**<lecture6. user interfaces>**

**Single Page Applications**

Previously, if we wanted a website with multiple pages, we would accomplish that using different routes in our Django application. Now, we have the ability to load just a single page and then use JavaScript to manipulate the DOM. One major advantage of doing this is that we only need to modify the part of the page that is actually changing. For example, if we have a Nav Bar that doesn’t change based on your current page, we wouldn’t want to have to re-render that Nav Bar every time we switch to a new part of the page.

Let’s look at an example of how we could simulate page switching in JavaScript:



Notice in the HTML above that we have three buttons and three divs. At the moment, the divs contain only a small bit of text, but we could imagine each div containing the contents of one page on our site. Now, we’ll add some JavaScript that allows us to use the buttons to toggle between pages.



line9: forEach 돌면서 모든 div 태그 style을 display = 'none'으로 설정

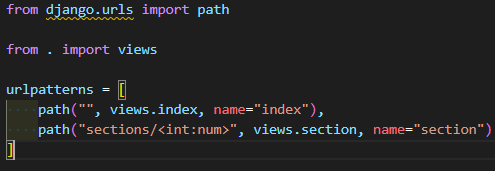
line13: showPage의 parameter 'page'를 받아서 style을 display = 'block'으로 변경.

line17: querySelectorAll과 forEach로 모든 button 태그 선택하고 button.onclick 하면 showPage의 parameter로 dataset의 page를 불러옴.

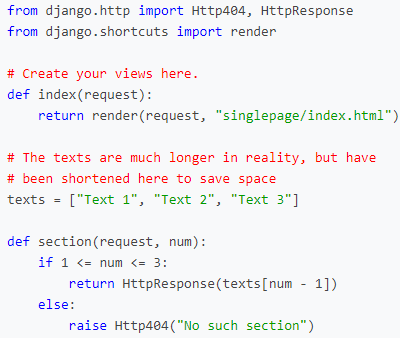
결과적으로 각 button을 클릭하면 showPage(page1), showPage(page2), showPage(page3)가 실행되어 id = page{$}가 매겨진 div 내의 내용들이 보이게 됨.

**Single Page Applications in Django**

**# singlepage/urls.py**



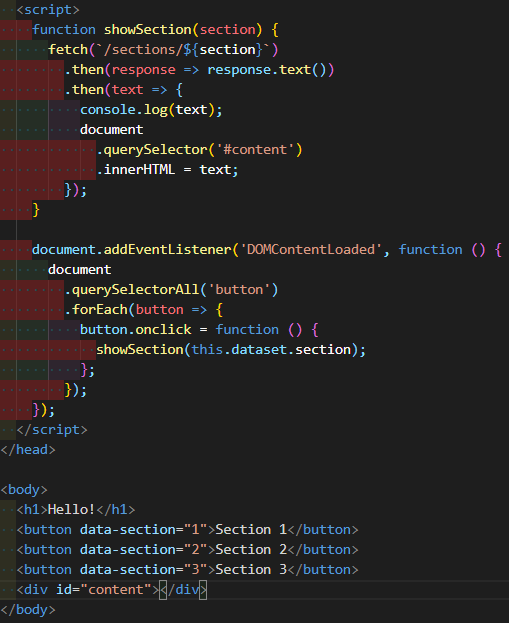
**# singlepage/views.py**

****

And two corresponding routes in views.py. Notice that the section route takes in an integer, and then returns a string of text based on that integer as an HTTP Response.

index.html 파일의 button 클릭을 통해 입력 받은 num값에 해당하는 array texts[num - 1]의 값을 반환.

