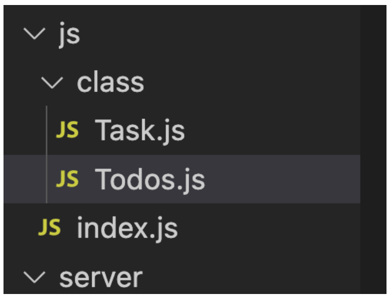
Exercise. Todo (part 4) – Create classes for software logic

This exercise demonstrates, how to implement classes for software logic. By following separate of concerns principle, it is not a good idea to include UI, software logic and http calls into a single file.

Key learnings for this part are:

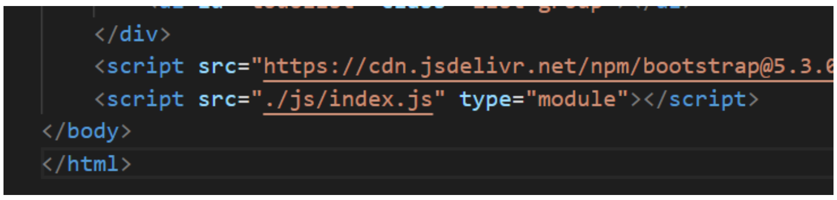
- Create classes for software logic and separating front-end code into manageable parts
- Object-oriented programming with JavaScript
- Exporting/importing files
- Using promises

Start by creating a subfolder class under js and add two JavaScript files for classes.



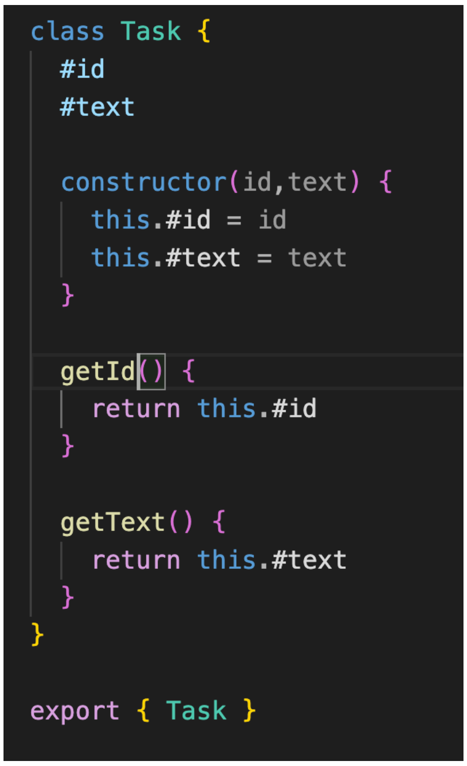
```
js
  class
    Task.js
    Todos.js
  index.js
server
```

Separate files need to be exported/imported and therefore index.js file needs to have a type module (otherwise JavaScript does not allow export/import on index.js).



```
    </div>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.
    <script src="./js/index.js" type="module"></script>
</body>
</html>
```

Define class for a Task. Task has properties id and a text, which are private (JavaScript uses # for declaring private member variables). Constructor is used to create new object. GetId and getText methods ("getters") are implemented to enable reading these values "outside" class. In object-oriented programming data encapsulation is important concept. For example, id and text for task are assigned when new object is created, and values can be read but not set outside this class. Finally export definition is required so other js files can import this file.

