DH2642 lab tutorial v4.1

Cristian Bogdan

How to use the tutorial

If you take the course on 50% pace (one period) then one tutorial week is one calendar week. If you take the course on 25% pace (two periods), one tutorial week corresponds to two calendar weeks. There are 3 tutorial weeks (so 6 weeks on 25% pace). You cannot choose the pace of the course, it is preset in the schedule.

Make sure to read the slide **and its notes** before starting to code. Most slides have notes!

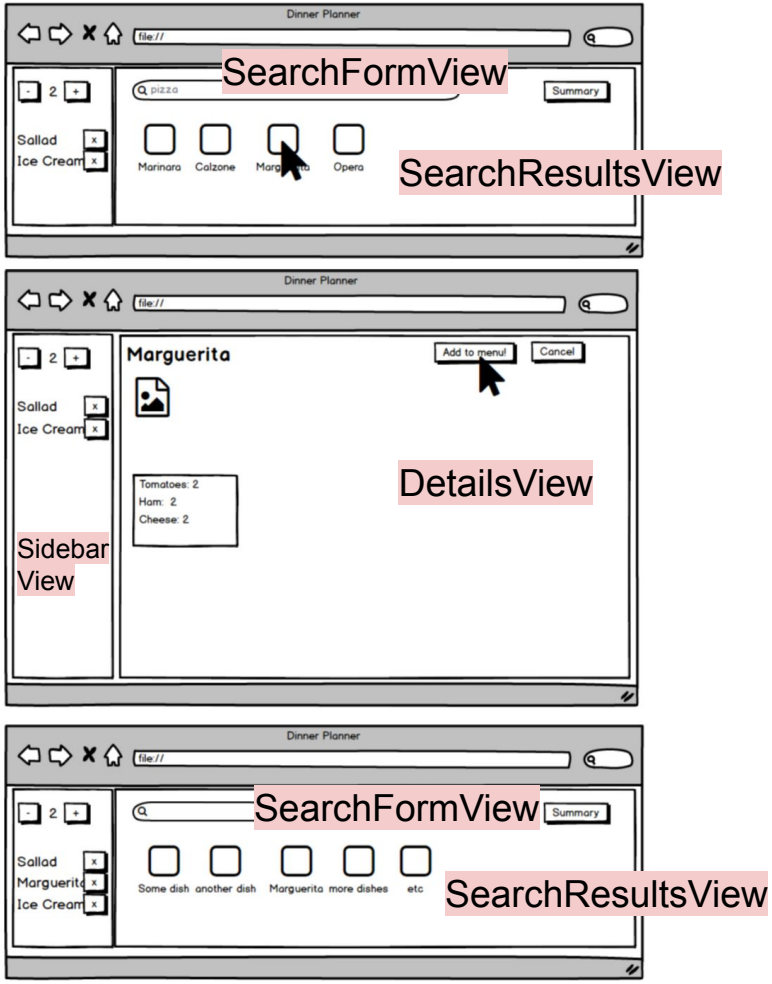
The tutorial strategy is to test *components* (Views, Presenters) *in isolation* first. This is similar to the way large projects (and hopefully your course project) are organized: you do not get to see the full product initially, but test smaller bits. If the small bits work, it is highly likely that your product will work. If any of them doesn't, you can be sure that the product will **not** work.

Also we first focus on *rendering* (drawing) the UI (Views) with the correct data, so your UI will not be initially interactive, or not fully interactive. *Please be patient*, later steps will add the interactivity.

There are few links in the slides but concepts are mentioned in **bold**. Search for them in lecture slides and/or on the internet

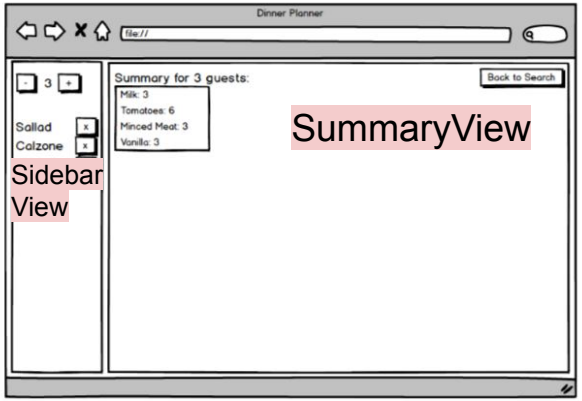
- Search is an engineering skill that you need to develop
- Not including links makes also the material easier to maintain :)

When you have completed a task, **commit to git**

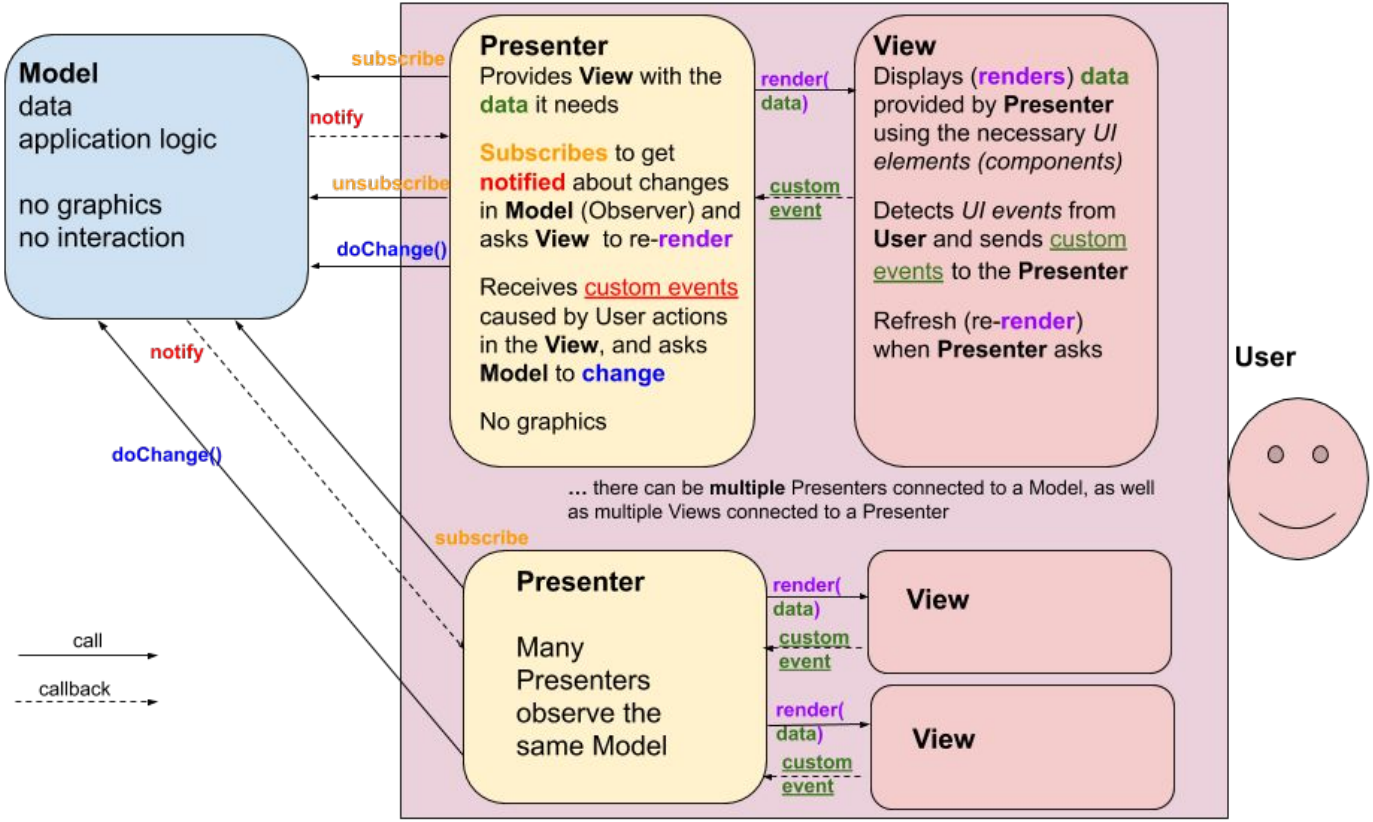


Use case: Dinner Planner

- The user is able to search for dishes in a SearchFormView and sees the results in the SearchResultsView
- By clicking on a search result, they can examine the dish in the DetailsView. There, they can add the dish to the menu.
- The menu and the number of guests is visible at all times in the SidebarView.
- At any time the user can access a shopping list in the SummaryView.



Model-View-Presenter



The tutorial is focused on testing components (Views, Presenters) *in isolation* first. We first focus on drawing (rendering) the UI (Views), so your UI will not be initially interactive, or not fully interactive.

A change in data (due to a user event in a View) always needs to be made in the Model (by the Presenter). So all of V, P, M need to be worked on before an interaction will work!

Please be patient, later steps will add the interactivity!

Git setup (in case of problems, skip this step and ask TA help at a lab session)

- In this course, you are required to use git for version control and submission.
- Make sure to have a git account at **https://gits-15.sys.kth.se/**
- Follow the official github instructions for configuring SSH. Don't skip the steps "Checked for existing SSH keys" and "Generated a new SSH key and added it to the ssh-agent". Note that **instead of github.com, you need to apply these instructions (and add your SSH key) to gits-15.sys.kth.se.**
- An empty repository will be created for you before course start
- `git clone git@gits-15.sys.kth.se:iprog-students/[your-git-username]-week1.git`
- `cd [your-git-username]-week1`
- `git pull`
- `git status`

Tutorial Week 1

Intro to JavaScript, HTML, JSX, Rendering, Events, State

```
testModel.html
js/
        DinnerModel.js
```

TW1.1 Simple JavaScript: starting the DinnerModel class

js/DinnerModel.js

```
class DinnerModel{
      constructor(guests =2){this.setNumberOfGuests(/*TODO*/);}
      setNumberOfGuests(x){ this.numberOfGuests= /*TODO*/;}
}
```

testModel.html

```
<html>
   <head>
      <script src= "js/DinnerModel.js"></script>
   </head>
   <body>Open Developer Tools to test the Model!</body>
</html>
```

Start a http server from your project folder

```
cd your_project_folder
# one of these commands should work:
python3 -m http.server
python -m SimpleHTTPServer
py -3 -m http.server
python -m http.server
```

Test at: http://localhost:8000/testModel.html

TW1.1 Testing and improving the Model

Open the developer tools (F12 or Ctrl-Shift-i, or Command-Alt-i), and type in the **Console**:

```
const myModel= new DinnerModel() // same as new DinnerModel(2)
myModel.numberOfGuests // should print 2
myModel.setNumberOfGuests(5)
myModel.numberOfGuests // should print 5
```

Change setNumberOfGuests() to **not accept** zero, negative or non-integer numbers and throw an error instead. This is called **Application Logic** and is the job of the Model.

```
myModel.setNumberOfGuests(0)  // should throw
myModel.setNumberOfGuests(-3)  // should throw
myModel.setNumberOfGuests(4.5)  // should throw
```

Commit and push your TW1.1 code

Commit after each TW step or even after each slide!

```
# make sure you are working in the project directory
git status
git add testModel.html js/DinnerModel.js
git commit -m TW1.1
git push
```

Done! You can now see in your remote repository in the browser **`https://gits-15.sys.kth.se/iprog-students/[your-git-username]--week1`** that your changes have been published.

