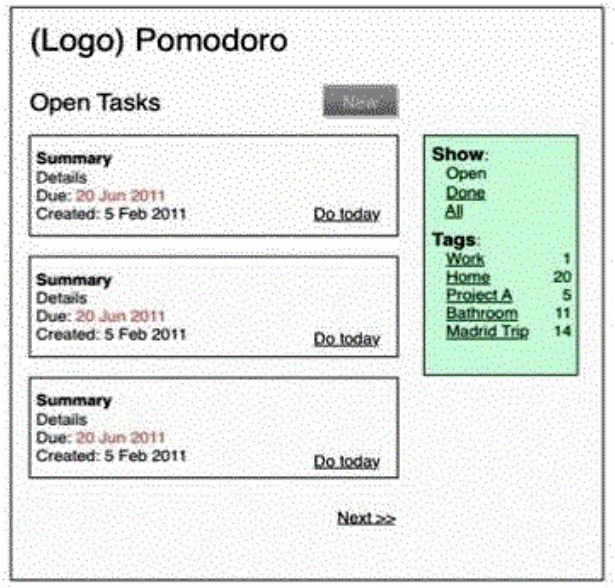
002 Grails plug-in을 통한 개발 확장

Grails plug-in을 통한 개발 확장

Table of Contents

이전에 작성한 PomodoroApp 의 화면을 아래와 같이 구성할 예정입니다.