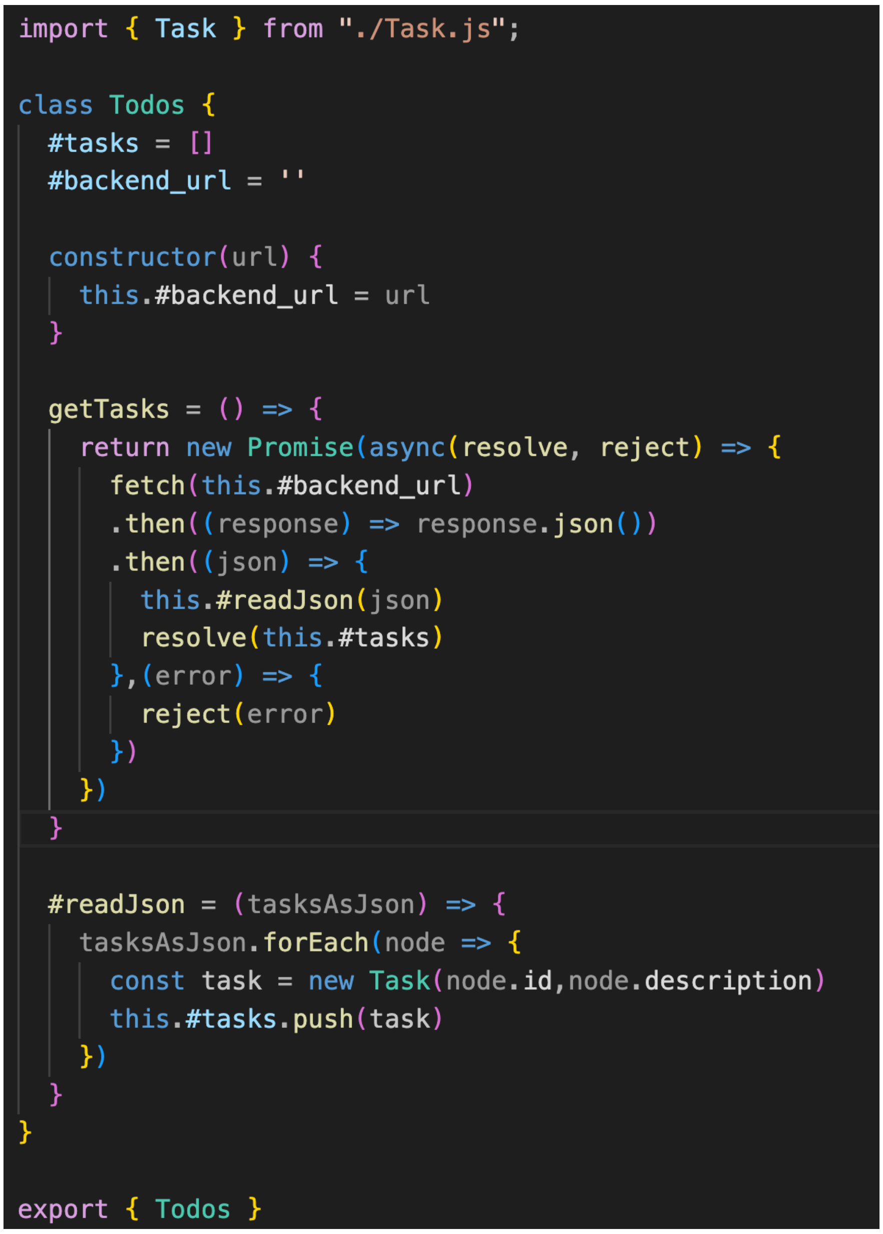


```
class Task {
  #id
  #text

  constructor(id,text) {
    this.#id = id
    this.#text = text
  }

  getId() {
    return this.#id
  }

  getText() {
    return this.#text
  }
}

export { Task }
```

Define class for Todos. This class will contain logic for retrieving and adding new tasks (implemented later). When Todos object is created, backend url is passed as an argument. All tasks as stored into array (even though at this point we could manage without an array) and later we could implement features, such as sorting, search etc.

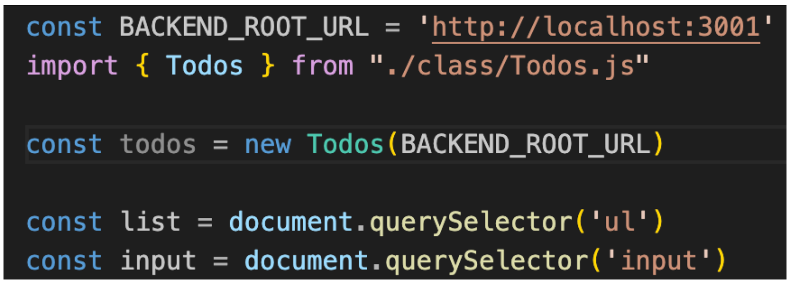
GetTasks method is asynchronous. Code is executed on the “background” while UI is displayed. Method will return a promise, so it is easy to detect, when data retrieval is finished, and UI can be updated. A private method (readJson) for reading JSON into array of tasks is also implemented. It is a good practise to divide code into smaller manageable parts.