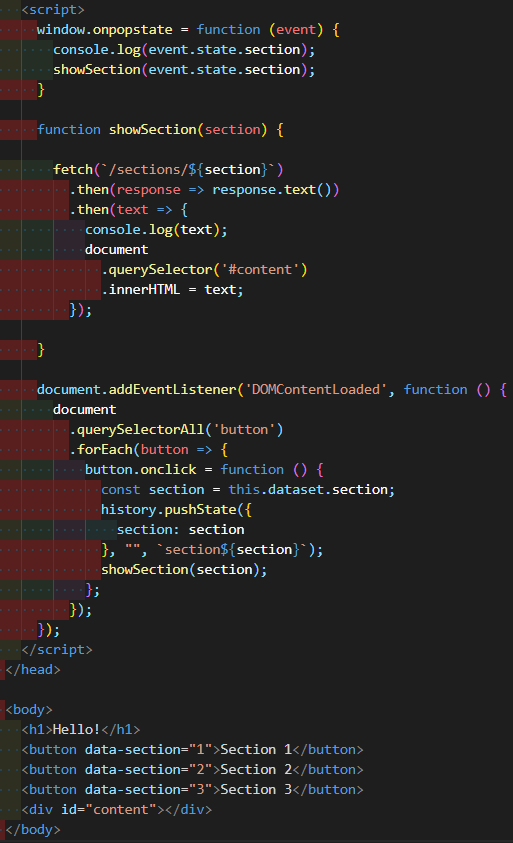
**# templates/singlepage/index.html**



within our index.html file, we’ll take advantage of AJAX, which we learned about last lecture, to make a request to the server to gain the text of a particular section and display it on the screen.

각 버튼을 클릭할 때마다 dataset의 data를 showSection의 parameter로 집어넣고, showSection의 parameter가 /sections/${section}에 성공적으로 입력되면 text형식의 data를 반환하고, text형식의 data를 성공적으로 반환하면, #content의 innerHTML로 text를 반환.

**JavaScript History API**



One disadvantage of our site though is that the URL is now less informative. You’ll notice in the video above that the URL remains the same even when we switch from section to section. We can solve this problem using the JavaScript History API. This API allows us to push information to our browser history and update the URL manually. Let’s take a look at how we can use this API. Imagine we have a Django project identical to the previous one, but this time we wish to alter our script to be employ the history API.

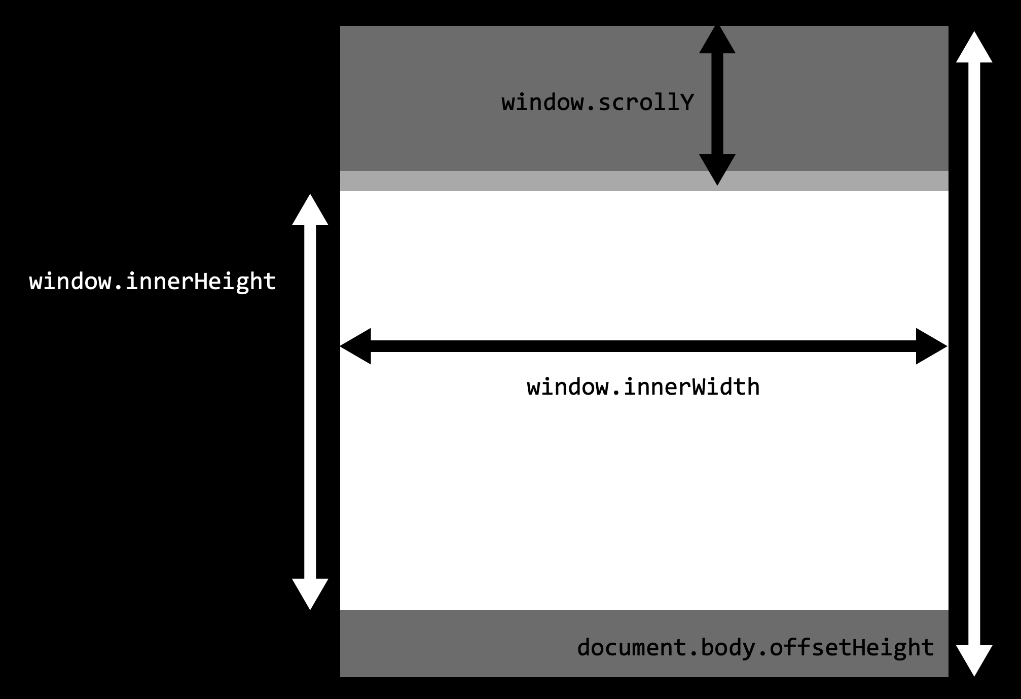
In the showSection function above, we employ the history.pushState function. This function adds a new element to our browsing history based on three arguments:

1. Any data associated with the state.
2. A title parameter ignored by most web browsers
3. What should be displayed in the URL
4. <https://developer.mozilla.org/ko/docs/Web/API/History/pushState>

The other change we make in the above JavaScript is in setting the onpopstate parameter, which specifies what we should do when the user clicks the back arrow. In this case, we want to show the previous section when the button is pressed. Now, the site looks a little more user-friendly:

history.pushState로써 세션 기록 항목에 state, title, url을 각각 저장함.

**Scroll**



* window.innerWidth: Width of window in pixels
* window.innerHeight: Height of window in pixels
* window.scrollY: How many pixels we have scrolled from the top of the page
* document.body.offsetHeight: The height in pixels of the entire document.