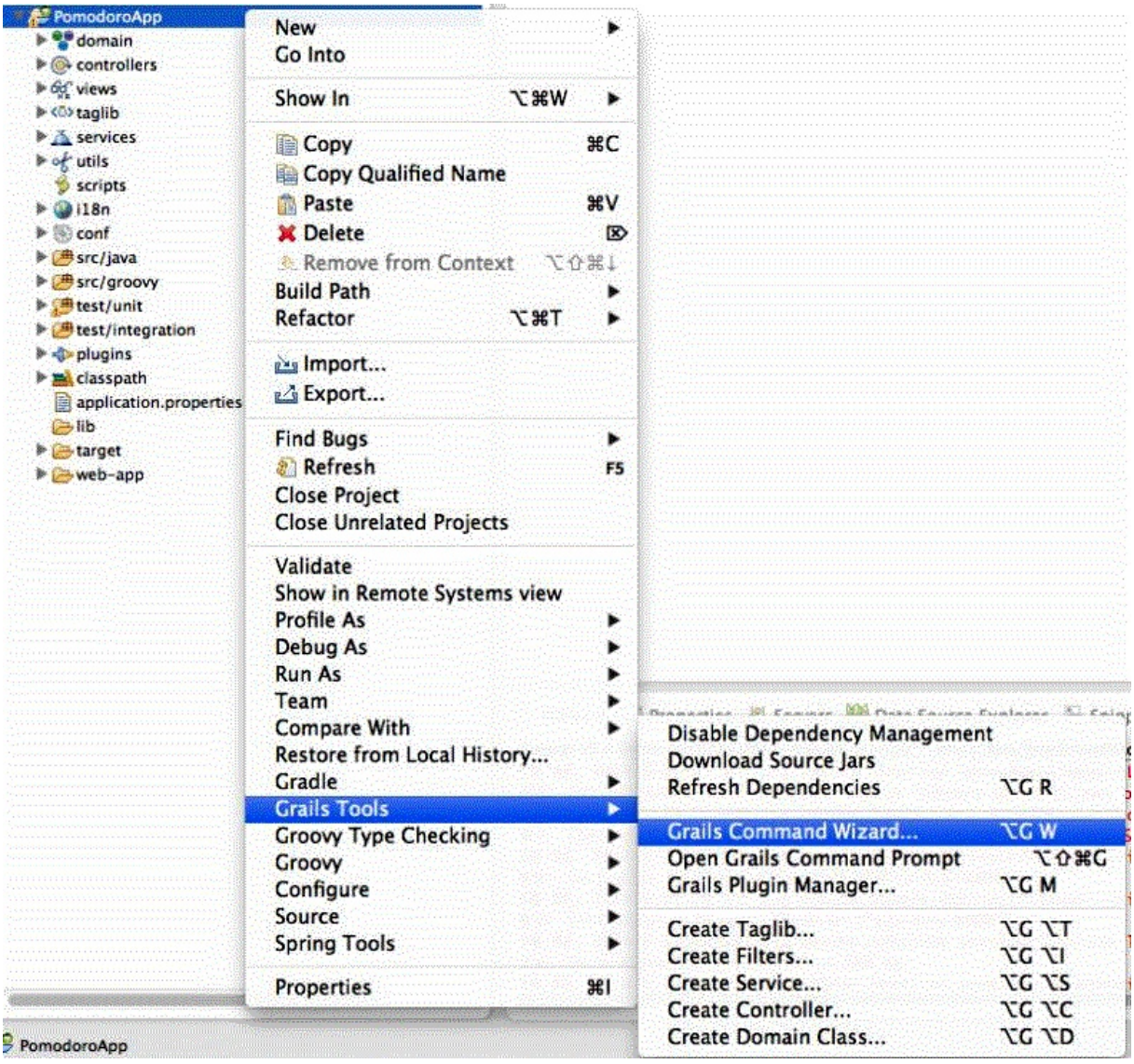


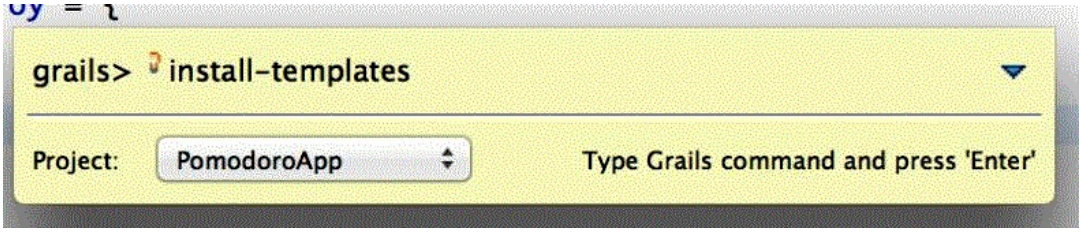
그 전에, plug-in을 추가하여 Blueprint 라는 css framework 을 추가하여 화면을 구성하겠습니다.

Plug-in 추가하기

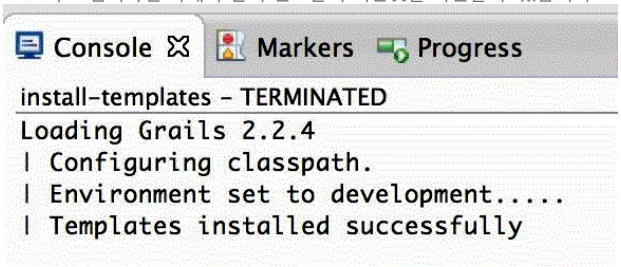
scaffolding 구조를 파악하기 위해 아래와 같은 플러그인을 추가합니다.

프로젝트 우 클릭 후 Grails Tools 에 Grails Tool Open Grails Command Prompt를 선택합니다.

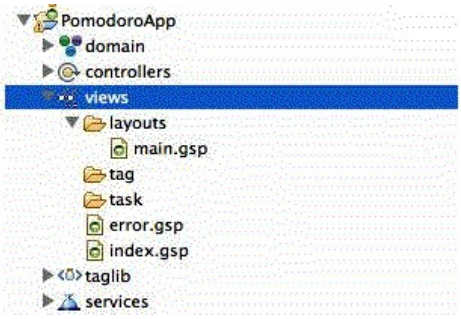




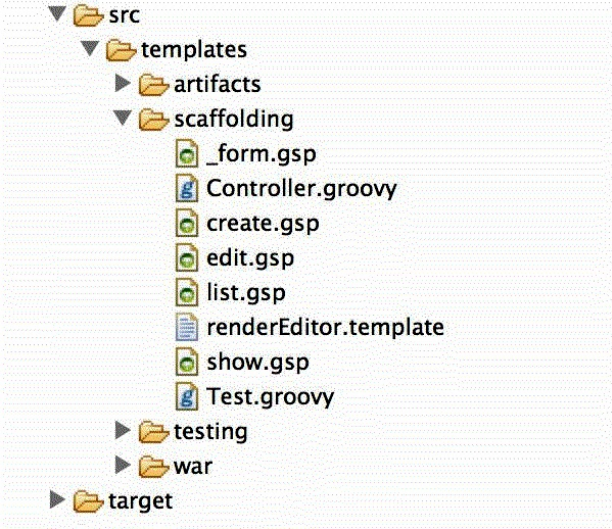
Command 입력 부분에 install-templates 라고 입력하면 아래와 같이 인스톨이 되는것을 확인할 수 있습니다.



이제 프로젝트를 새로 고침하면 views 영역이 새로 생겨난것을 확인 할 수 있고, 이 영역에 이전에 작성한 task와 tag라는 디렉토리가 자동으로 생성된것을 볼 수 있습니다.