Since getTasks method returns a promise, in case of success resolve is returned. If there is an error, reject is returned. Resolve returns retrieved tasks and reject returns information about the error.



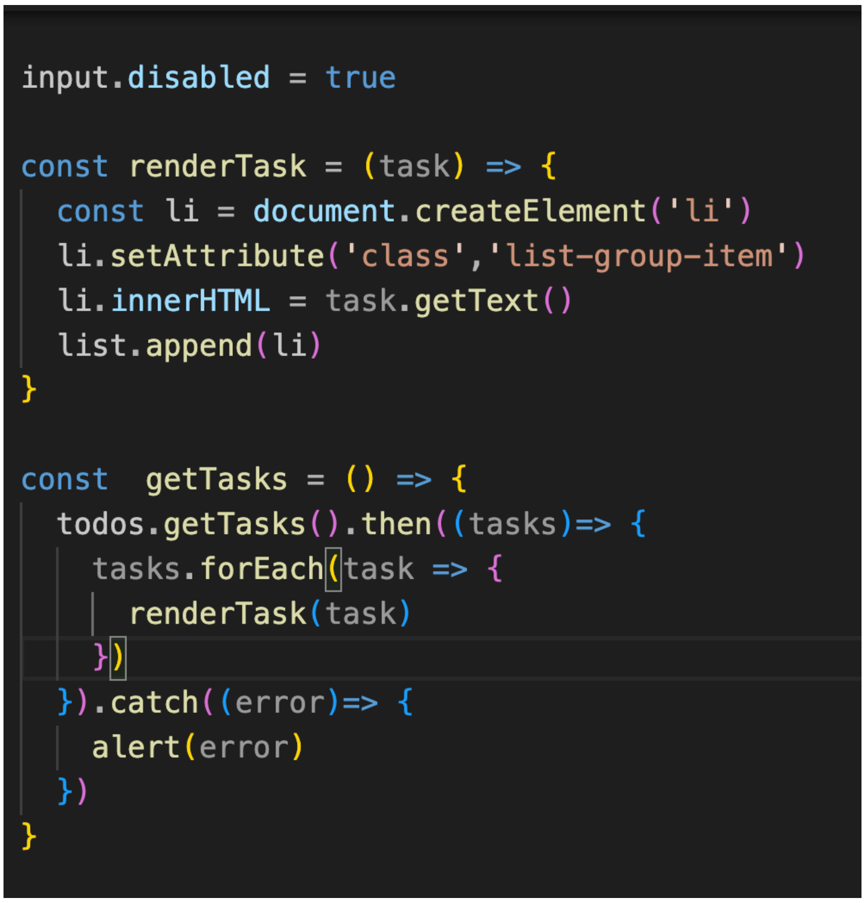
```
import { Task } from "./Task.js";

class Todos {
  #tasks = []
  #backend_url = ''

  constructor(url) {
    this.#backend_url = url
  }

  getTasks = () => {
    return new Promise(async(resolve, reject) => {
      fetch(this.#backend_url)
      .then((response) => response.json())
      .then((json) => {
        this.#readJson(json)
        resolve(this.#tasks)
      },(error) => {
        reject(error)
      })
    })
  }

  #readJson = (tasksAsJson) => {
    tasksAsJson.forEach(node => {
      const task = new Task(node.id,node.description)
      this.#tasks.push(task)
    })
  }
}

export { Todos }
```

Import required classes and create an object of Todos class.



