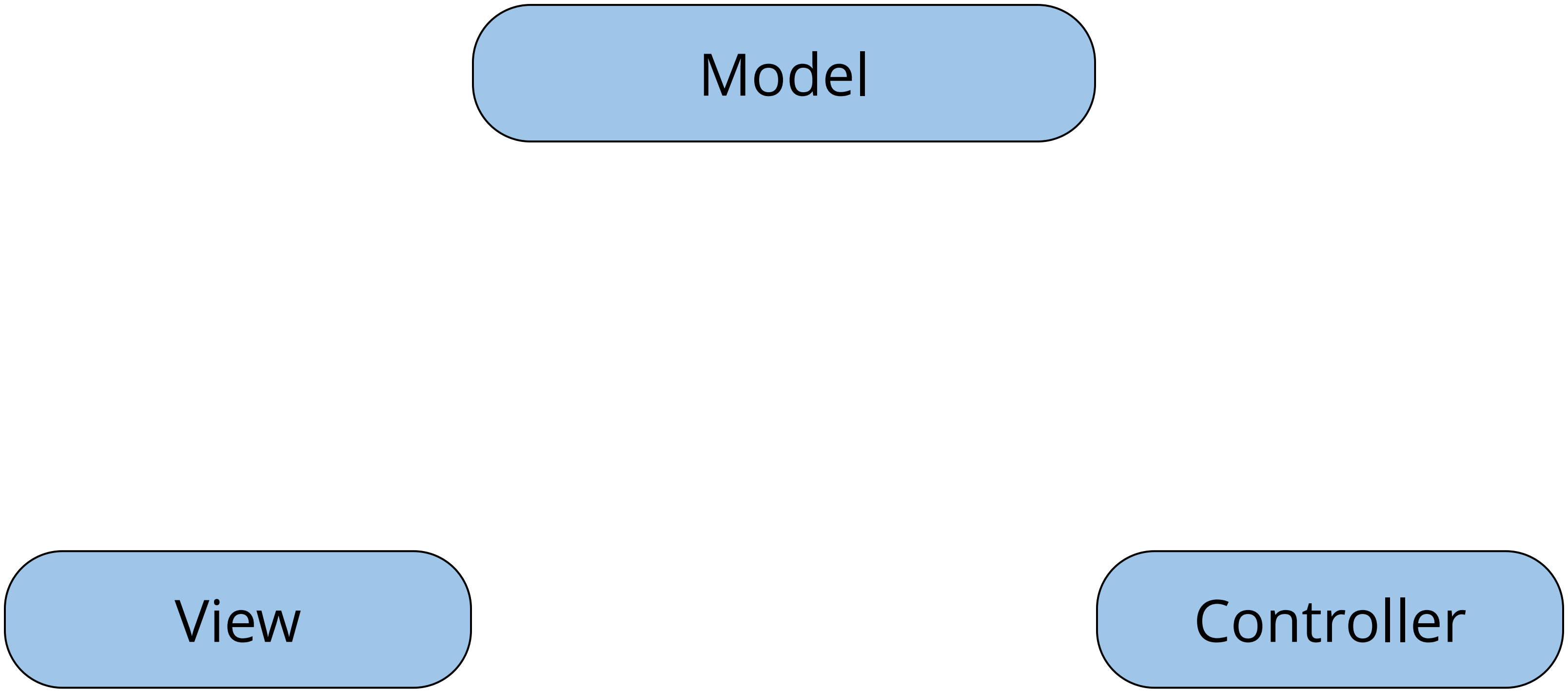
- Any sufficiently large project will usually mix both, at least to some extent.
- Server first: you have a lot of state that needs to be precisely synchronized across many users, the UI does not need too much interactivity (request-response)...
  - More centralized: the response is always what's in the database.
  - Less work for clients, no need for business logic written in JavaScript.
  - More bandwidth (sometimes, depends), more compute.

Which is "better"?

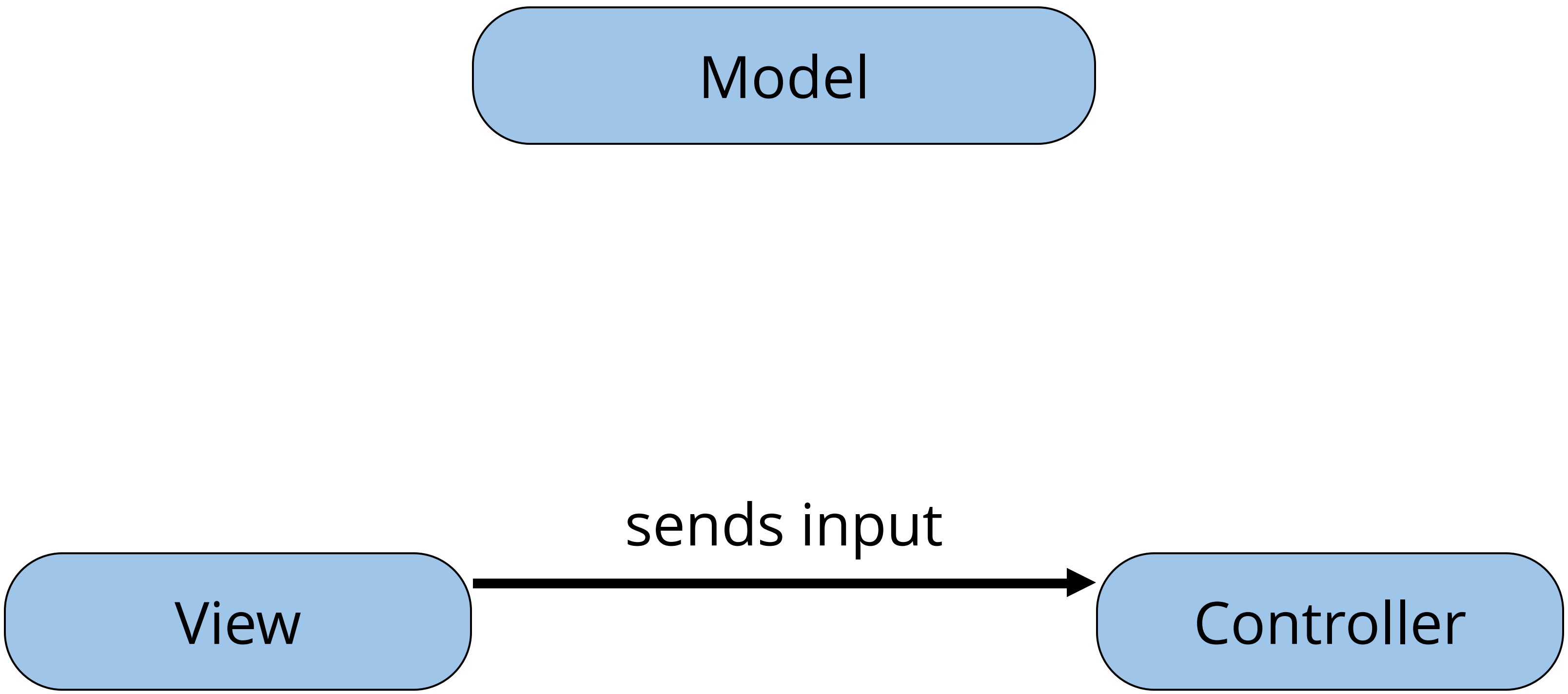
- Any sufficiently large project will usually mix both, at least to some extent.
- Server first: you have a lot of state that needs to be precisely synchronized across many users, the UI does not need too much interactivity (request-response)...
  - More centralized: the response is always what's in the database.
  - Less work for clients, no need for business logic written in JavaScript.
  - More bandwidth (sometimes, depends), more compute.
- Client first: More complex and responsive interaction patterns, reduced server costs, modular (multiple clients, one API)...
  - Can have offline or other "native" features.
  - More logic (generally more complex UIs, but also logic for retrieving and sending data, caching, etc.).