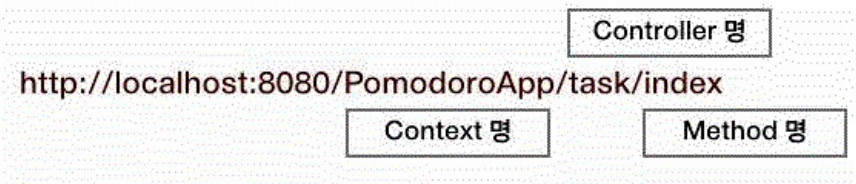
실제 이 플러그인을 인스톨 하고 나면 /src/templates/scaffolding 하위에 뼈대에 해당하는 파일들이 생성되는것을 확인할 수 있습니다.



이제 TaskController 를 열어 아래와 같이 index 라고 하는 빈 클로저를 선언합니다.

```
1  class TaskController {
2
3      static scaffold = Task
4
5      def index = { }
6  }
```

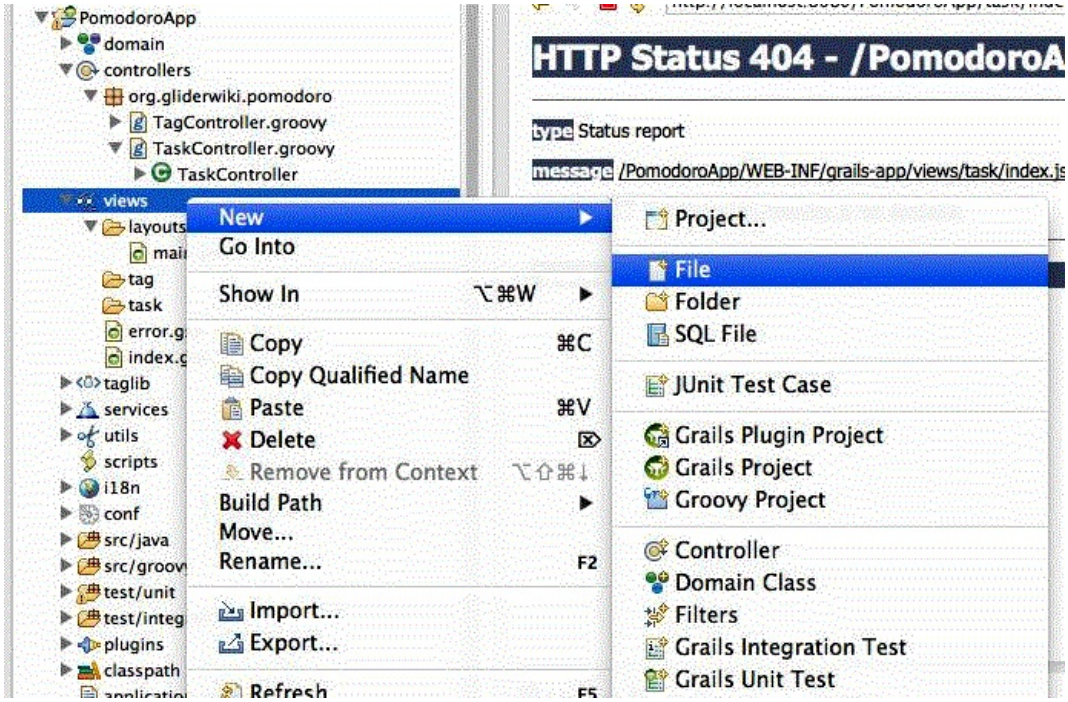
서버를 기동한 후 http://localhost:8080/PomodoroApp/task/index 로 접근하게 되면 404 not found 라고 나오는 것을 확인할 수 있습니다.
여기서 정의한 index라고 하는 빈 클로저는 컨트롤러의 url 액션과 매핑되는 부분으로 결과적으로 views 영역 안에 index.jsp 를 찾지 못해서 나오는 에러입니다.
즉, 뷰 리졸버를 정의하는 것처럼 Task라고 하는 Controller에 대응하는 Action이 실행되려면 task라고하는 Base 폴더에 index.gsp가 있어야 하는데 파일이 없으므로 에러를 뱉어내는 것입니다.

