# React Snake: A guided tour of a react application

1. Project structure
   1. Repository root contains typical configuration and management, including the server
   2. web-root houses the actual web app code
   3. The src and dist directories allow us to separate raw source code such as Typescript and SASS from the finished, browser-ready state
      1. Webpack compiles, bundles, and minifies sources
      2. Everything else is directly copied into dist
   4. Style: react components belong in individual files in components, data model and logic belong in model
2. Typescript feature overview: Lattice.ts and SnakeGameController.tsx
   1. Modules/import/export
      1. Use \* to put exports into an object
      2. Pattern-match style to pick and choose what to import
      3. Imports are just for reference by name, not for general type checking
   2. Interface: Size and PathCell
      1. A list of expected Properties and functions
      2. Optional types: use “?”
         1. Properties or function parameters can be optional
         2. Automatic “| undefined”
         3. Additionally, allows implicit undefined by omitting value
      3. Doesn’t need to be explicitly implemented – TS checks for conformity on the fly
      4. Note: readonly on the interface doesn’t require readonly on its implicit conformers – just its explicit conformers.
   3. Classes: Cell
      1. Typical C-like OOP classes
         1. Single inheritance plus arbitrarily many interfaces
         2. Fields/properties and functions
         3. Static options
         4. Public/protected/private (default public)
      2. Single constructor, with sugar for immediate property setting and “super”
      3. Must use “this” everywhere
   4. Misc: Direction
      1. Default parameter values
      2. Type aliasing
      3. String literal type
         1. Switch on enum/string union is exhaustive
      4. Type unions (and intersections)
         1. Guard on types with typeof or instanceof or even null checks
      5. Generics
   5. Not shown: enums, which are numbers with special names.
3. Winding through React Snake
   1. Forget all game logic for now; let’s focus on presentation.
   2. Connecting React to the DOM: SnakeGameController.tsx
      1. Call ReactDom.render with the root component and the DOM element to which it should be attached.
      2. Use the “spread” operator (...) to pass each property of an object as props to the component
      3. The ref property allows a parent to get the component or DOM element for a snippet of JSX.
         1. Use it sparingly. Generally, just pass everything you need as props rather than manually interacting with the element
         2. Helpful or necessary for things like text field focus, animations, and other things that fall outside the typical render pipeline.
   3. Everyday React component: SnakeGame
      1. Create a component class that extends React.Component
         1. Generic type parameters: props and state
         2. Style: declare a type for both (if used)
         3. Style: use undefined for type of unused state
         4. Export the Props type to allow explicit use by parent
      2. Listen to key presses
         1. Use ref to get the div element so we can focus it
         2. Delegate key press handling to parent (via prop callback)
            1. Pass handler from props to the child’s event prop
            2. Style: event handler names should start with handle and end with the event being handled
         3. Hook into lifecycle with componentDidUpdate to ensure the game remains focused
      3. Provide functions to allow parent “controller” to trigger state changes
         1. Don’t do this normally; this is only for the root node.
         2. Set the state based on the provided information
      4. Render the game based on current state and props
         1. Return JSX from the render function
         2. JSX must have a single parent tag/component
         3. Rule: Capital letters imply Components, lowercase means html
         4. Control flow by expressions that resolve to the proper tag
            1. Ternary conditional operator for this/that cases
            2. Boolean “bool &&” for this/none cases
            3. An array will be rendered as siblings

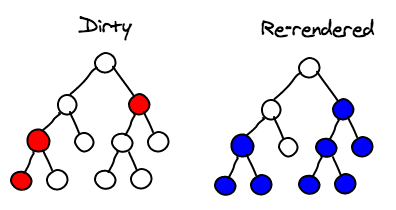
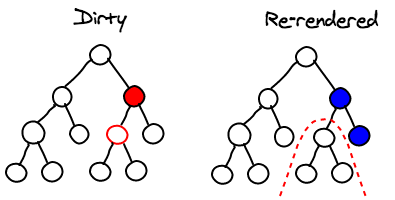
Precompute before JSX

Use functions like map

* 1. Arena
     1. Very simple component
     2. CSS magic around each piece of the game (snake, walls, food)
  2. Walls
     1. Walls returns an instance of CellContainer along with the cell size and a collection of Wall components.
        1. Use map to generate each wall
        2. Looped children need keys (see diagrams)
        3. Both CellContainer and Wall are just divs with special positioning styles
        4. Passing children this way are accessed through props.children
     2. Since walls never change, we can hardwire shouldComponentUpdate to return false (see diagrams)
  3. Snake, plus its cell children
     1. Another simple pass-through to CellContainer.
     2. Best practice: components should compose rather than inherit
        1. Still sort of an is-a relationship
        2. The cells are a mess of an inheritance hierarchy. Avoid this. Just replace “X is a subclass of Y” with “X renders a specialized Y”.
  4. FoodController: How to C amongst all the V
     1. For Separation of Concerns, we don’t want “logic” in our presentation
     2. We could use Flux.
     3. Otherwise, separate the logic into a container component that maintains state and updates the presentation component, Foods
     4. Now food is purely stateless presentation
        1. Style: Stateless functional components don’t need their own file.

# Some Odds and Ends A little bit of Diff

React avoids unnecessary DOM updates by first rendering *everything* to a virtual DOM and comparing changes to that. If a node (or an ancestor of a node) has changed, it will be re-rendered in the real DOM. However, although it’s much cheaper to render and compare virtual components, performance can often be improved by short circuiting the process with shouldComponentUpdate. This function is called with the previous and current props given to the component for comparison. If this function returns false, the node and its children will not even be rendered virtually, and React will assume it has not changed. By default, it returns true.

## Pure Components

When conforming to best practices, components will only need to be rendered if their props or state have changed. In the case where this is true (as determined by a **shallow** compare of state/props), React provides the PureComponent class that automatically provides a shallow-compare implementation of shouldComponentUpdate. In most cases, extend PureComponent instead of Component. Due to the shallow copy requirement, pure components must only make changes to persistent, immutable data structures. The [Immutable.js library](https://facebook.github.io/immutable-js/) provides immutable data structures that help keep components pure.

## Stateless Functional Components

When your pure components also have no state, you can make them a “stateless functional component”. Instead of creating a class that extends Component, you can create a function that takes props and returns an element. When rendering, React will simply call this function. Note: React doesn’t shortcut stateless components like it does pure components. It’s best used under a seldom-rendered parent.

## The Why and How of Keys

Keys allow React to optimize list re-rendering. When determining which components in a list have changed (or are new or deleted), it uses the key to match component instances. If the list is reordered or inserted before the end, unique keys allow React to reuse the existing components.  
Thus, make sure to provide unique keys when multiple sibling elements of the same component exist. Additionally, make sure those keys are bound to the element and won’t change on reorder.

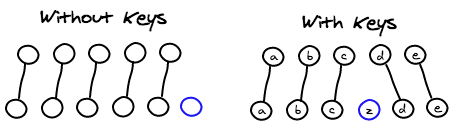


Diagram Credits: Christopher Chedeau, https://calendar.perfplanet.com/2013/diff/