It seems to be a contradiction. 'Technical design' is something for the old-timers in the IT business, those who think in terms of waterfall methods. 'AngularJS' is the toy of the Web nerds, rapidly building a nice app. Do you think they will wait for design documentation? This paper explains that technical design is important for professional web applications and provides concrete examples of how such a design could look like in case the AngularJS framework is used. I will focus on the form of design rather than design patterns and decisions.

Technical design in UML for AngularJS applications

and other web applications

Terminology

To avoid misunderstandings, I will first give a definition. In this paper, a *web application* is a client application executed by a web browser. This implies that, almost certainly, the application is written in HTML, CSS and JavaScript. Such an application is usually only one part of a solution, the other components running on one or several servers. Using this definition, I deliberately deviate from definitions like the one given by Wikipedia, which also considers the back end software to be part of the web application, because I need that for this paper. Besides, with my definition, the term is better delineated: many web applications use multiple APIs, including third party APIs (like Google's) and with the Wikipedia definition, it would be unclear which software exactly belongs to the web application.

An AngularJS application is a web application based on the AngularJS framework, an open source framework developed by Google.

UML means Unified Modeling Language, a standard notation for software models, maintained by the Object Management Group. UML defines all kinds of concepts and their graphical representations. For example, a class is represented by a rectangle.

To understand this paper, some basic knowledge of AngularJS and UML is required.

Technical design in an agile environment

It should be obvious, but let me put this question anyway: Why would you create functional and technical models for a web application? Nowadays, most web applications are developed by teams that have adopted an agile development method, like Scrum or Kanban. These methods do not prescribe the production of design documentation, which may give people the impression that such documentation is not needed. Furthermore, the Agile Manifesto says: “We value working software over comprehensive documentation”. The fact that the same manifesto also states that documentation is valuable, is sometimes ignored. Good developers can build high-quality applications without any technical design, but nevertheless, I plead for documenting any professional application up to a certain level of detail, not sacrificing an iterative process or prototyping. Why? Because it saves time and money and the users will get a better application! There is one precondition: the developers must support it. The savings arise from the fact that the design provides insight into the software structure and thereby reduces the chance of a bad structure (spaghetti code, inconsistencies, code duplication). Both the insight into the structure of the software and the high quality of the structure subsequently diminish...

a. the chance of bugs and consequently the time it takes to solve bugs;
b. the time it takes to determine the best way of integrating a particular change or extension;
c. the time it takes to rectify flaws in the structure later on.

Like I said, applications need technical documentation *up to a certain level of detail*, i.e. not too detailed. The design should be sufficient to subsequently find more details in the source code. If the source code has not been written yet, the design should be sufficient for the developer to fill in the details on his own discretion. By the way, it is not always required to have the design finished before the coding starts. You may decide to make a prototype first, evaluate it, design the ultimate situation and then improve the prototype according to the design.

I recommend to let the design and the code be written by different persons, closely working together and influencing each other. This leads to better quality compared to developers implementing certain functionality completely on their own. Depending on the individual skills and affinities, it might be a good option to let the same persons design and implement alternately.

Much more information about designing in an agile environment can be found on Scott Ambler’s website www.agilemodeling.com.

The context of a web application

Usually, creating a web application is not an isolated project, but part of a program, in which a back end is developed as well and maybe even a complete web site. From this context, all kinds of input for building the web application is generated (partly during the development), for example:

- Business process models and business object models.
- Functional requirements, in most cases written as user stories or use cases.
- Non-functional requirements, varying from business-level requirements to a reference architecture (see next section) and coding guidelines.
- User interface design, most often a prototype or a series of images.
- Back end API specifications and eventually the API(s) themselves.
- Documentation of JavaScript libraries that will be used or may be used.

It is not my intention to give a complete action plan to use this input to reach a technical design. The list of inputs just sketches the context and, at the same time, makes clear that this information will not be included in the technical design.

Architecture

If AngularJS applications are built regularly in the organisation, there may be a reference architecture, which prescribes general rules for the application architecture, for instance about which types of components are discerned and their mutual relationships. In that case, the technical design can refer to the reference architecture and you can explain your application architecture much more concisely.

But anyway, it is essential to describe the architecture for every specific application. Like the rest of the design, the architecture is a combination of text and diagrams. For making diagrams, I almost always follow the UML standard, avoiding the need to explain the meaning of the graphical elements I use. At the top level, an AngularJS application is divided in modules, so naturally, you start the design by creating a diagram depicting the modules. You don’t have to design the whole architecture in one go. As the project progresses, you will add new modules.