Don’t forget **git pull** before every working session! Even if you work alone, you may have pushed from some other machine/folder.

TW1.2 Rendering

We will now focus on creating the UI (rendering) for some Views in order to practice HTML, DOM trees and JSX.

The interactive *widgets* (buttons, input boxes, selections) of each view are the most important. The rest can be adjusted later.

Modern interaction programming uses very little HTML. The HTML document BODY contains just one DIV. Most of the interface is generated (and updated) from JavaScript, inside that DIV.

JSX (JavaScript XML) is a very convenient way to generate User Interface from JavaScript. JSX is supported by both frameworks we use (React and Vue) and many others.

TW1.2 SummaryView (early version)

Summary for 4 guests:

HTML:

```
<DIV>
Summary for <SPAN title="nr. guests">4</SPAN>  guests:
</DIV>
```

DOM tree:

```
DIV
    Summary for
    SPAN title
        4
    guests:
```

JSX:

```
const persons=4;

<div>
Summary for <span title="nr. guests">{persons}</span> guests:
</div>
```

We will write the JSX code in a file and test it in the next step(s) !

React setup (file: **react.html** in your project folder)

```
<html>
<head>
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
<script src="https://unpkg.com/react@17/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>
</head>
<body>
<div id="app"></div>
</body>

<script type="text/jsx">
const persons=4;
ReactDOM.render(
   <div>
       Summary for <span title="nr. guests">{persons}</span> guests:
   </div>
,  document.getElementById("app")
);
</script>

</html>
```

Test at: http://localhost:8000/react.html

Babel is needed to translate from JSX to JavaScript

*Under Developer Tools, **Sources**, check "Inline Babel Script". You can debug there.*

Babel generates calls to a function called hyperscript or createElement. You can find them in Elements in a <script> at the end of <head>

The **Orange-marked curly braces** are a way to "move" from JSX tags to JavaScript when writing JSX code. In the curly braces you can start new JSX tags, with new JSX->JS curly braces and so on...

Unfortunately there are many meanings for curly braces in JavaScript (function body, if/for code blocks, object literals, object destructuring), JSX adds also this one.

Vue setup (file: **vue.html** in your project folder)

```
<html>
<head>
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
<script>
  // Babel generates calls to React.createElement by default, so we redefine that:
  const React={createElement:Vue.h};
</script>
</head>

<body>
<div id="app"></div>
</body>

<script type="text/jsx">
const persons=4;
Vue.render(
<div>Summary for
      <span title="nr. guests">{persons}</span> guests:
</div>
,  document.getElementById("app")
);
</script>
</html>
```

Note that the JSX code is the same in React and Vue. **React JSX and Vue 3 JSX are almost identical**. In this lab we will program one JSX file per View. All View code will work with both React and Vue.

Test at: http://localhost:8000/vue.html

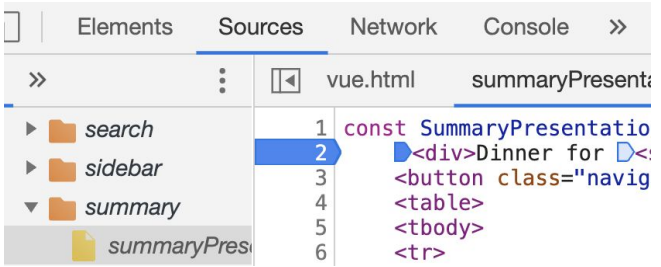
TW1.2 Developer Tools **Sources** and **Elements**

Check the HTML file under **Sources** and compare with **Elements.** Look especially under <body> in **Elements**

> The `<div id="app">` was extended by the framework with the content of the JSX

In **Elements** check the end of the `<head>`. Babel has added there a JavaScript `<script>` generated from the JSX. That's what the browser executes based on your JSX.

Under **Sources** you can also see a **Babel Script** in *italics.* This is where you can add breakpoints and debug in JSX like you can do in a normal JavaScript file.



```
react.html
vue.html
js/
        DinnerModel.js
        views/
                summaryView.js
```

TW1.2 JSX Custom Component

Isolate the SummaryView in a function. Name must start with a capital letter!

*Never change the **props**!*
Read them, never write!

```
// new file js/views/summaryView.js
function SummaryView(props){
    return (  // a lonely return on a line returns undefined. Parentheses needed
        <div>
            Summary for <span title="nr. guests">{props.persons}</span> guests:
        </div>
    );
}
```

In HTML (React and/or Vue)

```
<script src="js/views/summaryView.js" type="text/jsx"></script>
```

To **test**, write inside a `<script type="text/jsx" >` :

```
ReactDOM.render(<SummaryView persons={5} />, document.getElementById("app"));
```

Test at: http://localhost:8000/react.html

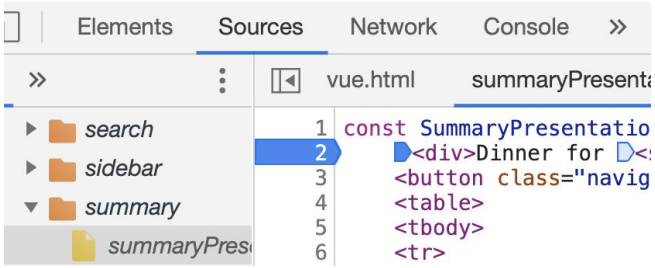
or:

```
Vue.render(<SummaryView persons={5} />, document.getElementById("app"));
```

Test at: http://localhost:8000/vue.html

In **Sources** you can place a breakpoint in **summaryView.js**, say at the "Summary for…" line. Reload the page, the execution should stop.

Locate the **Debugger** "Scope". Check the **props** object, have the right props been sent to the custom component? ***One frequent mistake is to forget to send certain props.***



JavaScript functions can be written in many ways

```
function SummaryView(props){
    return <div>   // no need for parentheses around <div> since return is not alone on the line
              Summary for <span title="nr. guests">{props.persons}</span> guests:
       </div>;
}

const SummaryView= function(props){/* same function body as before */};// anonymous function defined and set to a const

const SummaryView= (props)=> { /* same function body as before */ };    // thick arrow function. All are anonymous

const SummaryView =(props) =>          // thick arrow functions returning a single expression don't need a body
       <div>                           // expression (e.g. <div>) can start on the next line (unlike after return)
              Summary for <span title="nr. guests">{props.persons}</span> guests:
       </div>;

const SummaryView =     props =>  /* same expression as before */ // if single parameter, parentheses can be omitted

function SummaryView({persons}){                            // parameters use  JavaScript object destructuring
    return <div>
              Summary for <span title="nr. guests">{persons}</span> guests:
       </div>;
}

const SummaryView =({persons}) =>      // thick arrow function combined with object destructuring. Parentheses needed!
       <div>
              Summary for <span title="nr. guests">{persons}</span> guests:
       </div>;
```

TW1.2 SidebarView (early version)

```
react.html
vue.html
js/
        DinnerModel.js
        views/
                sidebarView.js
                summaryView.js
```

DOM tree:

DIV
- BUTTON disabled
  - --
- *guests*
- BUTTON
  - +

```
// new file js/views/sidebarView.js
function SidebarView(props)
```

Implement `SidebarView` **given this DOM tree**. Assume that the the number of guests is sent in the **guests** prop. **Hint:** `<button>+</button>`

Make the Minus button be DISABLED when the number of guests is 1 or less, so the user cannot reduce it any longer.

```
<button disabled={boolean expression} >
-
</button>
```

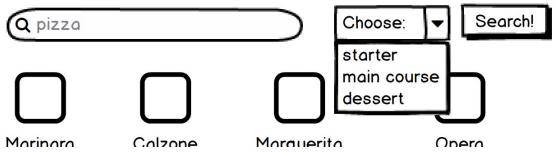
Don't forget to import sidebarView.js in the HTML like you did with summaryView!

```
Test <SidebarView guests={4} />
```

Test also `<SidebarView guests={1} />`
minus button should be disabled!

TW1.2 SearchFormView. Array rendering

```
react.html
vue.html
js/
        DinnerModel.js
        views/
                searchView.js
                sidebarView.js
                summaryView.js
```



DOM tree:

DIV
- INPUT
- **SELECT**
  - **OPTION**
    - **Choose:**
  - **OPTION** (*repeated*)
    - ***optionString***
- BUTTON
  - Search!

```
// new file js/views/searchView.js
function SearchFormview(props)
```

Assume that there is a prop called options, an array of strings. Use array rendering to render the HTML SELECT and the OPTIONs inside it.

```
<select>
      <option>Choose:</option>
      {props.options.map(
                function(opt){/*TODO*/})}
</select>
```

Hint: the **anonymous function** above should return a JSX <option > TODO </option>

Test in React or Vue

```
<SearchFormView options={["starter", "main course", "dessert"]} />
```

TW 1.3 DOM Events

// file js/views/searchView.js

```
function eventPrinter(evt){ console.log(evt);}
```

In SearchFormView add the listener:

```
<input onInput={eventPrinter} />
```

and

```
<select onChange={eventPrinter} >...
```

and

```
<button onClick={eventPrinter}>...
```

Only in Vue: change onInput for <input> to onChange! Note the difference !

In React, onChange behaves like onInput. We discuss this later

eventPrinter is the **event listener**. It is a *callback* because it is called back by the browser whenever needed.

Note that we don't call ~~eventPrinter()~~ when we pass it. **Never call a callback when you pass it!**

You can also define the function as **anonymous**. Test:

```
<input
      onInput={function(e){console.log(e.target.value);}}
/>
```

e.target is the actual INPUT DOM Element
value is a property of the INPUT DOM object

Thick arrow function. *Equivalent* code:

```
<input onInput={(e)=>{console.log(e.target.value);}}/>
```

We can simplify further

```
<input onInput={e=>console.log(e.target.value)}/>
```

Test also

```
<select onChange={e=>console.log(e.target.value) } >...
```

TW 1.3 JSX Custom events (callback props)

In the test HTML (react.html or vue.html):

```
<SearchFormView  options={["starter", "main course", "dessert"]}
      onSearch={()=>console.log("User wants to search!")}
      onText={txt=>console.log("User typed: ", txt)}
      onDishType={dishType=>console.log("User chose dish type: ", dishType)}
/>
```

In SearchFormView: replace the console.log event listeners with (note that the event object is often not needed by click handlers):

```
<button onClick={ event=> props.onSearch() } >...
```

A custom event can receive one or more parameters, like any function:

```
<input onChange={ e=> props.onText(e.target.value) } />
```

Trigger the **onDishType** custom event similar onSearch and onText.