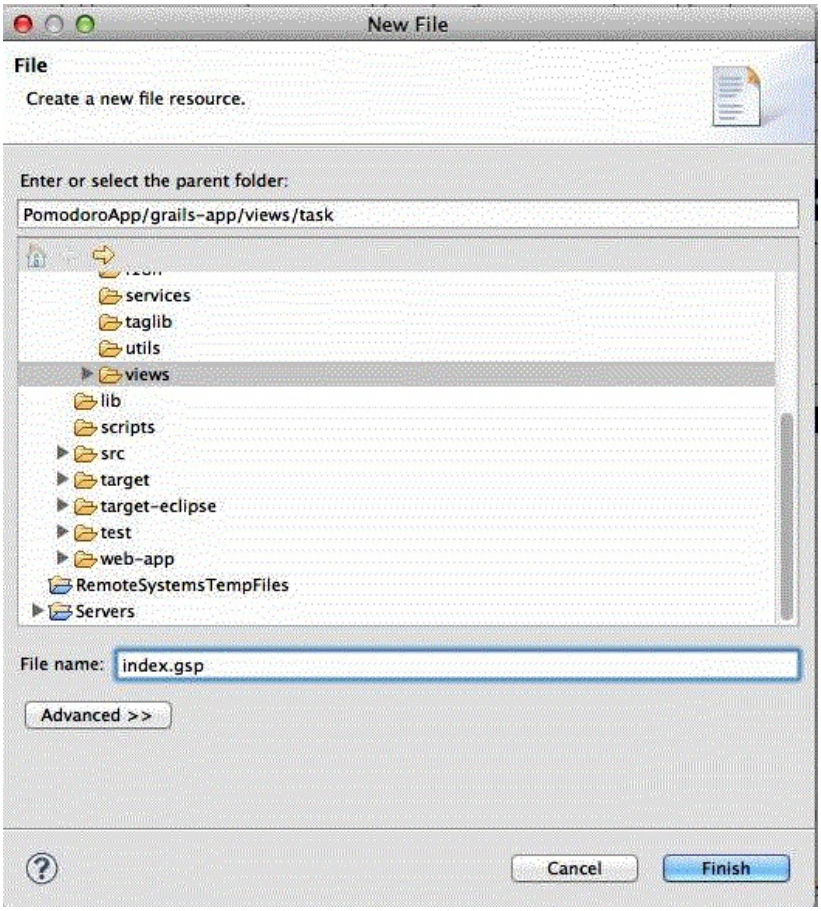


Grails의 Convention 으로 보면 Controller 명이 view 영역 하위에 디렉토리로 매핑되기 때문에 TaskController는 views/task 디렉토리를 보고 있고 빈 클로저로 선언한 index는 task 디렉토리 안의 index.gsp를 바라보게 됩니다.

실제 내부에서는 index 라고 하는 jsp를 찾기 때문에, gsp 가 컴파일 되어 실제 view가 구성된다고 생각하면 됩니다.

이제 views 영역 안에 아래와 같이 task 폴더와 index.gsp 파일을 만들어주도록 합니다.