Part 5: Managing state

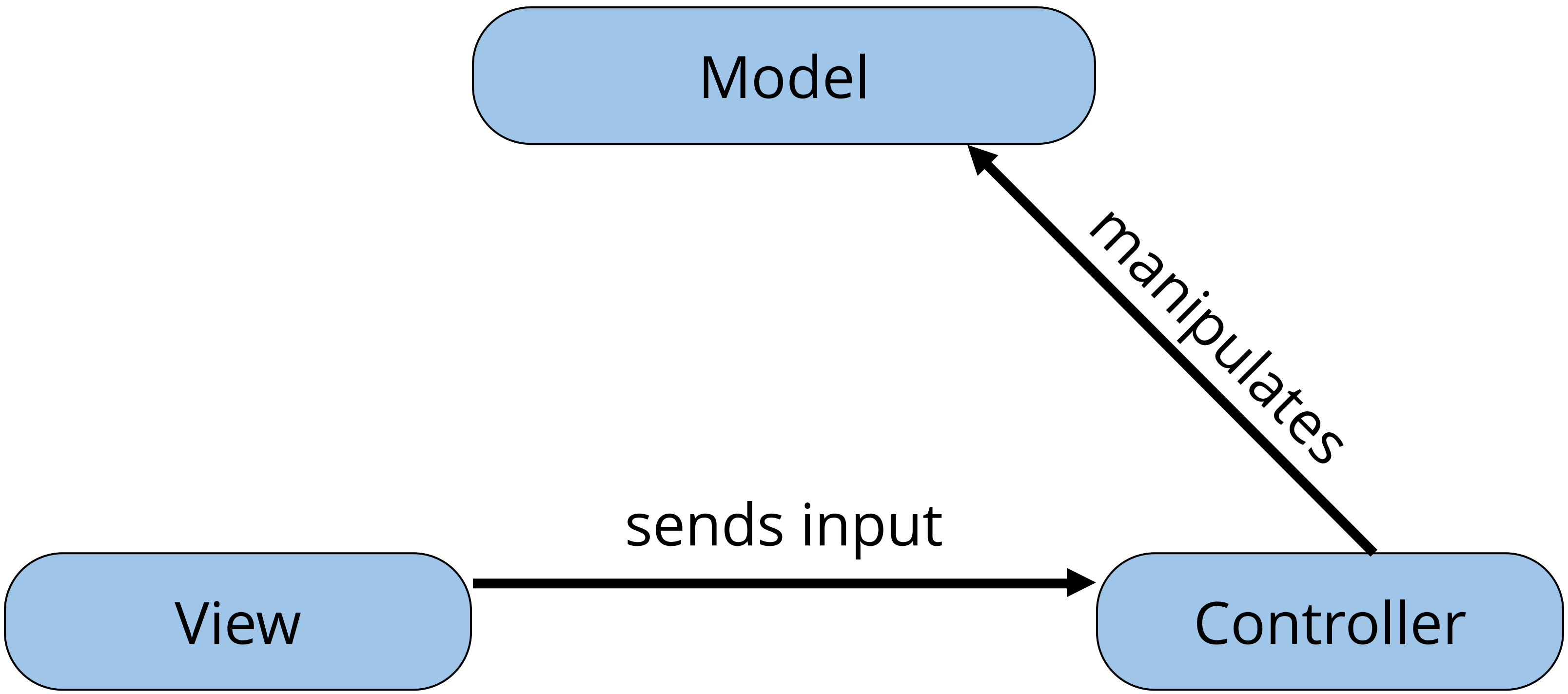
Model View Controller



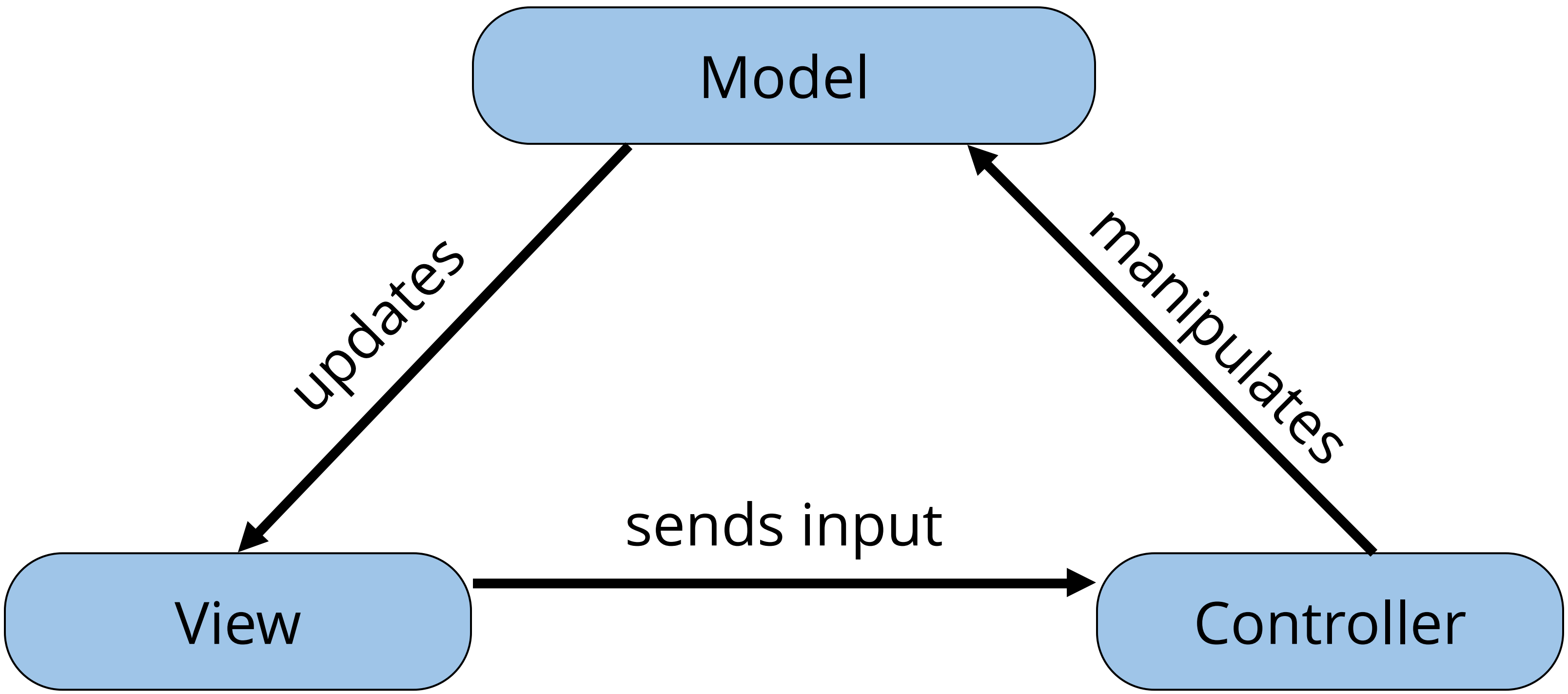
Model View Controller



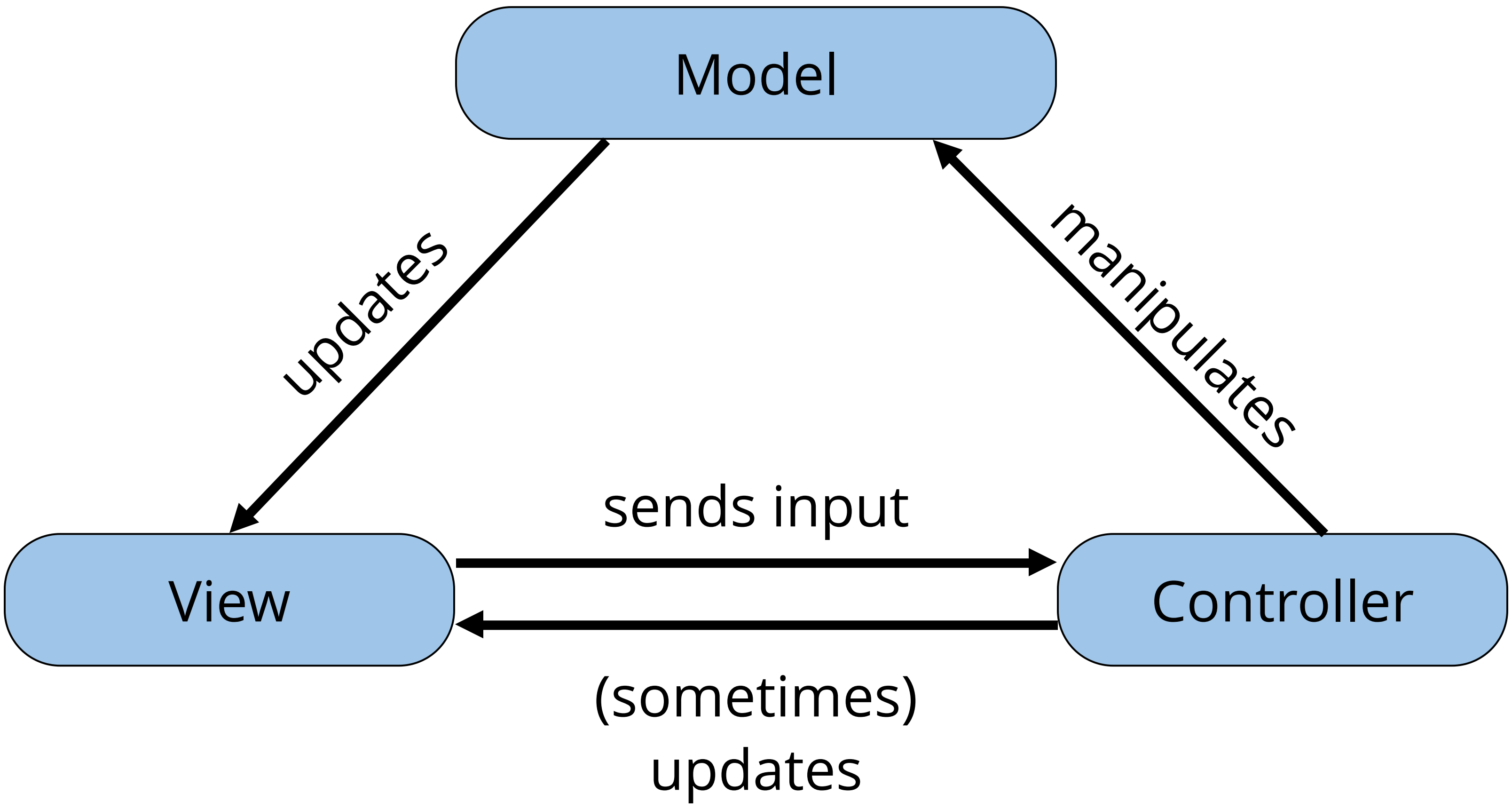
Model View Controller



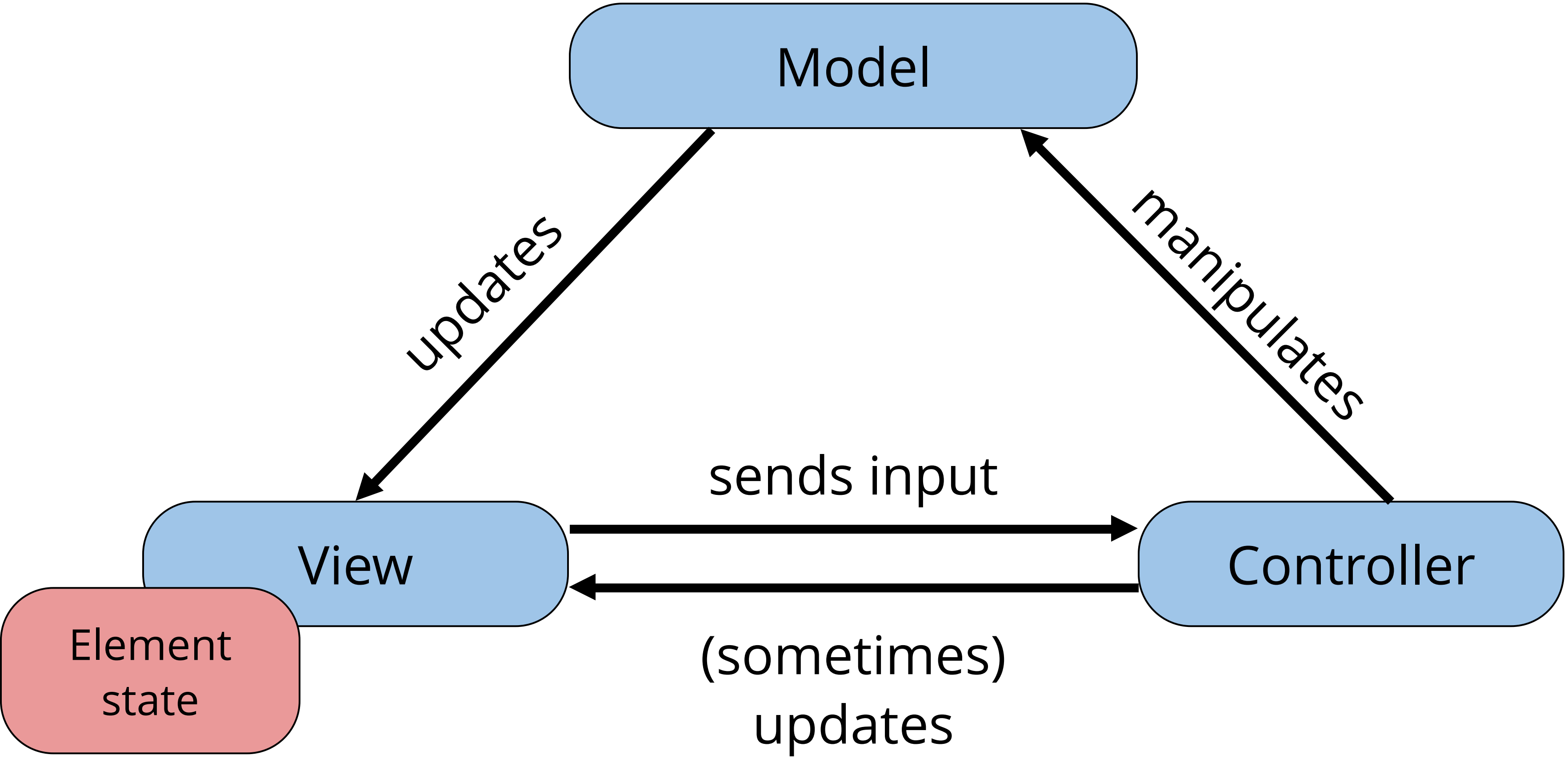
Model View Controller



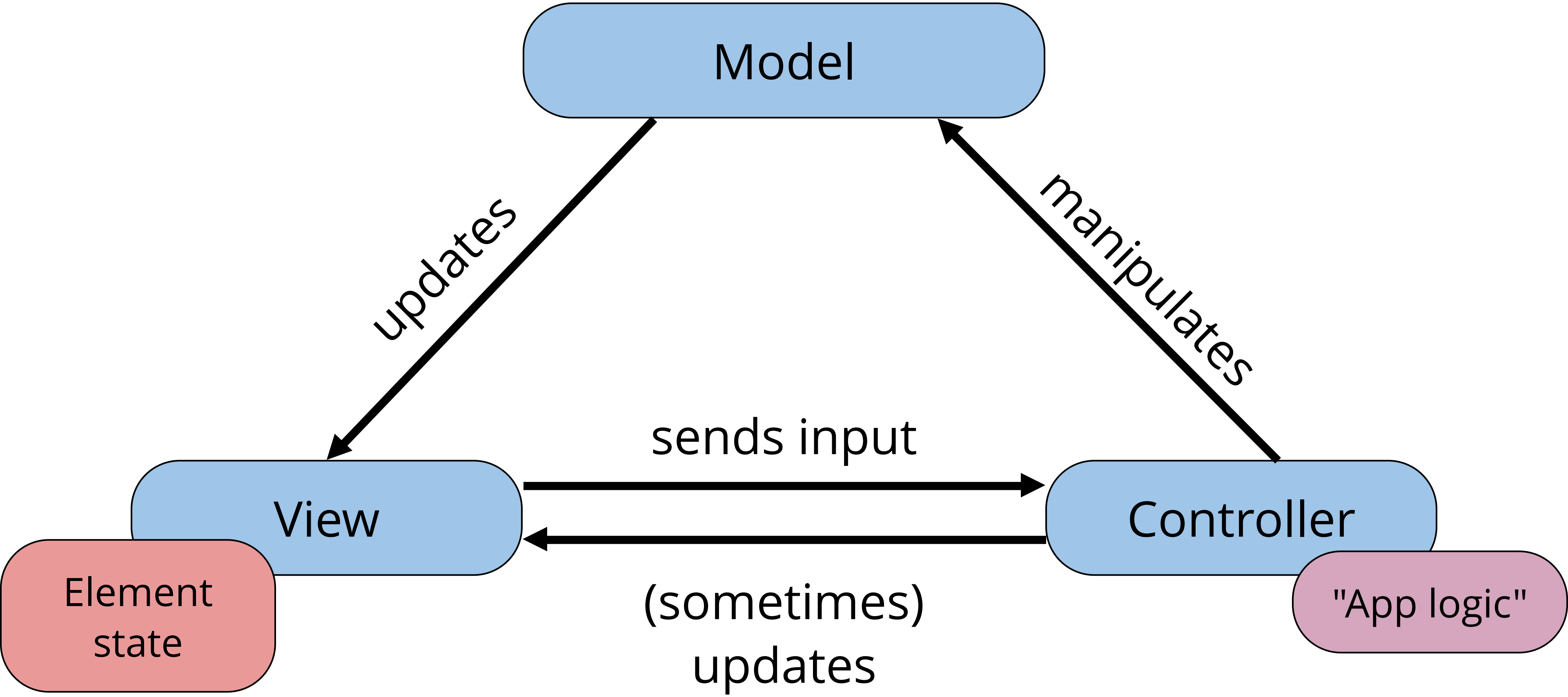
Model View Controller

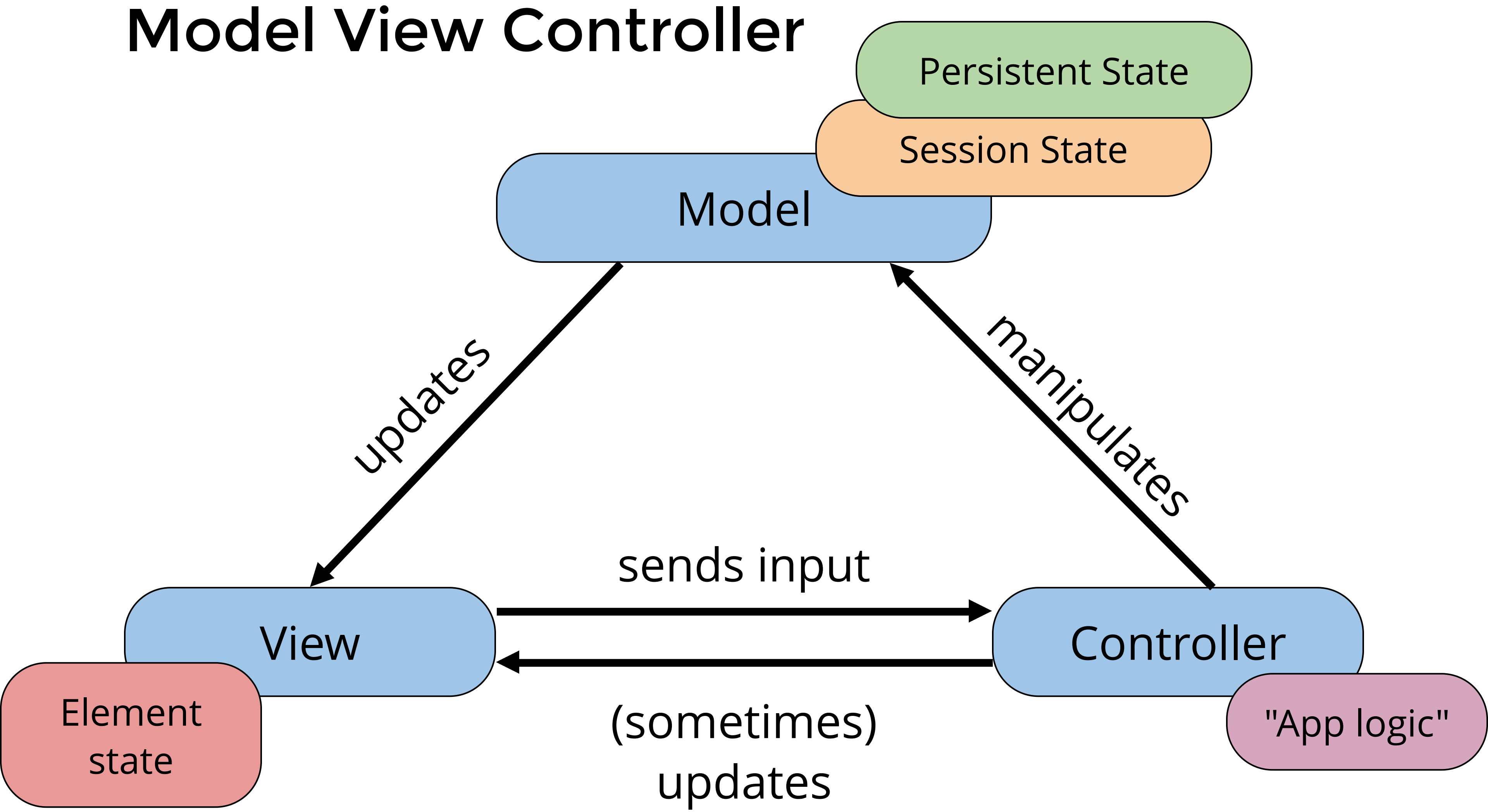


Model View Controller



Model View Controller





Model View Controller
Persistent State
Session State
Model
updates
manipulates
sends input
View
Controller
Element state
(sometimes) updates
"App logic"

Model View Controller

Model View Controller

- Multiple types of each component can interact together (e.g. one view per "entity" in a database, one controller per business process, and one view per page).

Model View Controller

- Multiple types of each component can interact together (e.g. one view per "entity" in a database, one controller per business process, and one view per page).
- Views can be often relatively large (e.g. a whole page) and relatively "active" in that they generate the complete Element hierarchy for every state change.

  Example: A HTML template that is filled with data.

Model View Controller

- Multiple types of each component can interact together (e.g. one view per "entity" in a database, one controller per business process, and one view per page).