TW1.3 SidebarView custom event

Send the `setGuests` custom event as a prop to SidebarView. In the test HTML file (react.html and/or vue.html). Take SearchFormView custom events as inspiration!

```
<SidebarView guests={3}
    setGuests={ /*TODO function that prints "the user wants a dinner for "+ the (only) function parameter + " guests" */}
/>
```

In SidebarView, **implement** the + and the - button **click** listeners to call `props.setGuests(param)` so that:

- Pressing the - button will print: "The user wants a dinner for 2 guests." (this will not get to 1 or less, **we fix that later**)
- Pressing the + button will print: "The user wants a dinner for 4 guests." (this will not get to 5 or more, we fix that later)

Hint: use props.**guests** in the click listeners! You may be tempted to *change* props.guests, that will not work. Never change the component props! Instead call `props.setGuests(param)` with the appropriate parameter.

Test also for `guests={4}` `guests={6}`

`guests={1}` should only let you press +

TW1.4 Vue *State* introduction

In vue.html <SCRIPT type="text/jsx">. Don't forget such a <script> for js/views/sidebarView.js !

```
// a Vue state-ful component, defined as Object, not Function
const VueSidebarLocalState= {  // JS object literal
    data(){ return {number:2 /* another object literal: */}; },
    render(){
       return <SidebarView guests={this.number}
              setGuests={/*TODO arrow function that assigns this.number from its only parameter*/}
               />;
            }
};
Vue.render(<VueSidebarLocalState />, …)
```

*Always use arrow functions in JSX (custom) event handlers that use **this**. E.g.* `setGuests`

You should be able to use the + and - buttons and see them take effect!

We want to connect the SidebarView to the DinnerModel. We do that in the next step

TW1.4 Vue Presenters (early version)

```
react.html
vue.html
js/
        DinnerModel.js
        views/
                searchView.js
                sidebarView.js
                summaryView.js
vuejs/
        sidebarPresenter.js
        summaryPresenter.js
```

Vue Presenters get as prop a reference to a DinnerModel object and use it to set the props of their View, including custom events.

Nothing to test yet, see next slide.

//vuejs/summaryPresenter.js. We program Presenters as Custom Components

```
function SummaryPresenter(props){
   return <SummaryView persons={props.model.numberOfGuests} />
}
```

//vuejs/sidebarPresenter.js

```
function SidebarPresenter(props){
   return <SidebarView guests={TODO }
                       setGuests= {/* TODO arrow function that calls
                           the props.model  setNumberOfGuests() */}
    />
}
```

Frequent mistake
setGuests={~~statements~~}

instead of
setGuests={param=>expr}

The first form is not a function! (custom) event handlers must be functions!

Re-visit all custom events written so far and identify param and expr. Assignments and function calls are expressions in JS!

TW1.4 App

```
react.html
vue.html
js/
        DinnerModel.js
        app.js
        views/
                searchView.js
                sidebarView.js
                summaryView.js
vuejs/
        sidebarPresenter.js
        summaryPresenter.js
```

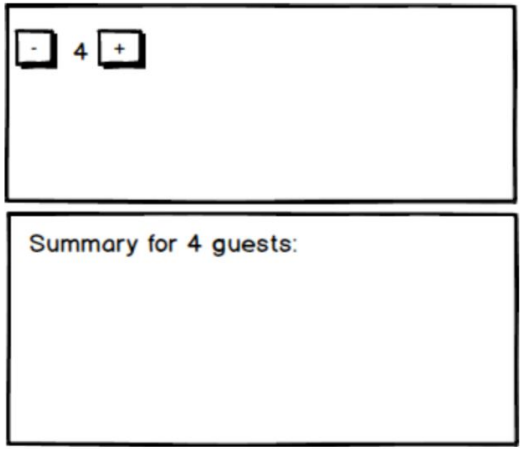
The App component is meant to graphically arrange the Presenter-View pairs.
It will be **common** for React and Vue.

//js/app.js

```
function RenderTest(){ console.log("Vue sub-component render test"); return false; }
function App(props){
   return  (
        <div>
             <SidebarPresenter model={props.model}  />
             <SummaryPresenter model={props.model}  />
             <RenderTest />
         </div>
    );
}
```

// Test in **vue.html** that the App is rendered properly. It will not be interactive yet!
// Don't forget the <script type=text/jsx> for app.js
// and <script> **(no need for type JSX!)** for DinnerModel.

```
const myModel= new DinnerModel();
Vue.render(<App model={myModel} />, ..)
```



TW1.4 Model as Vue top-level state

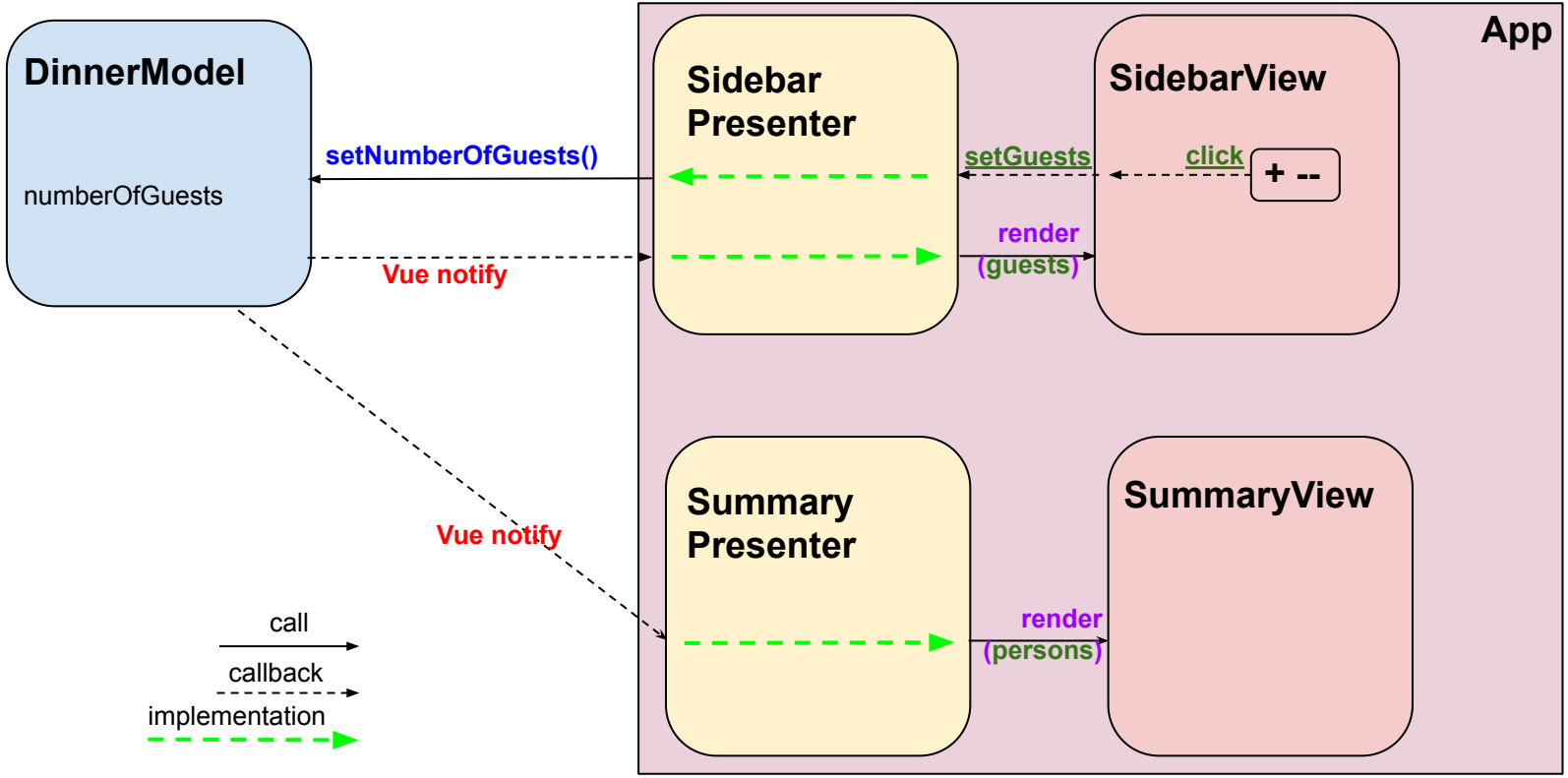
Instead of rendering App directly, we render a stateful component that has the Model as state. In **vue.html**

```
const TopLevelModel= {
   data(){ return  {model: new DinnerModel()}; },
   render(){  return <App model={/*TODO model from state*/} />; }
};
Vue.render(<TopLevelModel />, .. );
```

The app is now interactive. It updates when the Model changes because **Vue state check is deep:** if any of the state properties (this.model properties in this case) change, there will be a re-render of components that depend on that property.

Vue re-render is “selective”: RenderTest does not depend on any state (model in this case) property and is not re-rendered! Therefore **top-level state in Vue** does not lead to un-needed re-rendering (as you will see in TW1.5, it does in React).

TW1.4 MVP is now implemented in Vue for early Sidebar & Summary



TW1.5 React state introduction

In **react.html** we demonstrate component state by adding a <script>. Don't forget a <script> for js/views/sidebarView.js !

```
function ReactSidebarLocalState(){
    const numberState= React.useState(2); // an array with 2 elem. !
    const num=       numberState[0];     // a number, initially 2
    const setNumber= numberState[1];     // a function!
    return <SidebarView guests={num}
                        setGuests={x=> TODO call setNumber } />;
}
ReactDOM.render(<ReactSidebarLocalState />,...);
```

The first 3 lines can be simplified using JavaScript **array destructuring:**

```
const [num, setNumber]= React.useState(2);
```

TW1.5 React top-level state (and its issues)

```
// react.html
function RenderTest(){ console.log("React sub-component render test"); return false; }
function ReactTopLevelNumber(){
    const [num, setNumber]= React.useState(2);
    return (
      <div>
         <SidebarView guests={num} setGuests={x=> TODO call setNumber} />
         <SummaryView /*TODO*/ />
         <RenderTest />
      </div>
    );
}
ReactDOM.render(<ReactTopLevelNumber />, ..)
```

We cannot yet apply MVP using only React state, like we did with Vue, because, unlike Vue state check, React state check is ***shallow*** *(it only checks the model reference, not its properties). We will address this in TW3, using the Observer pattern.*

When a React component renders, all of its sub-components render unconditionally even if their props don't change (e.g. RenderTest has no props). React is "eager to re-render".

End of TW1

Congratulations! Commit to git! Push!

We will test whether **react.html** and **vue.html** display SidebarView and SummaryView and the two views are kept consistent with each other on user interaction.

Do not forget to submit the TW1 Canvas assignment!