The following code, for example, will change the backgroud color to green when we reach the bottom of a page:

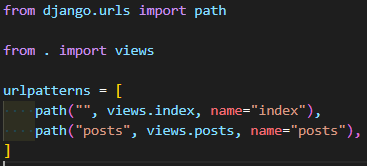
텍스트이(가) 표시된 사진

자동 생성된 설명

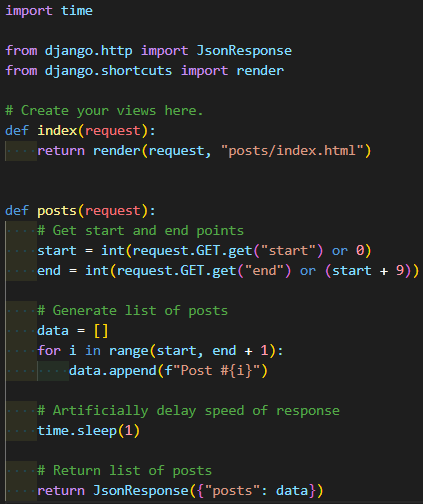
**Infinite Scroll**

if you’re on a social media site, you don’t want to have to load all posts at once, you might want to load the first ten, and then when the user reaches the bottom, load the next ten.

**# urls.py**

****

**# views.py**

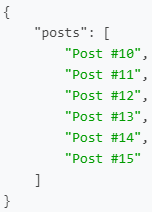
****

**# index.html**



스크롤해서 scrollY 좌표가 window 최하단에 닿으면 새로운 post 생성.

Notice that the posts view requires two arguments: a start point and an end point. In this view, we’ve created our own API, which we can test out by visiting the url localhost:8000/posts?start=10&end=15, which returns the following JSON:



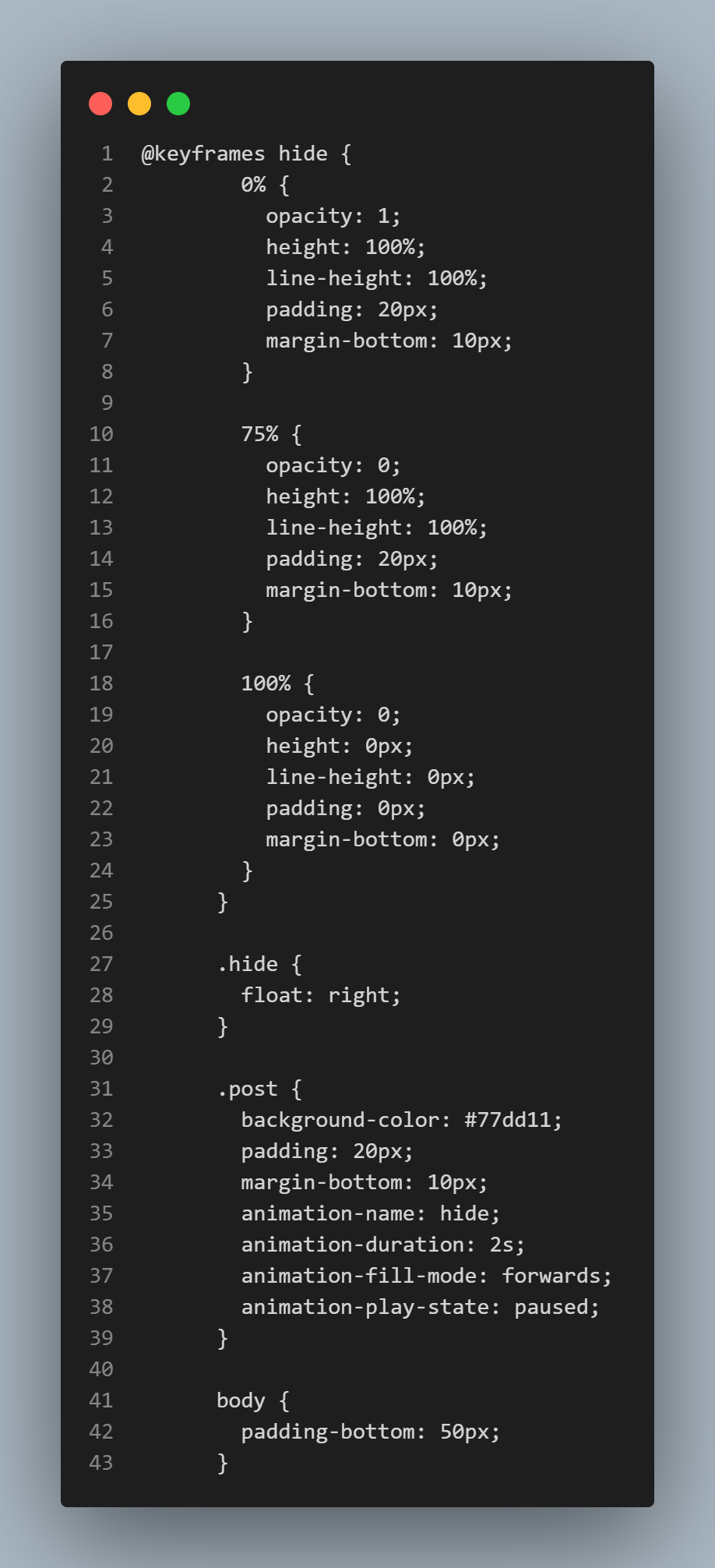
views.py와 index.html을 거치면 위처럼 json 형식의 data를 반환하여 post를 생성한다.