- Views can be often relatively large (e.g. a whole page) and relatively "active" in that they generate the complete Element hierarchy for every state change.
  Example: A HTML template that is filled with data.
- Model is relatively "passive": it ensures data validation, integrity and persistence.

Model View Controller

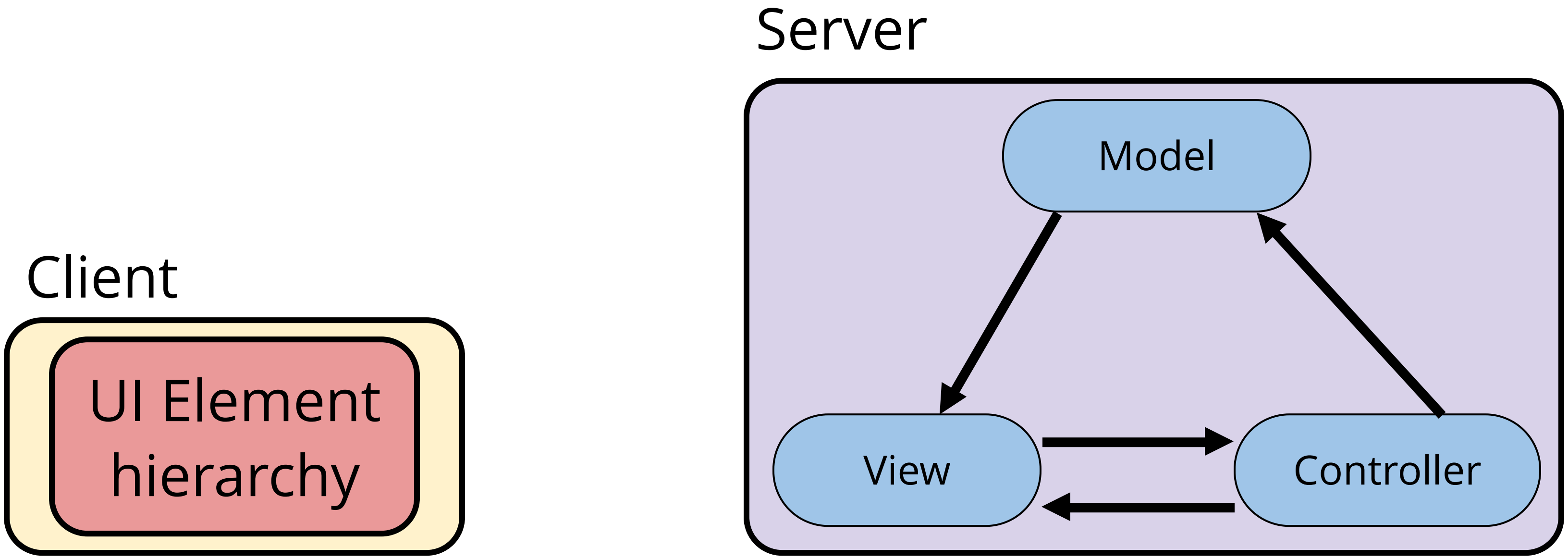
- Multiple types of each component can interact together (e.g. one view per "entity" in a database, one controller per business process, and one view per page).
- Views can be often relatively large (e.g. a whole page) and relatively "active" in that they generate the complete Element hierarchy for every state change.
  Example: A HTML template that is filled with data.
- Model is relatively "passive": it ensures data validation, integrity and persistence.
- Controller performs updates based on UI events.

Model View Controller

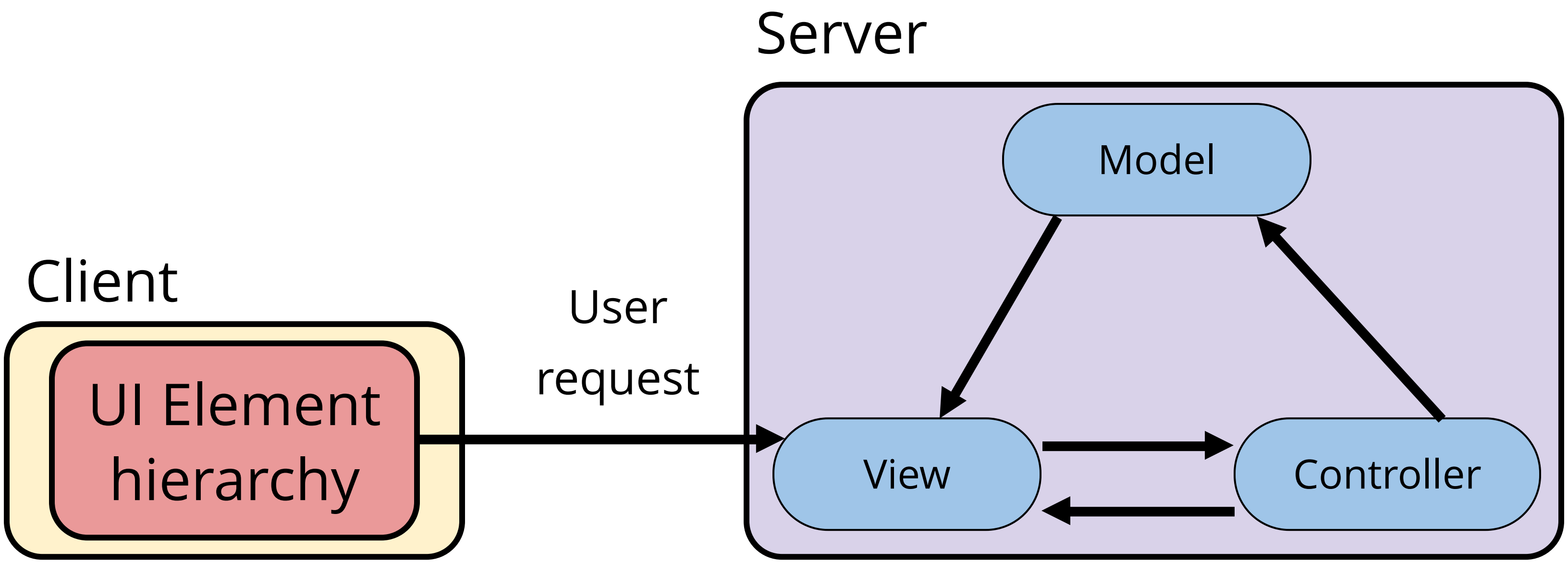
- Multiple types of each component can interact together (e.g. one view per "entity" in a database, one controller per business process, and one view per page).
- Views can be often relatively large (e.g. a whole page) and relatively "active" in that they generate the complete Element hierarchy for every state change.
  Example: A HTML template that is filled with data.
- Model is relatively "passive": it ensures data validation, integrity and persistence.
- Controller performs updates based on UI events.