Tutorial Week 2

Promises, CSS and finish Rendering

You need a lab partner. Join a TW2_TW3 group

TW2.1 setting up a Web API connection

You do not connect directly to the Spoonacular API because the calls to the API cost money, and we want to block e.g. API calls in infinite loops. The course proxy forwards your request to spoonacular. It also does caching so you get responses faster. It also calls for undefined parameters (wrongly) passed to the API.

We will connect to the Spoonacular API via an in-between server at KTH (called **proxy**)

We will first set up an API configuration file, **js/apiConfig.js**

```
const BASE_URL="https://brfenergi.se/iprog/group/NN/";   // the DH2642 proxy server
const API_KEY="3d2a031b4cmsh5cd4e7b939ada54p19f679jsn9a775627d767";
```

NN is your TW2_TW3 group number! Join a group ASAP if you haven't done so already! **TW2 and TW3 entail more work**, very suitable for splitting with a colleague!

Configuration files that contain keys like **js/apiConfig.js** must never get into git repositories. Create a **.gitignore** file in the project root folder (where **vue.html** and **react.html** sit). The .gitignore file contains only one line:

```
**/*Config.js*
```

Load the config file, and the new dishSource.js file in your React and Vue HTML files. No need to parse them with Babel! (no type="text/jsx")

```
<script src="js/apiConfig.js"></script>
<script src="js/dishSource.js"></script>
```

TW2.1 Basic API connection

*Check the Developer Tools **Network** tab to see your API calls being made. Explore the Request and Response tabs!*

```
//js/dishSource.js
const  DishSource={   // JS object creation literal
   apiCall(params) {
       return fetch(BASE_URL+params, {
             "method": "GET",
             "headers": {
               'X-Mashape-Key' : API_KEY,
               "x-rapidapi-host": "spoonacular-recipe-food-nutrition-v1.p.rapidapi.com",
          }
       })
       // from HTTP headers to HTTP response data
       .then(response => response.json())   ;
   }
   ,   // comma between object entries
   searchDishes(params){ return DishSource.apiCall(/* will do later */); }
   ,
   getDishDetails(id){ return DishSource.apiCall(/* will do later */); }
};
```

After you loaded your HTML, **test the API connection** at the **Console:**

```
DishSource.apiCall("/recipes/quickAnswer?"+new URLSearchParams({q:"How much vitamin C is in an apple?"}))
     .then(data=> console.log(data))
```

`/recipes/quickAnswer` is called an **API Endpoint.** For searchDishes and getDishDetails you will need to identify the endpoint by reading the API documentation.

TW2.1 Treat API errors (HTTP response code)

The previous slide assumes that the HTTP response code is 200 (OK). Detect the case when it is not OK, **by adding a then()** that checks the response and throws an error:

```
apiCall(params) {
   return fetch(/* as before */)
    // check HTTP response:
   .then(response=>{ /*TODO check response and throw an error if not OK (compose error msg from
                                                                                  response.statusText),
                        Otherwise if response contains 200/OK just return response */ })
   // from HTTP headers to HTTP response data:
   .then(response => response.json())   ;
}
```

Test in the **Console** like below. We are accessing a wrong endpoint so the HTTP response code will not be 200 but 404 (HTTP response code for “not found”)

```
DishSource.apiCall("/BlablaWrongEndpoint")
     .then(data=> console.log(data)).catch(error=> console.log("there was a problem", error))
```

This should work as before!

```
DishSource.apiCall("/recipes/quickAnswer?"+new URLSearchParams({q:"How much vitamin C is in an orange?"}))
     .then(data=> console.log(data)).catch(error=> console.log("there was a problem", error))
```

TW2.1 Search dishes

Browse the Spoonacular API documentation and find the **Search Recipes** API endpoint. Choose JavaScript and fetch() at the right to see how a search call is done.

The endpoint accepts many parameters but we will only use **type** and **query**. We will assume that searchDishes argument **params** is an Object containing properties named like the API parameters:

- **type**: the dish type (main course, dessert, etc)
- **query**: free text to be searched in the dish name, description, etc.

Example: `{type:"main course", query:"chicken"}  // object literal. Send {} or new Object() for "no parameters"`

In the API documentation you can also see that the Search API endpoint expects parameters to be sent via a **query string,** so you can use **URLSearchParams** for that. Implement in `DishSource`

```
searchDishes(params){ return DishSource.apiCall(/*TODO search endpoint + query string made from params */) }
```

If you test with

```
DishSource.searchDishes({type:"main course"}).then(console.log).catch(console.error)
```

you will see that the endpoint returns an Object, not an Array of dishes. One of the properties of this Object is an Array. We would like searchDishes to return a promise for that Array. You can add a **then**() to return the array from the returned data

```
searchDishes(params){ return DishSource.apiCall(/* as before*/)
                              .then(data=> /* return the property of data that is an array of results */); }
```

Test again, you should see an array being printed:

```
DishSource.searchDishes({type:"main course"}).then(console.log).catch(console.error)
```

TW2.1 Dish details

As you can see, the dish information in the Search results is minimalistic: dish name, dish **ID**, dish image. To get the full details of a dish, pass the **ID** to an API that you find an endpoint that gives detailed recipe info in the API documentation . Choose JavaScript and fetch() at the right to see how a search call is done.

Implement in `DishSource`

```
getDishDetails(id){ return DishSource.apiCall(/*TODO */) }
```

Test at the Console

```
DishSource.getDishDetails(547775).then(console.log).catch(console.error)
```

This *non-existing* dish ID should print in console.error (red)! If that doesn't happen, review your "treat HTTP error" code.

```
DishSource.getDishDetails(321654).then(console.log).catch(console.error)
```

Find more interesting dish IDs in search results and check how to prepare them!

TW2.2 Rendering and CSS

You are rendering and testing Views in 2.2. Do not expect the interaction to work fully since you have no Presenters to connect the Views to a Model!

In TW1 we only had one integer to render. Now we have much more data. So we will use that to practice and extend our HTML and CSS knowledge.

You may find that some of the views require quite some detail work, you may want to **split the work** with your TW2_TW3 partner! There are 4 main views (Search results, dish Details, Sidebar, Summary), you can split them 2+2. This can include the **custom events** raised by the respective view.

For starters, we will focus on HTML/CSS and **not** on the user experience of rendering asynchronous data, which we will address in TW2.5

Most our testing (in HTML) will get the data from a **promise** and render a View when the promise resolves (fulfills):

```
promise.then(data=>VueOrReact.render(<TestedView someProp={data}), document.getElementById("app")))

// illustration purposes only, do not try to make this code work yet!
DishSource.searchDishes({..}).then(data=>VueOrReact.render(<SearchResultsView searchResults={data}), ..))
DishSource.getDishDetails(ID).then(data=>VueOrReact.render(<DetailsView dish={data}), ..))
```

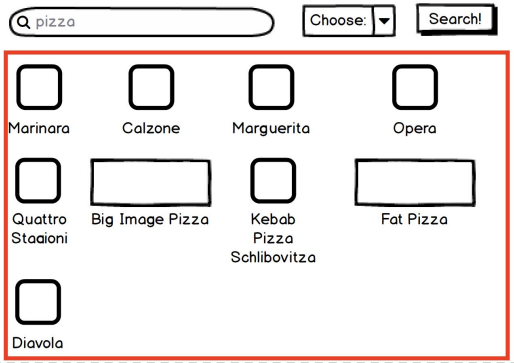
For Views that need several dishes (Sidebar, Summary) we use **Promise.all** which waits for a multiple promises to resolve (fulfill) and puts their result in an array (**values** below)

```
Promise.all( [DishSource.getDishDetails(AA),  DishSource.getDishDetails(BB)])
              .then(values=> ReactOrVue.render(<TestedView someProp={values}))
```

These are just test code examples. Concrete code in coming slides. For now, make sure that you understand how promises and rendering are used in these tests!

react.html
vue.html
style.css
js/
views/
searchView.js
sidebarView.js
summaryView.js

TW2.2 SearchResultsView. Basic CSS



DOM tree:

DIV
- SPAN class=`searchResult` ***(repeated)***
  - IMG src=*imageURL* height
  - DIV
    - *dish name*

***Note**: React accepts both **class**= and className=, though it may show a warning about class=. Vue accepts only **class**=, so we will use that in the lab! In projects, use class with Vue and className with React!*

In the HTML HEAD add:

```
<link rel="stylesheet" href="style.css">
<meta name="viewport" content="width=device-width,initial-scale=1">
```

Test in HTML

```
DishSource.searchDishes({query:"pizza"}).then(results=>
        VueOrReact.render(<SearchResultsView searchResults={results} />, ..));
```

// js/views/searchView.js (existing file, add to it)
function SearchResultsView(props)

Assume that there is a prop called **searchResults**, an array of Objects. Use **array rendering** to render a SPAN for each search result. Use the dish ID as **array rendering key** as it is guaranteed to be unique for each dish.

Find the image URL in the data by examining it in the Developer Tools.

Load the **dish image** (IMG src attribute) from `"https://spoonacular.com/recipeImages/"` **concatenated** with a property of the individual search result that contains an image file name (find it by examining the result, e.g. after a breakpoint

Make a CSS style file **style.css** in the project folder

Find CSS layout information in the **slide notes**

TW2.2 SearchResultsView custom event

If the user clicks anywhere on a search result SPAN (use **event bubbling**) the dishChosen custom event must be fired, with the dish ID as a parameter.

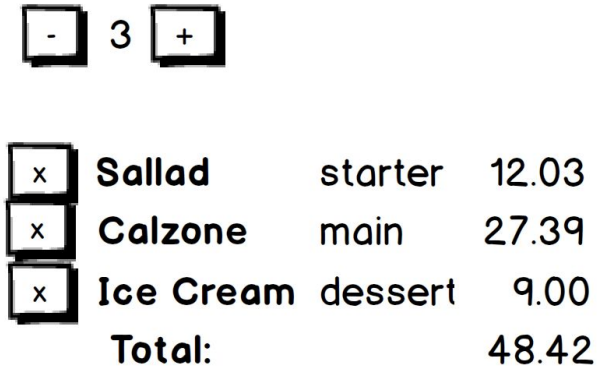
```
onClick={e=>props.dishChosen(/*TODO pass the dish id*/)}
```

Test in HTML:

```
DishSource.searchDishes({query:"pasta"}).then(results=>
          VueOrReact.render(<SearchResultsView searchResults={results}
                                               dishChosen={id=> console.log("The user chose dish ", id) }
                            />
                            , ..)
);
```

TW2.2 SidebarView (complete)

react.html
vue.html
js/
views/
searchView.js
sidebarView.js
summaryView.js



DIV
- BUTTON
  - --
- *numberOfGuests*
- BUTTON
  - +
- TABLE
  - TR *(repeated)*
    - TD
      - BUTTON
        - x
    - TD
      - *dishName*
    - TD
      - *dishType*
    - TD
      - *dishPrice*
  - TR
    - TD (empty)
    - TD
      - Total:
    - TD (empty)
    - TD
      - *totalPrice*

Test in HTML:

```
Promise.all(
     [DishSource.getDishDetails(AA),
     DishSource.getDishDetails(BB),
     DishSource.getDishDetails(CC)]
).then(values=> ReactOrVue.render(
    <SidebarView guests={5}
                 dishes={values} />
, ..))
```

Add the menu as the **dishes** prop. For now, you get it as example data. Use **Array rendering** to render one Table Row (TR) for each dish.