The applicable UML symbol for an AngularJS module is the package. To make its meaning absolutely clear, you could give the package the stereotype ≪module≫, as in Figure 1.

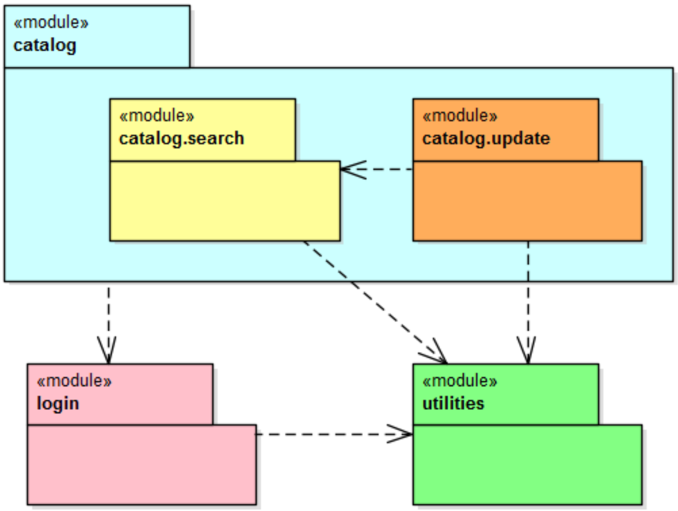


Figure 1. Overview of AngularJS-modules in a package diagram

The arrows, dependency associations, should match the dependencies as defined, or to be defined, in the source code:

```
angular.module('catalog.update', ['catalog.search', 'utilities']);
```

The same is true for nested packages:

```
angular.module('catalog', ['catalog.search', 'catalog.update', 'login']);
```

The nesting should match the folder structure of the development environment:

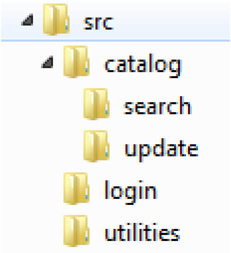


Figure 2. Folders in the development environment

You could extend the package diagram by displaying the JavaScript libraries and the back end APIs as packages, using different stereotypes of course, and then use dependency associations to indicate which modules make use of which libraries and APIs. If the diagram is getting too complex, you could divide the information over multiple diagrams, or decide to specify certain information not graphically, but in text only. For example, if ten modules depend on ‘utilities’ and the diagram becomes cluttered because of those ten arrows pointing at ‘utilities’, then you could leave out these arrows and explain the dependencies in the accompanying text.

I always give each module its own colour. In the rest of my technical design, all components get the same colour as the module to which they belong. This makes the boundaries between the modules visible at the first glance in every diagram.

A picture alone is not sufficient. Every module deserves a short description, which explains the purpose and the responsibilities of that module. It is useful to include references to user stories or parts of the user interface design.

The application architecture should also address aspects that pertain to the application as a whole, like:

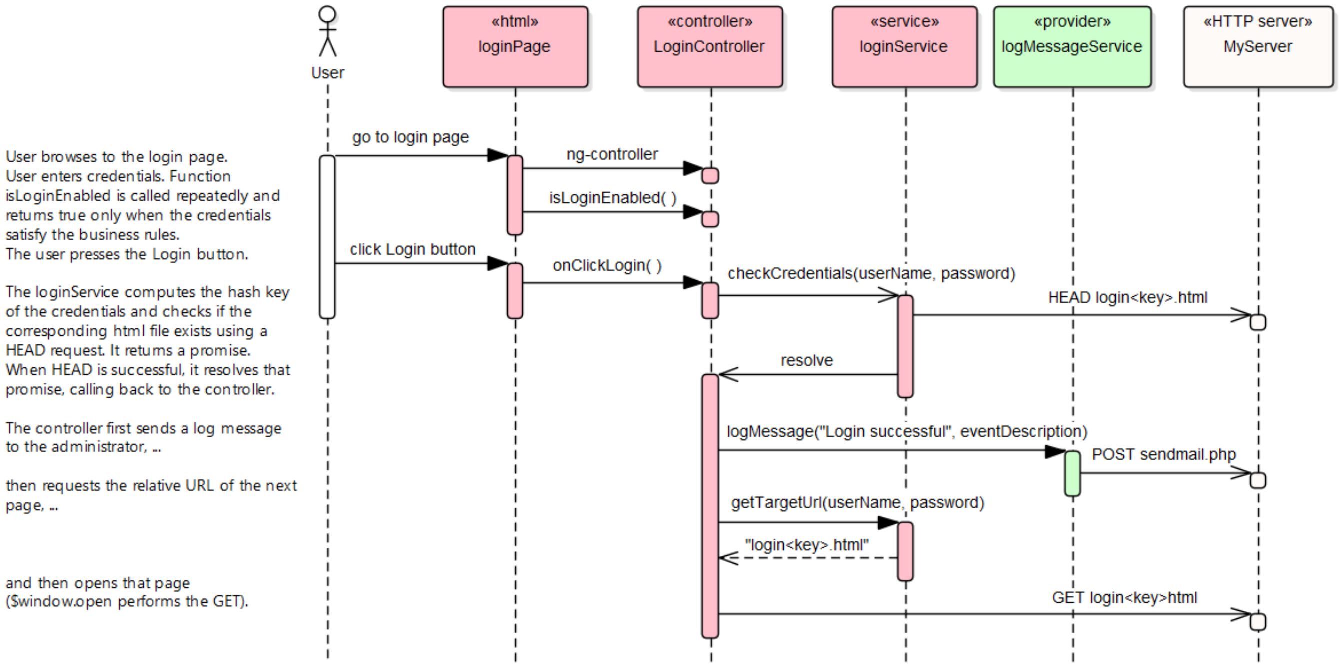
- A description of the production environment, including the use of a content delivery network (CDN), if applicable, and the caching strategy (how do users get the most recent version, while making optimal use of the browser cache and the CDN cache?).
- A description of the development environment, including unit test provisions.
- A concise overview of the build and deployment process. Which tools (Gulp, Protractor, Jenkins, ...) and which preprocessors are being used (SASS, uglify, ...)?
- The input validation and exception handling policies.
- Security. This is a back end issue in the first place, but the impact of security requirements on the web application should also be well designed.
- The collection of statistics about the usage and the behaviour of the application ('analytics').
- The way the web application is integrated with a content management system, if applicable.
- Coding guidelines (separately for HTML, CSS, JavaScript and AngularJS).

All or most of these issues exceed the context of a single application and can better be specified in a reference architecture, supporting the consistency among the various web applications within the organisation.

Scenarios

Having visualized the module structure, it is tempting to continue top-down by drawing the internal structure of every module. More about that in a minute, but first, let's follow a scenario-based approach. The sequence of the actions taking place in an AngularJS application is sometimes difficult to derive from the source code. This is because it is very event-based and full of callbacks. To provide insight into these kinds of scenarios, UML offers the so-called sequence diagrams. Figure 3 is an example of this type of diagram. The application components participating in the scenario are depicted as coloured rectangles, containing the names of the components. The component types are displayed as stereotypes. The colour of the component is equal to the colour of the module to which it belongs. This gives us insight into the division in modules at the first glance.

The example concerns a small login application. The user must enter his/her name and password, click the Login button and then, the application opens a particular HTML page showing the



User browses to the login page.
User enters credentials. Function isLoginEnabled is called repeatedly and returns true only when the credentials satisfy the business rules.
The user presses the Login button.

The loginService computes the hash key of the credentials and checks if the corresponding html file exists using a HEAD request. It returns a promise.
When HEAD is successful, it resolves that promise, calling back to the controller.

The controller first sends a log message to the administrator, ...

then requests the relative URL of the next page, ...

and then opens that page ($window.open performs the GET).

Figure 3. Sequence diagram

information the user is authorized for. The file name of that HTML page is login<key>.html, where <key> is a hash number, which is derived from the name and the password. After a successful login attempt, the application also sends an email to a particular email address, which serves as a kind of log book, holding a list of everyone who has logged in. The scenario displayed in Figure 3 is a successful login. Models like these provide a good insight into how the application works and at the same time, they can be related to user stories or use cases easily. This binds functional and technical design well together.

Lifelines

It is very important to make conscious decisions about what you show in a sequence diagram. If you include too many details, you are almost programming. This takes a lot of time and it is unfeasible to keep the diagram up to date. Let’s first determine which lifelines (as the vertical lanes in the sequence diagram are called officially) we include in the diagram and which not. Figure 3 shows, from left to right, first the user as a lifeline, then the application components participating in the scenario and the server at the end. What exactly are the application components I propose to show in sequence diagrams? In AngularJS terminology, the following component types are shown:

- HTML file (a template or a complete page)
- Controller
- Provider
- Factory
- Service
- Directive

This list can be extended or reduced according to your own judgement, as long as it is clear for everyone involved which choices have been made. All other JavaScript objects are too unimportant to be shown in the sequence diagram. For their interactions, one should look at the source code. If the application uses JavaScript components outside the AngularJS context, e.g. JQuery or Google Maps JavaScript API, then you should decide for each such a component whether it will be displayed as a lifeline in the sequence diagram *always* or *never*. In case of JQuery I would choose 'never' (too much detail) and in the case of Google Maps I would choose 'always'.