But where does this "live" in a Web app?

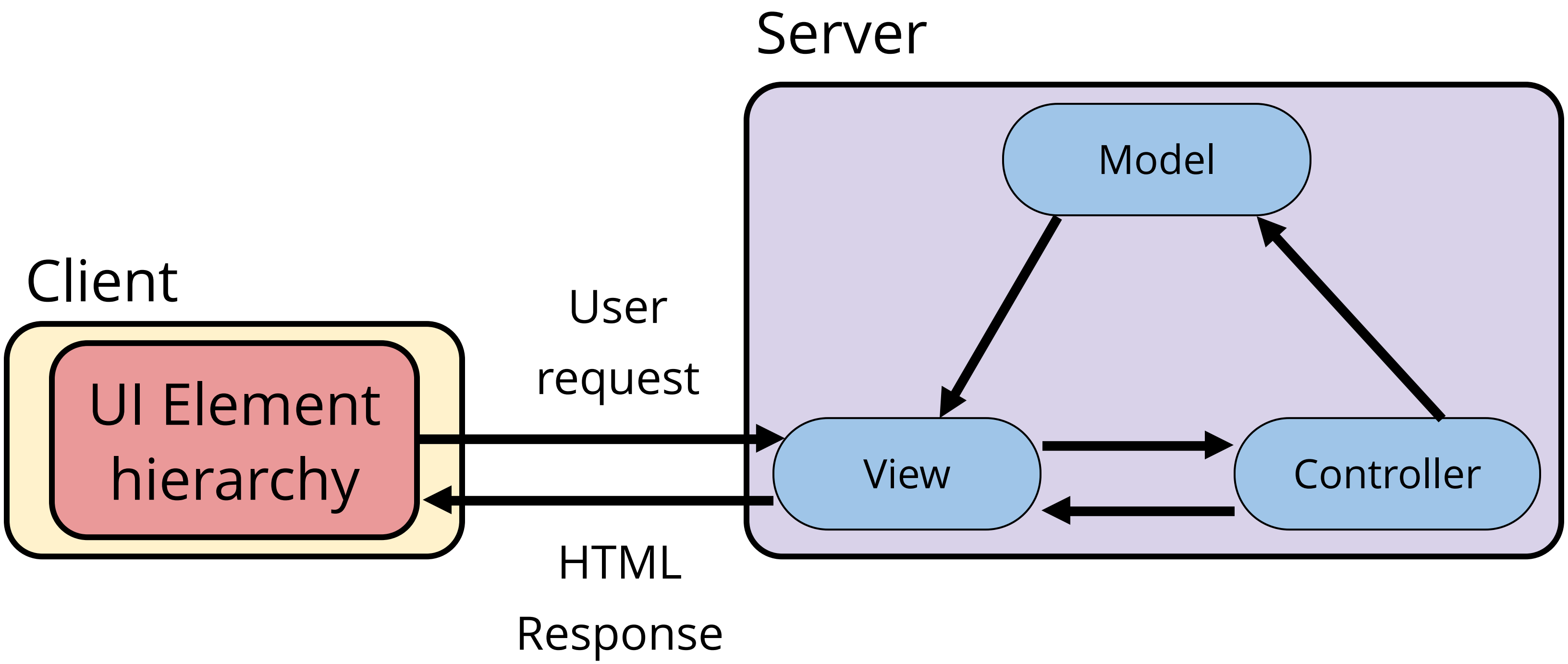
MVC, server-side rendering



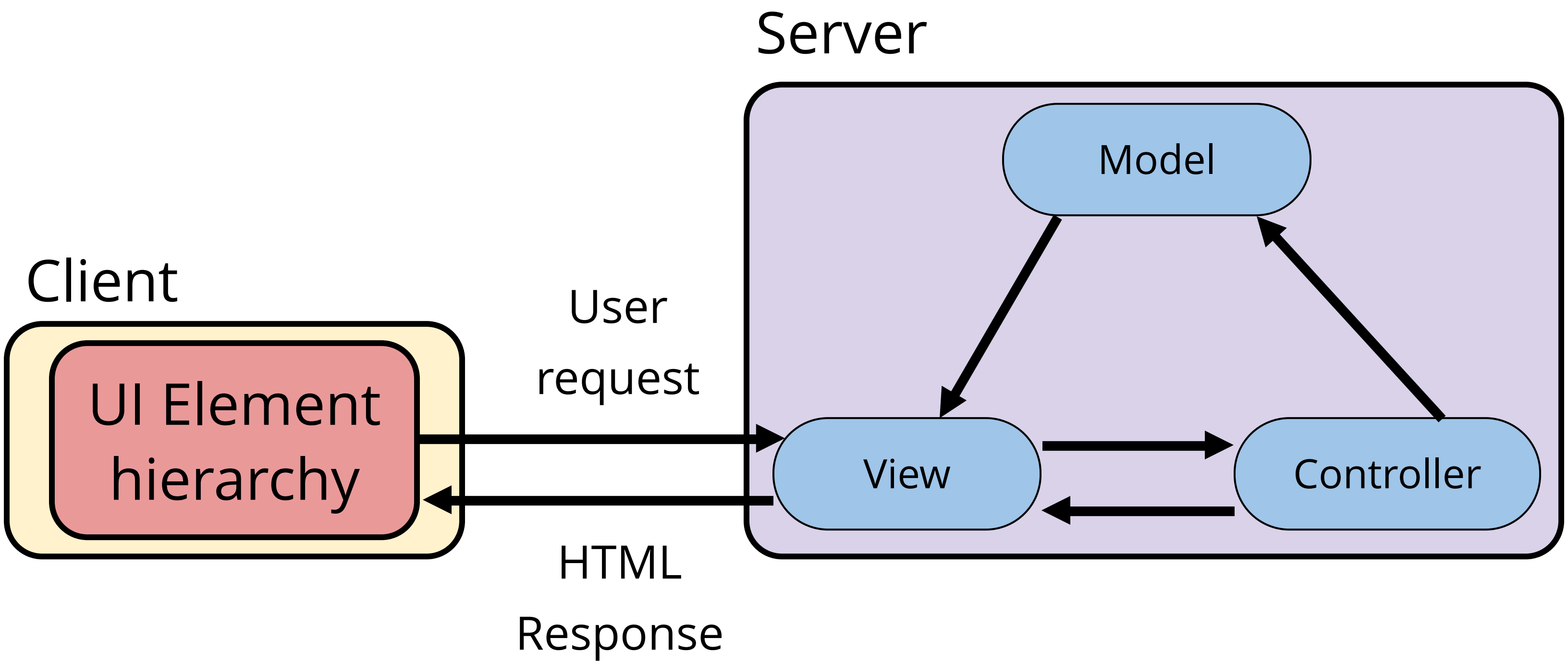
MVC, server-side rendering



MVC, server-side rendering

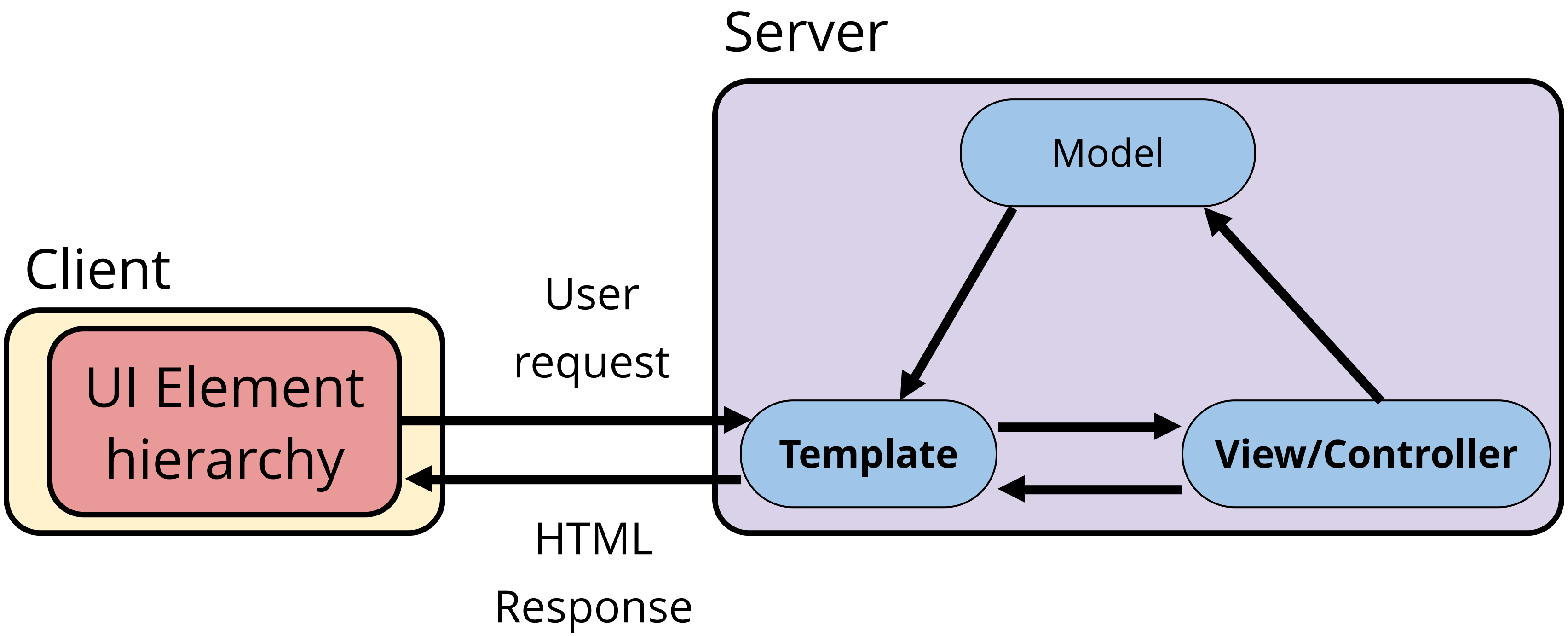


MVC, server-side rendering



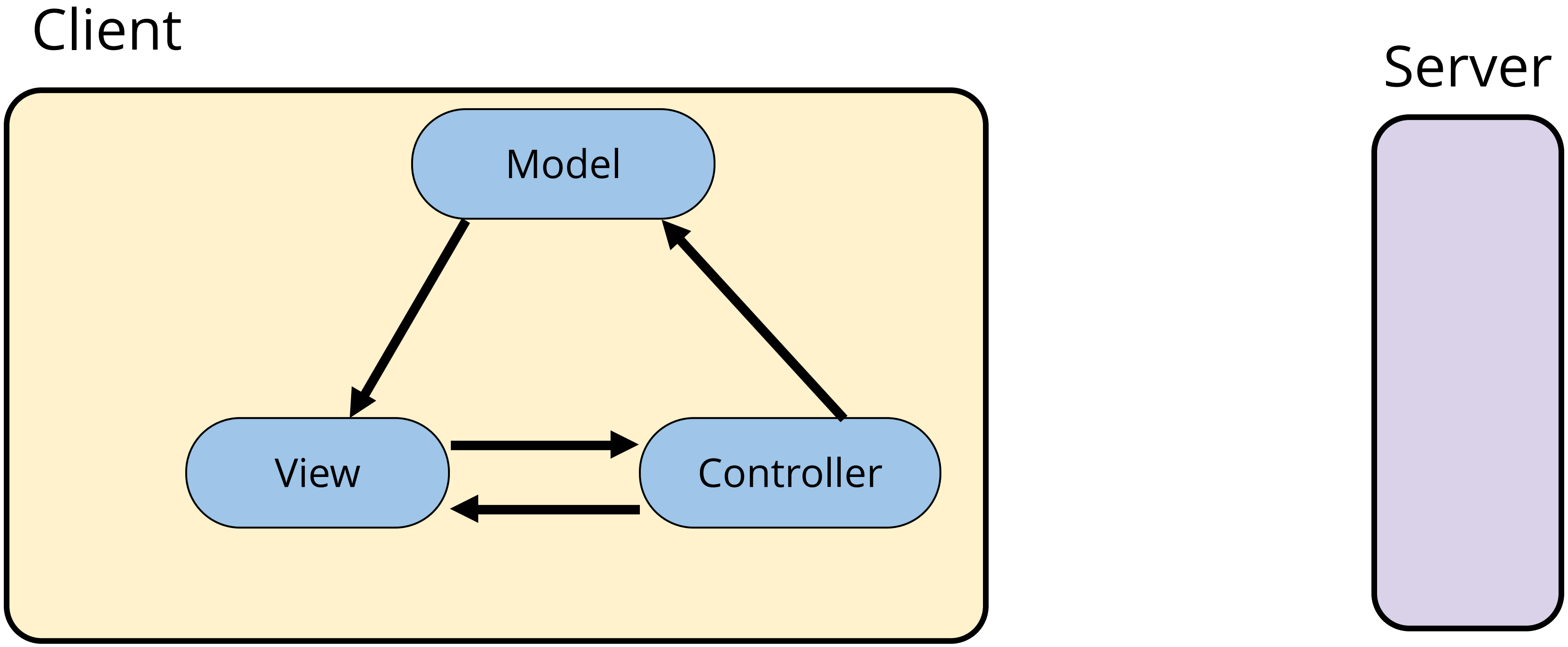
Server-first frameworks like ASP.NET (C#), Spring (Java), Django (Python), Rails (Ruby), Laravel (PHP) are usually "MVC-like".

Model-View-Template?



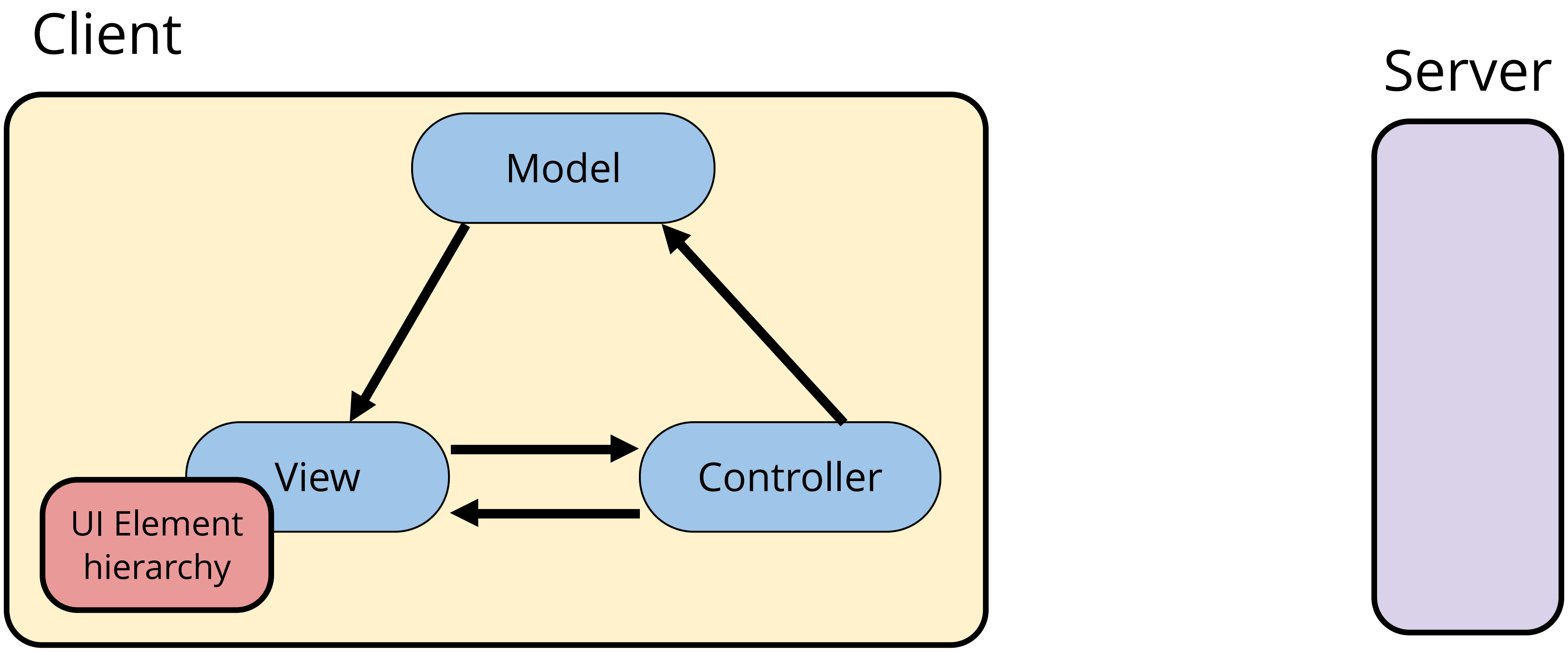