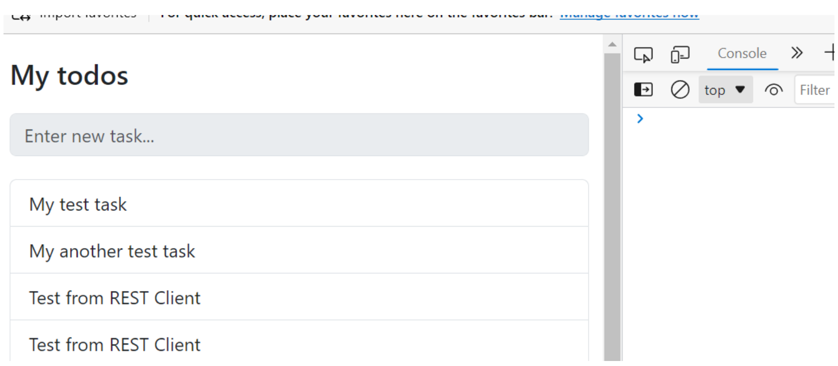
```
const BACKEND_ROOT_URL = 'http://localhost:3001'
import { Todos } from "./class/Todos.js"

const todos = new Todos(BACKEND_ROOT_URL)

const list = document.querySelector('ul')
const input = document.querySelector('input')
```

On index.js modify getTasks and renderTasks functions. GetTasks uses todos object (created based on Todos class). Returned promise is handled and in case data is returned, it is looped through, and a task object is passed into renderTask function. GetText method ("getters") is used to read private member variable value.



```
input.disabled = true

const renderTask = (task) => {
  const li = document.createElement('li')
  li.setAttribute('class','list-group-item')
  li.innerHTML = task.getText()
  list.append(li)
}

const  getTasks = () => {
  todos.getTasks().then((tasks)=> {
    tasks.forEach(task => {
      renderTask(task)
    })
  }).catch((error)=> {
    alert(error)
  })
}
```

Test that tasks are retrieved and displayed.



My todos

Enter new task...