Figure 3 has one lifeline for the server. If multiple back end APIs participate in the scenario, you could draw one lifeline per API. The white colour indicates that the server is not part of the application. However, if the APIs have particular colours in your architecture overview diagram, then you could use those colours in your sequence diagram as well.

Messages

Great; now we know which lifelines we put in the diagram: the user, the application components and the external components. Now, we have to draw the messages between the lifelines. My advice is, to show all messages that occur between the lifelines in the scenario at hand. In other words, if there is a call from component X to component Y for the current scenario in the source code, but this call is not mentioned in the sequence diagram, then either the sequence diagram must be extended or the source code must be adapted. However, the internal flow within a component is not shown. To know the internal flow, one must look into the source code. These rules make the meaning of the sequence diagrams transparent for everyone who needs to understand or implement the diagrams.

What exactly is a message in terms of JavaScript? The most obvious message is a function call. In Figure 3, function calls are represented by messages with parentheses, for example isLoginEnabled( ) and onClickLogin( ). These can be mapped one-to-one to source code:

```
<button type="submit"
    ng-click="ctrl.onClickLogin()"
    ng-disabled="!ctrl.isLoginEnabled()"> Login </button>
```

Parameters, if any, are written between the brackets, but if that would clutter the diagram, you may indicate that the parameters have been left out using an ellipsis.

A different kind of message concerns user actions. These are displayed as messages from the user towards the HTML component. The text at the arrow briefly describes what the user does, for example "open Login page". Because this activates loginPage.html, I have drawn an arrow from the user to loginPage.html. This is a complete HTML page, containing <script> tags for AngularJS, for the login application and for the utilities module, which contains the sendMailService and other utilities. If it would have been an HTML template, I would have drawn the same arrow.

Often, one component communicates with the other via AngularJS, or via another object not displayed as a lifeline. In such a case, I draw an arrow with a free-format text, indicating what happens. In Figure 3 for example, I have written the text “ng-controller” above the second message. The implementation of this message in loginPage.html is:

```
<body ng-controller="LoginController">
```

For the ease of understanding, I describe the scenario in the left margin as well.

A closed arrowhead represents a synchronous message and an open arrowhead an asynchronous message (or a reply message – I’ll come to that in a minute). When the message represents a function call, then it is synchronous, except if the function returns a promise. In Figure 3, the message checkCredentials is asynchronous (also shown in Figure 4). The source code is as follows:

```
loginService.checkCredentials(userName, password).then(loginSuccessful, loginFailed);
```

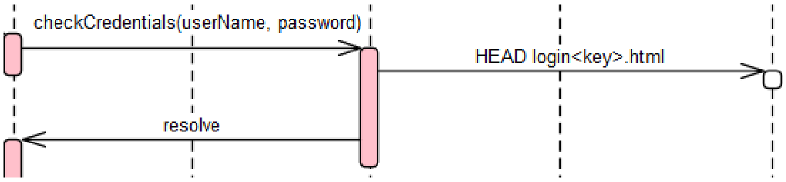


Figure 4. Asynchronous messages

The moment the loginService has verified that the credentials are valid, it will ‘resolve’ the promise, which will trigger a call to function loginSuccessful. The ‘resolve’ has been depicted in Figure 3 as an asynchronous message back to the controller. Because the controller is not called directly, but via the promise object, the arrow has a free-format text, in this case: ‘resolve’.

To check if the credentials are valid, the loginService performs an HTTP call of method ‘HEAD’. The source code is:

```
$http.head(targetUrl).then( ...
```

Because $http is not a lifeline, I have used a free-format text at this message as well. This function also returns a promise and therefore it is asynchronous.

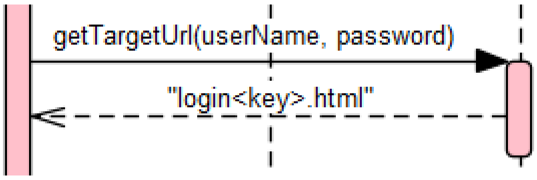


Figure 5. Synchronous message with reply message

If a function call returns data, you may draw a reply message, as in Figure 5. In this example, getTargetUrl returns a string. I don’t draw most reply messages in order to keep the diagram clean, but if data is returned that is crucial in the current scenario, I do draw a reply message.