Multiply all dish prices with the number of guests. Calculate the total price (see hints in **notes**)

Sort by dish type (sort hints in **notes**). Dishes that have "starter" in their type will always be listed before dishes that have "main course" which are always listed before dishes that have "dessert" in their type.

To check whether the prices are multiplied correctly with the guest number, change **guests** in the testing code.

See *values for AA, BB, CC* and **JavaScript hints** for sorting, totals, alignment **in the notes.**

TW2.2 SidebarView custom event: removeDish

A click on an **x** button means that the user wants to remove the dish from the menu, so we send it up as the removeDish **custom event.** Test in HTML:

```
Promise.all(
    [ DishSource.getDishDetails(AA),
     DishSource.getDishDetails(BB),
     DishSource.getDishDetails(CC)]
).then(values=> ReactOrVue.render(
    <SidebarView guests={5}
                         dishes={values}
                         removeDish={id=> console.log("user wants to remove dish with ID ", id)} />
      , .. )
)
```

TW2.2 SidebarView custom event: dishChoice

Make the *dish names* in the SidebarView render **hyperlinks** (text). A click on a dish link means that the user wants to see details about that dish, which they will do in another view. We signal this with the dishChoice **custom event.**

In order for the link not navigate to another page (which is the *default browser behavior* for link clicks), you need to call the event method **preventDefault**

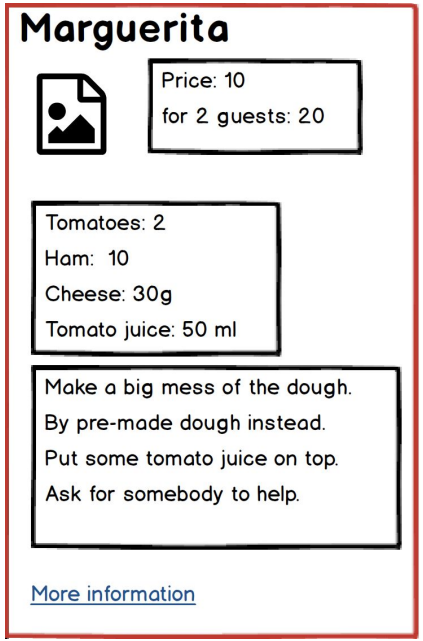
```
<a href="" onClick={e=>{ /*TODO call the method preventDefault of the event! */;
                         /*TODO fire the dishChoice custom event  with the dish id as parameter */;} }
>{/*TODO dish name as before */}</a>
```

Test in HTML:

```
Promise.all(
    [ DishSource.getDishDetails(AA),
     DishSource.getDishDetails(BB),
     DishSource.getDishDetails(CC)]
).then(values=> ReactOrVue.render(
    <SidebarView guests={5}
                         dishes={values}
                         removeDish={id=> console.log("user wants to remove dish with ID ", id)}
                         dishChoice={id=> console.log("user wants details of dish with ID ", id)} />
      , .. )
)
```

TW2.2 DetailsView and custom event

```
react.html
vue.html
style.css
js/
    views/
        detailsView.js
        searchView.js
        sidebarView.js
        summaryView.js
```



Add to menu! Cancel

This is not a specification!
Use your imagination
within your current HTML/CSS skills!

Test in HTML

```
DishSource.getDishDetails(547775).then(details=>
        VueOrReact.render(<DetailsView dish={details} people={3}  isDishInMenu={false}
                                dishAdded={()=>console.log("User wants to add this dish! ", details)}
 />, ..));
```

```
// js/views/detailsView.js
function DetailsView(props)
```

You are free to render as you wish so you do not get a DOM tree specification here. Create the CSS classes you need.

If you are unsure about how to lay out, just add a DIV for each section of the view (price, ingredients, recipe, etc). All the sections should be children of the root DetailsView DIV.

Clicking “Add to Menu” must trigger the dishAdded custom event with no parameters

If the isDishInMenu prop is **truthy**, the “Add to Menu” button will be disabled. That is, we don’t offer to add a dish twice. **Note:** this will not be interactive yet (pressing the button will not disable it!) because you need to add the dish in the menu (in the Model) first. DetailsView just renders the button enabled or disabled depending on the isDishInMenu prop.

See in the **notes** the minimum data required in this View

TW2.2 SummaryView (complete)

```
react.html
vue.html
js/
    dinnerModel.js
    views/
        detailsView.js
        searchView.js
        sidebarView.js
        summaryView.js
```

Dinner for 3 guests:

Ingredient	Aisle	Quantity
Cream	Dairy	300ml
Milk	Dairy	150ml
Salami Milano	Meat	100g
Pizza dough	Refrigerated	1

Test in HTML:

```
Promise.all(
     [DishSource.getDishDetails(AA),
     DishSource.getDishDetails(BB),
     DishSource.getDishDetails(CC)]
).then(values=> ReactOrVue.render(
    <SummaryView persons={5}
                 ingredients={getIngredients(values)} />
, ..))
```

An ingredient must show up exactly once. Quantities must be added up between various dishes that use a certain ingredient.

Quantity must be multiplied by the number of guests.

Sort by aisle and (for the same aisle) ingredient name.

See JavaScript help in **notes** for aggregating ingredients (**`getIngredients`**) . Make sure to keep such complex data calculations (like amounts, etc) outside the SummaryView. The job of the view is solely to display the results.

You can use HTML TABLE which you already learned in SidebarView. Use **`toFixed`** (defined earlier) to make numbers align nicely for the user. Measurement units (ml, g,..) also make a lot of sense for users who want to cook.

See the **notes** for **JavaScript help** on sorting by aisle, ingredient etc.

TW2.3 Dishes in the DinnerModel

Add the following to the DinnerModel that you started in TW1. Add more parameters to the DinnerModel **constructor**

`constructor(guests=2, dishes=[], currentDish=null)`

- a **`dishes`** property, set in the constructor from a constructor parameter with the default value **[]**
- **`addToMenu`**`(dish)` sets **`dishes`** to a new array (using e.g. **spread syntax**) with `dish` as its last member
- **`removeFromMenu(dishData)`** which uses **Array.filter** to remove from dishes the dish with the ID `dishData.id`
- a **`currentDish`** **property,** set to a constructor parameter with default value **null**
- **`setCurrentDish(id)`**

Test at the **Console.** A model with 5 guests and 2 dishes (*AA* and *CC*) should be printed.

```
const myModel=new DinnerModel(5);
Promise.all(
     [DishSource.getDishDetails(AA),
     DishSource.getDishDetails(BB),
     DishSource.getDishDetails(CC)]
).then(values=> values.map(d=>myModel.addToMenu(d)))
 .then(()=> { myModel.removeFromMenu({id:BB}); console.log(myModel); })
```

TW2.3 Dishes in Vue Sidebar and Summary presenters

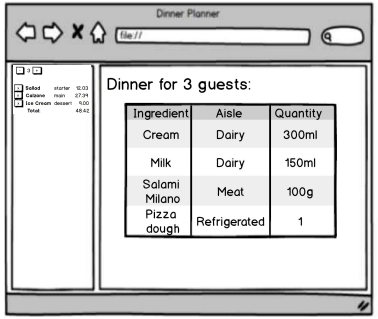
In the Vue SummaryPresenter and SidebarPresenter, set the correct props for the views using the model.dishes property. Set the Sidebar custom event handlers (dishChoice, removeDish) to change the model (setCurrentDish() and removeFromMenu()).

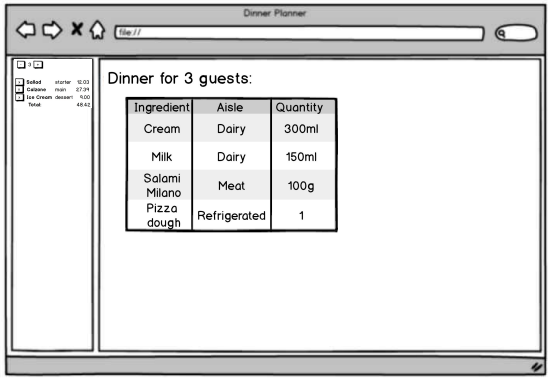
Test **vue.html:**

- should display the dishes and ingredients
- the + and - buttons should work as in TW1, but in addition they should update the dish prices and ingredient quantities in the UI
- pressing **x** in the sidebar should remove the dish in the model and in the UI! (removeDish custom event)
- clicking on a dish in the sidebar (dishChoice) should change `myModel.currentDish`. Check that with a breakpoint in DinnerModel setCurrentDish(). *You will need this in TW3*, so please test it!

```
const myModel=new DinnerModel(3); const TopLevelModel= /* as before */ ;
Promise.all(
     [DishSource.getDishDetails(AA),
     DishSource.getDishDetails(BB),
     DishSource.getDishDetails(CC)]
).then(function(values){
       values.map(d=>myModel.addToMenu(d));   // add the dishes to the model
       Vue.render(<TopLevelModel />,..);
      });
```

TW2.4 CSS Layout





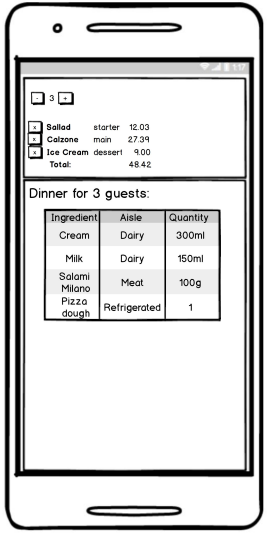
Make the sidebar behave like a sidebar, i.e. stay of constant width regardless of the window width. Both Sidebar and Summary height should increase if the window height increases. Only Summary should be affected by the width changes (you can test this easily if you just adjust the browser page width)

You can use CSS **flexbox** or **grid** to achieve this. See notes below (and lecture material) for flexbox hints. The code snippets assume flexbox, for grid you will need to adapt.

```
// js/app.js
const App= (props)=>
<div class="flexParent">
  <div class="sidebar debug"    ><SidebarPresenter ../></div>
  <div class="mainContent debug"><SummaryPresenter .. /></div>
</div>;

style.css:
.debug{     border-style: solid; }
.flexParent{ display:flex;   … TODO … }
.sidebar{ … TODO … }
.mainContent{  … TODO … }
```

*Test with **vue.html**! App is framework independent so this will later work in React!*

TW2.4 Responsive layout



On narrow screens the sidebar will be shown above the summary (or other content). Our app will thus be *responsive,* depending on what kind of device it is run on

If the screen is very narrow, we would like DIVs to be laid out one under the other (as they were before this step) rather than side by side. That basically means that we don't want the .flexParent class to exist for narrow screens.