"MVC-like": Other minor interpretations of what `View` and `Controller` stand for.

MVC, client-side rendering



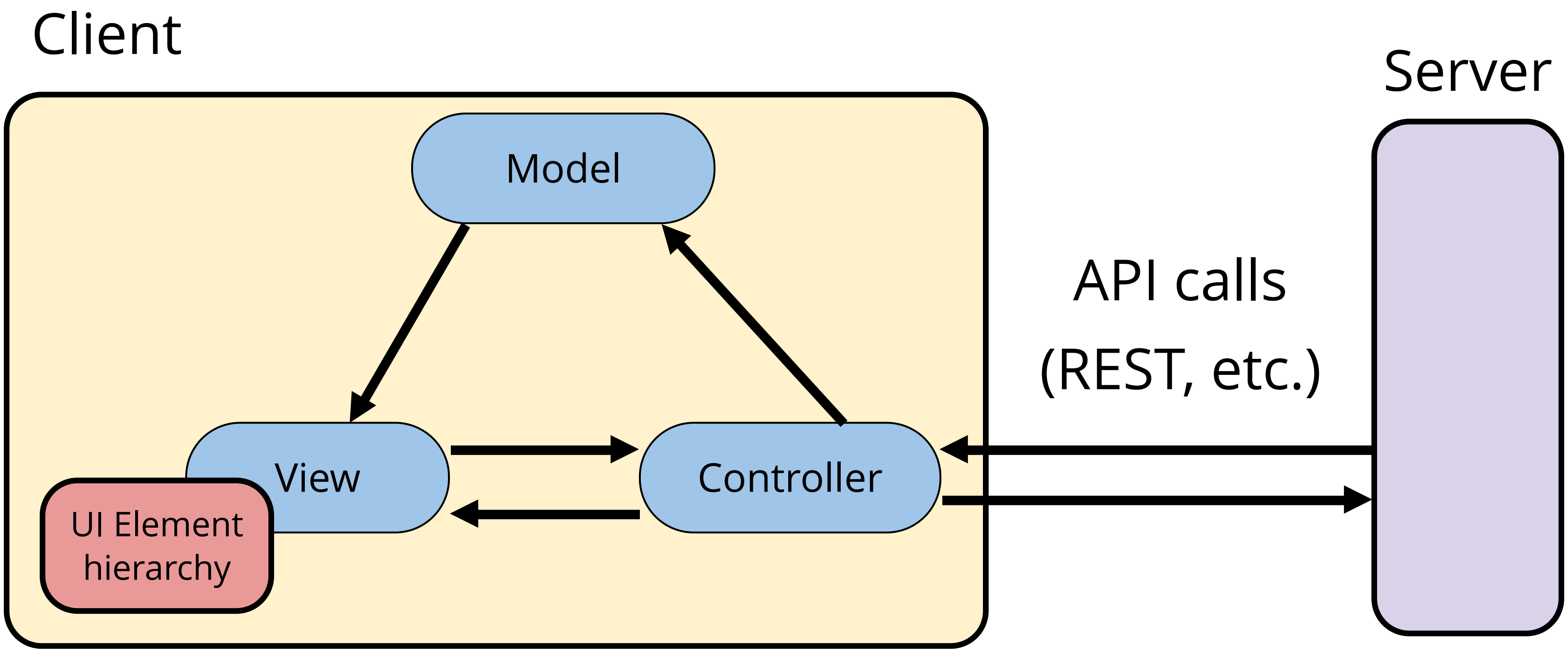
Not that common... Angluar (JavaScript; mostly older versions).

MVC, client-side rendering



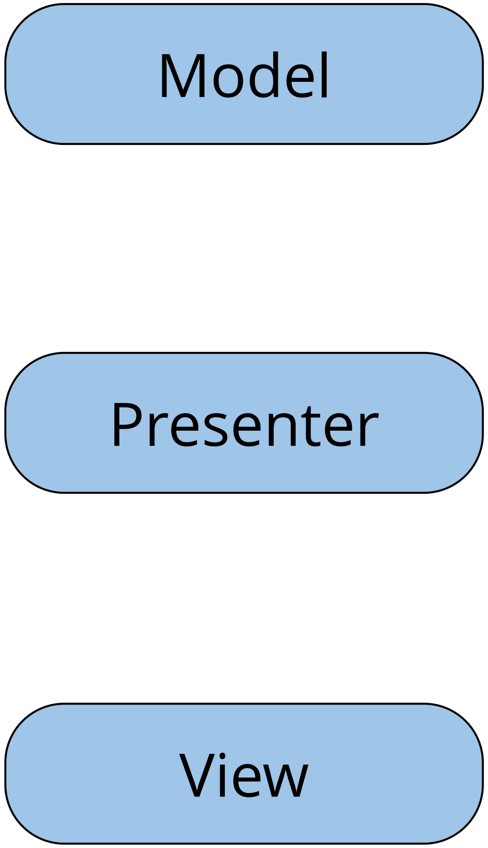
Not that common... Angluar (JavaScript; mostly older versions).

MVC, client-side rendering

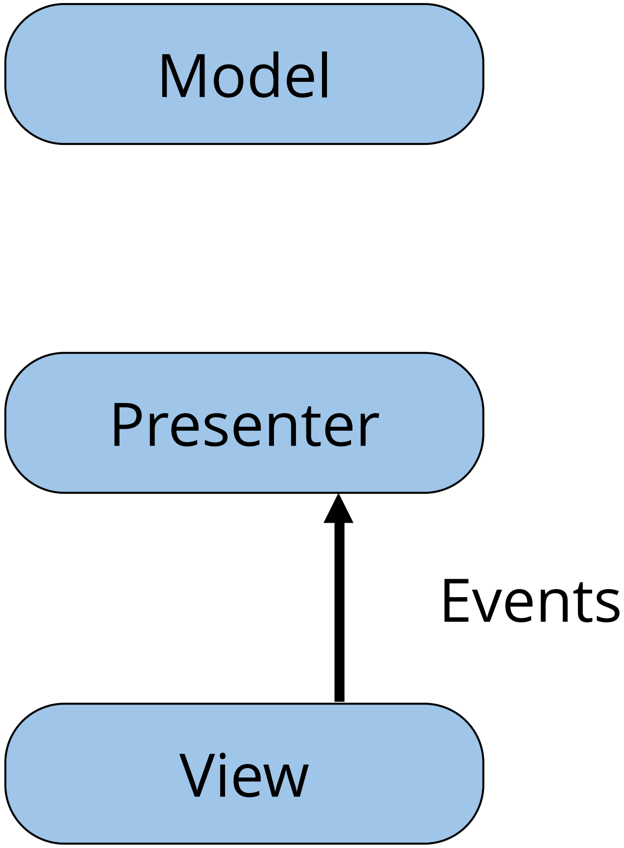


Not that common... Angluar (JavaScript; mostly older versions).