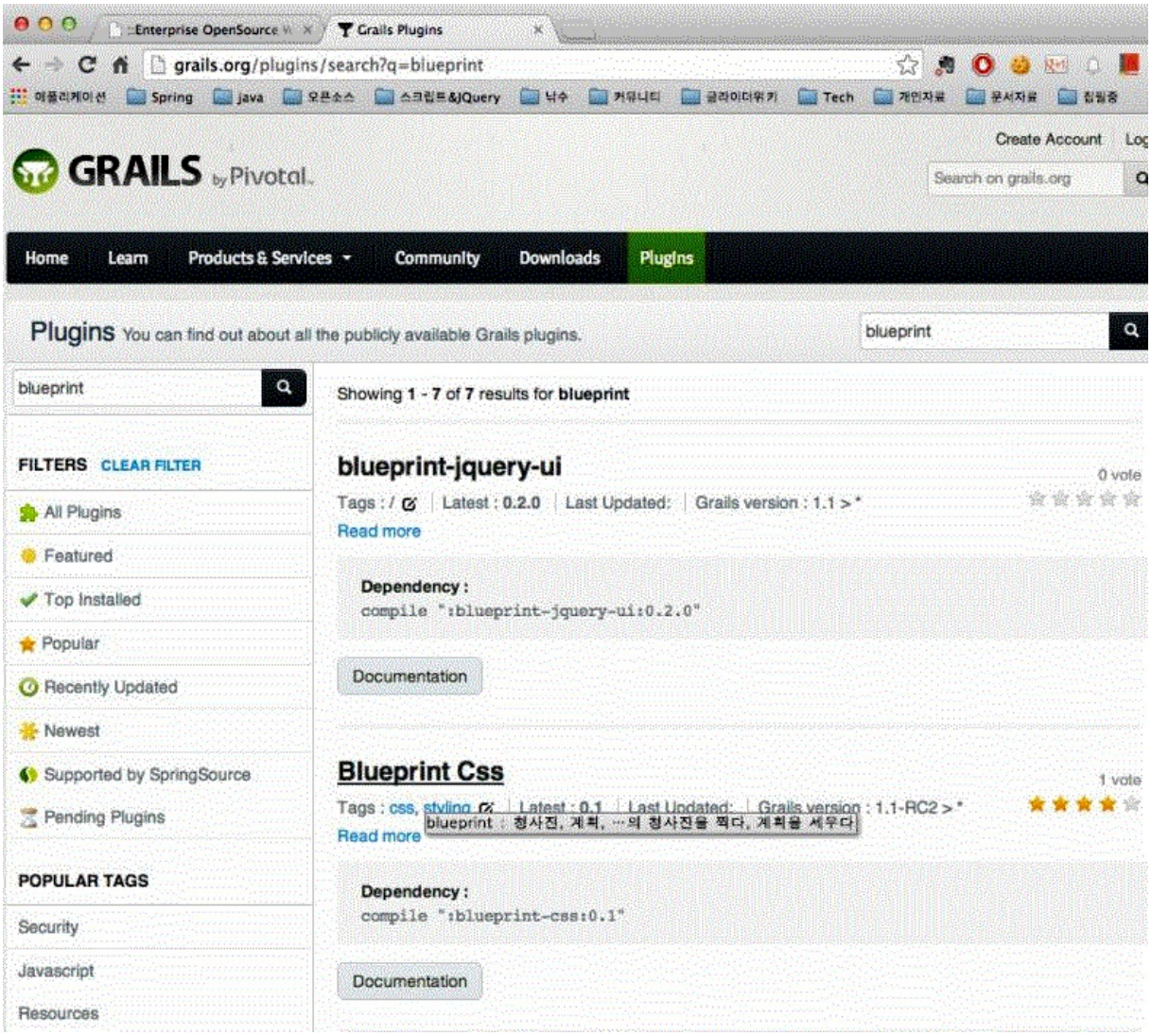




view 폴더의 경로에 task 라고 하는 경로를 적고 파일명에는 index.gsp로 생성하면 views 하위에 task라는 디렉토리가 생성이 되고 index.gsp파일이 생성된것을 확인할 수 있습니다. (task 폴더를 만든 후에 파일을 생성해도 됩니다.)

화면 구성하기

이제 화면을 구성하기 위해 blueprint라는 css 플러그인을 추가하도록 하겠습니다.
plugin은 grails의 공식 홈페이지(http://grails.org/plugins)에서 검색이나 카테고리 메뉴등을 통해서 어떤것들이 있는지 한 눈에 볼 수 있습니다.

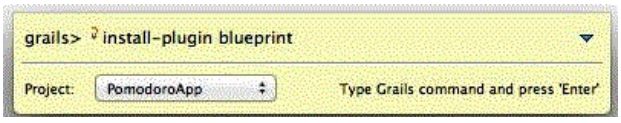


비교적 documentation이 잘 되어있으므로, 소개나 설치 및 사용법 등은 해당 페이지에서 확인이 가능합니다.

이제 Grails Tool 에 있는 Open Grails Command Prompt를 열어서 아래와 같이 blueprint 플러그인을 설치합니다.

blueprint plug-in

install-plugin blueprint



설치가 완료 되었으면 index.gsp를 작성하도록 하겠습니다.