Use the Developer Tools to test limited screens, e.g. a mobile configuration. You can also test on Desktop by drastically reducing the width of the browser window

We use a **media query** that only defines the flexParent class if the screen has a certain minimum width (say 250 pixels). Your job is now to figure out how to set the media query and define the *flexParent* within.

```
@media (… TODO … ){
      .flexParent{ … }
}
```

*Test with **vue.html**! App is framework independent so this will later work in React!*

TW2.5 Promise data rendering

When we work with promises in GUIs, we want to render:

1. Nothing if there is no promise yet
2. A “loading image” if the promise was set but it is not resolved yet
3. The error text, if the promise failed (**catch**)
4. The results, if the promise is resolved (**then**)

To test the transitions between these states, we will use component **State**. But before that we will implement a *function* to be used for **conditional rendering** with a View

```
promiseNoData(promise, data, error) || <View />
```

*See **JavaScript && and || evaluation** (lecture slides)*

In cases 1-3, promiseNoData will return **truthy**, so the View is **never rendered**.
In case 4, promiseNoData will return **falsy**, so <View/> is returned.
The View props usually make use of the promise result (data) but we won’t use that for **promiseNoData** implementation and tests (next slide)

TW2.5 Implement and test promiseNoData

```
// new file js/views/promiseNoData.js
function promiseNoData(promise, data, error)
```

*No **then** or **catch** in promiseNoData! Simply check whether the 3 parameters are **truthy** or **falsy** (not just undefined!) and return the indicated results*

Add promiseNoData.js to the HTML, with type **text/jsx**. Implement the **4 cases:**

`VueOrReact.render(promiseNoData(null),..)` case (1), will render: `<span>no data</span>`

`VueOrReact.render(promiseNoData("a promise", undefined, null),..)` case (2) will render an image like http://www.csc.kth.se/~cristi/loading.gif

`VueOrReact.render(promiseNoData("promise", null, "some error"),..)` (3) SPAN some error

`VueOrReact.render(promiseNoData("promise", "some data", null) || <div>Hello world</div>, ..)`

In case (4) promiseNoData must return **falsy**, so this will render: `<div>Hello world</div>`.

TW 2.5 Promise rendering with React state

In **react.html** we want to change the React state as the promise makes progress.

```
const searchPromise= DishSource.searchDishes({type:"main course", query:"pasta" });

function SearchTest(){
   const [data, setData]=React.useState(null);
   const [error, setError]=React.useState(null);
   React.useEffect(function(){
       searchPromise.then(dt=> /* TODO set dt in the component state! */)
                    .catch(er=> /*TODO set er in the component state! */)
   }, []);
   return promiseNoData(TODO) || <SearchResultsView searchResults={TODO} dishChosen={console.log}/>
}

const detailsPromise= DishSource.getDishDetails(523145);
// TODO: implement DetailsTest in a similar fashion!
// Set the DetailsView dish prop from the promise, 4 people, isDishInMenu true, dishAdded console.log
ReactDOM.render(<div><SearchTest /><DetailsTest/></div>, ..)
```

`React.useEffect(callback, [])` *executes the callback exactly once, after the first render.*
Find more in the component lifecycle lecture material.
Vue works in a similar way, see slide notes.

End of TW2

Congratulations! Commit! Push!

vue.html will show Sidebar and Summary laid out as per TW2.4 . Changing number of guests and removing dishes should work (per TW2.3).
Please demo TW2.4 window resizes (flex/grid) and narrow window (responsiveness, media query) to the TA.

react.html will show SearchResultsView and DetailsView with loading (“spinner”) images while data is being fetched (as per TW2.5)

Do not forget to submit the TW2 Canvas assignment!

Tutorial Week 3

Observer, subscriptions, navigation, persistence (Firebase)
Choose one framework to do TW3 in (React or Vue)

TW3.1 Observer: add the following methods to DinnerModel

Set **`this.observers`** to an empty array [] *first thing* in the **constructor**

addObserver(*callback*)

add ***callback*** to this.observers. See e.g. in lectures ***immutable* way to append x to arr**

removeObserver(*callback*)

remove ***callback*** from the **`observers`** array, e.g. `this.observers= this.observers.filter(..)`

notifyObservers()

Call all the callbacks in the array, e.g. `this.observers.forEach(cb=>/* call cb */)`
If an error occurs in an observer, all subsequent observers will lose the notification. Solution: try{ /*call cb */ }catch(..){..}
If an observer takes too long to address the notification (cb takes long), the other observers will be delayed! See **notes**

Call `this.notifyObservers()` at the end of each setter method (optional: call only **if the method changes the model**)

- setNumberOfGuests(nr) *should* (not obligatory) skip calling notifyObservers() if nr is the same as before
- addToMenu(d), *should* skip making any change (and not call notifyObservers()) if the dish is already in the menu
- removeFromMenu(d), *should* skip calingl notifyObservers() if the dish is not in the menu
- setCurrentDish(id) *should* skip calling notifyObservers() if id is the same as before (we will enforce this later)

TW3.1 Test Model as Observable

Load **vue.html** or **react.html** containing `<script>const myModel=new DinnerModel();</script>`

Test in the **Console**:

```
myModel.addObserver(()=>console.log(myModel));
const errorLogger= ()=>console.error(myModel);
myModel.addObserver(errorLogger);
```

//Both observers will print the values below, one in black, one in red

```
myModel.setNumberOfGuests(5)   // { numberOfGuests:5, dishes:[], currentDish:null, observers:[2 elem]}

DishSource.getDishDetails(547775).then(d=>myModel.addToMenu(d))
                                   // { numberOfGuests:5, dishes:[1 elem], currentDish:null, observers:..}
myModel.setNumberOfGuests(2)   // { numberOfGuests:2, dishes:[1 elem], currentDish:null, observers:..}
myModel.setCurrentDish(547775)   // { numberOfGuests:2, dishes:[1 elem], currentDish:547775, observers:..}
DishSource.getDishDetails(787321).then(d=>myModel.addToMenu(d))
                                   // { numberOfGuests:2, dishes:[2 elem], currentDish:547775, observers:..}
myModel.removeObserver(errorLogger)
```

//Only in black from now on:

```
myModel.removeFromMenu({id:787321})   // { numberOfGuests:2, dishes:[1 elem], currentDish:547775, observers:[1 elem]}
myModel.setCurrentDish(null)   // { numberOfGuests:2, dishes:[1 elem], currentDish:null, observers:[1 elem]}
```

TW3.1 React SidebarPresenter, SummaryPresenter as Observers

If you want to work with Vue in TW3 this is optional but easy and strongly recommended

reactjs/sidebarPresenter.js reactjs/summaryPresenter.js, for both add a JSX **<script> in react.html**!

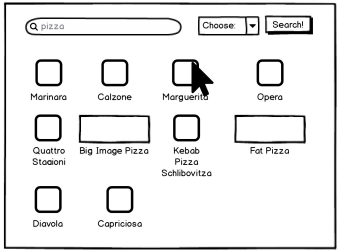
Both Presenters should receive a single prop (`model`) (like in Vue)

- In a React **effect** (or Observer **custom hook**) see **lecture notes on React subscribe/unsubscribe** or **custom hooks.**
  - **add an observer** to **props.model**
  - In the observer callback, set the model **`numberOfGuests`** and **`dishes`** in component state
- Based on number and the dishes from state, pass the needed props to SidebarView and SummaryView respectively
- Implement the custom event handlers for SidebarView **`setGuests`**, **`removeDish`**, **`dishChoice`** by calling appropriate methods of props.model (see similar Vue code in TW2.4)

Test in **react.html**! Load app.js as JSX. Remember that App is framework-independent so this should work like it did in Vue in TW2.4

```
const myModel=new DinnerModel(3);
Promise.all(
     [DishSource.getDishDetails(AA),
     DishSource.getDishDetails(BB),
     DishSource.getDishDetails(CC)]
).then(function(values){
       values.map(d=>myModel.addToMenu(d));   // add the dishes to the model
       ReactDOM.render(<App model={myModel}/>,..);
      });
```

This is just an intro slide.
Code guidance in next slide.



TW3.2 SearchPresenter goals and approach

SearchPresenter will display (and treat events from) both the SearchFormView (TW1.2, TW1.3) and SearchResultsView (TW2.2). It is a stateful component, in both Vue and React, see lectures for examples.

Initially the SearchResultsView shows some random search results to suggest to the user “searches can be performed here”. For that we set a **promise** in SearchPresenter **state** to DishSource.searchDishes({})

When the user presses **Search** in the SearchFormView, SearchPresenter changes **promise** in its state according to the *query* and *type* entered in the SearchFormView.

Changing **promise** in state will lead to a re-render, so the SearchResultsView will show the new results.

SearchPresenter does not depend on the Model so it does not need to add itself as an observer (in React). However, SearchPresenter does need the model as a **prop**, to be able to call the DinnerModel **setCurrentDish(id)** when the user clicks on a dish in the search results (dishChosen event fired by SearchResultsView, see TW2.2)

TW3.2 Search Presenter

In the framework of your choice (`reactjs/` or `vuejs/`) `searchPresenter.js`

- Component state properties: **`promise`, `data`, `error`**, all initially null
  - remember Vue stateful components `const SearchPresenter={ data(){ return ..} , props:["model"], ..}`
  - the above also declares a model prop, refer to it as **`this.model`** (you need it on next slide)
- **At component creation** (see component lifecycle lecture notes), set **`promise`** in state: a DishSource search with empty parameters {}
- **At `promise` change,** reset data and error from promise results
  - (see lecture: **execute when state changed (derived state), promise in state**)
- **Initial** render (not interactive yet). This presenter renders a little UI (div), see **notes**.

```
<div>
    <SearchFormView options={/* same as in TW1.2 */} />
    {promiseNoData(/* 3 params from component state */) ||
        <SearchResultsView searchResults={ /* from component state */}/>}
</div>
```

- Replace `<SummaryPresenter.. />` in **app.js** with `<SearchPresenter model={props.model} />` Keep the sidebar at the left!
- **Test your HTML!** A loading image should be followed by some initial search results

TW3.2 SearchPresenter events

- Add two more **state** properties: `searchQuery` and `searchType`, initial value ""
- Make the search form *drive* the **promise** in state.
  - implement `SearchFormView` event handlers `onText` to set `searchQuery` in state and `onDishType` to set `searchType` in state (see TW1.3)
  - Implement `SearchFormView` event handler `onSearch` to set `promise` in state based on `searchQuery` and `searchType` (see TW1.3)
    - This will trigger the **At promise change** mechanism from the previous slide
- When done, **test the HTML** again! The search results should update when pressing Search!
- Implement `SearchResultsView` `dishChosen` event handler (see TW2.2) to call **setCurrentDish(id)** on the **model** prop
  - Test by adding a breakpoint in the model and verify that the new dish id is set when clicking on a dish in `SearchResultsView`

TW3.2 DetailsPresenter goals and approach

DetailsPresenter needs to display the details (data) of the DinnerModel `currentDish`. The DetailsView (see TW2.2) also need the *number of guests* and the Presenter needs the menu (*dishes*) to set isDishInMenu

To achieve this, DetailsPresenter must get data from two sources

1. The Model
   a. React: use Observer (or Observer custom hook)
   b. In Vue, the component can be functional, like Vue SidebarPresenter and SummaryPresenter, since it has no state, just `props.model`
2. The **promise** used to retrieve the current dish data (which depends on the Model currentDish)

We thus have a currentDish-> promise -> currentDishData dependency (derived state). We will treat the dependency in the Model's setCurrentDish().