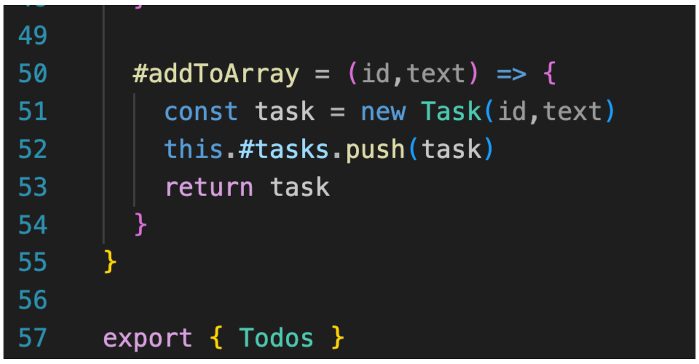
My test task

My another test task

Test from REST Client

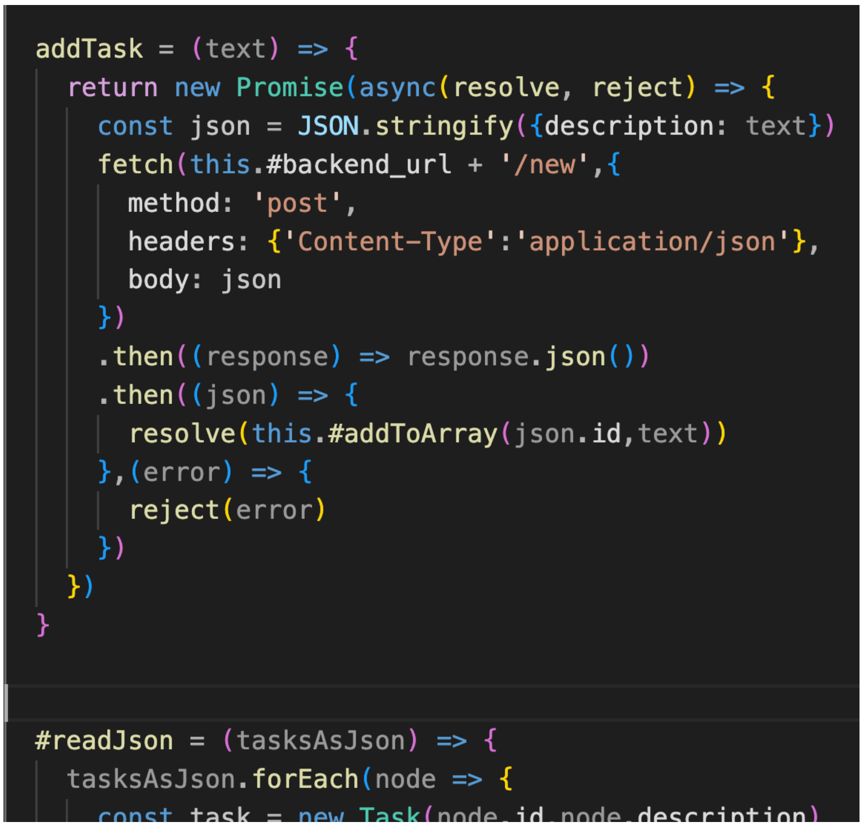
Test from REST Client

On todos.js add private method after readJson to add new task to tasks array. Method will also return added task.



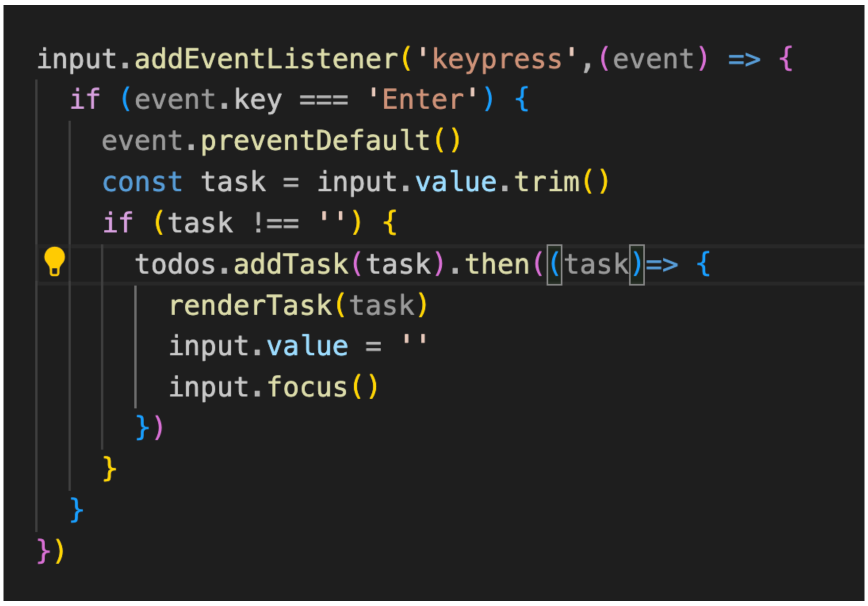
```
    #addToArray = (id,text) => {
      const task = new Task(id,text)
      this.#tasks.push(task)
      return task
    }
  }

export { Todos }
```

Add public method for adding a new task to Todos.js. Again, method returns a promise. If call is successful, task object is returned. Fetch executes post call and has some extra parameters, such as method, headers and body. Required endpoint (/new) is already implemented (and tested) on the backend receiving new task as JSON.



```
addTask = (text) => {
  return new Promise(async(resolve, reject) => {
    const json = JSON.stringify({description: text})
    fetch(this.#backend_url + '/new',{
      method: 'post',
      headers: {'Content-Type':'application/json'},
      body: json
    })
    .then((response) => response.json())
    .then((json) => {
      resolve(this.#addToArray(json.id,text))
    },(error) => {
      reject(error)
    })
  })
}


#readJson = (tasksAsJson) => {
  tasksAsJson.forEach(node => {
    const task = new Task(node.id,node.description)
```

Update addEventListener so it uses method implemented on Todos class. Add task receives description for task as parameter and inserted task object (with id returned from the database) is returned/resolved. That is rendered on UI, input field is emptied, and focus is set to it (so user can easily add another task).



```
input.addEventListener('keypress',(event) => {
  if (event.key === 'Enter') {
    event.preventDefault()
    const task = input.value.trim()
    if (task !== '') {
      todos.addTask(task).then((task)=> {
        renderTask(task)
        input.value = ''
        input.focus()
      })
    }
  }
})
```

Test out that app works. Tasks should be listed, and you should be able to add new tasks. Code for the website is now divided in a way, that classes contain all software logic and HTTP calls and Index.js contains just UI related code (implementation does not follow object-oriented approach).