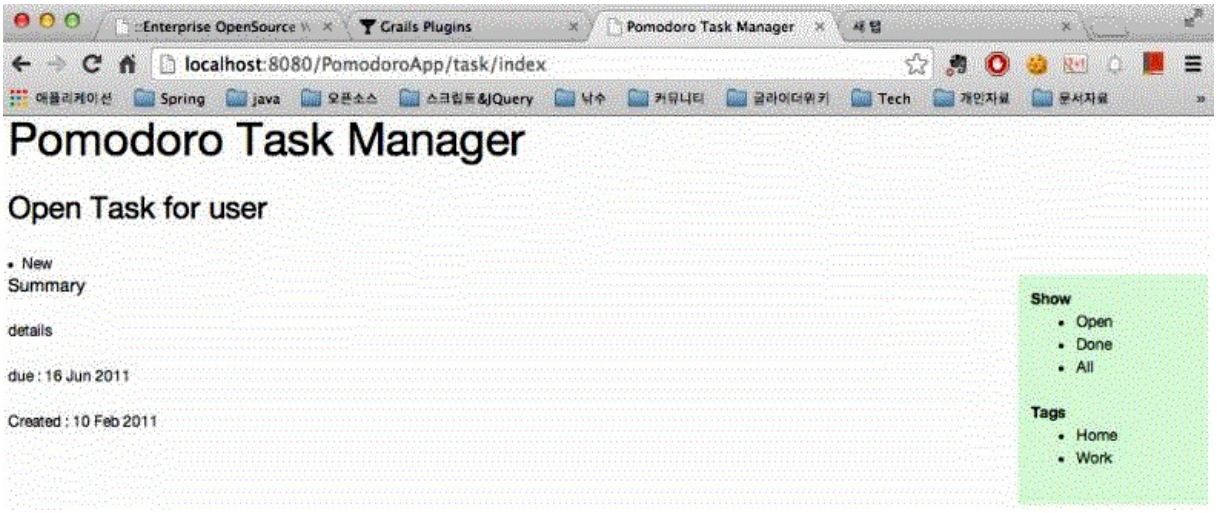
```
<!doctype html>
<html>
<head>
<title>Pomodoro Task Manager</title>
<blueprint:resources />
<link href="${resource(dir: 'css', file: 'app.css')}" type="text/css"
    rel="stylesheet">
</head>
<body>
    <div class="container">
        <h1 class="span-24 last">Pomodoro Task Manager</h1>
        <div class="span-24">
            <div class="span-16">
                <h2>Open Task for user</h2>
            </div>
            <div class="span-14">
                <ul class="actions">
```

```
                <li>New</li>
              </ui>
            </div>
            <div class="span-4 last"></div>
        </div>

        <div class="span-20">
            <div class="task">
                <h4>Summary</h4>
                <p>details</p>
                <p>due : 16 Jun 2011</p>
                <p>Created : 10 Feb 2011</p>
            </div>
        </div>

        <div class="span-4 last">
            <dl class="sidebar">
                <dt>Show</dt>
                <dd>
                    <ul>
                    <li>Open</li>
                    <li>Done</li>
                    <li>All</li>
                    </ul>
                </dd>
                <dt>Tags</dt>
                <dd>
                    <ul>
                    <li>Home</li>
                    <li>Work</li>
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
</body>
</html>
```

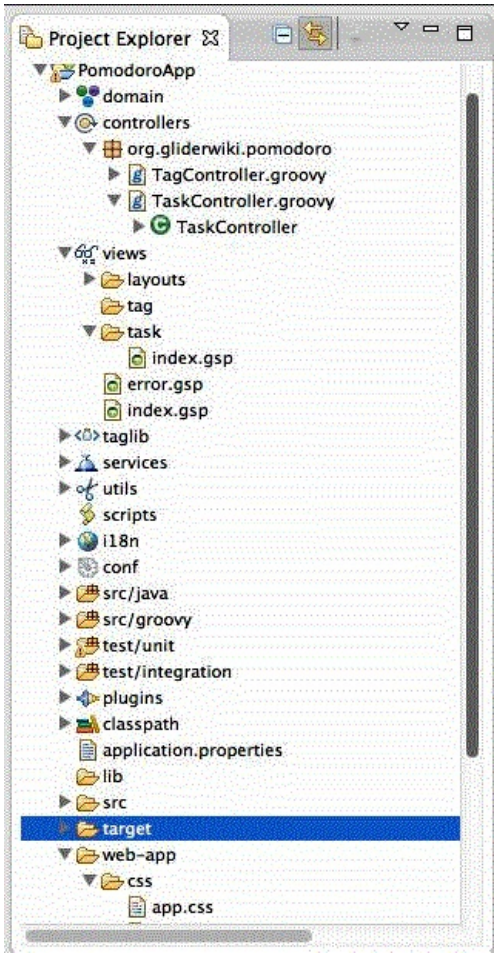
이제 app.css 파일을 작성합니다.
web-app 밑에 css 폴더 하위에 위치하게 됩니다.

```
.sidebar {
    background-color: #DDFFDD;
    padding: 10px;
}

ul.actions {
    list-style-type: none;
}
```

서버를 기동하고 결과를 확인해보도록 하겠습니다.



지금까지의 프로젝트 디렉토리 및 파일 구조는 아래와 같습니다.



