**Appendix C. C4 Model**

The C4 model (<https://c4model.com/>) is a system architecture diagram technique created by Simon Brown to decompose a system into various levels of abstraction. This section is a brief introduction to the C4 model. The website has good introductions and in-depth coverage of the C4 model, so we will only briefly go over the C4 model here; readers should refer to the website for more details. The C4 model defines four levels of abstraction.

A *context diagram* represents the system as a single box, surrounded by its users and other systems that it interacts with. Figure C.1 is an example context diagram of a new internet banking system that we wish to design on top of our existing mainframe banking system. Its users will be our personal banking customers, who will use our internet banking system via UI apps we develop for them. Our internet banking system will also use our existing email system. In figure C.1, we draw our users and systems as boxes and connect them with arrows to represent the requests between them.

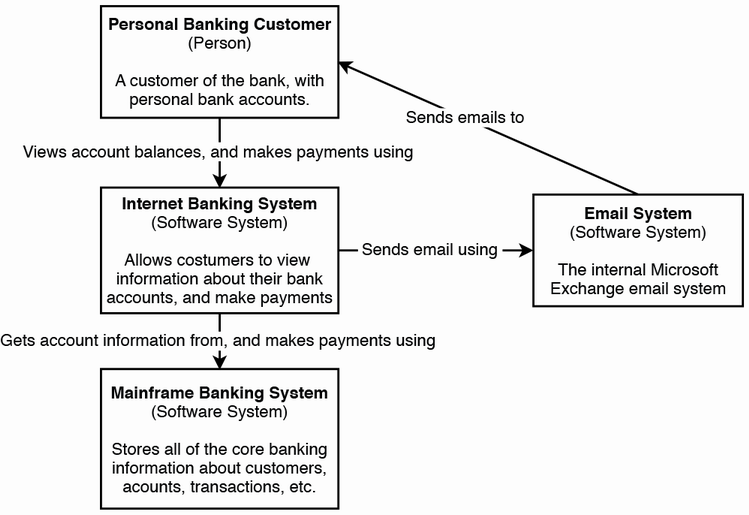


Figure C.1 A context diagram. Image from <https://c4model.com/>, licensed under <https://creativecommons.org/licenses/by/4.0/>. In this case, we want to design an internet banking system. Its users are our personal banking customers, who are people using our internet banking system via the latter’s UI apps. Our internet banking system makes requests to our legacy mainframe banking system. It also uses our existing email system to email our users. Many of the other shared services it may use are not available yet, and may be discussed as part of this design.

A *container diagram* is defined on c4model.com as “a separately runnable/deployable unit that executes code or stores data.” We can also understand containers as the services that make up our system. Figure C.2 is an example container diagram. We break up our internet banking system that we represented as a single box in figure C.1.

A web/browser user can download our single-page (browser) app from our web application service and then make further requests through this single-page app. A mobile user can download our mobile app from an app store and make all requests through this app.

Our browser and mobile apps make requests to our (backend) API application/service. Our backend service makes requests to its Oracle SQL database, mainframe banking system, and our email system.

A *component diagram* is a collection of classes behind an interface to implement a functionality. Components are not separately deployable units. Figure 6.3 is an example component diagram of our (backend) API application/service from figure 6.2, illustrating its interfaces and classes, and their requests with other services.

Our browser and mobile apps make requests to our backend, which are routed to the appropriate interfaces:

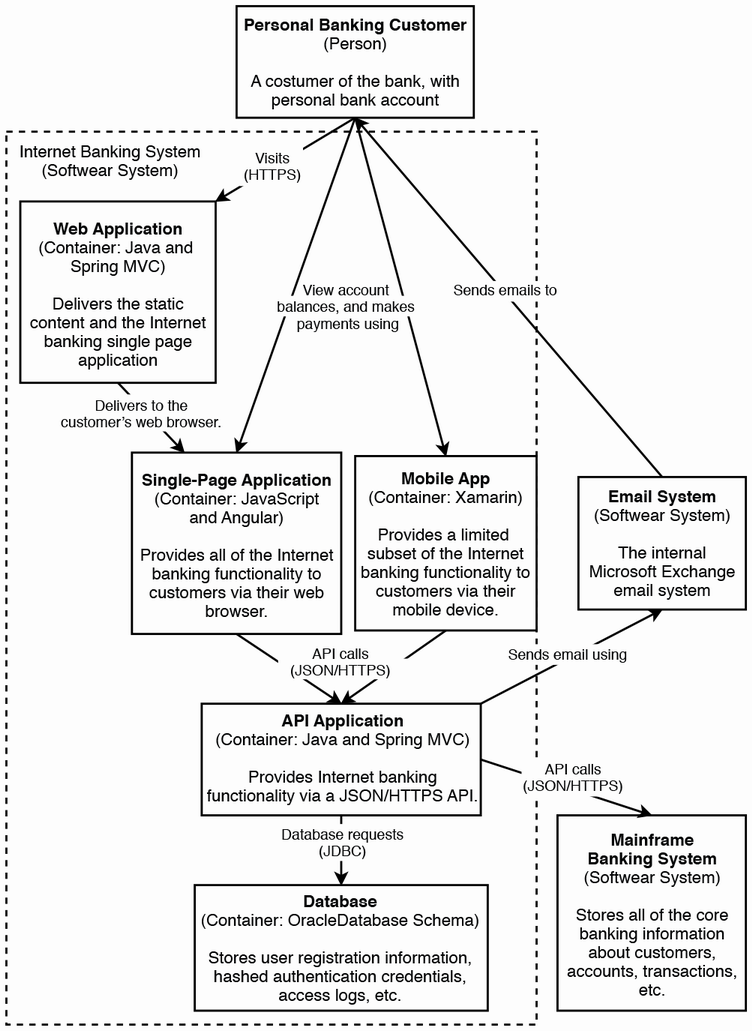


Figure C.2 A container diagram. Adapted from <https://c4model.com/>, licensed under <https://creativecommons.org/licenses/by/4.0/>.

Our sign-in controller receives sign in requests. Our reset password controller receives password reset requests. Our security component has functions to process these security-related functionalities from the sign-in controller and reset password controller. It persists data to an Oracle SQL database.

Our email component is a client that makes requests to our email system. Our reset password controller uses our email component to send password reset emails to our users.

Our account summary controller provides users with their bank account balance summaries. To obtain this information, it calls functions in our mainframe banking system façade, which in turn makes requests to our mainframe banking system. There may also be other components in our backend service, not illustrated in figure C.3, which use our mainframe banking system façade to make requests to our mainframe banking system.