**Animation**



버튼 클릭하면 animation 작동. h1 좌우로 왔다갔다 함.

**# CSS in index.html**



**# script in index.html**



post 내 hide 버튼 누르면 animation 효과 주면서 사라짐.

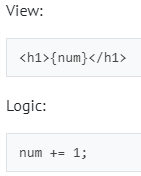
**React**

**imperative programming vs declarative programming**

imperative programming methods, where we give the computer a set of statements to execute. For example, to update the counter in an HTML page we might have have code that looks like this:



React allows us to use declarative programming, which will allow us to simply write code explaining what we wish to display and not worry about how we’re displaying it. In React, a counter might look a bit more like this:

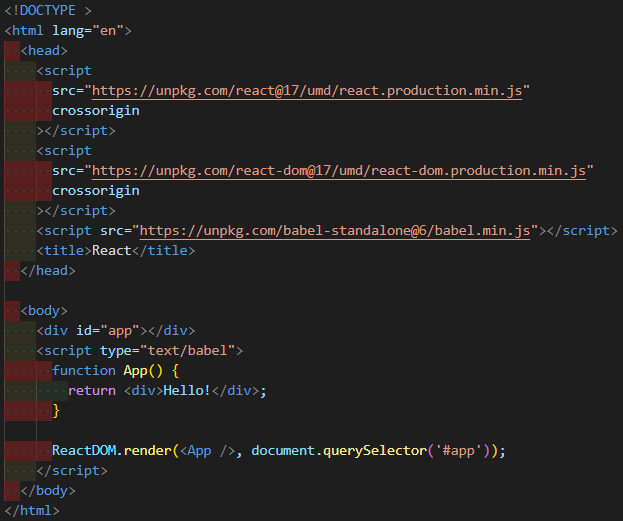


The beauty of React is that when the state changes, React will automatically change the DOM accordingly.

There are a number of ways to use React, (including the popular create-react-app command published by Facebook) but today we’ll focus on getting started directly in an HTML file. To do this, we’ll have to import three JavaScript Packages:

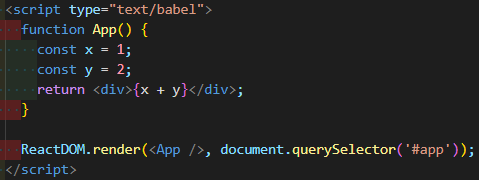
* React: Defines components and their behavior
* ReactDOM: Takes React components and inserts them into the DOM
* Babel: Translates from JSX, the language in which we’ll write in React, to plain JavaScript that our browsers can interpret. JSX is very similar to JavaScript, but with some additional features, including the ability to represent HTML inside of our code.

**# react.html**



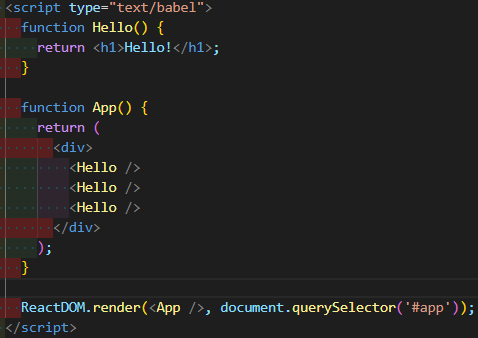
* In the body, we include a single div with an id of app. We almost always want to leave this empty, and fill it in our react code below.
* We include a script tag where we specify that type="text/babel". This signals to the browser that the following script needs to be translated using Babel.
* Next, we create a component called App. Components in React can be represented by JavaScript functions.
* Our component returns what we would like to render to the DOM. In this case, we simply return <div>Hello!</div>.
* The last line of our script employs the ReactDOM.render function, which takes two arguments:

1. A component to render
2. An element in the DOM inside of which the component should be rendered



javascript에서 처럼 {}로 묶어서 변수 활용 가능.