TW3.2 currentDish ---> details derived Application State

```
// DinnerModel
setCurrentDish(id){
     /* TODO if currentDish doesn't really change (use ===),
            we don't want to make a network access, so return */;
     /* TODO set the model current dish property to the new value */;

     this.currentDishDetails= null;  this.currentDishError= null;
     /* TODO notify observers, because current dish, details, error changed! */

     if(/* check that currentDish is truthy for getDishDetails() to make sense */)
         DishSource.getDishDetails(/* TODO */)
          .then(/* if currentDish is still id, set currentDishDetails
                   from promise results and notify observers */)
          .catch(/* if currentDish is still id, set currentDishError
                   from promise error and notify observers */ )
}
```

currentDish may have ch
*by the time **then** or **catc***
invokes its callback, hen
***id** checks.*

Test: `myModel.addObserver(()=> console.log(myModel));  myModel.setCurrentDish(758118);`
Must print `currentDishDetails:null` first! Then the model will notify again when the promise resolves.

TW3.2 DetailsPresenter

In the framework of your choice (`reactjs/` or `vuejs/`) `detailsPresenter.js`

- use DinnerModel properties currentDish, currentDishDetails, currentDishError, nr. guests, dishes (React: Observer/custom hooks, `Vue stateless comp.: props.model.currentDish`)
- **Initial** render (not interactive yet):

```
promiseNoData(/* currentDish instead of promise!*/, TODO, TODO)
  ||
  <DetailsView  isDishInMenu={ /*model dishes */.find(d=>/*id same as currentDish*/)}
                people={/* from model */}
                dish={/* details from model, see previous slide  */ }
   />
```

- in **app.js** render sidebar at the left, and in the right-hand side DIV (mainContent), render:
  `<SearchPresenter model={..}/>`
  `<DetailsPresenter model={..} />`

- **Test your HTML!** When clicking on a search result, the dish details should show up below the search results (after a loading image!)
  - This works because `currentDish` is changed in the DinnerModel by the SearchPresenter

TW3.2 DetailsPresenter event

Implement the event handler for DetailsView **`dishAdded`** (see TW2.2) to call the DinnerModel **addToMenu**(dishDetails).

Test in HTML (**Remove all dish Test data, Promise.all** etc).

vue.html:

```
const myModel=new DinnerModel();
const TopLevelModel= /* as before */;
Vue.render(<TopLevelModel/>,..)
```

react.html:

```
const myModel=new DinnerModel();
ReactDOM.render(<App model={myModel} />,..)
```

The data flows of the Dinner Planner are now complete, you should be able to search, add and remove dishes, change number of guests, etc. Summary is not visible, we will fix that in the next step (Navigation)

TW3.3 Navigation

We now want to show **Search**, **Details** and **Summary** ***alternatively*** depending on the application state.

- **Search** shown initially
- **Details** shown when the user clicks on a **search result** in Search, or on a **dish link** in Sidebar
- **Search** shown when the user presses **Cancel** or **Add to Menu** in Details
- **Summary** shown when the user presses **Summary** in Search
- **Search** shown again when the user presses **Back to Search** in Summary

You (or the user you evaluate with) may suggest other kinds of navigation.

Navigation is typically achieved with framework-specific components called **routers**. You can use them for your project. The goal at the lab is that you understand how routers work under the hood, using the `window.location.hash` browser DOM property. It is the #string shown at the end of the URL in the browser. We implement this in 3 steps:

1. If `window.location.hash` is anything else than `"#summary", "#search", "#details",` set it to `"#search"` aka the *default route*
2. implement a Presenter called Show that only presents its *nested components* if `window.location.hash` has a certain value. `<Show hash="#summary"><SummaryPresenter model={props.model} /></Show>`
3. Set `window.location.hash` when certain user events happen.

TW3.3 Default route

We do this in **js/app.js** as setting the default route is application-wide code.

```
function defaultRoute(){
    if(/* route is unknown (see below) */) window.location.hash="#search";
}
defaultRoute(); // when the application loads, set the default route!
```

To check whether the route is **known** you can e.g. use the **find** higher-order function:

```
["#search", "#summary", "#details"]
      .find((knownRoute)=> /* check whether window.location.hash matches knownRoute*/)
```

Test in your HTML, by accessing

- reactOrVue.html should turn into reactOrVue.html#search
- reactOrVue.html#detailz should turn into reactOrVue.html#search
- reactOrVue.html#random should turn into reactOrVue.html#search
- reactOrVue.html#details should stay as is
- reactOrVue.html#summary should stay as is
- reactOrVue.html#search should stay as is

If the route is accidentally set to something unknown, we fix it in **app.js** by adding `defaultRoute` as **hashchange** event listener!

```
window.addEventListener("hashchange", /*TODO */);  // test at Console by setting windows.location.hash
```

TW3.3 Show presenter implementation

Show is our first component that has nested components. Frameworks handle nested components differently.

In the framework of your choice (`reactjs/` or `vuejs/`) `showPresenter.js`

In Vue (stateful component!), make sure you declare a prop: `props:["hash"],`

Use lecture **subscribe at creation, unsubscribe at teardown** to subscribe to the "hashchange" window event. That will set a copy of `window.location.hash` in component state (we call it **hashState**)

Render a DIV containing the *nested components* `<div class={expr below} >{ nested components }</div>`

- if **hashState** is not equal to the **hash prop** the DIV class will be "hidden" (define hidden in **style.css** as **display:none**)
- if **hashState** is equal to the **hash prop**, the DIV class will be "".

The *nested components* are available as follows:

- Vue: `this.$slots.default()  // note: this will show a warning once for each <Show>`
- React: `props.children (the children prop)`

The right-hand side of the **App** (mainContent) will render **Search, Details, Summary** nested inside a Show *each*:

```
<Show hash="#search"><SearchPresenter model={props.model} /></Show>   <Show hash="#details">..</Show>  ..
```

Now you can **test** `reactOrVue.html`, `reactOrVue.html#search`, `reactOrVue.html#details`, `..#summary`

Or at the **console**: `window.location.hash="details" // # is added automatically`

TW3.3 Triggering navigation

We will trigger navigation (change **`window.location.hash`**) directly from the Views.

- See architectural remarks in the **notes**

JS syntax note: many of the navigation triggers add code to already existing event listeners. In such cases you may need to add **function body** curly braces: `event=>{ /* previous code */ ; window.location.hash=..; }`

If any navigation button is missing from your View, add it where you think is more suitable. See (in 3.4) if users find it. Styling like float:right could be useful if you want the button at the top right of the View.

- SearchFormView **Summary** button, `onClick={e=> window.location.hash="#summary" }`
- SummaryView **Back to Search** button, implement onClick!
- SearchResultsView dish click: `onClick={e=>{ /* as before */; window.location.hash="#details";} }`
- SidebarView dish link click: `onClick={e=>{ /* as before */; /* -> #details */ } }`
  - Alternatively, you can simply put the route as the **href** attribute `<a href="#details" onClick={/*as before */} >..</a>`
- Details **Add to Menu** button: `onClick={e=>{ /* as before */; /* -> #search */ } }`
- Details **Cancel** button should also navigate to Search

TW3.4 User evaluation assignment

Now the DinnerPlanner interaction is complete! so we are ready for user evaluation. Or you can wait until simple persistence is implemented.

Find the assignment in Canvas!

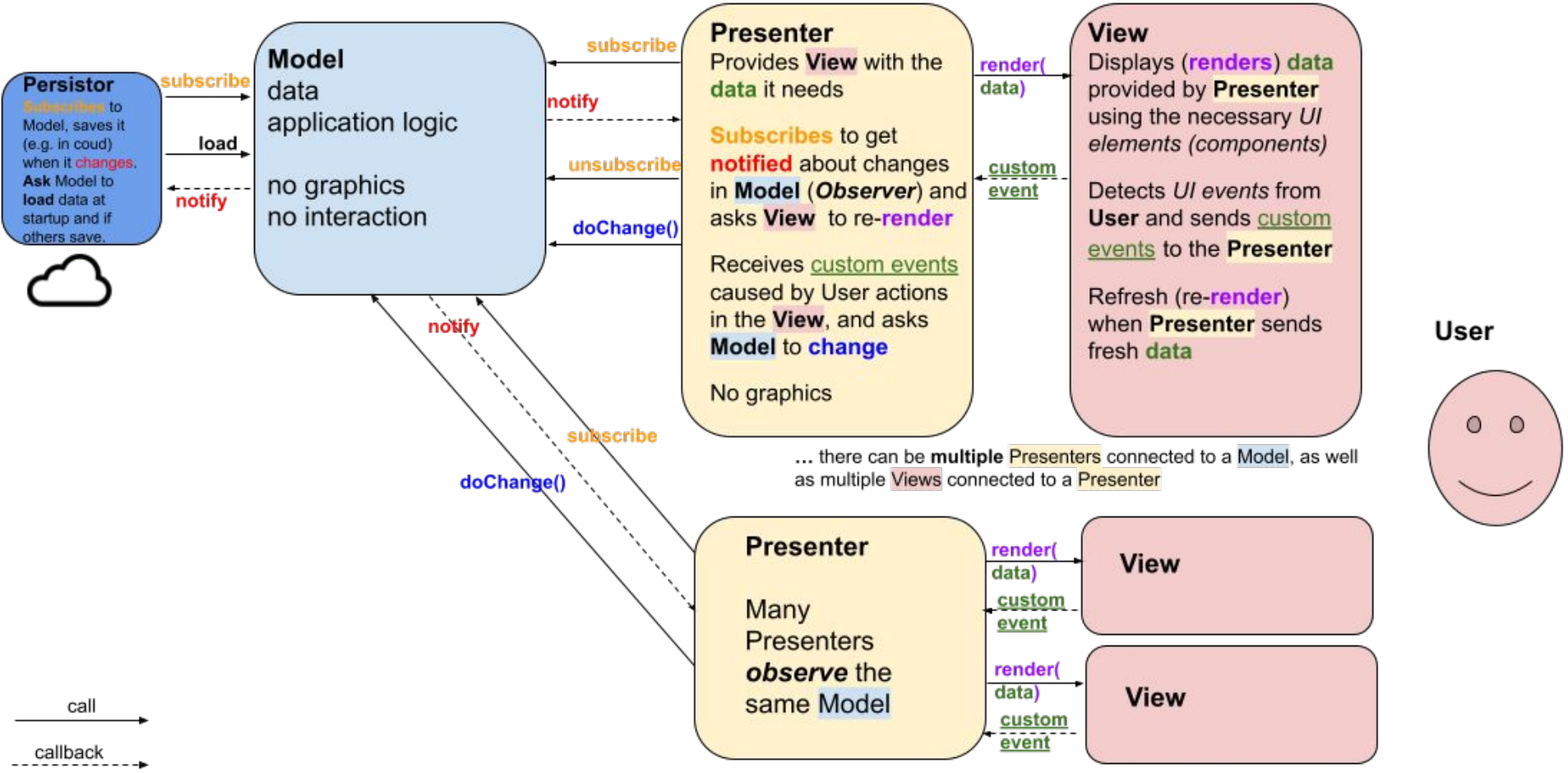
Improve the application based on the user feedback.