Failure scenario

And then there is the scenario in which the login attempt fails. We could make a second sequence diagram for this, we could combine the two scenarios in one diagram using a combined fragment (with operator 'alt'), or we could decide not to model the failure scenario at all. In this case, I think the failure scenario is not very interesting to model. The 'resolve' message in Figure 3 becomes a 'reject' message and then the scenario stops. It is better to explain this is a one-line note instead of complicating the diagram with a combined fragment or to introduce a complete new diagram. If the failure scenario would introduce a new interaction between components, then I would model the scenario. Depending on how much overlap there would be with the basic scenario, I would either include the failure scenario in the existing diagram, or create a new diagram. In general, I am not in favour of squeezing multiple scenarios in one diagram using multiple (maybe nested) combined fragments, because it reduces the readability.

Design per module, per component

Apart from the scenarios, we usually need to do some design work based on top-down decomposition: a design per module and, included in it, a distinct design for some components as well. For the AngularJS applications I have worked on, I particularly felt the need to have:

1. An overview of the structure of complex HTML pages.
2. A design of the generic services.

Figure 6 is an example of the first category mentioned above. It is not a UML diagram, but a schematic representation of the layout of HTML templates within an HTML page. It is recommended to make such a picture and to discuss it in the team before the implementation begins. For someone who needs to make a change in a later phase of the project, the HTML structure picture gives immediate insight into the structure of the page.

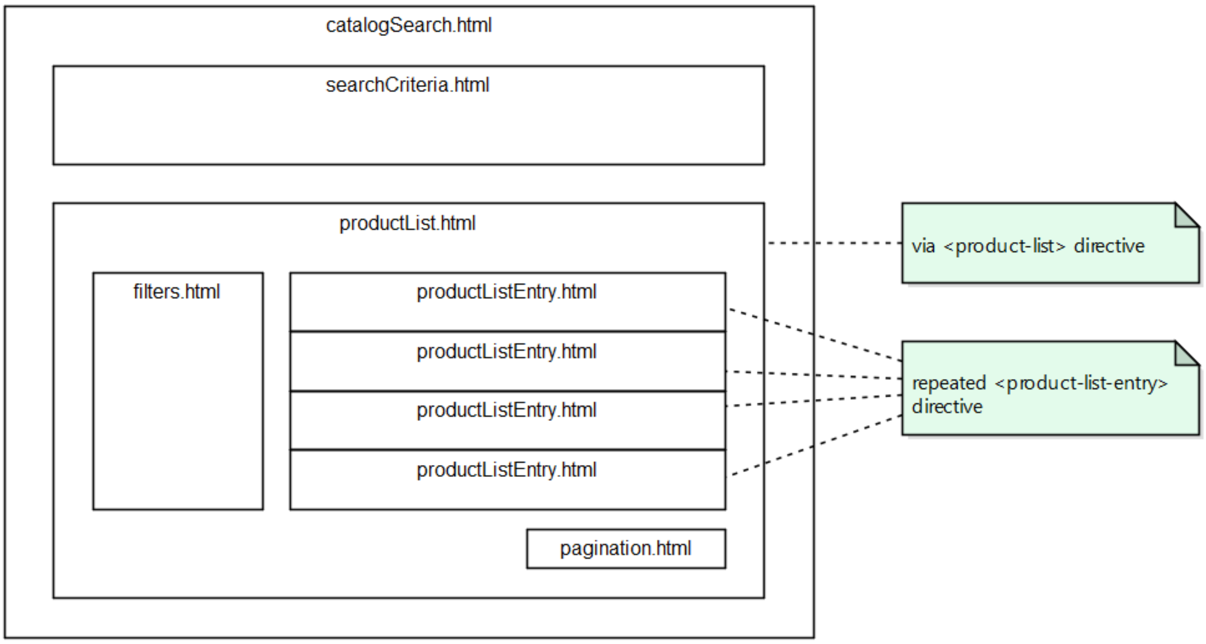


Figure 6. HTML structure

The other category is the design of generic services. Although the advantage of sequence diagrams is, that you can display the message flow through multiple layers of the application, this may also lead to a lot of duplication. In Figure 3, the logMessage function of the logMessageService is called once and it sends one HTTP message. Now, imagine that calls to logMessage occur frequently in all kinds of scenarios, should we display that HTTP message in every sequence diagram? When something changes in logMessage, we would have to change all these diagrams. Suppose logMessage does not use only one, but multiple other components, then it becomes clear that these repetitions are definitely undesirable. It is better to centralize the design of such generic services, as in Figure 7. Note, that the leftmost lifeline is now an anonymous caller.