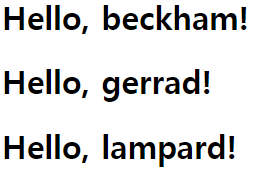
**Render components with other components**



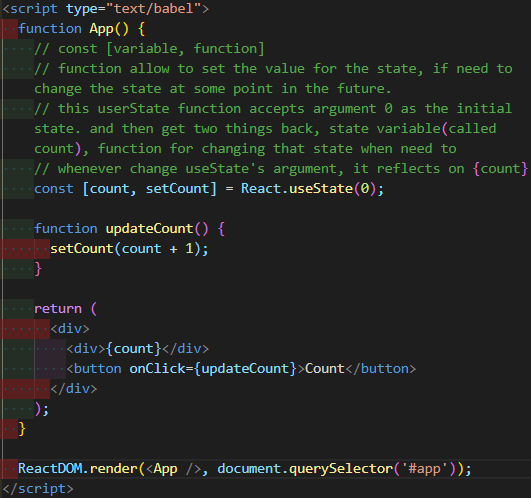
구조를 App() > Hello() \* 3 라고 생각하면 됨.

**Props**

react에 쓰이는 components를 props를 추가하여 보다 flexible하게 쓸 수 있다.



**useState**



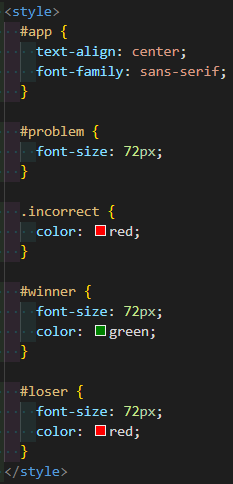
The **argument to** **useState** is the initial value of the state, which we’ll set to 0. The function returns both a variable representing the state and a function that allows us to update the state.

Now, we can work on what the function will render, where we’ll specify a header and a button. We’ll also add an event listener for when the button is clicked, which React handles using the **onClick** attribute.

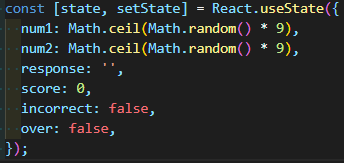
Finally, let’s define the **updateCount** function. To do this, we’ll use the **setCount** function, which can take as argument a new value for the state.

**Create simple game by react useState and functions**

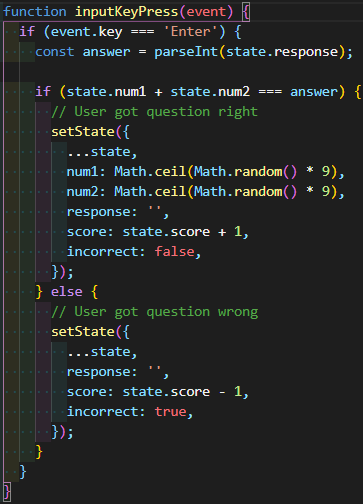
**# CSS in number\_game.html**

****

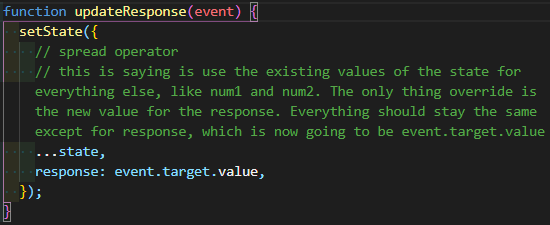
**# set useState in number\_game.html**

****

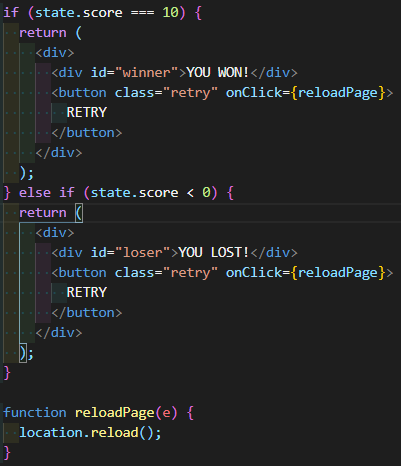
**# inputKeyPress() in number\_game.html**

****

**# updateResponse() in number\_game.html**

****

**# game end condition in number\_game.html**

****

**# rendering sets of page in number\_game.html**

****