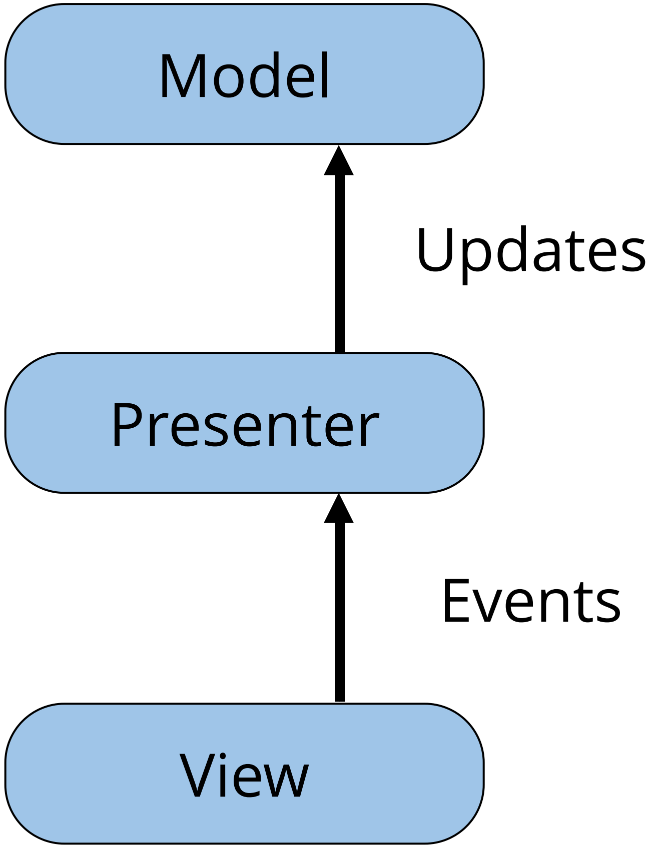
Model View Presenter



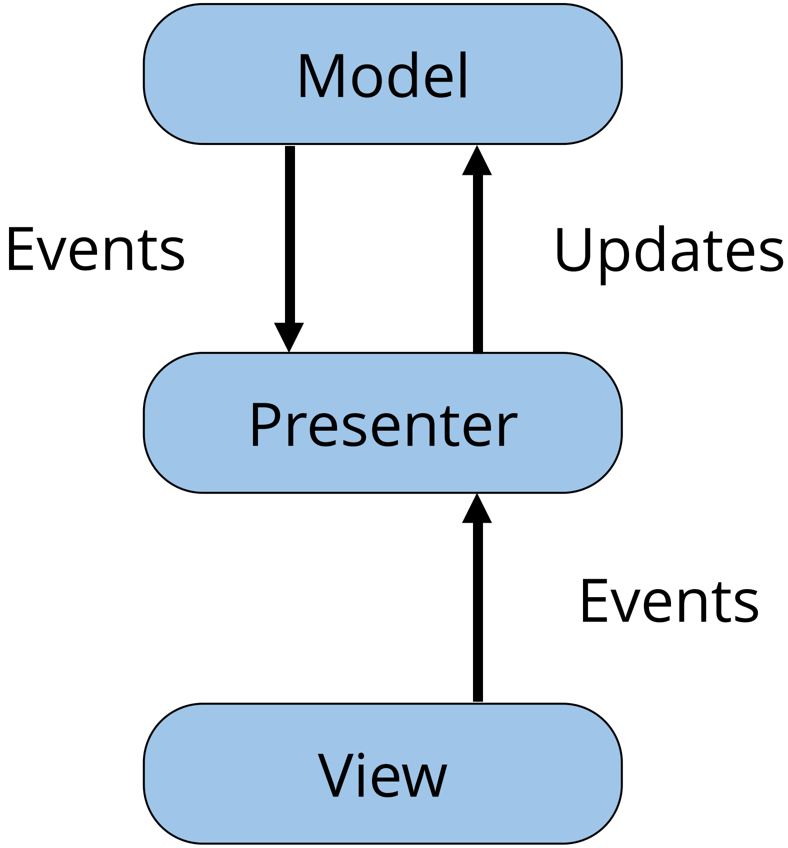
Model View Presenter



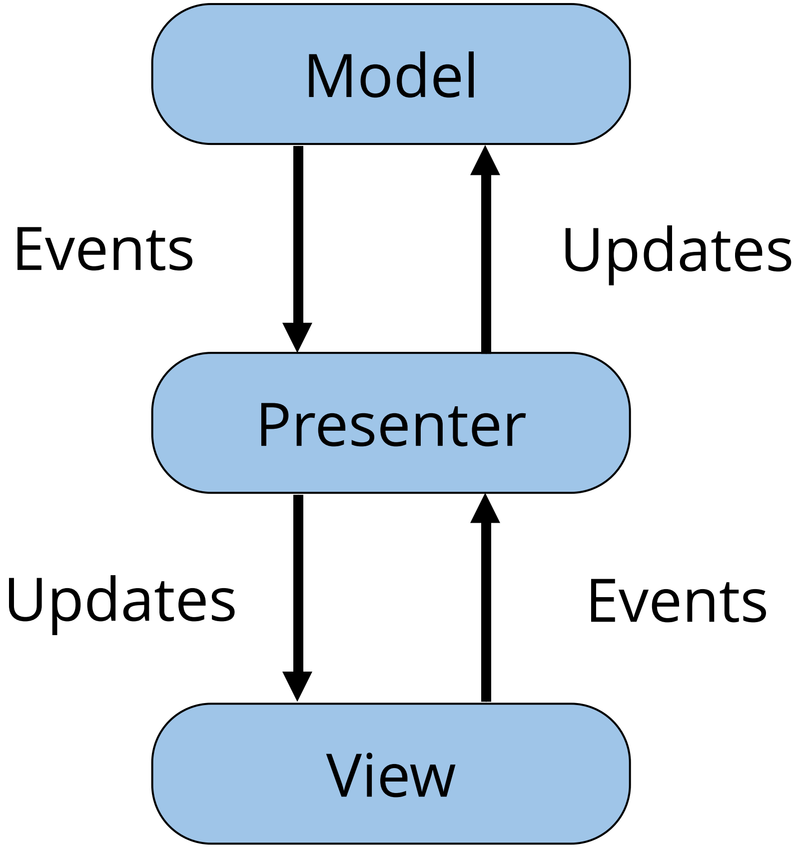
Model View Presenter



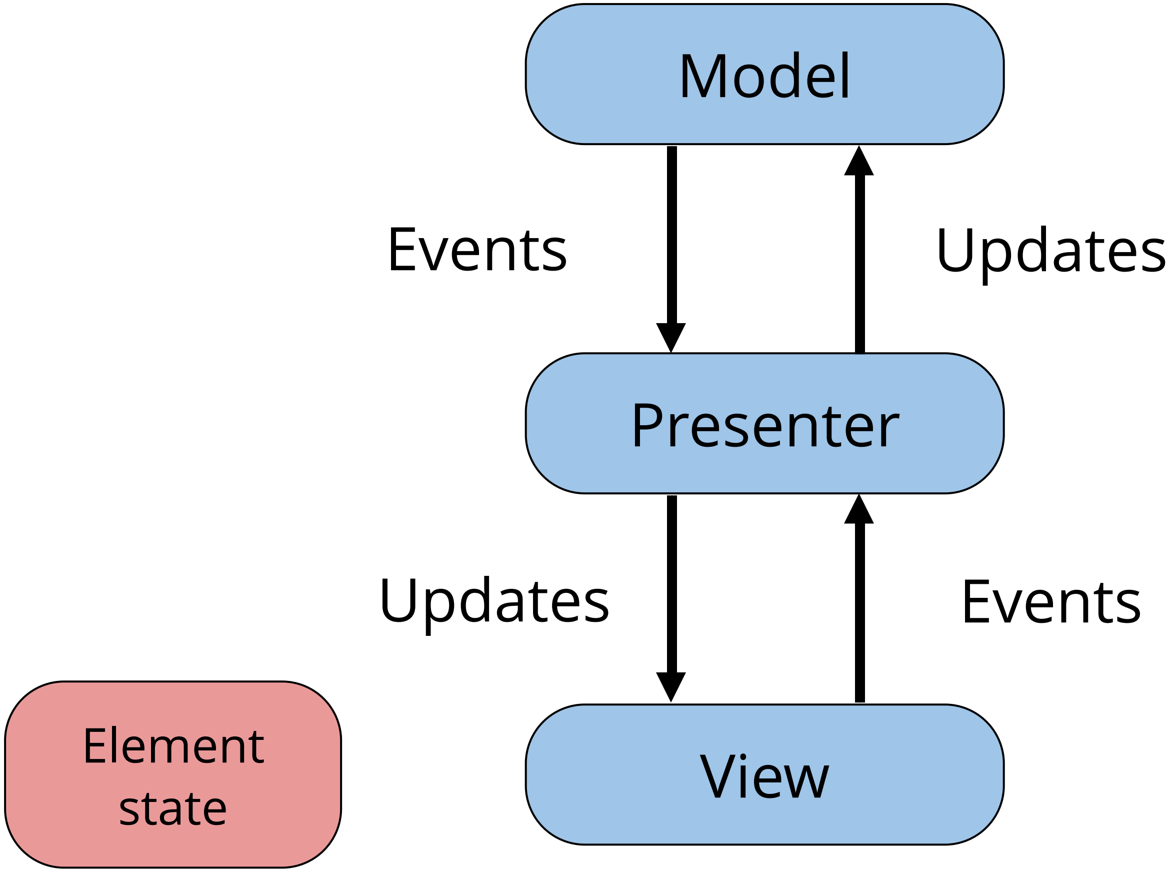
Model View Presenter



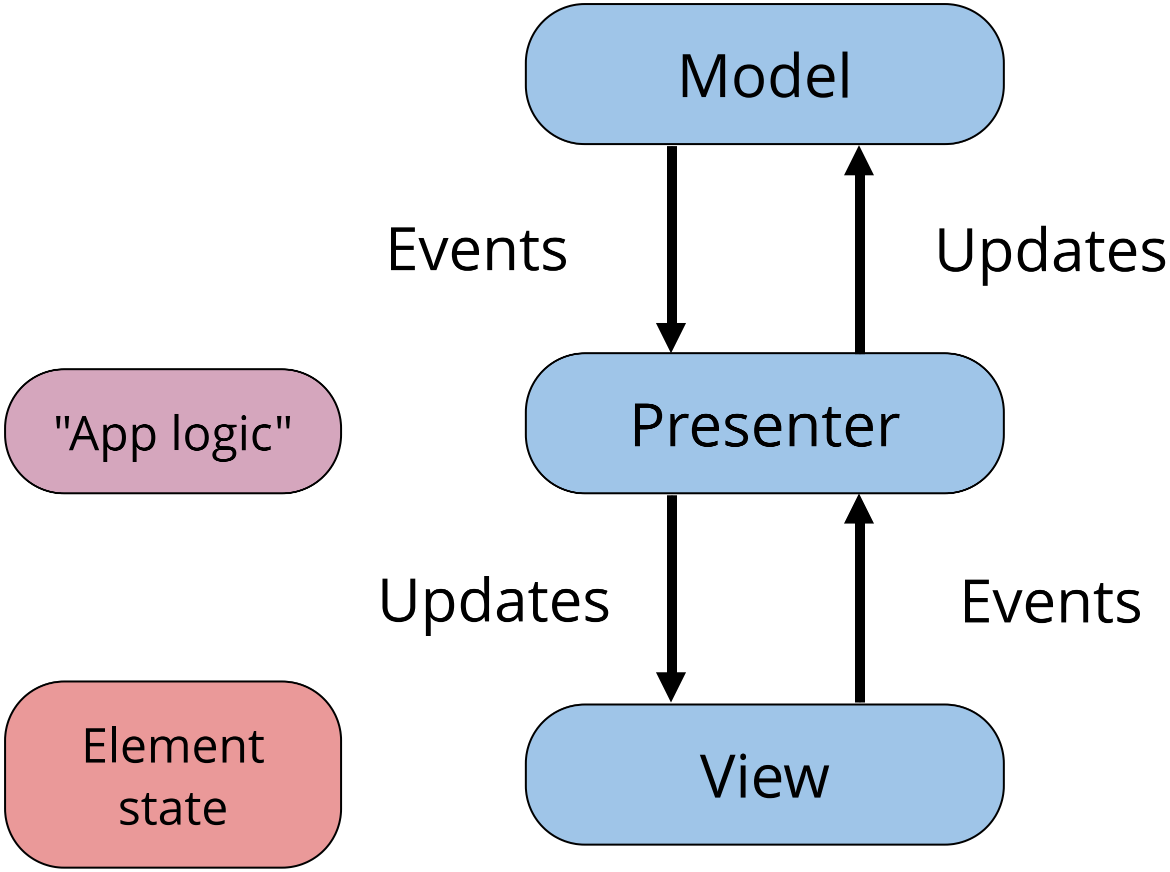
Model View Presenter



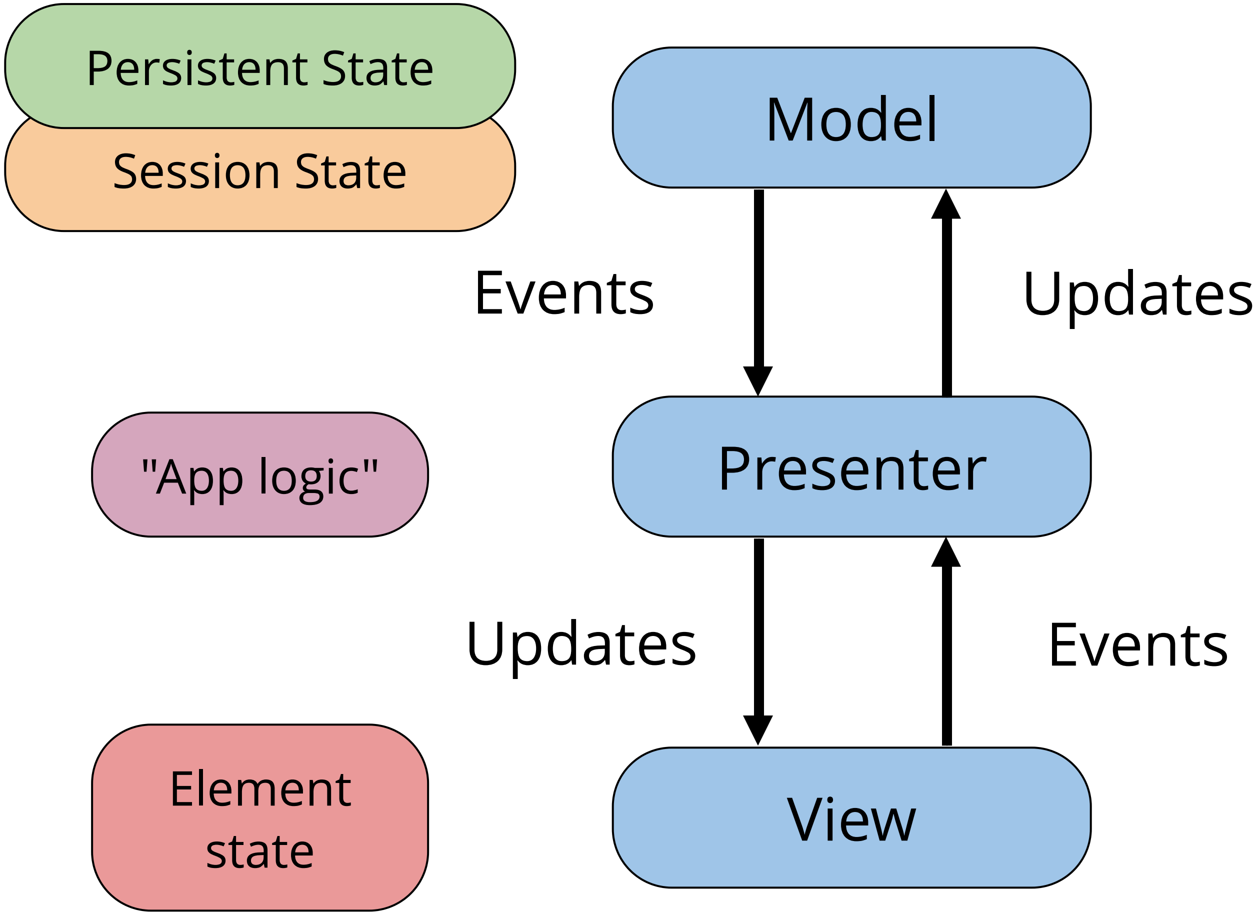
Model View Presenter



Model View Presenter



Model View Presenter



Model View Presenter

Model View Presenter

- Removes the ambiguity of who updates the `View`.

Model View Presenter

- Removes the ambiguity of who updates the `View`.
- The `View` is completely passive, all updates are done by the `Presenter`.

Model View Presenter

- Removes the ambiguity of who updates the `View`.
- The `View` is completely passive, all updates are done by the `Presenter`.
- `Presenters` tend to be more tightly integrated with the `View`, while `Controllers` tend to be more tied to the `Model`. Often leads to smaller/more granular `Presenters` in MVP vs. MVC.

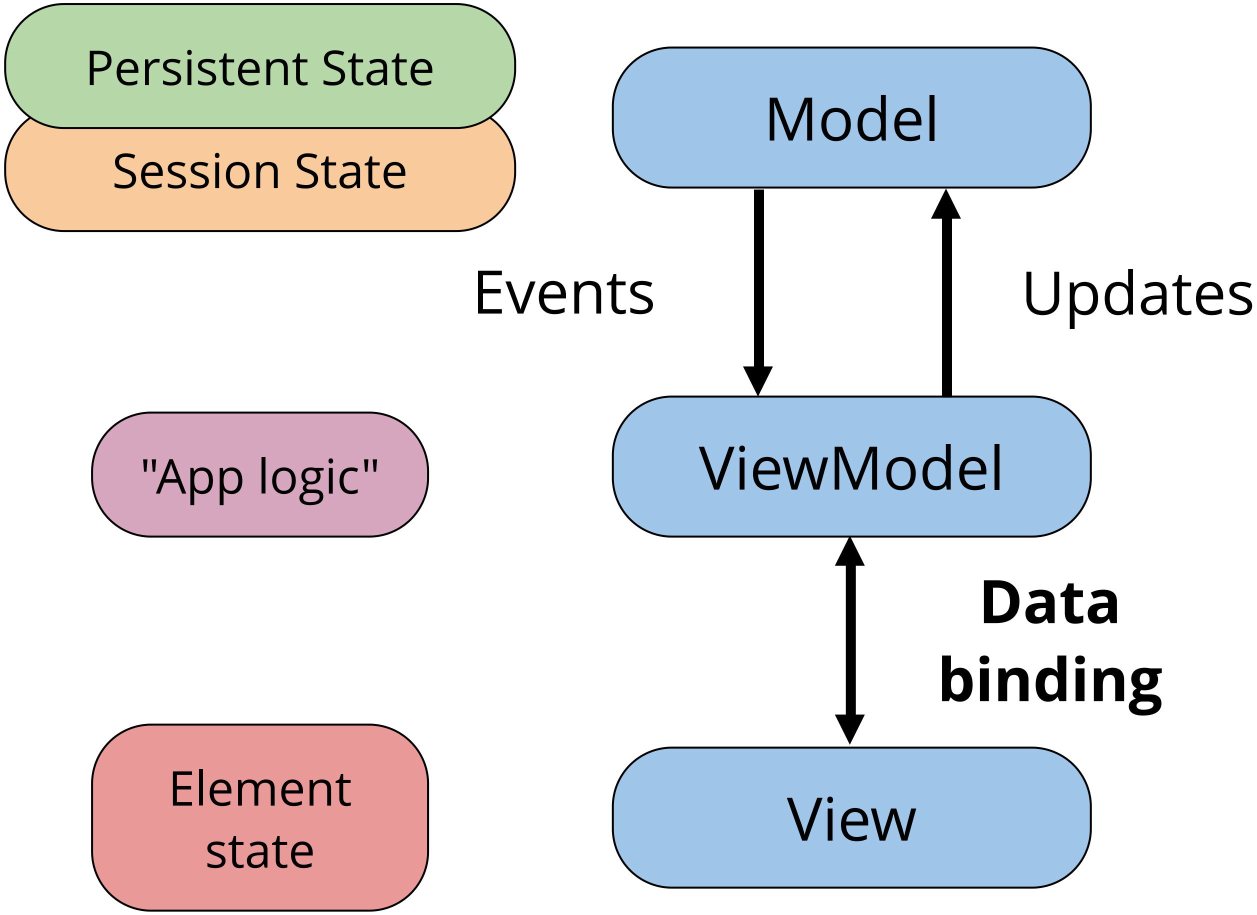
Model View Presenter

- Removes the ambiguity of who updates the `View`.
- The `View` is completely passive, all updates are done by the `Presenter`.
- `Presenters` tend to be more tightly integrated with the `View`, while `Controllers` tend to be more tied to the `Model`. Often leads to smaller/more granular `Presenters` in MVP vs. MVC.
- Most frameworks mentioned previously are primarily classified as MVP now, but the distinction is often slightly arbitrary (one can implement both).

Model View Presenter

- Removes the ambiguity of who updates the `View`.
- The `View` is completely passive, all updates are done by the `Presenter`.
- `Presenters` tend to be more tightly integrated with the `View`, while `Controllers` tend to be more tied to the `Model`. Often leads to smaller/more granular `Presenters` in MVP vs. MVC.
- Most frameworks mentioned previously are primarily classified as MVP now, but the distinction is often slightly arbitrary (one can implement both).
- Same considerations for server-side vs client-side rendering apply.

Model View ViewModel (MVVM)



Model View ViewModel

Model View ViewModel