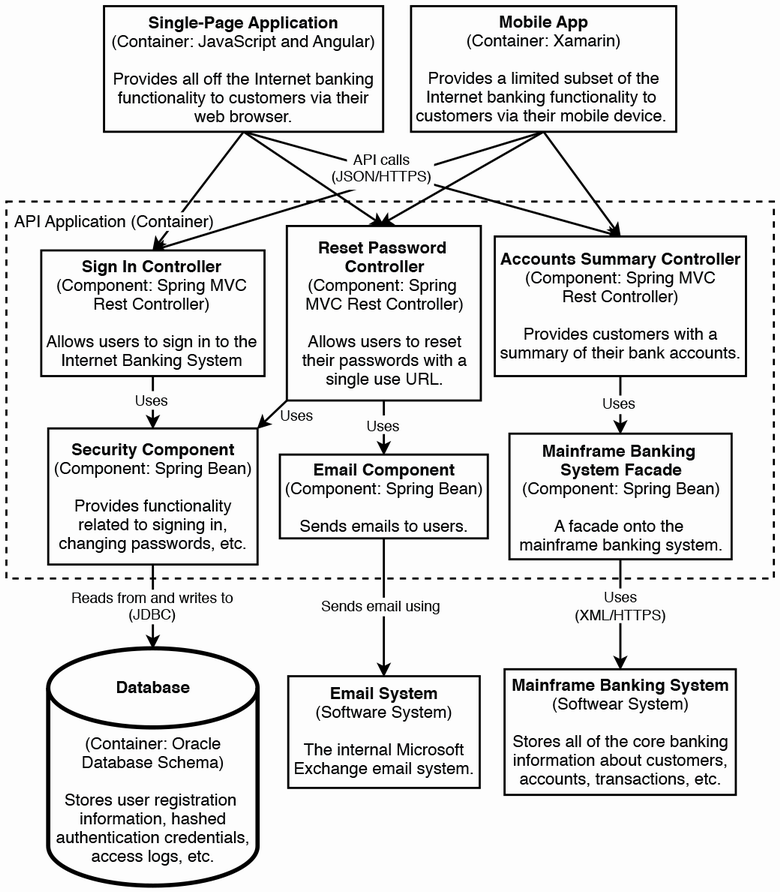


Figure C.3 A component diagram. Image adapted from <https://c4model.com/>, licensed under <https://creativecommons.org/licenses/by/4.0/>.

A *code diagram* is a UML class diagram. (Refer to other sources such as <https://www.uml.org/> if you are unfamiliar with UML.) You may use object-oriented programming (OOP) design patterns in designing an interface.

Figure C.4 is an example code diagram of our mainframe banking system façade from figure C.3. Employing the façade pattern, our MainframeBankingSystemFacade interface is implemented in our MainframeBankingSystemFacadeImpl class. We employ the factory pattern, where a MainframeBankingSystemFacadeImpl object creates a GetBalanceRequest object. We may use the template method pattern to define an AbstractRequest interface and GetBalanceRequest class, define an InternetBankingSystemException interface and a MainframeBankingSystemException class, and define an AbstractResponse interface and GetBalanceResponse class. A MainframeBankingSystemFacadeImpl object may use a BankingSystemConnection connection pool to connect and make requests to our mainframe banking system *and throw* a MainframeBankingSystemException object when it encounters an error. (We didn’t illustrate dependency injection in figure C.4.)

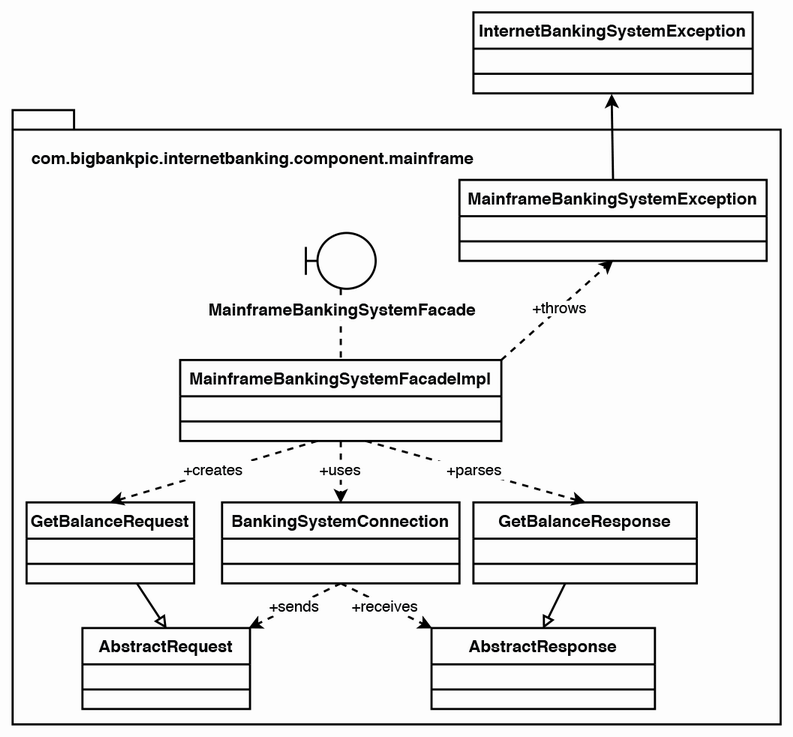


Figure C.4 A code (UML class) diagram. Image adapted from <https://c4model.com/>, licensed under <https://creativecommons.org/licenses/by/4.0/>.

Diagrams drawn during an interview or in a system’s documentation tend not to contain only components of a specific level, but rather usually mix components of levels 1–3.

The value of the C4 model is not about following this framework to the letter, but rather about recognizing its levels of abstraction and fluently zooming in and out of a system design.