TW3.5 Persisting the application state in the cloud

Up until now, re-loading the app led to starting over with an empty DinnerModel with 2 guests.

In this final tutorial section, we will save the application state (DinnerModel) in the cloud using a storage service called Firebase, which is recommended for your Project

- Every time the model changes, an **Observer** (advanced: Redux store subscribe() listener) will save model data to Firebase
- At application startup, we fill an empty model with data retrieved from Firebase
  - There is an issue here: the model will notify **observers** because it was changed (with data from Firebase), and that can lead to a useless Firebase save. We will put in place a protection against that
- Firebase can notify us about every change in its data store.
  - On such notifications, we can update the Model
  - That will update the UI
  - This means that we can load the app in several browser windows, they will all be kept consistent.



The persistor is a Model observer. But it also “observes” other changes coming from the cloud

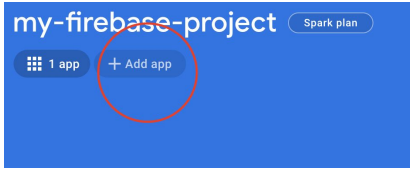
TW3.5 Firebase Setup: Project and App creation

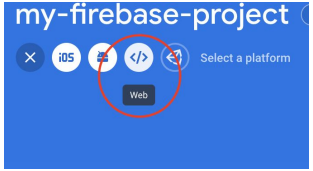
https://console.firebase.google.com/

Login with a Google account

Create a **new Project**

In the Project create a new **App** of type **Web**





Open your App and add a **Realtime Database**

Under Realtime Database / **Rules** (or at db creation) set the rules to

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

These rules will produce a red warning in the firebase console. That is expected and acceptable for the lab. See lecture notes for more advanced settings (useful for project).

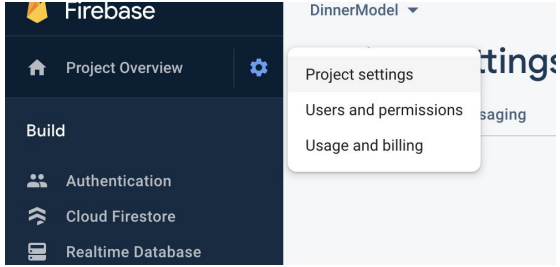
TW3.5 Firebase Setup in HTML / JS

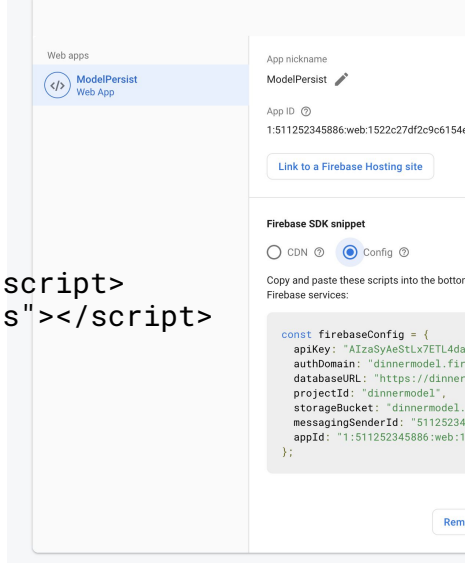
Under **Project Settings** (or at App creation) find your app and look for **Firebase SDK snippet.** Copy-paste **Config** (not CDN) into `js/firebaseConfig.js` It looks like:

```
const firebaseConfig = {
  apiKey: "AIzaSyAe . . . . . . . . ",
  authDomain: "YOUR_PROJECT.firebaseapp.com",
  databaseURL: ...
  ...
  appId: "1:5112...."
};

firebase.initializeApp(firebaseConfig);  // add this line
```

Time to load Firebase in HTML

```
<script src="https://www.gstatic.com/firebasejs/8.2.5/firebase-app.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.2.5/firebase-database.js"></script>
<script src="js/firebaseConfig.js"></script>

<script src="js/firebaseModel.js"></script>
```





TW3.5 Save to Firebase

We do not save the entire model to Firebase as that would attempt to save the observers and redundant data like currentDishDetails, currentDishError, etc.

```
// js/firebaseModel.js
function persistModel(model){
    model.addObserver(function(){
            firebase.database().ref("dinnerModel").set({  // object literal
                  guests: model./*TODO*/,
                  // TODO dishes, currentDish
            });
    });
}
```

In your HTML file:

```
const myModel= new DinnerModel();
persistModel(myModel);
// .. render as before
```

*Optional: we may want to ensure that we do not save to firebase too often. For example rapidly changing the number of guests can lead to frequent saves. A **throttle** function (e.g. from lodash) can help ensure that we save e.g. max once a second.*

Interact with the app to change your model so that the Observer fires, which will trigger the Firebase save.

Check in your Firebase console that the data was saved correctly.

TW3.5 Load from Firebase at startup

```
// js/dinnerModel.js
class DinnerModel{ ..// we add one method. addToMenu and removeFromMenu could be used, but this is shorter:
      setDishes(dishes){ this.dishes= [...dishes]; /* TODO notify observers! */ ;}
}
// js/firebaseModel.js
function persistModel(model){
      model.addObserver(/*as before */);
      firebase.database().ref("dinnerModel").once("value", function(data){
            if(data.val()){
                  model.setNumberOfGuests(data.val().guests);
                  // TODO setDishes, setCurrentDish
            }
      });
}
// react.html
const myModel= new DinnerModel();
persistModel(myModel);
// render as before
```

Note that **Firebase will not store null values**. So e.g. `data.val().currentDish` may be undefined. To address this:
`model.setCurrentDish(..|| null)`
`model.setDishes(..|| [])`

```
// vue.html
const myModel= new DinnerModel();
const TopLevelModel={
       data(){ return {model:myModel};}
       created(){ persistModel(this.model); }
       render(){ /* as before */}
};  // render as before
```

Vue *decorates* its state objects, wrapping a JavaScript **Proxy** around them. That is why Vue Presenters don't need to be Observers: Vue observes all state objects via **Proxy**.

Therefore in Vue **this.model !== myModel** !

The changes made to this.model (the Proxy) will also change myModel, but not the other way around! So we need to read **this.model** from Firebase instead of myModel.

TW3.5 Tuning update from Firebase

1. **Reload your HTML. The UI shows an empty model with 2 guests**
   A few (milli)seconds later, the UI gets populated with data from Firebase.
   To render only when data arrived from Firebase, **in your HTML** you can use a form of **once()** that returns a Promise.
   `firebase.database().ref("dinnerModel").once("value").then( ()=>ReactOrVue.render(/*as before */))`
   Now you should see directy the Firebase-saved data at first render. See discussion in **notes**.

2. When **persistModel**() calls model.**setNumberOfGuests**() etc, the model will notify,
   One of its observers is in **persistModel**(), which will uselessly save to Firebase the data that we just got from Firebase...
   To avoid that, we can **turn off the observer while reading from Firebase**, by setting a boolean flag

```
// js/firebaseModel.js
function persistModel(model){
      let loadingFromFirebase=false;  // boolean flag, used in a JS closure
      model.addObserver(/* as before but do nothing (e.g. return) if loadingFromFirebase is true! */ );
      firebase.database().ref("dinnerModel").once("value", function(data){
            loadingFromFirebase= true;
            if(data.val()){ /* as before */}
            loadingFromFirebase= false;  // see notes for safer code
      });
}
```

*This code uses a JavaScript **closure**.*
`loadingFromFirebase` *is available to both callbacks even after persistModel() has ended.*
*Identify other uses of **closure** in your code!*

TW3.5 Keep in sync with Firebase

Maybe two users plan the dinner at the same time, each with their own interactive device. When one user changes the dinner and saves in Firebase, we want Firebase to notify the other (notify both actually)

Change **once()** to **on()** in **persistModel()**

Now the `function(data){ ..if(data.val()).. }` callback will be invoked every time `"dinnerModel"` changes in the cloud (Firebase store).

Test by opening your app in **two browser windows**.

Changing the number of guests, the menu, or the current dish in one window should update the other window.

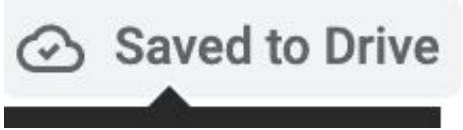
Optional: using several **on**() listeners will lead to more detailed model update:

```
firebase.database().ref("dinnerModel/guests").on("value", data=> /* set nr guests to data.val() */)
```

Furthermore, arrays like `dishes` give more detailed **events** like **child_added**, so one can write:

```
firebase.database().ref("dinnerModel/dishes").on("child_added", data=> /* add to menu! */)
```

TW3.5 **Optional**: Firebase “save status” UI



Add properties to the model: savedOK, saveError

In **firebaseModel.js**

- **set**() returns a promise! Add then() and catch() to it...
- drive the 2 properties above. Set them to null before attempting to save
- call notifyObservers when any of the 2 properties change
  - but make sure (using a boolean flag) that the Firebase Observer ignores such notifications!
  - one can also consider using a separate Model for Save status...

Add a Presenter (model Observer in React) + View rendered at e.g. the top or bottom, that display the 3 properties. For example (re-visit lecture material on **conditional rendering** using e.g. boolean?ifTrue:ifFalse conditional expressions, or boolean && ifTrue || ifFalse)

- “saving...” savedOK and saveError are null
- “saved” savedOK is not null, saveError is null
- “broken cloud” savedOK null, saveError is not null (Firebase works offline, the changes will be saved locally)

Since Firebase works offline, saveError will almost always be null. To show to the user whether their their data is saved in the cloud or just locally you can use ref(".info/connected”) . A Presenter can listen to it directly (similar to window.location.hash) or if it is useful for many Views, you can save it in the Model/Application state.

Congratulations, you are done with the Tutorial!

Commit! Push!

Time to start your project!

In your **README.md** file, indicate whether we should test **react.html** or **vue.html**,

or both if you want the two-framework bonus (see the Canvas “Bonus points” assignment)

- You can add the implementation for the second framework until the end of the course
- Also you can do any other bonus point work until the end of the course