여기까지 작성한 후 이제 DB 처리와 Logic 을 어떻게 화면으로 연결해야 하는지 알아보도록 하겠습니다.

Controller Action과 모델을 이용한 MVC 완성하기

실제 MVC 패턴에서는 도메인 객체를 통해 DB의 데이터를 담아 Controller로 View에 전달하는 형태로 구성됩니다.

먼저 Task 도메인에 아래와 같이 Status를 추가합니다.

```
class Task {
    String summary
    String details
    Date dateCreated
    Date deadline
    Long timeSpent = 0L
    String status

    static hasMany = [tag : Tag]

    static constraints = {
        summary blank:false, unique:true
        details blank:false
        deadline nullable:true
        timeSpent min:0L
    status inList:["Open", "Done"]
    }
}
```

Status 는 리스트 형태의 속성값인 Open 과 Done 을 통해 상태를 정의합니다.
할일 목록에서 실제 일이 시작(Open)단계인지, 종료(Done)단계인지를 구분하는 값입니다.

이제 BootStrap 파일을 열어서 초기 로딩시 지정된 Task의 기본 상태를 open으로 정의하도록 합니다.

```
import org.gliderwiki.pomodoro.Tag
import org.gliderwiki.pomodoro.Task

class BootStrap {

    def init = { servletContext ->
        def workTag = new Tag(name: "Work").save(failOnError:true)
        def homeTag = new Tag(name: "Home").save(failOnError:true)

```

```
        def task = new Task(
            summary:"Do 2nd intro screencast",
            details:"Create the second intro screencast for Grails",
            status:"Open"
            )
        task.addToTag(workTag)
        task.addToTag(homeTag)

        task.save(failOnError:true)
    }
    def destroy = {
    }
}
```

이제 빈 껍데기만 선언했던 TaskController 의 index 클로저에 아래와 같이 도메인 데이터를 Fetch하여 정렬해보도록 하겠습니다.

```
class TaskController {

    static scaffold = Task

    def index = {
        def tasks = Task.findByStatus("Open", [sort: "deadline", order:"asc"])
        def tags = Tag.list(sort:"name", order:"asc")

        return [tasks:tasks, tags:tags]
    }
}
```

정렬 옵션은 각각 deadline과 name으로 주고 view 영역에는 map형태로 tasks와 tags를 넘겨줍니다.
물론 네이밍은 openTasks:tasks 형태로 서로 구분하여 지정할 수 있습니다.

이제 index.gsp를 열어 fetch 해온 데이터를 보여주도록 합니다.

```
<dt>Tags</dt>
    <dd>
        <ul>
        <g:each in="${ tags }" var="tag">
        <li>${ tag.name }</li>
        </g:each>
        </ul>
    </dd>
```

콜렉션션타입을 표현하기 위해서는 gsp의 each 를 통해 가능합니다.

in 어트리뷰트를 통해 해당 리스트의 인덱스에 접근할 수 있습니다.
또한 li 엘리먼트 내에 gsp 표현식을 이용하여 name 프로퍼티에 접근 할 수 있습니다.

이제 화면을 실행해보면 별다른 에러없이 출력됨을 알수 있습니다.

만약 도메인 클래스에 status를 정의 하지 않았거나, BootStrap 내에 상태값을 추가 하지 않았다면 서버 실행 시점에 아래와 같은 에러를 만날수도 있습니다.

```
| Error 2014-03-08 07:28:14,678 [localhost-startStop-1] ERROR context.GrailsContextLoader  - Error initializing the application: Valida
- Field error in object 'org.gliderwiki.pomodoro.Task' on field 'status': rejected value [null]; codes [org.gliderwiki.pomodoro.Task.status.nu
Message: Validation Error(s) occurred during save():
- Field error in object 'org.gliderwiki.pomodoro.Task' on field 'status': rejected value [null]; codes [org.gliderwiki.pomodoro.Task.status.nu
    Line | Method
->>   17 | doCall                      in BootStrap$_closure1
```

마지막으로, view 영역의 template 구성을 해보도록 하겠습니다.
소스가 길어지거나 복잡해지면 View 영역도 모듈화를 할 필요가 있습니다. GSP Fragment 를 이용하여 Template의 조각 형태로 페이지를 구조화 할 수 있습니다.