- Very similar to MVP, but the relationship between the `View` and the `ViewModel` is *declarative*, while the relationship between the `Presenter` and the `View` is *imperative*.
  - `Presenter` listens to the events emitted by the `View` and manipulates it accordingly.
  - `ViewModel` does not interact with the `View` directly, but observes the declared properties through a special *data binding* layer.

Model View ViewModel

- Very similar to MVP, but the relationship between the `View` and the `ViewModel` is *declarative*, while the relationship between the `Presenter` and the `View` is *imperative*.
  - `Presenter` listens to the events emitted by the `View` and manipulates it accordingly.
  - `ViewModel` does not interact with the `View` directly, but observes the declared properties through a special *data binding* layer.
- More robust to changes in the View hierarchy.

Model View ViewModel

- Very similar to MVP, but the relationship between the `View` and the `ViewModel` is *declarative*, while the relationship between the `Presenter` and the `View` is *imperative*.
  - `Presenter` listens to the events emitted by the `View` and manipulates it accordingly.
  - `ViewModel` does not interact with the `View` directly, but observes the declared properties through a special *data binding* layer.
- More robust to changes in the View hierarchy.
- Less verbose, data binding eliminates a lot of "boilerplate" code related to events.

Model View ViewModel

- Very similar to MVP, but the relationship between the `View` and the `ViewModel` is *declarative*, while the relationship between the `Presenter` and the `View` is *imperative*.
  - `Presenter` listens to the events emitted by the `View` and manipulates it accordingly.
  - `ViewModel` does not interact with the `View` directly, but observes the declared properties through a special *data binding* layer.
- More robust to changes in the View hierarchy.
- Less verbose, data binding eliminates a lot of "boilerplate" code related to events.
- Most modern JavaScript frameworks (React, Svelte, Vue.js, ...) could be classified as MVVM (but most have other mechanisms, on top of MVVM, e.g. templates, that make this less clear)

We'll come back to declarative vs. imperative UI in the next lecture...