There is often a need to use other forms of design as well. In every project, I usually make some class diagrams to provide insight into complex data structures and a state machine diagram every now and then. Do not underestimate the importance of plain text, explaining briefly and clearly which design decisions have been made.

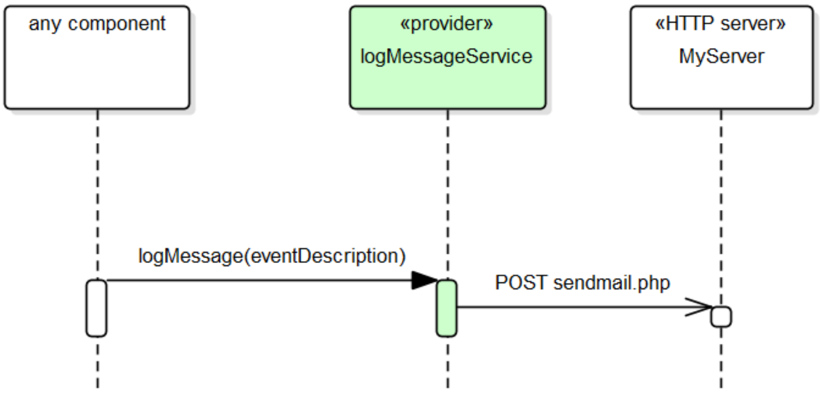


Figure 7. Design of a generic function

Tools

So, what would be a good way to store and maintain design documentation mainly based on UML? There is vast number of options, but let me mention three:

1. A wiki supporting UML.
2. A UML design tool with a document generator.
3. A word processor connected to a UML design tool.

This paper can't provide a thorough discussion of this subject, but I will give some arguments why these three options are used in practice.

A wiki supporting UML

An example of this option is Confluence, in combination with the Gliffy plugin. A great advantage of a wiki is, that it is easy to collaborate with multiple people on a single design. People who do not write the design, but only read it, can add comments. Changes are immediately visible for everyone and the history of changes can be monitored. In addition, the hypertext capabilities allow one to easily navigate through the design and to apply all kinds of cross references between the various pages of the design. In the case of Confluence, it is well integrated with other Atlassian tools, like Jira (issue management) and Bitbucket (source code management).

When you start a new project, be aware of the fact that you don't design only for the developers who will implement it, but also for the future people who need to know how the delivered software works. Now, suppose the software has been delivered and the project continues working on the next version. This means the wiki pages are going to reflect the new version and it will be difficult to see or specify which versions of the design pages match a particular version of the application. This is the downside of using a wiki.

A UML design tool with a document generator

An example is Enterprise Architect of SparxSystems. Such a tool provides a good support for UML design work. Creating and editing diagrams is easy and efficient. All UML elements are separate entities, which you can reuse in multiple diagrams. Changing the name or another property of such an element can be done at a central place and these changes automatically become visible in all diagrams containing that element. Plain text can also be entered in such a tool. Using the document generator, you can generate a Word document, but also an HTML website. The quality of the generated output is often limited, though.

Every proper UML design tool has some sort of version control, but it is cumbersome to view a previous version. I always make a complete copy of the design at every delivery moment, in order to be able to view the design of any version of the application easily afterwards.

A word processor connected to a UML design tool

An example is Microsoft Word in combination with eaDocX and Enterprise Architect. You can use all features of Word to write texts and you can insert the UML diagrams you have created in Enterprise Architect at the right spots in the document. The tool eaDocX makes sure, that changes in Enterprise Architect are also propagated to Word. This combination of tools provides both a good support for designing in UML and a decent word processor. This enables creating high quality design documents.

Another advantage is, that you can assign version numbers to documents. If these version numbers match (or can be mapped to) the version numbers of the software, you know exactly which version of the design belongs to which version of the software.

Epilogue

I hope this paper has provided some guidance to create a technical design for your application. If you want to view the complete source code of the login application, to compare this with the sequence diagram in Figure 3, then go to www.admiraalit.nl/admiraal/angular/loginapp.
Good luck!

Hans Admiraal, (freelance IT architect)
admiraal@aol.nl
www.admiraalit.nl