명명규칙은 해당 디렉토리에 언더스코프로 시작하는 gsp를 만들어서 구분할 수 있습니다.

task 디렉토리 하위에 _taskCard.gsp라는 파일을 만들어 보겠습니다.

```
<div class="task">
    <h4>${ task.summary }</h4>
    <p>${ task.details }</p>
    <div class="due"> due : <g:formatDate date="${ task.deadline }" format="dd MMM yyyy" /></div>
    <div class="created">Created : <g:formatDate date="${ task.dateCreated }" format="dd MMM yyyy" /></div>
</div>
```

Task 데이터를 출력하는 부분으로, 이제 index.gsp에 Task 영역은 template render를 통해 템플릿 구조로 변경하도록 합니다.

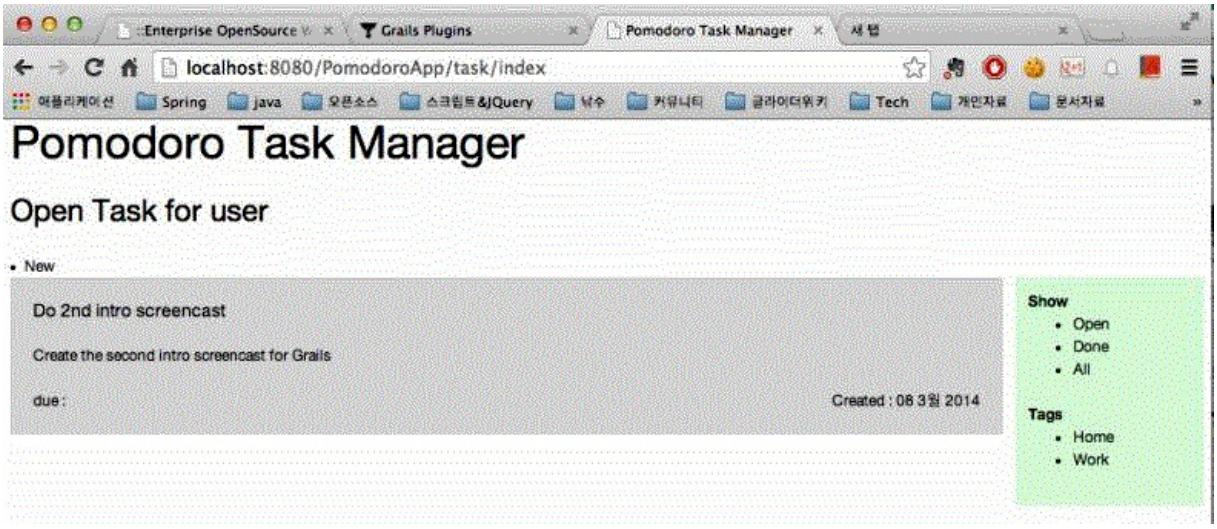
```
<!doctype html>
<html>
<head>
<title>Pomodoro Task Manager</title>
<blueprint:resources />
<link href="${resource(dir: 'css', file: 'app.css')}" type="text/css"
    rel="stylesheet">
</head>
<body>
    <div class="container">
        <h1 class="span-24 last">Pomodoro Task Manager</h1>
        <div class="span-24">
            <div class="span-16">
                <h2>Open Task for user</h2>
            </div>
            <div class="span-14">
                <ui class="actions">
                    <li>New</li>
                </ui>
            </div>
            <div class="span-4 last"></div>
        </div>

        <div class="span-20">
            <g:render template="taskCard" collection="${tasks}" var="task" />
        </div>

        <div class="span-4 last">
            <dl class="sidebar">
                <dt>Show</dt>
                <dd>
                    <ul>
                    <li>Open</li>
                    <li>Done</li>
                    <li>All</li>
                    </ul>
                </dd>
                <dt>Tags</dt>
                <dd>
                    <ul>
                    <g:each in="${ tags }" var="tag">
                    <li>${ tag.name }</li>
                    </g:each>
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
</body>
</html>
```

25라인에 추가된 template 구문은 언더스코프와 확장자를 쓰지 않고 렌더링 한다는 것을 확인할 수 있습니다.

css 파일을 아래와 같이 변경한 후 브라우저로 접근해 보도록 하겠습니다.

```
.sidebar {
    background-color: #DDFFDD;
    padding: 10px;
}

ul.actions {
    list-style-type: none;
}

.task {
    background-color: #DDD;
    position: relative;
    margin-bottom: 1em;
    padding: 1.5em;
}

.task .created {
    position: absolute;
    bottom: 1.5em;
    right: 1.5em;
}
```

페이지를 새로고침하면 아래와 같이 나오게 됩니다.



지정된 데이터와 Template이 제대로 호출 되었음을 확인할 수 있습니다.

관련 키워드

Grails