Part 7: Component-based design

Problem:

The UI Element hierarchy on its own is often quite hard to manage, even if we only consider "Element state" that has nothing to do with business logic. To achieve a specific style, we may need to combine a lot of Elements that need we to coordinate and thus expose unnecessary "implementation details".

UI Components

UI Components

- Encapsulates a complex, but self-contained UI `Element` hierarchy, making it reusable.

UI Components

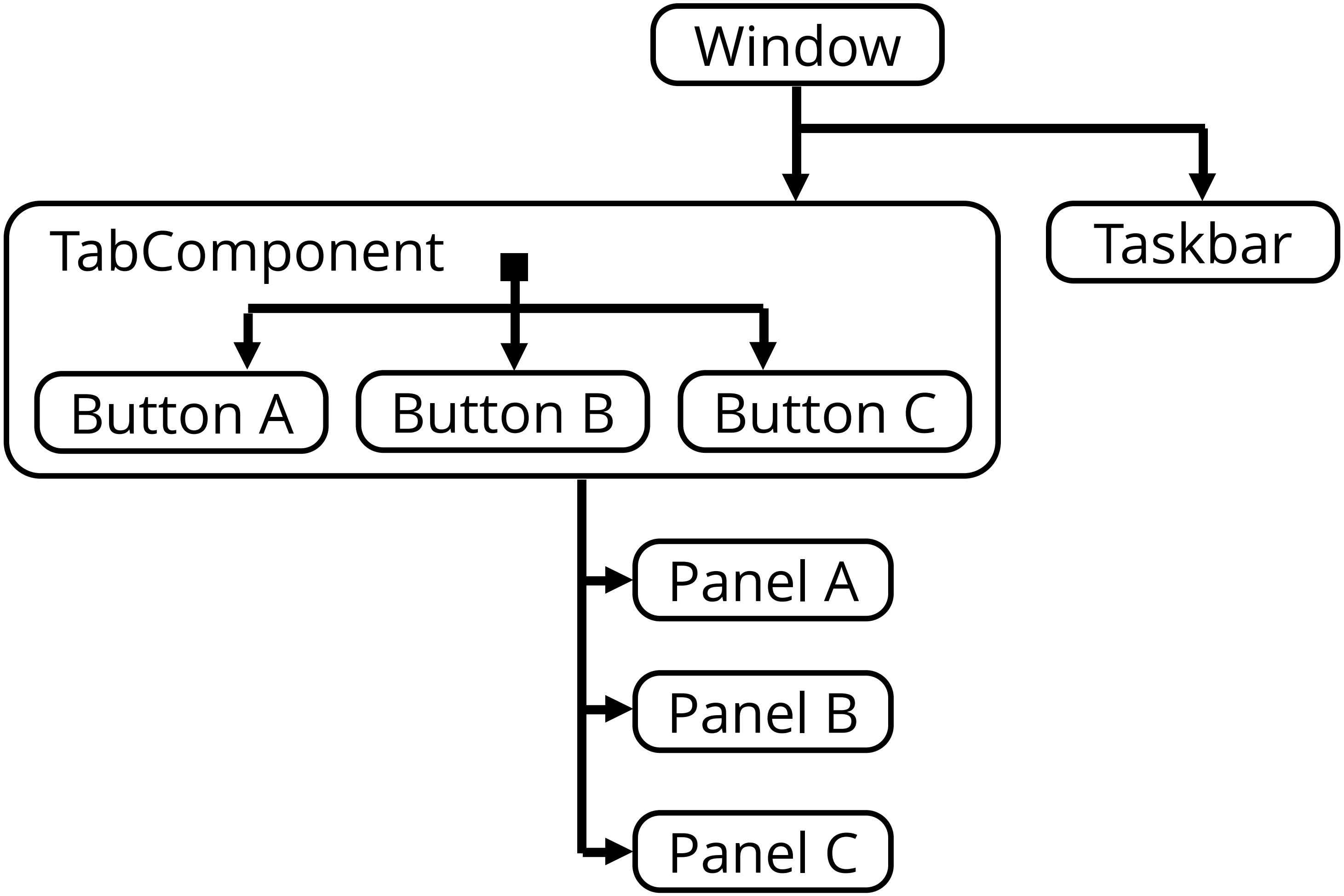
- Encapsulates a complex, but self-contained UI `Element` hierarchy, making it reusable.
- "From the outside", a `Component` appears as just another `Element`.

UI Components

- Encapsulates a complex, but self-contained UI `Element` hierarchy, making it reusable.
- "From the outside", a `Component` appears as just another `Element`.
- A well-designed `Component` encapsulates the state of multiple `Elements` that would otherwise have to interact with each other and share state.
    - Architecturally, this allows us to convert some of the "Session-like" state into pure Element state.

UI Components

- Encapsulates a complex, but self-contained UI `Element` hierarchy, making it reusable.
- "From the outside", a `Component` appears as just another `Element`.
- A well-designed `Component` encapsulates the state of multiple `Elements` that would otherwise have to interact with each other and share state.
    - Architecturally, this allows us to convert some of the "Session-like" state into pure Element state.
- Was hugely popularized by React (now used by most JS frameworks), but similar ideas have existed outside of Web development before.


Window
TabComponent
Taskbar
Button A
Button B
Button C
Panel A
Panel B
Panel C

WebComponents

- https://developer.mozilla.org/en-US/docs/Web/API/Web_components/

WebComponents

- https://developer.mozilla.org/en-US/docs/Web/API/Web_components/
- A Web standard for implementing UI Components.
  - Came after React and similar frameworks, but provides tighter integration with browsers (e.g. CSS and Event encapsulation).
  - You can mix WebComponents and existing JS frameworks, since each `Component` is just a standard HTML `Element`.

WebComponents

- https://developer.mozilla.org/en-US/docs/Web/API/Web_components/
- A Web standard for implementing UI Components.
    - Came after React and similar frameworks, but provides tighter integration with browsers (e.g. CSS and Event encapsulation).
    - You can mix WebComponents and existing JS frameworks, since each `Component` is just a standard HTML `Element`.
- Each `Component` behaves just like an ordinary HTML `Element` with a custom tag name.

WebComponents

- https://developer.mozilla.org/en-US/docs/Web/API/Web_components/
- A Web standard for implementing UI Components.
  - Came after React and similar frameworks, but provides tighter integration with browsers (e.g. CSS and Event encapsulation).
  - You can mix WebComponents and existing JS frameworks, since each `Component` is just a standard HTML `Element`.
- Each `Component` behaves just like an ordinary HTML `Element` with a custom tag name.
- A `Component` can have a *shadow DOM*, i.e. a hierarchy of HTML elements that is not accessible outside of the component.

WebComponents

- https://developer.mozilla.org/en-US/docs/Web/API/Web_components/
- A Web standard for implementing UI Components.
    - Came after React and similar frameworks, but provides tighter integration with browsers (e.g. CSS and Event encapsulation).
    - You can mix WebComponents and existing JS frameworks, since each `Component` is just a standard HTML `Element`.
- Each `Component` behaves just like an ordinary HTML `Element` with a custom tag name.
- A `Component` can have a *shadow DOM*, i.e. a hierarchy of HTML elements that is not accessible outside of the component.
- The standard adds a `<template>` tag to make writing reusable HTML markup easier.

WebComponents

WebComponents

- A component is either *autonomous*, or it extends an existing HTML element.

WebComponents

- A component is either *autonomous*, or it extends an existing HTML element.
- Tag names must contain a `"-"` to differentiate them from current (and future) HTML elements.

WebComponents

- A component is either *autonomous*, or it extends an existing HTML element.
- Tag names must contain a "`-`" to differentiate them from current (and future) HTML elements.

```
// Autonomous Web Component
class TabBar extends HTMLElement {
  constructor() {
    super();
  }
  // Element functionality written in here
}

customElements.define("tab-bar", TabBar);

// Usage in HTML:
// <tab-bar>Some content</tab-bar>
```

WebComponents

- Components have lifecycle events: `connected`, `adopted`, `disconnected`.

WebComponents

- Components have lifecycle events: `connected`, `adopted`, `disconnected`.

```
// Autonomous Web Component
class TabBar extends HTMLElement {
  constructor() {
    super();
  }

  connectedCallback() {
    console.log("Custom element added to page.");
  }

  disconnectedCallback() {
    console.log("Custom element removed from page.");
  }

  adoptedCallback() {
    console.log("Custom element moved to new page.");
  }

}
```

WebComponents

- A *shadow DOM* is used to attach child elements that should not be discoverable through the normal DOM.
  - They won't appear as child `Elements` nor can they be discovered using `querySelector` or similar.
  - If mode: "open" is set, the shadow DOM is accessible through `element.shadowRoot`.

WebComponents

- A *shadow DOM* is used to attach child elements that should not be discoverable through the normal DOM.
    - They won't appear as child `Elements` nor can they be discovered using `querySelector` or similar.
    - If mode: "open" is set, the shadow DOM is accessible through `element.shadowRoot`.

```
// Autonomous Web Component
class TabBar extends HTMLElement {
  static observedAttributes = ["color", "size"];

  connectedCallback() {
    // Create a shadow root
    const shadow = this.attachShadow({ mode: "open" });

    // Create internal span tag
    const wrapper = document.createElement("span");
    wrapper.setAttribute("class", "wrapper");
    wrapper.innerHTML = "Shadow DOM content";

    shadow.appendChild(wrapper);
  }
}
```

WebComponents

- Templates provide a friendlier mechanism for declaring shadow DOM elements

WebComponents

- Templates provide a friendlier mechanism for declaring shadow DOM elements

```
// Declared somewhere in our HTML:
<template id="tab-bar">
  <div id="tab-buttons"></div>
  <div id="tab-panels">
    <slot>Child elements will appear here.</slot>
  </div>
</template>

class TabBar extends HTMLElement {
  constructor() {
    super();
    let template = document.getElementById("tab-bar");
    let templateContent = template.content;

    const shadowRoot = this.attachShadow({ mode: "open" });
    shadowRoot.appendChild(templateContent.cloneNode(true));
  }
}
```

WebComponents

- Child elements can be placed depending on slot name.

WebComponents

- Child elements can be placed depending on slot name.

```
// Declared somewhere in our HTML:
<template id="tab-bar">
  <div id="tab-buttons">
    <slot name="header"></slot>
  </div>
  <div id="tab-panels">
    <slot>Child elements will appear here.</slot>
  </div>
</template>

// Usage with tab-bar:
<tab-bar>
    <span slot="header">Some header content.</span>
    <div>Other elements go to the default slot.</div>
</tab-bar>
```

WebComponents

- Same as JavaScript, the shadow DOM is also encapsulated from any "outside" CSS rules.
- We need to either add our own styles programmatically, or add a `<style>` (or `<link>`) directly into the shadow DOM.

WebComponents

- Same as JavaScript, the shadow DOM is also encapsulated from any "outside" CSS rules.
- We need to either add our own styles programmatically, or add a `<style>` (or `<link>`) directly into the shadow DOM.

```
<template id="tab-bar">
  <style>
    .tab-buttons {
      margin: 0 auto;
    }
  </style>
  <div id="tab-buttons">
    <slot name="header"></slot>
  </div>
  <div id="tab-panels">
    <slot>Child elements will appear here.</slot>
  </div>
</template>

```

More about WebComponents (and components in general) on the next seminar...

Takeaways

- User interfaces are represented as tree hierarchies of elements (widgets, views, ...).
- Interacting with elements triggers events.
- Events update application state. Managing state is one of the fundamental roles of UI frameworks.
- Very broadly, state can be understood as element state, session state and persistent state.
- Server-side vs. client-side rendering: where is the state converted into the UI hierarchy?
- MVC, MVP, MVVM: basic design patterns for separating state, logic and user interface.
- Component-based design: defining new UI elements that encapsulate non-trivial behavior.
- WebComponents, a modern standard for defining UI components in the browser.